STAB: SPECIFICATION-DRIVEN TESTING FOR ALGORITHMIC BOTTLENECKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Evaluating the efficiency of algorithmic code requires test cases that expose runtime bottlenecks. Previous methods generate efficiency test cases either by increasing input size or by generating code-specific inputs that make the given implementation run slowly. Consequently, they do not address the structural input conditions that drive the algorithmic worst case. We introduce **STAB**, a specification-driven pipeline that generates test cases that expose algorithmic bottlenecks from a natural-language problem specification alone. STAB separates the task into constraint-bound maximization and adversarial structure injection. (i) The constraint saturator extracts constraints and resolves large admissible size assignments using rule-based saturation and CP-SAT optimization over related variables. (ii) The adversarial scenario injector retrieves implementation-level adversarial construction principles from a curated scenario catalog using keyword matching and K-nearest neighbors (KNN). STAB encodes the problem specification, resolved boundary, and retrieved construction principles into a structured generation specification, from which the LLM synthesizes a Python generator function. On CodeContests, STAB raises the rate of generated test cases that expose algorithmic bottlenecks from 50.43% to 73.45% on average across open-source LLMs and from 57.45% to 71.85% on average across closed-source LLMs, with consistent gains across Python, Java, and C++. Our code is available at `https://anonymous.4open.science/r/Slow_TC-E2A8`.

1 INTRODUCTION

Large language models (LLMs) are widely used for code generation (Chen et al., 2021; Guo et al., 2024; Hui et al., 2024) and test generation (Yuan et al., 2024; Chen et al., 2024; Ryan et al., 2024; Schäfer et al., 2024; Lim et al., 2026). Existing benchmarks and test generation methods mainly check whether a program returns the expected output on a fixed test suite. For algorithmic problems with time limits, however, passing ordinary tests does not guarantee efficiency on large inputs allowed by the specification. This setting requires efficiency test cases, valid inputs that expose algorithmic bottlenecks rather than output correctness.

Generating efficiency test cases is not equivalent to making inputs large. Worst-case behavior often depends on structure as well as size. Reverse-sorted arrays stress pivot-sensitive sorting, path-shaped trees increase traversal depth, and dense graphs increase the work of shortest path routines. An efficiency test case must therefore combine a large valid input boundary with a structure that increases runtime. Existing efficiency-oriented test generators address only part of this requirement. EvalPerf (Liu et al., 2024) scales an LLM-generated input until a target implementation reaches a runtime or memory threshold. WEDGE (Yang et al., 2025) infers bottleneck conditions from executions of a target program and searches for inputs satisfying those conditions. Both approaches rely on input scaling or feedback from a specific implementation. They are useful for slowing a given program, but less suited to generating tests from the problem specification alone.

We propose **STAB** (**S**pecification-driven **T**esting for **A**lgorithmic **B**ottlenecks), a pipeline that generates efficiency test cases from a natural-language problem specification alone. STAB separates the task into constraint-bound maximization and adversarial structure injection. (i) The constraint saturator extracts input constraints from the specification and resolves the largest size parameters those constraints allow by considering relationships among variables through rule-based saturation

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.

and CP-SAT (Perron & Furnon, 2024) optimization. (ii) The adversarial scenario injector matches the problem to scenarios in a curated catalog through keyword matching and KNN, and retrieves the associated construction principles. STAB then encodes the problem specification, resolved boundary, and retrieved construction principles into a structured generation specification, from which the LLM synthesizes a Python generator function. This decomposition keeps the LLM's role narrow: it composes a valid generator from a resolved boundary and selected construction principles, rather than solving constraint satisfaction and adversarial input design at once. We evaluate STAB on CodeContests (Li et al., 2022) across Python, Java, and C++, using four LLMs that cover open-source and closed-source settings. For each generated test case, we count success when the test case drives an accepted solution to a runtime larger than the maximum runtime induced by the CodeContests test set for the same problem. STAB increases the share of generated test cases that expose algorithmic bottlenecks from 50.43% to 73.45% on average across open-source LLMs and from 57.45% to 71.85% on average across closed-source LLMs.

We make three main contributions:

- **Constraint-aware boundary resolution.** We introduce a constraint saturator that extracts constraints from specifications and computes large valid size assignments over related variables.
- **Scenario-guided adversarial construction.** We build a curated catalog of adversarial construction principles and retrieve problem-matched principles using keyword matching and KNN.
- **Empirical validation across models and languages.** We show that STAB outperforms standard prompting and prior efficiency-oriented generators across the CodeContests evaluation.

2 Related Work

2.1 Efficiency in LLM-based Code Generation

Research on LLM-based code generation has centered on functional correctness. Benchmarks such as HumanEval, MBPP, and APPS measure whether a generated program passes a fixed reference test suite (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021). Code-specialized models such as CodeLlama, DeepSeek-Coder, StarCoder, and the Qwen-Coder series have steadily raised this correctness baseline (Rozière et al., 2023; Guo et al., 2024; Li et al., 2023; Hui et al., 2024). AlphaCode and its follow-ups have extended LLM-based code generation to competitive programming, where problems typically specify input constraints and time limits and require algorithmic reasoning. In this setting, the gap between a functionally correct solution and an efficient solution becomes pronounced (Li et al., 2022; Ridnik et al., 2024). Prior work has also studied the efficiency of generated code. EffiLearner uses execution overhead profiles to improve the efficiency of LLM-generated code, and PerfCodeGen incorporates runtime feedback during self-refinement (Huang et al., 2024; Peng et al., 2025). These studies show that functional correctness alone does not fully characterize the quality of generated code. For algorithmic code generation, these results indicate that efficiency evaluation must accompany correctness evaluation.

2.2 LLM-assisted Test Case Generation

LLM-assisted test case generation has developed along multiple directions. Functional validation methods synthesize unit tests that check whether a program returns the expected output, with representative work including EvalPlus, ChatUniTest, and CodaMOSA (Liu et al., 2023; Chen et al., 2024; Lemieux et al., 2023). Robustness-oriented methods generate failure-inducing inputs that expose correctness or robustness failures in candidate programs (Wang et al., 2023; Zhang et al., 2025; Zhong et al., 2025; Lim et al., 2026). Efficiency evaluation methods generate test inputs that stress runtime behavior to assess the efficiency of candidate programs. EvalPerf (Liu et al., 2024) synthesizes an input generator and scales a size parameter until the measured runtime reaches a threshold. WEDGE (Yang et al., 2025) identifies constructs at the implementation level, such as loops, and generates inputs that increase the number of operations those constructs perform. In both cases, the generation signal comes from input size scaling or execution feedback from a particular implementation. Neither method explicitly infers the algorithmic structure implied by the problem or targets the slow input construction associated with that structure. Algorithmic problems, however, require efficiency test inputs that are both valid at large input sizes and structured to trigger worst-case

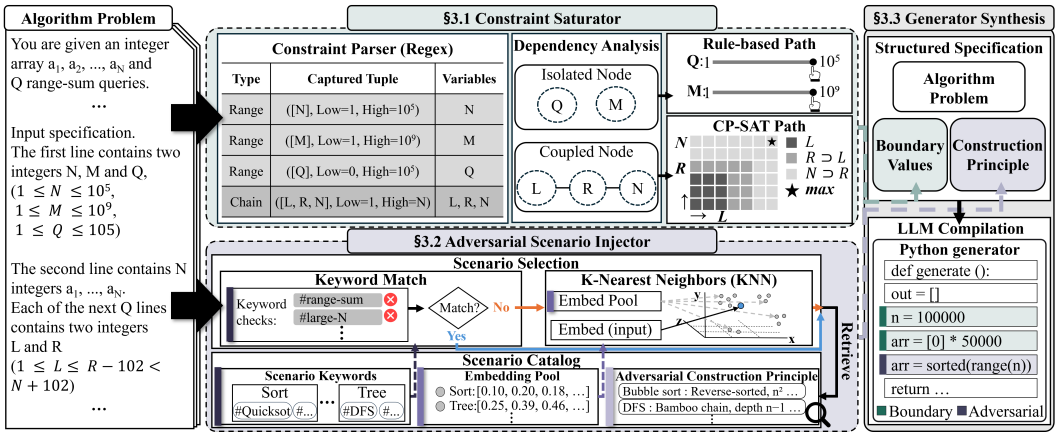

Figure 1: An overview of the STAB pipeline.

behavior. STAB addresses this gap by generating efficiency test cases from the problem specification alone. It resolves input constraints and encodes adversarial constructions implied by the algorithmic problem, rather than relying on input size or behavior observed from a specific implementation.

3 Notation

All notation is defined for a fixed implementation language, and the language index is omitted because Python, Java, and C++ are evaluated independently. For an evaluated CodeContests problem p, let $\mathcal{T}_{\mathrm{CC}}(p)$ and $\mathcal{T}_{\mathrm{gen}}(p)$ denote the CodeContests test set and the generated efficiency test set, respectively. The set $\mathcal{A}(p)$ contains the accepted solutions for p, and $\tau(s,t)$ denotes the execution time of solution s on test case t.

For each problem p, we selected five *evaluation solutions* from $\mathcal{A}(p)$. The solutions $s_{p,\mathrm{fast}}$ and $s_{p,\mathrm{slow}}$ have the lowest and highest mean runtimes on $\mathcal{T}_{\mathrm{CC}}(p)$, respectively, whereas s_{p,r_i}, $i \in \{1,2,3\}$, were sampled at random. We use $s_{p,r}$ to denote any of these five evaluation solutions. In the main evaluation, the *reference implementation* supplied to EvalPerf or WEDGE was excluded from this set.

4 The STAB Pipeline

Given the natural-language specification of problem p, STAB constructs $\mathcal{T}_{\mathrm{gen}}(p)$ without using a reference implementation. For each $t \in \mathcal{T}_{\mathrm{gen}}(p)$, STAB synthesizes a Python generator function that materializes the input. A direct LLM prompt must infer the valid boundary and the input construction from the problem specification while writing the generator. This can lead the model to satisfy obvious numeric bounds while missing the adversarial structure or to propose a plausible structure that violates the input format. STAB reduces this burden by supplying the final LLM call with two intermediate outputs. (i) The constraint saturator computes the largest admissible values for variables that define the valid input boundary. (ii) The adversarial scenario injector selects scenarios that match the problem and retrieves their construction principles. The LLM then composes a generator from the problem specification, the resolved boundary, and the selected principles. Figure 1 shows the process.

4.1 Constraint Saturator

The constraint saturator computes the largest admissible values for variables that define the valid input boundary of an algorithmic problem. Its output is a *resolved boundary*: an assignment to variables such as the number of elements, queries, vertices, or edges that satisfies the constraints extracted from the problem specification. This step accounts for relationships among variables rather than maximizing each variable independently. Variables that appear only in unary bounds can be saturated to their largest extracted values. Variables that appear together in product bounds, chained inequalities, or aggregate constraints must be resolved jointly because increasing one variable can

Table 1: Example adversarial construction principles from STAB's scenario catalog. Rows pair representative implementations with construction principles encoded in the structured generation specification. The full catalog of 13 scenarios and 51 implementations appears in Appendix M.

Scenario	Implementation	Adversarial Construction Principle
Sorting	Quicksort with first or last pivot	An already sorted sequence makes the pivot an extreme element at each partition, creating highly unbalanced recursion.
Hashing	Hash table with identity hash	Keys chosen as multiples of the bucket modulus collide into one bucket, so lookup traverses a long chain.
Tree traversal	DFS via system recursion	A bamboo tree with edges $(i, i+1)$ for $1 \leq i < n$ gives depth $n-1$ and maximizes recursion depth.

restrict the valid range of another. The constraint saturator therefore applies rule-based saturation to independent variables and CP-SAT optimization to coupled variable groups. The resulting boundary is encoded together with the selected construction principles and the problem specification into the structured generation specification, from which the LLM compiles a generator that instantiates the corresponding constructions within a validated input range.

Constraint Extraction. Algorithmic problem specifications often express input constraints as mathematical notation embedded in natural language, such as $1 \leq N \leq 10^5$. The constraint saturator first converts these constraints into normalized records that contain variables, relation types, and bounds. A regular expression parser is suitable here because the same constraint patterns repeatedly appear across problem specifications, including ranges, chained inequalities, products, and string length bounds. For example, it maps $1 \leq N \leq 10^5$ to a scalar range, $1 \leq a_i \leq 10^9$ to an indexed range, $1 \leq L \leq R \leq N$ to a chained inequality, $N \cdot M \leq 10^6$ to a product constraint, and $|s| \leq 10^5$ to a string length bound. The parser returns normalized constraint records containing variable names, constraint types, and integer bounds, which are used in dependency analysis.

Dependency Analysis. After constraint extraction, the constraint saturator builds a variable dependency graph. Each node is a variable in the normalized constraint records. A constraint relating two or more variables creates dependency edges among them. Variables with no dependency edge are treated as independent variables. Variables connected by dependency edges form a dependent variable group. Each dependent variable group is resolved jointly because increasing one variable can reduce the valid range of another. For example, N is treated as dependent when it appears not only in $1 \leq N \leq 10^5$ but also in a product constraint such as $N \cdot M \leq 10^6$ or in a chain such as $1 \leq L \leq R \leq N$. The graph groups variables rather than constraints before boundary resolution.

Boundary Resolution. The constraint saturator applies different procedures to independent variables and dependent variable groups.

- **Rule-based Path.** For each independent variable, the constraint saturator assigns the largest upper bound extracted for that variable. Since the variable has no dependency edge, increasing it to its bound cannot violate any constraint involving another variable.
- **CP-SAT Path.** For each dependent variable group, the constraint saturator formulates a CP-SAT model from the variable domains and the extracted constraints that relate variables in the group (Perron & Furnon, 2024). The CP-SAT solver returns a feasible assignment that makes the variables in the group large while respecting the extracted constraints. This path is used for chained inequalities, product constraints, and aggregate size constraints, where independent saturation can produce an invalid assignment.

The resolved boundary is not a proof of the global maximum over the full input space. It is a valid size configuration under the constraints extracted by the constraint saturator and serves as the validated size budget for generator construction.

4.2 Adversarial Scenario Injector

The adversarial scenario injector matches the problem specification to scenarios in the catalog using keyword matching and KNN. It supplies construction principles from the matched scenarios to the generator construction step. Among the sorting entries, one construction principle specifies a reverse-sorted sequence for pivot-sensitive sorting strategies. Given the supplied principles and the resolved boundary, the LLM composes a generator that emits the resolved boundary values for size-determining variables and encodes the input structure according to the selected principles.

Scenario Catalog. We maintain a catalog of 13 adversarial scenarios, each corresponding to an algorithm family such as sorting, hashing, tree traversal, or graph shortest path. For each scenario, the catalog stores (i) representative implementations and construction principles that trigger their worst-case behavior, (ii) scenario keywords used by the keyword matching step, and (iii) an anchor pool for KNN retrieval. We construct the anchor pool by matching unlabeled training problem descriptions to each scenario through its keywords. The descriptions come from a training split independent of our evaluation set, and we do not access reference implementations, generated tests, runtime measurements, or evaluation outcomes. The catalog is compiled from standard algorithms and competitive programming references (Sedgewick & Wayne, 2011; Laaksonen, 2017) and foundational work on algorithmic complexity attacks (McIlroy, 1999; Crosby & Wallach, 2003). Table 1 summarizes representative adversarial construction principles from the scenario catalog.

Scenario Selection. The adversarial scenario injector retrieves candidate scenarios through two mutually exclusive paths: Keyword Match and KNN. Keyword Match retrieves every scenario whose keywords appear explicitly in the problem specification. KNN is invoked only when Keyword Match retrieves no scenario. It returns the two scenarios with the highest cosine similarity. The outputs of the two paths are never merged or jointly ranked. This conditional rule prioritizes explicit lexical evidence when available and uses semantic similarity only as a fallback, reducing irrelevant construction principles passed to generator synthesis. Appendix E compares this rule with seven alternative retrieval configurations.

- **Keyword Match.** Scenario keywords take two forms. The first consists of algorithm or data structure names, such as "Dijkstra" for graph shortest path. The second consists of task-level cues, such as "non-decreasing" for sorting or "range query" for BST and segment structures. At inference, Keyword Match scans the problem specification and returns every scenario whose keywords appear.
- **K-Nearest Neighbors (KNN).** For each scenario, we embed its anchors with `SFR-Embedding-2_R`, L2-normalize the embeddings, and average them into a single scenario centroid. At inference, we embed the evaluation problem with the same model, compare it to the centroids by cosine similarity, and retain the top two scenarios.

4.3 Generator Synthesis

STAB encodes the generation task as a structured specification over three components: the resolved boundary from the constraint saturator, the candidate scenarios from the adversarial scenario injector, and the problem specification. The LLM compiles this specification into a Python generator.

5 Experimental Setup

5.1 Datasets

We used the validation and test splits of CodeContests (Li et al., 2022) for runtime evaluation in Python, Java, and C++. The training split was used only to build the KNN anchor pool from problem specifications (§4.2), without solutions, generated tests, runtime measurements, or evaluation outcomes. For each problem p and implementation language, we ranked all $s \in \mathcal{A}(p)$ by mean runtime on $\mathcal{T}_{\mathrm{CC}}(p)$ and selected the fastest, slowest, and three randomly sampled evaluation solutions. Appendix A reports problem counts; Appendix B details dataset construction and solution selection.

5.2 Evaluated Models and Baselines

Evaluated Models. We evaluate four LLMs: two open-source instruction-tuned models with comparable parameter counts and publicly accessible weights, Qwen-3.5-27B (Qwen-3.5) and gemma-4-31B-it (Gemma-4), and two closed-source models, OpenAI's gpt-5.4-nano (GPT-5.4) and Google's gemini-3.1-flash-lite (Gemini-3.1), with undisclosed parameter counts and training data.

Baseline Strategies. The evaluation compares STAB against three baselines under two input settings. Standard Prompting (Base) uses only the algorithmic problem specification as input, matching STAB's input condition, and generates a Python generator function that emits an efficiency test case. EvalPerf (Liu et al., 2024) and WEDGE (Yang et al., 2025) additionally require a reference implementation and therefore operate under richer input than STAB. We include them as representative prior methods for efficiency test generation.

5.3 Evaluation Metric

We define Algorithmic Slowdown Rate (ASR) by comparing the runtime induced by each generated efficiency test case $t \in \mathcal{T}_{\text{gen}}(p)$ with the largest runtime induced by $\mathcal{T}_{\text{CC}}(p)$ on the same evaluation solution $s_{p,r}$. Runtime $\tau(s,t)$ is measured in the DOMjudge (Eldering et al., 2004) environment described in Appendix C. ASR is computed separately for each implementation language.

Let $\mathcal{P}$ be the set of evaluated problems. For each problem p and evaluation solution $s_{p,r}$, the CodeContests runtime threshold is

$$\tau^*_{p,r} = \max_{t \in \mathcal{T}_{\text{CC}}(p)} \tau(s_{p,r}, t).$$

With $N_{\text{gen}} = \sum_{p \in \mathcal{P}} |\mathcal{T}_{\text{gen}}(p)|$, ASR for a fixed evaluation solution index r is

$$\text{ASR}_r = \frac{1}{N_{\text{gen}}} \sum_{p \in \mathcal{P}} \sum_{t \in \mathcal{T}_{\text{gen}}(p)} \mathbb{I}\big[\tau(s_{p,r}, t) > \tau^*_{p,r}\big],$$

where $\mathbb{I}[\cdot]$ returns 1 when the generated test case exposes a bottleneck and 0 otherwise.

6 Results and Analysis

We first compare STAB against Base prompting on the CodeContests benchmark in Section 6.1. The remaining sections address two questions through complementary analyses. Sections 6.2 and 6.3 examine whether STAB produces stable gains with a clear module-level decomposition, focusing on variation across heterogeneous evaluation solutions and the contribution of each pipeline module. Sections 6.4 and 6.5 then place STAB in context by comparing it against prior efficiency-oriented generators and the CodeContests test set $\mathcal{T}_{\text{CC}}(p)$.

6.1 Main Results

Table 2 reports ASR, the fraction of tests $t \in \mathcal{T}_{\text{gen}}(p)$ for which $\tau(s_{p,r}, t) > \tau^*_{p,r}$. STAB improves the average for all four LLMs, raising the model average from 53.94% to 72.65%. At the model level, the relative gains from Base to STAB are 31.56% for Qwen-3.5, 61.90% for Gemma-4, 49.26% for Gemini-3.1, and 7.20% for GPT-5.4. Appendix K reports execution validity.

The *Fast* and *Slow* groups contain $s_{p,\text{fast}}$ and $s_{p,\text{slow}}$, respectively, as selected by their mean runtimes on $\mathcal{T}_{\text{CC}}(p)$. Comparing their exposure rates therefore shows whether generated efficiency tests simply follow this initial ranking or reveal additional bottlenecks. Under Base, the *Fast* group has a lower mean ASR than the *Slow* group, 41.32% versus 59.92%. STAB raises both groups, but the relative increase is larger for *Fast* evaluation solutions, with average exposure rates of 59.35% for *Fast* and 74.01% for *Slow*. As a result, the *Fast-Slow* gap is reduced by 21.18% relative to Base. This reduced gap suggests that the larger separation under Base reflects limited bottleneck exposure rather than a robust efficiency difference between *Fast* and *Slow* evaluation solutions.

The *Random* group provides a second check of this interpretation. Because $s_{p,\text{slow}}$ is selected by mean runtime on $\mathcal{T}_{\text{CC}}(p)$, a generated test set that merely preserves the benchmark ordering should

Table 2: ASR across five evaluation solutions $s_{p,r}$. Random† reports the mean ASR over s_{p,r_i} for $i \in \{1, 2, 3\}$. Avg. reports ASR averaged over five evaluation solutions and three languages. Full results appear in Appendix L.

Model	Method	Python			Java			C++			Avg.
		Fast	Slow	Random†	Fast	Slow	Random†	Fast	Slow	Random†	
Qwen-3.5	Base	26.57%	57.10%	61.35%	27.60%	56.65%	59.43%	34.74%	57.19%	62.69%	54.02%
	STAB	**56.14%**	**74.20%**	**74.56%**	**59.46%**	**75.11%**	**77.41%**	**59.33%**	**69.11%**	**72.27%**	**71.07%**
Gemma-4	Base	42.32%	65.80%	41.77%	41.54%	57.38%	41.33%	58.07%	64.52%	41.16%	46.83%
	STAB	**60.00%**	**79.23%**	**79.84%**	**63.08%**	**82.17%**	**82.98%**	**62.22%**	**71.78%**	**76.76%**	**75.82%**
Gemini-3.1	Base	34.11%	54.88%	54.94%	35.48%	58.91%	59.79%	32.22%	42.52%	43.33%	48.82%
	STAB	**59.23%**	**75.65%**	**77.84%**	**59.82%**	**77.65%**	**79.91%**	**58.52%**	**67.11%**	**73.93%**	**72.87%**
GPT-5.4	Base	51.88%	68.21%	68.05%	57.47%	74.30%	73.82%	53.85%	61.63%	66.03%	66.07%
	STAB	**56.04%**	**73.91%**	**74.81%**	**58.01%**	**75.66%**	**77.31%**	**60.30%**	**66.59%**	**71.83%**	**70.83%**

keep *Slow* above *Random*. Base does not consistently preserve this ordering: *Random* exceeds *Slow* in 7 of the 12 model and language pairs, even though its aggregate *Random* average remains lower than *Slow*. Under STAB, the reversal becomes systematic: *Random* exceeds *Slow* in all 12 model and language pairs, with average exposure rates of 76.62% and 74.01%. Relative to Base, STAB improves the exposure rate more strongly for *Random* than for *Slow*, with relative increases of 36.48% and 23.52%, respectively. This pattern indicates that the randomly sampled evaluation solutions contain latent runtime bottlenecks that Base exposes only inconsistently, while STAB exposes them across all 12 pairs. Together with the reduced gap between *Fast* and *Slow*, the reversal in which *Random* exceeds *Slow* shows that STAB exposes bottlenecks more broadly and consistently across the five evaluation solutions than Base.

6.2 Consistency Across Evaluation Solutions

A single algorithmic problem admits multiple accepted solutions in $\mathcal{A}(p)$ with different runtime profiles because the solutions can use different algorithms, data structures, or implementation strategies. The same efficiency test case can be adversarial for one evaluation solution $s_{p,r}$ but less adversarial for another. A generator targeting an implementation artifact would perform strongly on one evaluation solution and weaken when the evaluation aggregates over all five solutions. We use this variation as a consistency check by evaluating every $t \in \mathcal{T}_{\text{gen}}(p)$ on the five evaluation solutions $s_{p,r}$ for each problem and implementation language.

Table 2 reveals two consistent patterns across the five evaluation solutions. First, STAB raises the bottleneck exposure rate over Base for the Fast, Slow, and Random groups. The largest relative gain occurs for $s_{p,\text{fast}}$, whose mean ASR increases from 41.32% to 59.35%, which is informative because fast solutions are the hardest to expose. The mean ASR for $s_{p,\text{slow}}$ increases from 59.92% to 74.01%, and the mean over the three randomly sampled evaluation solutions increases from 56.14% to 76.62%. The improvement therefore does not concentrate on any evaluation-solution group. Second, variation across the five evaluation solutions decreases. The standard deviation over the three randomly sampled evaluation solutions s_{p,r_i} drops from 2.55% under Base to 0.93% under STAB, and the average range across the five evaluation solutions is reduced by 26.02% relative to Base. The lower spread shows that STAB's gains do not favor any evaluation solution.

6.3 Per-Module Contribution

The ablation compares STAB with three variants that remove the constraint saturator (M1), the adversarial scenario injector (M2), or both modules. Removing M1 leaves the generator without a resolved valid boundary, whereas removing M2 leaves it without adversarial structural guidance. Table 3 shows that STAB achieves the highest ASR in all combinations. Appendix I reports the module-level accuracy of M1 and M2.

The single-module variants reveal complementary failure modes. The variant without M1 can still use the construction principle matched by M2 but must infer the input size from the problem specification, and the inferred values fall below the largest admissible bound when constraints involve coupled variables or aggregate bounds. The construction is therefore instantiated at a conservative scale

Table 3: Ablation study on STAB. M1 and M2 denote the constraint saturator and the adversarial scenario injector. Each ASR value is averaged over the five evaluation solutions $s_{p,r}$.

Method	$\mathrm{ASR}_{\mathrm{Python}}$	$\mathrm{ASR}_{\mathrm{Java}}$	$\mathrm{ASR}_{\mathrm{C++}}$
Gemini-3.1			
STAB	**73.68%**	**75.44%**	**69.48%**
− M1	70.32%	73.67%	66.47%
− M2	72.37%	75.40%	67.17%
− M1, M2	50.76%	54.75%	40.95%
GPT-5.4			
STAB	**70.88%**	**73.12%**	**68.48%**
− M1	64.15%	68.45%	61.51%
− M2	66.45%	70.19%	64.07%
− M1, M2	64.85%	70.64%	62.71%

that does not push the algorithm to its worst case. The variant without M2 can use the resolved boundary from M1 but lacks the adversarial construction principle needed to convert that boundary into an effective efficiency test case. The resulting input uses the boundary resolved by M1 but fills that boundary with a generic structure that does not expose the algorithmic bottleneck. The variant without M2 outperforms the variant without M1 on both Gemini-3.1 and GPT-5.4 across Python, Java, and C++, suggesting that boundary resolution contributes more than adversarial construction when only one module is present. STAB's additional gain over the variant without M2 nevertheless shows that adversarial construction principles are essential for fully effective efficiency test cases. Only the full pipeline satisfies both conditions by instantiating the matched construction principle at the size resolved by M1. Appendix F compares the contributions of M1 and M2 across disjoint size-only and structure-only buckets and examines how the two modules complement each other across bottleneck types. Appendix H compares generators from STAB and its two single-module variants on a shared problem, highlighting differences in boundary use and adversarial structure.

6.4 Comparison with Prior Methods

Table 4 compares STAB with EvalPerf and WEDGE, both of which require a reference implementation. STAB achieves the highest ASR in all 12 model-language combinations, showing that its advantage extends to both open-source and closed-source models. Averaged across four LLMs and three implementation languages, STAB achieves 72.65% ASR, compared with 35.37% for EvalPerf and 37.35% for WEDGE. The relative gains are 105.39% over EvalPerf and 94.51% over WEDGE.

The results indicate that the gap is not explained by access to a reference implementation, but by the kind of signal each method uses to construct test cases. EvalPerf remains effective when the worst case is mainly scale-driven, but its average ASR is 35.37% because increasing a size parameter does not specify the ordering, topology, or coupled dimensions needed by structural worst cases. WEDGE also uses a reference implementation, but its search signal is tied to the execution behavior of that implementation. Such a signal can find slow paths in the reference implementation, but it is less reliable when the bottleneck depends on the shape of the whole input, such as a reverse-sorted array, a bamboo tree, or a dense graph that maximizes both vertices and edges under the input constraints. STAB avoids these two failure modes by combining a resolved boundary derived from the problem specification with retrieved construction principles. This combination explains why STAB outperforms both methods requiring a reference implementation despite using only the problem specification at inference. Appendix G further analyzes these failure modes in the reference implementation setting. Appendix D additionally compares STAB with HARDTESTGEN and CodeContests-O, two code-based methods for correctness-oriented test generation. STAB's advantage extends to both efficiency-oriented and correctness-oriented test-generation methods.

6.5 Runtime Discrimination Analysis

We use Gemma-4 because it achieves the highest average ASR across Python, Java, and C++ in Table 2. For each problem p, we independently select one test case from $\mathcal{T}_{\mathrm{CC}}(p)$ and one from

Table 4: Comparison with prior efficiency-oriented generators across four models. Each ASR value is averaged over the five evaluation solutions $s_{p,r}$ per problem and implementation language.

Model	Method	$\text{ASR}_{\text{Python}}$	ASR_{Java}	$\text{ASR}_{\text{C++}}$
Qwen-3.5	EvalPerf	29.72%	34.64%	31.72%
	WEDGE	28.29%	33.77%	40.13%
	STAB	**70.80%**	**73.36%**	**69.05%**
Gemma-4	EvalPerf	36.41%	39.27%	33.41%
	WEDGE	31.39%	40.83%	54.98%
	STAB	**75.75%**	**78.84%**	**72.86%**
Gemini-3.1	EvalPerf	28.91%	31.89%	23.21%
	WEDGE	35.79%	41.09%	38.92%
	STAB	**73.68%**	**75.44%**	**69.48%**
GPT-5.4	EvalPerf	45.35%	47.84%	42.06%
	WEDGE	29.00%	36.18%	37.81%
	STAB	**70.88%**	**73.12%**	**68.48%**

$\mathcal{T}_{\text{gen}}(p)$. Within each test set, the selected test case maximizes $\max\{\tau(s_{p,\text{fast}}, t), \tau(s_{p,\text{slow}}, t)\}$. We then measure the runtime of the selected test case on both $s_{p,\text{fast}}$ and $s_{p,\text{slow}}$. Figure 2 plots the resulting runtime pairs on log-scaled axes. Points above the diagonal indicate test cases for which $s_{p,\text{slow}}$ runs longer than $s_{p,\text{fast}}$. Across Python, Java, and C++, STAB test cases widen the gap between fast and slow solutions. On Java, STAB increases the median fast runtime by 198% and the median slow runtime by 322% relative to the CodeContests test cases, and the share of problems where the slow solution takes longer than the fast solution rises from 69.8% to 85.1%. On C++, the CodeContests test cases leave 54.4% of problems in a very low-runtime regime, where the fast and slow solutions are difficult to distinguish by execution time. STAB reduces this fraction to 15.9%, making the runtime gap between fast and slow solutions more visible. The widened gap does not by itself imply that any specific solution is algorithmically superior. It indicates that STAB test cases make algorithmic complexity differences observable in measured runtime, supporting downstream tasks such as solution ranking.

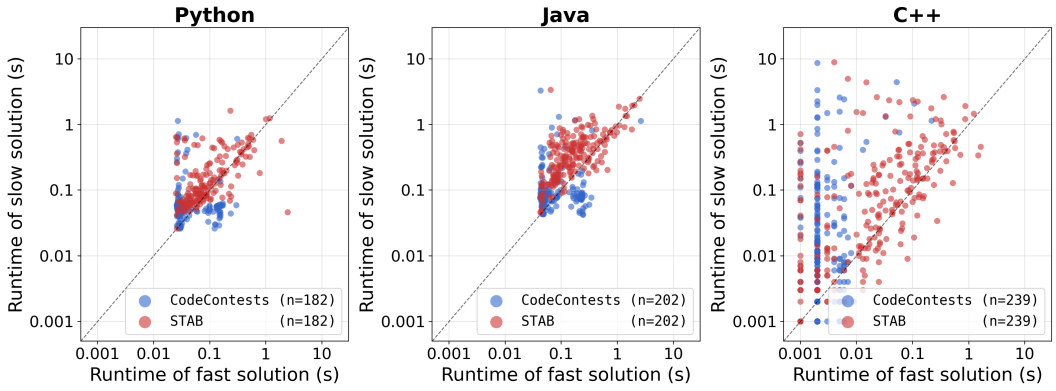

Figure 2: Runtime comparison of test cases selected from $\mathcal{T}_{\text{CC}}(p)$ (blue) and $\mathcal{T}_{\text{gen}}(p)$ (red) on $s_{p,\text{fast}}$ and $s_{p,\text{slow}}$ for Gemma-4. Points above the diagonal indicate that $s_{p,\text{slow}}$ runs longer than $s_{p,\text{fast}}$.

7 Conclusion

We present **STAB**, a specification-driven pipeline that generates efficiency test inputs from the problem specification alone. STAB combines constraint saturation for admissible boundary assignments with adversarial scenario retrieval from a 13-scenario catalog. On CodeContests, STAB outperforms standard prompting across three languages and four LLMs, and surpasses prior efficiency-oriented generators in the evaluated settings, all without using a reference implementation. Appendix J discusses limitations of the scenario catalog and constraint parser.

AI USE STATEMENT

We used the LLMs described in Section 5.2 to synthesize Python generator functions for efficiency test case generation. Their roles in STAB and the experimental settings are documented in Sections 4 and 5. The resulting test cases were evaluated by executing the evaluation solutions in the DOMjudge environment described in Appendix C.

During manuscript preparation, we used ChatGPT to assist with language editing. The authors reviewed and revised the AI-assisted text. The authors take responsibility for the final manuscript, including its methodology, experimental results, and conclusions.

ETHICS STATEMENT

STAB reuses adversarial construction principles documented in the algorithmic complexity attack literature (McIlroy, 1999; Crosby & Wallach, 2003). The constraint saturator keeps generated inputs within the size constraints in the problem specification, so STAB is intended for benchmark efficiency testing rather than attacks against deployed services. Use of STAB outside competitive programming benchmarks should be limited to controlled efficiency profiling.

REPRODUCIBILITY STATEMENT

The STAB code and the procedures for test case generation and evaluation are provided to support reproducibility. Section 4 describes the pipeline, and Appendix M provides the full adversarial scenario catalog. Dataset filtering, evaluation solution selection, and KNN anchor pool construction are described in Appendix B. Section 5.2 specifies the evaluated models and baselines, while Appendix A reports the hardware, LLM sampling parameters, and number of generated efficiency test cases per problem. The DOMjudge execution environment used for runtime measurement is described in Appendix C. The code is available at `https://anonymous.4open.science/r/Slow_TC-E2A8`.

REFERENCES

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

Jianfeng Cai, Jinhua Zhu, Ruopei Sun, Kangwen Zhao, Dongyun Xue, Mingxiao Feng, Wengang Zhou, and Houqiang Li. CodeContests-O: Powering LLMs via feedback-driven iterative test case generation. In *Findings of the Association for Computational Linguistics: ACL*, pp. 1054–1072, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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, Joshua 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. arXiv preprint arXiv:2107.03374, 2021.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. ChatUniTest: A framework for LLM-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE*, pp. 572–576, 2024.

Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

Jaap Eldering, Nicky Gerritsen, Keith Johnson, Thijs Kinkhorst, Mart Pluijmaekers, Michael Vasseur, and Tobias Werth. DOMjudge: Programming contest jury system. https://www.domjudge.org/, 2004.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. arXiv preprint arXiv:2401.14196, 2024.

Zhongmou He, Yee Man Choi, Kexun Zhang, Ivan Bercovich, Jiabao Ji, Junting Zhou, Dejia Xu, Aidan Zhang, Yixiao Zeng, and Lei Li. HardTestGen: A high-quality RL verifier generation pipeline for LLM algorithmic coding. In *Proceedings of the 14th International Conference on Learning Representations, ICLR*, 2026.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, NeurIPS*, 2021.

Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie Zhang. EffiLearner: Enhancing efficiency of generated code via self-optimization. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems, NeurIPS*, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. arXiv preprint arXiv:2409.12186, 2024.

Antti Laaksonen. *Guide to Competitive Programming - Learning and Improving Algorithms Through Contests*. Undergraduate Topics in Computer Science. Springer, 2017. ISBN 978-3-319-72546-8.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE*, pp. 919–931, 2023.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: May the source be with you! *Transactions on Machine Learning Research*, 2023.

Yujia Li, David H. 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. arXiv preprint arXiv:2203.07814, 2022.

Soohan Lim, Joonghyuk Hahn, Hyunwoo Park, Sang-Ki Ko, and Yo-Sub Han. ContractEval: A benchmark for evaluating contract-satisfying assertions in code generation. In *Findings of the Association for Computational Linguistics: ACL*, pp. 42549–42566, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems, NeurIPS*, 2023.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. In *Proceedings of the 1st Conference on Language Modeling, COLM*, 2024.

M. Douglas McIlroy. A killer adversary for quicksort. *Software: Practice and Experience*, 29(4): 341–344, 1999.

Yun Peng, Akhilesh Deepak Gotmare, Michael R. Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. PerfCodeGen: Improving performance of LLM generated code with execution feedback. In *Proceedings of the 2nd IEEE/ACM International Conference on AI Foundation Models and Software Engineering, FORGE*, pp. 1–13, 2025.

Laurent Perron and Vincent Furnon. OR-Tools. `https://developers.google.com/optimization/`, 2024.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. arXiv preprint arXiv:2401.08500, 2024.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950, 2023.

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using LLM. *Proceedings of the ACM on Software Engineering*, 1:951–971, 2024.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024.

Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011. ISBN 978-0-321-57351-3.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 13818–13843, 2023.

Jun Yang, Cheng-Chi Wang, Bogdan Alexandru Stoica, and Kexin Pei. WEDGE: Synthesizing performance constraints for evaluating and improving code efficiency. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems, NeurIPS*, 2025.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving ChatGPT for unit test generation. *Proceedings of the ACM on Software Engineering*, 1:1703–1726, 2024.

Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. exlong: Generating exceptional behavior tests with large language models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering, ICSE*, pp. 1462–1474, 2025.

Linghan Zhong, Samuel Yuan, Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. A tool for generating exceptional behavior tests with large language models. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE*, pp. 1193–1197, 2025.

A Implementation Details

We evaluate 270 CodeContests problems after filtering, consisting of 157 `test` problems and 113 `valid` problems. For scenario retrieval, we build the KNN anchor pool from 5,306 training problems matched by the 13-scenario catalog's keyword matcher. All experiments were conducted using three NVIDIA RTX PRO 6000 GPUs. For LLM sampling, we used temperature $= 1.0$, top-$p = 0.95$, and a maximum output length of 4096 tokens. For each problem, we generated five efficiency test cases. Generated test cases were judged in a containerized DOMjudge environment to measure runtime for each solution and language. All solutions were compiled and executed inside DOMjudge judgehosts under the time limit specified for each CodeContests problem.

B Detailed Dataset Construction

We use the CodeContests dataset (Li et al., 2022). The CodeContests evaluation splits contain 165 `test` problems and 117 `valid` problems. We first executed candidate accepted solutions on the CodeContests test cases and excluded problems for which at least one evaluation solution failed. This filtering removed 8 `test` problems and 4 `valid` problems, leaving 157 `test` problems and 113 `valid` problems, for a total of 270 evaluated problems. This filtering ensures that runtime thresholds are computed only for problems whose evaluation solutions pass $\mathcal{T}_{\mathrm{CC}}(p)$. The CodeContests training split contains 13,328 problems. We do not use the training split for runtime evaluation. Instead, we use it only to build the KNN anchor pool for the adversarial scenario injector. We apply the 13-scenario catalog's keyword matcher to the training problem descriptions and retain 5,306 matched problems, covering 39.8% of the training split. This anchor construction uses only problem specifications and does not use solution code, generated test cases, runtime measurements, or evaluation outcomes.

Solutions in CodeContests include multiple implementation languages, including Python 2. We excluded Python 2 because it is an End-of-Life environment without official support. Accordingly, we evaluated Python 3, Java, and C++ under the problem-specific time limits provided by CodeContests. For each problem p and implementation language, we measured the mean runtime of every $s \in \mathcal{A}(p)$ on $\mathcal{T}_{\mathrm{CC}}(p)$. We then selected $s_{p,\mathrm{fast}}$, $s_{p,\mathrm{slow}}$, and s_{p,r_i} for $i \in \{1, 2, 3\}$ as the five evaluation solutions. All five evaluation solutions passed $\mathcal{T}_{\mathrm{CC}}(p)$ under the DOMjudge environment. We ran $\mathcal{T}_{\mathrm{gen}}(p)$ on these evaluation solutions to identify runtime bottlenecks not exposed by $\mathcal{T}_{\mathrm{CC}}(p)$.

C DOMjudge Execution Environment

General operating system environments or simple sandboxes can introduce runtime variance across executions due to background processes, CPU scheduling fluctuations, and hardware cache effects. DOMjudge judgehosts execute submissions under controlled restrictions, using cgroups for process isolation, memory accounting, network restrictions, and CPU binding. DOMjudge enforces time limits primarily using CPU time, with a hard wall-clock limit as a secondary guard, and reports runtime and memory usage under this judgehost environment.

Consequently, this controlled environment reduces distortions from system noise while preserving language-specific runtime behavior. It provides a stable basis for measuring the time costs imposed by $\mathcal{T}_{\mathrm{gen}}(p)$ on the evaluation solutions $s_{p,r}$.

D Comparison with Additional Code-Based Methods

We evaluate HARDTESTGEN (He et al., 2026) and CodeContests-O (Cai et al., 2026), which use problem-specific code primarily to distinguish correct programs from rejectable ones. This comparison examines whether correctness-oriented objectives transfer to runtime bottleneck exposure. Table 5 shows that STAB achieves the highest ASR in all three implementation languages. Averaged across Python, Java, and C++, HARDTESTGEN, CodeContests-O, and STAB achieve 22.85%, 47.78%, and 70.83% ASR, respectively. The gap is not primarily attributable to invalid inputs. HARDTESTGEN and CodeContests-O generate test cases that satisfy the problem constraints in 99.7% and 93.0% of cases, respectively, and the resulting inputs execute correctly on the evaluation solutions $s_{p,r}$. Their lower ASR instead reflects the objective used to select test cases.

HARDTESTGEN targets failures in brute-force or otherwise rejectable programs, including failures caused by exceeding the time limit. A test case is successful when it causes such a program to time out or distinguishes it from a correct program. Its runtime signal therefore separates rejectable programs from correct programs rather than maximizing the slowdown of an evaluation solution that already satisfies the time limit.

CodeContests-O optimizes correctness discrimination through iterative feedback and does not include an explicit runtime objective. Its objective can favor larger inputs because an inefficient incorrect program can behave like a correct program on small test cases and diverge only when the input becomes sufficiently large. This indirect pressure helps explain why CodeContests-O achieves higher ASR than HARDTESTGEN. Once a test case distinguishes the correct and incorrect programs, however, the objective provides no incentive to increase the runtime of the correct program or to construct the problem-level adversarial structure needed to expose its runtime bottleneck.

ASR evaluates a different target. It runs each generated efficiency test case on evaluation solutions that already satisfy the time limit and checks whether the test exceeds the CodeContests runtime threshold for the same evaluation solution $s_{p,r}$. A valid test case can therefore provide sufficient correctness discrimination while imposing limited runtime pressure on an evaluation solution. The results show that the correctness-oriented objectives of HARDTESTGEN and CodeContests-O transfer only partially to runtime bottleneck exposure. STAB achieves higher ASR while remaining the only method in this comparison that does not consume a problem-specific implementation.

Table 5: Comparison with additional code-based stress-test generation methods on GPT-5.4.

Method	$\mathrm{ASR}_{\mathrm{Python}}$	$\mathrm{ASR}_{\mathrm{Java}}$	$\mathrm{ASR}_{\mathrm{C++}}$
HARDTESTGEN	6.18%	37.18%	25.19%
CodeContests-O	43.25%	52.13%	47.97%
STAB	**70.88%**	**73.12%**	**68.48%**

E Retrieval Strategy Ablation

We evaluated eight retrieval configurations on GPT-5.4 using 50 randomly sampled problems. For each problem and implementation language, generated tests were evaluated on $s_{p,\mathrm{fast}}$ and $s_{p,\mathrm{slow}}$. All configurations shared the same problem specification and M1-resolved boundary; only the retrieval output supplied by M2 differed. *No retrieval* supplied no construction principle, *Full catalog* supplied the complete scenario catalog, and *Keyword only* supplied no scenario when no keyword matched. *Union* injected the union of Keyword Match and KNN outputs, whereas STAB uses Keyword Match when available and invokes KNN only otherwise.

Table 6 shows that selective retrieval is more effective than unrestricted catalog exposure. Averaged across the three implementation languages, STAB achieves 76.24% ASR, compared with 73.71% for Union, 71.34% for the strongest KNN configuration ($k = 2$), 68.98% for No retrieval, and 67.07% for Full catalog. Keyword only records 52.53%, which is consistent with specifications that imply an algorithmic structure without naming it explicitly. Among the KNN configurations, $k = 2$ performs best. The current design uses a fixed $k = 2$ and no similarity threshold.

F Size-Only vs. Structure-Only Buckets

We conduct a bucketed ablation to examine whether M1's larger contribution reflects a benchmark dominated by size-driven bottlenecks. A problem is assigned to the size-only or structure-only bucket when its matched implementations belong exclusively to the Size Only or Structural class, respectively. The former captures problems for which input volume is sufficient to increase runtime, whereas the latter requires an order, shape, or topology in addition to scale. Excluding numerical and mixed-class problems leaves 25 size-only and 33 structure-only problems. Table 7 compares STAB with the variants that remove M1 or M2 on GPT-5.4 and Gemini-3.1, averaging ASR over three implementation languages and five evaluation solutions.

Table 6: Ablation of eight M2 retrieval configurations on GPT-5.4 using 50 randomly sampled problems.

Retrieval	$\mathrm{ASR}_{\mathrm{Python}}$	$\mathrm{ASR}_{\mathrm{Java}}$	$\mathrm{ASR}_{\mathrm{C++}}$
No retrieval	65.71%	72.43%	68.80%
Full catalog	58.00%	71.62%	71.60%
Keyword only	46.57%	56.22%	54.80%
KNN (k=1)	64.57%	72.70%	72.00%
KNN (k=2)	65.14%	74.86%	74.00%
KNN (k=5)	62.57%	71.08%	73.00%
Union	69.71%	77.03%	74.40%
STAB	**72.86%**	**80.27%**	**75.60%**

M1's contribution is not concentrated in size-only problems. Removing M1 produces a larger relative ASR decrease in the structure-only bucket than in the size-only bucket: 18.46% versus 13.93% for GPT-5.4 and 4.84% versus 1.97% for Gemini-3.1. The Structural label indicates that scale alone is insufficient, not that scale is irrelevant. A reverse-sorted sequence or a bamboo tree still requires enough elements or vertices for its structural worst case to produce an observable runtime effect. M1 therefore supplies the resolved boundary needed to instantiate the construction principle from M2 at sufficient scale.

M2 also contributes more strongly to structure-only problems. Removing M2 decreases ASR by 11.00% and 5.69% in the GPT-5.4 structure-only and size-only buckets, respectively; the corresponding decreases for Gemini-3.1 are 5.35% and 0.59%. In size-only problems, reaching the resolved boundary increases the workload regardless of input arrangement. In structure-only problems, a maximum-size input can still miss the slow path when its order or topology is generic. STAB achieves the highest ASR in all four model–bucket combinations, confirming that M1 determines the admissible scale while M2 determines how that scale is structured to expose the bottleneck.

Table 7: Size-only vs. structure-only ablation.

Model	Bucket	−M1	−M2	STAB
GPT-5.4	size-only	65.47%	71.74%	**76.07%**
	structure-only	63.58%	69.39%	**77.97%**
Gemini-3.1	size-only	77.05%	78.14%	**78.60%**
	structure-only	77.89%	77.47%	**81.85%**

G Reference-Implementation Evaluation

Table 8 examines prior methods in their most favorable setting: a C++ reference implementation is provided as input, and the generated test cases are evaluated on that same reference implementation. This setting allows EvalPerf and WEDGE to specialize their test cases to the code used during evaluation. However, their C++ ASR remains limited even under this reference implementation setting. On Gemini-3.1, EvalPerf reaches 22.52% ASR and WEDGE reaches 43.21%. On GPT-5.4, the corresponding values are 35.65% and 41.83%. These results indicate that access to the evaluated reference implementation does not by itself produce test cases that consistently expose efficiency bottlenecks.

The reference implementation setting reveals two failure modes. First, EvalPerf is mainly driven by input scaling. Because this scaling is not paired with explicit constraint resolution or an adversarial construction principle, the generated test cases can violate coupled constraints or remain large but structurally ordinary. Second, WEDGE derives slowdown conditions from executions of a reference implementation. This signal is strongest when the benchmark executions reveal a clear fast-slow runtime contrast, but it becomes weak when such a contrast is absent. Thus, although WEDGE's conditions can in principle be more targeted than EvalPerf's, their reliance on a specific reference implementation prevents consistent recovery of the adversarial construction principle needed to expose

Table 8: C++ ASR of prior efficiency-oriented generators under the reference implementation setting.

Model	Method	$\mathrm{ASR}_{\mathrm{C++}}$
Gemini-3.1	EvalPerf	22.52%
	WEDGE	43.21%
GPT-5.4	EvalPerf	35.65%
	WEDGE	41.83%

the algorithmic bottleneck. STAB avoids these failures by fixing both parts before generator synthesis: a validated boundary from the constraint saturator and an adversarial construction principle from the adversarial scenario injector. The LLM therefore synthesizes a generator from an already constrained and structured specification, rather than relying on input scaling or execution-derived slowdown signals. Together, these results support the main comparison: prior methods fall behind because they miss the combination of valid boundary resolution and adversarial construction principles that STAB encodes explicitly.

H Case Study: Complementary Modules

We present a generator-level case study on 1582_F1, *Korney Korneevich and XOR*, to show how the two STAB modules provide complementary information. The problem gives an array a of length n, where $1 \le n \le 10^5$ and $0 \le a_i \le 500$, and asks for every value $x \ge 0$ that can be obtained as the bitwise XOR of a strictly increasing subsequence of a. Here, increasing refers to the selected values rather than their positions. A valid subsequence must therefore satisfy $i_1 < i_2 < \cdots < i_m$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_m}$. Common solutions track which XOR values are reachable together with the smallest possible last selected value. Therefore, a strong efficiency test must satisfy two conditions: the input length n must be large enough to increase the number of processed elements, and the value distribution must activate a large portion of the $(\text{last_value}, \text{current_xor})$ state space. This makes the problem a balanced case in which boundary resolution and adversarial scenario guidance are both necessary.

Figure 3 shows how the STAB generator combines the boundary resolved by the constraint saturator with the construction principle supplied by the adversarial scenario injector. The constraint saturator resolves $n = 100{,}000$, saturating the valid boundary. Within this boundary, the generator first repeats a 502-element cycle formed by the increasing triples $(t, 2t, 3t)$ for $1 \le t \le 166$, followed by the extreme-value pattern $(0, 500, 1, 499)$. This base cycle contains 335 distinct values. The generator then replaces the first 2,000 positions of every 10,000-element block with a consecutive sequence modulo 501, shifting its starting value by 83 between blocks. Each replaced segment therefore traverses the full admissible value range. The resulting array combines repeated increasing transitions, alternating extreme values, and shifted full-range sweeps at the maximum valid length. This structure provides repeated opportunities to extend strictly increasing subsequences and update XOR states across the value range. In the case-study evaluation, the generated test exceeded the CodeContests runtime threshold for all five evaluation solutions, including the solution not exposed by the generator using only the constraint saturator.

Figure 4 shows the variant without the adversarial scenario injector. This variant keeps the resolved boundary but loses the adversarial value structure. It also uses $n = 100{,}000$, but divides the array into 20 constant blocks drawn from only 11 distinct values. These long constant regions provide substantially fewer increasing transitions than the repeated triples and full-range sweeps used by STAB. The resulting value distribution activates the reachable XOR states less aggressively, even though both generators use the same input length. Consequently, this test does not drive all five evaluation solutions beyond their CodeContests runtime thresholds. This result shows that boundary saturation alone is insufficient when the values within that boundary have limited diversity and structure.

Figure 5 shows the variant without the constraint saturator. This variant exhibits the opposite failure mode. Without the resolved boundary, it infers $n = 20$ from the problem specification. Given this inferred scale, the generator uses an XOR-diverse construction, $(i \cdot 7) \bmod 31$, guided by the adversarial scenario injector. This construction uses the small inferred length effectively: since 7

STAB generator for 1582_F1

```python
def generate() -> str:
    n = 100000
    cycle = []
    for t in range(1, 167):
        cycle.extend([t % 501, (2*t) % 501, (3*t) % 501])
    cycle.extend([0, 500, 1, 499])
    a = []
    while len(a) < n: a.extend(cycle)
    a = a[:n]
    inc = 0
    for k in range(0, n, 10000):
        for j in range(2000):
            if k + j < n: a[k+j] = (inc + j) % 501
        inc = (inc + 83) % 501
    return f"{n}\n" + " ".join(map(str, a))
```

Figure 3: Generator synthesized by GPT-5.4 using STAB for 1582_F1, *Korney Korneevich and XOR*.

Constraint Saturator generator for 1582_F1

```python
def generate() -> str:
    n = 100000
    values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    blocks = 20
    per_block = n // blocks
    a = []
    for b in range(blocks):
        a.extend([values[b % len(values)]] * per_block)
    a.extend([values[-1]] * (n - len(a)))
    return "\n".join([str(n), " ".join(map(str, a))])
```

Figure 4: Generator synthesized by GPT-5.4 using only the constraint saturator for 1582_F1.

is coprime to 31, the first 20 values are distinct and avoid trivial repetition. However, the inferred length is 5,000 times smaller than the valid upper bound 10^5. The construction is reasonable within the inferred scale, but the input is too small to create enough outer loop work. This result shows that adversarial construction alone remains under-scaled without boundary resolution.

This case study highlights the complementary roles of the two components. Without the adversarial scenario injector, the generator reaches the valid boundary but fills it with low-diversity constant blocks. Without the constraint saturator, the generator produces a diverse value sequence but instantiates it at $n = 20$, which is 5,000 times smaller than the valid upper bound. STAB combines the maximum valid length with increasing triples, extreme-value alternation, and shifted full-range sweeps. The resulting test case is therefore both large and structurally adversarial.

I Module-Level Accuracy

Table 9 directly evaluates whether M1 and M2 produce correct intermediate outputs. For M1, we compared the extracted and resolved boundaries with annotated boundary constraints at the constraint level. M1 achieves 90.66% accuracy. The remaining errors consist of omitted prose-only constraints (5.3%), unsupported notation patterns (2.9%), and parsing errors (1.1%). M1 uses a regex-based parser rather than an LLM. Constraints omitted from the resolved boundary remain available in the problem specification supplied to the generator, while the conservative parser avoids injecting unsupported bounds as boundary guidance.

M2 is evaluated at the problem level. Keyword Match runs first, and KNN is used only when Keyword Match retrieves no scenario. A retrieval is correct when at least one retrieved scenario belongs to the

Adversarial Scenario Injector generator for 1582_F1

```python
def generate() -> str:
    n = 20
    a = [(i * 7) % 31 for i in range(n)]
    return "\n".join([str(n), " ".join(map(str, a))])
```

Figure 5: Generator synthesized by GPT-5.4 using only the adversarial scenario injector for 1582_F1.

annotated relevant-scenario set. M2 achieves 72.59% overall accuracy, with 70.40% for Keyword Match and 74.48% for KNN. Keyword Match errors mainly arise from lexical overlap between related scenarios, whereas KNN errors occur more frequently for scenarios represented by fewer anchors. These evaluations complement the end-to-end ASR ablation by directly characterizing the reliability and failure modes of both modules.

Table 9: Module-level accuracy against ground truth. M1 is evaluated per constraint and M2 per problem.

Component	Accuracy
Module 1 (Constraint Saturator)	90.66%
Module 2 (Adversarial Scenario Injector)	72.59%
(a) Keyword Match	70.40%
(b) KNN	74.48%

J LIMITATIONS

STAB is designed to make efficiency test case generation for algorithmic bottlenecks both reproducible and specification-driven, but several practical constraints define its current scope and point to natural extensions.

Coverage of the scenario catalog. The adversarial scenario catalog spans 13 scenarios that correspond to algorithm families commonly encountered in competitive programming. The adversarial scenario injector retrieves multiple matching scenarios per problem and encodes their construction principles into the structured generation specification, but two cases remain outside its current coverage. Algorithm families not represented in the catalog have no corresponding construction principle to retrieve. Composite problems whose worst-case behavior emerges from interactions between scenarios receive only the union of individual construction principles, rather than an interaction-specific principle that captures the combined bottleneck. Both gaps require manual extension of the catalog or its interaction structure, which the current pipeline does not automate.

Robustness of the constraint parser. The constraint saturator uses a regular expression parser designed for common constraint formats, including ranges, chained inequalities, product bounds, and aggregate size bounds. This parser covers the constraint expressions that appear frequently in CodeContests, but it does not cover every way constraints can be written. Constraints with deeply nested conditions, implicit dependencies across variables, or prose-only restrictions can remain difficult to extract with regular expressions. Extending STAB to benchmarks with more irregular constraint descriptions requires a richer parser or an LLM-assisted constraint extraction step.

K EXECUTION VALIDITY OF STAB-GENERATED TEST CASES

We separately evaluated the execution validity of the efficiency test cases generated by STAB. For each execution record, we first ran the corresponding accepted evaluation solution on the generated input to obtain the expected output. We then evaluated the solution in DOMjudge using the resulting input–output pair. We categorized each execution into three outcomes: *Validity* for Accepted (AC),

Timeout for time limit exceeded (TLE), and *Invalid* for wrong answer or runtime error (WA/RE). Timeout was reported separately because TLE reflects excessive runtime rather than an incorrect output or runtime error.

Table 10 reports the execution outcomes for the four LLMs. Each percentage is computed over the evaluated execution records for the corresponding LLM. Validity ranges from 95.42% for Gemini-3.1 to 97.06% for Gemma-4, while Invalid accounts for 2.68–4.20% of the evaluated executions. Timeout accounts for 0.26–0.80% of the executions. Thus, 95.42–97.06% of the evaluated executions complete successfully and match the expected output, whereas 2.68–4.20% result in WA or RE. Execution validity is reported separately from ASR because the two evaluations characterize different properties. These results demonstrate that STAB achieves consistently high execution validity across four LLMs.

Table 10: Execution validity evaluation of STAB-generated test cases.

Model	Validity (AC)	Timeout (TLE)	Invalid (WA/RE)
Qwen-3.5	95.43%	0.37%	4.20%
Gemma-4	97.06%	0.26%	2.68%
Gemini-3.1	95.42%	0.53%	4.05%
GPT-5.4	95.75%	0.80%	3.45%

L Per-Solution Performance Analysis

Table 11 reports ASR separately for the five evaluation solutions $s_{p,r}$. The Fast and Slow columns correspond to $s_{p,\text{fast}}$ and $s_{p,\text{slow}}$, respectively. The R1, R2, and R3 columns correspond to s_{p,r_i} for $i \in \{1, 2, 3\}$. The Avg. column reports the mean ASR across the five evaluation solutions. This breakdown provides the per-evaluation-solution values used in the consistency analysis.

M Full Adversarial Scenario Catalog

Tables 12 and 13 report STAB's adversarial scenario catalog, covering 13 scenarios and 51 implementations. For each implementation, the tables list its vulnerability class and the adversarial construction principle a test case must preserve. The vulnerability class indicates whether the construction principle depends on input arrangement, exact values, or only the maximum admissible input size. Three vulnerability classes are defined as follows.

- *Structural.* A structural vulnerability arises when the relative arrangement of the input forces an implementation toward its worst case. The exact values are not essential, provided that the arrangement remains unchanged. For example, quicksort with a first or last pivot runs in $O(n^2)$ time on any strictly ascending sequence, since each partition removes only one element. Depth-first search via system recursion can overflow the call stack on a bamboo tree, since the recursion depth becomes n on a chain of n nodes. This class captures constructions whose effectiveness depends on shape, order, or adjacency rather than on exact numeric identities.
- *Numerical.* A numerical vulnerability arises when the slow input must satisfy value-specific conditions tied to the algorithm or to implementation parameters. The arrangement alone is insufficient, because replacing the values with arbitrary values of the same size destroys the condition that triggers the slow path. For example, Euclidean GCD performs its maximum number of divisions on consecutive Fibonacci values (F_k, F_{k+1}), because each quotient remains equal to one until the recurrence unwinds. An identity hash table with bucket count B collapses into one bucket when all keys satisfy $k_i \equiv 0 \pmod{B}$. Rabin–Karp with base b and modulus q can degrade to repeated verification when the text and pattern are constructed so that distinct substrings share the same polynomial hash modulo q. This class captures constructions whose effectiveness depends on arithmetic relations such as recurrence or divisibility.
- *Size Only.* A size-only vulnerability arises when neither the arrangement nor the values change the implementation's asymptotic runtime. The slow input is obtained by setting the size parameters to the largest values allowed by the specification. For example, AVL tree

Table 11: ASR across the five evaluation solutions $s_{p,r}$. Fast and Slow correspond to $s_{p,\text{fast}}$ and $s_{p,\text{slow}}$, respectively, and R1, R2, and R3 correspond to s_{p,r_i} for $i \in \{1, 2, 3\}$. Range is the difference between the maximum and minimum ASR across the five evaluation solutions.

Python								
Model	Method	Fast	Slow	R1	R2	R3	Range	Avg.
Qwen-3.5	Base	26.57%	57.10%	70.34%	59.90%	53.82%	43.77%p	53.55%
	STAB	56.14%	74.20%	75.85%	74.30%	73.53%	19.71%p	**70.80%**
Gemma-4	Base	42.32%	65.80%	43.29%	40.87%	41.16%	24.93%p	46.69%
	STAB	60.00%	79.23%	82.03%	79.61%	77.87%	22.03%p	**75.75%**
Gemini-3.1	Base	34.11%	54.88%	56.81%	54.88%	53.14%	22.70%p	50.76%
	STAB	59.23%	75.65%	79.81%	77.78%	75.94%	20.58%p	**73.68%**
GPT-5.4	Base	51.88%	68.21%	69.76%	68.50%	65.89%	17.88%p	64.85%
	STAB	56.04%	73.91%	76.81%	74.78%	72.85%	20.77%p	**70.88%**
Java								
Model	Method	Fast	Slow	R1	R2	R3	Range	Avg.
Qwen-3.5	Base	27.60%	56.65%	69.05%	54.66%	54.57%	41.45%p	52.51%
	STAB	59.46%	75.11%	76.47%	77.83%	77.92%	18.46%p	**73.36%**
Gemma-4	Base	41.54%	57.38%	42.17%	41.45%	40.36%	17.02%p	44.58%
	STAB	63.08%	82.17%	82.35%	82.62%	83.98%	20.90%p	**78.84%**
Gemini-3.1	Base	35.48%	58.91%	60.54%	58.82%	60.00%	25.06%p	54.75%
	STAB	59.82%	77.65%	78.82%	80.00%	80.90%	21.08%p	**75.44%**
GPT-5.4	Base	57.47%	74.30%	73.30%	72.67%	75.48%	18.01%p	70.64%
	STAB	58.01%	75.66%	78.55%	76.65%	76.74%	20.54%p	**73.12%**
C++								
Model	Method	Fast	Slow	R1	R2	R3	Range	Avg.
Qwen-3.5	Base	34.74%	57.19%	73.19%	60.81%	54.07%	38.45%p	56.00%
	STAB	59.33%	69.11%	71.70%	72.74%	72.37%	13.41%p	**69.05%**
Gemma-4	Base	58.07%	64.52%	41.33%	41.56%	40.59%	23.93%p	49.21%
	STAB	62.22%	71.78%	76.44%	77.48%	76.37%	15.26%p	**72.86%**
Gemini-3.1	Base	32.22%	42.52%	41.78%	44.59%	43.63%	12.37%p	40.95%
	STAB	58.52%	67.11%	72.74%	74.37%	74.67%	16.15%p	**69.48%**
GPT-5.4	Base	53.85%	61.63%	66.15%	66.74%	65.19%	12.89%p	62.71%
	STAB	60.30%	66.59%	71.19%	72.15%	72.15%	11.85%p	**68.48%**

operations keep height $O(\log n)$ for every insertion order, and KMP string matching scans the text and pattern in $O(n+m)$ time for all string pairs. This class captures implementations whose invariants make input volume the only effective way to increase runtime.

Table 12: Full catalog of STAB's 13 adversarial scenarios across 51 implementations. Class labels are Structural, Numerical, and Size Only. The adversarial construction principle column summarizes trigger conditions rather than full generator recipes.

Scenario	Implementation	Class	Adversarial Construction Principle
Sorting	Bubble sort	Structural	Strictly decreasing $a_i = n - i$
	Insertion sort	Structural	Strictly decreasing $a_i = n - i$
	Selection sort	Size Only	Maximum n, arbitrary permutation
	Merge sort	Structural	Alternating high and low values
	Quicksort (first or last pivot)	Structural	Monotone order against pivot position
	Quicksort (median of three)	Structural	Bentley–McIlroy sequence with extreme sampled median
	Quicksort (random pivot)	Structural	Exposed seed with extreme pivot ranks
	Heapsort (Floyd's sift-down)	Structural	Descending order causing long sift downs
	Shell sort	Structural	Gap-dependent interleaved descending order
Hashing	Hash table (identity, int keys)	Numerical	Keys congruent modulo bucket count
	Hash table (chain, no treeify)	Numerical	Equal hashCode strings such as Aa and BB
	Hash table (chain and treeify)	Numerical	Equal hashCode strings with tree fallback
	Hash table (SipHash, random seed)	Numerical	Exposed seed collision strings
	Rolling hash (single modulus)	Numerical	Collisions under chosen base and modulus
	Rolling hash (double modulus)	Size Only	Maximum volume with no feasible collision budget
Tree traversal	DFS via system recursion	Structural	Bamboo chain of n vertices
	DFS via explicit stack	Size Only	Maximum n, arbitrary shape
	Centroid decomposition	Structural	Star tree centered at one vertex
	LCA (naive parent walking)	Structural	Bamboo tree with deepest leaf queries
	Tree DP (single post-order)	Size Only	Maximum n, arbitrary tree
Graph shortest path	Dijkstra (lazy, PQ duplicates)	Structural	Dense graph with repeated distance decreases
	Dijkstra (proper heap)	Size Only	Maximum V and E
	Bellman–Ford	Size Only	Maximum V and E
	Floyd–Warshall	Size Only	Maximum V
	BFS (unit weight)	Size Only	Maximum V and E
	SPFA	Structural	Chain graph with strategic back edges

Table 13: Full catalog of STAB's 13 adversarial scenarios across 51 implementations. Part 2 covers string matching, BST and segment structures, DP and recursion, number theory, binary search, two pointers, bitmasks, combinatorics, and geometry.

Scenario	Implementation	Class	Adversarial Construction Principle
String matching	Naive string matching	Structural	Text $\mathrm{a}^{n-1}\mathrm{b}$, pattern $\mathrm{a}^{m-1}\mathrm{b}$
	Knuth–Morris–Pratt (KMP)	Size Only	Maximum n and m
	Rabin–Karp (single modulus)	Numerical	Colliding windows under chosen modulus
	Suffix array (basic compare sort)	Structural	Repetitive string with long common prefixes
	Z-algorithm	Size Only	Maximum string length
BST / segment	BST (unbalanced)	Structural	Sorted or reverse-sorted insertion order
	AVL tree	Size Only	Maximum n
	Splay tree	Size Only	Maximum operations, amortized bound intact
	Treap (randomized)	Structural	Exposed seed with monotone priorities
	Segment tree	Size Only	Maximum n and q
DP / recursion	Unmemoized recursion	Size Only	Maximum n for exponential recurrence
	Memoized DP (top-down)	Size Only	Maximum reachable state count
	Iterative bottom-up DP	Size Only	Maximum reachable state count
Number theory	Euclidean GCD	Numerical	Consecutive Fibonacci pair (F_k, F_{k+1})
	Sieve of Eratosthenes	Size Only	Maximum n
	Trial division	Numerical	Prime or semiprime near $N_{\max}$
Binary search	Binary search on answer	Structural	Full range with extreme boundary
	lower_bound on duplicates	Structural	All equal elements with repeated queries
Two pointers	Naive sliding window	Structural	Inputs causing repeated full rescans
Bitmasks	Held–Karp / bitmask DP	Structural	Maximum n, all states reachable
	SOS DP (naive submask iteration)	Structural	Maximum n, all entries nonzero
Combinatorics	Factorial without precompute	Structural	Maximum n, distinct full range queries
	Inclusion-exclusion	Structural	$k = 20$ to 25, all subsets nonzero
Geometry	Convex hull on collinear points	Structural	Maximum collinear point set
	Closest pair on grid (kd-tree)	Structural	Integer grid with tied distances