EquiBench: Benchmarking Code Reasoning Capabilities of Large Language Models via Equivalence Checking

Anonymous ACL submission

Abstract

Equivalence checking, i.e., determining whether two programs produce identical outputs for all possible inputs, underpins a broad range of applications, including software refactoring, testing, and optimization. We present the task of equivalence checking as a new way to evaluate the code reasoning abilities of large language models (LLMs). We introduce EquiBench, a dataset of 2400 program pairs spanning four programming languages and six equivalence categories. These pairs are systematically generated through program analysis, compiler scheduling, and superoptimization, covering nontrivial structural transformations that demand deep semantic reasoning beyond simple syntactic variations. Our evaluation of 17 state-of-the-art LLMs shows that OpenAI o3-mini achieves the highest overall accuracy of 78.0%. In the most challenging categories, the best accuracies are 62.3% and 68.8%, only modestly above the 50% random baseline for binary classification, indicating significant room for improvement in current models' code reasoning capabilities.

1 Introduction

Programming has emerged as a key application domain for large language models (LLMs), enabling tasks such as program synthesis (Chen et al., 2021; Austin et al., 2021; Jain et al., 2024), test generation (Yang et al., 2024a), bug detection (Yang et al., 2023), program repair (Xia et al., 2023), and code optimization (Shypula et al., 2023). Recently, there has been growing interest in evaluating how well LLMs can reason about the semantics of code (Ni et al., 2024; Liu et al., 2023; Gu et al., 2024; Chen et al., 2024a; Liu et al., 2024b), i.e., predicting program properties without running the program.

This paper introduces the task of **equivalence checking** as a new way to evaluate the code reasoning capabilities of LLMs. A classic challenge in programming languages and verification, equivalence checking involves determining whether two programs produce identical outputs for all possible inputs. Figure 1 presents examples of equivalent and inequivalent program pairs.

```
int main() {
  int a, b;
  ...
  if (a < b) {
    a = 1;
  }
  ...
}
```

```
int main() {
  int a, b;
  ...
  if (b > a) {
    a = 1;
  }
  ...
}
```

```
int main() {
  int a, b;
  ...
  if (b <= a) {
    a = 1;
  }
  ...
}
```

Equivalent Inequivalent

Figure 1: **An equivalent and an inequivalent program pair constructed using prior techniques.** Prior works generate such pairs through *basic statement-level syntactic modifications* with minimal semantic reasoning, whereas our approach, presented later, relies on structural program transformations that require much deeper semantic reasoning.

Compared to prior code reasoning tasks, evaluating LLMs using equivalence checking offers distinct advantages. Most notably, it presents a significantly more challenging benchmark than previous tasks, enabling a more rigorous assessment of LLMs' code reasoning capabilities. Equivalence checking requires LLMs to reason over *all possible inputs*, while prior work often focuses on *a single input*, such as output prediction, input prediction (Gu et al., 2024), input-specific program state prediction and execution simulation (Liu et al., 2023; Chen et al., 2024a; Ding et al., 2024; La Malfa et al., 2024; Ni et al., 2024).

Moveover, equivalence checking underpins a broad range of downstream applications, including software refactoring (Pailoor et al., 2024), software testing (Tian et al., 2024), and program optimization (Shypula et al., 2021), surpassing the scope of prior reasoning tasks. By requiring a deep understanding of program semantics and reasoning over all possible inputs, equivalence checking enables the analysis of an expressive range of program behaviors, even including many undecidable problems. Therefore, LLMs that perform well on equivalence checking are likely to be well-suited for tackling more complex programming tasks.

Our proposal requires a benchmark consisting of both equivalent and inequivalent program pairs covering different aspects of equivalence reasoning with varying degrees of difficulty. A large benchmark is essential, making it desirable to automate the benchmark generation process. Existing methods (Badihi et al., 2021; Maveli et al., 2024) mostly rely on *local syntactic changes* such as operand swaps (e.g., changing `a < b` to `b > a` for equivalent pairs or `b <= a` for inequivalent pairs; see Figure 1), which do not require deep semantic reasoning. However, these approaches are insufficient for benchmarking the equivalence reasoning capabilities of state-of-the-art LLMs. As many existing benchmarks have become saturated (Phan et al., 2025), a more challenging dataset is needed to rigorously assess LLMs' semantic reasoning abilities.

In this work, we introduce **EquiBench**, a new dataset of 2400 program pairs for equivalence reasoning. EquiBench spans four programming languages—Python, C, CUDA, and x86-64 assembly—providing a systematic benchmark to evaluate LLMs' code reasoning abilities.

The key technical challenge is to automatically generate (in)equivalent program pairs that demand *deep semantic reasoning beyond simple syntactic variations*. We propose several techniques to achieve this. First, to confirm that basic syntactic variations are well within the reasoning capabilities of state-of-the-art LLMs, we construct an equivalence category based on variable renaming, which barely requires semantic reasoning. Next, we generate equivalent programs by removing dead code, leveraging program analysis to go beyond trivial syntactic changes. By incorporating alias analysis and path feasibility analysis, we increase the difficulty of semantic reasoning in an automated manner. For GPU programs written in CUDA, we generate equivalent pairs by exploring different compiler scheduling strategies, such as loop tiling and shared memory caching, which involve structural transformations that extend far beyond statement-level modifications. We also use *superoptimization* to explore optimal instruction sequences beyond standard compiler optimizations, enabling more aggressive code restructuring. Finally, we include pairs with different algorithmic choices using submissions from online programming platforms.

Our experiments show that EquiBench is a challenging benchmark for LLM-based equivalence checking. Among the 17 models evaluated, OpenAI o3-mini performs best overall, yet achieves only 59.0% in the CUDA category despite achieving the highest overall accuracy of 78.0%. For the two most difficult categories, the best accuracy across all models is 62.3% and 68.8%, respectively. These numbers are only *modestly better than the random baseline*—i.e., 50% accuracy for binary classification. Further analysis shows that variable renaming, a purely syntactic modification, is the easiest equivalence category for models, with accuracy as high as 91.2%. We also find that models are *biased* toward classifying programs with significant structural, non-local transformations as inequivalent. Moreover, prompting strategies such as few-shot in-context learning and Chain-of-Thought (CoT) prompting *barely* enhance LLMs' semantic reasoning capabilities in equivalence checking, underscoring the fundamental difficulty of the task.

In summary, our contributions are as follows:

- **New Task and Dataset:** We introduce equivalence checking as a new task to assess LLMs' code reasoning capabilities. We present *EquiBench*, a benchmark for semantic equivalence checking spanning four languages and six equivalence categories.

- **Automated Generation:** We develop a fully automated pipeline to construct diverse (in)equivalent program pairs, using techniques from program analysis, compiler scheduling, and superoptimization. The pipeline covers transformations including syntactic changes, structural modifications, and algorithmic equivalence.

- **Evaluation and Analysis:** We evaluate 17 state-of-the-art models on EquiBench, with the highest overall accuracy reaching 78.0%. In the two most challenging categories, the best accuracy across all models is 62.3% and 68.8%, indicating significant room for improvement. Additionally, we analyze performance across different equivalence categories and prompting strategies.

2 Related Work

LLM Reasoning Extensive research has evaluated LLMs' reasoning capabilities across diverse tasks (Cobbe et al., 2021; Huang and Chang, 2022; Bubeck et al., 2023; Mirzadeh et al., 2024; Zhou et al., 2022; Ho et al., 2022; Wei et al., 2022; Chen et al., 2024b; Clark et al., 2018; Zhang et al., 2024). In the context of code reasoning, i.e., predicting a program's execution behavior without running it, CRUXEval (Gu et al., 2024) focuses on input-output prediction, while CodeMind (Liu et al., 2024b) extends evaluation to natural language specifications. Another line of work seeks to improve LLMs' code simulation abilities through prompting (La Malfa et al., 2024) or targeted training (Liu et al., 2023; Ni et al., 2024; Ding et al., 2024). Unlike prior work that evaluates LLMs on predicting program behavior for a specific input, our new code reasoning task and benchmark for equivalence checking assesses LLMs' ability to reason about all possible inputs.

Equivalence Checking Equivalence checking underpins applications such as performance optimization (Shypula et al., 2023; Cummins et al., 2023, 2024), code transpilation (Lu et al., 2021; Yang et al., 2024b; Ibrahimzada et al., 2024; Pan et al., 2024), refactoring (Pailoor et al., 2024), and testing (Felsing et al., 2014; Tian et al., 2024). Due to its undecidable nature, no algorithm can decide program equivalence for all program pairs while always terminating. Existing techniques (Sharma et al., 2013; Dahiya and Bansal, 2017; Gupta et al., 2018; Mora et al., 2018; Churchill et al., 2019; Badihi et al., 2020) focus on specific domains, such as SQL query equivalence (Zhao et al., 2023; Ding et al., 2023; Singh and Bedathur, 2024). EQBENCH (Badihi et al., 2021) and SeqCoBench (Maveli et al., 2024) are the main datasets for equivalence checking but have limitations. EQBENCH is too small (272 pairs) for LLM evaluation, while SeqCoBench relies only on statement-level syntactic changes (e.g., renaming variables). In contrast, our work introduces a broader set of equivalence categories and structural transformations, creating a more systematic and challenging benchmark for assessing LLMs' semantic reasoning capabilities.

3 Benchmark Construction

While we have so far discussed only the standard notion of equivalence (that two programs produce the same output on any input), there are other, more precise definitions of equivalence used for each category in the benchmark. For each category, we provide the definition of equivalence, which is included in the prompt when testing LLM reasoning capabilities. We describe the process of generating (in)equivalent pairs for the following six categories:

- **DCE**: C program pairs generated via the compiler's dead code elimination (DCE) pass (Section 3.1).
- **CUDA**: CUDA program pairs created by applying different scheduling strategies using a tensor compiler (Section 3.2).
- **x86-64**: x86-64 assembly program pairs generated by a superoptimizer (Section 3.3).
- **OJ_A, OJ_V, OJ_VA**: Python program pairs from online judge submissions, featuring algorithmic differences (OJ_A), variable-renaming transformations (OJ_V), and combinations of both (OJ_VA) (Section 3.4).

```
char b[2];                          | char b[2];
static int c = 0;                   | static int c = 0;
                                    |
int main() {                        | int main() {
  char* p1 = &b[0];                 |   char* p1 = &b[0];
  int*  p2 = &c;                    |   int*  p2 = &c;
  ...                               |   ...
  if (p1 == p2) {                   |   if (true) {
    c = 1; //dead code              |     c = 1; //live code
  }                                 |   }
  ...                               |   ...
  return 0;                         |   return 0;
}                                   | }
```

Figure 2: **An inequivalent pair from the DCE category in EquiBench.** In the left program, `c = 1` is dead code and has no effect on the program state, whereas in the right program, it is executed and alters the program state. Such cases are generated using the Dead Code Elimination (DCE) pass in compilers.

3.1 Pairs from Program Analysis (DCE)

Dead code elimination (DCE), a compiler pass, removes useless program statements. After DCE, remaining statements in the modified program naturally *correspond* to those in the original program.

Definition of Equivalence. Two programs are considered equivalent if, when executed on the same input, they *always* have identical *program states* at all corresponding points reachable by program execution. We expect language models to identify differences between the two programs, align their states, and determine whether these states are consistently identical.

```cuda
__global__ void GEMV(const float* A,
                     const float* x,
                     float* y,
                     int R,
                     int C) {

    // Calculate the row index
    // assigned to the thread
    int r = blockIdx.x * blockDim.x
            + threadIdx.x;

    // Return if out of bounds
    if (r >= R) return;
    float s = 0.0f;

    for (int c = 0; c < C; c++) {
        s += A[r * C + c] * x[c];
    }

    y[r] = s;
}
```

```cuda
__global__ void GEMV(const float* A, const float* x,
                     float* y, int R, int C) {
    __shared__ float tile[32]; // tiling with shared memory
    int r = blockIdx.x * blockDim.x + threadIdx.x;
    bool valid = (r < R);
    float s = 0.0f;
    for (int start = 0; start < C; start += 32) {
        for (int i = threadIdx.x; i < 32; i += blockDim.x) {
            int c = start + i;
            if (c < C) tile[i] = x[c]; // load x into tile
        }
        __syncthreads();
        if (valid) {
            for (int j = 0; j < min(32, C - start); j++) {
                s += A[r * C + (start + j)] * tile[j];
            }
        }
        __syncthreads();
    }
    if (valid) y[r] = s;
}
```

Figure 3: **An equivalent pair from the CUDA category in EquiBench.** Both programs perform matrix-vector multiplication ($y = Ax$). The right-hand program uses *shared memory tiling* to improve performance. Tensor compilers are utilized to explore different *scheduling strategies*, automating the generation.

Example. Figure 2 illustrates an inequivalent pair of C programs. In the left program, the condition `(p1 == p2)` compares the memory address of the first element of the array `b` with that of the static variable `c`. Since `b` and `c` reside in different memory locations, this condition can never be satisfied. As a result, the assignment `c = 1` is never executed in the left program but is executed in the right program. This difference in program state during execution renders the pair inequivalent.

Automation. This reasoning process is automated by compilers through *alias analysis*, which statically determines whether two pointers can reference the same memory location. Based on this analysis, the compiler's *Dead Code Elimination (DCE)* pass removes code that does not affect program semantics to improve performance.

Dataset Generation. We utilize CSmith (Yang et al., 2011) to create an initial pool of random C programs. Building on techniques from prior compiler testing research (Theodoridis et al., 2022), we implement an LLVM-based tool (Lattner and Adve, 2004) to classify code snippets as either dead or live. Live code is further confirmed by executing random inputs with observable side effects. Equivalent program pairs are generated by eliminating dead code, while inequivalent pairs are generated by removing live code.

3.2 Pairs from Compiler Scheduling (CUDA)

Definition of Equivalence. Two CUDA programs are considered equivalent if they produce the same mathematical output for any valid input, *disregarding floating-point rounding errors*. This definition *differs* from that in Section 3.1, as it does not require the internal program states to be identical during execution.

Example. Figure 3 shows an equivalent CUDA program pair. Both compute matrix-vector multiplication $y = Ax$, where A has dimensions (R, C) and x has size C. The right-hand program applies the *shared memory tiling* technique, loading `x` into shared memory `tile` (declared with `__shared__`). Synchronization primitives `__syncthreads()` are properly inserted to prevent synchronization issues.

Automation. The program transformation can be automated with tensor compilers, which provide a set of *schedules* to optimize loop-based programs. These schedules include loop tiling, loop fusion, loop reordering, loop unrolling, vectorization, and cache optimization. For any given schedule, the compiler can generate the transformed code. While different schedules can significantly impact program performance on the GPU, they do not affect the program's correctness (assuming no compiler bugs), providing the foundation for automation.

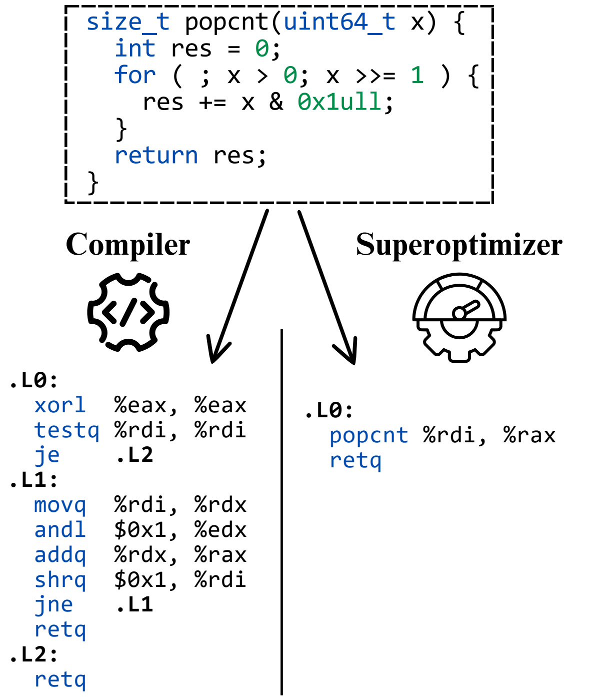

Figure 4: **An equivalent pair from the x86-64 category in EquiBench.** Both programs are compiled from the same C function shown above—the left using a compiler and the right using a *superoptimizer*. The function counts the number of set bits in the input `%rdi` register and stores the result in `%rax`. Their equivalence has been formally verified by the superoptimizer.

Dataset Generation. We utilize TVM as the tensor compiler (Chen et al., 2018) and sample tensor program schedules from TenSet (Zheng et al., 2021) to generate equivalent CUDA program pairs. Inequivalent pairs are created by sampling code from different tensor programs.

3.3 Pairs from a Superoptimizer (x86-64)

Definition of Equivalence. Two x86-64 assembly programs are considered equivalent if, for any input provided in the specified input registers, both programs produce identical outputs in the specified output registers. Differences in other registers or memory are ignored for equivalence checking.

Example. Figure 4 shows an example of an equivalent program pair in x86-64 assembly. Both programs implement the same C function, which counts the number of bits set to 1 in the variable `x` (mapped to the `%rdi` register) and stores the result in `%rax`. The left-hand program, generated by GCC with O3 optimization, uses a loop to count each bit individually, while the right-hand program, produced by a superoptimizer, leverages the `popcnt` instruction, a hardware-supported operation for efficient bit counting. The superoptimizer verifies that both programs are semantically equivalent. Determining this equivalence requires a solid understanding of x86-64 assembly semantics and the ability to reason about all possible bit patterns.

Automation. A superoptimizer searches a space of programs to find one equivalent to the target. Test cases efficiently prune incorrect candidates, while formal verification guarantees the correctness of the optimized program. Superoptimizers apply aggressive and non-local transformations, making semantic equivalence reasoning more challenging. For example, in Figure 4, while a traditional compiler translates the loop in the source C program into a loop in assembly, a superoptimizer can find a more optimal instruction sequence by leveraging specialized hardware instructions. Such semantic equivalence is beyond the scope of traditional compilers.

Dataset Generation. We use Stoke (Schkufza et al., 2013) to generate program pairs. Assembly programs are sampled from prior work (Koenig et al., 2021), and Stoke applies transformations to produce candidate programs. If verification succeeds, the pair is labeled as equivalent; if the generated test cases fail, it is labeled as inequivalent.

3.4 Pairs from Programming Contests

Definition of Equivalence. Two programs are considered equivalent if they solve the same problem by producing the same output for any valid input, as defined by the problem description. Both programs, along with the problem description, are provided to determine equivalence.

Example. Given the problem description in Figure 5, all four programs are equivalent as they correctly compute the Fibonacci number. The **OJ_A** pairs demonstrate **algorithmic** equivalence—the left-hand program uses recursion, while the right-hand program employs a for-loop. The **OJ_V** pairs are generated through **variable renaming**, a **pure syntactic transformation** that can obscure the program's semantics by removing meaningful variable names. The **OJ_VA** pairs combine **both** algorithmic differences and variable renaming.

Dataset Generation. We sample Python submissions using a publicly available dataset from Online Judge (OJ) (Puri et al., 2021). For OJ_A pairs, accepted submissions are treated as equivalent, while

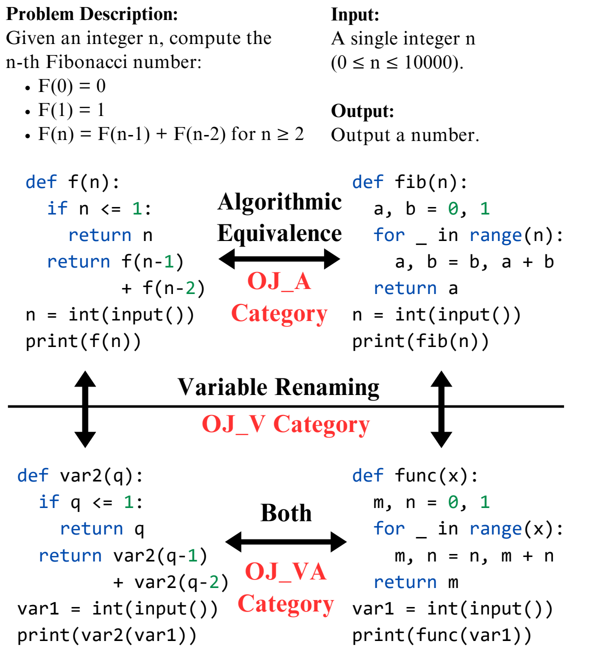

Figure 5: **Equivalent pairs from the OJ_A, OJ_V, OJ_VA categories in EquiBench.** OJ_A pairs demonstrate *algorithmic equivalence*, OJ_V pairs involve *variable renaming* transformations, and OJ_VA pairs combine *both* types of variations.

pairs consisting of an accepted submission and a wrong-answer submission are considered inequivalent. Variable renaming transformations are automated with an open-source tool (Flook, 2025).

4 Experimental Setup

EquiBench. Our dataset, EquiBench, consists of 2,400 program pairs across six equivalence categories. Each category contains 200 equivalent and 200 inequivalent pairs. Table 1 summarizes the lines of code, including the minimum, maximum, and average, for programs in each category, reflecting the wide variation in program lengths. As the dataset generation pipeline is fully automated, additional pairs can be generated as needed.

Category	Language	# Pairs	Lines of Code		
			Min	Max	Avg.
DCE	C	400	98	880	541
CUDA	CUDA	400	46	1733	437
x86-64	x86-64	400	8	29	14
OJ_A	Python	400	3	3403	82
OJ_V	Python	400	2	4087	70
OJ_VA	Python	400	3	744	35

Table 1: **Statistics of the EquiBench dataset.**

Research Questions. We investigate: 1) how different models perform on equivalence checking (Section 5.1); 2) whether prompting techniques, such as few-shot learning (Brown et al., 2020) and Chain-of-Thought (Wei et al., 2022), can enhance performance (Section 5.2); and 3) whether model predictions exhibit bias when judging program equivalence.

Models. We evaluate 17 large language models. For open-source models, including Mixtral (Jiang et al., 2024), Llama (Touvron et al., 2023), Qwen (Bai et al., 2023), DeepSeek (Liu et al., 2024a), we use Together AI, a model serving framework. For closed-source models (e.g., GPT-4 (Achiam et al., 2023), Claude-3.5 (Anthropic, 2024)), we access them via their official APIs, using the default temperature setting.

Prompts. The 0-shot evaluation is conducted using the prompt "You are here to judge if two programs are semantically equivalent. Here equivalence means {*definition*}. [Program 1]: {code1} [Program 2]: {code2} Please only output the answer of whether the two programs are equivalent or not. You should only output Yes or No." The definition of equivalence and the corresponding program pairs are provided for each category. Additionally, for the categories of OJ_A, OJ_V and OJ_VA, the prompt also includes the problem description. The full prompts used in our experiments for each equivalence category are in Appendix A.1.

Error Handling. Some models occasionally fail to follow the instruction to "output Yes or No". To address this issue, we use GPT-4o to parse model outputs. In cases where no result can be extracted, we randomly assign "Yes" or "No" as the model's output. These errors are very rare in advanced models but occur more frequently in smaller models.

5 Results

5.1 Model Accuracy

Table 2 shows the accuracy results for 17 state-of-the-art large language models on EquiBench under zero-shot prompting. Our findings are as follows:

Reasoning models achieve the highest performance, demonstrating a clear advantage over non-reasoning models. As shown in Table 2, reasoning models such as OpenAI o3-mini, DeepSeek R1, and o1-mini significantly outperform all others in our evaluation. This further underscores the

Model	DCE	CUDA	x86-64	OJ_A	OJ_V	OJ_VA	Overall Accuracy
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*
Llama-3.2-3B-Instruct-Turbo	50.0	49.8	50.0	51.5	51.5	51.5	50.7
Llama-3.1-8B-Instruct-Turbo	41.8	49.8	50.5	57.5	75.5	56.8	55.3
Mistral-7B-Instruct-v0.3	51.0	57.2	73.8	50.7	50.5	50.2	55.6
Mixtral-8x7B-Instruct-v0.1	50.2	47.0	64.2	59.0	61.5	55.0	56.1
Mixtral-8x22B-Instruct-v0.1	46.8	49.0	62.7	63.5	76.0	62.7	60.1
Llama-3.1-70B-Instruct-Turbo	47.5	50.0	58.5	66.2	72.0	67.5	60.3
QwQ-32B-Preview	48.2	50.5	62.7	65.2	71.2	64.2	60.3
Qwen2.5-7B-Instruct-Turbo	50.5	49.2	58.0	62.0	80.8	63.0	60.6
gpt-4o-mini-2024-07-18	46.8	50.2	56.8	64.5	91.2	64.0	62.2
Qwen2.5-72B-Instruct-Turbo	42.8	56.0	64.8	72.0	76.5	70.8	63.8
Llama-3.1-405B-Instruct-Turbo	40.0	49.0	75.0	72.2	74.5	72.8	63.9
DeepSeek-V3	41.0	50.7	69.2	73.0	83.5	72.5	65.0
gpt-4o-2024-11-20	43.2	49.5	65.2	71.0	87.0	73.8	65.0
claude3.5-sonnet-2024-10-22	38.5	**62.3**	70.0	71.2	78.0	73.5	65.6
o1-mini-2024-09-12	55.8	50.7	74.2	80.0	89.8	78.8	71.5
DeepSeek-R1	52.2	61.0	78.2	79.8	**91.5**	78.0	73.5
o3-mini-2025-01-31	**68.8**	59.0	**84.5**	**84.2**	88.2	**83.2**	**78.0**
Mean	47.9	52.4	65.8	67.3	76.4	67.0	62.8

Table 2: **Accuracy of 17 models on EquiBench under 0-shot prompting.** We report accuracy for each of the six equivalence categories along with the overall accuracy.

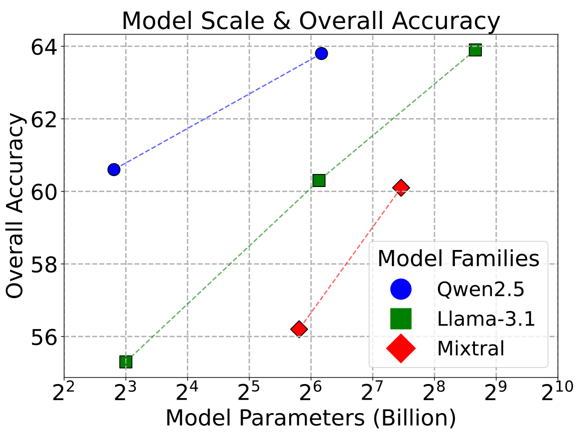

Figure 6: **Scaling Trend on EquiBench.**

complexity of equivalence checking as a code reasoning problem, where reasoning models exhibit a distinct advantage.

EquiBench is a challenging benchmark. Among the 17 models evaluated, OpenAI o3-mini achieves only 59.0% in the CUDA category despite being the top-performing model overall, with an accuracy of 78.0%. For the two most difficult categories, the highest accuracy across all models is 62.3% and 68.8%, respectively, only modestly above the random baseline of 50% accuracy for binary classification, highlighting the substantial room for improvement.

Pure syntactic changes (OJ_V) are the easiest for LLMs, while structural transformations are key to assessing deep semantic reasoning. As shown in the last row of Table 2, the OJ_V category achieves the highest mean accuracy, with DeepSeek-R1 leading at 91.5%. This is because OJ_V pairs are generated through trivial variable renaming, as seen in prior work (Badihi et al., 2021; Maveli et al., 2024). Additionally, combining variable renaming with algorithmic equivalence has little impact on difficulty, as indicated by the small drop in mean accuracy from OJ_A 67.3% to OJ_VA 67.0%. In contrast, all other categories involve non-local structural transformations, making them more challenging and essential for evaluating LLMs' deep semantic reasoning.

Scaling up models improves performance. Larger models generally achieve better performance. Figure 6 shows scaling trends for the Qwen2.5, Llama-3.1, and Mixtral families, where accuracy improves with model size. The x-axis is on a logarithmic scale, highlighting how models exhibit consistent gains as parameters increase.

5.2 Prompting Strategies Analysis

We study few-shot in-context learning and Chain-of-Thought (CoT) prompting, evaluating four strategies: 0-shot, 4-shot, 0-shot with CoT, and 4-shot with CoT. For 4-shot, prompts include 2 equivalent and 2 inequivalent pairs. Appendix A.1 details the prompts, and Table 3 shows the results.

Our key finding is that **prompting strategies *barely* improve performance on EquiBench**, highlighting the task's difficulty and need for

Model	0S	4S	0S-CoT	4S-CoT
o1-mini	71.5	71.5	**71.9**	**71.9**
gpt-4o	65.0	**66.5**	62.5	62.7
DeepSeek-V3	65.0	**66.9**	63.3	62.5
gpt-4o-mini	62.2	**63.5**	60.2	61.2

Table 3: **Accuracies of different prompting techniques.** We evaluate 0-shot and 4-shot in-context learning, both without and with Chain-of-Thought (CoT). Prompting strategies barely improve performance, highlighting the task's difficulty and the need for task-specific approaches.

Model	CUDA		x86-64	
	Eq	Ineq	Eq	Ineq
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*
o3-mini	27.5	90.5	69.5	99.5
o1-mini	2.5	99.0	50.0	98.5
DeepSeek-R1	28.0	94.0	57.5	99.0
DeepSeek-V3	8.5	93.0	44.0	94.5

Table 4: Accuracies on equivalent and inequivalent pairs in the CUDA and x86-64 categories under 0-shot prompting, showing that **models perform significantly better on inequivalent pairs**. Random guessing serves as an unbiased baseline for comparison. Full results for all models are shown in Appendix A.2.

deeper reasoning. Few-shot prompting provides only minor improvements over 0-shot, while Chain-of-Thought shows slight benefits for o1-mini but marginally reduces performance for other models, underscoring the task's complexity and the need for more advanced, task-specific approaches.

5.3 Bias in Model Prediction

We evaluate the prediction bias of the models and observe **a pronounced tendency to misclassify equivalent programs as inequivalent in the CUDA and x86-64 categories**. Table 4 presents the results for four representative models, showing high accuracy for inequivalent pairs but significantly lower accuracy for equivalent pairs, with full results for all models in Appendix A.2.

The bias in the CUDA category arises from extensive structural transformations, such as loop restructuring and shared memory optimizations, which make paired programs appear substantially different. In the x86-64 category, superoptimization applies non-local transformations to achieve optimal instruction sequences, introducing aggressive code restructuring that complicates equivalence reasoning and leads models to frequently misclassify equivalent pairs as inequivalent.

5.4 Case Studies

Models lack capabilities for sound equivalence checking. We find that simple changes that lead to semantic differences can confuse the models, causing them to produce incorrect predictions despite their correct predictions on the original program pairs. For example, o3-mini, which is one of the top-performing models in CUDA category, can correctly classifies the pair shown in Figure 3 as equivalent. Next, we introduce synchronization bugs into the right-hand program, creating two inequivalent pairs with the original left-hand program: (1) removing the first `__syncthreads();` allows reads before all writes complete, causing race conditions; (2) removing the second `__syncthreads();` lets faster threads overwrite shared data while slower threads read it. Despite these semantic differences, o3-mini misclassifies both pairs as equivalent.

Proper hints enable models to correct misjudgments. After o3-mini misclassifies the modified pairs, a hint about removed synchronization primitives allows it to correctly identify both as inequivalent, with accurate explanations highlighting data races. This suggests that training models on dedicated program analysis datasets, beyond only raw source code, may be useful for improving their code reasoning capabilities.

6 Conclusion

This paper presents EquiBench, a dataset for evaluating the code reasoning capabilities of large language models via program equivalence checking. Spanning four programming languages and six equivalence categories, EquiBench challenges models with diverse (in)equivalent program pairs generated through automated transformations, including syntactic changes, structural modifications, and algorithmic equivalence. Our evaluation shows that the best-performing model, OpenAI o3-mini, achieves only 59.0% in the CUDA category and 78.0% overall, with the most challenging categories achieving the best accuracies of just 62.3% and 68.8%, only modestly above the 50% random baseline. Few-shot learning and Chain-of-Thought prompting yield minimal gains, and models exhibit bias toward classifying programs with significant transformations as inequivalent. EquiBench provides a critical benchmark for advancing LLM-based code reasoning.

Limitations

We make every effort to ensure that all pairs are correctly labeled, but cannot guarantee complete accuracy due to potential bugs in the toolchains or errors in the inputs (e.g., solutions from programming contests may be accepted based on a limited set of test cases that might not fully expose underlying bugs in the accepted solutions).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2024. Anthropic. https://www.anthropic.com/news/claude-3-family.

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

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 13–24.

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.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

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

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024a. Reasoning runtime behavior of a program with llm: How far are we? *arXiv preprint cs.SE/2403.16437*.

Liangyu Chen, Bo Li, Sheng Shen, Jingkang Yang, Chunyuan Li, Kurt Keutzer, Trevor Darrell, and Ziwei Liu. 2024b. Large language models are visual reasoning coordinators. *Advances in Neural Information Processing Systems*, 36.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1027–1040.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*.

Manjeet Dahiya and Sorav Bansal. 2017. Black-box equivalence checking across compiler optimizations. In *Asian Symposium on Programming Languages and Systems*, pages 127–147. Springer.

Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving query equivalence using linear integer arithmetic. *Proceedings of the ACM on Management of Data*, 1(4):1–26.

Yangruibo Ding, Jinjun Peng, Marcus J Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024.

Semcoder: Training code language models with comprehensive semantics reasoning. *arXiv preprint arXiv:2406.01006*.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 349–360.

Daniel Flook. 2025. Python variable renaming tool. https://github.com/dflook/python-minifier.

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

Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. 2018. Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 365–382. Springer.

Namgyu Ho, Laura Schmid, and Se-Young Yun. 2022. Large language models are reasoning teachers. *arXiv preprint arXiv:2212.10071*.

Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-level compositional code translation and validation. *arXiv preprint arXiv:2410.24117*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Jason R Koenig, Oded Padon, and Alex Aiken. 2021. Adaptive restarts for stochastic synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 696–709.

Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, Samuele Marro, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. 2024. Code simulation challenges for large language models. *arXiv preprint arXiv:2401.09074*.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024b. Codemind: A framework to challenge large language models for code reasoning. *arXiv preprint arXiv:2402.09664*.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Nickil Maveli, Antonio Vergari, and Shay B Cohen. 2024. What can large language models capture about code functional equivalence? *arXiv preprint arXiv:2408.11081*.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*.

Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific equivalence checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 441–451.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*.

Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. 2024. Semantic code refactoring for abstract data types. *Proceedings of the ACM on Programming Languages*, 8(POPL):816–847.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele

Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, et al. 2025. Humanity's last exam. *arXiv preprint arXiv:2501.14249*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316.

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 391–406.

Alex Shypula, Pengcheng Yin, Jeremy Lacomis, Claire Le Goues, Edward Schwartz, and Graham Neubig. 2021. Learning to superoptimize real-world programs. *arXiv preprint arXiv:2109.13498*.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Rajat Singh and Srikanta Bedathur. 2024. Exploring the use of llms for sql equivalence checking. *arXiv preprint arXiv:2412.05561*.

Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 697–709.

Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. 2024. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1733–1745.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024a. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):709–735.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563*.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024b. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608.

Dylan Zhang, Curt Tigges, Zory Zhang, Stella Biderman, Maxim Raginsky, and Talia Ringer. 2024. Transformer-based models are not yet perfect at learning to emulate structural recursion. *arXiv preprint arXiv:2401.12947*.

Fuheng Zhao, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. 2023. Llm-sql-solver: Can llms determine sql equivalence? *arXiv preprint arXiv:2312.10321*.

Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A Appendix

A.1 Prompts

A.1.1 DCE Category

We show the prompts for 0-shot, 4-shot, 0-shot CoT, 4-shot CoT settings.

0-Shot. You are here to judge if two C programs are semantically equivalent.
Here equivalence means that, when run on the same input, the two programs always have the same program state at all corresponding points reachable by program execution.
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

0-shot CoT. You are here to judge if two C programs are semantically equivalent.
Here equivalence means that, when run on the same input, the two programs always have the same program state at all corresponding points reachable by program execution.
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please output the answer of whether the two programs are equivalent or not. You should output YES or NO in the end. Let's think step by step.

4-shot. You are here to judge if two C programs are semantically equivalent.
Here equivalence means that, when run on the same input, the two programs always have the same program state at all corresponding points reachable by program execution.
[Example 1]:
[Program 1]:

```
int main() {
  int x = 0;
  if (false) {
    x = 1;
  }
  return 0;
}
```

[Program 2]:

```
int main() {
  int x = 0;
  if (true) {
    x = 1;
  }
```

```
    return 0;
  }
```

[Answer]: NO

[Example 2]:

[Program 1]:

```
int main() {
  int x = 0;
  if (false) {
    x = 1;
  }
  return 0;
}
```

[Program 2]:

```
int main() {
  int x = 0;
  return 0;
}
```

[Answer]: YES

[Example 3]:

[Program 1]:

```
char b[2];
static int c = 0;
int main() {
  if (&b[0] == &c) {
      c = 1;
  }
  return 0;
}
```

[Program 2]:

```
char b[2];
static int c = 0;
int main() {
  c = 1;
  return 0;
}
```

[Answer]: NO

[Example 4]:

[Program 1]:

```
char b[2];
static int c = 0;
int main() {
  if (&b[0] == &c) {
      c = 1;
  }
  return 0;
}
```

[Program 2]:

```
char b[2];
static int c = 0;
int main() {
  return 0;
}
```

[Answer]: YES

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

4-shot CoT. You are here to judge if two C programs are semantically equivalent.
Here equivalence means that, when run on the same input, the two programs always have the same program state at all corresponding points reachable by program execution.

[Example 1]:
[Program 1]:

```
int main() {
  int x = 0;
  if (false) {
    x = 1;
  }
  return 0;
}
```

[Program 2]:

```
int main() {
  int x = 0;
  if (true) {
    x = 1;
  }
  return 0;
}
```

[Answer]: x = 1 in program 1 will not be executed, but x = 1 in program 2 will be executed, leading to different program states.
The answer is NO.

[Example 2]:
[Program 1]:

```
int main() {
  int x = 0;
  if (false) {
    x = 1;
  }
  return 0;
}
```

[Program 2]:

```
int main() {
  int x = 0;
  return 0;
}
```

[Answer]: x = 1 in program 1 will not be executed, and this statement does not exist in program 2. Program states are always the same.
The answer is YES.
[Example 3]:

[Program 1]:

```
char b[2];
static int c = 0;
int main() {
  if (&b[0] == &c) {
      c = 1;
  }
  return 0;
}
```

[Program 2]:

```
char b[2];
static int c = 0;
int main() {
  c = 1;
  return 0;
}
```

[Answer]: The if statement in program 1 checks whether the memory address of b[0] equals c's address. c = 1 will not be executed in program 1, leading to a program state different from program 2.
The answer is NO.

[Example 4]:
[Program 1]:

```
char b[2];
static int c = 0;
int main() {
  if (&b[0] == &c) {
      c = 1;
  }
  return 0;
}
```

[Program 2]:

```
char b[2];
static int c = 0;
int main() {
  return 0;
}
```

[Answer]: The if statement in program 1 checks whether the memory address of b[0] equals c's address. c = 1 will not be executed in program 1, so the two programs always have the same states.
The answer is YES.
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please output the answer of whether the two programs are equivalent or not. You should output YES or NO in the end. Let's think step by step.

A.1.2 CUDA Category

We show the prompts for 0-shot and 4-shot CoT settings.

0-Shot. You are here to judge if two CUDA programs are semantically equivalent.
Here equivalence means that, when run on the same valid input, the two programs always compute the same mathematical output (neglecting floating point rounding errors).
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

4-shot CoT. You are here to judge if two CUDA programs are semantically equivalent.
Here equivalence means that, when run on the same valid input, the two programs always compute the same mathematical output (neglecting floating point rounding errors).

[Example 1]:
[Program 1]:

```
__global__ void sgemm_naive(int M, int N, int K, float alpha,
  const float *A, const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

[Program 2]:

```
__global__ void sgemm_naive(int M, int N, int K, float alpha,
  const float *A, const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    C[x * N + y] = beta * tmp + alpha * C[x * N + y];
  }
}
```

[Answer]: Program 1 computes C = alpha*(A@B) + beta*C, while Program 2 computes C = beta*(A@B) + alpha*C.
The answer is NO.

[Example 2]:
[Program 1]:

```
__global__ void sgemm_naive(int M, int N, int K, float alpha,
  const float *A, const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

[Program 2]:

```
template <const uint BLOCKSIZE>
__global__ void sgemm_global_mem_coalesce(int M, int N,
  int K, float alpha, const float *A, const float *B,
  float beta, float *C) {
  const int cRow = blockIdx.x * BLOCKSIZE
                 + (threadIdx.x / BLOCKSIZE);
  const int cCol = blockIdx.y * BLOCKSIZE
                 + (threadIdx.x % BLOCKSIZE);

  if (cRow < M && cCol < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[cRow * K + i] * B[i * N + cCol];
    }
    C[cRow * N + cCol] = alpha * tmp
                       + beta * C[cRow * N + cCol];
  }
}
```

[Answer]: Both programs compute C = alpha*(A@B) + beta*C.
Program 2 improves performance with global memory coalescing, which does not change computation results.
The answer is YES.

[Example 3]:
[Program 1]:

```
__global__ void sgemm_naive(int M, int N, int K, float alpha,
  const float *A, const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

[Program 2]:

```
template <const int BLOCKSIZE>
__global__ void sgemm_shared_mem_block(int M, int N, int K,
  float alpha, const float *A, const float *B, float beta,
  float *C) {
  const uint cRow = blockIdx.x;
  const uint cCol = blockIdx.y;

  __shared__ float As[BLOCKSIZE * BLOCKSIZE];
  __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

  const uint threadCol = threadIdx.x % BLOCKSIZE;
  const uint threadRow = threadIdx.x / BLOCKSIZE;

  A += cRow * BLOCKSIZE * K;
  B += cCol * BLOCKSIZE;
  C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE;

  float tmp = 0.0;
  for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
    As[threadRow * BLOCKSIZE + threadCol] =
                    A[threadRow * K + threadCol];
    Bs[threadRow * BLOCKSIZE + threadCol] =
                    B[threadRow * N + threadCol];

    A += BLOCKSIZE;
    B += BLOCKSIZE * N;

    for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
      tmp += As[threadRow * BLOCKSIZE + dotIdx] *
             Bs[dotIdx * BLOCKSIZE + threadCol];
    }
  }
  C[threadRow * N + threadCol] = alpha * tmp
                    + beta * C[threadRow * N + threadCol];
}
```

[Answer]: Both programs aim to compute C = alpha*(A@B) + beta*C, but there are two synchronization bugs in Program 2.
Before entering the inner loop to compute tmp, there is no guarantee that the cache (As, Bs) is fully populated by all threads.
At the end of each iteration of bkIdx, faster threads may fetch the next block into the cache before slower threads are done.
The answer is NO.

[Example 4]:
[Program 1]:

```
__global__ void sgemm_naive(int M, int N, int K, float alpha,
  const float *A, const float *B, float beta, float *C) {
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

[Program 2]:

```
template <const int BLOCKSIZE>
__global__ void sgemm_shared_mem_block(int M, int N, int K,
  float alpha, const float *A, const float *B, float beta,
  float *C) {
  const uint cRow = blockIdx.x;
  const uint cCol = blockIdx.y;

  __shared__ float As[BLOCKSIZE * BLOCKSIZE];
  __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

  const uint threadCol = threadIdx.x % BLOCKSIZE;
  const uint threadRow = threadIdx.x / BLOCKSIZE;

  A += cRow * BLOCKSIZE * K;
  B += cCol * BLOCKSIZE;
  C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE;

  float tmp = 0.0;
  for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
    As[threadRow * BLOCKSIZE + threadCol] =
            A[threadRow * K + threadCol];
    Bs[threadRow * BLOCKSIZE + threadCol] =
            B[threadRow * N + threadCol];

    __syncthreads();
    A += BLOCKSIZE;
    B += BLOCKSIZE * N;

    for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
      tmp += As[threadRow * BLOCKSIZE + dotIdx] *
            Bs[dotIdx * BLOCKSIZE + threadCol];
    }
    __syncthreads();
  }
  C[threadRow * N + threadCol] = alpha * tmp
            + beta * C[threadRow * N + threadCol];
}
```

[Answer]: Both programs aim to compute C = alpha*(A@B) + beta*C.
Program 2 load a chunk of A and a chunk of B from global memory into shared memory.
Such shared memory cache-blocking improves performance but does not change the correctness of the computation (no bugs found).
The answer is YES.
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please output the answer of whether the two programs are equivalent or not. You should output YES or NO in the end. Let's think step by step.

A.1.3 x86-64 Category

We show the prompts for 0-shot and 4-shot CoT settings.

0-shot. You are here to judge if two x86-64 programs are semantically equivalent.
Here equivalence means that, given any input bits in the register `{def_in}`, the two programs always have the same bits in register `{live_out}`. Differences in other registers do not matter for equivalence checking.

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

4-shot CoT. You are here to judge if two x86-64 programs are semantically equivalent.
Here equivalence means that, given any input bits in the register `{def_in}`, the two programs always have the same bits in register `{live_out}`. Differences in other registers do not matter for equivalence checking.

[Example 1]: In this example, the input register is `%rdi`, and output register is `%rdi`.
[Program 1]:

```
movq -8(%rsp), %rdi
.L4:
sall (%rdi)
movq 8(%rdi), %rdi
.L6:
testq %rdi, %rdi
jne .L4
```

[Program 2]:

```
.L4:
movq -8(%rsp), %rdi
sall (%rdi)
movq 8(%rdi), %rdi
movq %rdi, -8(%rsp)
.L6:
movq -8(%rsp), %rdi
testq %rdi, %rdi
jne .L4
```

[Answer]: The additional instructions in Program 2 are: `movq %rdi, -8(%rsp)` and `movq -8(%rsp), %rdi`.
Program 2 stores the updated `%rdi` value back into `-8(%rsp)` after each iteration and reloads it before the next iteration. But this does not affect the value of `%rdi`.
The answer is YES.

[Example 2]: In this example, the input register is `%rdi`, and output register is `%rdi`.
[Program 1]:

```
movq -8(%rsp), %rdi
.L4:
sall (%rdi)
movq 8(%rdi), %rdi
.L6:
testq %rdi, %rdi
jne .L4
```

[Program 2]:

```
.L4:
movq -8(%rsp), %rdi
sall (%rdi)
movq 8(%rdi), %rdi
movq %rdi, -8(%rsp)
```

```
.L6:
movq -8(%rsp), %rdi
addq $1, %rdi
testq %rdi, %rdi
jne .L4
```

[Answer]: The additional instruction from Program 2 includes `addq $1, %rdi`, which increments `%rdi` by 1 before the test condition.
The two programs do not produce the same result for `%rdi`.
The answer is NO.

[Example 3]: In this example, the input register is `%rdi`, and output register is `%rax`.
[Program 1]:

```
.text
.globl _Z6popcntm
.type _Z6popcntm, @function
_Z6popcntm:
xorl   %eax,%eax
testq  %rdi,%rdi
je     .L_4005b0
nop
.L_4005a0:
movq   %rdi,%rdx
andl   $0x1,%edx
addq   %rdx,%rax
shrq   $0x1,%rdi
jne    .L_4005a0
retq
.L_4005b0:
retq
nop
nop
.size _Z6popcntm, .-_Z6popcntm
```

[Program 2]:

```
.text
.globl _Z6popcntm
.type _Z6popcntm @function
_Z6popcntm:
popcnt %rdi, %rax
retq
.size _Z6popcntm, .-_Z6popcntm
```

[Answer]: Both programs compute the population count (the number of 1s in a number's binary representation) of `%rdi` and store the result in `%rax`.
The answer is YES.

[Example 4]: In this example, the input register is `%rdi`, and output register is `%rax`.
[Program 1]:

```
.text
.globl _Z6popcntm
.type _Z6popcntm, @function
_Z6popcntm:
xorl   %eax, %eax
testq  %rdi, %rdi
je     .L_4005b0
nop
.L_4005a0:
movq   %rdi, %rdx
andl   $0x1, %edx
addq   %rdx, %rax
```

```
addq   $1, %rax
shrq   $0x1, %rdi
jne    .L_4005a0
retq
.L_4005b0:
retq
nop
nop
.size _Z6popcntm, .-_Z6popcntm
```

[Program 2]:

```
.text
.globl _Z6popcntm
.type _Z6popcntm @function
_Z6popcntm:
popcnt %rdi, %rax
retq
.size _Z6popcntm, .-_Z6popcntm
```

[Answer]: The instruction `addq $1, %rax` in Program 1 introduces a discrepancy by adding the number of loop iterations to the output register.
Program 2 simply computes the population count, but Program 1 adds an extra increment for each bit in `%rdi`.
The answer is NO.

The input register is `{def_in}`, and the output register is `{live_out}`.
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please output the answer of whether the two programs are equivalent or not. You should output YES or NO in the end. Let's think step by step.

A.1.4 OJ_A Category

We show the prompts for both 0-shot and 4-shot CoT settings.

0-shot. You are here to judge if two Python programs are semantically equivalent.
You will be given [Problem Description], [Program 1] and [Program 2].
Here equivalence means that, given any valid input under the problem description, the two programs will always give the same output.

[Problem Description]:

```
{problem_html}
```

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

4-shot CoT. You are here to judge if two Python programs are semantically equivalent. You will be given [Problem Description], [Program 1], and [Program 2]. Here equivalence means that, given any valid input under the problem description, the two programs will always give the same output.

[Example 1]:
[Problem Description]:
Given a single line of input containing integers separated by spaces, sort the integers in ascending order and print them in a single line separated by spaces.
Input: A single line containing integers A[i] ($-10^6 \leq A[i] \leq 10^6, 1 \leq n \leq 10^6$).
Output: A single line of integers sorted in ascending order.
Example Input: 4 2 5 1 3
Example Output: 1 2 3 4 5

[Program 1]:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

nums = list(map(int, input().split()))
sorted_nums = bubble_sort(nums)
print(" ".join(map(str, sorted_nums)))
```

[Program 2]:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

nums = list(map(int, input().split()))
sorted_nums = insertion_sort(nums)
print(" ".join(map(str, sorted_nums)))
```

[Answer]: Program 1 is bubble sort, and Program 2 is insertion sort.
The answer is YES.

[Example 2]:
[Problem Description]: Same as Example 1.
[Program 1]: Same as Program 1 from Example 1.
[Program 2]:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] < key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

```
    return arr

nums = list(map(int, input().split()))
sorted_nums = insertion_sort(nums)
print("␣".join(map(str, sorted_nums)))
```

[Answer]: Program 1 is bubble sort, and Program 2 has a bug (the loop condition incorrectly uses arr[j] < key instead of arr[j] > key).
The answer is NO.

[Example 3]:
[Problem Description]: Same as Example 1.
[Program 1]:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] < arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

nums = list(map(int, input().split()))
sorted_nums = bubble_sort(nums)
print("␣".join(map(str, sorted_nums)))
```

[Program 2]: Same as Program 2 from Example 1.

[Answer]: Program 1 has a bug for bubble sort (the comparison is reversed, causing incorrect swaps).
The answer is NO.

[Example 4]:
[Problem Description]: Same as Example 1.
[Program 1]: Same as Program 1 from Example 1.
[Program 2]:

```
nums = list(map(int, input().split()))
sorted_nums = sorted(nums)
print("␣".join(map(str, sorted_nums)))
```

[Answer]: Program 1 is bubble sort, and Program 2 uses Python's built-in sorting implementation.
The answer is YES.

[Problem Description]:

```
{problem_html}
```

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please output the answer of whether the two programs are equivalent or not. You should output YES or NO in the end. Let's think step by step.

A.1.5 OJ_V Category

We show the prompt for 4-shot CoT settings.

4-shot CoT. You are here to judge if two Python programs are semantically equivalent. You will be given [Problem Description], [Program 1] and [Program 2]. Here equivalence means that, given any valid input under the problem description, the two programs will always give the same output.

[Example 1]:

[Problem Description]:

Given a single line of input containing integers separated by spaces, sort the integers in ascending order and print them in a single line separated by spaces.

Input: A single line containing integers A[i] ($-10^6 <= A[i] <= 10^6, 1 <= n <= 10^6$).

Output: A single line of integers sorted in ascending order.

Example Input:

4 2 5 1 3

Example Output:

1 2 3 4 5

[Program 1]:

```
nums = list(map(int, input().split()))
sorted_nums = sorted(nums)
print(" ".join(map(str, sorted_nums)))
```

[Program 2]:

```
random_var1 = list(map(int, input().split()))
random_var2 = sorted(random_var1)
print(" ".join(map(str, random_var2)))
```

[Answer]: The only difference is in variable names, which do not affect the logic or output of the program. The answer is YES.

[Example 2]:

[Problem Description]:

```
Same as Example 1.
```

[Program 1]:

```
nums = list(map(int, input().split()))
sorted_nums = sorted(nums)
print(" ".join(map(str, sorted_nums)))
```

[Program 2]:

```
nums = list(map(int, input().split()))
sorted_nums = nums.sort()
print(" ".join(map(str, sorted_nums)))
```

[Answer]: Program 1 sorts the integers in the correct way. In Program 2, nums.sort() modifies the list in place and returns None. Program 2 will trigger a TypeError.
The answer is NO.

[Example 3]:

[Problem Description]:

Given a list of integers, remove all duplicate values while maintaining the order of their first appearance and print the resulting list in a single line, separated by spaces.

Input: A single line containing integers A[i] ($-10^6 <= A[i] <= 10^6, 1 <= n <= 10^5$).

Output: A single line containing the integers from the input with duplicates removed, in the order of their first appearance.
Example Input:
4 5 4 2 5 1 3
Example Output:
4 5 2 1 3
[Program 1]:

```
nums = list(map(int, input().split()))
unique_nums = []
for num in nums:
    if num not in unique_nums:
        unique_nums.append(num)
print("␣".join(map(str, unique_nums)))
```

[Program 2]:

```
random_var1 = list(map(int, input().split()))
random_var2 = []
for random_var3 in random_var1:
    if random_var3 not in random_var2:
        random_var2.append(random_var3)
print("␣".join(map(str, random_var2)))
```

[Answer]: The only difference is in variable names, which do not affect the logic or output of the program. The answer is YES.

[Example 4]:
[Problem Description]:

```
Same as Example 3.
```

[Program 1]:

```
nums = list(map(int, input().split()))
unique_nums = []
for num in nums:
    if num not in unique_nums:
        unique_nums.append(num)
print("␣".join(map(str, unique_nums)))
```

[Program 2]:

```
nums = list(map(int, input().split()))
unique_nums = []
for num in nums:
    if num in unique_nums:
        unique_nums.append(num)
print("␣".join(map(str, unique_nums)))
```

[Answer]: Program 1 correctly appends unique values to unique_nums by checking if num not in unique_nums.
Program 2 is incorrect because it uses if num in unique_nums, causing only duplicates to be appended to the list.
The answer is NO.

[Problem Description]:

```
{problem_html}
```

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please output the answer of whether the two programs are equivalent or not. You should output YES or NO in the end. Let's think step by step.

A.1.6 OJ_VA Category

We show the prompt for 4-shot CoT settings.

4-shot CoT. You are here to judge if two Python programs are semantically equivalent.
You will be given [Problem Description], [Program 1] and [Program 2].
Here equivalence means that, given any valid input under the problem description, the two programs will always give the same output.

[Example 1]:
[Problem Description]:
Given a single line of input containing integers separated by spaces, sort the integers in ascending order and print them in a single line separated by spaces.
Input: A single line containing integers A[i] (-10^6 <= A[i] <= 10^6, 1 <= n <= 10^6).
Output: A single line of integers sorted in ascending order.
Example Input:
4 2 5 1 3
Example Output:
1 2 3 4 5
[Program 1]:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

nums = list(map(int, input().split()))
sorted_nums = bubble_sort(nums)
print(" ".join(map(str, sorted_nums)))
```

[Program 2]:

```
def random_sort(rand_var1):
    for rand_var2 in range(1, len(rand_var1)):
        rand_var3 = rand_var1[rand_var2]
        rand_var4 = rand_var2 - 1
        while rand_var4 >= 0 and rand_var1[rand_var4] > rand_var3:
            rand_var1[rand_var4 + 1] = rand_var1[rand_var4]
            rand_var4 -= 1
        rand_var1[rand_var4 + 1] = rand_var3
    return rand_var1

rand_input = list(map(int, input().split()))
rand_output = random_sort(rand_input)
print(" ".join(map(str, rand_output)))
```

[Answer]: Program 1 is bubble sort, and Program 2 is insertion sort (though the variable names are randomized).
The answer is YES.

[Example 2]:
[Problem Description]:
Same as Example 1.
[Program 1]:
Same as Program 1 from Example 1.
[Program 2]:

```
def insertion_sort(rand_var1):
    for i in range(1, len(rand_var1)):
        key = rand_var1[i]
        j = i - 1
        while j >= 0 and rand_var1[j] < key:
            rand_var1[j + 1] = rand_var1[j]
            j -= 1
        rand_var1[j + 1] = key
    return rand_var1

nums = list(map(int, input().split()))
sorted_nums = insertion_sort(nums)
print(" ".join(map(str, sorted_nums)))
```

[Answer]: Program 1 is bubble sort, and Program 2 has a bug (the loop condition incorrectly uses rand_var1[j] < key instead of rand_var1[j] > key).
The answer is NO.

[Example 3]:
[Problem Description]:
Same as Example 1.
[Program 1]:

```
def rand_alg(rand_var):
    n = len(rand_var)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if rand_var[j] < rand_var[j + 1]:
                rand_var[j], rand_var[j + 1] = rand_var[j + 1], rand_var[j]
    return rand_var

nums = list(map(int, input().split()))
sorted_nums = rand_alg(nums)
print(" ".join(map(str, sorted_nums)))
```

[Program 2]:
Same as Program 2 from Example 1.
[Answer]: Program 1 has a bug for bubble sort (the comparison is reversed, causing incorrect swaps).
The answer is NO.

[Example 4]:
[Problem Description]:
Same as Example 1.
[Program 1]:
Same as Program 1 from Example 1.
[Program 2]:

```
nums = list(map(int, input().split()))
sorted_nums = sorted(nums)
print(" ".join(map(str, sorted_nums)))
```

[Answer]: Program 1 is bubble sort, and Program 2 uses Python's built-in sorting implementation. The answer is YES.

[Problem Description]:

```
{problem_html}
```

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please output the answer of whether the two programs are equivalent or not. You should output YES or NO in the end. Let's think step by step.

A.2 Model Prediction Bias

We evaluate the prediction bias of the models and observe a pronounced tendency to misclassify equivalent programs as inequivalent in the CUDA and x86-64 categories. The table here shows the full results on all models under 0-shot prompting.

Model	**CUDA**		**x86-64**	
	Eq	**Ineq**	**Eq**	**Ineq**
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*
deepseek-ai/DeepSeek-V3	8.5	93.0	44.0	94.5
deepseek-ai/DeepSeek-R1	28.0	94.0	57.5	99.0
meta-llama/Llama-3.1-405B-Instruct-Turbo	6.0	92.0	68.5	81.5
meta-llama/Llama-3.1-8B-Instruct-Turbo	2.0	97.5	1.0	100.0
meta-llama/Llama-3.1-70B-Instruct-Turbo	7.0	93.0	27.5	89.5
meta-llama/Llama-3.2-3B-Instruct-Turbo	0.0	99.5	0.0	100.0
anthropic/claude-3-5-sonnet-20241022	62.5	62.0	49.5	90.5
Qwen/Qwen2.5-7B-Instruct-Turbo	18.5	80.0	17.5	98.5
Qwen/Qwen2.5-72B-Instruct-Turbo	14.5	97.5	36.0	93.5
Qwen/QwQ-32B-Preview	35.0	66.0	39.0	86.5
mistralai/Mixtral-8x7B-Instruct-v0.1	18.0	76.0	50.5	78.0
mistralai/Mixtral-8x22B-Instruct-v0.1	10.5	87.5	32.5	93.0
mistralai/Mistral-7B-Instruct-v0.3	52.5	62.0	87.0	60.5
openai/gpt-4o-mini-2024-07-18	0.5	100.0	16.5	97.0
openai/gpt-4o-2024-11-20	0.0	99.0	68.5	62.0
openai/o3-mini-2025-01-31	27.5	90.5	69.5	99.5
openai/o1-mini-2024-09-12	2.5	99.0	50.0	98.5