INTERACTIVE PROGRAM INDUCTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Program induction refers to the process in which a learner aims to recover a program from a few given input–output examples. This paradigm broadens programming access by letting users demonstrate the rule-based behavior they want to apply to unseen data, instead of having to implement it themselves. However, recovering a program from finite input–output examples is fundamentally ambiguous, as a finite set of input–output examples is theoretically consistent with infinitely many programs whose output diverges on unseen inputs. We view this problem from a Bayesian perspective, where, given a prior p^0 over programs, identifying the target program means concentrating posterior probability mass on it by observing additional input–output examples. In the interactive program induction setting, the learner chooses maximally informative inputs and queries an oracle, such as the user, for their outputs. We derive a greedy query-selection algorithm and show that it identifies a target program $f^\star$ up to approximate equivalence after $\mathcal{O}(\log 1/p^0(f^\star))$ informative oracle queries. The algorithm uses a language model as the prior under a PAC-style relaxation of program equivalence, where the query-selection objective is estimated using importance sampling. On the CodeARC benchmark, the algorithm improves accuracy over the LM only agent by up to 22 percentage points while observing up to 10.5 fewer input–output examples.

1 INTRODUCTION

Many domain experts have a computer program clearly in mind that encodes certain rule-based behavior, but they cannot easily write it down themselves. For instance, a spreadsheet user often knows how they would like the column of names to be reformatted, but cannot write the regex. Similarly, a linguist knows how a sentence in their native language should be segmented, but cannot program the tokenizer. *Inductive programming*, also known as programming by example (Summers, 1977; Lieberman, 2001; Gulwani et al., 2012; 2017), aims to bridge this gap by recovering the program from input–output examples. The paradigm has been deployed at scale for specific applications, most prominently in Microsoft Excel's FlashFill (Gulwani, 2011).

However, recovering a target program from finite input–output examples is fundamentally ambiguous, since two programs that agree on an initial *finite* set of input–output pairs may still diverge on unseen data. In fact, any finite set of examples is theoretically consistent with infinitely many programs that compute distinct functions, so without an inductive bias, it is unclear which to choose (Solomonoff, 1964). Prior work addresses this challenge by restricting the program space to a hand-designed domain-specific language (DSL) with a small grammar (Shapiro, 1983; Gulwani et al., 2017), at the cost of expressivity (cf. App. A). Within a DSL, the programs consistent with the given examples can be enumerated, and the target one selected through human feedback, for instance by asking for further examples (Mayer et al., 2015; Ji et al., 2020), or through an oracle that answers such queries in place of a human (Solar-Lezama, 2008; Jha et al., 2010). Large language models trained on code (Chen et al., 2021; Austin et al., 2021; Li et al., 2022) may help to go beyond DSLs by providing distributions over implementations in a general-purpose programming language (e.g., Python) given an input–output specification. Still, when used as one-shot program inductors, they offer few mechanisms or guarantees for selecting the target program among the many consistent with the examples. For instance, Li & Ellis (2024) report a substantial gap to hand-engineered systems on programming-by-example, and Wei et al. (2025) report limited success of LLM agents doing autonomous program induction in Python with oracle feedback.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

In this work, we propose an algorithm for interactive program induction over a general-purpose programming language, and derive theoretical guarantees on the number of oracle queries it requires and on the accuracy of the recovered program. To this end, we treat the target program as the unknown of a Bayesian inference problem, with a language model prior p^0 over programs as inductive bias and a posterior that concentrates on $f^\star$ as input–output examples are observed, and we choose which inputs to query by sequential Bayesian experimental design (Lindley, 1956; MacKay, 1992; Grand et al., 2026; Kobalczyk et al., 2025). Using a notion of approximate equivalence between programs (exact equivalence being undecidable, Rice, 1953) and an expected information gain criterion for query selection, we show that the algorithm terminates after $\mathcal{O}(\frac{1}{\gamma}\log 1/p^0(f^\star))$ queries when every answer carries at least γ bits. Dasgupta (2004) analyzed this greedy rule over finite hypothesis classes, and our analysis carries over to programs because we measure agreement on a finite sample of inputs. Conditional on returning the target's equivalence class, the returned program is approximately correct with high probability. Finally, we evaluate the practical implications on the CodeARC program induction dataset (Wei et al., 2025) and show that the algorithm improves accuracy over the CodeARC LM agent while reducing the number of required oracle calls in most evaluated settings.

2 Information-Theoretic Program Induction

We formalize interactive program induction as the problem of recovering a **target function** $f^\star$ from input–output observations that are revealed incrementally. Let X be a countable set of **inputs** and Y a countable set of **outputs**, and let the target $f^\star\colon X \to Y$ be an unknown but fixed element of the candidate space Y^X. We observe $f^\star$ only through its behavior on selected inputs. At each round t we choose an input $x^{(t)} \in X$, the **query**, and obtain its output $y^{(t)} = f^\star(x^{(t)})$ from the oracle. Our goal is to return a function $\widehat{f}$ that closely approximates $f^\star$ after as few queries as possible.

2.1 Setup

A finite sample of input–output pairs does not pin down a target in Y^X, since infinitely many functions agree on any finite subset of X and disagree elsewhere (Solomonoff, 1964). Recovering $f^\star$ therefore requires two pieces of additional structure, which together turn this ill-posed recovery problem into a well-posed Bayesian inference problem. The first is a **program prior** p^0 over Y^X, which encodes the inductive bias. Many functions are consistent with a finite observation set, and the prior determines which of them are a priori plausible. The second is an **input distribution** μ over X, specifying the population of inputs on which the recovered function should agree with $f^\star$. The program prior decides *which* functions we prefer, and the input distribution decides *where* agreement is measured. We assume that p^0 has countable support with $p^0(f^\star) > 0$. A program that raises or exceeds a time limit on an input returns a distinguished error output, so every program defines a function on all of X. In the setting most relevant to this work (program synthesis over strings, where $X = Y = \Sigma^*$) the input distribution μ has countably infinite support. Comparing a candidate against $f^\star$ on every input in this support is infeasible, and even integrating against μ to estimate disagreement requires structure we do not assume. In the following, we derive the approximate notions of equivalence we use throughout.

2.2 Approximate Equivalence

Two facts about the interactive setting force us to relax exact program equivalence. First, exact equivalence of two computable functions is undecidable (Rice, 1953), so no algorithm can take two such functions and decide whether they agree on every input. Second, two functions can be compared on any single input by running them, but the query budget is finite while X is typically infinite, so agreement on every input cannot be certified by exhaustive testing. We therefore want an equivalence relation that (i) measures disagreement under the population μ and (ii) can be certified by inspecting only finitely many inputs. The first relaxation accomplishes (i) by declaring f equivalent to $f^\star$ when their disagreement under μ is small. For $\epsilon \in (0, 1)$,

$$f \sim_\epsilon f^\star \iff \mathrm{err}_\mu(f) \le \epsilon, \qquad \mathrm{err}_\mu(f) \stackrel{\text{def}}{=} \Pr_{x\sim\mu}[f(x) \neq f^\star(x)]. \tag{1}$$

Checking $\sim_\epsilon$ requires integration against μ, which we cannot do directly. The second relaxation makes equivalence checkable. We draw a random sample $S = \{x_1, \dots, x_N\}$ i.i.d. from μ and replace

agreement under μ with agreement on S,

$$f \sim_S f^\star \iff f(x) = f^\star(x) \text{ for all } x \in S. \tag{2}$$

Write $[f]_\sim$ for the $\sim_S$-class of f and $Y^X/\sim$ for the quotient, with $|Y^X/\sim| \leq |Y|^N$. Because S is finite, $\sim_S$ can be verified on observed data. Because S is drawn from μ, agreement on S also controls μ-error with high probability over the draw.

Proposition 2.1 (PAC Error Bound, cf. Blumer et al. 1987)**.** *Let $S = \{x_1, \ldots, x_N\}$ be drawn i.i.d. from μ. With probability at least $1 - \delta$ over the draw of S, every $f \in [f^\star]_\sim$ with $p^0(f) > 0$ satisfies*

$$\mathrm{err}_\mu(f) \leq \frac{\ln(1/p^0(f)) + \ln(1/\delta)}{N}. \tag{3}$$

Proof. In App. C. ■

The bound in Eq. (3) depends on f through $\ln(1/p^0(f))$. An f with more prior mass gets a smaller error bound from the same evaluation set, or reaches a given error bound with a smaller evaluation set. Throughout the remainder of the paper, for notational brevity, $\sim$ and $Y^X/\sim$ refer to $\sim_S$ and $Y^X/\sim_S$ for a fixed evaluation set S drawn once from μ^N.

2.3 Bayesian Updating

With the equivalence relation in place, we set up the Bayesian inference machinery on which our query selection rule will operate.

Update rule. Write $\mathcal{D}_0 \subset X \times Y$ for the finite set of seed examples $(x, f^\star(x))$ given with the task (Wei et al., 2025), possibly empty, and $\mathcal{D}_t = \mathcal{D}_0 \cup \{(x^{(s)}, y^{(s)})\}_{s=1}^t$ for the dataset after t oracle answers. The posterior over Y^X given $\mathcal{D}_t$ is

$$p(f \mid \mathcal{D}_t) = \frac{p^0(f) \cdot \prod_{(x,y)\in\mathcal{D}_t} \mathbb{1}\{f(x) = y\}}{\sum_{g\in Y^X} p^0(g) \cdot \prod_{(x,y)\in\mathcal{D}_t} \mathbb{1}\{g(x) = y\}}. \tag{4}$$

The posterior admits an equivalent recursive form,

$$p(f \mid \mathcal{D}_t) = \frac{p(f \mid \mathcal{D}_{t-1}) \cdot \mathbb{1}\{f(x^{(t)}) = y^{(t)}\}}{p(y^{(t)} \mid x^{(t)}, \mathcal{D}_{t-1})}, \tag{5a}$$

$$p(y \mid x, \mathcal{D}_t) \stackrel{\text{def}}{=} \sum_{g\in Y^X} p(g \mid \mathcal{D}_t) \cdot \mathbb{1}\{g(x) = y\}, \tag{5b}$$

where $p(y \mid x, \mathcal{D}_t)$ is the posterior predictive distribution of the output at an input x. The recursive form Eq. (5a) is the version we instantiate computationally in §4.

Pushforward onto $Y^X/\sim$. By Prop. 2.1, it suffices to identify the $\sim$-class of $f^\star$, since any $f \in [f^\star]_\sim$ has small $\mathrm{err}_\mu(f)$ with high probability over S. The natural inference target is therefore the pushforward of the posterior on Y^X onto the quotient,

$$p_\sim([f]_\sim \mid \mathcal{D}_t) \stackrel{\text{def}}{=} \sum_{g\in[f]_\sim} p(g \mid \mathcal{D}_t), \tag{6}$$

which assigns to each equivalence class the total posterior mass of its members. §2.4 selects queries to concentrate $p_\sim(\cdot \mid \mathcal{D}_t)$ on a single class as quickly as possible.

2.4 Greedy Information Gain Algorithm

We now describe an algorithm that constructs $\mathcal{D}_1, \mathcal{D}_2, \ldots$ by selecting a maximally informative query $x^{(t)} \in S$ at a time. Queries are selected from S because that is where the classes are defined. All programs in a class return the same output on every $x \in S$, so the answer to such a query eliminates whole classes and never splits one, and the posterior stays a distribution over the same classes. Any two distinct classes disagree on some $x \in S$, so queries in S suffice to identify $[f^\star]_\sim$. Each query is chosen by the expected information gain criterion of Bayesian experimental design (Lindley, 1956; MacKay, 1992), with the equivalence class of the target as the quantity of interest. More precisely, at each round $t = 1, 2, \ldots$:

1. **Select.** Choose the input $x^{(t)} \in S$ that maximizes the expected information gain (Shannon, 1948; MacKay, 1992) with respect to $p_\sim(\cdot \mid \mathcal{D}_{t-1})$,

$$x^{(t)} = \operatorname*{argmax}_{x \in S} \mathrm{H}\big[p_\sim(\cdot \mid \mathcal{D}_{t-1})\big] - \sum_{y \in Y} p(y \mid x, \mathcal{D}_{t-1})\, \mathrm{H}\big[p_\sim(\cdot \mid \mathcal{D}_{t-1} \cup \{(x, y)\})\big], \tag{7}$$

   where $\mathrm{H}[\cdot]$ denotes the Shannon entropy and $p(\cdot \mid x, \mathcal{D}_{t-1})$ is the posterior predictive of Eq. (5b).
2. **Query.** Observe $y^{(t)} = f^\star(x^{(t)})$ and set $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{(x^{(t)}, y^{(t)})\}$.
3. **Update.** Compute $p(\cdot \mid \mathcal{D}_t)$ via Eq. (5a) and the pushforward $p_\sim(\cdot \mid \mathcal{D}_t)$ via Eq. (6).

Terminate at the first round t such that some class $\widehat{c}$ reaches majority pushforward mass, i.e., when $p_\sim(\widehat{c} \mid \mathcal{D}_t) > \frac{1}{2}$, and return any $\widehat{f} \in \widehat{c}$. Conditional on $\widehat{c} = [f^\star]_\sim$, the returned $\widehat{f}$ is then consistent with $f^\star$ on S, and Prop. 2.1 gives $\mathrm{err}_\mu(\widehat{f}) \le \epsilon$ with probability at least $1 - \delta$ over the draw of S once $N \ge (\ln(1/p^0(\widehat{f})) + \ln(1/\delta))/\epsilon$.

3 Query Complexity and Correctness Guarantees

We now derive a bound for the termination round T of the greedy algorithm of §2.4 and characterize the returned program class $\widehat{c}$. Both are stated in terms of the entropy of the prior over classes and of the information a round is guaranteed to yield, which depends on the queries available in the evaluation set and on which of them the selection strategy picks.

Theorem 3.1 (Expected query complexity and correctness)**.** *Let $f^\star \sim p^0$ and let $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)] < \infty$ be the prior class entropy in bits. Write $\bar{\gamma}_t \stackrel{\text{def}}{=} \max_{x \in S} -\log_2 \max_y p(y \mid x, \mathcal{D}_{t-1})$ for the most information an answer to a query in S is guaranteed to carry at round t. The greedy strategy of §2.4 satisfies*

(i) $\mathbb{E}_{f^\star \sim p^0}\big[\sum_{t=1}^{T} \bar{\gamma}_t\big] \le \mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)]$;
(ii) $T \le N$ *for every target;*
(iii) *if $\bar{\gamma}_t \ge \gamma$ for all $t \le T$ and every $f^\star$ with $p^0(f^\star) > 0$, then* $\mathbb{E}_{f^\star \sim p^0}[T] \le \mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)]/\gamma$.

At termination, $\Pr[\widehat{c} = [f^\star]_\sim \mid \mathcal{D}_T] = p_\sim(\widehat{c} \mid \mathcal{D}_T) > \frac{1}{2}$*, i.e., for $f^\star \sim p^0$ the terminal mass is the exact probability that the returned program is $\sim_S$-equivalent to $f^\star$.*

Proof. In App. C. ∎

Theorem 3.2 (Per-instance query complexity and correctness)**.** *Fix $f^\star$ with $p^0(f^\star) > 0$ and let $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)] < \infty$. Write $\gamma_s \stackrel{\text{def}}{=} -\log_2 \max_y p(y \mid x^{(s)}, \mathcal{D}_{s-1})$ for the information the answer to the query of round s is guaranteed to carry. If $p_\sim([f^\star]_\sim \mid \mathcal{D}_0) \le \frac{1}{2}$ and $\gamma_s \ge \gamma$ for all $s \le T$, then any selection rule over S terminates after*

$$T \le \left\lfloor \frac{1}{\gamma} \log_2 \frac{1}{2 p_\sim([f^\star]_\sim \mid \mathcal{D}_0)} \right\rfloor + 1. \tag{8}$$

Moreover, $\widehat{c} \ne [f^\star]_\sim$ if and only if some class with prior mass strictly greater than $p_\sim([f^\star]_\sim \mid \mathcal{D}_0)$ is consistent with every query made. In particular, if no class outweighs $[f^\star]_\sim$ under the prior, termination returns $\widehat{c} = [f^\star]_\sim$, and under the hypotheses of Eq. (8) it does so within that bound.

Proof. In App. C. ∎

Thm. 3.1 bounds the number of queries in expectation over targets drawn from the prior. It is specific to the greedy rule, whose query has the largest expected information in S and therefore at least $\bar{\gamma}_t$, the largest guaranteed information any query in S offers. Thm. 3.2 bounds the number of queries for a fixed target program. Its proof does not use the greedy rule, so the bound applies to every selection rule over S with the guaranteed information of that rule's own queries, and §5 empirically evaluates several such rules. In both, the rate of convergence is set by $\bar{\gamma}_t$ over S and γ_t at the selected query, which are determined by how the program prior spreads mass across classes and how finely S, drawn from the input distribution, separates them.

Improvement over non-adaptive querying. Fix a target error ϵ and confidence δ. A non-adaptive strategy draws inputs at random from μ, has the oracle label all of them, and accepts a program that agrees with the target on every one. It needs $\Omega(\log(1/\delta)/\epsilon)$ labeled inputs, since a program that disagrees with $f^\star$ on a set of μ-mass ϵ passes m random inputs with probability $(1-\epsilon)^m \geq e^{-2m\epsilon}$ for $\epsilon \leq \frac{1}{2}$ (see, e.g., Shalev-Shwartz & Ben-David, 2014), and by Prop. 2.1 inputs of that order also suffice, up to the $\ln(1/p^0(f))$ term. The greedy algorithm needs an evaluation set of the same size, but the oracle labels only the $T \leq N$ inputs it queries. Since $f^\star$ lies in its own class, that class has prior mass at least $p^0(f^\star)$ whatever S is, so under the hypotheses of Thm. 3.2 the algorithm asks at most $\lfloor \frac{1}{\gamma} \log_2 \frac{1}{2p^0(f^\star)} \rfloor + 1$ queries, a number that does not depend on ϵ. The non-adaptive cost grows with $1/\epsilon$, while the adaptive cost is set by how plausible the target is under the prior. The PAC certificate of Prop. 2.1 then holds for the returned program, conditional on the algorithm returning the target's class.

The role of $|S|$. Combining Prop. 2.1 with Thms. 3.1 and 3.2 gives a complete guarantee. The greedy algorithm terminates after at most $\lfloor \frac{1}{\gamma} \log_2 \frac{1}{2p_\sim([f^\star]_\sim \mid \mathcal{D}_0)} \rfloor + 1$ queries when every answer carries at least γ bits, for a target drawn from the prior it returns the target's class with probability $p_\sim(\widehat{c} \mid \mathcal{D}_T) > \frac{1}{2}$, and in that case the returned program has error at most $(\ln(1/p^0(\widehat{f})) + \ln(1/\delta))/N$ with probability $1-\delta$ over S. The size of the evaluation set enters the two bounds in opposite directions. A larger S makes the error bound tighter, but it also refines $\sim$, so the target's class shrinks, its prior mass decreases, and the query bound grows. We evaluate the effect of $|S|$ empirically in §5.

4 Approximate Inference with a Language Model

The greedy algorithm of §2.4 requires the pushforward entropy $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_t)]$, a sum over equivalence classes weighted by posterior mass. The posterior is a distribution over the countably infinite support of the prior, so neither it nor this sum can be computed exactly. We approximate the posterior by a weighted set of programs sampled from a language model, and compute the pushforward from the weights.

4.1 Language Model Proposal

Language model preliminaries. An **alphabet** Σ is a finite, non-empty set of symbols, Σ^* the set of all strings $\boldsymbol{x}$ formed from Σ (including the empty string), and $|\boldsymbol{x}|$ the length of $\boldsymbol{x}$. A **language model** p_{LM} is a probability distribution over Σ^*. The **prefix probability** $\vec{p}_{\mathrm{LM}}(\boldsymbol{x}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{x}' \in \Sigma^*} p_{\mathrm{LM}}(\boldsymbol{x}\,\boldsymbol{x}')$ is the probability that a string drawn from p_{LM} begins with $\boldsymbol{x}$. Whenever $\vec{p}_{\mathrm{LM}}(\boldsymbol{x}) > 0$, the **conditional prefix probability** is

$$\vec{p}_{\mathrm{LM}}(\boldsymbol{x}' \mid \boldsymbol{x}) \stackrel{\text{def}}{=} \begin{cases} p_{\mathrm{LM}}(\boldsymbol{x})/\vec{p}_{\mathrm{LM}}(\boldsymbol{x}) & \text{if } \boldsymbol{x}' = \square, \\ \vec{p}_{\mathrm{LM}}(\boldsymbol{x}\,\boldsymbol{x}')/\vec{p}_{\mathrm{LM}}(\boldsymbol{x}) & \text{otherwise,} \end{cases} \tag{9}$$

where $\square \notin \Sigma$ is a distinguished end-of-string delimiter. Iterating Eq. (9) yields the autoregressive factorization

$$p_{\mathrm{LM}}(\boldsymbol{x}) = \vec{p}_{\mathrm{LM}}(\square \mid \boldsymbol{x}) \prod_{t=1}^{|\boldsymbol{x}|} \vec{p}_{\mathrm{LM}}(x_t \mid \boldsymbol{x}_{<t}), \tag{10}$$

which computes $p_{\mathrm{LM}}(\boldsymbol{x})$ from $|\boldsymbol{x}|+1$ next-token queries without enumerating Σ^*. The same definitions apply to a language model conditioned on a context string.

Proposal over programs. An LM is a distribution over strings, but our algorithm needs distributions over programs. We obtain them by prompting the LM with input–output examples and asking for a program consistent with them (App. F gives the prompts), so that its distribution over completions is a distribution over source code. Strictly, this makes p_{LM} a distribution over *implementations*, whereas a *program* is an input–output behavior, and two distinct strings $\boldsymbol{x}_1 \neq \boldsymbol{x}_2$ may decode to the same function while $p_{\mathrm{LM}}(\boldsymbol{x}_1) \neq p_{\mathrm{LM}}(\boldsymbol{x}_2)$. The equivalence relation $\sim$ from §2.2 and the pushforward $p_\sim$ from §2.3 collapse implementations that agree on S into a single class, so the algorithm operates at the level of programs, indifferent to syntactic differences between implementations, while sampling and density evaluation happen at the level of implementations, where Eq. (10) makes them tractable.

At each round t we render the observed examples into a prompt, a context string $\mathrm{prompt}(\mathcal{D}_t) \in \Sigma^*$ (App. F), and condition the LM on it. Its completions are implementations $\boldsymbol{x} \in \Sigma^*$, each of which decodes to a function by execution. The proposal over implementations is

$$q_t(\boldsymbol{x}) \stackrel{\text{def}}{=} p_{\mathrm{LM}}(\boldsymbol{x} \mid \mathrm{prompt}(\mathcal{D}_t)), \tag{11}$$

and we take the program prior to be the proposal before any query, $p^0 \stackrel{\text{def}}{=} q_0$. The fixed objects p^0 and μ define the guarantees of §3, while the proposal q_t is an approximation mechanism whose quality affects practical efficiency but not the per-instance query bound of Thm. 3.2.

4.2 Importance Sampling Estimates

Importance weights. We draw M implementations from q_t, write $f_1, \ldots, f_M$ for their decoded functions and $p^0(f_m)$, $q_t(f_m)$ for the densities of the sampled implementations, and assign self-normalizing importance weights

$$w_m = \frac{p^0(f_m) \cdot \prod_{(x,y)\in\mathcal{D}_t} \mathbb{1}\{f_m(x) = y\}}{q_t(f_m)}. \tag{12}$$

Programs inconsistent with the observed data $\mathcal{D}_t$ receive zero weight. For consistent programs, $p^0(f_m)/q_t(f_m)$ corrects for the mismatch between prior and proposal. Because q_t is already conditioned on $\mathcal{D}_t$, most samples should be consistent, and the weights should have low variance. At round 0 the weights reduce to the consistency indicator, since $p^0 = q_0$. Density ratios enter only when particles are resampled from a later q_t. Both densities are then the probability the LM assigns to the extracted code block under the respective prompt, without the reasoning that precedes it and without the sampling temperature. The weights are therefore an approximation to Eq. (12), which we do because intermediate reasoning and a tuned sampling temperature improve the sampled programs (App. E.5).

Pushforward mass. The pushforward mass on a class c is a posterior expectation, $p_\sim(c \mid \mathcal{D}_t) = \mathbb{E}_{f\sim p(\cdot\mid\mathcal{D}_t)}[\mathbb{1}\{f \in c\}]$, which the self-normalized importance sampling (SNIS) estimator (Owen, 2013) replaces by the weighted share of the particles in the class,

$$\widehat{p}_\sim(c \mid \mathcal{D}_t) = \frac{\sum_{m:\, f_m\in c} w_m}{\sum_{m=1}^{M} w_m}, \tag{13}$$

where class membership is checked by comparing outputs on S, as two programs share a $\sim$-class if and only if they agree on every $x \in S$.

Entropy and information gain. Plugging Eq. (13) into the definition of entropy gives the plug-in estimator

$$\widehat{\mathrm{H}}[p_\sim(\cdot \mid \mathcal{D}_t)] = -\sum_c \widehat{p}_\sim(c \mid \mathcal{D}_t) \log_2 \widehat{p}_\sim(c \mid \mathcal{D}_t). \tag{14}$$

The information gain for a candidate query x is estimated by partitioning the particles by predicted output $f_m(x)$, computing $\widehat{\mathrm{H}}$ on each partition with renormalized weights, and weighting by the estimated posterior predictive

$$\widehat{p}(y \mid x, \mathcal{D}_t) = \frac{\sum_{m:\, f_m(x)=y} w_m}{\sum_{m=1}^{M} w_m}. \tag{15}$$

The greedy step of §2.4 is a finite maximization over S, with $\widehat{\mathrm{H}}$ in place of H and $\widehat{p}$ in place of p. Alg. 1 summarizes the resulting procedure, with an effective-sample-size test that redraws the particles from the current proposal when the weights degenerate.

Approximation gaps. Under absolute continuity of the posterior with respect to q_t (i.e., $p(f \mid \mathcal{D}_t) > 0 \implies q_t(f) > 0$), the SNIS estimates $\widehat{p}_\sim(c \mid \mathcal{D}_t)$ converge to $p_\sim(c \mid \mathcal{D}_t)$ as $M \to \infty$ (e.g., Owen, 2013), and since $\widehat{\mathrm{H}}$ is continuous in these estimates, $\widehat{\mathrm{H}}[p_\sim(\cdot \mid \mathcal{D}_t)] \to \mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_t)]$ by the continuous mapping theorem. The guarantees of §3 are stated for the exact posterior, and for finite M two things can go wrong. The particles may miss $[f^\star]_\sim$ entirely, or a wrong class may exceed the threshold before enough of them land in $[f^\star]_\sim$. App. E.4 varies the particle budget, and Tab. 3 and App. B quantify how many runs terminate on a wrong class or without a consistent program.

Algorithm 1 Greedy Information Gain Program Induction

Require: Input distribution μ; program prior p^0; LM proposal q_t; oracle for $f^\star$; evaluation size N; particle budget M; ESS threshold τ_{ESS}.

Draw $S = \{x_1, \dots, x_N\} \overset{\text{i.i.d.}}{\sim} \mu$. ▷*Fix evaluation set*
Sample $f_1, \dots, f_M \sim q_0$; compute w_m via Eq. (12). ▷*Initialize particles*
$t \leftarrow 0$; $\mathcal{D}_0 \leftarrow$ seed examples.
while no class c has $\widehat{p}_\sim(c \mid \mathcal{D}_t) > \frac{1}{2}$:
  $t \leftarrow t + 1$.
  $x^{(t)} \leftarrow \operatorname{argmax}_{x \in S} \widehat{\mathrm{H}}[p_\sim(\cdot \mid \mathcal{D}_{t-1})] - \sum_y \widehat{p}(y \mid x, \mathcal{D}_{t-1})\, \widehat{\mathrm{H}}[p_\sim(\cdot \mid \mathcal{D}_{t-1} \cup \{(x, y)\})]$.
  Observe $y^{(t)} \leftarrow f^\star(x^{(t)})$; $\mathcal{D}_t \leftarrow \mathcal{D}_{t-1} \cup \{(x^{(t)}, y^{(t)})\}$. ▷*Query oracle*
  for $m = 1, \dots, M$:
    if $f_m(x^{(t)}) \neq y^{(t)}$: $w_m \leftarrow 0$
  Renormalize weights. ▷*Eliminate inconsistent particles*
  if $\mathrm{ESS} = \left(\sum_m w_m\right)^2 / \sum_m w_m^2 < \tau_{\text{ESS}}$:
    Resample $f_1, \dots, f_M \sim q_t$; recompute weights.
return $\widehat{f}$ in the majority class $\widehat{c} = \operatorname{argmax}_c \widehat{p}_\sim(c \mid \mathcal{D}_t)$.

5 Experiments

We now evaluate the practical implications of our work on a programming induction benchmark in Python (CodeARC; Wei et al., 2025). Its reference method, which we call the CodeARC LM agent, prompts a language model to autonomously choose the inputs to query, propose an implementation, and terminate when it feels confident, with no posterior over programs and no criterion for the choice of inputs. Our experiments aim to address three questions.

RQ1. Does Alg. 1 improve accuracy and oracle cost over the CodeARC LM agent?
RQ2. Does EIG ask queries with more guaranteed information than other selection criteria, and how much do the queries add over the prior alone?
RQ3. How does the choice of evaluation set size N trade off convergence and correctness, as hypothesized in §3?

5.1 Setup

Dataset. We evaluate the proposed approach on the inductive inference benchmark CodeARC (Wei et al., 2025), whose 1114 Python functions are drawn from HumanEval+, MBPP+, and APPS and range from 2 to 74 lines, from entry-level tasks to competition problems. Similarly to our setup, the benchmark simulates an inductive inference scenario where a language modeling agent needs to find diverse Python functions from input–output pairs through membership queries (MQ), which query $f^\star$ on a chosen input x to obtain $f^\star(x)$, and equivalence queries (EQ), which ask the oracle whether a candidate $\widehat{f}$ equals $f^\star$ and return a counterexample if it does not, under a fixed MQ and EQ invocation budget.[1] We use the *anonymized* split, which replaces all function names with a generic identifier and so suppresses any cues for the program prior. Following Wei et al. (2025), we provide a small number of *seed invocations* (initial input–output examples that we also use to condition μ), evaluating both 1 and 10 seed invocations, and set the query budget to 30 input–output pairs, which includes the seed invocations. For consistency with their results, the CodeARC agent may additionally issue 2 equivalence queries, while our algorithm makes none and submits its returned program once for evaluation. We evaluate on the full split and report means over 5 independent seed runs. For the models that overlap with Wei et al. (2025), we reproduce their reported results within seed noise.

Models. We evaluate our algorithm on six models of various sizes: Llama3.2-3B and Llama3.1-8B (Llama Team, 2024), Qwen2.5-7B and Qwen2.5-32B (Yang et al., 2025), Olmo3-7B (Team Olmo et al., 2025), and Ministral-8B (Mistral AI team, 2024). We use the instruction-tuned variant of every model. We chose these models for their differing performance on the CodeARC benchmark. We provide all used prompts in App. F.

[1] We note that our setup is equivalent to the CodeARC setup if the EQ budget is 1.

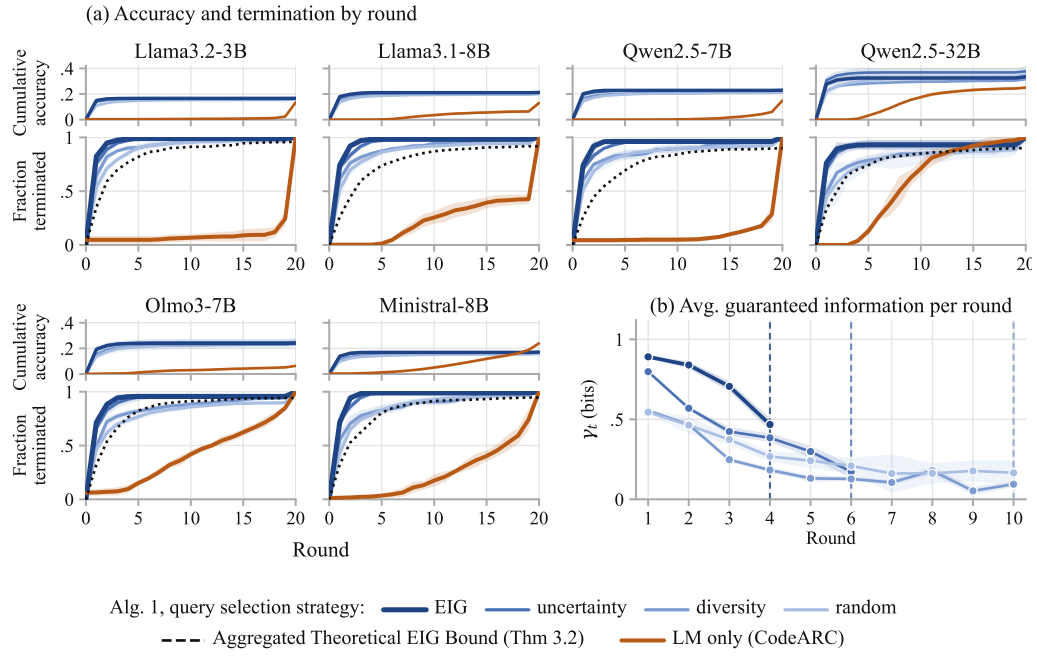

Figure 1: (a) Cumulative accuracy (top) and fraction of runs terminated (bottom) by oracle querying round, for Alg. 1 under four query selection strategies, the CodeARC agent (LM only), and the per-instance bound of Thm. 3.2 aggregated over the EIG runs. The termination panels use the runs whose initial particles contain the target's class. A run is terminated once it reaches the total query budget. (b) The information the selected query is guaranteed to yield, γ_t, by round, averaged over the runs still querying and pooled across models. A dashed line indicates the last round in which at least 200 runs of that strategy are still querying, which is where the strategy has typically terminated. We show 95% confidence intervals over five seeds.

Baselines and Evaluation. As we want to understand whether our algorithm can *efficiently* converge on the target program using our information gain criterion, we compare our algorithm to the full CodeARC baselines in terms of average number of input–output examples observed and accuracy upon termination. As noted before, CodeARC equivalence queries give the baseline strictly more oracle access than our method when the EQ budget exceeds 1, meaning our comparisons against CodeARC @ 2 EQ are against a more powerful oracle setting. Following Wei et al. (2025), we evaluate the correctness of the generated programs extrinsically using the differential testing tool MOKAV (Etemadi et al., 2025). We fix MOKAV's test-generating language model to Qwen2.5-32B—the best-performing model within the fleet—for all evaluated models and both methods, so that accuracies are comparable across models.

5.2 RESULTS

RQ1: Alg. 1 with EIG improves accuracy and query efficiency over the LM only agent. Tab. 1 shows that the proposed method improves over the LM only (CodeARC) agent in most of the evaluated settings in terms of the final accuracy (up to $+.22$) as well as oracle querying efficiency (up to 10.5 fewer input–output examples). Although the accuracy improvements are significant, we note that there is still a notable gap to perfect accuracy, which we attribute to the models' general performance ceiling in proposing valid code matching the shown input–output pairs (cf. Wei et al. (2025), and our failure mode analysis in Tab. 3). Aside from this, the method commonly struggles with early termination due to early posterior concentration, which is the case Thm. 3.2 excludes, a heavier class that stays consistent with every answer. We show in our ablation in Fig. 4 that this can, to some degree, be combated through the choice of proposal prompt and temperature.

Table 1: Results across models and methods on the full anonymized CodeARC split. Entries are means over 5 independent seeds with 95% confidence intervals over seeds. Avg I/O is the average number of input–output examples observed, including the seed invocations, out of a budget of 30.

		Seed Invocations			
		1		10	
Model	Method	Avg I/O ↓	Acc. ↑	Avg I/O ↓	Acc. ↑
Llama3.2-3B	LM only (CodeARC)	19.02 ±0.44	.134 ±.014	27.55 ±0.16	.125 ±.014
	Alg. 1 with EIG (ours)	**15.82** ±0.36	**.166** ±.010	**25.13** ±0.14	**.176** ±.008
Llama3.1-8B	LM only (CodeARC)	16.60 ±0.14	.129 ±.015	26.86 ±0.14	.142 ±.009
	Alg. 1 with EIG (ours)	**12.89** ±0.55	**.212** ±.011	**22.77** ±0.20	**.273** ±.015
Qwen2.5-7B	LM only (CodeARC)	19.54 ±0.26	.148 ±.010	27.88 ±0.19	.156 ±.015
	Alg. 1 with EIG (ours)	**9.01** ±0.42	**.229** ±.008	**20.92** ±0.11	**.322** ±.017
Qwen2.5-32B	LM only (CodeARC)	11.13 ±0.06	.250 ±.011	**18.75** ±0.08	.299 ±.014
	Alg. 1 with EIG (ours)	**10.83** ±0.32	**.334** ±.024	19.66 ±0.20	**.415** ±.010
Olmo3-7B	LM only (CodeARC)	13.34 ±0.16	.064 ±.007	21.42 ±0.15	.113 ±.016
	Alg. 1 with EIG (ours)	**5.90** ±0.33	**.240** ±.014	**19.62** ±0.17	**.332** ±.013
Ministral-8B	LM only (CodeARC)	16.22 ±0.12	**.239** ±.013	**22.91** ±0.18	**.264** ±.013
	Alg. 1 with EIG (ours)	**13.54** ±0.28	.170 ±.010	23.65 ±0.24	.226 ±.012

RQ2: EIG asks the most informative queries and terminates in the fewest rounds, at an accuracy within noise of the other criteria. The termination panels of Fig. 1 condition on runs whose initial particles contain the target's class, for all strategies alike. Compared to the other selection criteria within the proposed algorithm, EIG selects the most informative queries, .70 to .99 guaranteed bits per query against .31 to .52 for diversity and random selection, terminates in the fewest rounds, and reaches an accuracy within the seed intervals of the best criterion for every model (Fig. 1, Tab. 5). The queries yield a significant improvement over two controls, prior majority, which returns a program from the majority class of the initial particles without any query, and the CodeARC agent, which selects its queries heuristically (Tabs. 1 and 5). Tab. 4 reports the bound of Thm. 3.2 on the EIG runs, which is attained in .37 of them.

RQ3: Larger S makes convergence on the target's class rarer, though more reliable. Fig. 2 presents an ablation of the evaluation set size N from 2 to 256 for three models. As predicted in §3, a smaller N makes $[f^\star]_\sim$ coarse, since equivalence then requires agreement on only a few inputs. The posterior concentrates early on such a coarse class, and the program returned from it is often rejected by the oracle despite agreeing with $f^\star$ on S. As S grows, convergence on $[f^\star]_\sim$ becomes rarer but a reliable indicator of correctness. The rising conditional accuracy instantiates Prop. 2.1, since agreement with $f^\star$ on more sampled inputs means fewer disagreements on unseen ones.

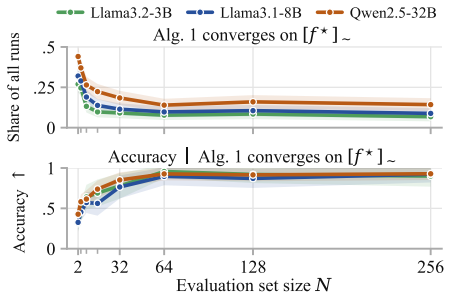

Figure 2: N ablation, three models. Top: share of runs converging on $[f^\star]_\sim$. Bottom: accuracy given convergence. 95% CIs over runs, three seeds.

6 Conclusion

We presented a Bayesian active-learning approach for efficient program induction with oracle feedback that uses a language model as both the program prior and the importance-sampling proposal. We proved that the greedy information-gain algorithm terminates after $\mathcal{O}(\log 1/p^0(f^\star))$ informative oracle invocations, and introduced an equivalence relation that yields a PAC certificate on population error. Empirically, the algorithm improves accuracy and reduces oracle calls over the CodeARC agent in most evaluated settings, and its queries improve over returning the prior's majority class. We see this work as a step toward principled human-in-the-loop program induction, where the same framework can accommodate richer oracle models, such as noisy users, and structured priors.

Reproducibility statement

All theoretical results are stated with their assumptions in §§ 2.2 and 3, and their proofs are given in App. C. The experimental setup is described in §5, and App. D lists the run configuration, including the particle budget, the evaluation set size, the query budget, the termination threshold and the model used for correctness evaluation. Appendices E.5 and F give the prompts and the range of sampling temperatures. The main results are means over 5 independent runs on the full anonymized CodeARC split, with the same evaluation sets shared across models, and the ablations in App. E are generally run on three seeds. The supplementary material contains the implementation of Alg. 1, the run scripts, and the analysis notebook that produces every figure and table.

AI use statement

We used generative AI tools for brainstorming, to polish writing and mathematical proofs, and to find some of the related work. We used these tools in compliance with the ICLR 2027 policy and reviewed all AI-generated text and retrieved resources. We take full responsibility for the content of the paper.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. HySynth: Context-free LLM approximation for guiding program synthesis. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Celeste Barnaby, Qiaochu Chen, Ramya Ramalingam, Osbert Bastani, and Işıl Dillig. Active learning for neurosymbolic program synthesis. *Proceedings of the ACM on Programming Languages*, 9 (OOPSLA2), 2025.

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Information Processing Letters*, 24(6):377–380, 1987.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Sanjoy Dasgupta. Analysis of a greedy active learning strategy. In *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2004.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 835–850. ACM, 2021.

Kevin Ellis, Adam Albright, Armando Solar-Lezama, Joshua B. Tenenbaum, and Timothy J. O'Donnell. Synthesizing theories of human language with Bayesian program induction. *Nature Communications*, 13:5024, 2022.

Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. Mokav: Execution-driven differential testing with llms. *Journal of Systems and Software*, 230:112571, December 2025. ISSN 0164-1212. doi: 10.1016/j.jss.2025.112571. URL http://dx.doi.org/10.1016/j.jss.2025.112571.

Daniel Golovin and Andreas Krause. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *Journal of Artificial Intelligence Research*, 42:427–486, 2011.

Daniel Golovin, Andreas Krause, and Debajyoti Ray. Near-optimal Bayesian active learning with noisy observations. In *Advances in Neural Information Processing Systems*, volume 23, 2010.

Gabriel Grand, Valerio Pepe, Joshua B. Tenenbaum, and Jacob Andreas. Shoot first, ask questions later? building rational agents that explore and act like people. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=EQhUvWH78U.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pp. 317–330, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926423. URL https://doi.org/10.1145/1926385.1926423.

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1–2):1–119, 2017.

Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54:693–726, 2017.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 215–224, 2010.

Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1143–1158, 2020.

Katarzyna Kobalczyk, Nicolas Astorga, Tennison Liu, and Mihaela van der Schaar. Active task disambiguation with LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=JAMxRSXLFz.

Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by LLMs? In *Advances in Neural Information Processing Systems*, 2024.

Yixuan Li, Julian Parsert, and Elizabeth Polgreen. Guiding enumerative program synthesis with large language models. In *Computer Aided Verification: 36th International Conference*, volume 14682 of *Lecture Notes in Computer Science*. Springer, 2024.

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, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Henry Lieberman (ed.). *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, CA, 2001.

Dennis V. Lindley. On a measure of the information provided by an experiment. *Annals of Mathematical Statistics*, 27:986–1005, 1956. doi: 10.1214/aoms/1177728069.

Llama Team. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

David J. C. MacKay. Information-based objective functions for active data selection. *Neural Computation*, 4(4):590–604, 1992. doi: 10.1162/neco.1992.4.4.590.

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015.

Mistral AI team. Un Ministral, des Ministraux. Mistral AI blog, October 2024. URL https://mistral.ai/news/ministraux.

Robert D. Nowak. The geometry of generalized binary search. *IEEE Transactions on Information Theory*, 57(12):7893–7906, 2011.

Art B. Owen. *Monte Carlo theory, methods and examples*. https://artowen.su.domains/mc/, 2013.

Wasu Top Piriyakulkij, Cassidy Langenfeld, Tuan Anh Le, and Kevin Ellis. Doing experiments and revising rules with natural language and probabilistic reasoning. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

H. Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. URL https://api.semanticscholar.org/CorpusID:120980829.

Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, January 2019. ISSN 2475-1421. doi: 10.1145/3290350. URL http://dx.doi.org/10.1145/3290350.

Burr Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. URL https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/. https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/.

Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3533–3546, 2022.

Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

R.J. Solomonoff. A formal theory of inductive inference. part i. *Information and Control*, 7(1): 1–22, 1964. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(64)90223-2. URL https://www.sciencedirect.com/science/article/pii/S0019995864902232.

Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM*, 24(1):161–175, 1977.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, Jacob Morrison, Jake Poznanski, Kyle Lo, Luca Soldaini, Matt Jordan, Mayee Chen, Michael Noukhovitch, Nathan Lambert, Pete Walsh, Pradeep Dasigi, Robert Berry, Saumya Malik, Saurabh Shah, Scott Geng, Shane Arora, Shashank Gupta, Taira Anderson, Teng Xiao, Tyler Murray, Tyler Romero, Victoria Graf, Akari Asai, Akshita Bhagia, Alexander Wettig, Alisa Liu, Aman Rangapur, Chloe Anastasiades, Costa Huang, Dustin Schwenk, Harsh Trivedi, Ian Magnusson, Jaron Lochner, Jiacheng Liu, Lester James V. Miranda, Maarten Sap, Malia Morgan, Michael Schmitz, Michal Guerquin, Michael Wilson, Regan Huff, Ronan Le Bras, Rui Xin, Rulin Shao, Sam Skjonsberg, Shannon Zejiang Shen, Shuyue Stella Li, Tucker Wilde, Valentina Pyatkin, Will Merrill, Yapei Chang, Yuling Gu, Zhiyuan Zeng, Ashish Sabharwal, Luke Zettlemoyer, Pang Wei Koh, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. Olmo 3, 2025. URL https://arxiv.org/abs/2512.13961.

Anjiang Wei, Tarun Suresh, Jiannan Cao, Naveen Kannan, Yuheng Wu, Kai Yan, Thiago S. F. X. Teixeira, Ke Wang, and Alex Aiken. CodeARC: Benchmarking reasoning capabilities of LLM agents for inductive program synthesis. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Q5pVZCrrKr.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

A Related Work

Interactive and oracle-guided program synthesis. Early works in oracle-guided program synthesis include CEGIS (Solar-Lezama, 2008), and is formalized in works by Jha et al. (2010) and Jha & Seshia (2017), where an automated verifier supplies counterexamples in place of a human. Interactive disambiguation with human oracles was deployed in FlashFill by Mayer et al. (2015) and studied theoretically by Ji et al. (2020) via version-space sampling. All of these operate over a finite DSL and use heuristic or symbolic distinguishability criteria. We work over a countably infinite program space and derive query-complexity bounds directly from the program prior.

Active learning for program synthesis. Barnaby et al. (2025) discuss active learning in the neurosymbolic setting, where neural components can mispredict and cause convergence failures. Their constrained conformal evaluation is complementary to our work, which assumes a symbolic target and bounds query complexity information-theoretically. Our query-selection criterion is an instance of Bayesian experimental design (Lindley, 1956; MacKay, 1992), applied sequentially as in active learning (Settles, 2012). Dasgupta (2004) showed that greedy information-gain maximization over a *finite* class uses at most $4\ln(1/\min_h p^0(h))$ times the expected number of queries of the optimal strategy. We bound the number of queries of the greedy rule itself, for programs, whose space is countable, by measuring agreement on a finite sample of inputs, and replace enumeration with importance sampling under an LM proposal. Our query bounds depend on the size of the evaluation set, because a larger set separates more programs into distinct classes. A finite hypothesis class has no such dependence. Golovin et al. (2010), Golovin & Krause (2011), and Nowak (2011) show that greedy rules over finite classes use nearly as few queries as the best possible adaptive plan of queries, which is intractable to compute. We bound the number of queries directly and make no such theoretical comparison.

Bayesian program induction. Bayesian priors over programs underlie a long line of work. Lake et al. (2015) used hand-designed generative models over strokes for few-shot concept learning, Ellis et al. (2021) bootstrapped reusable libraries via wake-sleep inference, Ellis et al. (2022) applied the same framework to linguistic theory synthesis, and Saad et al. (2019) sampled probabilistic programs from PCFG posteriors. These systems trade expressivity for tractability through a hand-crafted DSL or generative model. We keep the Bayesian perspective but substitute a language model for

Table 2: Computational cost per problem with 1 seed invocation on the full anonymized CodeARC split, means over runs. LM calls are generation requests to the language model. Programs are the candidate programs sampled by Alg. 1 and the implementations submitted by the agent. Wall-clock times were measured on different clusters under concurrent load and are indicative.

Model	Method	Wall-clock (s) ↓	LM calls ↓	Programs
Llama3.2 3B	Alg. 1 with EIG (ours)	102	33.1	112.9
	LM only (CodeARC)	289	4.7	2.6
Llama3.1 8B	Alg. 1 with EIG (ours)	184	49.8	110.9
	LM only (CodeARC)	330	7.1	2.8
Qwen2.5 7B	Alg. 1 with EIG (ours)	142	72.2	106.4
	LM only (CodeARC)	299	7.4	2.5
Qwen2.5 32B	Alg. 1 with EIG (ours)	244	77.8	105.1
	LM only (CodeARC)	991	9.3	2.5
Olmo3 7B	Alg. 1 with EIG (ours)	258	72.0	108.6
	LM only (CodeARC)	350	10.1	0.9
Ministral 8B	Alg. 1 with EIG (ours)	182	32.1	115.5
	LM only (CodeARC)	410	6.6	2.6

the hand-crafted prior, using importance sampling to make inference tractable over an unrestricted program space. Piriyakulkij et al. (2024) likewise combine LM-proposed hypotheses with Monte Carlo belief updates and information-theoretic experiment selection, for natural-language rules on a Zendo-style task.

LLMs for program synthesis. LLMs achieve strong results on natural-language-specified code generation (Chen et al., 2021; Austin et al., 2021; Li et al., 2022) but struggle with programming by example without domain-specific fine-tuning (Li & Ellis, 2024). HySynth (Barke et al., 2024) extracts a task-specific PCFG from LLM samples to guide bottom-up enumerative search, and Li et al. (2024) pursue a similar direction using syntactic guidance. Both treat the LM as a one-shot prior over a fixed DSL, without user interaction or posterior updating. Grouping sampled programs by their outputs on a set of inputs is used to select a submission in AlphaCode (Li et al., 2022), CodeT (Chen et al., 2023), and MBR-exec (Shi et al., 2022), and our equivalence relation $\sim_S$ is the same construction. Wei et al. (2025) introduced CodeARC, an interactive benchmark where agents iteratively query a hidden oracle and input selection is left to the agent's heuristics. We use the LM as a prior and importance sampling proposal within a Bayesian framework and choose oracle membership queries to maximize expected information gain.

B Limitations

Computational cost. Each round of the algorithm requires potentially drawing M program samples from the LM proposal q_t, evaluating each on the inputs in S, and recomputing importance weights and entropy estimates over the candidate pool. Larger evaluation sets, larger particle budgets, and longer programs all compound this cost, and the per-round latency is dominated by LM inference. Tab. 2 reports the cost per problem. Alg. 1 makes 32 to 78 LM calls and samples about 110 candidate programs, each of which it executes on S and on every query, against 5 to 10 calls for the agent. Its wall-clock time per problem is nevertheless below the agent's in our setup, since the agent's calls are long generations and it keeps querying for most of its budget. The query-efficiency gains we report are savings in oracle effort, and the compute cost is in LM calls and program executions. Finally, the importance weights at resampling require the LM's token log-probabilities of a program under two prompts, which closed APIs may not expose.

Dependence on the input distribution. Prop. 2.1 bounds the error under the input distribution μ, which we instantiate by prompting an LM with the seed examples. This needs few examples. A single seed invocation already yields the gains of Tab. 1, and rebuilding the evaluation set from 1 or 5 seed examples in place of 10 changes accuracy by at most .02 (App. D). What the bound does not guarantee is generalization beyond μ. A differential tester searches for a counterexample and may find one on an input that μ rarely produces, so the accuracies we report also measure how well agreement under μ generalizes. The more the prompt tells the LM about the intended inputs, the better that generalization can be expected to be.

Gap between theoretical and empirical correctness. Thms. 3.1 and 3.2 hold for the exact posterior. The algorithm works with a finite particle approximation of it, and with priors that are sometimes concentrated on a wrong class. Both show up in Tab. 3 as the .34 to .63 of runs that converge on a wrong program, since a wrong class can exceed the threshold before enough particles land in the target's class, or hold most of the prior mass to begin with. The remaining accuracy is lost to the models' capability ceiling. In .13 to .49 of the runs no sampled program is consistent with the observed examples, and the CodeARC agent built on the same models reaches .06 to .30 (Tab. 1). Our experiments already show stronger models narrowing this gap, with Qwen2.5-32B reaching the highest accuracy in every setting.

C Deferred Proofs

C.1 PAC Bound

Proposition 2.1 (PAC Error Bound, cf. Blumer et al. 1987)**.** *Let $S = \{x_1, \dots, x_N\}$ be drawn i.i.d. from μ. With probability at least $1 - \delta$ over the draw of S, every $f \in [f^\star]_\sim$ with $p^0(f) > 0$ satisfies*

$$\mathrm{err}_\mu(f) \le \frac{\ln(1/p^0(f)) + \ln(1/\delta)}{N}. \tag{3}$$

Proof. Fix f with $p^0(f) > 0$. Since S consists of N i.i.d. draws from μ, the probability that f agrees with $f^\star$ on all of S is

$$\Pr_{S\sim\mu^N}\left[f \in [f^\star]_\sim\right] = (1 - \mathrm{err}_\mu(f))^N \le \exp(-N \cdot \mathrm{err}_\mu(f)), \tag{16}$$

where we used $1 - u \le e^{-u}$. Let $\tau(f) \stackrel{\text{def}}{=} (\ln(1/p^0(f)) + \ln(1/\delta))/N$ be the right-hand side of Eq. (3), and let B_f be the event that $f \in [f^\star]_\sim$ and $\mathrm{err}_\mu(f) > \tau(f)$. The event B_f is empty if $\mathrm{err}_\mu(f) \le \tau(f)$. Otherwise Eq. (16) gives $\Pr_{S\sim\mu^N}[B_f] \le \exp(-N \cdot \tau(f)) = \delta \cdot p^0(f)$. The probability that any of these events occurs is at most the sum of their probabilities, so

$$\Pr_{S\sim\mu^N}\Big[\bigcup_{f\colon p^0(f)>0} B_f\Big] \le \sum_{f\colon p^0(f)>0} \delta \cdot p^0(f) \le \delta. \tag{17}$$

On the complement of this event, every $f \in [f^\star]_\sim$ with $p^0(f) > 0$ satisfies Eq. (3). ∎

C.2 Expected Query Complexity and Correctness

Throughout, S is fixed, and for a class c and $x \in S$ we write $c(x)$ for the common output of the programs in c. A class is *consistent* with $\mathcal{D}_t$ if $c(x^{(s)}) = y^{(s)}$ for all $s \le t$, and $[f^\star]_\sim$ is consistent with every $\mathcal{D}_t$. Ties in Eq. (7) are broken by a fixed rule.

Lemma C.1 (Mass recursion)**.** *For every class c and $1 \le t \le T$,*

$$p_\sim(c \mid \mathcal{D}_t) = \frac{p_\sim(c \mid \mathcal{D}_{t-1}) \cdot \mathbb{1}\{c(x^{(t)}) = y^{(t)}\}}{p(y^{(t)} \mid x^{(t)}, \mathcal{D}_{t-1})}, \tag{18}$$

$$\log_2 \frac{1}{p_\sim([f^\star]_\sim \mid \mathcal{D}_0)} = \sum_{s=1}^{t} \log_2 \frac{1}{p(y^{(s)} \mid x^{(s)}, \mathcal{D}_{s-1})} + \log_2 \frac{1}{p_\sim([f^\star]_\sim \mid \mathcal{D}_t)}. \tag{19}$$

Proof. All programs in a class agree on $x^{(t)} \in S$, so summing Eq. (5a) over $g \in c$ gives Eq. (18). For $c = [f^\star]_\sim$ the indicator equals one, and unrolling over $s \le t$ and taking logarithms gives Eq. (19). ∎

Lemma C.2 (Greedy selection)**.** *Let $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)] < \infty$. For every $t \le T$ and $x \in S$, the objective in Eq. (7) equals $\mathrm{H}[p(\cdot \mid x, \mathcal{D}_{t-1})]$. Moreover, $\gamma_t > 0$ for all $t \le T$, no input is queried twice, and $T \le N$.*

Proof. Let $c \sim p_\sim(\cdot \mid \mathcal{D}_{t-1})$, whose entropy is at most $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)]/p_\sim([f^\star]_\sim \mid \mathcal{D}_0) < \infty$. The answer $y = c(x)$ is a function of c with distribution $p(\cdot \mid x, \mathcal{D}_{t-1})$, and by Lem. C.1 the distribution of c given y is $p_\sim(\cdot \mid \mathcal{D}_{t-1} \cup \{(x, y)\})$. The chain rule for entropy gives

$$\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_{t-1})] = \mathrm{H}[p(\cdot \mid x, \mathcal{D}_{t-1})] + \sum_{y \in Y} p(y \mid x, \mathcal{D}_{t-1})\, \mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_{t-1} \cup \{(x, y)\})], \tag{20}$$

which is the first claim. For $t \le T$, no class has mass above $\frac{1}{2}$ under $\mathcal{D}_{t-1}$, so two classes have positive mass and differ on some $x' \in S$. Then $\mathrm{H}[p(\cdot \mid x^{(t)}, \mathcal{D}_{t-1})] \ge \mathrm{H}[p(\cdot \mid x', \mathcal{D}_{t-1})] > 0$, the predictive at $x^{(t)}$ is not a point mass, and $\gamma_t > 0$. The predictive at a previously queried input is a point mass, so no input is queried twice. Once all of S is queried, $[f^\star]_\sim$ is the only consistent class, and thus $T \le N$. ∎

Lemma C.3 (Posterior as conditional law)**.** *Let $f^\star \sim p^0$. For every class c and every $t \le T$, we have $\Pr[[f^\star]_\sim = c \mid \mathcal{D}_t] = p_\sim(c \mid \mathcal{D}_t)$.*

Proof. The queries are a deterministic function of the answers received so far. A target in class c therefore generates $\mathcal{D}_t$ if and only if c is consistent with it, that is, $\Pr[\mathcal{D}_t \mid [f^\star]_\sim = c] = \mathbb{1}\{c \text{ consistent with } \mathcal{D}_t\}$. Bayes' rule with $\Pr[[f^\star]_\sim = c] = p_\sim(c \mid \mathcal{D}_0)$ gives the claim. ∎

Theorem 3.1 (Expected query complexity and correctness)**.** *Let $f^\star \sim p^0$ and let $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)] < \infty$ be the prior class entropy in bits. Write $\bar\gamma_t \stackrel{\text{def}}{=} \max_{x \in S} -\log_2 \max_y p(y \mid x, \mathcal{D}_{t-1})$ for the most information an answer to a query in S is guaranteed to carry at round t. The greedy strategy of §2.4 satisfies*

(i) $\mathbb{E}_{f^\star \sim p^0}\big[\sum_{t=1}^{T} \bar\gamma_t\big] \le \mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)]$;
(ii) $T \le N$ *for every target;*
(iii) if $\bar\gamma_t \ge \gamma$ *for all* $t \le T$ *and every* $f^\star$ *with* $p^0(f^\star) > 0$*, then* $\mathbb{E}_{f^\star \sim p^0}[T] \le \mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)]/\gamma$.

At termination, $\Pr[\widehat{c} = [f^\star]_\sim \mid \mathcal{D}_T] = p_\sim(\widehat{c} \mid \mathcal{D}_T) > \frac{1}{2}$*, i.e., for* $f^\star \sim p^0$ *the terminal mass is the exact probability that the returned program is* $\sim_S$*-equivalent to* $f^\star$.

Proof. Claim (ii) follows from Lem. C.2. For (i), dropping the last term of Eq. (19) at $t = T$, which is non-negative, gives $\sum_{t=1}^{T} -\log_2 p(y^{(t)} \mid x^{(t)}, \mathcal{D}_{t-1}) \le -\log_2 p_\sim([f^\star]_\sim \mid \mathcal{D}_0)$. We take expectations over $f^\star \sim p^0$. The right-hand side becomes $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)]$ since $\Pr[[f^\star]_\sim = c] = p_\sim(c \mid \mathcal{D}_0)$. On the left-hand side, the event $\{T \ge t\}$ is determined by $\mathcal{D}_{t-1}$, and by Lem. C.3 the answer $y^{(t)}$ given $\mathcal{D}_{t-1}$ is distributed as $p(\cdot \mid x^{(t)}, \mathcal{D}_{t-1})$, so that the conditional expectation of the t-th term given $\mathcal{D}_{t-1}$ is $\mathrm{H}[p(\cdot \mid x^{(t)}, \mathcal{D}_{t-1})]$. Summing over $t \le N$ yields

$$\mathop{\mathbb{E}}_{f^\star \sim p^0}\Big[\sum_{t=1}^{T} \mathrm{H}[p(\cdot \mid x^{(t)}, \mathcal{D}_{t-1})]\Big] \le \mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)]. \tag{21}$$

By Lem. C.2 and because the Shannon entropy of a distribution is at least the negative logarithm of its largest probability, $\mathrm{H}[p(\cdot \mid x^{(t)}, \mathcal{D}_{t-1})] = \max_{x \in S} \mathrm{H}[p(\cdot \mid x, \mathcal{D}_{t-1})] \ge \bar\gamma_t$, which proves (i). Keeping the last term of Eq. (19) and applying Lem. C.3 once more shows that the left-hand side equals $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)] - \mathbb{E}[\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_T)]]$, so the only inequalities behind (i) are $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_T)] \ge 0$ and this lower bound on the Shannon entropy. For (iii), the hypothesis gives $\gamma T \le \sum_{t=1}^{T} \bar\gamma_t$ for every $f^\star$ with $p^0(f^\star) > 0$, and taking expectations and applying (i) gives the bound. Finally, the returned class $\widehat{c}$ is determined by $\mathcal{D}_T$, so Lem. C.3 gives $\Pr[\widehat{c} = [f^\star]_\sim \mid \mathcal{D}_T] = p_\sim(\widehat{c} \mid \mathcal{D}_T)$, which exceeds $\frac{1}{2}$ by the stopping rule. ∎

C.3 Per-Instance Query Complexity and Correctness

Lemma C.4 (Mass growth)**.** *Fix $f^\star$ with $p^0(f^\star) > 0$. Under any rule that selects queries from S, for every $t \le T$,*

$$p_\sim\big([f^\star]_\sim \mid \mathcal{D}_t\big) \ge p_\sim([f^\star]_\sim \mid \mathcal{D}_0) \cdot 2^{\sum_{s=1}^{t} \gamma_s}, \tag{22}$$

and under the greedy rule $\gamma_s > 0$ for every s.

Proof. By definition of γ_s, we have $p(y^{(s)} \mid x^{(s)}, \mathcal{D}_{s-1}) \le 2^{-\gamma_s}$, so Eq. (19) gives $-\log_2 p_\sim([f^\star]_\sim \mid \mathcal{D}_0) \ge \sum_{s=1}^{t} \gamma_s - \log_2 p_\sim([f^\star]_\sim \mid \mathcal{D}_t)$, which rearranges to Eq. (22). The greedy rule gives $\gamma_s > 0$ by Lem. C.2. ■

Theorem 3.2 (Per-instance query complexity and correctness)**.** *Fix $f^\star$ with $p^0(f^\star) > 0$ and let $\mathrm{H}[p_\sim(\cdot \mid \mathcal{D}_0)] < \infty$. Write $\gamma_s \stackrel{\text{def}}{=} -\log_2 \max_y p(y \mid x^{(s)}, \mathcal{D}_{s-1})$ for the information the answer to the query of round s is guaranteed to carry. If $p_\sim([f^\star]_\sim \mid \mathcal{D}_0) \le \frac{1}{2}$ and $\gamma_s \ge \gamma$ for all $s \le T$, then any selection rule over S terminates after*

$$T \le \left\lfloor \frac{1}{\gamma} \log_2 \frac{1}{2p_\sim([f^\star]_\sim \mid \mathcal{D}_0)} \right\rfloor + 1. \tag{8}$$

Moreover, $\widehat{c} \neq [f^\star]_\sim$ if and only if some class with prior mass strictly greater than $p_\sim([f^\star]_\sim \mid \mathcal{D}_0)$ is consistent with every query made. In particular, if no class outweighs $[f^\star]_\sim$ under the prior, termination returns $\widehat{c} = [f^\star]_\sim$, and under the hypotheses of Eq. (8) it does so within that bound.

Proof. If $\sum_{s \le t} \gamma_s > \log_2 \frac{1}{2p_\sim([f^\star]_\sim \mid \mathcal{D}_0)}$, then Lem. C.4 gives $p_\sim([f^\star]_\sim \mid \mathcal{D}_t) > \frac{1}{2}$ and the algorithm has terminated by round t. Now let t be the right-hand side of Eq. (8), which is the smallest integer with $t\gamma > \log_2 \frac{1}{2p_\sim([f^\star]_\sim \mid \mathcal{D}_0)}$. If $T > t$, then $\sum_{s \le t} \gamma_s \ge t\gamma$ and the algorithm would have terminated by round t, a contradiction. For correctness, by Eq. (4) the posterior mass of every class consistent with $\mathcal{D}_T$ is its prior mass divided by the same normalizer, so consistent classes are ordered under the posterior as under the prior. If $\widehat{c} \neq [f^\star]_\sim$, then $p_\sim(\widehat{c} \mid \mathcal{D}_T) > \frac{1}{2} > p_\sim([f^\star]_\sim \mid \mathcal{D}_T)$ with both classes consistent, so $\widehat{c}$ has strictly larger prior mass. Conversely, if a consistent class c' has larger prior mass than $[f^\star]_\sim$, then $p_\sim(c' \mid \mathcal{D}_T) > p_\sim([f^\star]_\sim \mid \mathcal{D}_T)$, and $\widehat{c} = [f^\star]_\sim$ would give a total mass above one. ■

D Experimental Details

Infrastructure. All models are served using vLLM[2], with tensor parallelism and GPU memory scaled to model size. Namely, models up to 8B parameters were run on a single GPU, while 32B models used either four 24 GB GPUs in parallel or a single 96 GB GPU. All servers run vLLM's batch-invariant kernels, so that seeded sampling is reproducible under concurrent load. Models are loaded in bfloat16. The algorithm's sampling and query-selection loop runs on a separate CPU-only node and communicates with the model server over an SSH tunnel.

Model parameters. We use $M = 64$ particles, an evaluation set of size $N = 64$, resample when the effective sample size falls below $\tau_{\mathrm{ESS}} = 4$, and terminate once a class holds more than half of the posterior mass. When no sampled program is consistent with the observed examples, the algorithm resamples once from the proposal, and if still no program is consistent it returns none, which is counted as incorrect and charged the full budget. Runs that return a program observe 2.3 to 2.7 input–output examples on average, depending on the model. The query budget we use and report is 30 input–output pairs, which includes the seed invocations. The evaluation set is shared across models. For each problem, we draw 256 distinct inputs once from the input distribution μ, instantiated with Llama3.1-8B at temperature 1.1, conditioned on all 10 seed invocations of the benchmark, and use a fixed random subsample of size N. The evaluation set therefore draws on seed examples that the algorithm does not see at 1 seed invocation. Its inputs coincide with a seed input in 0.3% of the entries, and rebuilding it from the 1 or 5 visible seed examples changed the accuracy of Llama3.2-3B, Llama3.1-8B, and Qwen2.5-32B by at most .02 on 100 problems, so we keep the shared evaluation set for all settings. The guarantee of Prop. 2.1 is with respect to this input distribution. The proposal prompt and sampling temperature are selected per model on a screening set (App. E.5, Fig. 4), and the prompts are given in App. F. The maximum context length is capped at 12288 tokens across all models.

Licensing. The CodeARC benchmark (Wei et al., 2025) is released under the Apache 2.0 license. Llama3.1-8B and Llama3.2-3B are released under the Llama 3.1 and Llama 3.2 Community Licenses respectively. Qwen2.5-7B and Qwen2.5-32B are released under the Apache 2.0 license. Olmo3-7B is released under the Apache 2.0 license, and Ministral-8B under the Mistral Research License.

[2]https://docs.vllm.ai/en/

Table 3: Terminal state of the runs behind Tab. 1 at 1 seed invocation ($N = 64$, five seeds), as a share of all runs, averaged over seeds as in Tab. 1. A run converges when one class exceeds half of the posterior mass. The five rows partition the runs, and the two rows with a correct program sum to the accuracy of Tab. 1.

	Llama3.2-3B	Llama3.1-8B	Qwen2.5-7B	Qwen2.5-32B	Olmo3-7B	Ministral-8B
Not converged, no proposal consistent with the oracle's answers	.49	.38	.24	.30	.13	.40
Not converged, query budget exhausted	.00	.00	.00	.01	.01	.00
Not converged, program correct	.00	.00	.00	.01	.00	.00
Converged, program wrong	.34	.40	.52	.36	.63	.42
Converged, program correct	.17	.21	.23	.33	.24	.17
Accuracy over all runs	.17	.21	.23	.33	.24	.17

Table 4: Termination round T of EIG against its per-instance bound derived in Thm. 3.2, $\lfloor \frac{1}{\gamma} \log_2 \frac{1}{2p_\sim([f^\star]_\sim \mid \mathcal{D}_0)} \rfloor + 1$. Runs on the diagonal, .37 of all runs, attain the bound.

	Bound					
T	1	2	3	4	5	≥ 6
1	777	387	182	108	93	331
2	–	106	112	60	46	94
3	–	–	19	28	18	40
4	–	–	–	1	3	18
5	–	–	–	–	–	3

E Additional Results

E.1 Failure Modes

Tab. 3 decomposes the runs of Tab. 1 by their terminal state, which locates the gap between the correctness guarantee of Thm. 3.1 and the accuracies of Tab. 1. Most runs fail before the stopping rule has anything to decide. In a large share of them, between .13 and .49 depending on the model, no sampled program is consistent with the oracle's answers, so the posterior is empty. In most of the remaining runs the posterior does converge, but on a wrong program, owing to the proposal placing its mass on a plausible near-miss that survives the few queries asked. Runs that keep querying until the budget is exhausted are rare, so the query budget is not what limits accuracy. Both failure modes are properties of the proposal, which is why the particle budget (App. E.4) and the prompt (App. E.5) move accuracy more than the choice of query selection strategy does (Tab. 5).

E.2 Per-Instance Bound Analysis

Tab. 4 compares, run by run, the termination round of EIG with the bound of Thm. 3.2 evaluated on that run. The bound is attained in .37 of the runs, and is loose when the guaranteed information is not uniform over the queries of a run, since γ is the minimum over the run's queries and a single uninformative query loosens the bound.

E.3 Query Selection Strategies

Tab. 5 reports, per model, the quantities of Fig. 1 at the end of the run for every query selection strategy of Alg. 1 and for the CodeARC agent, together with the average information the selected queries are guaranteed to yield. All strategies score the same candidate inputs under the same weighted particle posterior and differ only in the score. EIG maximizes the estimated information gain about the equivalence class, Eq. (7). Uncertainty sampling maximizes the number of distinct outputs predicted by the particles of nonzero weight. Diversity maximizes the smallest character-trigram distance, one minus the Jaccard similarity, to the inputs already queried. Random draws an input uniformly. The strategies separate cleanly on the guaranteed information. EIG guarantees close to one bit per query, uncertainty sampling somewhat less, and diversity and random selection about half as much. This ordering carries over to the number of input–output examples, where the informed strategies save about one example per run against random selection, but hardly to accuracy, where all four

Table 5: Query selection strategies of Alg. 1 and the CodeARC agent with 1 seed invocation on the full anonymized CodeARC split. Prior majority returns a program from the majority class of the initial particles without calling the oracle. Entries are means over 5 independent seeds (3 for prior majority) with 95% confidence intervals over seeds. Avg I/O is the average number of input–output examples observed, including the seed invocation, out of a budget of 30. Avg γ_t is the information the selected query is guaranteed to yield, averaged over all queries of the strategy.

Model	Method	Avg I/O ↓	Acc. ↑	Avg γ_t (bits) ↑
Llama3.2 3B	LM only (CodeARC)	19.03 ±0.44	.134 ±.014	–
	Prior majority (no query)	–	.146 ±.017	–
	EIG	15.81 ±0.36	**.167 ±.010**	**.697 ±.008**
	Uncertainty	**15.76 ±0.18**	.165 ±.007	.590 ±.010
	Diversity	15.96 ±0.22	.166 ±.008	.360 ±.004
	Random	16.15 ±0.30	.158 ±.016	.362 ±.022
Llama3.1 8B	LM only (CodeARC)	16.59 ±0.13	.130 ±.015	–
	Prior majority (no query)	–	.179 ±.015	–
	EIG	**12.88 ±0.55**	**.212 ±.011**	**.765 ±.039**
	Uncertainty	13.05 ±0.51	.210 ±.010	.640 ±.028
	Diversity	13.33 ±0.43	.204 ±.013	.387 ±.016
	Random	13.78 ±0.37	.200 ±.011	.382 ±.014
Qwen2.5 7B	LM only (CodeARC)	19.55 ±0.25	.149 ±.009	–
	Prior majority (no query)	–	.168 ±.006	–
	EIG	**9.00 ±0.42**	**.230 ±.008**	**.796 ±.019**
	Uncertainty	9.24 ±0.38	.229 ±.008	.676 ±.022
	Diversity	9.52 ±0.20	.213 ±.010	.433 ±.004
	Random	9.95 ±0.27	.213 ±.008	.448 ±.009
Qwen2.5 32B	LM only (CodeARC)	11.12 ±0.06	.250 ±.011	–
	Prior majority (no query)	–	.217 ±.016	–
	EIG	**10.84 ±0.32**	.335 ±.023	**.722 ±.079**
	Uncertainty	11.08 ±0.28	**.378 ±.035**	.608 ±.050
	Diversity	11.07 ±0.36	.316 ±.005	.313 ±.025
	Random	11.59 ±0.59	.353 ±.020	.337 ±.046
Olmo3 7B	LM only (CodeARC)	13.35 ±0.16	.063 ±.007	–
	Prior majority (no query)	–	.159 ±.031	–
	EIG	**5.90 ±0.33**	.241 ±.014	**.990 ±.026**
	Uncertainty	6.01 ±0.31	**.249 ±.027**	.856 ±.036
	Diversity	6.58 ±0.34	.235 ±.016	.521 ±.005
	Random	7.13 ±0.29	.243 ±.037	.519 ±.019
Ministral 8B	LM only (CodeARC)	16.22 ±0.12	**.239 ±.013**	–
	Prior majority (no query)	–	.136 ±.020	–
	EIG	**13.53 ±0.27**	.171 ±.010	**.791 ±.026**
	Uncertainty	13.63 ±0.17	.170 ±.003	.679 ±.023
	Diversity	14.00 ±0.34	.162 ±.011	.389 ±.031
	Random	14.67 ±0.50	.156 ±.006	.385 ±.027

strategies lie within a few hundredths of each other for every model. The accuracy of a run is decided by whether the proposal contains a correct program at all (App. E.1), and every strategy finds that program within a few queries once it is there. What the information criterion buys is therefore fewer oracle calls at the same accuracy, and against the CodeARC agent it buys both, for all models but Ministral-8B.

E.4 Particle Budget

Fig. 3 varies the number of particles M from 2 to 256 for three models on 100 problems with 1 seed invocation. The particle set is the hypothesis space the algorithm can choose from, and the figure shows what that implies. More particles produce more behavioral classes at round 1, so the first query has more to separate, and the algorithm ends with fewer input–output examples and higher accuracy. Accuracy keeps rising past the $M = 64$ of the main experiments, with no sign of saturation at 256, which agrees with the failure modes of App. E.1, since the runs that converge on a wrong program or on none are runs in which no correct program was sampled, and a larger particle set samples one more often. The budget of 64 is a compromise with the per-round cost, which grows with the number of programs that have to be executed on S (Tab. 2).

E.5 Proposal Prompt and Sampling Temperature

The proposal q_t depends on the prompt shown to the LM and on its sampling temperature. We consider four prompts (Prompts A–D, App. F), which differ in how strongly they ask for hypotheses that disagree on unseen inputs, and six temperatures between 0.7 and 1.3. We choose the prompt and

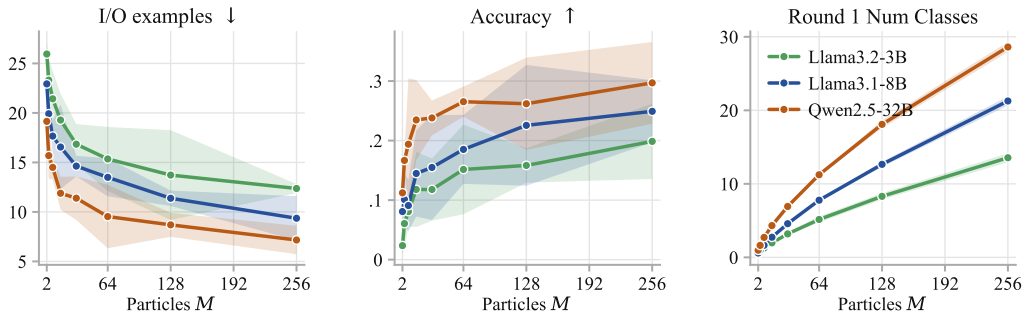

Figure 3: Particle-budget ablation with 1 seed invocation on 100 problems: input–output examples observed, accuracy, and the number of behavioral classes at round 1 against the number of particles M, with 95% confidence intervals over 3 seeds. $M = 64$ was used for the experiments as it yields a good trade-off between computational cost and accuracy.

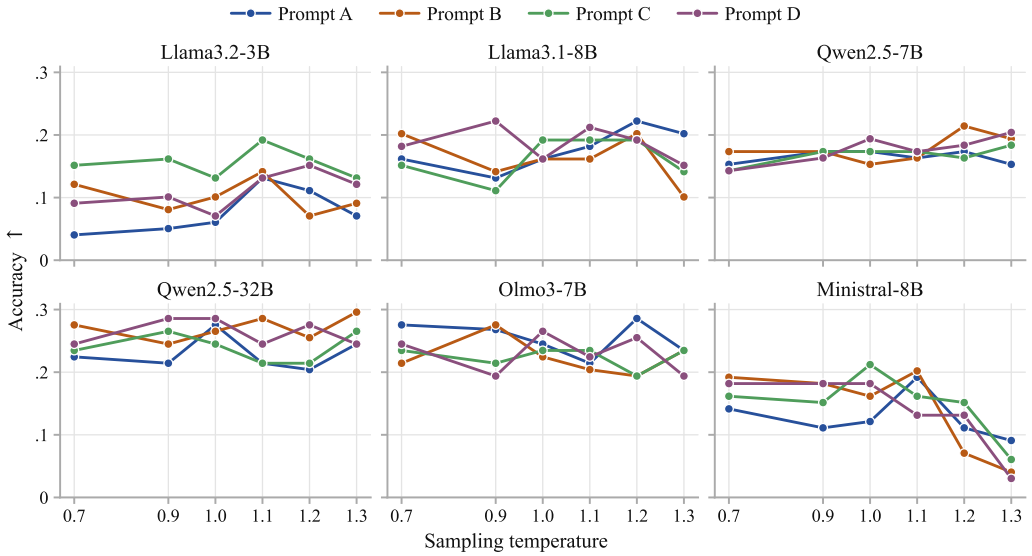

Figure 4: Accuracy of our algorithm against the sampling temperature for each proposal prompt, with 1 seed invocation on 100 problems. Prompts are ordered by accuracy pooled over models and temperatures.

temperature for each model in a preliminary screen over prompts and temperatures in this range. The 100 screening problems are part of the evaluation split. Recomputing Tab. 1 on the remaining 985 problems changes every entry by at most .007. Fig. 4 reports the accuracy of the full algorithm over the grid of prompts and temperatures. No prompt is best for every model. With about 100 problems per configuration, differences of a few hundredths are within sampling noise. The clear effects are that Ministral-8B degrades at temperatures above 1.1 and that Llama3.2-3B performs worst with the shortest prompt (Prompt A), with accuracy varying by .18 and .15 across the grid for these two models, compared with .07 to .12 for the others.

F Prompts

We show each prompt as the text the model receives, with placeholders for problem-specific content written as `<NAME>`. All prompts are sent as a user message.

F.1 DEFAULT CODEARC PROMPTS

The baseline is a multi-turn conversation. Prompt B1 opens it. Depending on the model's previous action, Prompt B2 or B3 is appended as the next user message, and Prompt B4 closes the conversation.

Prompt B1: first turn

````
You are a software engineer tasked with implementing a blackbox Python function
`<FUNCTION_NAME>`. Your goal is to implement this function correctly, ensuring it
works for ALL possible valid inputs, not just the example inputs provided.

Your primary goal is to get the function implementation exactly right.
Your secondary goal is to minimize the number of function invocations and
implementation attempts.

GIVEN EXAMPLES:
Function invocations:
<INPUT_INVOCATIONS>

Corresponding outputs:
<OUTPUT_INVOCATIONS>

STRATEGY OVERVIEW:
You should work in two phases:
1. EXPLORATION PHASE: Use function invocations strategically to fully understand
the function's behavior
2. IMPLEMENTATION PHASE: Only after thorough understanding, attempt an
implementation

This is the first iteration. You must decide between one of the following two
options:
  1. Generate more function invocations in the same format as the given examples.
  Choose this option if you are not confident about how the function works. Each
  invocation should be strategically designed to:
     - Test a specific hypothesis about the function's behavior
     - Explore edge cases (empty inputs, negative numbers, special characters, etc.)
     - Refine your understanding of patterns in the output
     - Verify your assumptions systematically

    You can generate up to <MAX_INVOCATIONS> function invocations across all
    iterations. You must generate exactly 1 invocation in this iteration.

    You should first reason about the problem and the invocations to generate.
    Then, return the function invocations in the following format:
    INVOCATIONS:
    ```python
    print('Result i: ' + str(<FUNCTION_NAME>(args)))
    print('Result i+1: ' + str(<FUNCTION_NAME>(args)))
    ...
    ```
    where each print statement is on a new line. ONLY include executable Python
    code within the code block. You must specify "INVOCATIONS" in the response.

  2. Generate an implementation of `<FUNCTION_NAME>`.

    You have only <MAX_CHECKS> debugging checks total. If your implementation is
    incorrect, you lose one check and are shown a counterexample. If you exhaust
    all checks with incorrect implementations, you CANNOT take any more actions and
    your final attempt will be considered your answer.

    You should ONLY choose this option if ALL of the following conditions are met:
    - You have seen enough diverse examples to understand the function completely
    - You have verified your understanding through strategic invocations
````

```
    - You have tested edge cases and corner cases thoroughly
    - You can confidently define the function for ALL possible valid inputs
    - You have a high degree of certainty about your implementation

    If you choose this option, you should first reason about the problem and the
    implementation to generate. Then, return the implementation in the following
    format:
    IMPLEMENTATION:
    ```python
    def <FUNCTION_NAME>(input1, input2, ...):
      ...
    ```
    ONLY include executable Python code within the code block. You must specify
    "IMPLEMENTATION" in the response.

Reason about which of the two actions described above to perform and then ONLY
perform one of the two actions, specifying which action ("INVOCATIONS" or
"IMPLEMENTATION") you performed in the response.
```

Prompt B2: next turn, after the model requested invocations

```
Here are the outputs of the function invocations from the previous iteration:
<OUTPUT_INVOCATIONS>

STRATEGY UPDATE:
Based on the new data above, you should update your understanding of the function
and decide on your next action.

Following the formats described previously, you must decide between one of the two
following options:
  1. Generate more function invocations to further explore the function's behavior.
  You have <MAX_INVOCATIONS> function invocations left across all remaining
  iterations. You must generate exactly 1 invocation in this iteration.
     [...same INVOCATIONS format block as Prompt B1...]

  2. Generate an implementation of `<FUNCTION_NAME>`.
     You have <MAX_CHECKS> debugging checks left.
     [...same IMPLEMENTATION format block as Prompt B1...]

Reason about which of the two actions described above to perform and then ONLY
perform one of the two actions, specifying which action ("INVOCATIONS" or
"IMPLEMENTATION") you performed in the response.
```

Prompt B3: next turn, after a submitted implementation failed

```
The implementation you generated in the previous iteration failed for the following
input (and potentially other inputs):
<FAILED_INPUTS>

Your implementation was incorrect. You now have <MAX_CHECKS> debugging checks
remaining.

This failure provides valuable information about a case you didn't account for. You
must decide between:

  1. Generate more function invocations to better understand why your
  implementation failed. You have <MAX_INVOCATIONS> function invocations left
  across all remaining iterations. You must generate exactly 1 invocation in this
  iteration.
```

```
     After a failed implementation, generating targeted invocations is almost
     always the best strategy; focus on:
     - Inputs similar to the failed case to understand the pattern you missed
     - Variations of the failed input to isolate the exact behavior
     - Edge cases related to the failure point
     - Inputs that can help discriminate between competing hypotheses
     [...same INVOCATIONS format block as Prompt B1...]

  2. Generate a revised implementation of `<FUNCTION_NAME>`.
     You have <MAX_CHECKS> debugging checks. Only choose this option if you are
     absolutely certain you understand why your previous implementation failed AND
     you are confident your new implementation addresses that issue AND accounts
     for all other possible inputs.
     [...same IMPLEMENTATION format block as Prompt B1...]

Reason about which of the two actions described above to perform and then ONLY
perform one of the two actions, specifying which action ("INVOCATIONS" or
"IMPLEMENTATION") you performed in the response.
```

Prompt B4: final turn, once all debugging checks are used

```
Here are the outputs of the function invocations from the previous iteration:
<OUTPUT_INVOCATIONS>

Now, synthesize all the information you've gathered across all iterations. Analyze
patterns, edge cases, and the function's overall behavior.

Generate your final implementation of the function. Make sure it handles all cases,
including the ones you've explicitly tested and others that follow the same pattern.

Return the final implementation in the following format:
```python
def <FUNCTION_NAME>(input1, input2, ...):
  ...
```
ONLY include executable Python code within the code block.
```

F.2 Proposal Prompts

These are the four proposal prompts of our algorithm, and App. E.5 describes which one is used for each model. The examples displayed are the seed invocations and all queries answered so far. The program prior uses the same prompt, displaying only the seed invocations.

Prompt A

```
Implement the hidden Python function `<FUNCTION_NAME>`.

Function invocations:
<INPUT_INVOCATIONS>

Corresponding outputs:
<OUTPUT_INVOCATIONS>

Return exactly one implementation using only the structure below without any
additional explanations:
IMPLEMENTATION:
```python
def <FUNCTION_NAME>(input1, input2, ...):
  ...
```
```

```
```

Prompt B

```
Implement the hidden Python function `<FUNCTION_NAME>` from the examples below.

GIVEN EXAMPLES:
Function invocations:
<INPUT_INVOCATIONS>

Corresponding outputs:
<OUTPUT_INVOCATIONS>

First, list concretely HOW the examples underdetermine the function: name specific
inputs NOT shown above on which two valid implementations would disagree (edge
cases, empty inputs, ties, ordering, types, unusual characters). Then pick ONE
unresolved interpretation -- not necessarily the most popular one -- and implement
it. The implementation must reproduce all given examples exactly. Return exactly
one implementation using only the structure below without any additional
explanations:
IMPLEMENTATION:
```python
def <FUNCTION_NAME>(input1, input2, ...):
  ...
```
```

Prompt C

```
You are a software engineer tasked with implementing a blackbox Python function
`<FUNCTION_NAME>` from a set of input-output pairs provided below.
Your goal is to implement this function correctly, ensuring it works for ALL
possible valid inputs, not just the example inputs provided.

GIVEN EXAMPLES:
Function invocations:
<INPUT_INVOCATIONS>

Corresponding outputs:
<OUTPUT_INVOCATIONS>

You should first briefly reason about the problem and all possible implementations
that may lead to this input-output behavior. Every proposed implementation must
satisfy all provided input-output examples exactly. Across proposals, favor
diversity in hypotheses and complexity: include simpler and more structured
alternatives (for example direct formulas, conditionals, loops, helper functions,
or different algorithmic strategies), while staying consistent with the examples.
Return exactly one implementation using only the structure below without any
additional explanations:
IMPLEMENTATION:
```python
def <FUNCTION_NAME>(input1, input2, ...):
  ...
```
```

Prompt D

```
You are a software engineer tasked with implementing a blackbox Python function
`<FUNCTION_NAME>` from a set of input-output pairs provided below.
Your goal is to implement this function correctly, ensuring it works for ALL
possible valid inputs, not just the example inputs provided.

GIVEN EXAMPLES:
Function invocations:
<INPUT_INVOCATIONS>

Corresponding outputs:
<OUTPUT_INVOCATIONS>

IMPORTANT: this same prompt is being sent to you many times independently, and your
answers are pooled into a set of competing hypotheses. Do NOT default to the single
most obvious implementation every time. Treat your answer as ONE DRAW from the
space of all distinct rules that reproduce the given examples exactly, sampled in
proportion to how plausible each rule is. Different draws should disagree with each
other on inputs that are NOT in the given examples, while all agreeing on the
examples themselves.

You should first briefly reason about the problem and enumerate several genuinely
different rules that all reproduce the given input-output behavior -- they may
differ in how they generalise (edge cases, empty or negative or extreme inputs,
types, ordering, off-by-one boundaries) or in algorithmic strategy (direct formula,
conditionals, loops, recursion, helper functions, library calls). Then pick ONE of
them and commit to it. Every proposed implementation must satisfy all provided
input-output examples exactly. Return exactly one implementation using only the
structure below without any additional explanations:
IMPLEMENTATION:
```python
def <FUNCTION_NAME>(input1, input2, ...):
    ...
```
```

F.3 Input Distribution Prompt

The prompt that instantiates the input distribution μ. It is sampled repeatedly, once per problem and before the first query, and each parsed completion becomes one element of S.

Input distribution prompt

```
You are choosing one informative test input for a hidden Python function
`<FUNCTION_NAME>`.

Known examples:
Function invocations:
<INPUT_INVOCATIONS>

Corresponding outputs:
<OUTPUT_INVOCATIONS>

Reply with ONLY the INVOCATIONS block below--no preamble, no headings, no
explanations, and no text outside the fenced Python block.

INVOCATIONS:
```python
print('Result 1: ' + str(solution(args)))
```
```

```
Prefer diverse inputs (edges, negatives, extremes, unusual structures) inside the
single print(...) line.
```