Can Multi-turn Self-refined Single Agent LMs with Retrieval Solve Hard Coding Problems?

Md Tanzib Hosain[1,3,*], **Md Kishor Morol**[2,3]
[1]American International University-Bangladesh, [2]Cornell University, [3]EliteLab.AI
20-42737-1@student.aiub.edu, mmorol@cornell.edu

Abstract

Among the hardest tasks for humans are those found in competitive programming where problems require sophisticated algorithmic thinking, puzzle solving, and the creation of effective code. As a domain to assess language models (LMs), it has not received enough attention, though. This study presents the ICPC benchmark, which consists of 254 international collegiate programming contest (ICPC) tasks. Each problem includes official analysis, reference code, and sample, high-quality unit, and hidden tests. We are able to develop and evaluate a variety of LM inference techniques for competitive programming with these resources. With zero-shot chain-of-thought prompting, we find that o1 only achieves a 19.1% pass@1 solve rate. With our best inference technique, which combines multi-turn self-judge with reflection and retrieval over episodic information, raises this to 42.2%. Furthermore, we conduct a new human-in-the-loop investigation to gain a deeper understanding of the remaining difficulties. Surprisingly, we discover that o1 can solve 17 out of 18 problems that were previously unsolvable by any model or technique with just a few specific instructions. A footstep toward LMs with grounded, imaginative, and algorithmic thinking is provided by our quantitative findings and qualitative research. We open-source our code at https://github.com/krarittt/zolve.

1 Introduction

A crucial area for assessing and implementing language models (LMs) is code generation. However, several well-known coding benchmarks, including HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), have become saturated with solve rates above 90% due to the scaling of LMs and the development of new inference techniques (Chen et al., 2023; Shinn et al., 2024; Wei et al., 2022; Zhou et al., 2022). We require more difficult benchmarks that highlight the shortcomings of current models, inference techniques and offer practical instincts for enhancing LM's algorithmic reasoning in order to spur additional advancement. Since competitive programming where problems are intended to rigorously assess human reasoning skills in difficult circumstances and the development of innovative algorithms, it is a perfect fit for this endeavor. To thoroughly assess algorithmic reasoning, prior investigations of competitive programming, however, have either lacked full unit test suites, problem analysis, or sufficient problem variety (Jain et al., 2024; Li et al., 2022; Hendrycks et al., 2021).

With 254 difficult competitive programming tasks from previous ICPC (including regional, continental, world final, etc.) contests, we provide a meticulously designed coding benchmark. As well as some sample tuples of inputs, outputs, and explanations, each challenge outlines a job to be completed in a made-up situation. Solving these problems require for both innovative and grounded thinking in addition to a broad variety of mathematical, computational, and common-sense expertise. With using zero-shot chain-of-thought prompting, even the best o1 only achieves a 19.1% pass@1 solution rate. Apart from that, in order to investigate more sophisticated inference-time techniques for competitive programming, our benchmark also gathers official analysis, reference code solutions, and excellent unit and hidden tests for every problem, as well as the relevant teaching materials in the form of competition programming textbooks. Using these resources, we develop a variety of baseline techniques based on take-a-deep-breath prompt (Yang et al., 2024), brainstorm then select (Summers-Stay et al., 2023), zero-shot-CoT (Kojima et al., 2022), LLM Stimuli (Li et al., 2023a), self-reflection (Shinn et al., 2024), few-shot prompting (Brown et al., 2020) and retrieval

[*]Work done while working as a remote RA at QCRI.

augmented generation- semantic and episodic retrieval (Su et al., 2024; Gao et al., 2023; Shypula et al., 2023), and their combinations.

We discover that multi-turn self-judge single agent LMs with retrieval over comparable problems and solutions together with self-reflection increases performance by 120.94% with respective to o1's zero-shot solve rate. Moreover, we conduct a unique human investigation to better understand the limitations and promise of LM reasoning toward competitive programming. In this study, humans engage with LMs in a conversational "tutoring" setup by pointing out errors and providing only a few tips. Interestingly, when we use a human-in-the-loop configuration, o1 solves 17 out of 18 tasks that can ever answer using any inference techniques. This suggests that stronger LMs may eventually be able to include high-quality input, that new techniques for producing such human-level corrective feedback must be developed, and the appropriate criterion for assessing model capabilities beyond the too stringent execution success should be reconsidered.

We require just black-box access to language model generations; no model-internal information, like as likelihoods or gradients, is required. We employ the same technique and prompt templates for all of our tasks. This makes it possible to apply our approach with popular public models that provide interfaces. Additionally, further model generation enhancements like prompt engineering, self-reflection, or retrieval, are orthogonal to the approach.

In summary, the contributions of our work are provided in the following. At first, the benchmark based on contest programming that includes excellent unit and hidden test cases, problem analysis, and supplementary materials is the ICPC benchmark, which we propose. After that, we develop and evaluate several LM inference techniques for contest programming. Later, we provide a unique method that uses a multi-turn self-judge single-agent LMs with retrieval process to increase the reasoning of modern language models. Our findings show that multi-turn self-judge single-agent LMs with retrieval and self-reflection together can significantly improve performance. Finally, we combine automated tests based on execution success with a new human-in-the-loop research to describe the strengths and weaknesses of LMs for contest programming. Latent differences across models are revealed when we discover that only some models are able to correctly integrate feedback.

2 Related Work

2.1 Problem Solving Coding Benchmarks

Numerous studies have examined language model performance on basic program synthesis (Zan et al., 2022; Austin et al., 2021; Chen et al., 2021; Yu et al., 2018) and HumanEval—the industry standard for evaluating new models on code synthesis. But with the help of inference techniques, existing models can tackle HumanEval problems with a 94% success rate (Zhou et al., 2023). This suggests that more challenging, intricate and self-contained coding challenges are required to test the limits of code reasoning. Thus, competitive programming questions have been suggested as a more challenging assessment metric. The majority of these tasks originate from online resources like Topcoder, LeetCode, Codeforces, Atcoder and others (Jain et al., 2024; Huang et al., 2023; Li et al., 2023c, 2022; Hendrycks et al., 2021). Still, a considerable number of these challenges are only described symbolically and lack thorough test cases that define correctness and quality problem evaluations. The model's capacity to use creative reasoning in grounded task environments—a critical skill of well-rounded reasoners—is thus only marginally assessed.

2.2 Inference Time Techniques

According to (Chen et al., 2023; Gao et al., 2023; Madaan et al., 2024; Shinn et al., 2024; Zhou et al., 2023; Le et al., 2022; Yao et al., 2022; Zelikman et al., 2023; Zhou et al., 2023), inference time methods have demonstrated notable success in enhancing reasoning abilities by conditioning generations on environment feedback, task-specific knowledge, natural language reflections, and planned summaries. Nevertheless, only basic program synthesis tasks like HumanEval and MBPP have utilized their usefulness on code domains thus far (Austin et al., 2021; Chen et al., 2021). In this study, we also discuss how well they perform in a far more challenging domain: competitive programming. We also draw inspiration for our retrieval augmented generation implementation from classical case-based reasoning literature (Aamodt and Plaza, 1994; Schank, 1983) and cognitive architectures for human reasoning (Sumers et al., 2023), which reflect the kinds of information that people find helpful in solving problems.

2.3 Human Agent Interaction (HAI)

Agent learning via human-provided feedback under synthetic tasks is examined by (Sumers et al., 2022). The purpose of (Macina et al., 2023) is to offer a set of tutoring guidelines for successfully including LMs in conversation problem solving. In order to assess the models' capacity to react to feedback, we use a set of interaction rulesets from (Shi et al., 2024).

3 Setup

3.1 Benchmarks

Table 1: Problem count based on contest venue. 'WF' and 'CF' denote World and Continental Finals, respectively.

Category	Problems#
WF & CF	167
Regional	87
Total	**254**

From previous ICPC coding competitions, because of lacking strong co-relation with reasoning problem standards (extreme simple problems) we filtered out some problems and finally 254 expert-written, superior competitive programming tasks make up the ICPC benchmark, presented in Figure 1 (For detail selection see Appendix C). An official human-written problem analysis stating the solution in detail with corresponding C++ code, some unit tests (sample and some synthesized tests) and hidden tests (synthesized tests) confirming solution correctness, time and memory limits confirming solution complexity and a problem description with instructions for reading and writing from standard input and output comprise each problem. Synthesized tests were produced from problem constraints with potential edge cases discussed in the official editorials and validated against official solutions to ensure correctness. This approach is standard in competitive programming research, mitigating reliance on public test cases (Schäfer et al., 2023). A model is provided with the problem description, time and memory constraints, and any samples and synthesized tests as unit tests that are available. After that, the model retrieves related reference documents and using that as episodic knowledge (see in Section 3.2) the model must provide a code solution that the same model judge (self-judge) judges and accepts if it enforces correctness and the intended asymptotic efficiency by yielding the predicted results on all unit tests (in this part, we selected the synthesized tests which don't exist in the hidden test cases) within the specified bounds and the process will terminate. In case the code fails on the unit tests, the whole process will repeat again until convergence or reach into the specified iteration (we found that $i = 2$ is ideal for o1 in this scenario–shown in Table 6). After that the solution will execute against the hidden tests to get the final pass/fail results. A custom HTML5 parser is used to gather 254 tasks[1] that explain contest materials. Regular expressions are then used to extract time and memory limits from problem descriptions. We choose 254 competitive programming tasks with complete problem analyses to aid in the creation of rich inference-time techniques and assessments. We parse a ground truth standalone C++ code snippet and an English-only analysis devoid of code for episodic knowledge retrieval. We ask GPT-4 to convert the code to C++ for tasks when C++ code is not accessible and we confirm that all code solutions pass hidden tests on the specified restrictions.

3.2 Baselines

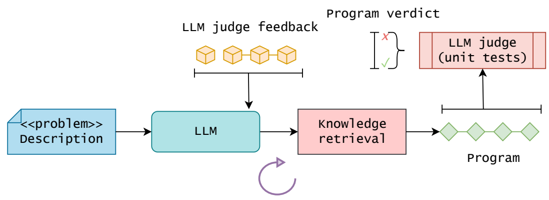

Figure 1: Framework architecture with Knowledge retrieval and Self-reflection.

We test a number of prompting and inference time strategies, including the take-a-deep-breath prompt (Yang et al., 2024), brainstorm then select (Summers-Stay et al., 2023), zero-shot-CoT (Kojima et al., 2022), LLM Stimuli (Li et al., 2023a), self-reflection (Shinn et al., 2024), few-shot prompting (Brown et al., 2020) and retrieval augmented generation- semantic[2] and episodic retrieval (Su et al., 2024; Gao et al., 2023; Shypula et al., 2023). As no single prompt performs better than the others (Table 3), we choose the episodic retrieval with reflection prompt in our single-agent LMs framework (Figure 1). Furthermore, to fully explore the potential of retrieval on the comparatively small dataset, we simulate a setup in which

[1] https://icpc.global/

[2] As our resource, we utilize the Algorithms for Competitive Programming textbook, which includes chapters on algorithmic principles written by humans.
https://cp-algorithms.com/

the model has seen every other problem in the ICPC set aside from the one it is currently solving. This is done by simulating an n-fold evaluation that presents one problem at a time. Although we get comparable results with a more traditional train-test split, as detailed in Section 4.2. Concatenating the problem description, solution and C++ solution code for each seen problem creates documents that may be retrieved. After adjusting for the number of problems to retrieve, p, we determine that $p = 2$ is ideal for o1. As pass@1 performance was declining, we decided not to try resampling for larger amounts of p in order to save budget. As a result, we publish these values (Table 5).

3.3 Metric

We use every method that has a Pass@1 evaluation and the methods from (Shi et al., 2024) for self-reflection and episodic retrieval, and we only give the models the execution outcomes of the exposed unit test cases. Fundamental studies were done using GPT-4, GPT-4o and o1 with some open source models tested in zero-shot setting only.

4 Results

4.1 Performance Baselines

Table 2: Pass@1 performances of various models for zero-shot problem-solving configuration.

Model	**Pass@1**
gpt-4	7.3
claude-3.5-sonnet	14.1
gpt-4o	14.2
qwen2.5-coder	14.8
athene-v2-chat	16.4
deepSeek-v3-chat	17.6
gemini-exp	18.3
o1	**19.1**

As a starting point, we assess the zero-shot performance of models that represent the state-of-the-art coding performance, such as GPT-4 (gpt-4-0613), GPT-4o (gpt-4o-2024-11-20), o1 (o1-2024-12-17), Claude-3.5-Sonnet (claude-3.5-sonnet-20240620), Gemini-Exp (gemini-exp-1206), Athene-V2-Chat (athene-v2-chat-72b), DeepSeek-V3-Chat, and Qwen2.5-Coder (qwen2.5-coder-32b-instruct) (Achiam et al., 2023; Team et al., 2024; Liu et al., 2024a; Hui et al., 2024). Table 2 provides an overview of this. If not otherwise noted, models were given chain-of-thought prompts (Wei et al., 2022); the complete prompts are shown in Appendix A. In accordance with earlier studies on competitive programming (Li et al., 2022; Hendrycks et al., 2021), we mainly use the unbiased pass@n metric as specified in (Chen et al., 2021). For that, we discover that compilation errors are not the primary cause of any model defects (see Section 5). This at least demonstrates that models are successful in producing syntactically sound code and points to more complex problems in generations, including miscommunications.

4.2 Performance Benchmarks

Aligning with (Shi et al., 2024; Shinn et al., 2024; Chen et al., 2023), we discover that stronger models have the emergent quality of being able to self-reflect successfully. Nevertheless, both episodic and semantic retrieval remain efficient; in fact, episodic retrieval even makes GPT-4o come close to o1's zero-shot performance (Table 3). This is probably due to the fact that self-reflection depends on the internal model's capacity to interpret binary, sparse reward signals. Conversely, retrieval enables models to make use of pre-existing logic and code fragments, necessitating less inherent model capabilities. Thus, our results support (Li et al., 2023b), which found that LMs are able to comprehend competitive programming solutions that are far more sophisticated than they are able to generate. Furthermore, combining episodic retrieval with reflection allows it to reach new heights, but not with semantic retrieval. The additional knowledge offered by our implementation of semantic retrieval trades off against its extended contexts, which existing LLMs are known to struggle with (Liu et al., 2024b; Shi et al., 2024). This offers one explanation for why combining the two might result in decreased performance.

Furthermore, instead of the model crucially interacting with the retrieved information itself, the opposing theory for retrieval success holds that adding obtained answers enhances memorizing effects for the problem under evaluation. To check for this, we eliminate crucial portions of the recovered solutions and see notable performance decreases. The created and officially published answers also do not significantly overlap, according to qualitative examination. Section 4.4 contains the experiment specifics.

Additionally, for maximizing the impact of retrieval on the comparatively short dataset at hand, our episodic retrieval assessment setup entails presenting one problem at a time that is retrieves from the solutions of all other test problems, as explained

Table 3: Pass@1 performances for various problem-solving configurations.

Inference technique	Model		
	gpt-4	gpt-4o	o1
zero_shot	7.3	14.2	19.1
brainstorm_then_select	8.6	16.9	21.7
few_shot	10.1	19.4	24.2
self_reflection	11.3	20.6	25.4
semantic_retrieval	12.4	22.1	27.3
semantic_retrieval + self_reflection	12.8	22.5	28.1
episodic_retrieval	13.2	23.3	29.0
semantic_retrieval + episodic_retrieval	14.5	24.4	29.8
semantic_retrieval + episodic_retrieval + self_reflection	16.4	27.1	33.2
episodic_retrieval + self_reflection	**24.3**	**38.4**	**42.2**

Table 4: Pass@1 performances when compared to our leave-one-out episodic retrieval situation, the outcomes of a normal train-test split are comparable across inference-time approaches.

Inference technique	Model		
	gpt-4	gpt-4o	o1
episodic_retrieval	10.9	18.6	22.7
self_reflection	11.1	20.4	24.2
episodic_retrieval + self_reflection	**21.3**	**33.8**	**35.4**

Table 5: o1 hyperparameter tuning on the number of problems to retrieve for episodic retrieval.

Problems	Pass@1
$p = 1$	28.1
$\mathbf{p = 2}$	**29.0**
$p = 3$	28.4

Table 6: o1 iteration tuning on the number of iterations for self-reflection. Without any reflection, the solve rate is $i = 0$. We see that after 2 repetitions, solve rates nearly stay the same.

Iterations	Pass@1
$i = 0$	21.3
$i = 1$	23.8
$\mathbf{i = 2}$	**25.6**
$i = 3$	25.4

in Section 3.2. Given how independent problems are and how little solution logic even problems with the same method type share, we anticipate that this will not result in any notable dataset leaking across evaluations. We did, however, rerun most of the inference-time methods against a more conventional train-test split arrangement. The conventional split, train size = 200, test size = 54 produces comparable results with somewhat lower retrieval efficacy, as seen in Table 4. This is due to the fact that fewer problems are retrieved overall, which results in a generally lower level of problem similarity between the problems that are recovered and the ones that are being addressed at the moment. Moreover, we recover the same optimal values as the leave-one-out configuration by re-tuning the number of recovered passages solely on this train set.

4.3 Performance HAI

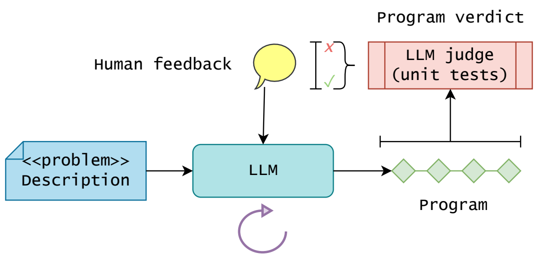

Figure 2: Framework architecture with integrating HAI.

Table 7: Feedback is integrated into o1's HAI interactive setting. (Final solve rate would be highly dependent on the problem-solving strength of the human performing the interactions with the models. For this case study, the participants who participated in this interaction module have Codeforces rating > 2500.)

Model	Final solve rate
gpt-4	0
gpt-4o	0
o1	0
o1 + interact	**94.4**

We discovered a broad range of model error distributions in benchmark assessments, ranging from minor off-by-one implementation problems to severe misconceptions. We conduct a human research using an interactive tutoring to further investigate how close a model is to resolving a particular task (Figure 2). Remarkably, we discover that the human-in-the-loop approach improves o1 performance from 0% to 94.4% (Table 7), 17 problems solved on a small set of 18 problems on which GPT-4, GPT-4o and o1 reach zero pass rate using all of the aforementioned inference-time methods, but does not improve GPT-4 and GPT-4o performance from 0%. When two models fail on a particular problem, one may be one adjustment away from a completely perfect solution, while the other may have a basic misunderstanding of the problem scenario. These human-in-the-loop results demonstrate that the solve rate may not fully represent the capabilities of models. This encourages improved measures for assessment that go beyond execution success, pass@n. As an alternative interpretation of our findings, it is possible that human-level corrective feedback might open more thinking abilities in o1, underscoring the need for improved techniques to produce such feedback. Appendix B contains a scenario of the interaction pathway.

4.4 Ablation Test

Table 8: Performance on various retrieval query ablations.

Query	Pass@1
problem_description	28.5
problem_description + proposed_code_solution	**29.0**
problem_description + proposed_solution + code_solution	29.8

Table 9: Performance on various episodic retrieval ablations.

Retrieval	of max performance
problem_description + code + solution	100.0
problem_description	**2.3**

For the ICPC problemset, we do ablation test on various prompts in order to establish the parameters for the primary experiments.

Apart from that, in the investigation on how the prompts impact problem-solving in a conversation, we create a variety of specific prompts for our suggested self-feedback single agent with retrieval framework. Appendix A incorporates the prompt designs and report the findings, identifying the prompt as the primary prompt for more research.

According to ablations on retrieval queries, the best retrieval queries make use of both the current problem description and a first solution attempt that includes code and an explanation. This makes it possible to accurately obtain pertinent algorithm descriptions from the underlying retrieval corpus, as retrieval over algorithmic keywords is not possible when only the issue descriptions are used. Since our local judge has not seen this first generation, we do not consider it an effort. For that, we found in Table 8, the majority of retrieval queries, in general, are rather effective; nevertheless, the best results are obtained by combining code proposes and proposed solutions, as this enables the greatest possible matching of pertinent keywords across the compared documents. Applying ablations to the corpora in Table 9, we tackle memorizing. If retrieving problem solutions was causing people to recite previously learned answers to the present problem, then eliminating important components of the obtained solutions would not lessen this impact. But we discover that it does: using only the problem description preserves just 2.3% of the performance, indicating that models are actually using the context-provided reasoning of related problems to guide their generations.

5 Errors

Table 10: Error distributions of episodic_retrieval + self_reflection, in %. TLE indicates time limit exceeded, and MLE Indicates memory limit exceeded. 'Other' generally represents errors stemming from models outputting incorrectly formatted code.

Model	Wrong Ans.	TLE	MLE	Runtime	Syntax + Other
gpt-4	58.81	5.33	0	10.16	1.38
gpt-4o	28.95	25.06	0	6.83	0.77
o1	27.87	23.56	0	5.78	0.59

Table 10 indicates on where models are trading raw speed for more profound reasoning capabilities. While gpt-4 provides rapid but often incorrect solutions, gpt-4o and o1 engage in a more computationally expensive process that yields correct answers far more frequently. The o1 model establishes itself as the superior agent in this analysis, demonstrating marginal but consistent gains over gpt-4o in both correctness and efficiency. Future work should investigate methods to mitigate the high computational cost (TLEs) of these advanced models without compromising their newfound accuracy, perhaps through optimized algorithms or more efficient self-reflection triggers.

6 Results Analysis

```
/*
Hungarian Algorithm (a.k.a. Kuhn-Munkres) for MIN-COST matching on an n x m matri
This version can handle the case n <= m by padding if necessary.
Complexity ~ O(n^2 * m).

We'll write a function hungarianMinCost(cost, n, m) that returns the minimal cost
of a perfect matching (matching each of the n rows exactly once with some column)

For maximum-sum matching, we do cost'[r][c] = K - originalWeight, etc.
*/

static const int INF = 1000000000;

// Hungarian algorithm for a rectangular n x m matrix.
// cost[r][c] is the "cost" of matching row r with column c.
// n <= m assumed, each row must match exactly one column (but columns can go unm
// Returns the sum of the chosen matching in "cost".
int hungarianMinCost(const vector<vector<int>>& cost, int n, int m) {
    // "u" and "v" will store the matching potentials for rows and columns.
    // "p" will store the matching for columns: p[c] = the row matched with colum
    // "way" is used in the internal min-finding step.
    vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
    for (int i = 1; i <= n; i++) {
        p[0] = i;
        int j0 = 0; // current column 0
        vector<int> minv(m + 1, INF);
        vector<bool> used(m + 1, false);
        do {
            used[j0] = true;
            int i0 = p[j0], j1 = 0;
            int delta = INF;
```

...

(a) zero_shot: WA

```
class Solution {
public:
    int maxScore(vector<vector<int>>& grid) {
        int n = grid.size();
        int m = grid[0].size();

        // 1) Gather all cells (value + row index)
        vector<array<int, 3>> cells; // {value, row, col}
        cells.reserve(n * m);
        for (int r = 0; r < n; ++r) {
            for (int c = 0; c < m; ++c) {
                cells.push_back({grid[r][c], r, c});
            }
        }

        // 2) Sort cells by value descending
        sort(cells.begin(), cells.end(),
             [](auto &a, auto &b) {
                 return a[0] > b[0];
             });

        // To track which rows/values are already used
        vector<bool> usedRow(n, false);
        // Problem values go up to 100, so let's track usage with size 101
        vector<bool> usedValue(101, false);

        int answer = 0;

        // 3) Greedy: pick from largest to smallest
        for (auto &cell : cells) {
            int val = cell[0];
```

...

(b) semantic retrieval + self_reflection: WA

```
class Solution {
public:
    int maxScore(vector<vector<int>>& grid) {
        int n = (int)grid.size();
        int m = (int)grid[0].size();

        // 1) Collect all distinct values
        unordered_set<int> st;
        for(int r = 0; r < n; ++r){
            for(int c = 0; c < m; ++c){
                st.insert(grid[r][c]);
            }
        }

        // Convert to a sorted vector (descending order)
        vector<int> distinctVals(st.begin(), st.end());
        sort(distinctVals.begin(), distinctVals.end(), greater<int>());
        int k = (int)distinctVals.size();

        // 2) For each distinct value, find which rows contain it
        // rowHasValue[val] = bitmask of rows that have 'val'
        // but since val can go up to 100, we can store it in a map from val->bit
        // Alternatively, we can store for each index in distinctVals, the set of
        vector<int> rowsMask(k, 0);
        // For each row, we gather its values in a set for quick membership:
        vector< unordered_set<int> > rowSets(n);
        for(int r=0; r<n; r++){
            for(int c=0; c<m; c++){
                rowSets[r].insert(grid[r][c]);
            }
        }
```

...

(c) episodic retrieval + self_reflection: AC

Figure 3: Pathway of solving problems of self-feedback single agent with retrieval (P1).

We see in Figure 3 (P1) that, within some trials of incorrect solution, with retrieval + reflection state the reasoning about related problem settings could be inherited by single agent LMs. That is why, the retrieved solution and code gives it access to sample reasoning over this complex and error-prone problem context, enabling single agent LMs to produce code that is more correct.

```
#include <bits/stdc++.h>
using namespace std;

static const int MAXN = 200000;  // per constraints

vector<int> adj[MAXN+1];
long long dp[MAXN+1], w[MAXN+1];
long long global_best;

// We'll write a DFS that computes dp[v] = best path sum starting at v.
void dfs(int v, int p) {
    // dp[v] initially is just w[v]
    dp[v] = w[v];

    // best_down1, best_down2 track the two largest dp[child] among children
    long long best_down1 = 0, best_down2 = 0;

    for (int c: adj[v]) {
        if (c == p) continue;
        dfs(c, v);

        long long candidate = dp[c];
        if (candidate > 0) {
            // see if it improves our top-two
            if (candidate > best_down1) {
                best_down2 = best_down1;
                best_down1 = candidate;
            } else if (candidate > best_down2) {
                best_down2 = candidate;
            }
        }
    }

    // best path that "passes through v" might use v plus up to two children
    long long local_path = w[v] + best_down1 + best_down2;
    global_best = max(global_best, local_path);
```

episodic retrieval + self_reflection: AC

Figure 4: Pathway of solving problems of self-feedback single agent with retrieval (P2).

A textbook chapter on route-removal and tree splitting strategies, which are indirectly related to the problem of eliminating the vertices on a path between two selected nodes, was retrieved by the single agent LMs, shown in Figure 4 (P2). Interestingly, the official editorial's brief reference chapter on the specific tree technique was not retrieved. After closer examination, the chapter's retrieval score was lower since it was noticeably lacking in specifics. This demonstrates how the retrieval engine may be used to filter out less-than-ideal documents and choose more pertinent sources, especially those that deal with increasing the number of connected elements by deliberately deleting a path from a tree. For that, algorithmic notions and textual reasoning can be employed by single agent LMs.

For HAI, while GPT-4's reprises frequently prove ineffective. While GPT-4o was receptive but could not able to reach into the solution state, we discovered that o1 was more receptive to general input that its algorithm or comprehension of an environment notion was flawed and more able to arrive at the right approach on its iterative try. For instance, in the problem Appendix B (P3), o1 demonstrated superior problem-solving through iterative feedback. Initially, when prompted to provide a solution, o1 submitted an incorrect code. After receiving feedback highlighting several bugs and requesting a verification of its understanding, o1 engaged in a constructive dialogue. It analyzed a sample case together with the user, identified the impossibility of tiling in the given scenario, and correctly concluded that the output should be "None".

When prompted to implement the corrected logic based on this understanding, o1 successfully delivered an accurate and accepted solution. In contrast, GPT-4 and GPT-4o fails to make meaningful progress despite similar interaction, highlighting o1's enhanced ability to comprehend and act upon detailed instructions and iterative guidance. Appendix B contains a scenario of iterative interaction pathway.

7 Discussion and Conclusion

At the end, the benchmark of competitive programming problems—complete with official analysis, reference code, and rigorous unit tests—offers a robust platform for evaluating and advancing language models in competitive programming settings. By introducing the self-feedback single agent with retrieval framework, we demonstrate how self-reflection and retrieval of episodic information can substantially improve solve rates. Moreover, the human-in-the-loop study underscores the transformative potential of targeted guidance, enabling solutions to nearly all previously unsolvable problems. Collectively, these findings mark a significant step toward language models that can engage in grounded, imaginative, and algorithmic thinking. We hope this work will illuminate the challenges that lie ahead and provide a strong foundation and a promising roadmap for future research at the intersection of natural language processing and advanced problem solving.

Limitations

This study primarily focuses on competition-level code generation, where it does not studies tasks such as software engineering tasks, e.g., SWE-bench (Jimenez et al., 2023). The method primarily focuses on improving accuracy, while it does not aim for minimizing costs.

References

Agnar Aamodt and Enric Plaza. 1994. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, and 1 others. 2023. Competition-level problems are effective llm evaluators. *arXiv preprint arXiv:2312.02143*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Cheng Li, Jindong Wang, Yixuan Zhang, Kaijie Zhu, Wenxin Hou, Jianxun Lian, Fang Luo, Qiang Yang, and Xing Xie. 2023a. Large language models understand and can be enhanced by emotional stimuli. *arXiv preprint arXiv:2307.11760*.

Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. 2023b. Explaining competitive-level programming solutions using llms. *arXiv preprint arXiv:2307.05337*.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023c. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024b. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Jakub Macina, Nico Daheim, Sankalan Pal Chowdhury, Tanmay Sinha, Manu Kapur, Iryna Gurevych, and Mrinmaya Sachan. 2023. Mathdial: A dialogue tutoring dataset with rich pedagogical properties grounded in math reasoning problems. *arXiv preprint arXiv:2305.14536*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Roger C Schank. 1983. *Dynamic memory: A theory of reminding and learning in computers and people*. cambridge university press.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. 2024. Can language models solve olympiad programming? *arXiv preprint arXiv:2404.10952*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

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

Hongjin Su, Howard Yen, Mengzhou Xia, Weijia Shi, Niklas Muennighoff, Han-yu Wang, Haisu Liu, Quan Shi, Zachary S Siegel, Michael Tang, and 1 others. 2024. Bright: A realistic and challenging benchmark for reasoning-intensive retrieval. *arXiv preprint arXiv:2407.12883*.

Theodore Sumers, Robert Hawkins, Mark K Ho, Tom Griffiths, and Dylan Hadfield-Menell. 2022. How to talk so ai will learn: Instructions, descriptions, and autonomy. *Advances in neural information processing systems*, 35:34762–34775.

Theodore R Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. 2023. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*.

Douglas Summers-Stay, Clare R Voss, and Stephanie M Lukin. 2023. Brainstorm, then select: a generative language model improves its creativity score. In *The AAAI-23 Workshop on Creative AI Across Modalities*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, and 1 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Songhua Yang, Hanjie Zhao, Senbin Zhu, Guangyu Zhou, Hongfei Xu, Yuxiang Jia, and Hongying Zan. 2024. Zhongjing: Enhancing the chinese medical capabilities of large language model through expert feedback and real-world multi-turn dialogue. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19368–19376.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet nl2code: A survey. *arXiv preprint arXiv:2212.09420*.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv:2207.05987*.

A Prompt

ZERO-SHOT

Please reply with a C++ solution to the below problem. Make sure to wrap your code in `'```C++'` and `'```'` Markdown delimiters, and include exactly one block of code with the entire solution (in the final code step).
Reason through the problem and think step by step. Specifically:
1. Restate the problem in plain English.
2. Conceptualize a solution first in plain English.
3. Write a pseudocode solution.
4. Output the final C++ solution with your solution steps in comments.

```
[BEGIN PROBLEM]
{INSERT PROBLEM HERE}
[END PROBLEM]
```

SELF-REFLECTION

You were previously solving a coding problem. Here is the problem that you were solving:
`{problem_dict[query['problem_id']]`
`['description']}`
And here are all your past attempts, as well as how your code fared on the unit tests for the problem:
`{query['reflection_buffer']}`
Think carefully about where you went wrong in your latest solution, first outputting why you think you went wrong. Then, given your insights, try to fix the solution, outputting a block of correct C++ code to be executed and evaluated again. Make sure to wrap your code in `'```C++'` and `'```'` Markdown delimiters.

EPISODIC-RETRIEVAL

Please reply with a C++ solution to the below problem. Make sure to wrap your code in `'```C++'` and `'```'` Markdown delimiters, and include exactly one block of code with the entire solution (in the final code step). You will also be given multiple somewhat similar problems, as well as the solution to those similar problems. Feel free to use those problems to aid your problem-solving process.
1. Restate the problem in plain English.
2. Conceptualize a solution first in plain English.
3. Write a pseudocode solution.
4. Output the final C++ solution with your solution steps in comments.

`[BEGIN SIMILAR PROBLEMS]`
`{query['retrieval_text']}` (Similar problem problem + solution goes here)
`[END SIMILAR PROBLEMS]`
Now it's your turn. Here is the problem you are to solve:
`[BEGIN PROBLEM]`
`{problem_dict[query['problem_id']]`
`['description']}` (Description of problem goes here)
`[END PROBLEM]`

EPISODIC-RETRIEVAL + SELF-REFLECTION

You were previously solving a coding problem. Here is the problem that you were solving:
`{problem_dict[query['problem_id']]`
`['description']}`

You were also given a couple of similar problems to the problem above along with their solutions to aid you in solving the problem at hand. Here are the similar problems you were given:
`{query['retrieval_text']}`

And here was your original response:
`{query['original_response']}`

Here was the judge result of the above solution:
`{query['judge_response']}`

Think carefully about where you went wrong. Then, try to fix the solution, outputting a block of correct C++ code to be executed and evaluated again. Make sure to wrap your code in `'```C++'` and `'```'` Markdown delimiters.

SELF-JUDGE

You are a judge. Your task is to judge the solution of a coding problem. Here is the problem for which the solution you have to judge:
`{problem_dict[query['problem_id']]`
`['description']}`

And here is the solution along with test cases against which to judge:
`{query[['problem_id']]['solution','test_case']}`

Please produce a score (based on the number of test cases passed) with reasoning behind your judgement of the solution to the problem.

RANDOM TEST CASE SYNTHESIZE

You are a programming contest expert. Given a competitive programming problem and it's standard solution code, you need to write a C++ program to generate random test input data for the problem. Please ensure that the generated test data satisfies all constraints in the problem description. Your C++ program should generate a set of valid test input data when executed, which should test the correctness and efficiency of solutions. The range of generated random data should be consistent with the requirements of the problem, do not use small range for simplicity. Your program must use the system's default time as the random seed and output only the test input data (without any extra prompts or commentary). In the end, YOU MUST provide the complete C++ code in a code block enclosed with `'"'C++'` and `'"''` Markdown delimiters.

CORNER TEST CASE SYNTHESIZE

You are a programming contest expert. Given a competitive programming problem and its standard solution code, you need to write a C++ program that generates diverse random test input data for the problem. Unlike standard generators, your program must randomly decide at runtime which type of test input to produce, choosing from multiple types that include edge cases, boundary extreme values, and specially structured cases. You must ensure that the input data generated after each run of this generator and its output data is greatly different and diverse. The generated data must satisfy all constraints detailed in the problem description and cover the full range of allowed values, ensuring that any submitted solution is thoroughly tested for both correctness and efficiency. Your program must use the system's default time as the random seed and output only the test input data (without any extra prompts or commentary). In the end, YOU MUST provide the complete C++ code in a code block enclosed with `'"'C++'` and `'"''` Markdown delimiters.

INTERACTION

You are to interact with a given model to try to solve a given coding question. A problem-solving session ends whenever the model has generated code 3 times. Between code generations, you may speak to the model in conversation as many times as you would like. However, the way you interact with the model must be very specific: your goal is to act akin to a tutor and guide the model towards the right answer, without giving away any underlying details about the true solution of the problems.
You MAY NOT provide the models with:
1. The correct algorithm to fix their initial solution.
2. Exact lines of code where they went wrong.
3. Explanations of the problem or explanations of misunderstandings.

You MAY provide the models with:
1. General concepts/data structures that may be useful for solving the problem.
2. Walking through a sample input-output of the problem to better verify problem understanding.
3. Short concise general directions on where the code went wrong.

Here are examples of correct pathway:

B Qualitative Analysis: HAI

FEEDBACK PHASE 1

P3:
Please reply with a C++ solution to the below problem. Make sure to wrap your code in 'C++' and ' 'Markdown delimiters, and include exactly one block of code with the entire solution. Feel free to either return just the one code block with your solution or the one code block with explanatory text before and/or after – however, you will only be evaluated on the correctness of your code.
`[BEGIN PROBLEM]` Archimedes conducted his famous experiments on buoyancy. But not everyone knows that while he was taking a bath, he was too focused and didn't notice the moment when the water overflowed over the edge of the bath and flooded the floor near the wall. His expensive parquet was irreversibly damaged!

Archimedes noticed that not all was lost, and there were still several undamaged parquet pieces. The parquet near the wall had the shape of a long narrow stripe of 2×n cells. Archimedes had an unlimited supply of 1×2 parquet pieces that could be placed parallel or perpendicular to the wall. Archimedes didn't want to cut the parquet pieces. As a great scientist, he figured out that there was exactly one way to restore the parquet by filling the damaged area of the parquet with the non-overlapping 1×2 cell shaped pieces.

Help historians to check Archimedes' calculations. For the given configuration of the 2×n parquet floor, determine whether there is exactly one way to fill the damaged parquet cells with the 1×2 cell parquet pieces. If Archimedes was wrong, find out whether there are multiple ways to restore the parquet, or there are no ways at all.

RESPONSE PHASE 1: WA

```cpp
    ```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int T;
    cin >> T;
    while(T--){
        int n;
        cin >> n;
        string row0, row1;
        cin >> row0 >> row1;
        // Initialize DP
        // // dp_prev[mask] where mask is 0,1,2,3
        // // Initialize all to 0
        int dp_prev[4] = {0,0,0,0};
        dp_prev[0] = 1;
        for(int i=0; i<n; i++){
            int dp_next[4] = {0,0,0,0};
            for(int mask=0; mask<4; mask++){
                if(dp_prev[mask]==0) continue;
                // Check consistency
                bool invalid = false;
                if( (mask & 1) && row0[i] == '#') invalid = true;
                if( (mask & 2) && row1[i] == '#') invalid = true;
                if(invalid) continue;
                // Compute coverage needs
                bool top_covered = (mask & 1) != 0;
                bool bottom_covered = (mask & 2) != 0;
                bool c0 = (!top_covered && row0[i] == '.');
                bool c1 = (!bottom_covered && row1[i] == '.');
                if(c0 && c1){
                ...
```

FEEDBACK PHASE 2

This code is not correct. There are several bugs in your code, but I think it's important to verify your understanding of the problem. Let's converse and make sure you understand the problem correctly. Let's go through the sample case together:
1
7
....
..#.#..

RESPONSE PHASE 2

We have a 2×7 grid where certain cells are undamaged (#). After analyzing, it's clear that placing 1×2 tiles without overlapping or covering the undamaged cells is impossible. The undamaged cells at positions (1,2) and (1,4) block potential vertical and horizontal placements, leaving some damaged cells uncovered regardless of the tiling strategy. Therefore, the correct output for this case is indeed "None," confirming that there's no valid way to restore the parquet under the given constraints. This aligns with Archimedes' observation that sometimes restoration isn't feasible.

FEEDBACK PHASE 3

Alright. You are right this time! So to verify you understand, can you implement it with your previous response?
```

RESPONSE PHASE 3: AC

```
    ```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int T;
    cin >> T;
    while(T--){
        int n;
        cin >> n;
        string row0, row1;
        cin >> row0 >> row1;

        // DP arrays: dp_prev and dp_next, each with 4 masks
        // Initialize dp_prev to 0 and set dp_prev[0] = 1
        int dp_prev[4] = {0, 0, 0, 0};
        dp_prev[0] = 1;

        for(int i=0; i<n; i++){
            int dp_next[4] = {0, 0, 0, 0};
            for(int mask=0; mask<4; mask++){
                if(dp_prev[mask] == 0) continue;

                // Validate the current mask against the grid
                bool invalid = false;
                // Check top cell
                if( (mask & 1) ){
                    if(row0[i] != '.') invalid = true;
                }
                // Check bottom cell
                if( (mask & 2) ){
                    if(row1[i] != '.') invalid = true;
                    ...
```
```

C Selected Contest Venues

Table 11: Selected ICPC Venues.

Venue	Category
The 2011 ICPC World Final (WF)	World Final (WF)
The 2012 ICPC World Final (WF)	World Final (WF)
The 2013 ICPC World Final (WF)	World Final (WF)
The 2014 ICPC World Final (WF)	World Final (WF)
The 2015 ICPC World Final (WF)	World Final (WF)
The 2016 ICPC World Final (WF)	World Final (WF)
The 2017 ICPC World Final (WF)	World Final (WF)
The 2018 ICPC World Final (WF)	World Final (WF)
The 2019 ICPC World Final (WF)	World Final (WF)
The 2020 ICPC World Final (WF)	World Final (WF)
The 2021 ICPC World Final (WF)	World Final (WF)
The 2022 ICPC World Final (WF)	World Final (WF)
The 2023 ICPC World Final (WF)	World Final (WF)
The 2024 ICPC Asia East Continent Final Contest (AECFC)	Continent Final (CF)
The 2024 ICPC North America Championship (NAC)	Continent Final (CF)
The 2024 ICPC Asia Chengdu Regional Contest (ACRC)	Regional
The 2024 ICPC Asia Hangzhou Regional Contest (AHRC)	Regional
The 2024 ICPC Asia Hong Kong Regional Contest (AHKRC)	Regional
The 2024 ICPC Asia Nanjing Regional Contest (ANRC)	Regional
The 2024 ICPC Asia Shanghai Regional Contest (ASRC)	Regional
The 2024 ICPC Asia Shenyang Regional Contest (ASRC)	Regional
The 2024 ICPC Northwestern Europe Regional Contest (NERC)	Regional
The 2024 ICPC Central Europe Regional Contest (CERC)	Regional