

LLMs are Brittle to Simple Code Transformations: Introducing CETBench – A Benchmark for Code-Equivalence Checking

Neeva Oza^{1†*} Ishaan Govil^{1†*} Parul Gupta¹ Dinesh Khandelwal²
Dinesh Garg² Parag Singla¹

¹Indian Institute of Technology Delhi ²IBM Research

Abstract

We study how well LLMs can determine whether two programs are functionally equivalent. This is an important problem because benchmarking code equivalence helps assess LLM capability in tasks such as code rewriting and translation. To this end, we introduce CETBench— Code Equivalence with Transformations Benchmark — built from a repository of programs that may solve the same or different tasks. Each dataset instance is created by sampling a program pair and applying a random sequence of predefined code transformations, yielding either equivalent or non-equivalent pairs. Our analysis shows that even simple transformations cause a significant performance drop in state-of-the-art LLMs on code-equivalence checking. These challenges are further amplified in the cross-lingual setting when comparing programs written in different languages. To remedy this, we present a simple fine-tuning-based approach to boost LLM performance on the transformed pairs of programs. Our approach for dataset generation is generic, supporting cross-lingual equivalence checking, generation of program pairs with varying difficulty levels, and the application of diverse transformations. In our experiments, we perform ablations over the difficulty level of original programs, as well as the kind of transformations used in generating pairs for equivalence checking. Our analysis presents deep insights into the working of LLMs for the task of code-equivalence, and points to the fact that they may still be far from what could be termed as a semantic understanding of the underlying code.

1 Introduction

The field of AI-assisted code generation and understanding has witnessed a rapid rise in adoption and capability over the past few years, with models increasingly integrated into real-world software

development workflows (Jin et al., 2025). The reasons behind this trend include (i) the emergence of powerful, general-purpose large language models and (ii) the development of standardised benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), which evaluate text-to-code generation. In this work, our focus is on the related task of checking *code-equivalence*: given two pieces of code, p_1 and p_2 , how effective are LLMs in determining whether the two pieces of code are equivalent or not. While there are many definitions of equivalence in the literature, we focus on the most general one, which is *functional equivalence*, i.e., two programs are said to be equivalent if for any given input, they produce exactly the same output. Doing this task successfully can help with multiple auxiliary tasks, such as verifying the correctness of code rewrites, code translation, etc. While there has been a good amount of work in the classical literature on this task, using AST and program analysis (Baxter et al., 1998; Gabel et al., 2008), this is very tedious, does not scale, and often has multiple challenges. Of late, there have been some attempts to examine the capabilities of LLMs for the task of code-equivalence checking, with some of the approaches being parallel to our work (Maveli et al., 2025; Wei et al., 2025). We highlight the differences from these works, as well as contrast with other related work in the related work section.

In this work, we provide a general-purpose dataset to evaluate the robustness of LLMs for the undecidable task of code equivalence. The dataset is generated using the following steps: (a) we start with a repository of programs written to solve a variety of programming tasks. (b) We construct a set of pairs of programs by randomly picking programs which (i) solve the same task (either correctly or incorrectly) (ii) solve different tasks. (c) each pair in the set is taken through a series of (randomly picked) pre-defined transformations, such

* Equal Contribution, [†]Work done while at IIT Delhi.
Corresponding Authors:
neevahoza@gmail.com, ishaangovil870@gmail.com

as *variable renaming*, *swap of if-else blocks*, etc., to generate the corresponding perturbed pair. We note that some of the transformations are semantic preserving (sp), and others are not (np). These steps result in the dataset of perturbed pairs, and are referred to as CETBench. We perform a detailed analysis of relative performance among different pair categories, before and after introducing the perturbations. Our analysis shows that for a variety of SOTA LLMs, including both open source and closed source, the performance drops significantly on the perturbed dataset. Of specific interest is the drop in performance when the starting pair represents a single program. We present additional insights on LLM performance, we vary the difficulty level (MikeMirzayanov, 2018; Ebtekar, 2019) of the original program repository. Additional ablations present the variation in performance across a variety of LLMs, with varying kinds of transformations applied. Furthermore, we extend our analysis to the cross-lingual setting, revealing that these challenges are exacerbated when comparing code from different programming languages.

Finally, we present a simple fine-tuning-based approach to boost LLM performance on the unperturbed as well as perturbed datasets. Best results are obtained when the fine-tuning is done using a combination of pairs from the perturbed as well as unperturbed datasets. We also demonstrate that the knowledge gained from this fine-tuning successfully transfers to unseen, more difficult tasks in a zero-shot setting. Overall, our experiments present deep insights into the working of LLMs for the task of code-equivalence checking for a variety of categories. Further, our findings clearly point to the fact that LLMs are still quite far from semantic understanding of underlying programs, and likely rely on statistical patterns for the task of code equivalence.

The contributions of this paper can be summarised as follows: (a) We present a novel dataset of perturbed program pairs generated via a series of code transformations applied randomly to pairs of programs in a repository. (b) We present a detailed analysis demonstrating a significant drop in LLM performance over the perturbed program pairs. We also present additional ablations across varying program difficulty, cross-lingual settings, and the types of transformations applied. (c) We propose a fine-tuning approach that not only boosts performance on the perturbed dataset (proving LLMs can learn with explicit supervision) but also demonstrates

strong generalizability, as the learned skills successfully transfer to more difficult and cross-lingual tasks. *We release our dataset and code here*¹.

2 Related Work

Determining code equivalence—whether two programs yield identical outputs for all possible inputs—is a foundational challenge in computer science, relevant to program verification (Chakraborty et al., 2023), refactoring (Liu et al., 2025), translation (Roziere et al., 2020), and optimization (Rosas et al., 2024). While undecidable in the general case (Goldblatt and Jackson, 2012), several heuristic and formal methods have been developed.

Traditional Approaches: Static program analysis techniques (e.g., AST comparison (Baxter et al., 1998), data-flow analysis (Gabel et al., 2008)) struggle with semantics-preserving syntactic variations. Formal methods like symbolic execution (Kim et al., 2011) and SMT solving (Stolee et al., 2014) offer stronger guarantees but often lack scalability. Test-case-based execution is more practical in real-world settings; however, it has fundamental limitations in establishing true functional equivalence. This limitation arises because test suites inherently cover only a finite subset of potentially infinite program input domains and thus passing these tests merely indicates identical behaviour for those specific inputs rather than universal functional equivalence. Additionally, developing comprehensive test suites remains a significant practical challenge for complex, real-world software.

LLM-as-a-Judge for Code Evaluation: The *LLM-as-a-Judge* paradigm (Gu et al., 2024)—where LLMs are used to assess task outputs—has gained traction for its scalability and flexibility in both NLP (Zheng et al., 2023; Dong et al., 2024) and code-related tasks (Tong and Zhang, 2024; Zhuo, 2024; He et al., 2025). However, functional equivalence checking demands deeper semantic reasoning, making LLM reliability in this setting an open question. It requires more than understanding surface-level syntax or problem intent; it demands robust semantic reasoning to judge whether two programs, even if structurally dissimilar, will produce identical behaviour across all possible inputs.

Benchmarks for Code Equivalence: Prior datasets like BigCloneBench (Svajlenko et al.,

¹<https://github.com/dair-iitd/CETBench>

2014) suffer from false positives and class imbalance (Krinke and Ragkhitwetsagul, 2022). GPT-CloneBench (Alam et al., 2023) relies heavily on LLM-generated code, limiting its diversity. EqBench (Badihi et al., 2021) is limited in scale and diversity, with only 147 equivalent and 125 non-equivalent C/Java pairs, making it less suitable for robust LLM evaluation.

Recent Parallel Work: Two recent benchmarks are closely related to our work: (More detailed experimental comprision in Appendix E)

SeqCoBench: Maveli et al. (2025) apply over 20 single-step transformations to Python programs from MBPP (Austin et al., 2021), focusing on subtle functional changes. In contrast, CETBench applies composed sequences of up to seven transformations, often across distinct but algorithmically similar programs. Additionally, our work evaluates CETBench on frontier models such as GPT-4o, extending beyond the models used in SeqCoBench.

EquiBench: Wei et al. (2025) present a multi-language benchmark. While three of EquiBench’s six categories (OJ_A, OJ_V, OJ_VA) are Python-based and sourced from online judge submissions like CETBench, key differences remain:

- **Transformation Depth:** CETBench applies composed sequences of up to seven diverse transformations. In contrast, EquiBench’s OJ_A and OJ_V use single transformation types (algorithmic difference or variable renaming), while OJ_VA combines only two. Our compositional complexity provides a more rigorous test of LLM robustness.
- **Nature of Negative Pairs:** CETBench generates negative pairs via subtle, semantic-breaking changes between similar or identical programs, unlike EquiBench’s use of syntactically distinct wrong-answer submissions, which are often easier to classify.
- **Prompting Context:** EquiBench includes problem descriptions in prompts; CETBench omits them to focus solely on LLMs’ standalone code reasoning abilities.
- **Cross-Lingual:** CETBench is the first benchmark to include comparing two programs written in two different languages and applying a transformation on one.

3 CETBench: Constructing a Perturbation-Based Benchmark

To evaluate the ability of LLMs to determine program equivalence, we construct a perturbation-based benchmark that systematically generates *both* semantically equivalent and non-equivalent pairs of code. Our design mirrors practical scenarios in code translation (Roziere et al., 2020), repair (Le Goues et al., 2019), and synthesis (Devlin et al., 2017), where seemingly minor changes in the code can either leave the intended behaviour unchanged or silently change it.

3.1 Source Corpora

Our benchmark is constructed by extending the CodeContests dataset², originally introduced by DeepMind to train AlphaCode (Li et al., 2022). It consists of approximately 10000 competitive programming problems curated from platforms such as Codeforces³. Each problem includes complex problem descriptions, test cases (input–output pairs) for validation, correct/incorrect human solutions in multiple programming languages, and metadata including difficulty ratings, time/memory constraints, and competition source tags. For this benchmark implementation, we use Python, Java and C++ language solutions from the CodeContests dataset and perturb only the Python language solution. However, our perturbation-based methodology is general and can be applied to other programming languages and datasets, as long as they include correct solutions to the problems.

For each problem, we group the solutions that solve it into a *cluster*. We denote a cluster of solutions for a given problem i as P_i . Such a cluster P_i may contain multiple solutions, including those that are correct, incorrect, syntactically different but functionally equivalent, or even identical. The entire collection of these clusters, across all N problems, is denoted $\mathcal{P} = \{P_1, \dots, P_N\}$.

3.2 Perturbation Pipeline

Our pipeline begins by sampling an initial pair of human-written programs, (s_1, s_2) , from \mathcal{P} using the SAMPLEPAIR function (see Algorithm 1). Please note that for the benchmark dataset, \mathcal{P} is filtered to contain only Python programs. And for the cross-lingual setting described in Section 4.7, pro-

²https://github.com/google-deepmind/code_contests

³<https://codeforces.com/>

grams from multiple languages are filtered accordingly. This function is designed to select pairs that reflect diverse initial relationships—such as being identical, functionally equivalent (but syntactically different), one correct and one incorrect solution to the same problem, or correct solutions to entirely different problems (these initial relationships are further detailed in the dataset categorization Section 3.4). If the pair obtained after sampling from \mathcal{P} doesn’t fall under any one of these four types, it is discarded.

Once an initial pair of programs is selected, the algorithm applies a sequence of k perturbations ($1 \leq k \leq K_{\max}$). Before applying any perturbations, the algorithm randomly decides whether the full sequence should preserve the original semantic relationship between the program pair or introduce a semantic break. To preserve semantics, all k perturbations must be semantic-preserving. To break semantics, at least one of the k perturbations must be semantic-non-preserving, with the remainder (if any) drawn from the semantic-preserving variants. At each step, one of the two programs (s_1 or s_2) is randomly selected for modification. A perturbation type is then randomly selected from the set of available types described in the types of perturbation section 3.3, and the corresponding variant—semantic-preserving or semantic-non-preserving—is applied based on the overall goal for the sequence. Algorithm 1 formalizes the procedure for generating perturbed code pairs via controlled semantic modifications.

3.3 Types of Perturbations

We design seven distinct perturbation types to generate controlled code variations, each with *semantic-preserving* and *semantic-non-preserving* variants that simulate both benign refactorings and subtle logic-breaking edits. The semantic-preserving variants are theoretically behaviour-preserving transformations, empirically validated by executing the perturbed code against all test cases from the original problem; instances that fail validation are discarded. Similarly, semantic non-preserving transformations are validated and retained only if they cause at least one test case to fail for the program they are applied to, while all test cases continue to pass for the other program. This perturbation set thus enables structured control over code semantics, facilitating fine-grained evaluation of LLMs’ ability to distinguish syntactic from semantic changes. Table 1 summarizes all

Algorithm 1 Perturbed Program Pair Generation

Require: A set of solution clusters $\mathcal{P} = \{P_1, \dots, P_N\}$ with $P_i = \{s_{i1}, \dots, s_{i|P_i|}\}$; perturbation budget $K_{\max}, \text{max_tries}$

- 1: $\{s_1, s_2\} \leftarrow \text{SAMPLEPAIR}(\mathcal{P})$ ▷ With probability p_{same} both from one P_i , else from different clusters
- 2: $k \sim \text{Uniform}(1, K_{\max})$ ▷ Number of perturbations to apply.
- 3: $\text{preserve} \sim \text{Bernoulli}(0.5)$ ▷ Decide if the k -step perturbation sequence should be net semantic-preserving.
- 4: **if not** preserve **then**
- 5: $k_{\text{neg}} \sim \text{Uniform}(1, k)$ ▷ Number of semantic-non-preserving perturbations.
- 6: **else**
- 7: $k_{\text{neg}} \leftarrow 0$
- 8: **end if**
- 9: **for** $j = 1$ to $k - k_{\text{neg}}$ **do**
- 10: $\hat{s} \leftarrow \text{RANDOMCHOICE}(\{s_1, s_2\})$
- 11: Apply to \hat{s} : the *semantic-preserving version* of a randomly chosen perturbation type. Retry up to max_tries on failure.
- 12: **end for**
- 13: $\hat{s} \leftarrow \text{RANDOMCHOICE}(\{s_1, s_2\})$
- 14: **for** $j = 1$ to k_{neg} **do**
- 15: Apply to \hat{s} : the *semantic-non-preserving version* of a randomly chosen perturbation type. Retry up to max_tries on failure.
- 16: **end for**
- 17: **return** Perturbed solution pair

perturbation types, with detailed explanations and examples in Appendix A.3.

3.4 Dataset Categorization:

Each resulting pair from Algorithm 1 assigned a structured label $\langle \text{prefix}, k, \text{flag} \rangle$. The *prefix* encodes the initial relationship of the pair before any perturbations were applied. The value k is the total number of perturbations that were actually applied. The *flag* indicates the net semantic effect of the perturbations. The following are the four possible values of the *prefix*:

id	identical correct submissions
fe	functionally equivalent but syntactically different solutions
ne	one correct, one incorrect submission
di	correct submissions of <i>different</i> problems

Example: A pair starting from correct solutions

Perturbation Type	Semantic-Preserving Variant	Semantic-Non-Preserving Variant
If-Else Swapping	Negates the if condition and swaps the if/else blocks.	Negates the if condition but does not swap the blocks.
For-While Swapping	Correctly converts a for loop to an equivalent while loop.	Converts to a while loop with an intentional error (e.g., faulty iterator update).
If Condition Flipping	Rewrites the if condition to a logically equivalent form (e.g., $a < b$ to not ($a \geq b$)).	Inverts the relational operator in the condition (e.g., $a < b$ to $a \geq b$).
Variable Renaming	Consistently renames a variable across its entire scope.	Renames a variable inconsistently, breaking its semantic link in one or more places.
Boolean Inversion	Inverts a boolean variable's declaration and all its subsequent uses.	Inverts the variable's declaration but leaves its subsequent uses unchanged.
Statement Reordering	Swaps two statements that are data-independent.	Swaps two data-dependent statements, altering the execution logic.
Expression Reformatting	Refactors a sub-expression into a temporary variable without changing the final result.	Incorrectly refactors an expression, altering the final computational result.

Table 1: Summary of Perturbation Types. Each type has a semantic-preserving (SP) variant that maintains program logic and a semantic-non-preserving (NP) variant that intentionally breaks it.

to *different problems* (di), then perturbed twice ($k = 2$), with one of those perturbations being a semantic-non-preserving ($k_{\text{neg}} = 1$), would be labeled $\langle di, 2, np \rangle$.

Our benchmark results in program pairs classified into *eight categories* by their structured labels (reflecting four initial relationship types combined with two overall semantic impact flags). The dataset consists of *id* and *di* categories each containing 400 original program pairs, evenly split into 200 *np* and 200 *sp* perturbed samples, while the *fe* and *ne* categories each contain 2000 original pairs, with 1000 *np* and 1000 *sp* samples per category. For the benchmark test set, we chose $K_{\text{max}} = 5$. Hence, k ranges from 1 through 5. Please note that here, k is the count of perturbation applied to a given sample and each perturbation can be any of the 7 types of perturbations in Table 1. For more discussion on the choice of sample count, please refer to Table 11 in Appendix A and Appendix F. CETBench enables a comprehensive and fine-grained assessment of large language models under controlled semantic variation, facilitating systematic analysis of model robustness and failure modes across different solution types and perturbation impacts.

4 Experiments

4.1 Methodology

To observe the effect of perturbation on LLM’s performance in equivalence checking, we curate two parallel sets of code pairs. One set contains the categorised perturbed code pairs generated using our dataset generation method, while the other contains the corresponding original pairs prior to perturbation. For benchmarking, we use problems with a difficulty level of 900 from the CodeContests dataset. We prompt various LLMs to check whether the codes in a given pair are semantically equivalent for both sets. Detailed description of the prompts is present in the Appendix A. The LLMs used in the experiments are Qwen: Qwen2.5-Coder-7B-Instruct, Mistral: Mistral-7B-Instruct-v0.3, DSC: Deepseek-coder-6.7b-instruct, GPT-4o: GPT-4o-2024-08-06. We additionally report results for newer models (GPT-5-nano: GPT-5-nano-2025-08-07 and GPT-5.1: GPT-5.1-2025-11-13). We report accuracies and F1 scores.

4.2 Main Results

Table 2 shows the effect of perturbations on various models for the task of code similarity detection. There are four values corresponding to each category (id, fe, ne, and di). Within each row, the left column corresponds to non-semantic-preserving perturbations, and the right column to semantic-preserving perturbations. Overall accuracy and F1 scores are computed by aggregating results across all four categories, for both the original (top row) and perturbed (bottom row) code pairs. We observe a significant decrease in overall performance across 4 different LLMs. We observe that for the *id* category - i.e. pairs of identical code, all models have a significant drop in performance after perturbation, while having nearly 100% accuracy on the corresponding unperturbed set. Moreover, a similar trend is observed for *fe* category - i.e. functionally equivalent original pairs. This suggests that models severely struggle to tell similar-looking code as inequivalent on adding simple perturbations. This drop is surprising, since humans can easily spot such semantic differences. We observe proprietary model GPT-4o being able to gain an advantage in identifying inequivalent pairs, especially for *ne* category, by being able to spot cases of semantic inequivalence. However, it still remains vulnerable in the positive class. Both these effects tend to counterbalance each other, keeping the weighted

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	99.0	80.7	78.7	44.0	42.9	100.0	100.0	80.6	80.6	81.2	81.2	72.5	66.8
	30.5	98.0	39.7	76.9	61.4	47.6	99.5	100.0	69.2	69.2	78.7	78.7	69.2	78.7
Mistral	100.0	100.0	90.3	90.3	25.2	24.7	86.5	89.5	76.3	76.3	82.6	65.1	71.7	58.4
	0.2	98.0	16.5	86.5	33.0	27.8	90.0	89.0	56.6	68.3	64.4	49.9	44.8	54.9
DSC	100.0	100.0	100.0	99.9	0.1	0.3	2.5	2.0	50.6	50.6	51.1	32.5	63.1	1.9
	0.0	99.5	0.0	99.9	1.2	0.3	2.5	2.5	23.6	50.6	27.8	20.2	37.6	2.8
GPT-4o	99.5	100.0	72.5	73.0	67.7	66.7	100.0	100.0	84.9	84.9	91.1	76.8	75.5	78.1
	81.0	97.0	77.4	71.8	91.8	72.2	100.0	100.0	86.4	85.7	92.3	77.3	66.7	88.1
GPT-5-nano	100.0	100.0	69.8	71.8	87.1	86.6	100.0	100.0	89.4	89.4	94.0	83.4	81.1	85.7
	90.5	98.5	78.4	70.7	96.1	89.8	100.0	100.0	90.5	88.5	94.7	82.2	72.9	91.5
GPT-5.1	100.0	100.0	50.4	57.1	85.2	85.2	100.0	100.0	84.7	84.7	90.5	75.6	70.7	80.4
	93.0	90.5	92.6	54.8	96.9	88.2	100.0	100.0	89.5	83.9	93.8	79.4	67.3	91.4

Table 2: Results comparing performance of various models before and after perturbations. First 10 columns show accuracy values. W-Avg is average accuracy after re-weighting examples to have equal weight on positively and negatively labelled examples. Macro F1, Micro F1, and F1 scores of each individual labels are also reported.

accuracy nearly stable, though there is a significant decrease in F1 for the positive class. We observe similar trends for GPT-5-nano and GPT-5.1 as well. We also observe low performance for *fe* category for GPT-5.1 before perturbation.

4.3 Comparison to Existing Benchmarks

We present performance on the Python split of the EquiBench dataset (Wei et al., 2025) in Table 3a. EquiBench contains three categories: OJA, OJV, and OJVA. The positive-label pairs in OJA are analogous to our *fe* category before perturbation, while the negative-label pairs in OJA correspond to *ne* before perturbation. The positive and negative pairs in OJV are similar to $\langle id, k, sp \rangle$ and $\langle id, k, np \rangle$, respectively. Likewise, OJVA is similar to the perturbed *fe* category in our dataset. For SeqCoBench (Maveli et al., 2025), the dataset contains samples that are similar to the $\langle fe, 1, np \rangle$ and $\langle fe, 1, sp \rangle$ categories in CETBench, and the comparison results are reported in Table 3b. We observe that the perturbed pairs in our dataset are more challenging: performance decreases consistently after perturbation, whereas this decrease is not observed for the positive-label pairs when moving from OJA to OJVA. For results comparing against additional SOTA benchmarks, please see Appendix E.

4.4 Comparison with a Non-LLM Baseline

To evaluate the complexity of our dataset, we compare against ASTNN (Zhang et al., 2019), a specialized code clone detection model that does not rely on LLMs. ASTNN decomposes the abstract syntax tree into statement-level subtrees, encodes

them as vectors using an RNN, aggregates these representations with a GRU to obtain a program-level embedding, and applies a classifier to predict code equivalence. From Table 4, we observe that: (a) when trained on unperturbed data, ASTNN collapses to predicting equivalence and fails to learn discriminative patterns; (b) training on perturbed data increases sensitivity to transformations, but the improvement remains marginal and close to random performance; and (c) perturbed pairs remain challenging for ASTNN, with performance significantly lower than that of LLM-based models.

4.5 Ablation Analysis

We conduct a comprehensive analysis to understand how various models respond to three key factors: the presence of Chain-of-Thought (CoT) reasoning, different difficulty levels of the problem, and the type of perturbation applied.

4.5.1 Effect of Chain of Thought Prompt

From table 5, we observe that asking the model to first generate a CoT of reasoning before concluding to the final answers helps weaker models like Mistral, DeepSeekCoder significantly. We observe a significant boost in their performance in both original pairs as well as the perturbed pairs, as compared to prompting without chain of thought. However, we observe a drop in the performance of GPT-4o in perturbed pairs. This signifies the brittleness of reasoning for equivalence on the generated perturbations. We present the results for newer models for this and the following sections in Appendix C. We also experiment with self-consistent decoding

Model	OJA		OJV		OJVA	
	neg	pos	neg	pos	neg	pos
Qwen	39.0	83.5	36.0	99.5	41.0	83.5
Mistral	17.5	91.5	16.0	99.0	19.0	90.5
DSC	99.5	1.0	99.5	2.0	99.5	2.0

(a) Performance on EquiBench

Model	neg	pos
Qwen	92.2	92.9
Mistral	81.1	98.1
DSC	3.5	99.4

(b) Performance on SeqCoBench

Table 3: Comparing the performance of our Benchmark to existing Benchmarks

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0	50.0	50.0	50.0	31.6	63.2	0.0
ASTNN	47.5	59.5	43.3	56.7	43.8	43.1	43.0	39.0	47.0	50.7	63.6	44.1	33.0	55.3

Table 4: Results comparing performance of ASTNN baseline.

and observe the performance to be consistent with that in Table 2. Details and results are present in Appendix B.1.

4.5.2 Error Analysis

On human evaluation of the Chain of thought generations, it is observed that for original pairs of programs, LLMs struggle to correctly identify that two different algorithms are solving the same problem and hence are equivalent. For many of the $\langle fe, k, np \rangle$ samples where the model correctly generates the final answer as no, its reasoning is flawed - incorrectly stating that the difference in the algorithm leads to inequivalent behaviour. The model fails to focus on the perturbation and the error caused by it in the intended algorithm. Apart from this, a significant number of correct final generations have missing reasoning details despite explicit instructions in the prompt to add them. For detailed results and sample generation examples, please refer to the Appendix H.

4.5.3 Failure-Mode-Based Benchmarking

We also analysed how different types of perturbations impact the models by comparing the performance drop (delta) before and after applying individual perturbation; the detailed results are provided in Appendix B.3. On observing GPT-4o’s delta plot, we observe that it struggles on various perturbations except the Variable Renaming type of perturbation. We hypothesised that the presence of a relatively easy-to-spot perturbation is helping the model to label inequivalent pairs correctly. To test this hypothesis, we generated a separate test set consisting of all the types of perturbations ex-

cept Variable Renaming. In table 6, the results of GPT-4o on this set show a significant decrease in the $\langle fe, k, np \rangle$ category and a higher decrease in the $\langle id, k, np \rangle$ category, unlike in table 2. And we now observe a drop in GPT-4o’s average accuracy for a combination of different types of perturbations, similar to individual perturbations, pinning down its weakness. We also performed analysis of perturbing code pairs of different difficulty levels and found a consistent gap in perturbed pairs for each of the difficulty levels. Detailed results are presented in Appendix B.2.

4.6 Finetuning

We fine-tuned multiple large language models (LLMs) using three distinct data configurations: the original clean dataset, a perturbed version of the dataset, and a balanced 50:50 mixture of both. To assess performance, we evaluated each model against two test sets: one comprising unseen, unperturbed programs and another comprising perturbed programs. The models trained on the combined 50:50 dataset consistently achieved the best results (shown in table 7) for each LLM. (Other configurations are in Appendix D.) These optimal models not only demonstrated improved performance on both the unperturbed and perturbed test sets but also reached a level of performance comparable to GPT-4o. Further experimental details are available in Appendix A.

4.7 Cross-lingual Equivalence

We extend our benchmark to evaluate cross-lingual program equivalence. For this, we created a test set of Java-Python code pairs of the same size as

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
	100.0	98.5	56.8	57.7	54.5	57.5	99.0	100.0	79.6	79.6	87.3	69.0	67.6	70.4
Qwen	24.1	85.7	43.3	58.9	63.5	59.6	99.5	99.5	66.7	68.6	77.2	58.0	77.2	43.9
	100.0	98.9	56.8	57.7	63.7	64.5	96.0	93.0	78.8	78.8	86.9	68.4	65.5	71.5
Mistral	1.57	94.9	39.6	59.6	59.0	60.2	93.0	89.5	62.1	67.2	71.5	55.9	42.8	68.9
	98.9	98.7	87.2	87.2	23.8	24.0	89.0	86.0	74.3	74.3	81.1	62.4	69.4	55.5
DSC	16.7	92.3	18.8	85.2	30.4	26.0	92.0	89.5	56.3	48.5	65.8	67.1	43.8	53.2
	100.0	100.0	76.2	76.3	60.5	61.1	100.0	100.0	84.3	84.3	90.5	75.7	75.3	76.2
GPT-4o	65.5	99.0	47.6	74.6	74.0	66.4	100.0	100.0	78.3	81.1	86.6	68.0	56.6	79.5

Table 5: Results comparing various models using COT prompt

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
	100.0	100.0	81.0	78.0	67.0	67.0	100.0	100.0	86.6	86.6	92.2	89.0	87.0	90.1
GPT-4o	75.5	94.0	65.5	80.5	90.5	71.0	100.0	100.0	84.7	85.6	91.2	83.1	74.1	92.0

Table 6: Results on GPT-4o on combinations of all other perturbations except variable-renaming

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
	98.0	97.0	81.0	81.5	46.3	45.1	99.5	99.0	80.9	80.9	87.6	71.5	73.1	69.8
Qwen	82.7	96.5	82.2	90.1	85.3	39.4	99.5	97.5	84.2	87.3	90.1	75.8	66.9	84.8
	100.0	99.5	63.6	63.7	61.5	59.3	99.0	98.5	80.6	80.6	88.0	65.7	87.7	64.7
Mistral	77.0	87.5	76.7	67.6	87.5	55.6	100.0	98.0	81.2	80.0	88.9	71.2	58.5	83.9
	100.0	100.0	84.1	83.9	42.6	41.0	99.5	100.0	81.3	81.3	87.5	71.2	73.6	68.7
DSC	89.0	91.41	84.2	88.9	87.4	39.0	100.0	98.5	84.8	86.5	90.4	76.2	67.0	85.4

Table 7: Effect of finetuning

our monolingual sets. We generate two different variants of the cross-lingual dataset: (a) Single code perturbed, and (b) Both codes perturbed. In the single code perturbed cross-lingual setting, we apply perturbations only to the Python programs. We then compare the performance of LLMs on perturbed pairs and their corresponding original versions (unperturbed pairs). As shown in Table 8, the LLMs find it more difficult to verify equivalence for unperturbed cross-lingual pairs than for their monolingual counterparts. Furthermore, perturbing the Python program leads to an additional drop in performance, mirroring the trend observed in the monolingual setting. In the Both codes perturbed setting, we randomly select either the Java or the Python code to be perturbed. The results are shown in Table 9. They are quantitatively similar to those in Table 8, pointing to the finding that our analysis is agnostic to which language(s) the perturbation is applied to in the cross-lingual setting. Further details for cross-lingual datasets generation

are present in A.1. Similar results were observed for C++ and Python pairs, with detailed results available in Appendix B.

4.8 Transferability of Finetuned Models

To determine if the skills learned during fine-tuning generalize, we evaluated models in a zero-shot transfer setting. Models fine-tuned on 900-level Python-Python pairs were tested without further training on: (1) Python programs of higher difficulty levels, and (2) cross-lingual code pairs. Table 10a shows that finetuned models exhibit stable improvements across difficulty levels, indicating effective knowledge transfer. Similarly, Table 10b demonstrates consistent accuracy gains across Python-Python, Python-Java, and Python-C++ pairs, where the perturbation is always applied to a single Python program. Notably, models with initially poor performance improved the most, highlighting the benefits of our fine-tuning in challenging or noisy settings.

Model	fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp						
Qwen	87.5	87.8	33.4	33.9	100.0	99.5	73.7	77.2	81.2	68.3	72.1	64.4
	46.7	82.8	65.7	41.3	100.0	100.0	72.8	75.0	82.0	63.0	52.8	73.2
Mistral	46.7	82.8	63.7	64.5	96.0	93.0	78.8	78.8	86.9	68.4	65.5	71.5
	39.6	59.6	59.0	60.2	93.0	89.5	62.1	67.8	71.5	55.9	64.0	68.9
DSC	99.7	99.3	0.9	0.8	3.5	2.5	34.5	50.7	35.8	33.1	63.3	2.9
	2.2	99.5	2.6	1.0	10.5	2.5	19.7	51.6	22.5	21.8	38.0	5.6
GPT-4o	68.0	64.7	74.8	76.6	100.0	100.0	80.7	76.9	88.6	74.6	71.1	78.1
	79.7	64.8	93.4	79.5	100.0	100.0	86.2	75.9	92.1	75.3	62.4	88.2

Table 8: Results comparing various models for pairs with one Java code and one Python code

Model	fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp						
Qwen	81.6	84.4	37.9	40.9	100.0	100.0	73.9	76.0	82.3	67.7	68.3	67.2
	39.6	82.1	60.9	42.1	100.0	100.0	70.8	75.3	80.3	60.0	48.3	71.7
Mistral	93.6	93.4	12.6	12.9	72.0	73.0	59.6	68.1	67.8	54.7	64.7	44.8
	13.3	91.7	22.2	15.3	78.0	73.5	49.0	66.1	59.1	42.2	39.8	44.6
DSC	99.3	99.2	1.4	0.6	8.0	19.0	37.9	53.3	41.8	34.5	60.2	8.9
	2.0	98.9	2.4	1.2	16.5	13.5	22.4	53.0	27.1	22.0	35.2	8.8
GPT-4o	61.9	62.9	78.1	76.8	100.0	100.0	79.9	75.6	88.0	73.5	67.5	79.6
	83.5	61.9	95.7	77.4	100.0	100.0	86.4	76.6	92.1	74.7	60.3	89.2

Table 9: Results comparing various models for pairs with one Java code and one Python code (Both Perturbed)

Model	900		1200		1600		1900		2200		Model	Py-Py		Py-Java		Py-C++	
	B	A	B	A	B	A	B	A	B	A		B	A	B	A	B	A
Qwen	80.6	80.9	76.1	74.9	78.1	77.5	74.8	76.8	80.6	80.4	Qwen	81.4	82.7	73.7	75.6	73.3	67.1
	69.2	84.2	62.7	80.3	68.1	81.4	62.1	81.6	66.6	75.6		74.2	83.6	72.8	77.0	74.3	78.6
Mistral	76.3	80.6	67.0	84.3	65.5	79.4	66.7	77.5	58.2	77.3	Mistral	77.2	81.3	59.8	72.5	58.9	72.0
	56.6	81.2	46.3	77.4	44.3	75.5	46.2	72.7	40.4	72.5		57.4	79.7	51.5	79.6	48.2	77.2
DSC	50.6	81.3	52.3	82.9	50.7	86.1	53.0	78.1	49.9	81.4	DSC	50.6	81.8	34.5	75.3	34.7	75.5
	23.6	84.8	28.3	84.1	27.3	82.7	29.0	81.6	25.9	79.6		26.8	83.2	19.7	80.9	19.0	81.3

(a) Transfer across problem difficulty levels

(b) Transfer across languages

Table 10: Testing finetuned model across difficulty levels and cross-lingual setting. **B**: Before, **A**: After Finetuning

5 Conclusion

We have introduced CETBench, a comprehensive benchmark for evaluating LLMs on the task of code-equivalence checking via controlled program transformations. Our experiments show that even simple perturbations applied to program code can significantly degrade the performance of state-of-the-art LLMs, highlighting their limitations in understanding functional semantics. CETBench’s methodology is generalizable, supporting extension to multiple programming languages and transformation types, and enables deeper analysis of LLM capabilities. Fine-tuning with perturbed ex-

amples improves performance, suggesting room for enhancement through targeted supervision. These findings underscore that current LLMs rely heavily on surface patterns rather than true semantic understanding of code.

Limitations

While our analysis presents deep insights into the performance of LLMs on the task of code equivalence under various perturbations, several limitations remain. First, although we have performed experiments on a diverse set of models, the findings may not hold true for other models or future versions of the current models.

The dataset uses a binary Yes/No labelling scheme for functional equivalence, which oversimplifies nuanced cases where equivalence may be partial, conditional, or input-dependent. Furthermore, while the transformation set covers a wide range of syntactic and semantic changes, it is not exhaustive. Real-world coding variations—such as advanced refactoring patterns, concurrency-related modifications, or differences in library usage—are not fully captured.

Finally, although fine-tuning leads to performance gains within the distribution of the dataset, it remains unclear how well these models generalise to unseen perturbation types or real-world scenarios that lack the synthetic structure of our benchmarks.

Acknowledgements

This work was supported by an IBM AI Horizons Network (AIHN) grant and IBM SUR Awards. Parag is supported by Shanthi and K Ananth Krishnan Young Faculty Chair Professorship in AI, apart from the IBM AI Horizon Networks (AIHN) grant. We thank IIT Delhi HPC facility⁴ for computational resources, and Google Gemini Credits grant. We thank anonymous reviewers for their insightful comments that helped in further improving our paper. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views or official policies, either expressed or implied, of the funding agencies.

References

Ajmain I Alam, Palash R Roy, Farouq Al-Omari, Chanchal K Roy, Banani Roy, and Kevin A Schneider. 2023. Gptclonebench: A comprehensive benchmark of semantic clones and cross-language clones using gpt-3 model and semanticclonebench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–13. IEEE.

⁴<https://supercomputing.iitd.ac.in/>

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. *Program synthesis with large language models*. *ArXiv*, abs/2108.07732.

Sahar Badihi, Yi Li, and Julia Rubin. 2021. Eqbench: A dataset of equivalent and non-equivalent program pairs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 610–614. IEEE.

Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE.

Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking llm-generated loop invariants for program verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9164–9175.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. *Evaluating large language models trained on code*. *ArXiv*, abs/2107.03374.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR.

Yijiang Dong, Tiancheng Hu, and Nigel Collier. 2024. Can llm be a personalized judge? In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 10126–10141.

Aram Ebtekar. 2019. *How to interpret contest ratings*. Codeforces Blog. Accessed: 2025-05-18.

Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330.

Robert Goldblatt and Marcel Jackson. 2012. Well-structured program equivalence is highly undecidable. *ACM Transactions on Computational Logic (TOCL)*, 13(3):1–8.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*.

- Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2025. From code to courtroom: LLMs as the new software judges. *arXiv preprint arXiv:2503.02246*.
- Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2025. From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future. *Preprint, arXiv:2408.02479*.
- Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310.
- Jens Krinke and Chaoyong Ragkhitwetsagul. 2022. Big-clonebench considered harmful for machine learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE.
- Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Communications of the ACM*, 62(12):56–65.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, and 11 others. 2022. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097.
- Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study. *Automated Software Engineering*, 32(1):26.
- Nickil Maveli, Antonio Vergari, and Shay B Cohen. 2025. What can large language models capture about code functional equivalence? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6865–6903, Albuquerque, New Mexico.
- MikeMirzayanov. 2018. Codeforces: Problem difficulties. Codeforces Blog. Accessed: 2025-05-18.
- Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal Kumar Roy, and Kevin A. Schneider. 2019. Clcda: Cross language code clone detection using syntactical features and api documentation. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037.
- Ruchi Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir A. Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In *NeurIPS Datasets and Benchmarks*.
- Miguel Romero Rosas, Miguel Torres Sanchez, and Rudolf Eigenmann. 2024. Should ai optimize your code? a comparative study of current large language models versus classical optimizing compilers. *arXiv preprint arXiv:2406.12146*.
- Baptiste Roziere, Marie-Anne Lachaux, Lowik Chausot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611.
- Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–45.
- Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.
- Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.
- Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yaofeng Sun, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, and 1 others. 2025. Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking. *arXiv preprint arXiv:2502.12466*.
- Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P Xing, and 1 others. 2023. Judging LLM-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 46595–46623.
- Terry Yue Zhuo. 2024. Ice-score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2232–2242.

A Experimental Details

To keep our results reproducible, we use these hyperparameters of LLMs in our experiments: We set sampling to False, temperature to 0. Hence, we report the final results using a single run of the inference experiments.

For finetuning, we used LoRA finetuning using A100 GPU. We finetuned for nearly 48 GPU hours and selected the model with the least validation loss. The training and validation datasets were generated by the same algorithm 1. We initially split the 900-level problems in the CodeContests dataset into non-overlapping sets of 8, 8 and 40 problems, respectively, for test, validation and train sets. We then use the correct and incorrect submissions to these sets of problems to get the solution clusters for sampling. We used 2000 examples of each of the 8 categories and their original pairs for training. And we used 200 examples of each of the categories in the validation set. We experimented with learning rates from $1e - 3$ to $1e - 6$, and experimented with different combinations of $[q_proj, k_proj, v_proj, o_proj]$ layers for LoRA.

We made use of the `trl`⁵ library for SFTTrainer for finetuning and `ast`⁶ and `astor`⁷ libraries for dataset perturbation. We would be publicly releasing the code, which includes the setting of seeds for any randomness used, along with setting the final seeds and hyperparameters used for reproducing the experiments.

The zero-shot prompt used in our experiments is shared in the following page. We also experimented with various few-shot prompting methods, but the performance of models degraded. We hypothesize that this is caused by a larger context that involves multiple long pieces of code pairs. In Table 33, we present the results of few shot prompt with 4 examples each of positive and negative labels given along with the instruction of zero-shot prompt. It is observed that models get bias towards positive label.

A.1 Cross-Lingual Dataset Generation

The cross-lingual setting differs from the procedure in Section 3 mainly in how we filter the original program clusters. Our source corpus, CodeContests, contains submissions to problems in multiple

languages, along with a language label for each code snippet. Using these labels, we filter the solutions to retain only those written in the selected languages. This yields a collection \mathcal{P} containing the two languages used in our experiments, e.g., Java and Python for Tables 8 and 9.

In Algorithm 1, we additionally ensure at step 1 that the sampled code pair from \mathcal{P} contains programs written in different languages, i.e., the languages of s_1 and s_2 are distinct. For the single-code perturbation setting, we skip steps 10 and 13 in Algorithm 1, and designate the Python program in the selected pair as \hat{s} , i.e., the code marked for perturbation. For the both-code perturbation setting, we keep the randomization unchanged.

A.2 Data Statistics

Original Category	Sample Count	Perturbed Category	Sample Count
id	400	$\langle id, k, np \rangle$	200
		$\langle id, k, sp \rangle$	200
fe	2000	$\langle fe, k, np \rangle$	1000
		$\langle fe, k, sp \rangle$	1000
ne	2000	$\langle ne, k, np \rangle$	1000
		$\langle ne, k, sp \rangle$	1000
di	400	$\langle di, k, np \rangle$	200
		$\langle di, k, sp \rangle$	200

Table 11: Sample Counts of original and perturbed program-pairs

⁵<https://github.com/huggingface/trl>, https://huggingface.co/docs/trl/en/sft_trainer

⁶<https://docs.python.org/3/library/ast.html>

⁷<https://astor.readthedocs.io/en/latest/>

Basic Prompt

Given two programs, you must check if they solve the same problem - that is, for any input, they must produce identical outputs.

Your task is to answer only with one word: Yes or No.

No explanation, no extra words, no formatting.

Program 1: {prog1}

Program 2: {prog2}

COT Prompt

You are given two programs. Your task is to analyze whether they solve the same problem — meaning, for any given input, both programs must always produce identical outputs.

Start by reasoning step-by-step through the logic and structure of both programs. You must write your reasoning steps.

After your analysis, you must end your response with exactly one of the following:

FINAL_ANSWER_IS_YES

FINAL_ANSWER_IS_NO

Only use this exact format for the final answer, and do not write anything after it.

Program 1: {prog1}

Program 2: {prog2}

```
n, m = map(int, input().split())
a = sorted(list(map(int, input().split())))
res = a[n-1] - a[0]
for i in range(m - n + 1):
    temp = a[i+n-1] - a[i]
    if temp < res: res = temp
print(res)
```

(a) Original Code

```
shfxn, hkdgf = map(int, input().split())
a = sorted(list(map(int, input().split())))
res = a[shfxn - 1] - a[0]
for i in range(hkdgf - shfxn + 1):
    temp = a[i + shfxn - 1] - a[i]
    if temp < res:
        res = temp
print(res)
```

(b) Perturbed Pair program -1

```
firig, ozsoi = map(int, input().split())
res = a[firig - 1] - a[0]
a = sorted(list(map(int, input().split())))
for i in range(ozsoi - firig + 1):
    temp = a[i + firig - 1] - a[i]
    if temp < res:
        res = temp
print(res)
```

(c) Perturbed Pair program -2

Figure 1: Example of perturbation on programs.

A.3 Detailed explanation of types of perturbation

Given below are the implementation details and examples of the various types of perturbations:

1. If-Else Swapping: *Semantic-preserving:* The if condition is logically negated, and the if/else blocks are interchanged, maintaining the original logic. *Non-preserving:* The if condition is logically negated, but blocks are *not* swapped.

Listing 1: If-else swapping examples

```
# Original
if a < b:
    print("A")
else:
    print("B")

# --- Semantic-preserving ---
if not (a < b):
    print("B")
else:
    print("A")

# --- Semantic-non-preserving ---
if not (a < b):
    print("A")
else:
    print("B")
```

2. For-While Swapping *Semantic-preserving:* Convert for loop to equivalent while. *Non-preserving:* Converted to a while loop with an intentional error, such as incorrect iterator initialization, a flawed loop termination condition, or an improper iterator update.

Listing 2: For-while swapping examples

```
# Original
for i in range(10):
    print(i)

# --- Semantic-preserving ---
i = 0
while i < 10:
    print(i)
    i += 1

# --- Semantic-non-preserving ---
i = 1
while i < 10:
    print(i)
    i += 1
```

3. If Condition Flipping: *Semantic-preserving:* The condition of an if statement is transformed to an equivalent form (e.g., $a < b$ to $\text{not}(a \geq b)$), leaving the body unchanged. *Non-preserving:* The if condition's relational operator is inverted (e.g., $a < b$ to $a \geq b$) or the condition negated.

Listing 3: If flipping examples

```
# Original
if a < b:
    print("bigger")

# --- Semantic-preserving ---
if not (a >= b):
    print("bigger")

# --- Semantic-non-preserving ---
if a >= b:
    print("bigger")
```

4. Variable Renaming: *Semantic-preserving:* One or more randomly selected variables are consistently renamed to new, unique identifiers (e.g., random five-letter strings) across all their occurrences within their scope. *Non-preserving:* Similar to above, but for at least one selected variable, one of its occurrences is renamed to a different identifier than its other occurrences, effectively breaking its semantic link.

Listing 4: Variable renaming examples

```
# Original
x = 10
y = 15
if x > y:
    print("bigger")

# --- Semantic-preserving ---
abcde = 10
hogav = 15
if abcde > hogav:
    print("bigger")

# --- Semantic-non-preserving ---
abcde = 10
hogav = 15
if noagd > hogav: # inconsistent
    renaming
    print("bigger")
```

5. Boolean Variable Inversion: *Semantic-preserving:* The initialization of a selected boolean variable is inverted (e.g., True to False). All subsequent usages of this variable in conditions are then modified to use its negation (e.g., $\text{if}(\text{Flag})$ becomes $\text{if}(\text{not Flag})$). *Non-preserving:* The initialization of a boolean variable is inverted, but its subsequent usages are not altered to reflect this change.

Listing 5: Boolean Variable Inversion examples

```
# Original
Flag = True
if Flag:
    print("Go")

# --- Semantic-preserving ---
Flag = False
```

```

if not Flag:
    print("Go")

# --- Semantic-non-preserving ---
Flag = False
if Flag:
    print("Go")

```

6. Statement Reordering: *Semantic-preserving:* Two data-independent statements (whose relative execution order doesn't affect program outcome) are swapped. *Non-preserving:* Two data-dependent statements are swapped, potentially causing errors like using variables before definition or with stale values.

Listing 6: Statement Swap examples

```

# Original
a = 5
b = 0
print(2 * a)

# --- Semantic-preserving ---
a = 5
print(2 * a)
b = 0

# --- Semantic-non-preserving ---
print(2 * a)
b = 0
a = 5

```

7. Expression Reformatting: *Semantic-preserving:* A sub-expression is computed and stored in a new temporary variable, which then replaces the sub-expression in the main expression, maintaining the original computation. *Non-preserving:* A temporary variable may be used, but the main expression is reconstructed incorrectly (e.g., wrong operands/operators, re-introducing original terms), altering the result.

Listing 7: Expression reformatting examples

```

# Original
ans = a + b - c

# --- Semantic-preserving ---
temp = a + b
ans = temp - c

# --- Semantic-non-preserving ---
temp = a + b
ans = temp + b - c

```

B Extended Analysis

As we perturb the dataset to generate equal halves of the four original categories into *sp* and *np*, the previously positive samples from *id* and *fe* categories now have half of the samples as positive and

the remaining half as negative. Due to this, in the final perturbed dataset, we have 75% negative samples and the remaining 25% as positive samples. We want the model to be equally good on both positives and negatives. Hence, we evaluate the performance on weighted accuracy, which takes the average of the average accuracy on positive samples and the same on negative samples.

B.1 Effect of Self-Consistency

To study the effect of inference-time decoding strategies, we evaluate self-consistency decoding (Wang et al., 2023), as shown in Table 12. For each sample, we generate 5 responses (temperature = 1) and select the final prediction via majority voting. Self-consistency does not yield a significant improvement in accuracy. We observe that model predictions are highly consistent across runs; for most samples, the same answer is produced in all 5 iterations. This suggests that the errors stem from inherent limitations of the model rather than variance in decoding, and are therefore not mitigated by repeated sampling.

B.2 Increasing Problem Difficulty Level

We also investigate how LLM model performance varies with increasing problem difficulty, using levels 900, 1200, 1600, 1900, and 2200, which represent a progression from simple, beginner-level problems to complex tasks requiring advanced algorithmic reasoning. These levels correspond to Codeforces problem ratings, a widely used metric in competitive programming. As shown in Fig 2a and Fig 2b, there is a drop in accuracy—both before and after-perturbation across difficulty levels. The drop, however, remains constant across difficulty levels, which suggests that whether the problem is basic or complex, introducing perturbations impacts the model's reasoning in a similar way. For GPT-4o, we observe fluctuations in performance. Further, we note that while the drop for GPT-4o in terms of accuracy is negative, it is primarily due to high performance on *ne* and *di* categories. Similarly, Fig 6a and Fig 6b show that F1 score metric too, fluctuations are observed for GPT-4o. For closer analysis, we refer to the corresponding graph in Fig 4a and Fig 4b, plotting weighted Acc vs difficulty level, where we see that the drop is always positive across difficulty levels, pointing to nuanced performance behaviour of highly complex models.

B.3 Effect of Individual Perturbations

We also analysed how different types of perturbations impact the model by comparing the performance drop (delta) before and after applying each perturbation. We chose 5 perturbation types for which dataset generation was the quickest. Fig 3a and Fig 3b show that logic-altering perturbations like boolean inversion, if condition flipping, and if-else swapping have a slightly higher delta, indicating that the model struggles to understand the transformed logic. In contrast, surface-level changes such as variable renaming and statement reordering are comparatively easier for the models to interpret, resulting in comparatively slightly lower delta values.

Similarly, Fig 5a and Fig 5b contain these plots for weighted accuracy metric, while Fig 7a and Fig 7b contain these plots for F1 metric.

Tables 16 through 19 contain detailed results of analysis experiments. Table 21 contains the results of Python-C++ pairs of cross-lingual programs.

B.4 Disclosing the information about the perturbations to models

We tried to prompt the model to check for equivalence, along with providing information on possible kinds of perturbations applied to the programs. We added the examples of both *np* and *sp* types of all 7 types of perturbations. The examples used are the same as Appendix A. Comparing the results in table 23 with table 2, we observe that adding this information hurts the performance of the GPT-4o model. It did not help the Qwen and the Deepseek-Coder models. It improves the performance of Mistral for categories with negative labels, but hurts its performance for categories with positive labels. Similarly, by comparing the results of table 32 to table 5, we observe that adding the information of perturbation to the Chain-of-Thought prompt hurts the performance of the Mistral and the Deepseek-Coder models. It doesn't improve GPT-4o's performance. It improves Qwen's performance for some of the negative categories, but the decrease in the positive categories and the *di* category hurts overall performance. This shows that overall, the models struggle on the perturbed set even after having additional information about the perturbations disclosed to them.

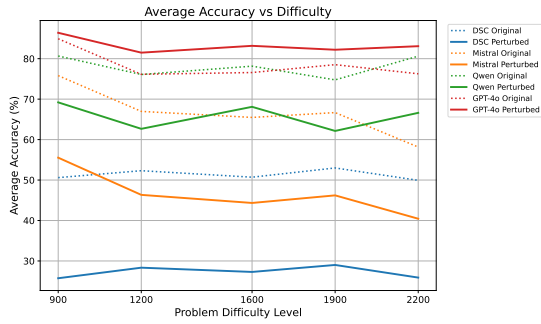
C Performance of Newer Models

Here we present the results of GPT-5-nano and GPT-5.1 on the remaining experiments. Table 24 shows performance using the Chain of Thought prompt. Table 25 shows the performance on the dataset with perturbations except variable renaming. And Table 8 shows performance on the cross-lingual task. Similar to our observation for GPT-4o in Sections 4.5.1, 4.5.3 and 4.7, we observe a drop in weighted average accuracy and positive-F1 score. Overall, we find these models to be biased towards labelling pairs with differing syntax as negative, as observed by low performance for the *fe* and $\langle fe, k, sp \rangle$ categories (even lower than open models for many cases).

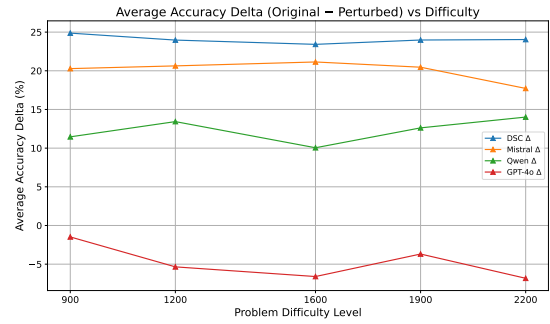
D Different Configurations of Training Data for Finetuning

D.1 Training on only unperturbed pairs

As observed in Table 27, clean performance is maintained, but robustness gains on perturbed inputs are modest when finetuning only on clean or unperturbed training data.

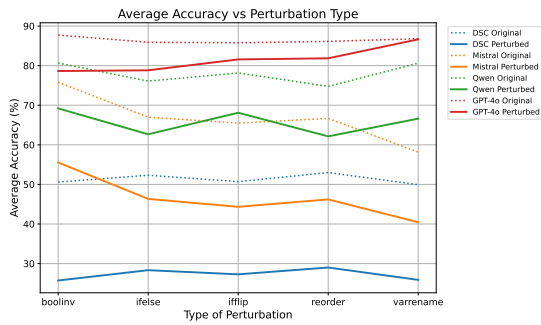


(a) Average Accuracy vs Problem Difficulty

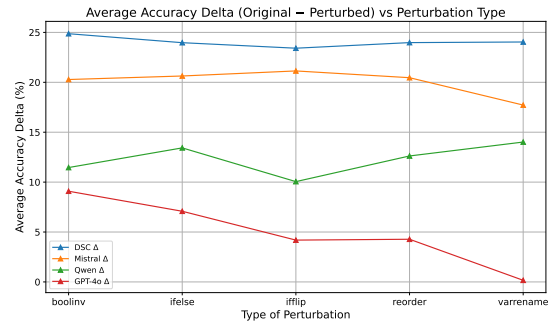


(b) Average Accuracy Delta vs Problem Difficulty

Figure 2: Change in Average Accuracy with varying Difficulty Level for different Models

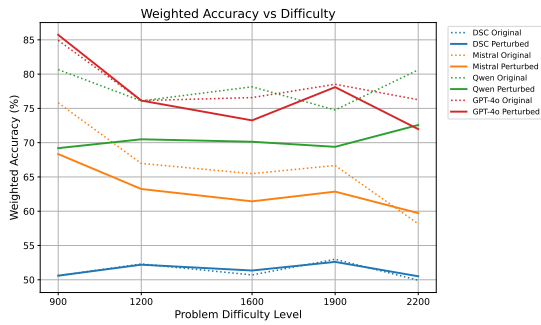


(a) Average Accuracy vs Perturbation Type

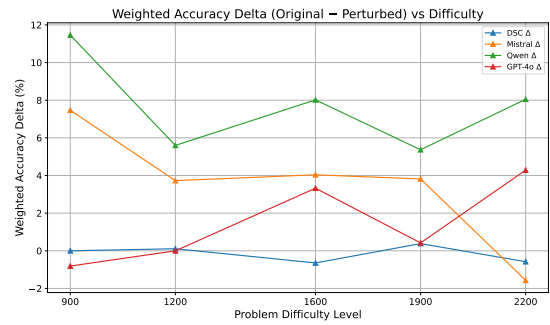


(b) Average Accuracy Delta vs Perturbation Type

Figure 3: Change in Average Accuracy with applying individual Perturbation Types for different Models

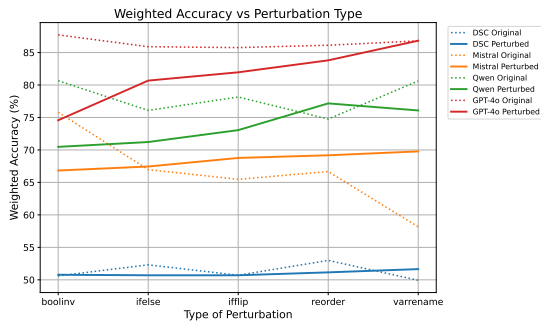


(a) Weighted Accuracy vs Problem Difficulty

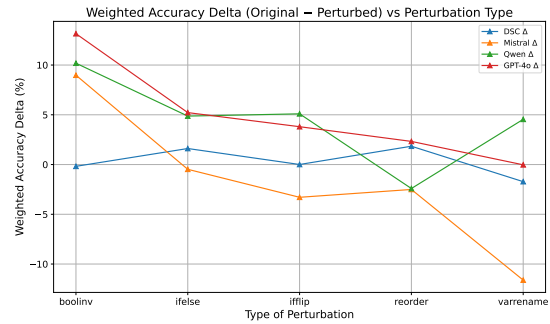


(b) Weighted Accuracy Delta vs Problem Difficulty

Figure 4: Change in Weighted Accuracy with varying Difficulty Level for different Models

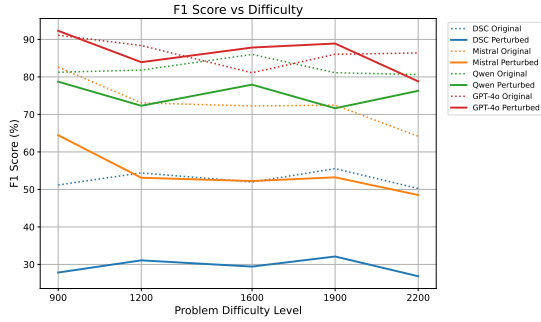


(a) Weighted Accuracy vs Perturbation Type

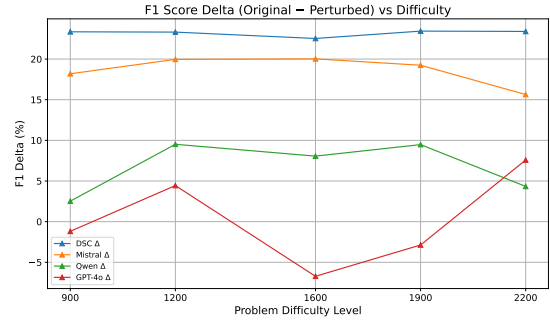


(b) Weighted Accuracy Delta vs Perturbation Type

Figure 5: Change in Weighted Accuracy with applying individual Perturbation Types for different Models

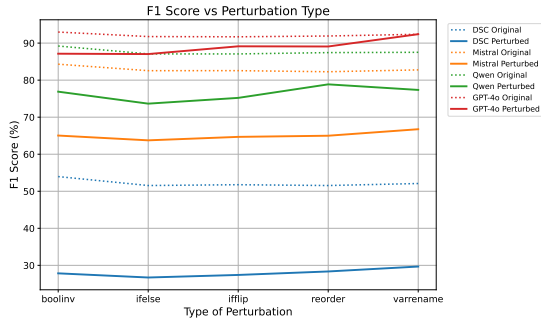


(a) F1 Score vs Problem Difficulty

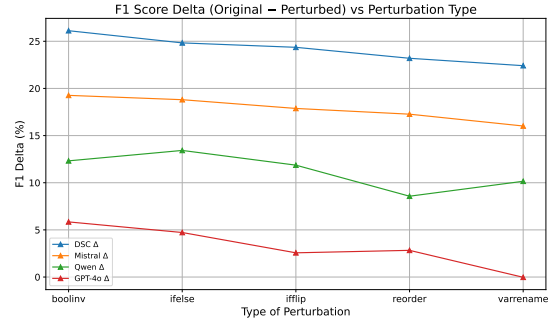


(b) F1 Score Delta vs Problem Difficulty

Figure 6: Change in F1 score with varying Difficulty Level for different Models



(a) F1 Score vs Perturbation Type



(b) F1 Score Delta vs Perturbation Type

Figure 7: Change in F1 score with applying individual Perturbation Types for different Models

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	99.0	82.5	82.0	39.7	38.5	100.0	100.0	80.2	80.2	86.6	69.4	72.0	66.8
Mistral	2.1	97.7	17.8	85.6	57.7	35.0	93.5	91.2	56.6	68.3	64.4	49.9	44.9	54.9
DSC	2.6	96.4	5.5	95.7	10.0	8.3	15.0	20.0	31.7	53.1	38.0	27.9	38.5	17.3

Table 12: Results comparing performance of various models using self-consistency decoding method.

D.2 Training on only perturbed pairs

As observed in Table 28, training exclusively on perturbed data improves robustness but leads to noticeable degradation on unperturbed inputs, particularly for Mistral.

E Extended Comparison with existing benchmarks

For comparison with existing benchmarks other than EquiBench (Wei et al., 2025) and Seq-CoBench (Maveli et al., 2025), we consider CodeNet (Puri et al., 2021) and CLCDSA (Nafi et al., 2019). Due to the absence of executable test cases in these datasets, we could not validate perturbed code. Instead, we evaluated the models on unperturbed samples corresponding to our *id*, *fe* and *di* categories (using the same sample counts as CETBench). Results for these are presented in Tables 29, 30 and 31. We observe that unperturbed pairs of other datasets are equally or less challenging than the unperturbed CodeContests pairs currently used in CETBench (and less challenging compared to perturbed pairs of CETBench). This highlights that CETBench exposes specific brittleness to simple transformations that other benchmarks do not capture. Also, DeepSeek-Coder (DSC) exhibits a strong bias towards predicting positive labels across all datasets (high accuracy on *id* or *fe*, near 0% on *ne* or *di*), confirming the trend observed in our main paper.

Prompt with Perturbation Information

Given two programs, you must check if they solve the same problem - that is, for any input, they must produce identical outputs.

Information regarding the programs: One or both of the programs may be modified from their original form using 7 different categories of perturbations.

Each of the 7 categories has 2 subtypes of perturbations: semantic preserving or not preserving. Given below are the examples for each of them:

1. If-else swapping examples:

```
# Original
if  $a < b$ :
print("A")
else:
print("B")
```

```
# Semantic-preserving
```

```
if not ( $a \geq b$ ):
print("B")
else:
print("A")
```

```
# Semantic-non-preserving
```

```
if not ( $a < b$ ):
print("A")
```

```
else:
print("B")
```

2. ...

...

...

The original pair of programs before perturbation may or may not be solving the same problem. Given all this information, now check if the final given pair solves the same problem or not.

Your task is to answer only with one word: Yes or No.

No explanation, no extra words, no formatting.

Program 1: {prog1}

Program 2: {prog2}

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
900	100.0	99.0	80.7	78.7	44.0	42.9	100.0	100.0	80.6	80.6	81.2	81.2	72.5	66.8
	30.5	98.0	39.7	76.9	61.4	47.6	99.5	100.0	69.2	69.2	78.7	78.7	69.2	78.7
1200	100.0	100.0	81.4	80.9	24.2	22.2	100.0	100.0	76.1	76.1	81.8	62.5	68.1	57.0
	26.0	95.0	32.5	77.3	42.1	28.4	100.0	100.0	62.6	70.5	72.3	53.2	45.0	61.5
1600	100.0	100.0	57.4	60.3	53.7	53.7	100.0	100.0	78.1	78.1	86.0	66.1	64.1	68.2
	21.0	92.5	53.4	55.9	64.9	57.1	100.0	100.0	68.1	70.1	77.9	59.0	44.4	73.6
1900	100.0	100.0	76.9	73.2	24.7	23.3	100.0	100.0	74.8	74.8	81.1	60.6	65.3	55.9
	30.5	98.0	39.7	76.9	61.4	47.6	99.5	100.0	62.1	69.4	71.6	52.8	43.1	62.4
2200	100.0	100.0	77.5	78.6	45.9	44.5	99.2	99.3	80.6	80.6	87.4	70.3	71.8	68.8
	19.0	96.0	39.1	73.0	59.6	51.1	96.5	98.6	66.6	72.6	76.3	59.3	48.4	70.2

Table 13: Effect of varying difficulty level on Qwen model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
900	100.0	100.0	90.3	90.3	25.2	24.7	86.5	89.5	76.3	76.3	82.6	65.1	71.7	58.4
	0.2	98.0	16.5	86.5	33.0	27.8	90.0	89.0	56.6	68.3	64.4	49.9	44.8	54.9
1200	100.0	100.0	95.5	96.5	10.7	8.6	61.0	63.5	66.9	66.9	73.1	53.5	68.1	38.8
	1.5	99.0	4.5	95.1	16.2	12.9	74.0	67.5	46.3	63.2	53.1	39.4	42.2	36.5
1600	98.9	98.4	96.1	96.0	10.8	11.1	58.5	54.0	65.5	65.5	72.3	52.6	67.5	37.7
	3.0	97.0	5.4	94.3	16.1	14.4	63.5	61.0	44.3	61.4	52.2	38.3	41.5	35.0
1900	99.5	99.5	96.5	94.8	8.0	7.6	64.5	63.0	66.7	66.7	72.5	52.5	67.5	37.5
	2.0	99.5	7.4	92.8	13.4	12.6	70.5	71.5	46.2	62.8	53.2	39.0	41.6	36.4
2200	99.5	100.0	99.5	99.3	13.0	13.0	19.7	21.9	58.2	58.2	64.1	46.4	66.7	26.2
	0.5	99.5	3.8	97.1	18.3	17.3	39.8	47.3	40.4	59.7	48.5	36.2	41.5	30.8

Table 14: Effect of varying difficulty level on Mistral model.

F Variance bounds

To analyse the variance in performance of models, we performed the following two experiments:

F.1 Variance on different original pairs, for the same set of programming problems

For this experiment, we generated four parallel sets of original and perturbed datasets. Here, we sampled the original pairs from the solution set of the same programming problems. We calculated the standard deviation in accuracies of the Qwen model on sets of original pairs, and as observed in table 34, we see negligible variance.

F.2 Variance on different perturbed pairs for the same original pairs

Here, we generated four parallel perturbed datasets by applying steps 2-17 of algorithm 1 four times on a sampled pair. We calculated the standard deviation in the accuracies of Qwen. We previously observed higher variance for pairs perturbed from *fe* and *ne* categories. Due to this, we increased their number of samples in the test set from 400 to 2000 in each category. The table 35 shows that the

final variance is negligible compared to the drop in performance from the original pairs.

G Error bounds

For calculating the error bounds for each type of perturbation, we calculated the percentage of erroneous samples generated over a total number of perturbation attempts. The results for the same are in Table 36. The low values of error percent denote the efficiency of our perturbation techniques.

H Error Analysis

Sample programs from the *fe* category are shown in Figures 8 and 9. The original pair correctly solves the same problem (“670-A Holidays” from Codeforces⁸). However, as illustrated in Figure 10, the Chain-of-Thought (CoT) reasoning generated by Mistral fails to recognize that the two programs are functionally equivalent, despite correctly identifying the underlying algorithms. This issue persists under both semantic-preserving (SP) and

⁸<https://codeforces.com/problemset/problem/670/A>

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
900	100.0	100.0	100.0	99.9	0.1	0.3	2.5	2.0	50.6	50.6	51.1	32.5	63.1	1.9
	0.0	99.5	0.0	99.9	1.2	0.3	2.5	2.5	26.3	50.6	27.8	20.2	37.6	2.8
1200	100.0	99.5	99.8	99.7	0.1	0.4	9.0	10.0	52.3	52.3	54.4	34.8	63.9	5.6
	1.0	100.0	1.2	99.9	1.0	0.6	12.5	10.5	28.3	52.2	31.1	22.1	38.2	5.9
1600	98.9	98.4	99.6	99.7	0.7	0.3	3.0	5.0	50.7	50.7	52.0	33.1	63.2	3.0
	0.5	99.5	0.8	99.5	1.5	0.5	10.5	5.5	27.3	51.4	29.4	21.2	37.9	4.5
1900	99.6	100.0	99.6	99.7	0.4	0.7	11.0	13.0	53.0	53.0	55.5	35.7	64.1	7.3
	0.0	100.0	1.0	99.6	0.8	0.7	15.5	14.5	29.0	52.6	32.1	22.6	38.3	7.0
2200	98.5	99.5	100.0	100.0	0.0	0.0	1.1	0.0	49.9	49.9	50.2	31.8	63.1	0.4
	1.0	99.5	0.5	100.0	0.6	0.1	2.8	2.7	25.9	50.5	26.8	19.7	37.7	1.8

Table 15: Effect of varying difficulty level on Deepseek-Coder model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
900	99.5	100.0	72.5	73.0	67.7	66.7	100.0	100.0	84.9	84.9	91.1	76.8	75.5	78.1
	81.0	97.0	77.4	71.8	91.8	72.2	100.0	100.0	86.4	85.7	92.3	77.3	66.66	88.1
1200	100.0	99.5	40.2	40.3	65.2	63.9	100.0	100.0	76.6	76.6	81.1	59.1	44.9	73.3
	78.5	92.5	86.2	38.4	87.7	68.6	100.0	100.0	81.4	76.1	88.4	66.0	47.7	84.4
1600	100.0	99.5	17.4	18.9	87.9	88.9	100.0	100.0	76.6	76.6	81.1	59.1	44.9	73.3
	76.5	88.0	96.4	18.8	97.1	90.8	100.0	100.0	83.2	73.2	87.8	64.2	40.7	87.6
1900	99.5	100.0	47.8	45.1	66.6	68.7	100.0	100.0	78.5	78.5	86.0	66.1	60.7	71.5
	71.0	98.0	81.6	41.8	92.5	72.8	100.0	100.0	82.2	78.1	88.9	68.2	51.1	85.3
2200	100.0	100.0	10.0	9.4	96.7	94.0	100.0	100.0	76.2	76.2	78.8	56.2	38.5	74.0
	72.0	91.0	97.4	8.5	99.4	96.4	100.0	100.0	83.1	72.0	86.4	60.8	33.5	88.0

Table 16: Effect of varying difficulty level on GPT-4o model.

semantic-non-preserving (NP) perturbations. Figure 11 shows that the model continues to misclassify equivalent programs after SP perturbations. Figure 12 demonstrates that the model fails to detect inequivalence after NP perturbations; although it correctly understands the original algorithms, it overlooks the semantic impact of the perturbation.

Tables 40, 37, 38, and 39 report how often model predictions in Chain-of-Thought (CoT) prompts change after perturbation. Each entry corresponds to the percentage of instances where the model’s final answer remains unchanged or flips when moving from the original to the perturbed pair. Highlighted values indicate cases where perturbations adversely affect model predictions. For example, values such as 18.5% (for “No” remaining unchanged in $\langle fe, k, np \rangle$ in Table 40) suggest that part of the measured performance may stem from incorrect reasoning on the original pairs themselves. Additionally, Table 39 reports the percentage of CoT generations from DeepSeek-Coder that omit reasoning and directly produce an answer. This behavior was not observed for other models.

We further analyze the errors through the follow-

ing questions:

Why does CoT hurt performance for GPT-4o?

To investigate this, we qualitatively analyze 50 randomly sampled program pairs where GPT-4o produces correct predictions without CoT but fails with CoT. (Figures 13, 17, 15 and 19 are few such samples.) We observe that for negative pairs generated via NP perturbations, CoT reasoning often fails to capture subtle semantic differences introduced by the perturbation, leading the model to incorrectly predict equivalence. (As observed in Figures 14 and 16.) Conversely, for positive pairs with SP perturbations, the model is misled by local structural changes and incorrectly predicts non-equivalence. (As observed in Figures 18 and 20.) These observations suggest that the model struggles to reason about the semantic impact of local perturbations. Additional reasoning steps introduced by CoT appear to amplify this confusion, resulting in degraded performance compared to direct prediction. Examining this further, and possibly connecting this analysis with how the LLM attends to the reasoning chain during COT, is an interesting

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	100.0	99.5	77.3	76.3	51.3	56.9	99.5	100.0	82.6	82.6	89.2	73.4	73.6	73.1
	30.5	81.0	28.9	75.2	58.0	59.7	100.0	100.0	66.7	70.5	76.9	59.3	48.3	70.4
ifelse	99.5	100.0	80.2	78.0	48.1	44.2	100.0	100.0	80.4	80.4	87.1	69.9	71.7	68.1
	23.2	92.5	24.9	78.4	46.1	47.6	100.0	100.0	64.1	71.2	73.6	55.8	46.8	64.9
iffliip	99.5	100.0	80.5	79.8	42.8	41.8	100.0	100.0	80.5	80.5	87.1	70.0	72.0	68.0
	30.3	97.0	25.2	78.5	48.9	45.8	100.0	100.0	65.7	73.0	75.2	56.5	47.5	65.6
reorder	100.0	100.0	82.2	83.2	42.6	41.5	100.0	100.0	81.2	81.2	87.4	70.9	73.2	68.6
	25.5	100.0	45.0	83.2	63.3	42.7	100.0	100.0	70.0	77.2	78.9	62.8	53.3	72.3
varrename	100.0	100.0	82.5	83.3	42.4	42.1	100.0	100.0	81.3	81.3	87.5	71.0	73.3	68.7
	20.0	100.0	37.4	81.9	63.6	46.3	100.0	100.0	68.6	76.1	77.3	61.7	52.2	71.2

Table 17: Effect of different perturbations on Qwen model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	99.5	99.0	81.9	82.5	36.7	38.7	84.5	88.0	76.4	76.4	84.3	66.6	70.4	62.7
	1.5	96.0	20.1	80.0	37.6	37.3	88.5	89.0	56.2	66.8	65.0	50.2	43.8	56.6
ifelse	99.4	100.0	88.1	88.3	25.6	27.7	90.0	89.5	76.0	76.0	82.6	65.0	71.7	58.3
	0.5	94.0	14.9	86.9	27.6	31.1	93.5	94.5	55.9	67.4	63.7	48.7	44.3	53.1
iffliip	99.5	100.0	87.9	89.0	27.7	25.1	88.5	91.5	76.1	76.1	82.6	64.6	70.8	58.3
	30.3	97.0	25.2	78.5	48.9	45.8	100.0	100.0	56.8	68.8	64.7	48.8	44.6	53.0
reorder	100.0	100.0	88.8	87.8	24.8	26.7	89.0	89.5	75.8	75.8	82.3	64.1	70.6	57.6
	25.5	100.0	45.0	83.2	63.3	42.7	100.0	100.0	57.4	69.2	65.0	51.0	45.6	56.4
varrename	100.0	100.0	88.6	90.0	24.1	28.6	92.0	89.0	76.5	76.5	82.8	65.0	71.3	58.7
	0.4	100.0	22.0	84.5	37.2	35.1	94.5	91.0	58.5	69.8	66.7	51.9	45.9	57.9

Table 18: Effect of different perturbations on Mistral model.

direction for future work.

Why do some models have low performance before perturbation?

We analyze model predictions for the *ne* category and observe that accuracy below 50% on original pairs indicates a strong bias toward the positive label. In this category, the two programs correspond to different implementations of the same problem, often exhibiting high structural similarity, with one program containing a subtle error. Due to this structural similarity, models tend to predict equivalence based on surface-level patterns rather than correctly identifying semantic differences. This behavior is consistent with our broader findings that LLMs rely more on syntactic cues than deep semantic reasoning. After introducing perturbations, this bias is partially reduced, leading to improved performance in the *ne* category. This improvement likely arises because perturbations disrupt superficial structural similarity, making it easier for the model to distinguish between equivalent and non-equivalent programs.

I Licence Details and Ethics

Please refer table 41 for details of Licences. The data used from CodeContests dataset contains code submissions of various users on codeforces platform. The userid or name of the owner of the submissions are not present in the dataset.

<pre> P1: a = int(input()) z = a//7 x = a%7 if x ==0:print(z*2,z*2) elif x<3:print(z*2,(z*2)+x) elif x>5:print((z*2)+1,(z*2)+2) else:print(z*2,(z*2)+2) </pre> <p>(a) Original Pair program -1</p>	<pre> P1: a = int(input()) z = a // 7 x = a % 7 if not (x != 0): print(z * 2, z * 2) elif x < 3: print(z * 2, z * 2 + x) elif x <= 5: print(z * 2, z * 2 + 2) else: print(z * 2 + 1, z * 2 + 2) </pre> <p>(b) SP Perturbed Pair program -1</p>	<pre> P1: a = int(input()) z = a // 7 x = a % 7 if x == 0: print(z * 2, z * 2) elif x >= 3: print(z * 2, z * 2 + x) elif x <= 5: print(z * 2, z * 2 + 2) else: print(z * 2 + 1, z * 2 + 2) </pre> <p>(c) NP Perturbed Pair program -1</p>
---	---	--

Figure 8: Sample pairs (Program -1).

<pre> P2: def p1(): n = int(input()) for i in range(0,n,1234567): for j in range(0,n-i,123456): if (n-i-j)%1234 == 0: return 'YES' else: return 'NO' def p2(): n,m = [int(i) for i in input().split()] l = [int(i) for i in input().split()] for i in range(n): if m-i-1 < 0 : return l[m-1] m = m-i-1 def p3(): n = int(input()) b = n//7*2+min(2,n%7) a = n//7*2 if n == n//7*7+6: a = a+1 print(a,b) p3() </pre> <p>(a) Original Pair program -2</p>	<pre> P2: def p1(): n = int(input()) for i in range(0,n,1234567): for j in range(0,n-i,123456): if (n-i-j)%1234 == 0: return 'YES' else: return 'NO' def p2(): n,m = [int(i) for i in input().split()] l = [int(i) for i in input().split()] for i in range(n): if m-i-1 < 0 : return l[m-1] m = m-i-1 def p3(): n = int(input()) b = n//7*2+min(2,n%7) a = n//7*2 if n == n//7*7+6: a = a+1 print(a,b) p3() </pre> <p>(b) SP Perturbed Pair program -2</p>	<pre> P2: def p1(): n = int(input()) for i in range(0,n,1234567): for j in range(0,n-i,123456): if (n-i-j)%1234 == 0: return 'YES' else: return 'NO' def p2(): n,m = [int(i) for i in input().split()] l = [int(i) for i in input().split()] for i in range(n): if m-i-1 < 0 : return l[m-1] m = m-i-1 def p3(): n = int(input()) b = n//7*2+min(2,n%7) a = n//7*2 if n == n//7*7+6: a = a+1 print(a,b) p3() </pre> <p>(c) NP Perturbed Pair program -2</p>
---	---	---

Figure 9: Sample pairs (Program -2).

LLM Output:

```

<> Reasoning:

Program 1:
- The program takes an integer input 'a'.
- It calculates the quotient 'z' of 'a' divided by 7 and the remainder 'x' of 'a' divided by 7.
- Depending on the value of 'x', it prints two numbers.
- If 'x' is 0, it prints the square of 'z'.
- If 'x' is between 1 and 2 (inclusive), it prints the square of 'z' plus 'x'.
- If 'x' is between 4 and 6 (inclusive), it prints the square of 'z' plus 1 or 2, depending on the value of 'x'.
- If 'x' is 3 or 5, it prints the square of 'z' plus 2.

Program 2:
- The program defines three functions: p1, p2, and p3.
- Function p1 takes an integer input 'n'.
- It iterates through all possible values of 'i' from 0 to 'n' with a step of 1234567.
- For each 'i', it iterates through all possible values of 'j' from 0 to 'n-i' with a step of 123456.
- If the difference between 'n' and the sum of 'i' and 'j' is a multiple of 1234, it returns 'YES'.
- If no such 'i' and 'j' are found, it returns 'NO'.
- Function p2 takes two integers 'n' and 'm' as input.
- It iterates through all possible values of 'i' from 0 to 'n'.
- If 'm-i-1' is less than 0, it returns the value of 'l[m-1]'.
- If 'm-i-1' is not less than 0, it sets 'm' to 'm-i-1'.
- Function p3 takes an integer input 'n'.
- It calculates 'b' as the product of the quotient of 'n' divided by 7 plus 2, and the minimum of 2 and the remainder of 'n' divided by 7.
- It calculates 'a' as the product of the quotient of 'n' divided by 7 plus 2.
- If 'n' is equal to the product of the quotient of 'n' divided by 7 plus 7 and 6, it increments 'a' by 1.
- It prints 'a' and 'b'.
- The last line calls function p3 and prints the result.

Final_Answer_Is_No

The two programs do not solve the same problem. Program 1 calculates two numbers based on the remainder of an input number divided by 7, while program 2 checks if a number can be expressed as the sum of two numbers with specific conditions on the step sizes of the iterations. Program 2 also defines and uses additional functions that are not used in program 1.</s>

```

Figure 10: Sample generation of Mistral on original pair

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	100.0	99.5	99.7	99.3	1.9	1.3	5.0	9.0	52.0	52.0	54.0	34.9	63.7	6.1
	0.0	99.5	0.2	99.6	1.1	0.8	5.5	4.5	26.4	50.8	27.8	20.4	37.7	3.0
ifelse	99.5	100.0	99.7	99.8	0.6	0.5	3.0	2.5	50.7	50.7	51.5	32.8	63.2	2.3
	1.0	100.0	0.2	99.7	0.6	0.3	3.5	1.5	25.8	50.5	26.7	19.6	37.5	1.6
iffliip	99.5	100.0	99.8	99.9	0.6	0.4	2.5	4.0	50.8	50.8	51.8	32.9	63.3	2.5
	0.0	99.5	0.1	99.9	0.7	0.4	4.5	4.5	26.2	50.7	27.4	20.0	37.7	2.4
reorder	100.0	100.0	99.9	99.9	0.5	0.4	3.5	2.0	50.8	50.8	51.5	32.8	63.4	2.2
	0.0	100.0	0.1	99.9	2.2	0.4	7.5	3.0	26.7	51.1	28.3	20.9	37.9	3.8
varrename	100.0	100.0	99.8	100.0	0.8	0.4	4.0	3.5	51.1	51.1	52.1	33.2	63.5	2.9
	0.5	100.0	0.7	99.9	1.1	0.9	11.0	6.0	27.5	51.6	29.7	21.4	38.0	4.7

Table 19: Effect of different perturbations on Deepseek-Coder model.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
boolinv	100.0	100.0	72.4	71.8	80.4	77.2	100.0	100.0	87.7	87.7	93.0	80.9	78.9	82.9
	79.0	61.0	52.5	71.9	86.2	78.5	100.0	100.0	78.6	74.6	87.1	70.7	57.8	83.6
ifelse	100.0	100.0	75.9	75.6	66.1	69.6	100.0	100.0	85.9	85.9	91.7	78.2	77.2	79.3
	60.1	95.0	49.3	73.8	79.5	72.8	100.0	100.0	78.8	80.7	87.0	69.8	58.1	81.5
iffliip	100.0	100.0	73.8	72.6	70.1	70.1	99.5	100.0	85.8	85.8	91.7	78.0	76.6	79.5
	60.1	95.0	49.3	73.8	79.5	72.8	100.0	100.0	78.8	80.7	87.0	69.8	58.1	81.5
reorder	100.0	100.0	75.3	76.8	69.1	67.8	100.0	100.0	86.1	86.1	91.9	78.6	77.6	79.6
	71.0	98.0	81.6	41.8	92.5	72.8	100.0	100.0	81.8	83.8	89.1	74.5	63.8	85.3
varrename	100.0	100.0	75.6	77.3	72.0	69.5	100.0	100.0	86.8	86.8	92.4	79.6	78.5	80.8
	79.0	100.0	75.5	74.4	93.0	71.7	100.0	99.5	86.6	86.8	92.4	78.0	67.8	88.2

Table 20: Effect of different perturbations on GPT-4o model.

Model	fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp						
Qwen	81.6	82.6	37.4	38.0	100.0	100.0	73.3	74.8	81.6	67.6	70.1	65.0
	52.5	74.9	70.9	47.8	100.0	100.0	74.3	72.2	83.7	51.6	75.6	63.6
Mistral	96.3	96.7	11.1	13.0	66.5	70.0	58.9	68.1	66.9	55.8	68.8	42.9
	0.8	95.6	17.9	14.5	76.5	77.0	48.2	64.6	57.0	41.9	43.2	40.6
DSC	99.9	99.6	0.9	0.6	3.0	4.0	34.7	50.9	36.0	33.2	63.4	3.0
	0.9	99.7	2.8	1.0	0.8	1.5	19.0	51.1	21.1	21.1	37.9	4.2
GPT-4o	62.0	59.0	80.5	79.2	100.0	100.0	80.0	74.4	88.0	73.1	68.2	78.0
	82.6	57.8	95.6	80.5	100.0	100.0	86.1	73.2	91.8	73.7	59.2	88.2

Table 21: Results comparing various models for C++ and Python pairs of code

Model	fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp						
Qwen	79.3	83.1	44.1	50.2	100.0	100.0	76.1	77.4	84.6	71.1	71.8	70.3
	35.2	82.2	58.7	50.5	100.0	100.0	71.1	75.5	80.6	61.3	50.2	72.4
Mistral	94.1	96.1	10.9	11.8	82.0	88.0	63.7	71.6	69.9	58.5	69.0	48.0
	9.7	94.3	15.9	14.4	79.8	81.3	49.2	67.3	57.6	42.8	41.8	43.8
DSC	100.0	99.9	0.1	0.0	8.2	12.1	36.7	52.5	39.4	34.5	63.2	5.8
	99.9	0.1	0.9	0.3	13.1	8.9	20.5	52.3	23.7	21.4	37.8	4.9
GPT-4o	55.8	58.1	81.8	80.4	100.0	100.0	79.4	73.8	87.4	71.8	65.6	78.1
	85.0	56.1	96.7	82.4	100.0	100.0	86.7	74.5	92.1	74.1	59.2	89.0

Table 22: Results comparing various models for pairs with one C++ code and one Python code (Both Perturbed)

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	99.0	78.8	78.4	44.0	41.5	100.0	100.0	80.2	80.2	86.9	69.6	71.4	67.7
	13.5	99.0	38.1	74.4	60.0	47.8	100.0	100.0	66.6	73.3	75.4	59.4	48.8	69.9
Mistral	100.0	99.0	68.4	70.4	50.7	47.9	90.0	93.5	77.5	77.5	85.8	67.4	67.8	67.0
	15.5	86.5	46.5	58.8	65.6	60.7	95.5	98.5	66.0	68.1	76.1	58.6	44.4	72.7
DSC	100.0	98.5	99.8	99.6	0.4	0.3	0.5	0.5	49.9	50.0	50.3	31.9	63.1	0.8
	0.0	99.5	0.2	99.8	0.1	0.3	0.0	0.5	25.0	49.9	25.2	18.9	37.5	0.4
GPT-4o	100.0	100.0	79.9	79.3	54.3	56.0	100.0	100.0	83.7	83.7	89.9	74.9	75.3	74.5
	58.0	98.0	62.4	73.8	82.5	61.4	100.0	100.0	79.5	81.6	87.6	70.7	59.2	82.2

Table 23: Results of performance after disclosing information about perturbation to the prompts.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
GPT-5-nano	100.0	100.0	58.3	64.7	91.1	90.5	100.0	100.0	88.1	88.1	92.8	81.0	77.4	84.6
	90.0	98.5	85.9	62.3	97.0	90.5	100.0	100.0	90.5	87.1	94.6	81.4	71.1	91.8
GPT-5.1	100.0	100.0	61.2	63.9	93.8	93.7	100.0	100.0	89.1	89.1	93.4	82.5	79.1	86.0
	93.0	99.5	91.8	62.5	98.8	93.8	100.0	100.0	92.4	88.6	95.6	84.2	75.1	93.3

Table 24: Results comparing GPT-5 models using COT prompt

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
GPT-5-nano	99.0	100.0	75.0	74.5	94.0	93.5	100.0	100.0	92.0	92.0	95.5	93.2	91.6	94.8
	89.5	98.5	85.0	69.5	97.5	95.0	100.0	100.0	91.9	89.2	95.5	90.0	83.8	95.9
GPT-5.1	100.0	99.5	62.5	54.0	92.0	89.5	100.0	100.0	87.2	87.2	92.2	89.0	86.0	91.9
	90.5	91.5	87.0	53.0	99.5	92.5	100.0	100.0	89.2	83.6	93.6	85.9	77.1	94.7

Table 25: Results of GPT-5 models on combinations of all other perturbations except variable-renaming

Model	fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp						
GPT-5-nano	73.8	74.2	89.5	90.7	100.0	100.0	88.0	84.5	93.3	83.8	81.2	86.4
	79.4	72.2	96.1	91.1	100.0	100.0	89.8	82.8	94.3	81.1	71.0	91.1
GPT-5.1	51.1	49.7	91.4	92.6	100.0	100.0	80.8	73.2	87.6	72.0	64.1	79.6
	93.2	49.0	98.7	92.4	100.0	100.0	89.0	72.9	92.9	75.4	60.0	90.7

Table 26: Results comparing GPT-5 models for pairs with one Java code and one Python code

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	100.0	100.0	80.5	82.2	45.7	44.8	100.0	99.0	81.5	81.5	87.9	71.6	73.4	69.8
	76.0	96.5	68.8	80.6	57.2	45.6	99.5	99.5	78.0	78.0	86.3	70.4	70.5	70.4
Mistral	100.0	98.5	56.2	57.1	56.2	55.0	94.0	94.0	76.4	76.4	85.1	65.1	62.9	67.4
	76.5	81.5	51.4	51.6	60.1	61.3	94.0	94.5	71.4	71.4	82.2	63.2	58.8	67.6
DSC	100.0	99.0	99.9	99.9	1.4	2.7	96.5	93.0	74.0	74.0	75.2	57.4	70.4	44.4
	0.0	99.0	0.1	99.9	1.8	2.6	97.0	92.5	49.1	65.9	50.4	38.1	42.8	33.5

Table 27: Results comparing performance of models trained only using unperturbed dataset.

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
	96.5	97.5	77.1	77.2	47.7	46.6	100.0	99.5	80.3	80.3	87.4	70.6	71.6	69.5
Qwen	87.0	93.0	80.1	84.2	85.8	41.5	100.0	98.0	83.7	85.3	90.0	74.4	64.5	84.3
	44.5	42.0	66.9	68.7	44.0	42.8	86.5	87.0	60.3	60.3	73.6	59.4	59.2	59.7
Mistral	76.0	70.5	60.0	73.2	65.1	35.4	93.5	80.0	69.2	70.1	80.7	60.3	48.2	72.3
	89.5	82.6	85.8	85.5	35.2	34.8	97.5	98.0	76.1	76.1	83.9	67.6	71.1	64.1
DSC	91.1	83.2	81.2	88.3	85.9	34.4	100.0	97.5	82.7	83.7	89.0	74.0	64.3	83.7

Table 28: Results comparing performance of models trained only using perturbed dataset.

Model	id	fe	di
Qwen	100.0	83.0	99.8
Mistral	100.0	87.8	96.0
DSC	100.0	100.0	2.0

Table 29: Results on CLCDSA – A (Easy level problems) dataset

Model	id	fe	di
Qwen	100.0	83.4	99.0
Mistral	100.0	92.3	69.0
DSC	99.0	99.2	8.0

Table 30: Results on CLCDSA – D (Difficult level problems) dataset

Model	id	fe	di
Qwen	100.0	81.2	100.0
Mistral	100.0	90.8	83.8
DSC	100.0	100.0	7.5

Table 31: Results on CodeNet dataset

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
Qwen	33.5	79.0	56.6	37.8	68.7	68.1	85.8	91.5	65.1	62.9	77.0	54.0	34.4	73.7
Mistral	5.5	90.5	14.7	85.4	28.7	24.9	76.0	76.0	50.2	63.0	60.0	44.8	42.0	47.6
DSC	25.5	44.0	14.3	36.9	18.9	17.5	33.5	43.5	29.3	33.0	44.1	25.0	19.1	30.9
GPT-4o	66.0	96.5	50.3	74.8	72.2	63.1	100.0	100.0	77.9	80.4	86.4	67.6	56.1	79.1

Table 32: Results on perturbed set after disclosing information about perturbation to the COT prompt

Model	id		fe		ne		di		Avg	W-Avg	Mac-F1	Mic-F1	PosF1	NegF1
	np	sp	np	sp	np	sp	np	sp						
	100.0	98.5	100.0	100.0	0.0	0.1	0.0	0.5	49.9	49.9	50.0	31.7	63.1	0.2
Qwen	8.5	98.5	6.7	100.0	8.0	0.1	6.0	0.5	28.5	52.1	31.8	23.8	38.6	9.1
	99.5	99.0	99.4	99.3	0.3	1.0	0.0	0.5	49.9	49.9	50.3	32.0	63.0	1.1
Mistral	0.0	99.5	0.2	99.3	0.5	1.1	1.0	1.0	25.3	50.0	25.9	19.4	37.4	1.3
	99.5	98.0	99.7	99.9	0.1	0.4	0.0	0.5	49.8	49.8	50.1	31.7	62.9	0.5
DSC	0.5	98.0	0.3	100.0	0.1	0.0	0.0	0.5	24.9	49.6	25.2	18.9	37.4	0.3

Table 33: Few shot prompt

Model	id	fe	ne	di
Unperturbed	0	4.77e-3	4.92e-2	0
NP Perturbed	4.02e-2	9.32e-2	5.66e-2	3.54e-3
SP Perturbed	6.86e-3	5.02e-2	6.78e-2	0

Table 34: Variance on changing original pairs

Model	id	fe	ne	di
NP Perturbed	8.92e-4	8.33e-4	4.35e-4	9.17e-5
SP Perturbed	3.22e-4	1.22e-4	1.21e-3	4.92e-4

Table 35: Variance on changing perturbed pairs

Perturbation type	error % in sp	error % in np
If-else swapping	3.86	11.68
For-while swapping	4.64	5.529
If condition flipping	0.12	4.37
Variable renaming	5.35	9.86
Boolean variable inversion	2.10	0.55
Statement reordering	1.93	8.99
Expression reformatting	2.88	3.33

Table 36: Error bounds in dataset generation

```
LLM Output:
<> Reasoning:
Program 1:
- The program takes an integer input 'a'.
- It calculates the quotient 'z' of 'a' divided by 7 and the remainder 'x' of 'a' divided by 7.
- If the remainder is not 0, it checks if the remainder is less than 3, less than or equal to 5, or greater than 5. Depending on the range, it calculates a different output.
- If the remainder is 0, it always calculates the same output, which is twice the quotient.

Program 2:
- The program defines three functions: p1, p2, and p3.
- Function p1 takes an integer input 'n' and checks if there exist two numbers 'i' and 'j' such that 'i' is a multiple of 1234567, 'j' is a multiple of 123456, and 'n' can be expressed as the sum of 'i', 'j', and a multiple of 1234. If such numbers exist, it returns 'YES'; otherwise, it returns 'NO'.
- Function p2 takes two integers 'n' and 'm' as input, as well as a list 'l'. It iterates through the list 'l' and returns the 'm'-th element of the list if 'm' is less than the length of the list; otherwise, it returns an element from the list.
- Function p3 takes an integer input 'n' and calculates the output 'b' as the product of the quotient of 'n' divided by 7 and 2, plus the minimum of 2 and the remainder of 'n' divided by 7. It calculates the output 'a' as the product of the quotient of 'n' divided by 7 and 2. If 'n' is a multiple of 49 (i.e., 'n' is a multiple of 7*7 plus 6), it increments 'a' by 1. It then prints 'a' and 'b'.

Since the logic and structure of both programs are different, they do not solve the same problem.

FINAL_ANSWER_IS_NO</s>
```

Figure 11: Sample generation of Mistral on SP perturbed pair

```
LLM Output:
<>
FINAL_ANSWER_IS_YES
Explanation:
Both programs take an integer n as input, which represents the number of names to be entered.

Program 1:
1. It initializes an empty dictionary named 'names' to store the count of each unique first letter of the names.
2. It then enters a loop that runs n times. In each iteration, it takes a name as input, extracts the first letter of the name, and increments the count of that first letter in the 'names' dictionary if it already exists, or initializes it to 1 if it doesn't.
3. After processing all the names, it initializes a counter 'count' to 0.
4. It then enters another loop that iterates over each key in the 'names' dictionary.
5. For each key, it checks if the count of that key is greater than or equal to 2. If so, it calculates the number of combinations of pairs of identical first letters in the names, depending on whether the count is even or odd.
6. It then adds the calculated number of combinations to the 'count' variable.
7. Finally, it prints the value of the 'count' variable.

Program 2:
1. It initializes an empty dictionary named 'd' to store the count of each unique first letter of the names.
2. It then initializes the alphabet list 'alpha' with the lowercase letters from 'a' to 'z'.
3. It enters a loop that runs n times. In each iteration, it takes a name as input, extracts the first letter of the name, and increments the count of that first letter in the 'd' dictionary if it already exists, or initializes it to 1 if it doesn't.
4. After processing all the names, it initializes a counter 'count' to 0.
5. It then enters another loop that iterates over each key in the 'alpha' list.
6. For each key, it calculates a list 'li' containing the number of pairs of identical first letters in the names that can be formed with the current key, considering both even and odd counts.
7. It then iterates over each element in the 'li' list and checks if the element is greater than or equal to 2. If so, it calculates the number of combinations of pairs of identical first letters in the names, depending on the value of the element.
8. It then adds the calculated number of combinations to the 'count' variable.
9. Finally, it prints the value of the 'count' variable.

Both programs perform the same operations on the same input data, so they will always produce identical outputs for the same input. Therefore, the final answer is YES.</s>
```

Figure 12: Sample generation of Mistral on NP perturbed pair

```

drghw = int(input())
qyxke = list(map(int, input().split()))
o = qyxke.count(1)
t = qyxke.count(2)
if o == 0 or t == 0:
    print(0)
else:
    a = qyxke[0]
    c = 0
    d = 0
    if not a != 1:
        c = c + 1
    elif a == 2:
        d = d + 1
    ans = 0
    t = 0
    for i in range(1, drghw):
        if qyxke[i] == qyxke[i - 1]:
            if qyxke[i] == 1:
                d = d + 1
            else:
                c = c + 1
        else:
            t = 1
            if qyxke[i] == 1:
                d = 1
            else:
                c = 1
        if t == 1:
            ans = max(ans, 2 * min(c, d))
    print(ans)

```

(a) NP Perturbed Pair program -1

```

n=int(input())
l=list(map(int,input().split()))
o=l.count(1)
t=l.count(2)
if o==0 or t==0:
    print(0)
else:
    a=l[0]
    c=0
    d=0
    if a==1:
        c=c+1
    elif a==2:
        d=d+1
    ans=0
    t=0
    for i in range(1,n):
        if l[i]==l[i-1]:
            if l[i]==1:
                c=c+1
            else:
                d=d+1
        else:
            t=1
            if l[i]==1:
                c=1
            else:
                d=1
        if t==1:
            ans=max(ans,2*(min(c,d)))
    print(ans)

```

(b) NP Perturbed Pair program -2

Figure 13: Error Analysis of GPT-4o's CoT: < id, k, np > pair - second if statement has the counters of c and d swapped

Qwen's Generation	id		fe		ne		di	
	np	sp	np	sp	np	sp	np	sp
Original: Yes, Perturbed: Yes	73.4	82.1	36.5	36.6	15.4	16.8	0.0	0.0
Original: Yes, Perturbed: No	26.0	14.3	31.1	25.7	29.2	24.4	0.5	1.5
Original: No, Perturbed: Yes	0.4	3.6	18.8	23.1	20.4	23.0	0.5	0.5
Original: No, Perturbed: No	0.0	0.0	13.6	14.6	35.0	35.8	98.9	98.0

Table 37: Analysis of percentage of answers changed or unchanged by Qwen on the program pair before and after perturbation.

```

To determine if the two programs solve the same problem and produce identical outputs for any given input, let's analyze them step-by-step.

### Program 1 Analysis:

1. Input Reading:
- `drghw = int(input())`: Reads an integer input, presumably the length of the list.
- `qyxke = list(map(int, input().split()))`: Reads a list of integers.

2. Counting 1s and 2s:
- `o = qyxke.count(1)`: Counts the number of 1s in the list.
- `t = qyxke.count(2)`: Counts the number of 2s in the list.

3. Initial Check:
- `if o == 0 or t == 0: print(0)`: If there are no 1s or no 2s, print 0 and exit.

4. Initialization:
- `a = qyxke[0]`: Takes the first element of the list.
- `c = 0, d = 0`: Initializes counters for consecutive 1s and 2s.

5. First Element Check:
- `if not a != 1: c = c + 1`: If the first element is 1, increment `c`.
- `elif a == 2: d = d + 1`: If the first element is 2, increment `d`.

6. Main Loop:
- `ans = 0, t = 0`: Initializes `ans` to store the result and `t` as a flag.
- Loop through the list starting from the second element:
  - If the current element is the same as the previous, increment the respective counter (`c` for 1s, `d` for 2s).
  - If the current element is different from the previous, set `t = 1` and reset the counter for the current element to 1.
  - If `t == 1`, update `ans` with `max(ans, 2 * min(c, d))`.

7. Output:
- `print(ans)`: Print the result.

### Program 2 Analysis:

1. Input Reading:
- `n=int(input())`: Reads an integer input, presumably the length of the list.
- `l=list(map(int,input().split()))`: Reads a list of integers.

2. Counting 1s and 2s:
- `o=l.count(1)`: Counts the number of 1s in the list.
- `t=l.count(2)`: Counts the number of 2s in the list.

3. Initial Check:
- `if o==0 or t==0: print(0)`: If there are no 1s or no 2s, print 0 and exit.

4. Initialization:
- `a=l[0]`: Takes the first element of the list.
- `c=0, d=0`: Initializes counters for consecutive 1s and 2s.

5. First Element Check:
- `if a==1: c=c+1`: If the first element is 1, increment `c`.
- `elif a==2: d=d+1`: If the first element is 2, increment `d`.

6. Main Loop:
- `ans=0, t=0`: Initializes `ans` to store the result and `t` as a flag.
- Loop through the list starting from the second element:
  - If the current element is the same as the previous, increment the respective counter (`c` for 1s, `d` for 2s).
  - If the current element is different from the previous, set `t = 1` and reset the counter for the current element to 1.
  - If `t == 1`, update `ans` with `max(ans, 2 * min(c, d))`.

7. Output:
- `print(ans)`: Print the result.

### Comparison:
- Both programs read an integer and a list of integers.
- Both programs count the number of 1s and 2s and check if either is zero, printing 0 if so.
- Both programs initialize counters for consecutive 1s and 2s and handle the first element similarly.
- Both programs loop through the list, updating counters and calculating the maximum possible value of `2 * min(c, d)` when a change between 1 and 2 is detected.
- Both programs print the same result.

The only difference is in the syntax used for checking the first element (`if not a != 1` vs. `if a == 1`), which is logically equivalent.

Therefore, both programs solve the same problem and produce identical outputs for any given input.

FINAL_ANSWER_IS_YES

```

Figure 14: Error Analysis of GPT-4o's CoT: < id, k, np > pair - The response misses the perturbation.

Mistral's Generation	id		fe		ne		di	
	np	sp	np	sp	np	sp	np	sp
Original: Yes, Perturbed: Yes	98.5	94.5	38.7	42.4	19.2	18.7	0.1	0.0
Original: Yes, Perturbed: No	1.5	4.5	18.6	16.3	16.3	15.6	3.0	2.6
Original: No, Perturbed: Yes	0.5	1.5	19.6	19.1	20.5	20.7	6.0	9.9
Original: No, Perturbed: No	0.0	0.5	22.9	22.2	43.9	45.0	90.0	87.4

Table 38: Analysis of percentage of answers changed or unchanged by Mistral on the program pair before and after perturbation.

```

n = int(input())
t = [int(i) for i in input().split()]
if n == 2:
    print(2)
    exit()
mas = []
count = 1
for i in range(n - 1):
    if not (t[i] != t[i + 1]):
        count += 1
    else:
        mas.append(count)
        count = 1
mas.append(count)
min_d = []
for i in range(len(mas) - 1):
    min_d.append(min(mas[i], mas[i + 1]))
print(max(min_d) * 2)

```

(a) NP Perturbed Pair program -1

```

if __name__ == '__main__':
    n = int(input())
    sushiRow = ''.join(['0' if i == '2' else i for i in input().split()])
    sushiRow = sushiRow if sushiRow[0] == '1' else bin(int(sushiRow, 2) ^ int('1' * n, 2))[2:]
    count, i, aux = 0, 0, 0
    maxlen = 0
    while i < n:
        while sushiRow[i] == '1':
            count = count + 1
            i = i + 1
            if i >= n:
                break
        maxlen = max(maxLen, 2 * min(aux, count))
        aux = count
        count = 0
        if i < n:
            break
        while sushiRow[i] == '0':
            count = count + 1
            i = i + 1
            if i >= n:
                break
        maxlen = max(maxLen, 2 * min(aux, count))
        aux = count
        count = 0
    print(maxLen)

```

(b) NP Perturbed Pair program -2

Figure 15: Error Analysis of GPT-4o's CoT: < fe, k, np > pair - the break statements in while loops in Program 2 make it exit after processing just 2 rows.

To determine if the two programs solve the same problem, we need to analyze their logic and see if they produce identical outputs for any given input.

Program 1 Analysis:

- Input Handling:**
 - The program reads an integer 'n' and a list of integers 't' of length 'n'.
- Special Case:**
 - If 'n == 2', the program immediately prints '2' and exits.
- Main Logic:**
 - The program initializes an empty list 'mas' and a counter 'count' set to '1'.
 - It iterates over the list 't' from index '0' to 'n-2'.
 - If 't[i] == t[i + 1]', it increments 'count'.
 - If 't[i] != t[i + 1]', it appends 'count' to 'mas' and resets 'count' to '1'.
 - After the loop, it appends the last 'count' to 'mas'.
- Finding Maximum:**
 - It initializes an empty list 'min_d'.
 - It iterates over 'mas' and appends 'min(mas[i], mas[i + 1])' to 'min_d'.
 - Finally, it prints 'max(min_d) * 2'.

Program 2 Analysis:

- Input Handling:**
 - The program reads an integer 'n' and a list of integers, which it converts to a string 'sushiRow' where '2' is replaced by '0'.
- Binary Conversion:**
 - If the first character of 'sushiRow' is not '1', it performs a bitwise XOR with a string of '1's of length 'n' and updates 'sushiRow'.
- Main Logic:**
 - It initializes 'count', 'i', 'aux', and 'maxLen' to '0'.
 - It uses a while loop to iterate over 'sushiRow'.
 - It counts consecutive '1's and updates 'maxLen' with '2 * min(aux, count)'.
 - It then counts consecutive '0's and updates 'maxLen' similarly.
 - The loop continues until the end of 'sushiRow'.
- Output:**
 - It prints 'maxLen'.

Comparison:

- Program 1 is counting consecutive equal numbers in the list 't' and then finding the maximum length of two consecutive segments of equal numbers, multiplied by 2.
- Program 2 is effectively doing the same but on a binary string representation of the input list, where '2' is treated as '0'. It counts consecutive '1's and '0's and finds the maximum length of two consecutive segments, multiplied by 2.

Both programs are essentially solving the same problem: finding the maximum length of two consecutive segments of equal numbers (on binary digits) and returning twice that length. The transformation in Program 2 (replacing '2' with '0') does not affect the logic since it consistently treats '2' as '0' throughout.

Therefore, both programs produce identical outputs for any given input.

FINAL_ANSWER_IS_YES

Figure 16: Error Analysis of GPT-4o's CoT: < fe, k, np > pair - The response incorrectly says that the loop continues until the end of 'sushiRow'.

```

a = [int(x) for x in input().split()]
b = [int(x) for x in input().split()]
sum = 0
for i in range(len(b)):
    sum = sum + b[i]
c = a[1] - sum
if sum + 10 * (len(b) - 1) <= a[1]:
    print(c // 5)
else:
    print(-1)

```

(a) SP Perturbed Pair program -1

```

a=[int(x) for x in input().split()]
b=[int(x) for x in input().split()]
sum=0
for i in range(len(b)):
    sum=sum+b[i]
c=a[1]-sum
if(sum + 10*(len(b)-1)>a[1]):
    print(-1)
else:
    print(c//5)

```

(b) SP Perturbed Pair program -2

Figure 17: Error Analysis of GPT-4o's CoT: < id, k, sp > pair - Semantically equivalent if statements

```

To determine if the two programs solve the same problem, we need to analyze their logic and structure step-by-step.

### Program 1 Analysis:

1. Input Reading:
- The program reads two lines of input.
- The first line is split into integers and stored in list `a`.
- The second line is split into integers and stored in list `b`.

2. Sum Calculation:
- A variable `sum` is initialized to 0.
- A loop iterates over the elements of list `b`, adding each element to `sum`.

3. Calculation of `c`:
- `c` is calculated as `a[1] - sum`.

4. Conditional Check and Output:
- The program checks if `sum + 10 * (len(b) - 1) > a[1]`.
- If true, it prints `-1`.
- Otherwise, it prints `c // 5`.

### Program 2 Analysis:

1. Input Reading:
- The program reads two lines of input.
- The first line is split into integers and stored in list `a`.
- The second line is split into integers and stored in list `b`.

2. Sum Calculation:
- A variable `sum` is initialized to 0.
- A loop iterates over the elements of list `b`, adding each element to `sum`.

3. Calculation of `c`:
- `c` is calculated as `a[1] - sum`.

4. Conditional Check and Output:
- The program checks if `sum + 10 * (len(b) - 1) <= a[1]`.
- If true, it prints `c // 5`.
- Otherwise, it prints `-1`.

### Comparison:

- Both programs read the input in the same way and calculate `sum` and `c` identically.
- The key difference lies in the conditional check:
  - Program 1 checks if `sum + 10 * (len(b) - 1) > a[1]` and prints `-1` if true, otherwise it prints `c // 5`.
  - Program 2 checks if `sum + 10 * (len(b) - 1) <= a[1]` and prints `c // 5` if true, otherwise it prints `-1`.

The conditions are complementary:
- In Program 1, the condition for printing `-1` is `sum + 10 * (len(b) - 1) > a[1]`.
- In Program 2, the condition for printing `c // 5` is `sum + 10 * (len(b) - 1) <= a[1]`.

This means that the two programs will produce different outputs for the same input because they have opposite conditions for the same output.

Therefore, the two programs do not solve the same problem.

FINAL_ANSWER_IS_NO"

```

Figure 18: Error Analysis of GPT-4o’s CoT: < id, k, sp > pair - The response misses the semantic equivalence of lines with different syntax.

DSC’s Generation	id		fe		ne		di	
	np	sp	np	sp	np	sp	np	sp
Original: Yes, Perturbed: Yes	83.0	92.0	73.3	78.5	59.9	63.5	2.5	3.5
Original: Yes, Perturbed: No	16.0	6.5	14.0	8.9	16.3	12.5	8.0	10.5
Original: No, Perturbed: Yes	0.5	1.5	7.5	6.7	9.6	10.5	5.5	6.5
Original: No, Perturbed: No	0.5	0.0	5.1	5.8	14.2	13.5	83.9	79.5
Missing Reason	44.5	40.5	7.3	6.9	5.9	5.1	0.5	0.5

Table 39: Analysis of percentage of answers changed or unchanged by Deepseek-Coder on the program pair before and after perturbation.

```

clas_1 = {"a": 0, "b": 0, "c": 0, "d": 0, "e": 0, "f": 0, "g": 0, "h": 0, "i": 0, "j": 0, "k": 0, "l": 0,
         "m": 0, "n": 0, "o": 0, "p": 0, "q": 0, "r": 0, "s": 0, "t": 0, "u": 0, "v": 0, "w": 0, "x": 0,
         "y": 0, "z": 0}
clas_2 = {"a": 0, "b": 0, "c": 0, "d": 0, "e": 0, "f": 0, "g": 0, "h": 0, "i": 0, "j": 0, "k": 0, "l": 0,
         "m": 0, "n": 0, "o": 0, "p": 0, "q": 0, "r": 0, "s": 0, "t": 0, "u": 0, "v": 0, "w": 0, "x": 0,
         "y": 0, "z": 0}
n = int(input())
for i in range(n):
    st = input().lower()
    if clas_1[st[0]] % 2 == clas_2[st[0]] % 2:
        clas_1[st[0]] += 1
    else:
        clas_2[st[0]] += 1
s = 0
for i in clas_1.values():
    if i > 1:
        s = s + (i * (i - 1)) // 2
for i in clas_2.values():
    if i > 1:
        s = s + (i * (i - 1)) // 2
print(s)

```

(a) SP Perturbed Pair program -1

```

from collections import defaultdict
import math

def fact(n):
    res = 1
    for i in range(2, n + 1):
        res = res * i
    return res

def ncr(n, r):
    return fact(n) / (fact(r) * fact(n - r))

n = int(input())
a = []
for i in range(n):
    x = input()
    a.append(x)
d = defaultdict(lambda : 0)
for i in range(n):
    z = a[i][0]
    d[z] += 1
count_1, count_2 = 0, 0
j = 0
for keys, values in d.items():
    if not values >= 2:
        continue
    s = values // 2
    rest = values - s
    if j % 2 == 0:
        if s > rest:
            if s > 1:
                count_1 += ncr(s, 2)
            if rest > 1:
                count_2 += ncr(rest, 2)
        else:
            if rest > 1:
                count_1 += ncr(rest, 2)
            if s > 1:
                count_2 += ncr(s, 2)
    elif s > rest:
        if s > 1:
            count_2 += ncr(s, 2)
        if rest > 1:
            count_1 += ncr(rest, 2)
    else:
        if rest > 1:
            count_2 += ncr(rest, 2)
        if s > 1:
            count_1 += ncr(s, 2)
print(int(count_1 + count_2))

```

(b) SP Perturbed Pair program -2

Figure 19: Error Analysis of GPT-4o's CoT: < fe, k, sp > pair - Two different programs performing the same operation.

GPT-4o's Generation	id		fe		ne		di	
	np	sp	np	sp	np	sp	np	sp
Original: Yes, Perturbed: Yes	34.5	99.0	47.4	66.4	21.3	27.9	0.0	0.0
Original: Yes, Perturbed: No	65.5	1.0	29.0	1.0	18.2	11.0	0.0	0.0
Original: No, Perturbed: Yes	0.0	0.0	5.1	8.2	4.7	5.7	0.0	0.0
Original: No, Perturbed: No	0.0	0.0	18.5	15.5	55.8	55.4	100.0	100.0

Table 40: Analysis of percentage of answers changed or unchanged by GPT-4o on the program pair before and after perturbation.

```

To determine if the two programs solve the same problem, let's analyze each program step-by-step.

### Program 1 Analysis:

1. **Initialization**:
   - Two dictionaries `clas_1` and `clas_2` are initialized with keys as lowercase letters ('a' to 'z') and values set to 0.

2. **Input Handling**:
   - An integer `n` is read from input, representing the number of strings to be processed.
   - A loop runs `n` times, each time reading a string `st`, converting it to lowercase, and checking the first character `st[0]`.

3. **Distribution Logic**:
   - For each string, the program checks if the parity (even or odd) of the count of the first character in `clas_1` is the same as in `clas_2`.
   - If the parities are the same, it increments the count in `clas_1`; otherwise, it increments the count in `clas_2`.

4. **Combinatorial Calculation**:
   - A variable `s` is initialized to 0.
   - For each value in `clas_1` and `clas_2`, if the value is greater than 1, it adds the number of ways to choose 2 items from that value to `s` using the formula `(i * (i - 1)) // 2`.
   - Finally, it prints `s`.

### Program 2 Analysis:

1. **Helper Functions**:
   - `fact(n)`: Computes the factorial of `n`.
   - `ncr(n, r)`: Computes the binomial coefficient (n choose r) using the factorial function.

2. **Input Handling**:
   - An integer `n` is read from input, representing the number of strings.
   - A list `a` is populated with `n` strings from input.

3. **Frequency Counting**:
   - A `defaultdict` `d` is used to count the occurrences of the first character of each string in `a`.

4. **Distribution and Combinatorial Calculation**:
   - Two counters `count_1` and `count_2` are initialized to 0.
   - For each character in `d`, if its count `values` is less than 2, it is skipped.
   - Otherwise, `values` is split into two groups: `s` (half of `values`) and `rest` (the remainder).
   - Depending on the parity of the loop index `j`, the program decides which group contributes to `count_1` and which to `count_2`.
   - It calculates combinations of 2 from each group using `ncr` and adds them to the respective counters.
   - Finally, it prints the sum of `count_1` and `count_2`.

### Comparison:

- **Distribution Logic**:
  - Program 1 uses parity checks to distribute counts between `clas_1` and `clas_2`.
  - Program 2 uses a round-robin approach based on the index `j` to distribute counts between `count_1` and `count_2`.

- **Combinatorial Calculation**:
  - Both programs calculate the number of ways to choose 2 items from each group of counts greater than 1.
  - Program 1 uses integer arithmetic, while Program 2 uses a factorial-based approach.

- **Output**:
  - Both programs ultimately sum the combinatorial counts and print the result.

### Conclusion:

The two programs implement different logic for distributing counts and calculating combinations, but they aim to solve the same problem: counting the number of ways to choose 2 items from groups of strings with the same starting character. However, due to the different distribution logic, they may not produce identical outputs for all inputs.

FINAL_ANSWER_IS_NO

```

Figure 20: Error Analysis of GPT-4o's CoT: < fe, k, sp > pair.

Model / Tool / Dataset	License
HuggingFace	Apache License, Version 2.0
DeepSeek-Coder	Code: MIT License; Model: DeepSeek License Agreement
Qwen	Tongyi Qianwen License Agreement
GPT-4o	Proprietary (OpenAI Terms of Use)
Mistral	Apache License, Version 2.0
CodeContests Dataset	CC BY 4.0 (Creative Commons Attribution 4.0)

Table 41: Licenses of the different models, tools, and datasets used in our experiments.