



# CodeFlowBench: A Multi-turn, Iterative Benchmark for Complex Code Generation

Sizhe Wang<sup>†1,2</sup>, Zhengren Wang<sup>†1,3</sup>, Dongsheng Ma<sup>1</sup>

Yongan Yu<sup>1,4</sup>, Rui Ling<sup>1</sup>, Zhiyu Li<sup>\*3</sup>, Feiyu Xiong<sup>3</sup>, Wentao Zhang<sup>\*1</sup>

<sup>1</sup>Peking University <sup>2</sup>Shanghai University of Finance and Economics

<sup>3</sup>Institute for Advanced Algorithms Research, Shanghai <sup>4</sup>McGill University

{lizy,xiongfy}@iaar.ac.cn, wentao.zhang@pku.edu.cn

Dataset Code

## Abstract

Modern software development demands code that is maintainable, testable, and scalable by organizing the implementation into modular components with iterative reuse of existing codes. We formalize this iterative, multi-turn paradigm as *codeflow* and introduce **CodeFlowBench**, the first benchmark designed to comprehensively evaluate LLMs' ability to perform codeflow – implementing new functionality by reusing existing functions over multiple turns. CodeFlowBench comprises 5,258 problems from Codeforces and is continuously updated via an automated pipeline, which decomposes each problem into subproblems with unit tests based on dependency tree analysis and dataflow analysis. We further propose a novel evaluation framework featured dual assessment protocol and structural metrics derived from dependency trees. Extensive experiments on 16 popular LLMs reveal significant performance degradation in multi-turn scenarios. For instance, o1-mini retains only 20.8% Pass@1 in multi-turn scenario versus 37.8% in single-turn scenario. More fine-grained analysis illustrates that model performance inversely correlates with dependency complexity. These findings not only highlight the critical challenges for supporting real-world workflows, but also establish CodeFlowBench as an essential tool for advancing code generation research.

## 1 Introduction

Large Language Models (LLMs) have revolutionized code generation, with benchmarks like HumanEval [5] and MBPP [3] establishing foundational standards. As LLM capabilities advance, their role in real-world software development has expanded beyond solving toy problems to supporting complex workflows [21, 23, 28]. Modern benchmarks such as DevBench [25] and SWE-Bench [22] now emphasize practical scenarios like bug fixing. However, current benchmarks [9, 15, 35] still overlook the critical aspect of real-world development: the multi-turn and iterative *codeflow* scenario.

**The CodeFlow Task** In modern software engineering, multi-turn and iterative workflows are becoming increasingly prevalent, as the cornerstones of best practices like agile development [24, 1, 6]. By breaking down complex tasks into manageable subproblems, progressively refine solutions and reuse modular functions, developers can achieve faster delivery, reduced redundancy and enhanced maintainability in teamwork [12, 7]. For example, React's core package alone sees over 37 million

---

<sup>†</sup> Equal contribution; <sup>\*</sup> Corresponding author.

The first author completed this work during an internship at Peking University.

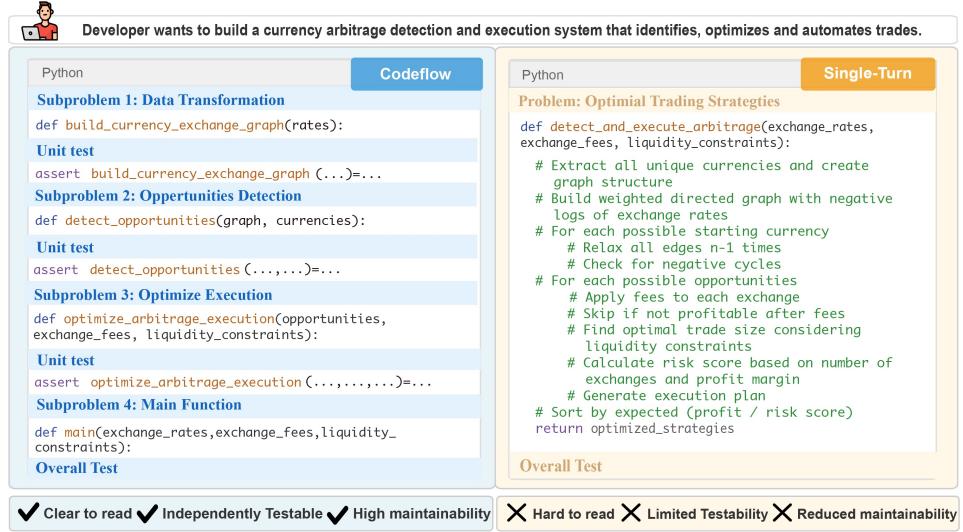


Figure 1: An example of building a currency arbitrage detection and execution system. The left side demonstrates a *codeflow* workflow, where modular functions are reused to construct the main logic. The right side represents a single-turn implementation, combining all steps into a monolithic block.

weekly downloads across 2 300+ dependent modules, illustrating the productivity gains of modular reuse [4]. As shown in Fig. 1 and 2, by building solutions by traversing the function dependency tree from the bottom up, the structured *codeflow* can both enable parallel development and improve readability, testability and maintainability through modular and responsibility boundaries. For LLMs to integrate effectively into such workflows, they must handle multi-turn interactions, manage dependencies between code segments and generate reusable components for subsequent iterations.

**Motivation** Despite the growing demands, current benchmarks have not fully captured the multi-turn and iterative aspects of codeflow. Firstly, most benchmarks, such as HumanEval and MBPP, focus only on single-turn code generation. While recent benchmarks like BigCodeBench [40] and SWE-Bench [22] have begun to incorporate practical development scenarios, they still remain at the level of single-turn code modification (or two-turn code generation), leaving the multi-turn generation capabilities unknown. Secondly, the few existing multi-turn benchmarks such as MTPB [31], only focus on single-function programming, lacking both unit tests and sufficient complexity to reflect real-world dependencies. Finally, due to the absence of an update mechanism, previous static datasets risk contamination and unreliable assessment. Therefore, there is a pressing need for a more challenging, well-annotated and frequently updated benchmark specially designed for codeflow.

To bridge this gap, we propose CodeFlowBench, a novel evaluation benchmark that sources its problems from the competitive programming platforms like Codeforces<sup>1</sup>, featuring **high difficulty** (competition-level problems), **high quality** (peer-reviewed editorial solutions), and **periodic updates** (frequent problem additions), elaborated in Appendix A. CodeFlowBench leverages platforms' frequent release of new problems to ensure continuous updates for the uncontaminated benchmark. Based on dependency decomposition and online submission check, an automated, lightweight pipeline is further developed to process original problems into sets of multi-turn, iterative subproblems with verified test cases and solutions. Remarkably, CodeFlowBench also introduces a specialized evaluation framework, including structural labels and metrics derived from dependency trees, to comprehensively assess multi-turn performance and provide more insights for model shortcomings.

**Contributions** In this paper, we present the first edition of CodeFlowBench, which comprises 5,258 complex problems fetched from the Codeforces problem archive accumulated until now. To ensure ongoing fairness and freshness, we will periodically perform the automated pipeline and update the benchmark with brand-new, uncontaminated problems. Our main contributions are threefold:

<sup>1</sup><https://codeforces.com/>

- **Pipeline Innovation:** We develop a data curation pipeline based on competition platform. This pipeline decomposes official solutions based on function dependencies to create multi-turn, iterative code problems that require code reuse. This lightweight pipeline is fully automated, rigorously verifiable, and easy to maintain, facilitating frequent updates.
- **Benchmark Construction:** We introduce the first benchmark specifically designed to evaluate the multi-turn, iterative code generation capabilities for codeflow settings. It features a large and diverse collection of 5,258 complex code problems. By leveraging official editorial solutions, automatically verifying generated code, and capturing test cases through input-output streams, we ensure the quantity and quality of our dataset.
- **Evaluation Design & Insights:** We propose a novel evaluation framework that contrasts multi-turn and single-turn code generation patterns. Our framework introduces new structural labels and metrics derived from the dependency trees, aiming to better capture the unique characteristics of multi-turn tasks. These innovations bridge the gap between traditional evaluation metrics and the specific challenges posed by iterative code generation. Extensive experiments reveal the huge performance drop in the codeflow scenarios, even for large reasoning models, highlighting the need for more advanced multi-turn capabilities.

## 2 Related Work

**Code Generation Benchmarks** The landscape of code generation benchmarks has evolved from simpler to more complex tasks to keep pace with the rapid development of LLMs [26, 32, 39], but still fail to comprehensively capture the multi-turn and iterative features of real-world scenarios. Early works like HumanEval [5] and MBPP [3] focus on standalone functions with low complexity and limited dependency environments. Recent benchmarks have emerged to evaluate more complex and realistic scenarios, yet have obvious limitations: most benchmarks such as APPS [13], LiveCodeBench [20] and SWE-Bench [22] are limited to single-turn code generation or modification. For the few existing multi-turn benchmarks, MTPB [31] focus on overly simplistic single-function programming without paired unit tests, while InterCode [38] discusses interactive coding with execution feedback. In stark contrast, *codeflow* structures the development into multi-turn processes, ensuring each component is maintainable, testable and reusable. These limitations highlight the crucial gaps for codeflow benchmarking, and the CodeFlowBench pioneers this research line.

**Code Generation LLMs** Recent years have witnessed unprecedented progress in code generation capabilities of LLMs. Early works such as Codex [5] and AlphaCode [26] demonstrated proficiency in tasks ranging from code completion to competition-level problem solving. With the scaling up of pre-trained models, exemplified by GPT-4 [2], Code-Llama [33], Deepseek-Coder [10] and Qwen2.5-Coder [16], these advanced models have impressive performance across various programming tasks, languages and domains. Building on these foundations, the code generation capabilities have further advanced through instruction tuning and agent frameworks. Models such as WizardCoder [29] and Magicoder [36] leverage instruction tuning to improve intent alignment and interactive dialogue capabilities, while agent frameworks like AgentCoder [14] and MapCoder [18] enable autonomous planning, iterative refinement, and self-evaluation. Despite these advancements, the community still remains unknown about “*how well and how deeply LLMs can perform codeflow*”—a critical paradigm for real-world software engineering. Our CodeFlowBench thus provides a principled framework for advancing both model development and evaluation [21].

## 3 CodeFlowBench

CodeFlowBench is a lively-updated benchmark which currently contains 5,258 diverse, high-quality and challenging problems. An example problem of CodeFlowBench is shown in Figure 2. In this section, we introduce the data curation pipeline and evaluation framework of CodeFlowBench.

### 3.1 Data Curation Pipeline

We designed an automated, lightweight data-curation pipeline to generate complex multi-turn coding problems. The pipeline consists of two main phases: (1) *Data Preparation (Stages I–III)*, which involves source-specific routines tailored to collect and normalize raw problem data from various

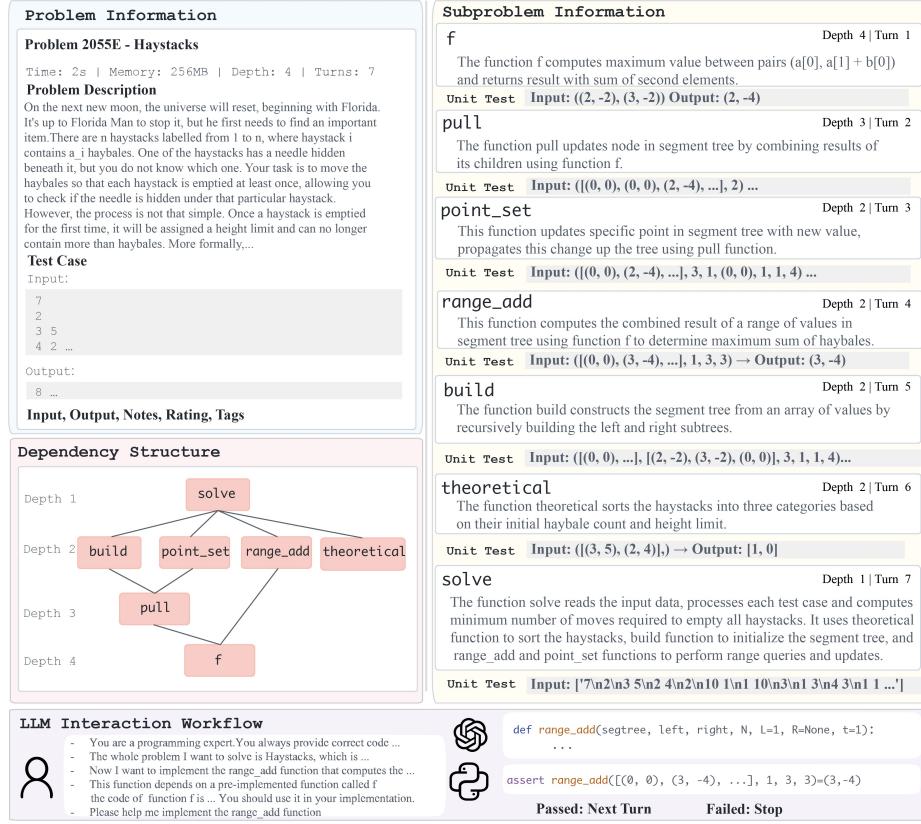


Figure 2: An example problem from CodeFlowBench, sourced from Codeforces problem 2055E. The top-left block displays the problem information. The bottom-left block illustrates the dependency structure, with nodes as subproblems and edges as dependency relationships. The right block details each subproblem, where models solve these subproblems iteratively. The bottom block shows the LLM interaction workflow for each turn.

platforms. (2) *Subproblem Generation (Stages IV–VI)*, which employs an universally applicable problem decompostion framework. This framework systematically analyzes dependency relationships between functions or classes, and back-translates functions into coherent and sequential subproblems. Notably, our implementation is the first to integrate both official problem editorials and automated submission checker into the data curation process. Leveraging official editorials and submission checker provides reliability benefits, such as eliminated copyright issues or data cleaning needs, and it also ensures the *codeflow* implementation is high-quality and optimal for each problem. As shown in Fig. 3, our pipeline specifically comprises the following stages:

**Stage I: Scrape Problem Information** We initiate the process by visiting the Codeforces problem list and navigating to each problem’s page to extract the following information: *problem ID, title, time and memory limits, problem description, input/output specifications, sample tests, notes*. Furthermore, by querying the official Codeforces API, we retrieve each problem’s rating and tags. Please refer to Appendix B.1 for information details and examples.

**Stage II: Scrape Editorial Information** We gather editorial content from corresponding tutorial pages, where each problem’s ID, textual explanation, and solution code are listed sequentially. By identifying problem IDs through URL and text patterns, we extract the relevant content and associate it with the problem list generated in Stage I. Low-quality entries are filtered out, and solutions are categorized into text and code components. Please refer to Appendix B.2 for more details.

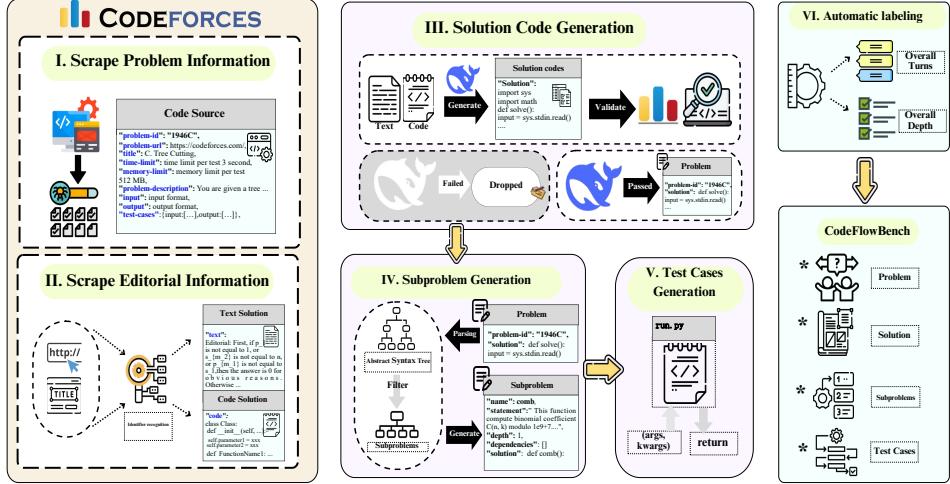


Figure 3: The data curation pipeline. Left: in the data-collection stage, we scrape problems and corresponding text and code solutions. Middle: in the data-processing and subproblem-generation stage, we validate solutions, parse codes into abstract syntax trees, decompose original problems into individual subproblems, and generate test cases. Right: the final labeling stage for complexity analysis, and the curated CodeFlowBench problem set.

**Stage III: Solution Code Generation** We input both the problem and its explanation information into the Deepseek-R1 model [11], which either directly uses the provided code implementation or organizes the existing textual solution and converts it into parsable solution code that meets the requirements for extracting function dependencies and constructing subproblems. The generated code is then submitted to the Codeforces platform for correctness verification, and we retain only those codes that successfully solve the problem. The prompt template can be found in Appendix B.3.

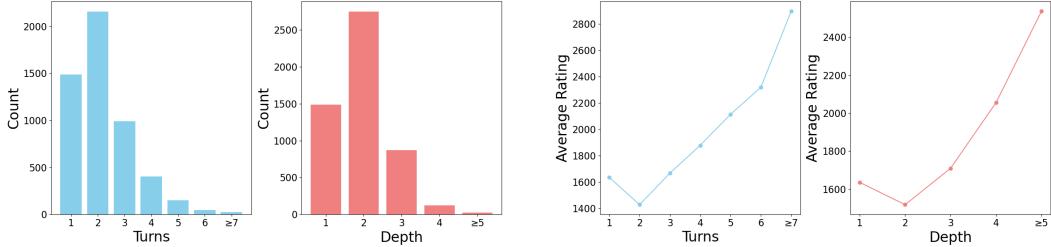
**Stage IV: Subproblem Generation** We parse the abstract syntax tree (AST) of codes from Stage III to extract function dependencies, filter out built-in functions, and conduct topological sort so that lower-level ones come first. We then treat each function as a subproblem and record its solution code, AST depth, and dependencies information. These information, along with the problem and explanation, are fed into Deepseek-V3 [27] to back-translate a subproblem description. Finally, a subproblem contains the following fields: *name*, *statement*, *depth*, *dependencies*, *solution*. Detailed explanations of the prompt template and an example can be found in Appendix B.4.

**Stage V: Test Cases Generation** For every problem, publicly accessible test cases from the Codeforces platform are scraped and used to execute the respective solution codes. During execution, we capture input-output pairs at every function call location, which serve as test cases for the corresponding subproblems. Further elaboration on this process can be found in Appendix B.5.

**Stage VI: Automatic Labeling** To quantify the complexity of the curated problems for fine-grained analysis, we define two metrics: **Overall-Turns**—the number of turns required to solve the entire problem; **Overall-Depth**—the depth of the AST, where the main/solve function as the root and subproblems as nodes. As shown in Fig. 4, we report the distributions of overall-turns and overall-depths, and correlate them with the difficulty ratings provided by Codeforces. The strong correlation observed confirms that our label effectively captures problems’ intrinsic complexity.

### 3.2 Evaluation Framework

**Task Definition** The evaluation design of CodeFlowBench follows best practices defined in HumanEval [5] but introduces a novel multi-turn iterative paradigm. That is, models are required to implement a given function in each round. However, CodeFlowBench introduces distinct differences in the supporting materials provided to the models. Beyond the function signature and problem description, we include dependency information and pre-implemented functions for code reuse.



(a) Distributions of overall-turns and overall-depth.

(b) The Correlations with Rating Levels.

Figure 4: Statistics of the overall-turns and overall-depth metrics in CodeFlowBench. Subfigure (b) shows inflection points at turns = 1 and depth = 1. This is attributed to the fact that competition-level problems are not restricted to multi-turn or deeply nested structures. Partial difficult problems are designed to be solvable by single function such as number theory related problems.

Additionally, we provide an overarching background description of the problem. This reflects the real-world scenario where developers often possess overall understanding of the entire problem when implementing code incrementally.

As illustrated in Fig. 2, models must implement target functions while leveraging dependency relationships and pre-implemented components from prior turns. Formally, given  $T$  total turns, the function implementation  $C_t$  at turn  $t$  is generated based on:

- Function signature  $F_t$  and subproblem description  $S_t$  for the current target function,
- Background context  $B$  describing the overarching programming problem,
- Dependency specifications  $\mathcal{D} = \{k_1, \dots, k_d\}$  indicating prerequisite functions  $F_{k_1}, \dots, F_{k_d}$ ,
- Verified functional implementations  $\{C_1, \dots, C_{t-1}\}$  of previous dependencies.

For baseline comparison, we define the single-turn variant where the model generates all components simultaneously. The mathematical expressions of both settings are presented below:

$$C_t \leftarrow p(\cdot \mid F_t, S_t, \{F_{k_1}, F_{k_2}, \dots, F_{k_d}\}, \{C_1, C_2, \dots, C_{t-1}\}, B) \quad (1 \leq t \leq T),$$

$$\{C_1, C_2, \dots, C_T\} \leftarrow p(\cdot \mid \{F_1, F_2, \dots, F_T\}, \{S_1, S_2, \dots, S_T\}, B).$$

**Performance Metrics** We adopt widely-used Pass@k [5] as the main metric for both multi-turn and single-turn cases, but augment it with novel diagnostics for multi-turn analysis. While Pass@k evaluates final success rates, *the coarse-grained nature fails to capture partial progress in failed attempts* - two models may fail solve the entire problem at different stages but receive identical scores.

To address this limitation, we propose a new metric, Pass Turn ( $PT$ ), which identifies the exact turn at which a model fails by leveraging unit tests for each subproblem. However, only considering failing turns may be biased due to the arbitrary topological ordering at the same depth. We further define, Pass Depth ( $PD = D - d$ ), for bottom-up programming, where  $d$  and  $D$  are working depth and total depth. For statistical significance, we define the Average Pass Turn (APT) and Average Pass Depth (APD) by averaging across problems grouped by turn or depth. For multiple trials, we define APD@k and APT@k following the Pass@k expression. Refer to Appendix C for more details.

## 4 Experiments

**Experiment Setup** For comprehensive experiments, we evaluate both close-sourced and open-sourced models. The closed-source models include the GPT family (o3-mini, o1-mini, GPT-4.1-mini, GPT-4o-mini and GPT-4o) [19, 17], Claude-3.5-Sonnet and Gemini-2.0-flash. The open-source models include the Qwen family (Qwen-2.5-Coder-7B-Instruct, Qwen-2.5-Coder-32B-Instruct, Qwen-2.5-72B-Instruct and QwQ-32B) [16, 37, 34], the DeepSeek family (DeepSeek-v3 and DeepSeek-R1) [27, 11], the Llama family (Llama-3.1-8B-Instruct and Llama-3.3-70B-Instruct) [8] and Yi-Coder-9B-Chat [30]. To eliminate potential data leakage and unnecessary evaluation

Table 1: Performance comparison on CodeFlowBench with Pass@1 and APD@1 (Average Pass Depth) metrics. As a reference for overall APD, CodeFlowBench’s average problem depth is 1.94.

Model	Pass@1			APD@1 (Average Pass Depth)			
	Multi-turn	Single-turn	Overall	Depth1	Depth2	Depth3	Depth4
<b>Closed-Source</b>							
o3-mini	22.7	<b>38.9</b>	0.570	<b>0.322</b>	0.585	0.818	<b>1.250</b>
o1-mini	20.8	37.8	0.541	0.233	0.581	0.818	1.125
GPT-4.1-mini	<b>24.4</b>	38.7	<b>0.602</b>	0.265	<b>0.673</b>	<b>0.873</b>	1.042
GPT-4o	17.5	30.2	0.494	0.177	0.539	0.758	1.042
GPT-4o-mini	13.8	22.0	0.423	0.138	0.438	0.697	1.167
Gemini-2.0-flash	17.3	31.6	0.496	0.183	0.527	0.782	1.042
Claude-3.5-Sonnet	13.6	36.0	0.414	0.117	0.444	0.648	1.167
<b>Open-Source (7B-Level)</b>							
Llama-3.1-8B-Instruct	0.9	3.5	0.208	0.011	0.224	0.412	<b>0.792</b>
Qwen2.5-Coder-7B-Instruct	2.3	<b>15.0</b>	<b>0.233</b>	0.018	0.247	<b>0.436</b>	0.750
Yi-Coder-9B-Chat	<b>5.0</b>	13.8	0.228	<b>0.021</b>	<b>0.270</b>	0.388	0.500
<b>Open-Source (32B-Level)</b>							
Qwen2.5-Coder-32B-Instruct	8.6	19.8	0.316	0.067	0.342	<b>0.570</b>	0.917
QwQ-32B-Preview	<b>17.3</b>	<b>32.7</b>	<b>0.487</b>	<b>0.261</b>	<b>0.539</b>	<b>0.570</b>	<b>1.042</b>
<b>Open-Source (70B-Level)</b>							
Llama-3.3-70B-Instruct	<b>15.0</b>	<b>27.6</b>	<b>0.448</b>	<b>0.163</b>	<b>0.465</b>	<b>0.733</b>	<b>1.042</b>
Qwen2.5-72B-Instruct	9.1	21.3	0.301	0.110	0.314	0.497	0.583
<b>Open-Source (Large Model)</b>							
Deepseek-V3	18.0	35.7	0.529	0.219	0.549	0.836	<b>1.208</b>
Deepseek-R1	<b>20.5</b>	<b>46.1</b>	<b>0.569</b>	<b>0.303</b>	<b>0.606</b>	<b>0.842</b>	0.916

overhead, we select the most recent 1000 problems from CodeFlowBench for tests. Each model is evaluated in both multi-turn and single-turn scenarios, with Pass@k and APD@k as the primary metrics. Please refer to Appendix D.1 and D.2 for more implementation details.

#### 4.1 Main Experiments

**Overall Performance** Table 1 presents the extensive experiments across 1,000 latest problems. The highest observed Pass@1 score of 24.4% and maximum APD@1 of 0.602 demonstrate the benchmark’s rigorousness, with even state-of-the-art LLMs struggling to achieve satisfactory performance. While closed-source models generally outperform the open-source ones, exceptions exist—notably DeepSeek-R1 surpasses several proprietary models. Notably, two model categories exhibit parameter-efficient competence: specialized coding models and reasoning models. Small coding models like Qwen2.5-Coder and Yi-Coder deliver surprising Pass@1 relative to model size, while reasoning models like QwQ-32B-Preview approach GPT-4o’s performance via test-time scaling. These findings suggest that domain-specific and reasoning models are promising directions for future advancements.

**Multi-Turn versus Single-Turn** Table 1 also reveals the substantial performance gap between the multi-turn and single-turn scenarios. This gap underscores the inherent complexity of multi-turn code generation, which requires models to “look before and after” for long-context coherence across iterative function implementations. Furthermore, we observe distinct features of models’ performance across single-turn and multi-turn scenarios: while models like Claude-3.5-Sonnet and DeepSeek-R1 excel in single-turn scenarios, they exhibit up to +60% performance degradation in multi-turn settings. In contrast, models such as GPT-4o and Llama-3.3-70B-Instruct demonstrate more consistent performance. The fundamental difference lies in the cognitive demands: single-turn evaluation examines complex reasoning capabilities, while multi-turn evaluation needs to generate reusable, modular functions, and integrate them into larger components. These insights expose under-explored limitations such as iterative development and dependency management.

**Depth-Wise Performance** The performance stratification across dependency depths reveals distinct model characteristics. Small open-source models (7B-32B parameters) show pronounced depth-specific biases—Qwen2.5-Coder-7B achieves 0.750 APD@1 on Depth4 problems versus 0.018 on Depth1, suggesting emergent compositional abilities despite limited parameter counts. In contrast, larger models exhibit more balanced performance profiles, with GPT-4.1-mini maintaining 0.265-1.042 APD@1 across depths. Surprisingly, top performers like DeepSeek-R1 show no significant

Table 2: Performance comparison on CodeFlowBench with the APT@1 (Average Pass Turn) metric. As a reference for overall APT, CodeFlowBench’s average problem turn is 2.20.

Model	Overall	Turn1	Turn2	Turn3	Turn4	Turn5
<b>Closed-Source</b>						
o3-mini	0.600	<b>0.322</b>	0.632	0.777	0.983	0.571
o1-mini	0.581	0.233	0.645	0.798	0.879	0.667
GPT-4.1-mini	<b>0.646</b>	0.265	<b>0.760</b>	<b>0.803</b>	<b>1.034</b>	<b>0.762</b>
GPT-4o	0.537	0.177	0.604	0.697	1.017	0.476
GPT-4o-mini	0.467	0.137	0.501	0.638	0.982	0.476
Gemini-2.0-flash	0.541	0.183	0.595	0.702	0.948	0.667
Claude-3.5-Sonnet	0.461	0.117	0.508	0.574	1.000	0.571
<b>Open-Source (7B-Level)</b>						
Llama-3.1-8B-Instruct	0.232	0.011	0.245	<b>0.404</b>	0.534	<b>0.476</b>
Qwen2.5-Coder-7B-Instruct	<b>0.258</b>	0.018	0.270	0.394	<b>0.638</b>	0.286
Yi-Coder-9B-Chat	0.247	<b>0.021</b>	<b>0.314</b>	0.324	0.362	0.286
<b>Open-Source (32B-Level)</b>						
Qwen2.5-Coder-32B-Instruct	0.352	0.067	0.391	0.532	0.569	<b>0.666</b>
QwQ-32B-Preview	<b>0.515</b>	<b>0.261</b>	<b>0.606</b>	<b>0.553</b>	<b>0.828</b>	0.381
<b>Open-Source (70B-Level)</b>						
Llama-3.3-70B-Instruct	<b>0.493</b>	<b>0.163</b>	<b>0.515</b>	<b>0.681</b>	<b>1.000</b>	<b>0.476</b>
Qwen2.5-72B-Instruct	0.330	0.110	0.350	0.452	0.517	0.429
<b>Open-Source (Large Model)</b>						
Deepseek-V3	0.572	0.219	0.622	0.750	<b>0.966</b>	0.810
Deepseek-R1	<b>0.609</b>	<b>0.304</b>	<b>0.677</b>	<b>0.766</b>	<b>0.966</b>	<b>0.860</b>

advantage on Depth4 problems (APD@1: 0.916 vs. 1.208 for DeepSeek-V3), indicating current reasoning patterns inadequately handle deep dependency chains. This suggests models could benefit from explicit dependency tracking mechanisms and iterative verification loops during code generation. For another perspective, we report APT@1 in Table 2. Notably, the trends of APT closely align with APD, reinforcing our observations of models’ depth-wise performance.

## 4.2 Analysis and Discussion

**Dependency Structure Challenges in Multi-turn Scenarios** A deeper look of solved problems reveals a striking imbalance: the majority of correctly addressed cases correspond to problems with simple, linear dependency structures (e.g., shallow call graphs or sequential function compositions). However, as problem architectures evolve toward modular and hierarchical dependencies (e.g., nested function calls, interdependent components), state-of-the-art models exhibit significant performance degradation. This phenomenon is empirically validated in Figure 5, which illustrates the Pass@1 scores across varying turn counts. The consistent performance trajectory—initially high for 1-2 turn problems followed by approximately exponential decline as turn counts increase—demonstrates the inherent challenges in multi-turn code generation. Even top-performing models (e.g., Deepseek-R1 with 20.5% Pass@1) fail to solve problems requiring more than six turns. This underscores the critical limitation to balance local correctness and global integration across iterative development cycles. To quantify dependency structure complexity, we introduce the *Dependency Structure Complexity (DSC)* metric, defined as the ratio of total turns to the maximum depth in the AST. Figure 13 presents the models’ performance across different DSC intervals, revealing that most models perform well on problems with linear dependency structures but struggle significantly as the dependency structure becomes more complex. We leave the discussion on DSC in Appendix E.1.

**Fine-Grained Error Types in Multi-turn Generation** Given the significant performance gap between models in multi-turn and single-turn scenarios, we conducted studies to identify the underlying reasons. We categorized errors into three primary types: (1) **Incomplete Reasoning (IR)**: Models often handle only straightforward “happy-path” cases and fail to generalize. They may oversimplify key requirements, omit boundary or atypical cases, or choose naive algorithms whose logic or performance collapses on larger inputs. This reflects a limitation in the models’ reasoning abilities. (2) **Insufficient Globalization (IG)**: While a function’s logic may run correctly in isolation, it may omit necessary imports, global constants, or shared-state interactions, preventing proper integration into the broader application or runtime. This indicates a limitation in the models’ ability to manage global context. (3) **Instruction Misinterpretation (IM)**: Given multi-turn prompts, models could solve

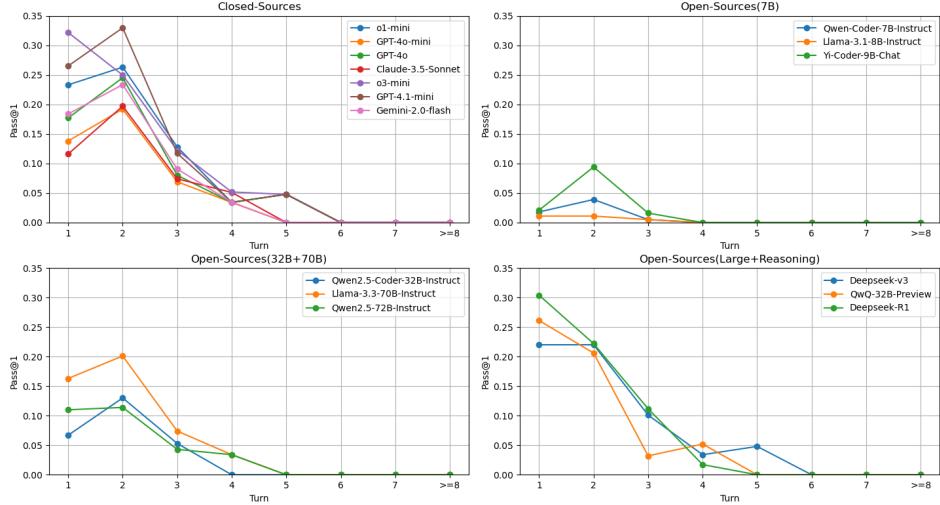


Figure 5: The Pass@1 metrics on multi-turn problems grouped by model categories and turn number.

isolated subproblems but lack a coherent understanding of the overarching goal. Typical failures include misusing helper functions or implementing disorganized code within the top-level function, i.e. incorrect integration of components. To quantify the distribution of errors, we randomly sampled examples and manually annotated error categories. The proportion is calculated in Table 3.

Table 3: Proportion of each error type on Deepseek-V3 and o1-mini. The IR Remediation Rate is the proportion of IR errors in multi-turn scenario which are correctly answered in single-turn scenario.

Model	IR(%)	IG(%)	IM(%)	Others(%)	IR Remediation Rate(%)
Deepseek-v3	47.2	14.6	32.6	5.6	16.7
o1-mini	38.7	11.8	43.0	6.5	8.3

Table 3 offers several insights. First, Incomplete Reasoning remains the dominant error type which is the main challenge of competition-level problems. To distinguish the effect of multi-turn scenario, we further introduce **IR Remediation Rate** to quantify how much of IR can be fixed when switching from multi-turn to single-turn scenarios. The statistics show that, most IR errors (30% of all error types) originate from the multi-turn scenario. Furthermore, Insufficient Globalization and Instruction Misinterpretation are inherently specific to multi-turn scenarios. The high rate of IM errors highlights that models struggle with consistent instruction following across multiple turns, or the codeflow task. Finally, IG errors reveal that models often lack "global awareness", highlighting the importance of both local correctness and global coherence evaluation. We leave case studies in Appendix E.2.

## 5 Conclusion

This paper introduces CodeFlowBench, the first benchmark specifically designed to evaluate multi-turn iterative code generation capabilities in realistic development workflows, i.e., the *codeflow*. Comprising 5,258 competition-level programming problems curated from Codeforces, our benchmark makes three key contributions: (1) an automated pipeline for decomposing complex problems into dependency-aware subproblems with paired unit tests, (2) a novel evaluation framework with proposed structural metrics, such as APT@k, APD@k, and DSC, to quantify multi-turn performance, and (3) the discovery of substantial performance gaps between multi-turn and single-turn scenarios (up to 60% performance degradation). Our fine-grained analysis identifies dominant failure modes and provides insights for further advancements. Extensive experiments across 16 popular LLMs highlight the substantial challenges posed by both the codeflow task and our benchmark. We believe CodeFlowBench not only illuminates critical limitations in existing LLMs but also paves the way for more realistic and powerful code generation systems. **Future Work:** CodeFlowBench currently remains confined to competition-level problems, which should be expanded with repository-level challenges. We also plan to design and train dedicated code generators tailored to the codeflow task.

## References

- [1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.
- [2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- [3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [4] Adam Boduch. *React material-ui cookbook: build captivating user experiences using react and material-ui*. Packt Publishing Ltd, 2019.
- [5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [6] W Alec Cram. Agile development in practice: Lessons from the trenches. *Information Systems Management*, 36(1):2–14, 2019.
- [7] Daniel Feitosa, Apostolos Ampatzoglou, Antonios Gkortzis, Stamatia Bibi, and Alexander Chatzigeorgiou. Code reuse in practice: Benefiting or harming technical debt. *Journal of Systems and Software*, 167:110618, 2020.
- [8] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [9] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.
- [10] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- [11] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [12] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management science*, 54(1):180–193, 2008.
- [13] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.
- [14] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.
- [15] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie Zhang. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544, 2024.
- [16] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- [17] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

- [18] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.
- [19] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.
- [20] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- [21] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.
- [22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- [23] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.
- [24] Craig Larman. *Agile and iterative development: a manager's guide*. Addison-Wesley Professional, 2004.
- [25] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. Devbench: A comprehensive benchmark for software development. *CoRR*, 2024.
- [26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- [27] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
- [28] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.
- [29] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.
- [30] AI Meet yi coder. A small but mighty llm for code, september 2024. URL <https://01-ai.github.io/blog.html>.
- [31] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.
- [32] Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, et al. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*, 2025.
- [33] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- [34] Qwen Team. Qwq: Reflect deeply on the boundaries of the unknown. *Hugging Face*, 2024.

- [35] Zhengren Wang, Rui Ling, Chufan Wang, Yongan Yu, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. Maintaincoder: Maintainable code generation under dynamic requirements. *arXiv preprint arXiv:2503.24260*, 2025.
- [36] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*, 2023.
- [37] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [38] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854, 2023.
- [39] Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint arXiv:2412.21199*, 2024.
- [40] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

## Appendix

<b>A Features of CodeFlowBench</b>	<b>13</b>
<b>B Data Curation Pipeline</b>	<b>13</b>
B.1 Detail of Problem Scraping . . . . .	13
B.2 Detail of Solution Scraping . . . . .	15
B.3 Detail of Solution Code Generation . . . . .	17
B.4 Detail of Subproblems Generation . . . . .	17
B.5 Detail of Test Cases Generation . . . . .	17
<b>C Mathematical Expression of <math>PD@k</math> &amp; <math>APD@k</math></b>	<b>20</b>
<b>D Experiment</b>	<b>21</b>
D.1 The Setting of Experiment . . . . .	21
D.2 Instruction Templates . . . . .	22
D.3 Pass Turn Result . . . . .	23
<b>E Discussion</b>	<b>24</b>
E.1 Discussion on Models' DSC Performance . . . . .	24
E.2 Error Case Study . . . . .	24
<b>F Impact Statement</b>	<b>28</b>

### A Features of CodeFlowBench

We collect problems from Codeforces, which offer following advantages to our benchmark:

**High difficulty.** Even state-of-the-art LLMs have low pass rates on Codeforces problems [32]. Compared to simple and straightforward code implementation, competitive problems can make multi-turn, iterative solution process more realistic and meaningful.

**High quality.** Unlike many open-source repositories, Codeforces publishes official editorial and provides automated correctness checks, ensuring the integrity of our dataset's problems.

**Periodic updates.** To maintain uncontaminated test data and ensure fairness, it is essential that benchmarks can be refreshed regularly. Codeforces releases new problems at a high frequency, enabling continuous updates for CodeFlowBench.

### B Data Curation Pipeline

#### B.1 Detail of Problem Scraping

The original problem contain two parts. The first part is scraped from its corresponding codeforces official website. An example problem page is shown in Figure 6.

The second part is scraped by *Problemsets.Problems*<sup>2</sup> API provided by Codeforces. We record the rating and tags of each problem. Rating is a metric that reflect the difficulty of the problem and tags is a list that contains knowledge scope of the problem.

---

<sup>2</sup><https://codeforces.com/api/problemset.problems>

## E. Opening Portals

time limit per test: 2 seconds  
memory limit per test: 256 megabytes

Pavel plays a famous computer game. A player is responsible for a whole country and he can travel there freely, complete quests and earn experience.

This country has  $n$  cities connected by  $m$  bidirectional roads of different lengths so that it is possible to get from any city to any other one. There are portals in  $k$  of these cities. At the beginning of the game all portals are closed. When a player visits a portal city, the portal opens. Strange as it is, one can teleport from an open portal to an open one. The teleportation takes no time and that enables the player to travel quickly between rather remote regions of the country.

At the beginning of the game Pavel is in city number 1. He wants to open all portals as quickly as possible. How much time will he need for that?

### Input

The first line contains two space-separated integers  $n$  and  $m$  ( $1 \leq n \leq 10^5$ ,  $0 \leq m \leq 10^5$ ) that show how many cities and roads are in the game.

Each of the next  $m$  lines contains the description of a road as three space-separated integers  $x_i, y_i, w_i$  ( $1 \leq x_i, y_i \leq n$ ,  $x_i \neq y_i$ ,  $1 \leq w_i \leq 10^9$ ) — the numbers of the cities connected by the  $i$ -th road and the time needed to go from one city to the other one by this road. Any two cities are connected by no more than one road. It is guaranteed that we can get from any city to any other one, moving along the roads of the country.

The next line contains integer  $k$  ( $1 \leq k \leq n$ ) — the number of portals.

The next line contains  $k$  space-separated integers  $p_1, p_2, \dots, p_k$  — numbers of the cities with installed portals. Each city has no more than one portal.

### Output

Print a single number — the minimum time a player needs to open all portals.

Please, do not use the `%lld` specifier to read or write 64-bit integers in C++. It is preferred to use the `cin`, `cout` streams or the `%I64d` specifier.

### Examples

input	<input type="button" value="Copy"/>
3 3	
1 2 1	
1 3 1	
2 3 1	
3	
1 2 3	
output	<input type="button" value="Copy"/>
2	
input	<input type="button" value="Copy"/>
4 3	
1 2 1	
2 3 5	
2 4 10	
3	
2 3 4	
output	<input type="button" value="Copy"/>
16	
input	<input type="button" value="Copy"/>
4 3	
1 2 1000000000	
2 3 1000000000	
3 4 1000000000	
4	
1 2 3 4	
output	<input type="button" value="Copy"/>
3000000000	

### Note

In the second sample the player has to come to city 2, open a portal there, then go to city 3, open a portal there, teleport back to city 2 and finally finish the journey in city 4.

Figure 6: An example page of problems on Codeforces, which contains *problem ID, title, time and memory complexity limits, problem description, input description, output description, sample tests, notes of each problem*. The original problem is 196E

```

"problem-id": "1946E",
"problem-url": "https://codeforces.com/problemset/problem/1946/E",
"title": "E. Girl Permutation",
"time-limit": "time limit per test 2 seconds",
"memory-limit": "memory limit per test 256 megabytes",
"problem-description": "Some permutation of length n is guessed. You are given the indices of its prefix maximums and suffix maximums. Recall that a permutation of length k is an array of size k such that each integer from 1 to k occurs exactly once. Prefix maximums are the elements that are the maximum on the prefix ending at that element. More formally, the element a_i is a prefix maximum if a_i > a_j for every j < i. Similarly, suffix maximums are defined; the element a_i is a suffix maximum if a_i > a_j for every j > i. You need to output the number of different permutations that could have been guessed. As this number can be very large, output the answer modulo 10^9 + 7.",
"input": "Input\nEach test consists of several test cases. The first line contains a single integer t (1 ≤ t ≤ 10^4) — the number of test cases. Then follows the description of the test cases.\nThe first line of each test case contains three integers n, m_1 and m_2 (1 ≤ m_1, m_2 ≤ n ≤ 2 ⋅ 10^5) — the length of the permutation, the number of prefix maximums, and the number of suffix maximums, respectively.\nThe second line of each test case contains m_1 integers p_1 < p_2 < ... < p_{m_1} (1 ≤ p_i ≤ n) — the indices of the prefix maximums in increasing order.\nThe third line of each test case contains m_2 integers s_1 < s_2 < ... < s_{m_2} (1 ≤ s_i ≤ n) — the indices of the suffix maximums in increasing order.\nIt is guaranteed that the sum of n over all test cases does not exceed 2 ⋅ 10^5.",
"output": "Output\nFor each test case, output a single integer on a separate line — the number of suitable permutations modulo 10^9 + 7.",
"sample-test": {"input": ["6\n1 1 1\n4 2 3\n1 2 3\n5 3 4\n1 2 3\n2 3 4\n5\n20 5 4\n1 2 3 4 12\n12 13 18 20\n6 2 3\n1 3\n3 4 6"], "output": ["1\n3\n1\n0\n317580808\n10"]},
"note": "Note\nThe following permutations are suitable for the second test case:\n[1, 4, 3, 2]\n[2, 4, 3, 1]\n[3, 4, 2, 1]\n\nThe following permutations are suitable for the sixth test case:\n[2, 1, 6, 5, 3, 4]\n[3, 1, 6, 5, 2, 4]\n[3, 2, 6, 5, 1, 4]\n[4, 1, 6, 5, 2, 3]\n[4, 2, 6, 5, 1, 3]\n[4, 3, 6, 5, 1, 2]\n[5, 1, 6, 4, 2, 3]\n[5, 2, 6, 4, 1, 3]\n[5, 3, 6, 4, 1, 2]\n[5, 4, 6, 3, 1, 2]"],
"rating": 2200,
"tags": ["combinatorics", "dp", "math", "number theory"],

```

Figure 7: An example of original coding problem we obtained in stage I. To make the content more clear, we remove html denotations that contained in original content. The original problem is 1946E, which is used for illustration in Figure 3

Combine the above two part, we obtain a original coding problem for CodeFlowBench, a full example is shown in Figure 7.

## B.2 Detail of Solution Scraping

During the problem-scraping process, we also collected the links under the “Contest material” section on the right side of each problem page and identified which of those led to solution blogs. Crawling these editorial pages is fairly complex, because Codeforces’ official write-ups are hosted as personal blogs whose formats can vary over time, making content extraction more challenging. Although each round’s problems (e.g. “123A,” “123B,” etc.) live on separate pages, all of the editorials for a given numeric ID usually appear on a single blog page. We therefore need to assign each sub-problem’s write-up (A, B, C, ...) to its corresponding problem. To do this, our crawler first locates the distinct anchor points that mark each sub-problem section, then extracts the content between each anchor and the next as that problem’s editorial. The benefit of this approach is that, while in the end we only need the solution code itself, editorials often consist of plain text explanations, a single code snippet, or multiple code variants and languages. Relying on a purely mechanical scraper makes it difficult to isolate exactly the code we want, so it’s more effective to pass the raw editorial content to an LLM for final organization and extraction.

In general, there are two ways to locate an anchor: by URL and by text. Blogs label problems in many different forms. However, most blogs make that label into a hyperlink pointing back to the original problem, which gives us a reliable way to identify the anchor.

Therefore, our primary and most precise method is to search for a URL containing the problem ID (for example, “problemset/problem/2060/A” or “contest/blog/2060/A”) and treat its position as the anchor. Once the anchor is found, we scan the surrounding page and extract its content to obtain

the editorial. Anchor scanning and recording also relies on a problem-ID reference table, which we built from the IDs of all problems scraped in the first step. Its main role is to guide the code when matching anchors: for instance, if the table shows that numeric ID 2060 has sub-IDs “A” through “G”, the scraper first reads those sub-IDs, then walks through the page using the URL-based or text-based method to record the exact anchor for each sub-ID. An example that fit URL anchor identification technique in shown in Figure 8a.

However, there are still little parts of early editorial blogs didn’t include URL hyperlinks, so in those cases we fall back on regular-expression-based text matching wherever possible. Based on the formats we observed, we designed two main matching strategies:

- **Difficulty-label matching.** A number of blogs publish all of a round’s problems on one page and mark them with labels like “Div2” or “Div1” (since most rounds contain two Div2-level problems and several Div1-level ones). To handle this, we use our problem-ID reference table to identify all sub-IDs belonging to the same numeric contest but with different difficulty levels, tag them accordingly in the table, and then, during the anchor-matching process, if the scraper detects a difficulty label it will also try to match anchors based on that label. An example is shown in Figure 8b.
- **Problem-label matching.** Beyond difficulty tags, many blogs use the literal “Problem A”, “Problem B”, etc. format. We include a specific regex pattern to detect those “Problem+sub-ID” labels and assign each section to the correct sub-problem.

[Codeforces Round 998 \(Div. 3\) Editorial](#)

By [erry](#), 3 months ago,

[Rating Predictions](#)

**2060A - Fibonacciness**

Problem Credits: [Proof\\_by\\_QED](#)

Analysis: [larush](#)

[Solution](#) [Solution 2](#) [Code \(C++\)](#) [Rate The Problem!](#)

**2060B - Farmer John's Card Game**

Problem Credits: [Lilypad](#)

Analysis: [larush](#)

[Solution](#) [Code \(C++\)](#) [Rate The Problem!](#)

**2060C - Game of Mathletes**

Problem Credits: [LMeyling](#)

Analysis: [macaqueDev](#)

[Solution](#) [Code \(C++\)](#) [Rate The Problem!](#)

**2060D - Subtract Min Sort**

Problem Credits: [Proof\\_by\\_QED](#)

Analysis: [Proof\\_by\\_QED](#)

[Solution](#) [Code \(C++\)](#) [Rate The Problem!](#)

[Codeforces Round #138, tutorial.](#)

By [I\\_love\\_Nasty](#), 13 years ago, translation,

**Div2 A. Parallelepiped**

You were given areas of three faces of a rectangular parallelepiped. Your task was to find the sum of lengths of its sides.

Let  $a$ ,  $b$  and  $c$  be the lengths of the sides that have one common vertex. Then the numbers we are given are  $s_1 = ab$ ,  $s_2 = bc$  and  $s_3 = ca$ . It is easy to find the lengths in terms of faces areas:  $a = \sqrt{s_1 s_2 / s_3}$ ,  $b = \sqrt{s_1 s_3 / s_2}$ ,  $c = \sqrt{s_2 s_3 / s_1}$ . The answer is  $4(a + b + c)$ , because there are four sides that have lengths equal to  $a$ ,  $b$  and  $c$ . The complexity is  $O(1)$ .

**Div2 B. Array**

You were given an array  $a$  consisting of  $n$  integers. Its elements  $a_i$  were positive and not greater than  $10^5$  for each  $1 \leq i \leq n$ . Also you were given positive integer  $k$ . You had to find minimal by inclusion segment  $[l, r]$  such that there were exactly  $k$  different numbers among  $a_l, \dots, a_r$ . The definition of the “minimal by inclusion” you can read in the statement.

Let us make a new array  $cnt$ . In the beginning its element  $cnt_i$  is equal to number of occurrences of number  $i$  in array  $a$ . It is possible to make this array because elements of  $a$  are not very large. Amount of nonzero elements in  $cnt$  is equal to amount of different elements in  $a$ . There is no solution if this number is less than  $k$ .

If it is not true, we have to find the answer segment  $[l, r]$ . In the beginning let  $[l, r] = [1, n]$ . We decrease its right end  $r$  by 1 until amount of different elements on the segment  $[l, r]$  is less than  $k$ . We can keep the amount of different numbers in following way: we decrease  $cnt_{a_r}$  by 1 if we delete element number  $r$ . Then we have to decrease current number of different elements by 1 if  $cnt_{a_r}$  becomes zero. After this we return the last deleted element back to the segment in order to make amount of different elements equal to  $k$ . Then we have to do the same with the left end  $l$ , but we have not to decrease but to increase its value by 1 on each step. Finally, we get a segment  $[l, r]$ . The amount of different numbers on it is equal to  $k$  and on every its subsegment is less than  $k$ . Therefore, this segment is an answer. The complexity is  $O(n)$ .

**Div2 C/Div1 A. Bracket sequence**

You were given a bracket sequence  $s$  consisting of brackets of two kinds. You were to find regular bracket sequence that was a substring of  $s$  and contains as many <>-braces as possible.

We will try to determine corresponding closing bracket for every opening one. Formally, let a bracket on the  $i$ -th position be opening, then the closing bracket on the position  $j$  is corresponding to it if and only if a substring  $s_{i-1} \dots s_j$  is the shortest regular bracket sequence that begins from the  $i$ -th position. In common case there can be brackets with no corresponding ones.

We can string the  $s$  and put positions with the opening brackets into a stack. Let us proceed the  $i$ -th position. If  $s_i$  is an opening bracket we simply put it on the top of the stack. Otherwise, we have to clean the stack if the stack is empty or the bracket on the top does not correspond to the current one. But if the bracket on the top is ok we just remove the top of the stack and remember that the bracket on position  $i$  is corresponding to the bracket removed from the top. So, we find all the correspondings for all the brackets.

Then we can split  $s$  into blocks. Let block be a segment  $[l, r]$  such that the bracket on the  $r$ -th position is corresponding for the bracket on the  $l$ -th and there is no couple of corresponding brackets on positions  $x$  and  $y$  such that  $[l, r] \subset [x, y]$  and  $[l, r] \neq [x, y]$ . It is easy to understand that the blocks do not intersect and the split is unique. We can join the consequent blocks into the regular bracket sequences. We should join as many blocks as possible in order to get the maximal number of braces. We get several substrings that are regular bracket sequences after we join all the consecutive blocks. The answer is the substring that has the largest amount of braces <>. The complexity is  $O(|s|)$ .

(a) (b)

Figure 8: Subfigure (a) is an example of using URL anchor identification technique. The anchor here is each subtitle displayed as “2060A-Fibonacciness”. The blue font color of such subtitle indicates a URL to original problem page is setted. In practice, the existence of URL is identified by analysing the HTML code of this website. For this kind of website, we identify these subtitles as anchors and scrape the content between each subtitle. Subfigure (b) is an example of using div anchor identification technique. The anchor here is each subtitle displayed as “Div2A. Parallelepiped”. These kinds of subtitle occurs in a contest round that contain two div level problems. We use reference table to project the div notation to original problem id and then identify them as corresponding anchor.

```

"problem-id": "1946E",
"solutions":
[{
    "type": "text",
    "content": "Editorial First, if  $p_1$  is not equal to 1, or  $s_{m_2}$  is not equal to  $n$ , or  $p_{m_1}$  is not equal to  $s_1$ , then the answer is 0 for obvious reasons... Otherwise, we know exactly where the number  $n$  is located, at position  $s_1$ . Next, we have  $\binom{n-1}{s_1-1}$  ways to divide the numbers from 1 to  $n-1$  into two sets — the numbers that will be in the left part and the numbers that will be in the right part (the left part — all indices  $< s_1$ , the right part — all indices  $> s_1$ ). We solve for the left part, and similarly for the right part. For the left part, the position of the maximum ( $p_{m_1-1}$ ) is again defined, and the maximum itself is also unique in the set of numbers for the left part, so we can again divide the left part into two, with  $\binom{p_{m_1}-2}{p_{m_1-1}-1}$  ways to do so, but we can also arrange the numbers between indices  $p_{m_1-1}$  and  $p_{m_1}$  (non-inclusive) in any order, i.e.,  $(p_{m_1} - p_{m_1-1} - 1)!$  ways. Then we solve similarly for the left set (i.e., for indices less than  $p_{m_1-1}$ ).
},
{
    "type": "code",
    "content": "#include<bits/stdc++.h>\nusing i64 = long long;\ntemplate<class T> constexpr T\npower(T a, i64 b) {...}"
}]

```

Figure 9: The scraped and processed solution we obtained in stage II. The original problem is 1946E

After crawling the editorial for each problem, we applied a series of processing steps to ensure quality. Firstly, we removed any editorials that were too short or empty—these problems were excluded from the dataset. To make it easier for an LLM to understand and process them, we then split each editorial into two parts: the code solution and the textual explanation, so that the model can consult the code first and then the accompanying text. The processed result is shown in Figure 9.

### B.3 Detail of Solution Code Generation

Although we've already scraped the official solution for each problem, an LLM-based post-processing step is still required for two main reasons: (1) **Presence of “global code segments”**. Some solutions include code that isn't encapsulated in any function. We must split the entire codebase into multiple functions and ensure that the top-level function can fully solve the problem. Since these global segments can't be recognized during standard parsing, we rely on an LLM to reorganize the provided code so that it becomes fully parsable. (2) **Early solutions exist only as text**. Some of the older official solutions consist solely of textual descriptions without any runnable code. We need an LLM to convert those narratives into executable code. The prompt template for code convert is shown in Figure 10. The Codeforces official judging system is used to verify code correctness. We employ an automated submission bot that navigates to the Codeforces submission page<sup>3</sup>, fills in all required fields, and submits the solution. The site will be automatically redirected to the results page after submission, from which we scrape the verification outcome.

### B.4 Detail of Subproblems Generation

While the parsing process is automatically, a LLM is still needed for generating natural language description for each subproblem. The prompt template is shown in Figure 11. A example of subproblems is shown in Figure 12.

### B.5 Detail of Test Cases Generation

Overall, CodeFlowBench's test suite is composed of two parts:

- **Top-level function tests** For the final subproblem—the top-level function (e.g., `main` or `solve`) that handles overall input and output—we use the public test cases provided by the Codeforces platform.

---

<sup>3</sup><https://codeforces.com/problemset/submit>

**Here's the information for the problem:**

```
"problem-id": "{problem_id}",
"title": "{title}",
"time_limit": "{time_limit}",
"memory_limit": "{memory_limit}",
"problem_description": "{problem_description}",
"input": "{problem_input}",
"output": "{problem_output}",
"sample-test": "{sample_testcases}",
"note": "{note}",
```

**Here's the editorial of this problem:**

```
"solution_text_part": "{text_solution}",
"solution_code_part": "{code_solution}",
```

**Your task is to generate a correct Python code solution that solves the problem, here's the instruction:**

1. If the "solution\_code\_part" is not empty and contain valid code, you should only focus on that part. If the original code is not in Python, convert it to Python. You can directly check the code and output it as your generated code.
2. Remember if the "solution\_code\_part" is not empty and contain valid code, use the existing code as always the priority, but if the "solution\_code\_part" is empty, you should analyze the "solution\_text\_part" field together with the problem details provided above and generate the Python code-solution that implements the solution with the following characteristics:
  - There should be a main/solve function that handle the overall logic and I/O.
  - Each function should have a single responsibility.
  - The main solver function should handle input/output and orchestrate the helper functions.
  - Ensure the code is properly indented and formatted. Remember to use 4 spaces for indentation.
  - Ensure that you have set proper new line notation before the 'if \_\_name\_\_ == "\_\_main\_\_":' line and "import" statements.
  - Avoid nest function in another function, make each function separated.
  - Each line of code should be contained in a function except calling packages.
3. At last, return your generated code in the "code-solution" field of the output JSON, following the format specified in the system instructions.

Example output data:

```
```json
{
"problem-id": "2055E",
"code-solution": "..."
}```
```

Please ensure the code you generate can solve the problem correctly, which means:

- The code should be free of syntax errors.
- The code should run correctly on the sample test provided and other edge cases.
- The code should have optimal time and memory complexity to solve the problem without surpassing the time and memory limits.
- Each line of code should be contained in a function except calling packages.

You only need to return the json with the "code-solution" field filled following the ```json mark in the next line.  
```json

Figure 10: The prompt template used for code conversion in stage III. The whole content of the example output data is not shown for length limitation.

**Here is the problem information and analyzed functions:**

```
Title: {title}
Time Limit: {time_limit}
Memory Limit: {memory_limit}
Problem Description: {problem_description}
Input: {problem_input}
Output: {problem_output}
Note: {note}
Extracted Functions:
{json.dumps(subproblems, indent=4)}
Text Solution: {solution_text_part}
Missing Class: {missing_class}
```

**Your task is to:**

1. Based on the function that "Extracted Functions" field contains, for each Class and Function , use the problem information provided to write a subproblem description that includes:
  - Name of the function/class.
  - The purpose of the function/class.
  - How it contributes to solving the overall problem.
  - Any dependencies it has on other functions. Set a reminder of functions that need to be called.
2. Return the results in the following JSON format. Use the subproblem statement you write to fill in the "statement" field and the information of each function to fill the "depth" and "dependencies" fields.

```
'''json
{
"problem-id": "{problem_id}",
"subproblems": [
{
"name": "...",
"statement": "...",
"solution": "...",
"depth": "...",
"dependencies": [...]
},
]
}
'''
```

**Reminder:**

- Please follow the system instruction and ensure that each function is described accurately and comprehensively. If a function has no dependencies, you can just return the main or top-level function description.
  - For the solution code of each subproblem, please return the whole complete function definition, including the function signature, parameters, and body.
  - You need to output the subproblems following the dependency order, starting from the functions that have no dependencies and let the function which depends on other functions be described later.
  - Don't forget to include both the whole Class and the function inside the Class as solution in the solution code as a subproblem, otherwise there will be a gap between the funciton in the class and outside the class.
- You only need to return the json with the "subproblems" list filled following the "'''json" mark in the next line.

```
'''json
```

Figure 11: The prompt template used for generating natural language description for each subproblem in stage III.

```

"problem-id": "1946E",
"subproblems":
[
  {
    "name": "comb",
    "statement": "Compute the binomial coefficient C(n, k) modulo 10^9 + 7. This function is essential for calculating the number of ways to choose k elements from a set of n elements, which is a key part of determining the number of valid permutations in the problem. The function uses precomputed factorials and inverse factorials to efficiently compute the result.",
    "solution": "MOD = 10 ** 9 + 7\nfact = [1] * max_fact\ninv_fact = [1] * max_fact\n\ncomb(n, k):\nif n < 0 or k < 0 or n < k:\n    return 0\nreturn fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD",
    "depth": 1,
    "dependencies": []
  },
  {
    "name": "solve",
    "statement": "Read input, process each test case, and compute the number of valid permutations. The function first checks if the given prefix and suffix maximums are valid. If not, it outputs 0. Otherwise, it calculates the number of valid permutations by dividing the problem into left and right parts, using the 'comb' function to compute binomial coefficients and factorials to account for the arrangement of elements between indices. The result is computed modulo 10^9 + 7 and printed for each test case.",
    "solution": "MOD = 10 ** 9 + 7\nfact = [1] * max_fact\n\nsolve():\nimport sys\ninput = sys.stdin.read().split()\nptr = 0\nptr = int(input[ptr])\nptr += 1\nfor _ in range(0):\nn = int(input[ptr])\nm1 = int(input[ptr + 1])\nm2 = int(input[ptr + 2])\nptr += 3\np = list(map(int, input[ptr:m1]))\nptr += m1\ns = list(map(int, input[ptr:ptr + m2]))\nptr += m2\nif p[0] != 1 or s[-1] != n or p[-1] != s[0]:\n    print(0)\n    continue\ns = s[0]\nres = comb(n - 1, s0 - 1)\nfor i in range(m1 - 2, -1, -1):\n    next_p = p[i + 1]\ncurrent_p = p[i]\nk = next_p - current_p - 1\n\n    c = comb(next_p - 2, k)\nf = fact[k]\nres = res * c % MOD\n\n    res = res * f % MOD\nfor i in range(1, m2):\n    prev_s = s[i - 1]\ncurrent_s = s[i]\ntotal = n - prev_s - 1\nk = current_s - prev_s - 1\n\n    c = comb(total, k)\nf = fact[k]\nres = res * c % MOD\n\n    res = res * f % MOD\n\nprint(res)",
    "depth": 0,
    "dependencies": ["comb"]
  }
]

```

Figure 12: An example of subproblems we obtained in stage IV. The solution code of 1946E contains a `comb` function which serve as the basic tool and is reused in `solve` function to address the whole problem. It's obvious that in its AST, the `comb` function is the leave node in depth 1 and the `solve` function is the root node in depth 0.

- **Subproblem function tests** For every other subproblem (i.e. functions invoked by higher-level code), we wrap each function call in a helper that redirects `stdin` and `stdout` to an internal buffer and records the resulting I/O. This can generate redundant calls for the same function, so we apply two safeguards to keep the test suite concise: deduplication of identical test cases and limiting the total number of test cases per function. These measures ensure comprehensive coverage without unnecessary duplication or excessive test-case volume.

Such method can ensure the test case generation pipeline to be automatic and the case itself to be correct.

## C Mathematical Expression of $PD@k$ & $APD@k$

For a given problem, we define  $PD@k$  as the expected maximum pass depth over  $k$  independent trials of the model. Directly using only those  $k$  results leads to high variance, just as with the  $pass@k$  metric. By analogy to the unbiased estimator for  $pass@k$ , which leverages  $n$  trials ( $n > k$ ) to reduce variance, we derive a similar estimator for  $PD@k$ .

Let the pass depths from  $n$  trials be:

$$\{d_1, d_2, \dots, d_n\},$$

and let

$$d_{(1)} \leq d_{(2)} \leq \dots \leq d_{(n)}$$

denote these depths in ascending order. Then an unbiased estimator for  $PD@k$  is

$$PD@k = \sum_{j=k}^n d_{(j)} \frac{\binom{j-1}{k-1}}{\binom{n}{k}}. \quad (1)$$

To see how this arises, consider sampling a random subset of size  $k$  from the  $n$  depths. Let  $M = \max\{d_{i_1}, d_{i_2}, \dots, d_{i_k}\}$  be the maximum depth in that subset. An unbiased estimator of  $M$  is  $\mathbb{E}[M]$ . By construction  $\mathbb{E}[M] = PD@k$ , this estimator is *unbiased* since its expected value exactly equals the true expected maximum depth. Consider all possible values of  $M$ , we have:

$$\mathbb{E}[M] = \sum_m m P(M = m) = \sum_{j=k}^n d_{(j)} P(M = d_{(j)}). \quad (2)$$

The probability that  $M = d_{(j)}$  equals the probability of choosing  $d_{(j)}$  along with  $k - 1$  depths from the first  $j - 1$  smaller values:

$$P(M = d_{(j)}) = \frac{\binom{j-1}{k-1}}{\binom{n}{k}}. \quad (3)$$

Combining (3) and (4) immediately recovers (1).

Finally, we define

$$APD@k = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} PD@k(p) \quad (4)$$

the average  $PD@k$  over the set  $\mathcal{P}$  of all problems.

## D Experiment

### D.1 The Setting of Experiment

Due to the substantial size of the total question pool ( $N = 5258$ ), we implemented stratified sampling with proportional allocation across overall-depth categories to select 1,000 test samples, as detailed in Table 4. To validate sampling quality, we conducted a  $\chi^2$  test comparing the overall-turn distributions between the population and sampled data (Table 5). The statistical analysis yielded a p-value of 0.1246, indicating no significant difference ( $\alpha = 0.05$ ) in distribution patterns. This confirms the representativeness of our sampling strategy and ensures the validity of subsequent analytical outcomes.

Table 4: Comparison of Overall-Depth Proportions Between the Population and the Sample

| Overall-Depth | # Population | Pop. Proportion (%) | # Sample     | Sam. Proportion (%) |
|---------------|--------------|---------------------|--------------|---------------------|
| 1             | 1,488        | 28.3                | 283          | 28.3                |
| 2             | 2,751        | 52.3                | 523          | 52.3                |
| 3             | 870          | 16.5                | 165          | 16.5                |
| 4             | 125          | 2.4                 | 24           | 2.4                 |
| $\geq 5$      | 24           | 0.5                 | 5            | 0.5                 |
| <b>Total</b>  | <b>5,258</b> | <b>100.0</b>        | <b>1,000</b> | <b>100.0</b>        |

Table 5: Comparison of Overall-Turn Proportions Between the Population and the Sample

| Overall-Turn | # Population | Pop. Proportion (%) | # Sample     | Sam. Proportion (%) |
|--------------|--------------|---------------------|--------------|---------------------|
| 1            | 1,488        | 28.3                | 283          | 28.3                |
| 2            | 2,158        | 41.0                | 437          | 43.7                |
| 3            | 990          | 18.8                | 188          | 18.8                |
| 4            | 402          | 7.6                 | 58           | 5.8                 |
| 5            | 149          | 2.8                 | 21           | 2.1                 |
| 6            | 46           | 0.9                 | 6            | 0.6                 |
| $\geq 7$     | 25           | 0.5                 | 7            | 0.7                 |
| <b>Total</b> | <b>5,258</b> | <b>100.0</b>        | <b>1,000</b> | <b>100.0</b>        |

## D.2 Instruction Templates

### Multi-turn Test

You are a Programming Expert. You always provide correct and reliable code solutions. You will be provided with the background of the whole problem, a programming problem and may also some pre-implemented functions. If pre-implemented functions provided, you need to call the pre-implemented functions and write a new function to solve the problem.

## Background of the whole problem:  
{problem\_description}

## Problem Description: You need to complete name function.  
{statement}

## Dependency information:  
To solve the problem, you need to utilize the ## Pre-implemented functions {dependencies} provided.

## Pre-implemented functions:  
{history}

## Guidelines:  
- Ensure the function is executable and meets the requirement.  
- Handle ## Dependency information correctly.  
- Provide clear and concise comments to explain key parts of the code.

Return your response by filling the function body following the function signature provided. Just generate the function and don't output any examples.

In the instruction template for multi-turn testing, we implemented a set of basic heuristics to adapt to different problem types. For example, if a question has no dependencies, we omit the “## Dependency information” section. If no prior code is provided—which is common when the overall depth is 1—we exclude the “## Pre-implemented functions” section. Furthermore, if it is the final turn of the code, we append the following snippet:

```
import sys
def name():
    input = sys.stdin.read().split()
```

### Single-turn Test

You are a Programming Expert. You always provide correct and reliable code solutions. You are required to solve a problem which consists of multiple subproblems, each with its own requirements. You will be provided with the background of the problem and description of all subproblems. You need to generate the complete implementations for all subproblems in a single response. The response for the final subproblem will be tested using `stdin` and `stdout`. Ensure the corresponding code meet this requirement.

```
## Background of the whole problem:  
{problem_description}  
  
## Problem Description:  
{combined_subproblem_description}  
## Subproblem {name}  
# Description:  
You need to complete {name} function.  
{statement}  
To solve the problem, you need to utilize your pre-implemented function {dependencies}.  
  
## Guidelines: - Ensure that all functions are executable and meet their respective requirements.  
- For each subproblem, correctly handle any dependency information.  
- Provide clear and concise comments explaining the key parts of the code.  
- For the last subproblem name, please use 'import sys\ndef {name}():\n    input = sys.stdin.read().split()\n' as  
the beginning.
```

Return your response by generating all functions in a single code block.

Similarly, if the problem does not have any dependencies, we will also omit the section related to `{dependencies}`.

### D.3 Pass Turn Result

Table 6: Multi-turn pass@turn results for various models.

| Model                            | Average | Turn-1 | Turn-2 | Turn-3 | Turn-4 | Turn-5 |
|----------------------------------|---------|--------|--------|--------|--------|--------|
| <i>Closed-Source</i>             |         |        |        |        |        |        |
| o3-mini                          | 0.600   | 0.322  | 0.632  | 0.777  | 0.983  | 0.571  |
| o1-mini                          | 0.581   | 0.233  | 0.645  | 0.798  | 0.879  | 0.667  |
| GPT-4.1-mini                     | 0.646   | 0.265  | 0.760  | 0.803  | 1.034  | 0.762  |
| GPT-4o                           | 0.537   | 0.177  | 0.604  | 0.697  | 1.017  | 0.476  |
| GPT-4o-mini                      | 0.467   | 0.137  | 0.501  | 0.638  | 0.982  | 0.476  |
| Gemini-2.0-flash                 | 0.541   | 0.183  | 0.595  | 0.702  | 0.948  | 0.667  |
| Claude-3.5-Sonnet                | 0.461   | 0.117  | 0.508  | 0.574  | 1.000  | 0.571  |
| <i>Open-Source (7B-Level)</i>    |         |        |        |        |        |        |
| Qwen2.5-Coder-7B-Instruct        | 0.258   | 0.018  | 0.270  | 0.394  | 0.638  | 0.286  |
| Llama-3.1-8B-Instruct            | 0.232   | 0.011  | 0.245  | 0.404  | 0.534  | 0.476  |
| Yi-Coder-9B-Chat                 | 0.247   | 0.021  | 0.314  | 0.324  | 0.362  | 0.286  |
| <i>Open-Source (32B-Level)</i>   |         |        |        |        |        |        |
| Qwen2.5-Coder-32B-Instruct       | 0.352   | 0.067  | 0.391  | 0.532  | 0.569  | 0.666  |
| QwQ-32B-Preview                  | 0.515   | 0.261  | 0.606  | 0.553  | 0.828  | 0.381  |
| <i>Open-Source (70B-Level)</i>   |         |        |        |        |        |        |
| Llama-3.3-70B-Instruct           | 0.493   | 0.163  | 0.515  | 0.681  | 1.000  | 0.476  |
| Qwen2.5-72B-Instruct             | 0.330   | 0.110  | 0.350  | 0.452  | 0.517  | 0.429  |
| <i>Open-Source (Large Model)</i> |         |        |        |        |        |        |
| Deepseek-V3                      | 0.572   | 0.219  | 0.622  | 0.750  | 0.966  | 0.810  |
| Deepseek-R1                      | 0.609   | 0.304  | 0.677  | 0.766  | 0.966  | 0.860  |

## E Discussion

### E.1 Discussion on Models' DSC Performance

For a problem, the DSC metric is defined as:

$$DSC(\text{problem}) = \frac{\text{Overall-Turns}(\text{problem})}{\text{Overall-Depth}(\text{problem})}$$

Recall that the overall-turn and overall-depth of a problem are derived from its AST, corresponding to the number of nodes and the depth of its AST. Based on this, we can see that a high *DSC* value indicates a problem with a complex dependency structure. Figure 13 presents the *pass@1* scores of models across different *DSC* intervals. It can be observed that most models are only capable of solving problems with *DSC* equal to 1, which corresponds to a simple linear dependency structure. Only a few leading models are able to solve a limited number of problems with *DSC* values below 1.33. All models struggle significantly when faced with problems involving more complex structures.

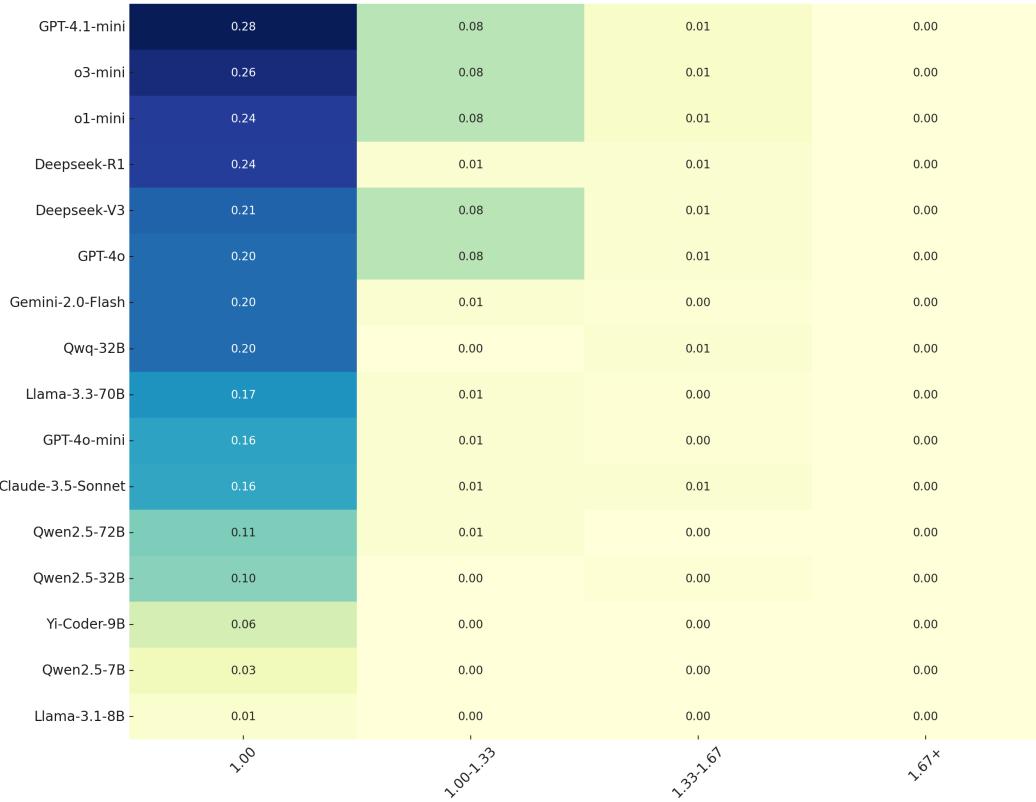


Figure 13: Heatmap of models' *pass@1* scores on multi-turn problems within different *DSC* intervals.

### E.2 Error Case Study

We have defined three typical error types for test in multi-turn pattern, in this part we will introduce several example to illustrate them. We have simplify and arrange model's outputs to make it clear to read. The content of "generated" field is model's output. The content of "harness\_result" field is the verification result by running it with test cases. "1" denotes accepted, "0" denotes wrong answer and "wrong" denotes running error.

```

"problem-id": "1799F",
"subproblems":
[
    {
        "name": "comb",...},
        {"name": "sub",...},
        {"name": "solve_case",
        "statement": "The solve_case function determines the minimum cost of an array after performing the allowed operations. It sorts the array in descending order and iterates over possible numbers of elements to which both operations are applied. For each scenario, it calculates the cost by applying the half and sub operations optimally. The function uses dynamic programming to efficiently compute the minimum cost, ensuring that the constraints on the number of operations are respected.",
        "depth": 1,
        "dependencies": ["half", "sub"],
        "test_code": [ {"input": "(3, 2, 1, 1, [9, 5, 3])", "output": "11"}, ... ],
        "generated":
            def solve_case(n: int, b: int, k1: int, k2: int, a: List[int]) -> int:
                # Sort in descending order to apply operations on the largest elements first
                a_sorted = sorted(a, reverse=True)
                min_cost = float('inf')
                # t = number of elements receiving both operations
                max_t = min(k1, k2, n)
                for t in range(max_t + 1):
                    rem_k1 = k1 - t
                    rem_k2 = k2 - t
                    temp = a_sorted.copy()
                    # Apply both operations to the first t elements
                    for i in range(t):
                        temp[i] = sub(half(temp[i]), b)
                    # Build list of potential gains for remaining elements
                    gains = []
                    for i in range(t, n):
                        orig = temp[i]
                        gain_half = orig - half(orig)
                        gain_sub = orig - sub(orig, b)
                        best_gain = max(gain_half, gain_sub)
                        gains.append((best_gain, gain_half, gain_sub, i))
                    # Prioritize by largest possible reduction
                    gains.sort(reverse=True, key=lambda x: x[0])
                    used1 = used2 = 0
                    for best_gain, g1, g2, idx in gains:
                        if used1 < rem_k1 and used2 < rem_k2:
                            if g1 >= g2:
                                temp[idx] = half(temp[idx])
                                used1 += 1
                            else:
                                temp[idx] = sub(temp[idx], b)
                                used2 += 1
                        elif used1 < rem_k1:
                            temp[idx] = half(temp[idx])
                            used1 += 1
                        elif used2 < rem_k2:
                            temp[idx] = sub(temp[idx], b)
                            used2 += 1
                        else:
                            break
                    # Compute total cost for this split
                    current_cost = sum(temp)
                    min_cost = min(min_cost, current_cost)
            return min_cost
        "harness_result": [1,1,1,1,1,1,0]
    }
]

```

Figure 14: Example of an Incomplete Reasoning (IR) Error by Deepseek-V3. The original problem is 1799F. The algorithm produced by the model fails to find the global optimum because it greedily sorts each element according to the single-step gain—choosing either the halving operation or the subtraction operation in isolation. In reality, achieving the global optimum requires a two-dimensional dynamic-programming solution that considers both operations jointly.

```

"problem_id": "2005D",
"subproblems": [
  {
    "name": "compute_prefix_suffix",
    "statement": "Compute the prefix and suffix GCDs for the given array. The prefix GCD at index i is the GCD of all elements from the start of the array up to index i. Similarly, the suffix GCD at index i is the GCD of all elements from index i to the end of the array. This function is essential for determining the GCDs of subarrays, which is crucial for finding the optimal swap range to maximize the sum of GCDs.",
    "depth": 1,
    "dependencies": [],
    "test_code": [
      "input": "([11, 4, 16, 17, 3, 24, 25, 8], 8)",
      "output": "[[0, 11, 1, 1, 1, 1, 1, 1, 1], [0, 1, 1, 1, 1, 1, 1, 8, 0]]}, ... ]",
    ],
    "generated": "
def compute_prefix_suffix(arr, n):
    prefix_gcd = [0] * (n + 2)
    suffix_gcd = [0] * (n + 2)
    # Compute prefix GCD
    prefix_gcd[0] = 0 # No elements before index 0
    if n >= 1:
        prefix_gcd[1] = arr[0]
        for i in range(2, n + 1):
            prefix_gcd[i] = gcd(prefix_gcd[i - 1], arr[i - 1])
    # Compute suffix GCD
    suffix_gcd[n + 1] = 0 # No elements after index n
    if n >= 1:
        suffix_gcd[n] = arr[n - 1]
        for i in range(n - 1, 0, -1):
            suffix_gcd[i] = gcd(suffix_gcd[i + 1], arr[i - 1])
    return (prefix_gcd, suffix_gcd)
",
    "harness_result": ["wrong", "wrong", "wrong", "wrong", ...]
  }
]

```

Figure 15: Example 1 of an Insufficient Globalization(IG) error by Deepseek-V3. The original problem is 2005D. The model's generated code omitted the import `math` statement, resulting in an error when calling `gcd`. This issue stems from improper handling of external imports.

```
"problem-id": "1734F",
"subproblems":
[
    {"name": "popcount",...},
    {"name": "solve",...}
    {
        "name": "main",
        "statement": "Read input, process multiple test cases, and output the Hamming distance for each test case. The function reads the number of test cases and for each test case, it reads the values of 'n' and 'm', clears the cache, and computes the Hamming distance using the 'solve' function. The results are then printed for each test case.",
        "depth": 0,
        "dependencies": ["solve"],
        "test_code": "[{"input": "[6\n1 1\n5 10\n34 211\n73 34\n19124639 56348772\n12073412269 96221437021]", "output": "[1\n6\n95\n20\n2808137\n48102976088\\n"}],...",
        "generated": "
import sys
def main():
    input = sys.stdin.read().split()
    ptr = 0
    T = int(input[ptr])
    ptr += 1
    for _ in range(T):
        n = int(input[ptr])
        m = int(input[ptr + 1])
        ptr += 2
        print(solve(n, m))
    "
    "harness_result": ["wrong"...]}
]"}
```

Figure 16: Example 2 of an Insufficient Globalization(IG) error by Deepseek-V3. The original problem is 1734F Because the program reads new inputs and performs fresh calculations for each test case, failing to clear the cache beforehand can inadvertently reuse stale results from a previous case. This leads to incorrect output due to unintended interaction between caching and I/O.

```

"problem-id": "1575H",
"subproblems":
[
    {
        "name": "build_transition",
        "generated": true,
        "def build_transition(b):
            m = len(b)
            prefix = compute_prefix(b)
            transition = [{} for _ in range(m)]
            match_counts = [0] * m
            for state in range(m):
                for c in ['0', '1']:
                    if state < m and c == b[state]:
                        next_state = state + 1
                    else:
                        next_state = 0
                    if state > 0:
                        j = prefix[state - 1]
                        while j > 0 and c != b[j]:
                            j = prefix[j - 1]
                        if c == b[j]:
                            next_state = j + 1
                        else:
                            next_state = 0
                    transition[state][c] = next_state
            # The match count for the final state is 1
            if state == m - 1:
                match_counts[state] = 1
            return transition, match_counts
    },
    {
        "name": "min_changes",
        "statement": "Compute the minimum number of changes required in the string 'a' to have exactly 'k' occurrences of the string 'b' as a substring. This function uses dynamic programming with three states: position in 'a', position in 'b', and the number of matches left. The transition table built by 'build_transition' is used to efficiently compute the state transitions.",
        "depth": 1,
        "dependencies": ["build_transition"],
        "test_code": [{"input": "('100101011', '101', 0)", "output": "1"}, ...],
        "generated": true,
        "def min_changes(a, b, k):
            n = len(a)
            m = len(b)
            if m == 0 or m > n:
                return 0 if k == 0 else -1 # edge cases
            transition, match_counts = build_transition(b)
            max_matches = n - m + 1
            # Initialize DP table: dp[state][matches] = min changes
            INF = float('inf')
            dp_prev = [[INF] * (max_matches + 2) for _ in range(m + 2)]
            dp_prev[0][0] = 0 # initial state: 0 matches, state 0, 0 changes
            for i in range(n):
                dp_next = [[INF] * (max_matches + 2) for _ in range(m + 2)]
                current_char = a[i]
                for state in range(m + 1):
                    for matches in range(max_matches + 1):
                        if dp_prev[state][matches] == INF:
                            continue
                        # Try both possible characters ('0' and '1')
                        for c in ('0', '1'):
                            new_state = transition[state][c] if state < m else 0
                            new_matches = matches
                            if new_state == m:
                                new_matches += 1
                                if new_matches > max_matches:
                                    continue # cannot exceed max_matches
                            cost = dp_prev[state][matches] + (0 if c == current_char else 1)
                            if cost < dp_next[new_state][new_matches]:
                                dp_next[new_state][new_matches] = cost
                dp_prev = dp_next
            # After processing all characters, find the minimal cost for exactly k matches
            min_cost = INF
            for state in range(m + 1):
                if dp_prev[state][k] < min_cost:
                    min_cost = dp_prev[state][k]
            return min_cost if min_cost != INF else -1
        "harness_result": [1,0,0,0,1,1,1]
    }
]

```

Figure 17: Example of an Instruction Misinterpretation(IM) error by Deepseek-V3. The original problem is 1575H. Although the DP routine correctly unpacks both transition and match\_counts from build\_transition(b), it merely treats match\_counts as an ordinary list. In reality, match\_counts carries two essential pieces of semantic information.(1)Match indicator. It denotes a complete match of b.(2)Backtrack hint. It's a hint in combination with the prefix function, indicating how far the automaton should jump back after a match to continue detecting overlapping occurrences. This error is caused by models' misinterpretation to the dependency relationship between tool function and top-level function

## F Impact Statement

This paper introduces CodeFlowBench, a comprehensive benchmark for evaluating code generation models in realistic multi-turn, dependency-driven development scenarios. For research, CodeFlowBench fills a critical gap by providing a standardized suite of tasks that require iterative reasoning, function dependency management, and end-to-end solution assembly. By exposing models' deficiencies in global awareness, instruction consistency, and dependency handling, CodeFlowBench will catalyze the design of new architectures and training paradigms that explicitly model iterative workflows and cross-turn coherence. Its open dataset and evaluation protocol invite the community to develop and compare dependency-aware generation strategies, driving progress toward more robust and developer-friendly code assistants.

In industry, CodeFlowBench offers a practical yardstick for assessing the readiness of AI coding tools in real-world software development, where tasks rarely appear as isolated single-step prompts. Integrating CodeFlowBench into CI/CD pipelines can help organizations detect and remediate weaknesses in model-powered code suggestions before deployment, reducing debugging overhead and technical debt. By highlighting the importance of function reuse, import management, and state consistency across revisions, CodeFlowBench insights can inform best practices for AI-augmented coding workflows, accelerating adoption of reliable co-programming solutions. There are broader societal implications in enabling safer, more maintainable AI-generated code, yet none that we believe warrant special emphasis here.

## NeurIPS Paper Checklist

### 1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

Answer: [Yes]

Justification: The main claims are clearly presented in the abstract and introduction, and are further elaborated and substantiated throughout the subsequent sections of the paper.

Guidelines:

- The answer NA means that the abstract and introduction do not include the claims made in the paper.
- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

### 2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [Yes]

Justification: Please refer to the Section 5 in the paper.

Guidelines:

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

### 3. Theory assumptions and proofs

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

#### 4. Experimental result reproducibility

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Please refer to the Section 3.1 and Appendix D.1 in the paper.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general, releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  - (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  - (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  - (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  - (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

#### 5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We provide well-documented repositories to store dataset and code. The URL can be found in the first page.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://nips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so “No” is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (<https://nips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

## 6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: Please refer to the Section 4 and Appendix D.1, D.2 in the paper.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

## 7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Please refer to the Section 4 and Appendix D.1 in the paper.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

## 8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Please refer to the Section 4 and Appendix D.1 in the paper.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

## 9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

Answer: [Yes]

Justification: This paper conforms, in every respect, with the NeurIPS Code of Ethics.

Guidelines:

- The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

## 10. Broader impacts

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Please refer to the Appendix F in the paper.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

## 11. Safeguards

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: The paper poses no such risks.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

## 12. Licenses for existing assets

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: The creators and original owners of all assets used in the paper are properly credited, and the licenses and terms of use are explicitly mentioned and fully respected.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, [paperswithcode.com/datasets](https://paperswithcode.com/datasets) has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

### 13. New assets

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [NA]

Justification: The paper does not release new assets.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

### 14. Crowdsourcing and research with human subjects

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: The paper doesn't include crowdsourcing or research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

### 15. Institutional review board (IRB) approvals or equivalent for research with human subjects

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

### 16. Declaration of LLM usage

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLM is used as a tool in data curation pipeline. Please refer to the Section 3.1 and Appendix B.3, B.4 in the paper.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (<https://neurips.cc/Conferences/2025/LLM>) for what should or should not be described.