

# ICE-CODER: INTEGRATING WHITE-BOX AND BLACK-BOX TESTING IN EXECUTION-GUIDED MULTI-AGENT CODE GENERATION

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

LLM-based coding agents are programs that utilise LLMs to automate code generation tasks. Typically, they incorporate code execution capabilities which, together with automated test generation and/or debugging methods, enhance the reliability of the generated code. However, the effectiveness of these approaches remains limited in complex problems (such as competitive programming problems) where bugs surface only in convoluted edge cases. This work builds upon multi-agent code generation techniques which emulate software engineering environments. In particular, to address obscure edge cases, we take inspiration from code coverage tools and code reviews to generate white-box tests, on top of existing black-box test generation approaches. Test case outputs are validated through a process of deliberation using the LLM. By increasing the quantity and quality of the test cases, we obtain more reliable generated code. We evaluated ICE-Coder on LiveCodeBench-Hard. Out of the 90 problems, it solves 72, compared to the baseline of 55.

## 1 INTRODUCTION

Code generation has emerged as a critical application of large language models (LLMs) (Zan et al., 2023; Jiang et al., 2024; Huynh & Lin, 2025). Recent work underscores this: general-purpose LLMs are often assessed on code generation benchmarks such as HumanEval (Chen et al., 2021a) and LiveCodeBench (Jain et al., 2025), while prompt engineering techniques such as Chain-of-Thought (Wei et al., 2022), Reflexion (Shinn et al., 2023) and Language Agent Tree Search (Zhou et al., 2024) demonstrate their effectiveness in enhancing LLM-based code generation.

Building on these ideas, *LLM-based coding agents* (Hong et al., 2024; Huang et al., 2024b; Islam et al., 2024; Zhong et al., 2024; Dong et al., 2024; Qian et al., 2024; Wang et al., 2024; Shi et al., 2024) have been developed to specifically target the task of code generation. These typically improve upon existing prompt engineering strategies by combining them with testing and debugging techniques.

The success of these approaches demonstrates the crucial role that *testing* plays in automated code generation.

More broadly, two key paradigms have arisen in LLM-based coding agents — (1) to better mirror the way humans behave when they code (Zhong et al., 2024; Huang et al., 2024a; Wang et al., 2024; Shi et al., 2024), and (2) to emulate a software engineering team using multiple agents (Huang et al., 2024b; Islam et al., 2024; Dong et al., 2024; Qian et al., 2024).

While many multi-agent frameworks simulate test generation agents, they do not fully simulate the extensive testing done in mature software engineering

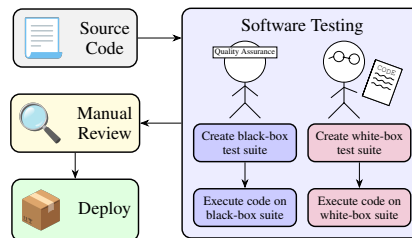


Figure 1: A typical software testing pipeline. The code is tested on black-box and white-box (coverage-based) test cases, after which it must pass a manual check by a reviewer before being deployed.

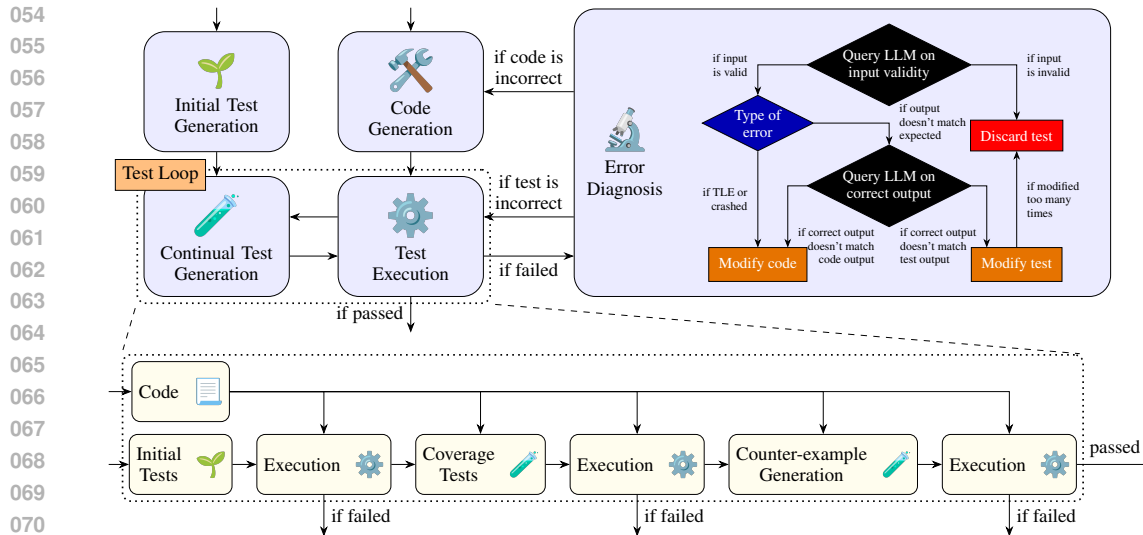


Figure 2: ICE-Coder’s pipeline. ICE-Coder cycles between code generation, testing, and error diagnosis. The testing loop is inspired by the software testing pipeline shown in Figure 1. The initial tests correspond to the black-box test cases written by test engineers; the coverage tests emulate the white-box test cases; the counter-example generation mirrors the code review. The code is executed on each of these tests, and tests that fail are passed to the error diagnosis component. Here, by querying the LLM, we decide if the code should be modified, the test case should be modified, or the test case should be discarded. See Figure 4 in Appendix C for a simplified end-to-end execution example.

teams. For example, on top of the black-box tests written by test engineers, developers are often required to write their own *white-box tests*, which aim to maximise *code coverage*. Large companies like Google have sophisticated infrastructures (Ivankovic et al., 2019) to track, manage, and enforce code coverage. Additionally, code must typically undergo *code review* before being accepted (Sadowski et al., 2018), where other developers examine the code to look for bugs and other issues.

These white-box testing steps are crucial for effective bug detection. For particularly challenging problems, the absence of insight into the approach prevents black-box tests from identifying the often subtle or unexpected edge cases it involves, making these white-box tests all the more important. Therefore, we propose simulating coverage-based testing and code reviews in our coding agent.

Another challenge that existing coding agents face is in validating the correctness of generated test case outputs. The expected outputs generated together with test cases are frequently incorrect (Chen et al., 2023; Huang et al., 2024b). Existing coding agents that generate test cases either rely on the (potentially incorrect) expected outputs generated (Huang et al., 2024b), or do consensus voting on the output among the generated codes (Chen et al., 2023). The former approach introduces false positives and the latter approach potentially introduces false negatives for commonly overlooked bugs. For challenging problems in particular, many candidate solutions may fail to solve the same (important) edge case. To address these, we instead propose a *prompt-based approach* to validating test case correctness, similar to how software engineers will deliberate on the correct outputs when the code output differs from the test case output.

In this work, we introduce ICE-Coder, which:

1. combines the two approaches of emulating the software engineering environment and simulating how individual team members think,
2. adds coverage-based testing and “code review” simulations on top of existing black-box testing, and
3. resolves conflicting test case outputs and code outputs through deliberation similar to a software engineering team.

Our approach provides better confidence in the generated code by increasing both the *quantity* and *quality* of test cases.

We evaluated ICE-Coder on LiveCodeBench-Hard on nine LLM models, which differ in size and reasoning ability. ICE-Coder improved the pass@1 rate for all models except GPT 4.1-nano, with the greatest improvement in DeepSeek V3 (59.26% from 27/90 to 43/90) and Grok 3-mini (high) (52.8% increase from 36/90 to 55/90). The best overall solve rate of 72/90 was obtained with GPT o4-mini (high), for which the base model solved just 55/90 problems (thus, we obtained a 30.9% increase). We further investigated the poor performance of GPT 4.1-nano, and posit that it is primarily due to the large initial test generation. Additionally, we compared ICE-Coder with other agentic code generation frameworks, and show that it achieves the state-of-the-art results on LiveCodeBench-Hard.

## 2 RELATED WORK

**LLM-based Coding Agents** Code generation has emerged both as an important application of LLMs, and as a useful benchmark for the development of LLMs. Apart from the direct training techniques used on the LLMs themselves, several prompt engineering and/or multi-agent frameworks have been developed to enhance the effectiveness of pre-trained LLMs at code generation tasks by incorporating code generation. At their core, these methods rely on iteratively refining the code based on feedback, as is the central thesis of Madaan et al. (2023). Later methods additionally integrate testing or debugging capabilities. Broadly speaking, these can be divided into two categories — those that use multi-agent collaboration to simulate roles in a software engineering team (Hong et al., 2024; Huang et al., 2024b; Islam et al., 2024; Dong et al., 2024; Qian et al., 2024), and those that use prompt engineering techniques on a single LLM agent to have it reason about the problem like a developer (Zhong et al., 2024; Huang et al., 2024a; Wang et al., 2024; Shi et al., 2024).

In Huang et al. (2024b), a separate code generation and test generation agent are used. They also include a test executor agent, though it does not make queries to an LLM. The test executor agent is tasked with generating basic, edge, and large test cases. Generated codes are run on the generated cases, and iteratively refined until they pass all test cases. Dong et al. (2024) uses an analyst, coder, and tester, where the analyst breaks the task down into simpler subtasks for the coder. Similarly, Islam et al. (2024) uses 4 agents, Hong et al. (2024) uses 5 agents and Qian et al. (2024) uses 7 agents, each incorporating finer-grained planning steps. A crucial component of all of these methods is test generation — each of them has a dedicated test generation agent that queries an LLM for test cases.

However, as was studied by Chen et al. (2023); Huang et al. (2024b), the test cases generated by LLMs are unreliable. Chen et al. (2023) proposes a kind of consensus approach, where test cases are deemed accurate when several generated codes yield its expected output. Conversely, Huang et al. (2024b) simply relies on the independence between test and code generation to produce accurate tests, settling for an 89.6% test accuracy rate on HumanEval when using GPT-4. However, such approaches are ineffective for challenging tasks such as competitive programming problems. In these problems, (1) deriving the answer based on the inputs is generally difficult, potentially resulting in the initially generated expected output being incorrect, and (2) there may be important edge cases that many candidate solutions get incorrect. Therefore, we propose (1) using a Chain-of-Thought approach to solicit correct outputs from the LLMs whenever there are disagreements between the code output and expected output, and (2) combining the black-box test generation methods used by AgentCoder with white-box and coverage-based testing methods to derive potentially useful edge cases.

In Zhong et al. (2024), the authors identify a fixed number of code blocks to track the execution state of the code before and after. Shi et al. (2024) introduces a hierarchical “sub-function  $\rightarrow$  file  $\rightarrow$  project” structure for tracing. The traces are then used to identify buggy code blocks for iterative refinement. However, these approaches assume that the high-level structure of the existing code is correct, and that only minor edits to certain code blocks are needed to fix the bugs. We found that this assumption may not be true for more challenging problems. Instead, inspired by their methods, we took a less structured approach to generating execution traces and debugging, allowing the LLMs more freedom to identify the type of bug in the code.

**LLM-based Test Generation** Besides their applications in collaborative coding agents, the use of LLMs to test existing codebases is itself a subject of independent interest. Early work (Yuan et al., 2024; Chen et al., 2024) simply prompted LLMs to emit tests and then measured how well they compiled and executed. Later work combines LLMs with traditional test generation methods that aim to maximise *code coverage* — the set of lines, or combinations of lines, which were executed. Popular measures of code coverage are line coverage, statement coverage, branch coverage, and path coverage, which differ in level of granularity. Early work relied on constraint solvers to resolve branch conditions and generate tests; this approach is known as *symbolic testing* (King, 1976; Howden, 1977). This was later combined with concrete execution to give *concolic execution* (Godefroid et al., 2005; Sen & Agha, 2006; Cadar et al., 2008; Sapra et al., 2013). Coverage-guided test generation is also used as the fitness measure with evolutionary/genetic algorithms to give mutation-/generational test generation tools (Zalewski, 2010; Fraser & Arcuri, 2011; Liu & Yu, 2022).

Yet, symbolic constraint-solving remains challenging, especially in dynamically-typed general-purpose languages like Python, due to the large variable space and frequent use of external libraries (Sapra et al., 2013; Chen et al., 2021b).

Recently, several approaches attempt to use LLMs to solve for inputs that maximise code coverage (Pizzorno & Berger, 2025; Gu et al., 2025; Pan et al., 2025; Ryan et al., 2024). For example, Pizzorno & Berger (2025) first breaks the code under test down into code segments, and derives a set of “interesting” lines — one from each segment. In each prompt, it lists the still-uncovered “interesting” lines based on the current test suite, and iteratively queries the LLM for new tests to add to the test suite. Pan et al. (2025) generates an initial test suite by prompting the LLM, fixes compilation errors in the generated tests, then expands the test suite by prompting the LLM to generate tests for statements that are not already covered. A survey of 161 professional software engineers found a preference for Aster’s test cases of those generated by other tools, with 88% of responses stating that the tests generated for Python could be added to a regression suite with no or only minor changes.

Given the role that test generation plays in multi-agent code generation tools, we believe that incorporating such coverage-guided test generation techniques in coding agents is a promising approach to boosting their effectiveness.

### 3 METHODS

Figure 2 illustrates the overall pipeline, consisting of code generation, initial test generation, a test loop, and error diagnosis. The problem is first presented to the code generator and initial test generator respectively. Then, the generated code is executed in the test loop, which also generates additional white-box tests on top of the initial black-box ones. If the output of the code does not match the expected output of a given test, we perform “error diagnosis” to determine if the code or the test case has to be changed, or if the test case is malformed. Then, we return to the test loop after making appropriate changes.

#### 3.1 CODE GENERATION

**Initial Code Generation** To maximise the chances of generating correct code, we use Chain-of-Thought (CoT) prompting, specifically directing the LLM to focus on correctness and efficiency. The LLM is asked to (1) understand the task, (2) relate the task to similar problems, (3) analyse various methods to solve the problem, (4) analyse the time complexity, (5) choose the most efficient, correct method, (6) write the solution in pseudocode, and (7) generate annotated Python code. The detailed prompt templates are provided in Appendix H.

**Code Refinement** When modifying incorrect code, we first record the execution trace (local variable changes for each line of execution) of the failed code on the test case, which provides the LLM with more information on the execution. To keep within the token limit, we record a shortened execution trace by only printing the environment the first 1-2 times each line is executed. Since many of these lines are executed consecutively, we simply log the change in variable values between consecutive lines, rather than the entire context. Then, we pass this execution trace, along with the failed code and test case, to a separate LLM to generate a reflection on why the code is wrong. We find that this reflection step prevents non-reasoning LLMs from creating a single if condition to deal

with the specific test case, instead encouraging the LLM to generalise the problem. Thereafter, we pass the old code and the reflection to the LLM to generate a new piece of code, using a similar CoT prompt. In the case that the LLM never finds a correct solution, we limited the number of code regenerations to eight to prevent ICE-Coder from running indefinitely.

### 3.2 INITIAL TEST GENERATION

The **initial test set** is generated independently from the code. This makes it impervious to spillover biases from the code. They are an effective first line of defence, and are especially useful in flagging out codes that misinterpret the problem, or that completely overlook a scenario. It is also at this stage that we generate large-scale test cases to stress-test the code. The initial test set is obtained in a similar way as AgentCoder (Huang et al., 2024b) — by prompting the LLM to generate basic, edge and large-scale test cases. Unlike AgentCoder, we dedicate one prompt for each type of test case, thus yielding a larger initial test set. For edge cases, we first prompt the LLM to generate a list of edge scenarios for the problem, then prompt the LLM for test cases for each edge scenario. Especially for large and/or reasoning models, we found that increasing the number of test cases in this way is effective in ensuring that each functional scenario is tested and catching obscure edge cases. See Algorithm 2 in Appendix B for the details.

The LLMs were allowed to generate input/output pairs either as strings or as Python expressions. If Python expressions are generated, they are evaluated and piped as inputs to the generated code during test execution. Allowing the generation of Python expressions (1) helps to save tokens when generating large-scale test cases, and (2) avoids the frequent counting errors made by LLMs when generating test inputs.

### 3.3 TEST LOOP

---

#### Algorithm 1 Test Loop

---

**Inputs:** Problem statement  $P$ , Code  $\sigma$ , Test Suite  $\psi$   
**Outputs:** Counter-example  $\tau'$ , Updated Test Suite  $\psi'$

- 1:  $\text{statementsLeft} \leftarrow \text{GETEXECUTABLESTATEMENTS}(\sigma)$   
    Existing tests (*initial test set + any other tests generated previously*)
- 2: **for**  $\tau \in \psi$  **do**
- 3:    $\text{testResult}, \text{statementsRun} \leftarrow \text{execute}(\sigma(\tau.\text{inputs}))$
- 4:   **if**  $\text{statementsRun} \setminus \text{statementsLeft} \neq \emptyset$  **then**  $\psi.\text{moveToFront}(\tau)$  **end if**
- 5:    $\text{statementsLeft} \leftarrow \text{statementsLeft} \setminus \text{statementsRun}$
- 6:   **if**  $\text{testResult}$  is ERROR  $\vee \text{testResult} \neq \tau.\text{expected}$  **then**
- 7:      $\psi.\text{moveToFront}(\tau)$
- 8:     **return**  $\tau, \psi$
- 9:   **end if**
- 10: **end for**
- 11: Coverage-guided testing
- 12: **while**  $\text{statementsLeft} \neq \emptyset$  **do**
- 13:    $\rho \leftarrow \text{statementsLeft}.\text{getOneElement}()$
- 14:    $\tau \leftarrow \text{PROMPTLLM}(\text{format}(\text{generateTestForLine}, P, \sigma, \rho.\text{lineNum}))$
- 15:    $\psi.\text{addToFront}(\tau)$
- 16:    $\text{testResult}, \text{statementsRun} \leftarrow \text{execute}(\sigma(\tau.\text{inputs}))$
- 17:    $\text{statementsLeft} \leftarrow \text{statementsLeft} \setminus \text{statementsRun}$
- 18:   **if**  $\text{testResult}$  is ERROR  $\vee \text{testResult} \neq \tau.\text{expectedOutput}$  **then return**  $\tau, \psi$  **end if**
- 19: **end while**
- 20: Counter-example generation
- 21:  $\tau \leftarrow \text{PROMPTLLM}(\text{format}(\text{generateCounterexample}, P, \sigma))$
- 22:  $\psi.\text{addToFront}(\tau)$
- 23:  $\text{testResult}, \text{statementsRun} \leftarrow \text{execute}(\sigma(\tau.\text{inputs}))$
- 24: **if**  $\text{testResult}$  is ERROR  $\vee \text{testResult} \neq \tau.\text{expected}$  **then return**  $\tau, \psi$  **end if**
- 25: **return** NONE,  $\psi$

---

270 Three types of tests are run in the test loop — (1) the initial black-box tests, (2) coverage-guided  
271 white-box tests, and (3) direct queries for counter-examples to the code. The test suite is executed in  
272 order. Whenever a test case covers a new line of code or discovers a bug, it is moved to the front of  
273 this linked list. Thus, test cases most likely to discover bugs are prioritised. Newly generated tests  
274 are added to the front of the linked list. See Algorithm 1 for the full details on test execution.

275 The test executor relays a counter-example input (if found) to the error diagnosis component. If the  
276 test case is found to be incorrect or invalid, the test executor can continue from where it left off.  
277 Besides that, even after finding a valid counter-example to the code, the main logic will continue  
278 running the test executor if not all statements have been covered. However, any subsequent counter-  
279 examples returned by the test executor will not be deliberated on.

280 **Coverage-guided tests** When executing the code, we aim for complete statement coverage. This  
281 ensures that each statement behaves as expected. For example, statements involving floating point  
282 operations may unexpectedly generate spurious digits in an otherwise logically-sound code. This  
283 might only be discovered by executing that specific statement. Additionally, keeping and re-using  
284 coverage-guided test cases helps us avoid regression when refining codes by ensuring that previously  
285 accounted-for cases are still handled correctly. Thus, for statements not covered after executing the  
286 existing test suite, we prompt the LLM with the problem, the code, and each of their line numbers,  
287 and ask the LLM to generate tests that target each of those specific lines. See Appendix E for an  
288 example of how a coverage-guided test case was used to expose a bug.

289 **Counter-example generation** Finally, if it had passed all prior tests, we directly query the LLM to  
290 generate a single counter-example to the code as a final check. Similar to a code review, this directly  
291 checks if there is anything the code might have missed. See Appendix F for an example of how a  
292 coverage-guided test case was used to expose a bug. To prime the LLM to find a counter-example,  
293 we tell it that there is a bug in the code and ask it to find a test case that exposes the bug. See  
294 Appendix H for the prompt templates used.

### 295 3.4 ERROR DIAGNOSIS

296 We perform **error diagnosis** whenever either (1) the code output does not match the expected output  
297 of the test case, (2) the code crashes on the test case, or (3) the code exceeds the time limit on a test  
298 case. In case 1, we query a separate LLM on which output is correct.

299 In all cases, we first verify if the input formatting is correct by querying the LLM. Generated tests  
300 sometimes have counting errors with parameters (e.g. the test claims to input a 20-character string,  
301 but only produces a 19-character string). Sometimes, generated tests do not fit within the parameter  
302 constraints for the size of input variables (e.g. sets an input to 0 when the problem states that it  
303 cannot be 0), or do not correspond to a valid instance of the problem, thereby having no solution  
304 (e.g. produces a general graph for a problem where the input should be a tree). This first check seeks  
305 to address these problems. If the inputs are invalid, the test case is discarded entirely.  
306

307 Otherwise, if the time limit was exceeded (TLE) or it crashed, we return the code to the code gener-  
308 ator for modification. If the code output does not match the expected output, we additionally query  
309 the LLM on what the correct output should be, we then modify the test case expected outputs and/or  
310 the code based on the result of this query. (Due to the difficulty of reasoning about large test cases,  
311 we do not check the code outputs against the expected outputs for large test cases. In other words,  
312 large test cases are mainly used to test the efficiency of the code.)  
313

314 Finally, if the expected output of a test case had been modified more than once, this suggests that the  
315 solution to the problem instance is not obvious to the LLM. In such instances, we simply discard the  
316 test case. Also, to balance token usage with accuracy, we limit the number of times error diagnosis  
317 is conducted to 16. Upon crossing this limit, we simply return the code as is.  
318

## 319 4 EVALUATION

### 320 4.1 EXPERIMENTAL SETUP

321 We evaluated ICE-Coder against the latest version (August 2024) of LiveCodeBench-Hard (Jain  
322 et al., 2025), which is a popular dataset for competitive programming problems that includes prob-  
323

lems from CodeForces, LeetCode, and AtCoder. We chose LiveCodeBench for evaluation as it draws problems from a wide variety of sources. As current models are already able to solve most of the easier problems, we specifically targeted the hard section of this competitive programming dataset. We used the pass@1 metric for evaluation. The generated codes were run on our server (Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz) with a time limit of 6 seconds<sup>1</sup> for each test case.

First, ICE-Coder was run on several different popular LLMs to test its LLM-agnosticism. The LLMs — Gemini-2.5 Pro, DeepSeek V3, DeepSeek R1, Grok 3-mini, GPT 4.1-nano, GPT 4.1-mini, and GPT o4-mini — were chosen to (1) reflect a variety of different providers, (2) test out different types of models (e.g. reasoning/non-reasoning, mini/larger), and (3) for good baseline performance on the LiveCodeBench leaderboard. The exact checkpoints of the models used are detailed in Appendix A for reproduction. For the baseline, we used the initial code generation prompt template (provided in Appendix H) to generate the codes. Since we automated the extraction of codes, we allowed the LLMs to attempt at most 3 attempts to output codes in that format (the first attempt conforming to the format was taken to be the submission).

Next, we compared ICE-Coder against other LLM-based agentic code generation tools — AgentCoder (Huang et al., 2024b), MapCoder (Islam et al., 2024), LDB (Zhong et al., 2024), and LPW (Lei et al., 2025). These represent a variety of different methods. AgentCoder was picked as the prompts used in ICE-Coder were adapted from it. Conversely, LDB showcases a debugging-based method to improve code generation. Additionally, LDB and LPW were picked for good performance on HumanEval (Chen et al., 2021a), and MapCoder was picked for good performance on MBPP<sup>2</sup> (Austin et al., 2021). Unfortunately, we could not find other open-source agentic tools that target the LiveCodeBench dataset. For tools that generate test cases, we additionally measure the code coverage of the generated test cases on the generated codes. Unfortunately, since LiveCodeBench does not provide canonical solutions, we cannot measure test case accuracy. The tests were conducted with GPT o4-mini (high) as the base model, since it is the best-performing base model on the LiveCodeBench leaderboard at the time of writing (though notably, when we performed our own benchmarking, Gemini-2.5 Pro and DeepSeek R1 performed better).

## 4.2 RESULTS

As shown in Table 1, ICE-Coder improves the performance of all the models tested on LiveCodeBench-Hard, apart from GPT 4.1-nano (see Appendix D for analysis). The best model on which it performs is GPT o4-mini (high), and the steepest improvement is seen with DeepSeek V3.

As shown in Table 2, ICE-Coder achieves state-of-the-art results (80.00% or 72 out of 90) when used on GPT o4-mini (high). Additionally, it achieves the highest average code coverage of 97.82%.

Table 1: pass@1 Rates of ICE-Coder on LiveCodeBench-Hard

Model	Baseline	ICE-Coder	Improvement
Gemini-2.5 Pro	65.56%	74.44%	13.56%
DeepSeek V3	30.00%	47.78%	59.26%
DeepSeek R1	62.22%	74.44%	19.64%
Grok 3-mini	36.67%	45.56%	24.24%
Grok 3-mini (high)	40.00%	61.11%	52.78%
GPT 4.1-nano	7.78%	5.56%	-28.57%
GPT 4.1-mini	28.89%	38.89%	34.62%
GPT o4-mini	60.00%	77.78%	29.63%
GPT o4-mini (high)	61.11%	80.00%	30.91%

<sup>1</sup>The time limit of 6s is based on the LiveCodeBench implementation on GitHub.

<sup>2</sup>Since we are only testing open-source agentic methods, other methods on the MBPP leaderboard such as EG-CFG (Lavon et al., 2025) and QualityFlow (Hu et al., 2025) were not included in our comparison.

<sup>3</sup>If no syntactically-valid code is produced, the code coverage for that problem is marked as 0%.

<sup>4</sup>We attribute the lower-than-baseline pass@1 rate of AgentCoder to the prompt used — the prompt used for the baseline mentioned competitive programming, thus priming the LLM to generate more efficient code.

<sup>5</sup>LPW is made for functional problems. The codes produced by LPW contain many print statements for debugging. During evaluation, we replaced these print statements with pass statements. Had we not replaced these print statements and merely suppressed them, this pass@1 rate drops to 28.89% due to time limit exceeded errors. As for the results, we posit that the planning workflow proposed by LPW is less effective on a reasoning model like GPT o4-mini.

Table 2: Results on LiveCodeBench-Hard with GPT o4-mini

Framework/Tool	Average Code Coverage <sup>3</sup>	pass@1 rate
Baseline	NA	61.11%
AgentCoder	92.43%	60.00% <sup>4</sup>
MapCoder	NA	71.11%
LDB	92.31%	78.89%
LPW	82.29%	54.44% <sup>5</sup>
ICE-Coder	97.82%	80.00%

### 4.3 ANALYSIS

We divide our analysis based on whether the initial code generated is correct.

If the initial code is already correct, the goal is to avoid regression. Ideally, we want the code to pass all the generated test cases and be returned. While this happens most of the time, incorrectly produced test cases do complicate things for some of the problems. We’ve attempted to use detailed CoT (see Appendix H) to mitigate this in the diagnosis phase. The blanket input validity check was also inspired by regressions during our experimentation, where minor formatting errors (e.g. presence or absence of spaces after commas) in the inputs did not crash the code, but affected the diagnosis results (e.g. both should not have spaces after commas, but since the input incorrectly contained spaces, the diagnosis favoured the output with the spaces). Nevertheless, there were still incorrect test cases that get through the discussion phase, especially in calculation-heavy questions and non-reasoning models. This is partially mitigated by our provision of the execution trace in the reflection stage, where the LLM might realise based on the trace that the code is actually correct, and the expected output is not. Still, there remain issues with LLMs being overly trusting of the expected outputs claimed by the prompt. Another potential solution is to equip the LLMs with a calculator tool, e.g., using a ReAct framework (Yao et al., 2023). We leave this to future work.

See Appendix G for an example of regression with DeepSeek V3. These “bad” test cases are most frequently contributed by the initial test generation, where each LLM response yields multiple test cases at once. Such scenarios also explain the poor performance of ICE-Coder on GPT 4.1-nano (see Appendix D for more).

If the initial code is incorrect, we need to (1) find a test case that demonstrates why the code is incorrect, and (2) correct the code. As demonstrated by our ablation study in Section 4.4, each component of ICE-Coder is essential in this process. In Appendices E and F, we detail examples where coverage-guided tests and counter-example generation were able to find bugs. In general, the large initial black-box test set serves as an effective first line of defence against bugs. However, especially when the solution code is complex, the initial test set is often insufficient in exhausting all branching scenarios. In these cases, coverage-guided tests and counter-example generation supplement and complement the initial test set. Finally, the execution traces help with diagnosing the problem and correcting the code.

### 4.4 ABLATION STUDY

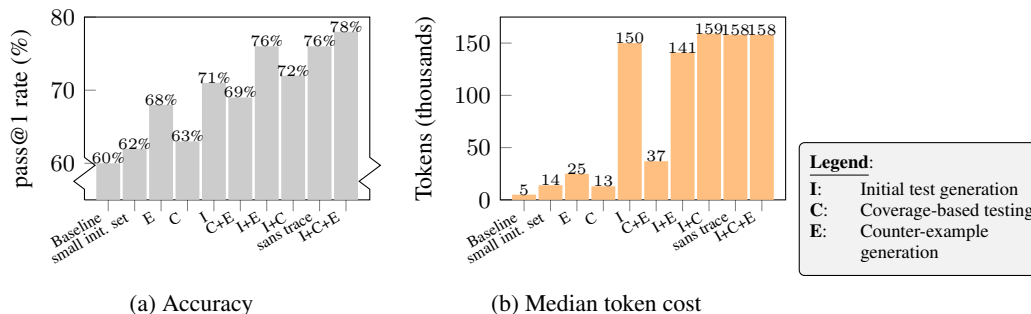


Figure 3: Ablation study with GPT o4-mini

We conducted an ablation study on the three types of tests generated — the initial black-box test generation, the coverage-guided test generation, and the counter-example generation. We tested all 8 combinations of these test generation methods, using GPT o4-mini as the base model. We also report the token usage for each. We also ran two more experiments — (1) with a smaller initial test set using a single prompt to get basic, edge, and large test cases (similar to AgentCoder)<sup>6</sup>, and (2) without producing the execution trace for the reflection step<sup>7</sup>. The results of the ablation study on the testing methods are shown in Figure 3.

As shown in Figure 3a, the pass@1 rate increases with each type of test added. On their own, each type of test is able to achieve significant improvements over the baseline, with the most significant improvement coming from the initial test generation. Yet, at the same time, the initial test generation uses by far the greatest number of tokens. (Note: for some of the tests, adding a component on top of the initial test generation decreased the total token usage; this is likely due to randomness and variance in the initial test generation, which dominates the token usage.)

Conversely, coverage-guided testing and counter-example generation add minimal token overhead to the code generation process whilst improving the pass@1 rates. When compared more fairly to a smaller initial test set (with similar token usage), they achieve better pass@1 rates.

Similarly, the addition of tracking of execution traces adds negligible token overhead whilst improving accuracy rates. In fact, while the execution traces increase the number of input tokens used (from 59K to 67K median), they slightly decrease the expenditure of reasoning tokens (from 84K to 76K median). (The median response tokens are around the same for both — about 15K.)

## 5 THREATS TO VALIDITY

We only evaluated ICE-Coder on LiveCodeBench-Hard, and have not tested it on other datasets. Each experiment was only conducted once on the relatively small 90-problem dataset, which means some variance should be expected of the results. Additionally, the evaluation was conducted on our Intel(R) Xeon(R) Gold 6248R CPU server, which may have a different execution speed from other evaluation platforms.

We set certain constant limits, e.g. number of code regenerations (8), number of error diagnosis steps performed (16), number of test case updates (1), etc. when doing our experiments. We have not spent too much effort analysing how these constants affect the balance between code accuracy and token cost.

For the comparison with other agentic tools, since they were not tailored for LiveCodeBench, some of them only write Python functions. To test these, we had to slightly modify the LiveCodeBench problem descriptions to make them functional, rather than I/O-based<sup>8</sup>. Thus, while we tried our best to keep the modifications minimal, the results could nevertheless be affected by how we modified the problem descriptions.

## 6 CONCLUSION

In this paper, we introduce ICE-Coder, which improves on previous multi-agent code generation frameworks for complex problems by introducing white-box and coverage-guided tests, and by taking a prompt-based approach to figuring out correct generated test outputs. In doing so, it simulates a typical software engineering pipeline, with black-box functional tests, white-box coverage tests, and code reviews. At the same time, it also mirrors how individual developers work, deliberating on the correct test outputs and using the execution trace to find bugs. ICE-Coder achieves state-of-the-art performance on the LiveCodeBench-Hard dataset, with an 80% pass@1 rate when used with GPT o4-mini (high). It also improves the pass@1 rates on a range of other models, including a 59% improvement on DeepSeek V3 and a 53% improvement on Grok-3 mini (high).

<sup>6</sup>This experiment was conducted with the small initial test set as the only source of test cases.

<sup>7</sup>This experiment was conducted with all three types of tests.

<sup>8</sup>We transformed them to `solve(inp: str) -> str` functions, with the problem description as a docstring.

## REFERENCES

- 486  
487  
488 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan,  
489 Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large  
490 language models, 2021. URL <https://arxiv.org/abs/2108.07732>.
- 491 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe:  
492 Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), December 2008.  
493 ISSN 1094-9224. URL <https://doi.org/10.1145/1455518.1455522>.
- 494 Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu  
495 Chen. Codet: Code generation with generated tests. In *The Eleventh International Confer-*  
496 *ence on Learning Representations*, 2023. URL <https://openreview.net/forum?id=ktrw68Cmu9c>.
- 497  
498  
499 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared  
500 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri,  
501 Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan,  
502 Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian,  
503 Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fo-  
504 tios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex  
505 Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders,  
506 Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec  
507 Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob Mc-  
508 Grew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large  
509 language models trained on code, 2021a. URL <https://arxiv.org/abs/2107.03374>.
- 510 Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest:  
511 A framework for llm-based test generation, 2024. URL <https://arxiv.org/abs/2305.04764>.
- 512  
513 Yu-Fang Chen, Wei-Lun Tsai, Wei-Cheng Wu, Di-De Yen, and Fang Yu. Pyct: A python  
514 concolic tester. In *Programming Languages and Systems: 19th Asian Symposium, APLAS*  
515 *2021, Chicago, IL, USA, October 17–18, 2021, Proceedings*, pp. 38–46, Berlin, Heidelberg,  
516 2021b. Springer-Verlag. ISBN 978-3-030-89050-6. URL [https://doi.org/10.1007/978-3-030-89051-3\\_3](https://doi.org/10.1007/978-3-030-89051-3_3).
- 517  
518 Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM*  
519 *Trans. Softw. Eng. Methodol.*, 33(7), September 2024. ISSN 1049-331X. URL <https://doi.org/10.1145/3672459>.
- 520  
521 Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented  
522 software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Confer-*  
523 *ence on Foundations of Software Engineering, ESEC/FSE '11*, pp. 416–419, New York, NY,  
524 USA, 2011. Association for Computing Machinery. ISBN 9781450304436. URL <https://doi.org/10.1145/2025113.2025179>.
- 525  
526 David Gale. A theorem on flows in networks. *Pacific Journal of Mathematics*, 7(2):1073 – 1082,  
527 1957.
- 528  
529 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In  
530 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and*  
531 *Implementation, PLDI '05*, pp. 213–223, New York, NY, USA, 2005. Association for Computing  
532 Machinery. ISBN 1595930566. URL <https://doi.org/10.1145/1065010.1065036>.
- 533  
534 Sijia Gu, Noor Nashid, and Ali Mesbah. Llm test generation via iterative hybrid program analysis,  
535 2025. URL <https://arxiv.org/abs/2503.13580>.
- 536 Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao  
537 Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao,  
538 Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collab-  
539 orative framework. In *The Twelfth International Conference on Learning Representations*, 2024.  
URL <https://openreview.net/forum?id=VtmBAGCN7o>.

- 540 W. E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Trans. Softw.*  
541 *Eng.*, 3(4):266–278, July 1977. ISSN 0098-5589. URL [https://doi.org/10.1109/TSE.](https://doi.org/10.1109/TSE.1977.231144)  
542 1977.231144.
- 543 Yaojie Hu, Qiang Zhou, Qihong Chen, Xiaopeng Li, Linbo Liu, Dejiao Zhang, Amit Kachroo, Talha  
544 Oz, and Omer Tripp. Qualityflow: An agentic workflow for program synthesis controlled by llm  
545 quality checks, 2025. URL <https://arxiv.org/abs/2501.17167>.
- 546 Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. Codecot: Tackling code syntax errors in  
547 cot reasoning for code generation, 2024a. URL <https://arxiv.org/abs/2308.08784>.
- 548 Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder:  
549 Multi-agent-based code generation with iterative testing and optimisation, 2024b. URL <https://arxiv.org/abs/2312.13010>.
- 550 Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey  
551 of challenges, techniques, evaluation, and applications, 2025. URL <https://arxiv.org/abs/2503.01245>.
- 552 Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code  
553 generation for competitive problem solving. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar  
554 (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944, Bangkok, Thailand, August 2024. Association  
555 for Computational Linguistics. URL [https://aclanthology.org/2024.acl-long.](https://aclanthology.org/2024.acl-long.269/)  
556 269/.
- 557 Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. Code coverage at google. In  
558 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 955–963, 2019.
- 559 Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Ar-  
560 mando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamina-  
561 tion free evaluation of large language models for code. In *The Thirteenth International Confer-*  
562 *ence on Learning Representations*, 2025. URL [https://openreview.net/forum?id=](https://openreview.net/forum?id=chfJJYC3iL)  
563 [chfJJYC3iL](https://openreview.net/forum?id=chfJJYC3iL).
- 564 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language  
565 models for code generation, 2024. URL <https://arxiv.org/abs/2406.00515>.
- 566 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July  
567 1976. ISSN 0001-0782. URL <https://doi.org/10.1145/360248.360252>.
- 568 Boaz Lavon, Shahar Katz, and Lior Wolf. Execution guided line-by-line code generation, 2025.  
569 URL <https://arxiv.org/abs/2506.10948>.
- 570 Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. Planning-driven programming:  
571 A large language model programming workflow, 2025. URL [https://arxiv.org/abs/](https://arxiv.org/abs/2411.14503)  
572 [2411.14503](https://arxiv.org/abs/2411.14503).
- 573 Xiangjun Liu and Ping Yu. Randoop-ts: Random-based test generator with test suite reduction. In  
574 *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, Internetware ’22, pp. 221–230,  
575 New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450397803. URL  
576 <https://doi.org/10.1145/3545258.3545280>.
- 577 Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri  
578 Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad  
579 Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine:  
580 iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on*  
581 *Neural Information Processing Systems*, NIPS ’23, Red Hook, NY, USA, 2023. Curran Associates  
582 Inc.
- 583 Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Aster: Natural  
584 and multi-language unit test generation with llms, 2025. URL [https://arxiv.org/abs/](https://arxiv.org/abs/2409.03093)  
585 [2409.03093](https://arxiv.org/abs/2409.03093).

- 594 Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Effective high coverage test generation for  
595 python, 2025. URL <https://arxiv.org/abs/2403.16218>.  
596
- 597 Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen,  
598 Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Commu-  
599 nicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar  
600 (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*  
601 *(Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, August 2024. Association  
602 for Computational Linguistics. URL [https://aclanthology.org/2024.acl-long.  
603 810/](https://aclanthology.org/2024.acl-long.810/).
- 604 Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ra-  
605 manathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test gen-  
606 eration in regression setting using llm. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. URL  
607 <https://doi.org/10.1145/3643769>.  
608
- 609 H. J. Ryser. Combinatorial properties of matrices of zeros and ones. *Canadian Journal of Mathe-*  
610 *matics*, 9:371–377, 1957. doi: 10.4153/CJM-1957-044-3.
- 611 Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern  
612 code review: a case study at google. In *Proceedings of the 40th International Conference on*  
613 *Software Engineering: Software Engineering in Practice*, ICSE-SEIP ’18, pp. 181–190, New  
614 York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356596. URL  
615 <https://doi.org/10.1145/3183519.3183525>.  
616
- 617 Samir Sapra, Marius Minea, Sagar Chaki, Arie Gurfinkel, and Edmund M. Clarke. Finding errors  
618 in python programs using dynamic symbolic execution. In Hüsnü Yenigün, Cemal Yilmaz, and  
619 Andreas Ulrich (eds.), *Testing Software and Systems*, pp. 283–289, Berlin, Heidelberg, 2013.  
620 Springer Berlin Heidelberg. ISBN 978-3-642-41707-8.
- 621 Koushik Sen and Gul Agha. Cute and jcute: concolic unit testing and explicit path model-checking  
622 tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*,  
623 CAV’06, pp. 419–423, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 354037406X. URL  
624 [https://doi.org/10.1007/11817963\\_38](https://doi.org/10.1007/11817963_38).  
625
- 626 Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. From code to correctness:  
627 Closing the last mile of code generation with hierarchical debugging, 2024. URL [https:  
628 //arxiv.org/abs/2410.01215](https://arxiv.org/abs/2410.01215).
- 629 Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Re-  
630 flexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on*  
631 *Neural Information Processing Systems*, 2023. URL [https://openreview.net/forum?  
632 id=vAE1hFckW6](https://openreview.net/forum?id=vAE1hFckW6).  
633
- 634 Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. IN-  
635 TervenOR: Prompting the coding ability of large language models with the interactive chain of  
636 repair. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association*  
637 *for Computational Linguistics: ACL 2024*, pp. 2081–2107, Bangkok, Thailand, August 2024.  
638 Association for Computational Linguistics. URL [https://aclanthology.org/2024.  
639 findings-acl.124/](https://aclanthology.org/2024.findings-acl.124/).
- 640 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi,  
641 Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language  
642 models. In *Proceedings of the 36th International Conference on Neural Information Processing*  
643 *Systems*, NIPS ’22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.  
644
- 645 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan  
646 Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International*  
647 *Conference on Learning Representations*, 2023. URL [https://openreview.net/forum?  
id=WE\\_vluYUL-X](https://openreview.net/forum?id=WE_vluYUL-X).

648 Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou.  
649 Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, 1(FSE), July  
650 2024. URL <https://doi.org/10.1145/3660783>.

651 Michal Zalewski. American fuzzy lop: a security-oriented fuzzer. *URI:*  
652 <http://lcamtuf.coredump.cx/afl>, 2010.

654 Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and  
655 Jian-Guang Lou. Large language models meet NL2Code: A survey. In Anna Rogers, Jordan  
656 Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Associ-*  
657 *ation for Computational Linguistics (Volume 1: Long Papers)*, pp. 7443–7464, Toronto, Canada,  
658 jul 2023. Association for Computational Linguistics. URL [https://aclanthology.org/  
659 2023.acl-long.411/](https://aclanthology.org/2023.acl-long.411/).

660 Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debug-  
661 ger via verifying runtime execution step by step. In Lun-Wei Ku, Andre Martins, and Vivek  
662 Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp.  
663 851–870, Bangkok, Thailand, August 2024. Association for Computational Linguistics. URL  
664 <https://aclanthology.org/2024.findings-acl.49/>.

666 Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Lan-  
667 guage agent tree search unifies reasoning, acting, and planning in language models. In Ruslan  
668 Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and  
669 Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learn-*  
670 *ing*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 21–27  
671 Jul 2024. URL <https://proceedings.mlr.press/v235/zhou24r.html>.

## 672 A MODEL CHECKPOINTS

673 Where applicable, the exact checkpoints of the models used for benchmarking are detailed here for  
674 reproduction.

675 Model	676 Checkpoint
677 Gemini-2.5 Pro	(latest stable release)
678 DeepSeek V3	DeepSeek-V3-0324
679 DeepSeek R1	DeepSeek-R1-0528
680 Grok 3-mini	(latest stable release)
681 GPT 4.1-nano	gpt-4.1-nano-2025-04-14
682 GPT 4.1-mini	gpt-4.1-mini-2025-04-14
683 GPT o4-mini	o4-mini-2025-04-16

## 684 B INITIAL TEST GENERATION ALGORITHM

---

### 685 Algorithm 2 Initial Test Generation

---

686 **Inputs:** Problem statement  $P$   
687 **Outputs:** Test suite  $\psi$

- 688 1: scenarios  $\leftarrow$  PROMPTLLM(format(generateEdgeScenarioPrompt,  $P$ ))
- 689 2: scenarios.add(BasicScenario)
- 690 3: scenarios.add(LargeScenario)
- 691 4: TestMap  $\leftarrow$  new MAP()
- 692 5: **for** scenario  $\in$  scenarios **do**
- 693 6: TestMap[scenario]  $\leftarrow$  PROMPTLLM(format(generateTestPrompt,  $P$ , scenario))
- 694 7: **end for**
- 695 8: TestMap.REMOVEDUPLICATETESTS()
- 696 9:  $\psi \leftarrow$  INTERLEAVE(TestMap)
- 697 10: **return**  $\psi$

---

## C EXAMPLE END-TO-END EXECUTION FLOW

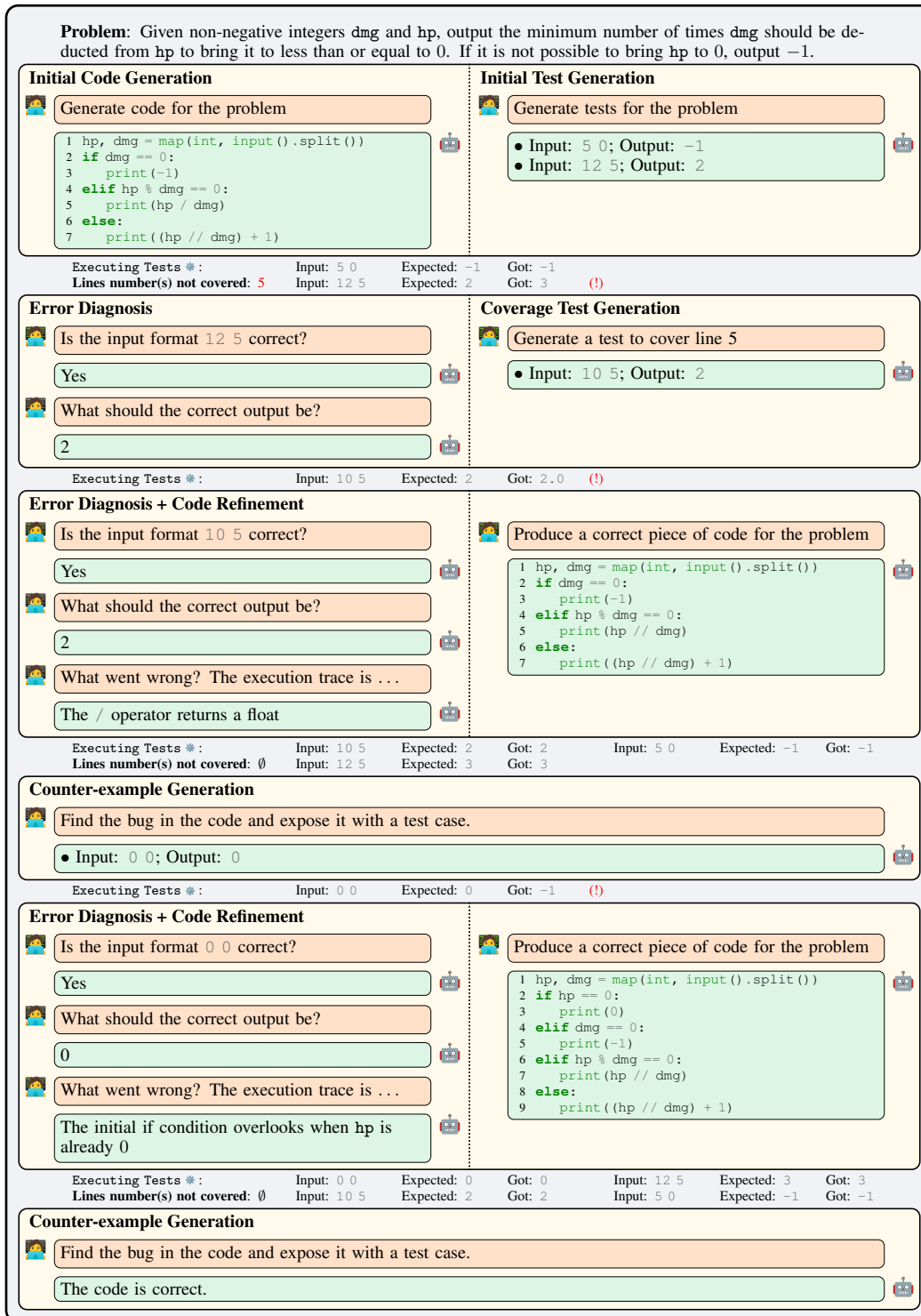


Figure 4: Illustration of the execution flow on an example toy problem. For the actual prompts used, refer to Appendix H.

## D INVESTIGATING GPT 4.1-NANO ANOMALY

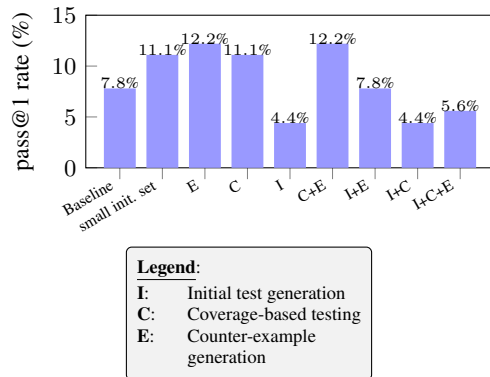


Figure 5: Component Analysis with GPT 4.1-nano

We investigated the anomalous results for GPT 4.1-nano further by running it with each combination of testing components. The results are detailed in Figure 5.

With GPT 4.1-nano, we found that ICE-Coder performs better without the initial black-box test generation. With GPT 4.1-nano, ICE-Coder obtains a 12.2% pass@1 rate without the initial test generation, and only 5.56% when the initial test generation was included. GPT 4.1-nano frequently produced test cases with incorrect expected outputs or formats, which it frequently is unable to identify in the diagnosis/deliberation stage, thus negatively affecting its results. The risk of generating such “bad” test cases is incurred for each test case we generate. Therefore, with GPT 4.1-nano, using a small number of targeted test cases is more effective than using a large initial black-box test

set.

## E COVERAGE-GUIDED TESTING CASE STUDY

Here, we examine how ICE-Coder found a bug in its candidate solution for the “Cans and Openers” problem (question ID abc312\_f of LiveCodeBench taken from AtCoder) due to coverage-guided test generation when using GPT o4-mini.

The “Cans and Openers” problem is reproduced here:

There are  $N$  items.

Each of these is one of a pull-tab can, a regular can, or a can opener.

The  $i$ -th item is described by an integer pair  $(T_i, X_i)$  as follows:

- If  $T_i = 0$ , the  $i$ -th item is a pull-tab can; if you obtain it, you get a happiness of  $X_i$ .

- If  $T_i = 1$ , the  $i$ -th item is a regular can; if you obtain it and use a can opener against it, you get a happiness of  $X_i$ .

- If  $T_i = 2$ , the  $i$ -th item is a can opener; it can be used against at most  $X_i$  cans.

Find the maximum total happiness that you get by obtaining  $M$  items out of  $N$ .

**Input**

The input is given from Standard Input in the following format:

```

N M
T1 X1
T2 X2
⋮
TN XN

```

**Output**

Print the answer as an integer.

810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863

### Constraints

- $1 \leq M \leq N \leq 2 \times 10^5$
- $T_i$  is 0, 1, or 2.
- $1 \leq X_i \leq 10^9$
- All input values are integers.

### Sample Input 1

8 4  
0 6  
0 6  
1 3  
1 5  
1 15  
2 1  
2 10  
2 100

### Sample Output 1

27

If you obtain the 1-st, 2-nd, 5-th, and 7-th items, and use the 7-th item (a can opener) against the 5-th item, you will get a happiness of  $6 + 6 + 15 = 27$ . There are no ways to obtain items to get a happiness of 28 or greater, but you can still get a happiness of 27 by obtaining the 6-th or 8-th items instead of the 7-th in the combination above.

### Sample Input 2

5 5  
1 5  
1 5  
1 5  
1 5  
1 5

### Sample Output 2

0

### Sample Input 3

12 6  
2 2  
0 1  
0 9  
1 3  
1 5  
1 3  
0 4  
2 1  
1 8  
2 1  
0 1

864 0 4

865

866 Sample Output 3

867

868 30

869

870 There is an efficient  $O(n \log n)$  solution to the problem. To understand the solution, consider the  
 871 values  $u_p$  and  $u_r$ , which represent the total satisfaction obtained from the chosen pull-tab cans and  
 872 from the chosen regular cans respectively. We want to maximise both  $u_p$  and  $u_r$ , but there is a  
 873 trade-off between them due to the limited number of items we can pick. A Pareto efficient choice of  
 874 items is one in which we cannot increase either  $u_p$  or  $u_r$  without decreasing the other, and the Pareto  
 875 front is the set of all Pareto efficient choices of items. On one end of the Pareto front, we pick the  $M$   
 876 highest-satisfaction pull-tab cans (and fill in any remaining slots with the best combination of can  
 877 openers and regular cans). On the other end of the Pareto front, we pick the best combination of can  
 878 openers and regular cans (and then fill in any remaining slots with the highest-satisfaction pull-tab  
 879 cans). Note that “the best combination of can openers and regular cans” can be found via a greedy  
 880 algorithm, where we pick the best can opener if we have no “opens” left, and the best regular can if  
 881 we do have “opens” left.

882 The algorithm then traverses the Pareto front, starting on one end, and moving to the other by  
 883 replacing cans one at a time. In each iteration, a greedy (hence efficient) substitution is made,  
 884 and the satisfaction updated and recorded. Then, the maximum satisfaction is output. This can be  
 885 achieved in  $O(n \log n)$  via either prior sorting, or using sorted data structures such as binary heaps  
 886 or binary trees.

887 Initially, GPT o4-mini outputs a buggy solution which attempts a similar approach. It starts with  
 888 a set of only pull-tab cans. Then, it iterates through all possible numbers of can openers. In each  
 889 iteration, it replaces a pull-tab can with a can opener, then greedily swaps pull-tab cans for better  
 890 regular cans, limited to the available “opens”:

```

891 import sys
892 import threading
893 def main():
894     import sys
895     import heapq
896
897     data = sys.stdin.read().split()
898     N, M = map(int, data[:2])
899     idx = 2
900
901     tabs = []
902     cans = []
903     openers = []
904
905     # Read and bucket items by type
906     for _ in range(N):
907         t = int(data[idx]); x = int(data[idx+1])
908         idx += 2
909         if t == 0:
910             tabs.append(x)
911         elif t == 1:
912             cans.append(x)
913         else: # t == 2
914             openers.append(x)
915
916     # We will try k = 0..min(#openers, M)
917     openers.sort(reverse=True)
918
919     # Max-heaps for unused tabs and cans (store negatives)
920     outs_tab = [-x for x in tabs]
```

```

918     heapq.heapify(outs_tab)
919     outs_can = [-x for x in cans]
920     heapq.heapify(outs_can)
921
922     # picks = min-heap of (x, is_can) for the items currently
923     ↪ counted
924     picks = []
925     cur_sum = 0
926     can_in_picks = 0
927
928     cap = 0           # total capacity from chosen openers
929     t = M             # how many non-opener slots we must fill
930
931     # Initial fill with tabs only (capacity=0 => no cans allowed)
932     while len(picks) < t and outs_tab:
933         x = -heapq.heappop(outs_tab)
934         heapq.heappush(picks, (x, False))
935         cur_sum += x
936
937     ans = 0
938     max_k = min(len(openers), M)
939
940     for k in range(max_k + 1):
941         # Record the best sum for choosing exactly k openers
942         if cur_sum > ans:
943             ans = cur_sum
944         if k == max_k:
945             break
946
947         # 1) Add the k-th opener
948         cap += openers[k]
949         # 2) One more slot is taken by that opener
950         t -= 1
951
952         # 3) Remove the smallest item from picks (since t
953         ↪ decreased by 1)
954         if picks:
955             x_min, is_can_min = heapq.heappop(picks)
956             cur_sum -= x_min
957             if is_can_min:
958                 can_in_picks -= 1
959
960         # 4) Refill picks up to size t
961         while len(picks) < t:
962             # best can if we still have capacity for more cans
963             x_can = None
964             if can_in_picks < cap and outs_can:
965                 x_can = -outs_can[0]
966             # best tab
967             x_tab = -outs_tab[0] if outs_tab else None
968
969             # If no candidate at all, break
970             if x_can is None and x_tab is None:
971                 break
972
973             # pick whichever yields higher X
974             if x_can is not None and (x_tab is None or x_can >
975             ↪ x_tab):
976                 # take can

```

```

972         heapq.heappop(outs_can)
973         heapq.heappush(picks, (x_can, True))
974         cur_sum += x_can
975         can_in_picks += 1
976     else:
977         # take tab
978         heapq.heappop(outs_tab)
979         heapq.heappush(picks, (x_tab, False))
980         cur_sum += x_tab
981
982     # 5) Swap loop to improve sum by swapping in better cans
983     ↪ if capacity allows
984     while can_in_picks < cap and outs_can and picks:
985         x_can = -outs_can[0] # best unused can
986         x_min, is_can_min = picks[0] # smallest current pick
987         if x_can <= x_min:
988             break
989
990         # do the swap
991         heapq.heappop(outs_can)
992         heapq.heappop(picks)
993         cur_sum += (x_can - x_min)
994         # the new pick is a can
995         heapq.heappush(picks, (x_can, True))
996
997         # put back the old smallest into its proper outs-heap
998         if is_can_min:
999             # swapped can → can
1000             heapq.heappush(outs_can, -x_min)
1001             # can_in_picks does not change
1002         else:
1003             # swapped tab → can
1004             heapq.heappush(outs_tab, -x_min)
1005             can_in_picks += 1
1006
1007     sys.stdout.write(str(ans) + "\n")
1008
1009 if __name__ == "__main__":
1010     main()

```

1010 However, this strategy fails because the pull-tab can is strictly replaced by the can-opener at the start  
1011 of the iteration, even when there is spare space. Though a while loop is used to add regular cans to  
1012 the set of picks to fill up spare capacity, this does not work when there are few regular cans. This  
1013 is caught by a coverage-guided test case for the `if x_can is None and x_tab is None`  
1014 condition to break the while loop for refilling picks “If no candidate at all”:

```

1015 6 6
1016 0 2
1017 0 3
1018 0 4
1019 1 10
1020 1 9
1021 2 2

```

1023 Here, it is possible to pick all items. Thus, the expected output is 28. However, the code outputs  
1024 26, picking only 5 of the items and leaving out the first pull-tab can.

1025 After several iterations of reflection and refinement, the LLM finds the correct solution:

```

1026 import sys
1027 import threading
1028
1029 def main():
1030     import sys
1031     data = sys.stdin.read().split()
1032     it = iter(data)
1033     N = int(next(it))
1034     M = int(next(it))
1035
1036     tabs = []
1037     cans = []
1038     openers = []
1039
1040     # 1) Read and classify items
1041     for _ in range(N):
1042         t = int(next(it)); x = int(next(it))
1043         if t == 0:
1044             tabs.append(x)
1045         elif t == 1:
1046             cans.append(x)
1047         else:
1048             openers.append(x)
1049
1050     # 2) Sort descending
1051     tabs.sort(reverse=True)
1052     cans.sort(reverse=True)
1053     openers.sort(reverse=True)
1054
1055     # 3) Prefix sums of opener capacities
1056     Kmax = min(len(openers), M)
1057     cap = [0] * (Kmax + 1)
1058     for i in range(1, Kmax + 1):
1059         cap[i] = cap[i-1] + openers[i-1]
1060
1061     import heapq
1062     sel = [] # min-heap of currently selected items
1063     other = [] # max-heap (store negatives) of the rest
1064     sum_sel = 0
1065     K = M # current # of slots for tabs+cans
1066
1067     # DS_insert maintains the top-K items in 'sel' and the rest in
1068     # ↪ 'other'
1069     def DS_insert(x):
1070         nonlocal sum_sel
1071         if len(sel) < K:
1072             # room to push directly
1073             heapq.heappush(sel, x)
1074             sum_sel += x
1075         else:
1076             # if x is better than the smallest in sel, swap it in
1077             if K > 0 and sel and x > sel[0]:
1078                 y = heapq.heappop(sel)
1079                 sum_sel -= y
1080                 # the dropped item goes to 'other'
1081                 heapq.heappush(other, -y)
1082                 # now insert x into sel
1083                 heapq.heappush(sel, x)
1084                 sum_sel += x

```

```

1080         else:
1081             # otherwise x is never selected
1082             heapq.heappush(other, -x)
1083
1084     # 4) Insert all pull-tabs first
1085     for x in tabs:
1086         DS_insert(x)
1087
1088     best = sum_sel
1089     j_inserted = 0
1090
1091     # 5) Sweep over k = number of openers chosen
1092     for k in range(1, Kmax + 1):
1093         # a) unlock and insert newly available cans
1094         j_target = min(cap[k], len(cans))
1095         while j_inserted < j_target:
1096             DS_insert(cans[j_inserted])
1097             j_inserted += 1
1098
1099         # b) pay one slot for the k-th opener
1100         K -= 1
1101         # if we have too many in sel, drop the smallest
1102         if len(sel) > K:
1103             y = heapq.heappop(sel)
1104             sum_sel -= y
1105             # the dropped item is gone for good
1106
1107         # c) update the best answer
1108         if sum_sel > best:
1109             best = sum_sel
1110
1111     # 6) Output result
1112     sys.stdout.write(str(best))
1113
1114 if __name__ == "__main__":
1115     main()

```

## F COUNTER-EXAMPLE GENERATION CASE STUDY

Here, we examine how ICE-Coder found a bug in its candidate solution for the “maximum-number-of-groups-with-increasing-length” problem (question ID 2919 of LiveCodeBench taken from LeetCode) during the counter-example generation phase when using GPT o4-mini.

The “maximum-number-of-groups-with-increasing-length” problem is reproduced here:

You are given a 0-indexed array `usageLimits` of length  $n$ .  
Your task is to create groups using numbers from 0 to  $n - 1$ , ensuring that each number,  $i$ , is used no more than `usageLimits[i]` times in total across all groups.  
You must also satisfy the following conditions:

Each group must consist of distinct numbers, meaning that no duplicate numbers are allowed within a single group.  
Each group (except the first one) must have a length strictly greater than the previous group.

Return an integer denoting the maximum number of groups you can create while satisfying these conditions.

1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187

Example 1:

Input: `usageLimits = [1, 2, 5]`

Output: 3

Explanation: In this example, we can use 0 at most once, 1 at most twice, and 2 at most five times.

One way of creating the maximum number of groups while satisfying the conditions is:

Group 1 contains the number [2].

Group 2 contains the numbers [1, 2].

Group 3 contains the numbers [0, 1, 2].

It can be shown that the maximum number of groups is 3.

So, the output is 3.

Example 2:

Input: `usageLimits = [2, 1, 2]`

Output: 2

Explanation: In this example, we can use 0 at most twice, 1 at most once, and 2 at most twice.

One way of creating the maximum number of groups while satisfying the conditions is:

Group 1 contains the number [0].

Group 2 contains the numbers [1, 2].

It can be shown that the maximum number of groups is 2.

So, the output is 2.

Example 3:

Input: `usageLimits = [1, 1]`

Output: 1

Explanation: In this example, we can use both 0 and 1 at most once.

One way of creating the maximum number of groups while satisfying the conditions is:

Group 1 contains the number [0].

It can be shown that the maximum number of groups is 1.

So, the output is 1.

Constraints:

$$1 \leq \text{usageLimits.length} \leq 10^5$$

$$1 \leq \text{usageLimits}[i] \leq 10^9$$

For easier analysis, we let  $n = \text{usageLimits.length}$  and  $m = \max_i \text{usageLimits}[i]$ . ( $m$  is not used in our analysis, but for naive greedy assignment algorithms,  $m$  is an important factor.)

It turns out that there exists an efficient  $O(n \log n)$  algorithm for this problem, assuming standard arithmetic operations can be performed in  $O(1)$  time (i.e. RAM model). The algorithm involves binary searching for the correct output, performing a feasibility check at each iteration  $k$  of the binary search.

The feasibility check involves checking the Gale-Ryser condition (Gale, 1957; Ryser, 1957). In the  $t$ -th iteration, we perform a weaker check to determine whether it is possible to assign indices to the  $t$  largest groups (i.e. groups  $k, k-1, \dots, k-t+2, k-t+1$ ). This is done by checking the *demand*, or the number of elements in the  $t$  largest groups (i.e.  $\sum_{i=k-t+1}^k i = (k - (k-t+1) + 1)(k + (k-t+1))/2 = t(2k-t+1)/2$ ), against the *supply*, or the number of elements which can fit in  $t$  groups if each group must have distinct elements (i.e.  $\sum_{i=1}^n \min(\text{usageLimits}[i], t)$ ). The correctness of this feasibility check is given by the Gale-Ryser theorem (Gale, 1957; Ryser, 1957).

1188 Notice that where  $i^* = \min_{i:\text{usageLimit}[i]>t}$ ,  $\text{supply}_t = \sum_{i=1}^n \min(\text{usageLimits}[i], t) =$   
 1189  $\sum_{i=1}^{i^*-1} \text{usageLimits}[i] + \sum_{i=i^*}^n t = \sum_{i=1}^{i^*-1} \text{usageLimits}[i] + t(n - i^* + 1)$ .  $i^*$  can be found  
 1190 via binary search, so by pre-computing  $\sum_{i=1}^{i^*-1} \text{usageLimits}[i]$ ,  $\text{supply}_t$  can be computed in  $\log n$   
 1191 time. Thereafter, by pre-computing  $\text{supply}_t$  for  $t \in [1, n]$ , the feasibility check can be done in  
 1192  $O(n)$  time, yielding an  $O(n \log n)$  algorithm. (As it turns out, an  $\tilde{O}(n)$  algorithm is sufficient to  
 1193 pass the time constraints for this problem, and so pre-computing  $\text{supply}_t$  was not necessary to pass  
 1194 this problem.)  
 1195

1196 Initially, GPT o4-mini gets the binary search algorithm, but performs the feasibility check incor-  
 1197 rectly. Instead of checking if  $\forall t \in [1, k], \text{demand}_{k,t} \leq \text{supply}_t$ , it simply checks if  $\text{demand}_{k,k} \leq$   
 1198  $\text{supply}_k$ :

```
1199 import sys
1200
1201 def main():
1202     data = sys.stdin.read().strip()
1203     # Input is given as a Python-style list, e.g. "[1, 2, 5]"
1204     usageLimits = eval(data)
1205     n = len(usageLimits)
1206
1207     # Feasibility check: can we form k groups of sizes 1,2,...,k ?
1208     def canMake(k):
1209         # Sum of min(usageLimits[i], k) across all i
1210         s = 0
1211         # Early exit if sum already big enough
1212         target = k * (k + 1) // 2
1213         for x in usageLimits:
1214             s += x if x < k else k
1215             if s >= target:
1216                 return True
1217         return s >= target
1218
1219     # Binary-search k in [0..n]
1220     low, high = 0, n
1221     while low < high:
1222         mid = (low + high + 1) // 2
1223         if canMake(mid):
1224             low = mid
1225         else:
1226             high = mid - 1
1227
1228     sys.stdout.write(str(low))
1229
1230 if __name__ == "__main__":
1231     main()
```

1231 Surprisingly, this algorithm seems to work in most cases, even for the initially generated edge cases.  
 1232 For example, it returns the correct answer if `usageLimits` is an array consisting of repetitions of  
 1233 the same number (e.g. `[1, 1, 1, ..., 1, 1]`), or if it consists of only consecutive numbers  
 1234 (e.g. `[1, 2, 3, ..., 19, 20]`). As a result, the initial test set and coverage-guided tests fail  
 1235 to find bugs.

1236 However, upon being asked to identify a counter-example, the LLM returns the test case  
 1237 `usageLimits = [4, 4, 1, 1]`. In the above code, `canMake(4)` returns `True` since  
 1238 `demand4,4 = supply4 = 10`. However, we can easily notice that group 3 must contain at least  
 1239 one of the last 2 indices, whilst group 4 must contain both of the last 2 indices. Thus, one  
 1240 of the last two indices must be used twice, which is not possible. Indeed, we can verify that  
 1241 `demand4,3 = 9 > 8 = supply3`.

After reflecting on the counter-example, GPT o4-mini comes up with a correct solution:

```

1242 import sys
1243 import threading
1244 def main():
1245     import sys, bisect
1246     data = sys.stdin.read().strip()
1247     usage = eval(data)
1248     n = len(usage)
1249     usage.sort()
1250     # build prefix sums
1251     prefix = [0]*(n+1)
1252     for i,u in enumerate(usage):
1253         prefix[i+1] = prefix[i] + u
1254
1255     # check whether we can build k increasing groups 1,2,...,k
1256     def feasible(k):
1257         # for each t from 1..k ensure total_supply(t) >= demand(t)
1258         # total_supply(t) = sum_i min(usage[i], t)
1259         # demand(t) = sum of the largest t group-sizes = t*(2*k -
1260             ↪ t + 1)//2
1261         for t in range(1, k+1):
1262             # how many usage[i] <= t
1263             idx = bisect.bisect_right(usage, t)
1264             supply = prefix[idx] + (n-idx)*t
1265             demand = t*(2*k - t + 1)//2
1266             if supply < demand:
1267                 return False
1268         return True
1269
1270     # binary search on k in [0..n]
1271     lo, hi = 0, n
1272     while lo < hi:
1273         mid = (lo + hi + 1)//2
1274         if feasible(mid):
1275             lo = mid
1276         else:
1277             hi = mid - 1
1278
1279     # since groups start at size 1, answer = lo
1280     sys.stdout.write(str(lo))
1281
1282 if __name__ == "__main__":
1283     main()

```

## 1282 G REGRESSION CASE STUDY

1284 Here, we examine how ICE-Coder resulted in a regression for the “Sandwiches” problem (question  
1285 ID abc318\_e of LiveCodeBench taken from AtCoder) when run on DeepSeek V3.

1286 The “Sandwiches” problem is reproduced here:

1288  
1289 You are given a sequence of positive integers of length  $N$ :  $A =$   
1290  $(A_1, A_2, \dots, A_N)$ . Find the number of triples of positive integers  $(i, j, k)$   
1291 that satisfy all of the following conditions:

- 1292
- 1293 -  $1 \leq i < j < k \leq N$ ,
- 1294 -  $A_i = A_k$ ,
- 1295 -  $A_i \neq A_j$ .

1296 Input

1297

1298 The input is given from Standard Input in the following format:

1299

1300  $N$   
 $A_1 A_2 \dots A_N$

1301

1302 Output

1303

1304 Print the answer as an integer.

1305

1306 Constraints

1307

1308 -  $3 \leq N \leq 3 \times 10^5$

1309

1310 -  $1 \leq A_i \leq N$

1311 - All input values are integers.

1312

1313 Sample Input 1

1314

1315 5  
 1 2 1 3 2

1316

1317 Sample Output 1

1318

1319 3

1320

1321 The following three triples of positive integers  $(i, j, k)$  satisfy the conditions:

1322

1323 -  $(i, j, k) = (1, 2, 3)$

1324 -  $(i, j, k) = (2, 3, 5)$

1325 -  $(i, j, k) = (2, 4, 5)$

1326

1327 Sample Input 2

1328

1329 7  
 1 2 3 4 5 6 7

1330

1331 Sample Output 2

1332

1333 0

1334

1335 There may be no triples of positive integers  $(i, j, k)$  that satisfy the conditions.

1336

1337 Sample Input 3

1338

1339 13  
 9 7 11 7 3 8 1 13 11 11 11 6 13

1340

1341 Sample Output 3

1342

1343 20

1344

1345  
 1346 A correct  $O(N)$  solution to the problem is to iterate through the list  $A$  once, keeping track of the  
 1347 total number of valid triples in the subarray  $A_1, A_2, \dots, A_k$ ,  $\text{runningTotal}_k$ , in the  $k$ -th iteration.  
 1348 In iteration  $k$ , we add the number of new triples involving the  $k$ -th element of  $A$  to  $\text{runningTotal}$ .  
 1349 Therefore, in iteration  $k$ , we have to find the number of triples involving the index  $k$  in the subarray  
 $A_1, A_2, \dots, A_k$ .

In order to do so, let us introduce some supplementary notation. Let  $s$  be the integer at  $A_k$ . Let  $\text{count}_k$  be the number of occurrences of  $s$  in  $A_1, \dots, A_k$  (i.e. the index  $k$  is the  $\text{count}_k$ -th occurrence of  $s$  in the array  $A$ ). We let  $\text{other}_k = k - \text{count}_k$  be the number of integers in the subarray  $A_1, A_2, \dots, A_k$  which are not  $s$ . We overload this notation and let  $\text{other}_{i,k} = \text{other}_k - \text{other}_i$  be the number of integers in the subarray  $A_{i+1}, A_{i+2}, \dots, A_k$  which are not  $s$ . Then, we have:

$$\begin{aligned}
 & \text{runningTotal}_k - \text{runningTotal}_{k-1} \\
 &= \sum_{i=1}^k \mathbb{1}_{A_i=A_k}(\text{other}_{i,k}) \\
 &= \sum_{i=1}^k \mathbb{1}_{A_i=A_k}(\text{other}_k - \text{other}_i) \\
 &= \sum_{i=1}^k \mathbb{1}_{A_i=A_k}(\text{other}_k) - \sum_{i=1}^k \mathbb{1}_{A_i=A_k}(\text{other}_i) \\
 &= (\text{count}_k)(\text{other}_k) - \sum_{i=1}^k \mathbb{1}_{A_i=A_k}(\text{other}_i) \\
 &= (\text{count}_k)(k - \text{count}_k) - \sum_{i=1}^k \mathbb{1}_{A_i=A_k}(i - \text{count}_i)
 \end{aligned}$$

It turns out that keeping track of  $\text{count}_k$  and  $\sum_{i=1}^k \mathbb{1}_{A_i=A_k}(i - \text{count}_i)$  using hash maps (or Python dictionaries) can be done efficiently in expectation.

DeepSeek V3's initial code correctly applies this method, solving the problem (albeit with some redundant steps):

```

1377 import sys
1378 from collections import defaultdict
1379
1380 def solve():
1381     input = sys.stdin.read().split()
1382     ptr = 0
1383     N = int(input[ptr])
1384     ptr += 1
1385     A = list(map(int, input[ptr:ptr+N]))
1386
1387     # Step 1: Compute prefix counts
1388     prefix = [0] * (N + 1)
1389     count = defaultdict(int)
1390     for k in range(1, N + 1):
1391         x = A[k-1]
1392         prefix[k] = count[x]
1393         count[x] += 1
1394
1395     # Step 2: Reset count and compute running sums
1396     count = defaultdict(int)
1397     running_sum = defaultdict(int)
1398     answer = 0
1399     for k in range(1, N + 1):
1400         x = A[k-1]
1401         current_prefix = prefix[k]
1402         c = count[x]
1403         answer += c * (k - current_prefix) - running_sum[x]
1404         running_sum[x] += (k - current_prefix)
1405         count[x] += 1
1406     print(answer)

```

1404  
1405 solve()  
1406  
1407 However, on the input:  
1408  
1409 7  
1410 3 3 3 5 3 3 3  
1411  
1412 the expected output in the test generated was 10, where the correct output should be 9 — there are  
1413  $3 \times 3$  ways to pair up one index from the left of the 5, and one from the right of it.  
1414 Yet, in the diagnosis phase, the output 10 is picked due to a calculation error and a hallucinated  
1415 triple:  
1416  
1417 To solve this problem, we need to find the number of triples  $(i, j,$   
1418  $k)$  such that:  
1419 1.  $(1 \leq i < j < k \leq N)$ ,  
1420 2.  $(A_i = A_k)$ ,  
1421 3.  $(A_i \neq A_j)$ .  
1422  
1423 ### Step-by-Step Reasoning:  
1424  
1425 1. **Understanding the Problem**:  
1426 - We need to count all possible triples  $(i, j, k)$  where the  
1427 first and third elements are equal, and the middle element  
1428 is different from them.  
1429 - The indices must be in increasing order:  $(i < j < k)$ .  
1430  
1431 2. **Key Insight**:  
1432 - For each value  $(x)$  in the array, we can consider all pairs  
1433 of indices  $(i, k)$  where  $(A_i = A_k = x)$  and  $(i < k)$ .  
1434 - For each such pair  $(i, k)$ , the number of valid  $(j)$ 's is the  
1435 number of elements between  $(i)$  and  $(k)$  (i.e.,  $(k - i$   
1436  $- 1)$ ) minus the number of elements between  $(i)$  and  $(k)$   
1437 that are equal to  $(x)$ .  
1438  
1439 3. **Efficient Calculation**:  
1440 - For each occurrence of  $(x)$ , we can keep track of their  
1441 positions.  
1442 - For each  $(x)$ , iterate over all pairs of its positions  $(i, k$   
1443  $)$  where  $(i < k)$ .  
1444 - For each pair  $(i, k)$ , the number of valid  $(j)$ 's is  $((k - i$   
1445  $- 1) - (\text{number of } x \text{ 's between } i \text{ and } k))$ .  
1446 - Summing this over all valid pairs  $(i, k)$  for all  $(x)$  gives  
1447 the total count.  
1448  
1449 4. **Optimization**:  
1450 - Instead of iterating over all pairs  $(i, k)$  for each  $(x)$ , we  
1451 can use prefix sums to compute the number of valid  $(j)$ 's  
1452 efficiently.  
1453 - For each  $(x)$ , let the positions be  $(p_1, p_2, \dots, p_m$   
1454  $)$ . The total valid  $(j)$ 's for all pairs  $(i, k)$  is:  
1455 
$$\sum_{i=1}^m \sum_{k=i+1}^m (p_k - p_i - 1) - (k - i - 1)$$
  
1456 Simplifying, this becomes:  
1457 
$$\sum_{i=1}^m \sum_{k=i+1}^m (p_k - p_i - 1 - (k - i - 1))$$
  

$$= \sum_{i=1}^m \sum_{k=i+1}^m (p_k - p_i - k + i)$$

```

1458     \]
1459     Further simplification:
1460     \[
1461     \sum_{i=1}^m \sum_{k=i+1}^m (p_k - k) - (p_i - i)
1462     \]
1463     - This can be computed efficiently using prefix sums over  $(p_k$ 
1464        $- k)$  for each  $(x)$ .
1465
1466 5. **Applying to the Given Input**:
1467     - Input:
1468       \[
1469       7
1470       3 3 3 5 3 3 3
1471       \]
1472     - The positions of 3 are: 1, 2, 3, 5, 6, 7 (0-based or 1-based?
1473       Assuming 1-based here).
1474     - For  $(x = 3)$ , the positions are  $(p = [1, 2, 3, 5, 6, 7])$ .
1475     - Compute  $(p_k - k)$  for each  $(p_k)$ :
1476       -  $(p_1 - 1 = 0)$ ,
1477       -  $(p_2 - 2 = 0)$ ,
1478       -  $(p_3 - 3 = 0)$ ,
1479       -  $(p_4 - 4 = 1)$ ,
1480       -  $(p_5 - 5 = 1)$ ,
1481       -  $(p_6 - 6 = 1)$ .
1482     - The prefix sum array  $(S)$  for  $(p_k - k)$  is  $([0, 0, 0, 1,$ 
1483        $2, 3])$ .
1484     - For each  $(i)$ , the sum of  $((p_k - k) - (p_i - i))$  for  $(k$ 
1485        $> i)$  is:
1486       - For  $(i=1)$ :  $((0 + 0 + 1 + 2 + 3) - 5 \cdot 0 = 6)$ ,
1487       - For  $(i=2)$ :  $((0 + 1 + 2 + 3) - 4 \cdot 0 = 6)$ ,
1488       - For  $(i=3)$ :  $((1 + 2 + 3) - 3 \cdot 0 = 6)$ ,
1489       - For  $(i=4)$ :  $((1 + 1) - 2 \cdot 1 = 0)$ ,
1490       - For  $(i=5)$ :  $(1 - 1 \cdot 1 = 0)$ ,
1491       - For  $(i=6)$ :  $(0)$ .
1492     - Total sum:  $(6 + 6 + 6 + 0 + 0 + 0 = 18)$ .
1493     - However, this seems incorrect because the expected answer is
1494       likely smaller. Let's re-examine the calculation.
1495
1496 6. **Alternative Approach**:
1497     - For each pair  $(i, k)$  where  $(A_i = A_k = 3)$ , count the
1498       number of  $(j)$ 's between  $(i)$  and  $(k)$  where  $(A_j \neq$ 
1499        $3)$ .
1500     - Pairs of  $(i, k)$ :
1501       -  $(1, 2)$ :  $(j)$ 's: none (since  $(k - i - 1 = 0)$ ),
1502       -  $(1, 3)$ :  $(j = 2)$  (valid, since  $(A_2 = 3 \neq 3)$ ? No,  $($ 
1503          $A_2 = 3)$ , so invalid),
1504       -  $(1, 5)$ :  $(j = 2, 3, 4)$  (valid  $(j)$ 's: 4, since  $(A_4 = 5$ 
1505          $\neq 3)$ ),
1506       -  $(1, 6)$ :  $(j = 2, 3, 4, 5)$  (valid  $(j)$ 's: 4),
1507       -  $(1, 7)$ :  $(j = 2, 3, 4, 5, 6)$  (valid  $(j)$ 's: 4),
1508       -  $(2, 3)$ :  $(j)$ 's: none,
1509       -  $(2, 5)$ :  $(j = 3, 4)$  (valid  $(j)$ 's: 4),
1510       -  $(2, 6)$ :  $(j = 3, 4, 5)$  (valid  $(j)$ 's: 4),
1511       -  $(2, 7)$ :  $(j = 3, 4, 5, 6)$  (valid  $(j)$ 's: 4),
1512       -  $(3, 5)$ :  $(j = 4)$  (valid  $(j)$ 's: 4),
1513       -  $(3, 6)$ :  $(j = 4, 5)$  (valid  $(j)$ 's: 4),
1514       -  $(3, 7)$ :  $(j = 4, 5, 6)$  (valid  $(j)$ 's: 4),
1515       -  $(5, 6)$ :  $(j)$ 's: none,

```

```

1512     - (5, 7):  $(j = 6)$  (valid  $(j)$ 's: 6, since  $(A_6 = 3 \neq$ 
1513        $3)$ ? No, invalid),
1514     - (6, 7):  $(j)$ 's: none.
1515   - Summing the valid  $(j)$ 's:
1516     - (1, 5): 1,
1517     - (1, 6): 1,
1518     - (1, 7): 1,
1519     - (2, 5): 1,
1520     - (2, 6): 1,
1521     - (2, 7): 1,
1522     - (3, 5): 1,
1523     - (3, 6): 1,
1524     - (3, 7): 1,
1525     - (5, 7): 1.
1526   - Total: 10 valid triples.

```

```

1527   ### Final Answer:
1528   The correct output is `10`.

```

```

1529   ANSWER:

```

```

1530   ``
1531   10
1532   ``

```

When modifying the code, DeepSeek V3 again miscounts the number of triples, this time under-counting. However, it trusts our expected output. In any case, it produces an incorrect piece of code, leading to a regression.

```

1538   ### Understanding the Problem

```

The problem requires counting the number of triples  $(i, j, k)$  in a sequence of integers where:

1.  $(1 \leq i < j < k \leq N)$ ,
2.  $(A_i = A_k)$ ,
3.  $(A_i \neq A_j)$ .

Given the constraints ( $N$  up to  $(3 \times 10^5)$ ), the solution must be efficient, likely  $(O(N))$  or  $(O(N \log N))$ .

```

1548   ### Analyzing the Given Code

```

The provided code attempts to solve the problem in two steps:

1. **Prefix Count Calculation**: For each position  $(k)$ , it calculates how many times  $(A_k)$  has appeared before  $(k)$  (stored in `prefix[k]`).
2. **Running Sum Calculation**: For each  $(k)$ , it maintains a running sum of certain values to compute the answer.

The core logic for counting valid triples is in the second loop (lines 23–29). The formula used is:

```

1558    $[\text{answer} += c \times (k - \text{current\_prefix}) - \text{running\_sum}[x]]$ 

```

where:

- $(c)$  is the count of how many times  $(x = A_k)$  has been seen so far.
- $(\text{current\_prefix})$  is the number of times  $(x)$  appeared before  $(k)$  (from the first step).
- $(\text{running\_sum}[x])$  is a cumulative value for  $(x)$ .

```

1566   ### Identifying the Bug
1567
1568   The code fails on the input:
1569   ```
1570   7
1571   3 3 3 5 3 3 3
1572   ```
1573   Expected output: 10, but the code outputs 9.
1574
1575   Let's manually compute the expected triples:
1576   - For  $(i=1)$  ( $A=3$ ), possible  $(k)$  values are 3,5,6,7 (since  $(A_i = A_k = 3)$ ).
1577     - For  $(k=3)$ :  $(j)$  can be 2 ( $A=3$ , but  $(A_i = A_j = 3)$ ,
1578       invalid).
1579     - For  $(k=5)$ :  $(j)$  can be 2,3,4 ( $A=3,3,5$ ). Valid  $(j)$  is 4 ( $A=5$ ).
1580     - For  $(k=6)$ :  $(j)$  can be 2,3,4,5 ( $A=3,3,5,3$ ). Valid  $(j)$  are
1581       4 ( $A=5$ ).
1582     - For  $(k=7)$ :  $(j)$  can be 2,3,4,5,6 ( $A=3,3,5,3,3$ ). Valid  $(j)$ 
1583       are 4 ( $A=5$ ).
1584     - Total for  $(i=1)$ :  $1 (k=5) + 1 (k=6) + 1 (k=7) = 3$ .
1585
1586   - For  $(i=2)$  ( $A=3$ ), possible  $(k)$  values are 5,6,7.
1587     - For  $(k=5)$ :  $(j)$  can be 3,4 ( $A=3,5$ ). Valid  $(j)$  is 4 ( $A=5$ ).
1588     - For  $(k=6)$ :  $(j)$  can be 3,4,5 ( $A=3,5,3$ ). Valid  $(j)$  is 4 ( $A=5$ ).
1589     - For  $(k=7)$ :  $(j)$  can be 3,4,5,6 ( $A=3,5,3,3$ ). Valid  $(j)$  is
1590       4 ( $A=5$ ).
1591     - Total for  $(i=2)$ :  $1 + 1 + 1 = 3$ .
1592
1593   - For  $(i=3)$  ( $A=3$ ), possible  $(k)$  values are 6,7.
1594     - For  $(k=6)$ :  $(j)$  can be 4,5 ( $A=5,3$ ). Valid  $(j)$  is 4 ( $A=5$ ).
1595     - For  $(k=7)$ :  $(j)$  can be 4,5,6 ( $A=5,3,3$ ). Valid  $(j)$  is 4 ( $A=5$ ).
1596     - Total for  $(i=3)$ :  $1 + 1 = 2$ .
1597
1598   - For  $(i=4)$  ( $A=5$ ), no  $(k)$  exists where  $(A_k = 5)$  and  $(k > 4)$  (since  $(A_4=5)$  and next 5 is beyond the array).
1599     - Total for  $(i=4)$ : 0.
1600
1601   - For  $(i=5)$  ( $A=3$ ), possible  $(k)$  is 7.
1602     - For  $(k=7)$ :  $(j)$  can be 6 ( $A=3$ ). Invalid since  $(A_i = A_j = 3)$ .
1603     - Total for  $(i=5)$ : 0.
1604
1605   - For  $(i=6)$  ( $A=3$ ), no  $(k)$  exists beyond it.
1606     - Total for  $(i=6)$ : 0.
1607
1608   Summing up:  $3 (i=1) + 3 (i=2) + 2 (i=3) = 8$ . Wait, this
1609   contradicts the expected output of 10. Hmm, perhaps I missed
1610   some cases. Let me recount:
1611
1612   Wait, for  $(i=1, k=5)$ :  $(j)$  can be 2,3,4 ( $A=3,3,5$ ). Valid  $(j)$ 
1613   is 4 ( $A=5$ ). So 1.
1614   For  $(i=1, k=6)$ :  $(j)$  can be 2,3,4,5 ( $A=3,3,5,3$ ). Valid  $(j)$  is
1615   4 ( $A=5$ ). So 1.
1616   For  $(i=1, k=7)$ :  $(j)$  can be 2,3,4,5,6 ( $A=3,3,5,3,3$ ). Valid  $(j)$ 
1617   is 4 ( $A=5$ ). So 1.
1618   Total for  $(i=1)$ : 3.
1619

```

1620  
1621 For  $(i=2, k=5)$ :  $(j)$  can be 3,4 ( $A=3,5$ ). Valid  $(j)$  is 4 ( $A=5$ )  
1622 . So 1.  
1623 For  $(i=2, k=6)$ :  $(j)$  can be 3,4,5 ( $A=3,5,3$ ). Valid  $(j)$  is 4 ( $A=5$ ). So 1.  
1624  
1625 For  $(i=2, k=7)$ :  $(j)$  can be 3,4,5,6 ( $A=3,5,3,3$ ). Valid  $(j)$  is  
1626 4 ( $A=5$ ). So 1.  
1627 Total for  $(i=2)$ : 3.  
1628  
1629 For  $(i=3, k=6)$ :  $(j)$  can be 4,5 ( $A=5,3$ ). Valid  $(j)$  is 4 ( $A=5$ )  
1630 . So 1.  
1631 For  $(i=3, k=7)$ :  $(j)$  can be 4,5,6 ( $A=5,3,3$ ). Valid  $(j)$  is 4 ( $A=5$ ). So 1.  
1632  
1633 Total for  $(i=3)$ : 2.  
1634  
1635 For  $(i=4, k=...)$ : No  $(k)$  with  $(A_k=5)$  beyond  $(i=4)$ . So 0.  
1636  
1637 For  $(i=5, k=7)$ :  $(j)$  can be 6 ( $A=3$ ). Invalid since  $(A_i = A_j = 3)$ . So 0.  
1638  
1639 Total:  $3 + 3 + 2 = 8$ . But expected is 10. Maybe the expected  
1640 output is wrong? Or perhaps the counting is missing something.  
1641  
1642 Wait, perhaps the expected output is incorrect, but the code  
1643 outputs 9, which is also not matching our manual count of 8.  
1644 This suggests the code's logic might be overcounting or  
1645 undercounting in some way.  
1646  
1647 **### Debugging the Code**  
1648  
1649 The code's logic is non-trivial. Let's try to understand what it's  
1650 doing:  
1651 1. `prefix[k]` is the number of times  $(A_k)$  has appeared before  
1652  $(k)$ .  
1653 2. In the second loop, for each  $(k)$ , it updates `answer` using:  
1654 
$$[\text{answer}] += c \times (k - \text{current\_prefix}) - \text{running\_sum}[x]$$
  
1655 where:  
1656 -  $(c)$  is the count of  $(x)$  so far.  
1657 -  $(k - \text{current\_prefix})$  is roughly the number of  $(j)$ 's  
1658 's where  $(A_j \neq x)$  between the last occurrence of  $(x)$   
1659 and  $(k)$ .  
1660 - `running_sum[x]` seems to accumulate these values.  
1661  
1662 The issue might be that the formula doesn't correctly account for  
1663 all valid  $(j)$ 's when multiple  $(i)$ 's are involved.  
1664 Specifically, the subtraction of `running_sum[x]` might be  
1665 causing undercounting.  
1666  
1667 **### Correct Approach**  
1668  
1669 A better approach is to:  
1670 1. For each value  $(x)$ , maintain a list of its indices.  
1671 2. For each occurrence of  $(x)$  at position  $(k)$ , the number of  
1672 valid  $(i)$ 's is the number of  $(i)$ 's before  $(k)$  where  $(A_i = x)$ .  
1673 3. For each such  $(i)$ , the number of valid  $(j)$ 's is the number  
of elements between  $(i)$  and  $(k)$  that are not  $(x)$ .

1674 This can be optimized using prefix sums:  
 1675 - Precompute prefix sums where at each position, we count how many  
 1676 times each value has occurred up to that point.  
 1677 - For each  $x$ , iterate over its indices, and for each pair  $((i$   
 1678  $, k))$ , the number of valid  $j$ 's is  $((k - i - 1) - (\text{prefix}[k-1][x] - \text{prefix}[i][x]))$ .

1680 However, with  $N$  up to  $(3 \times 10^5)$ , we need an  $(O(N))$   
 1681 solution. The given code attempts this but seems to have a  
 1682 flaw in the formula.  
 1683

1684 **### Suggested Fix**

1685  
 1686 The bug is in the formula:  
 1687  $[\text{answer} += c \times (k - \text{current\_prefix}) - \text{running\_sum}[x]]$   
 1688  $[\text{running\_sum}[x]]$   
 1689 This seems to undercount because it subtracts `running_sum[x]`  
 1690 which might not correctly account for all valid  $j$ 's.

1691  
 1692 Instead, we can:  
 1693 1. For each  $x$ , keep track of the sum of indices where  $x$   
 1694 has appeared.  
 1695 2. For each new occurrence of  $x$  at  $k$ , the number of new  
 1696 valid triples is:  
 1697  $[(\text{count}[x] \times (k - 1) - \text{sum\_indices}[x]) -$   
 1698  $\text{prefix}[k]]$   
 1699 where:  
 1699 -  $(\text{count}[x])$  is the number of times  $x$  has appeared  
 1700 before  $k$ .  
 1701 -  $(\text{sum\_indices}[x])$  is the sum of indices where  $x$   
 1702 has appeared before  $k$ .  
 1703 -  $(\text{prefix}[k])$  is the number of times  $x$  has  
 1704 appeared before  $k$  (which seems redundant here).

1705 This is complex, and perhaps a simpler approach is to iterate over  
 1706 all possible  $x$ , and for each  $x$ , iterate over its  
 1707 indices, and for each pair of indices  $((i, k))$ , the number  
 1708 of valid  $j$ 's is the number of elements between  $i$  and  
 1709  $k$  that are not  $x$ . This can be optimized using prefix  
 1710 sums.

1711  
 1712 **### Rewriting the Code**

1713  
 1714 Here's a corrected approach:  
 1715 1. Precompute for each position, the number of times each value  
 1716 has occurred up to that point (prefix counts).  
 1717 2. For each value  $x$ , iterate over its indices, and for each  
 1718 pair  $((i, k))$ , the number of valid  $j$ 's is:  
 1719  $[(k - i - 1) - (\text{prefix}[k-1][x] - \text{prefix}[i][x])]$   
 1720 which simplifies to:  
 1721  $[(k - i - 1) - (\text{number of } x \text{'s between } i \text{ and } k)]$   
 1722

1723  
 1724 However, with  $N$  up to  $(3 \times 10^5)$ , we need to avoid  $(O(N^2))$   
 1725 operations. The given code's approach is likely correct  
 1726 in spirit but has an off-by-one or similar error.

1727 **### Final Answer**

1728  
 1729 The bug in the code is in the formula used to update `answer` in  
 1730 the second loop. The current formula undercounts because it  
 1731 incorrectly subtracts `running\_sum[x]`. A corrected version  
 1732 should more carefully account for the valid `(j)`'s between `(i`  
 1733 and `(k`).

1734  
 1735 Here's a corrected version of the code:

```
1736 ```python
1737 import sys
1738 from collections import defaultdict
1739
1740 def solve():
1741     input = sys.stdin.read().split()
1742     ptr = 0
1743     N = int(input[ptr])
1744     ptr += 1
1745     A = list(map(int, input[ptr:ptr+N]))
1746
1747     # Step 1: Compute prefix counts
1748     prefix = [0] * (N + 1)
1749     count = defaultdict(int)
1750     for k in range(1, N + 1):
1751         x = A[k-1]
1752         prefix[k] = count[x]
1753         count[x] += 1
1754
1755     # Step 2: Reset count and compute running sums
1756     count = defaultdict(int)
1757     running_sum = defaultdict(int)
1758     answer = 0
1759     for k in range(1, N + 1):
1760         x = A[k-1]
1761         current_prefix = prefix[k]
1762         c = count[x]
1763         answer += c * (k - 1 - current_prefix) - running_sum[x]
1764         running_sum[x] += (k - 1 - current_prefix)
1765         count[x] += 1
1766     print(answer)
1767
1768 solve()
1769 ```
```

1768 Key changes:

- 1769 1. Changed `c \* (k - current\_prefix)` to `c \* (k - 1 -`  
 1770 `current\_prefix)` to correctly account for the indices.
- 1771 2. This adjustment ensures that the number of valid `(j)`'s is  
 1772 correctly calculated between `(i` and `(k`.

1773  
 1774 This should now correctly output `10` for the input:

```
1775 ```
1776 7
1777 3 3 3 5 3 3 3
1778
```

1779  
 1780  
 1781

## 1782 H PROMPTS

1783

1784 We detail the prompts used in this section. The prompts were adapted from and inspired by those  
 1785 used in AgentCoder Huang et al. (2024b). The parts highlighted in **RED** are to be substituted with  
 1786 appropriate values for the problem.

1787

### 1788 H.1 CODE GENERATION

1789

#### 1790 **Initial Generation**

1791

1792 **\*\*Task\*\***: As a competitive programmer familiar with  
 1793 advanced data structures and algorithms, you are  
 1794 required to solve a given coding problem. Use a  
 1795 Chain-of-Thought approach to break down the  
 1796 problem, find a correct and efficient algorithm,  
 1797 create pseudocode, and then write the code in  
 1798 Python language.

1799

**\*\*Instructions\*\***:

- 1800 1. **\*\*Understand and Clarify\*\***: Make sure you
- 1801 understand the task.
- 1802 2. **\*\*Brainstorm\*\***: Relate the task to similar problems
- 1803 , and identify any relevant data structures and
- 1804 algorithms.
- 1805 3. **\*\*Analyze\*\***: Using examples, edge cases, and
- 1806 extreme cases, analyze various methods to solve
- 1807 the problem, with a focus on correctness and
- 1808 efficiency.
- 1809 4. **\*\*Time Complexity\*\***: Analyze the time complexity of
- 1810 the algorithms.
- 1811 5. **\*\*Algorithm/Method Selection\*\***: Decide on the most
- 1812 efficient way to solve the task correctly.
- 1813 6. **\*\*Pseudocode Creation\*\***: Write down the steps you
- 1814 will follow in pseudocode.
- 1815 7. **\*\*Code Generation\*\***: Translate your pseudocode into
- 1816 annotated, executable Python code, leaving

1817 **\*\*Code Formatting\*\***: Please write code in

```
1818 ```python
1819 [REPLACE THIS WITH YOUR CODE]
1820 ```
```

1821 format.

1822

1823 # For example:

1824

```
1825 ## Example Prompt:
1826 Output the sum of a list of numbers.
1827 The first line is the length of the list, the next
1828 line is the space-delimited list of numbers.
```

1829 ## Example Completion:

```
1830 ```python
1831 import sys
1832 n = eval(sys.stdin.readline().rstrip('\r\n'))
1833 lst = [int(i) for i in sys.stdin.readline().rstrip('\r\n').split()]
1834 output = 0
1835 for i in range(n):
```

```

1836         output += lst[i]
1837     sys.stdout.write(f'{output}\n')
1838     ```
1839
1840
1841     **Problem**:
1842
1843     PROBLEM DESCRIPTION
1844
1845     Use a Chain-of-Thought approach to break down the
1846     problem, find a correct and efficient algorithm,
1847     create pseudocode, and then write a correct code
1848     in Python language.
1849
1850     **Instructions**:
1851     1. **Understand and Clarify**: Make sure you
1852         understand the task.
1853     2. **Brainstorm**: Relate the task to similar problems
1854         , and identify any relevant data structures and
1855         algorithms.
1856     3. **Analyze**: Using examples, edge cases, and
1857         extreme cases, analyze various methods to solve
1858         the problem, with a focus on correctness and
1859         efficiency.
1860     4. **Time Complexity**: Analyze the time complexity of
1861         the algorithms.
1862     5. **Algorithm/Method Selection**: Decide on the most
1863         efficient way to solve the task correctly.
1864     6. **Pseudocode Creation**: Write down the steps you
1865         will follow in pseudocode.
1866     7. **Code Generation**: Translate your pseudocode into
1867         annotated, executable Python code, leaving
1868         comments explaining each step in the code.
1869
1870     In the time complexity analysis, keep in mind that
1871     operations on big integers in Python are not
1872     always constant-time. Optimize accordingly.
1873
1874     **Code Formatting**: Please write code in
1875     ```python
1876     [REPLACE THIS WITH YOUR CODE]
1877     ```
1878     format.
1879
1880     Remember to read the input from STDIN using `sys.stdin
1881     .read` and write the output to STDOUT using `sys.
1882     stdin.write`.
1883     Include all necessary import statements but WRITE ONLY
1884     THE SOLUTION AND DO NOT PROVIDE TEST CASES.
1885
1886     Reflection
1887
1888     **Task**: As a programmer, you are required to find
1889     the bug in a code. Use a Chain-of-Thought approach
1890     to trace the code and identify the bug.
1891
1892     **Instructions**:

```

- 1890 1. **\*\*Understand and Clarify\*\***: Make sure you  
1891 understand the problem.  
1892 2. **\*\*Code Tracing\*\***: Understand how the code runs.  
1893 3. **\*\*Reflection\*\***: Reflect on whether each execution  
1894 state is correct.  
1895 4. **\*\*Bug Identification\*\***: Identify the problem with  
1896 the code.

1897 **PROBLEM DESCRIPTION**

1898 **\*\*Code\*\***:  
1899 **GENERATED\_CODE**

1900 **\*\*Issue\*\***:

1901 **TEST INPUTS**

1902 The code is expected to **EXPECTED\_OUTPUTS**

1903 But the code **ERRONEOUS\_BEHAVIOUR**.

1904 Below is a snippet of the execution trace of the code  
1905 on the inputs:

1906 **EXECUTION TRACE**

1907 Identify the problem with the code. Describe in detail  
1908 why the code went wrong.

1909 Suggest if the code should be modified or re-written  
1910 entirely.

1911 **Code Refinement**

1912 **\*\*Task\*\***: As a competitive programmer familiar with  
1913 advanced data structures and algorithms, you are  
1914 required to solve a given coding problem. Use a  
1915 Chain-of-Thought approach to break down the  
1916 problem, find a correct and efficient algorithm,  
1917 create pseudocode, and then write the code in  
1918 Python language.

1919 **\*\*Instructions\*\***:

- 1920 1. **\*\*Understand and Clarify\*\***: Make sure you  
1921 understand the task.  
1922 2. **\*\*Brainstorm\*\***: Relate the task to similar problems  
1923 , and identify any relevant data structures and  
1924 algorithms.  
1925 3. **\*\*Analyze\*\***: Using examples, edge cases, and  
1926 extreme cases, analyze various methods to solve  
1927 the problem, with a focus on correctness and  
1928 efficiency.  
1929 4. **\*\*Time Complexity\*\***: Analyze the time complexity of  
1930 the algorithms.  
1931 5. **\*\*Algorithm/Method Selection\*\***: Decide on the most  
1932 efficient way to solve the task correctly.  
1933 6. **\*\*Pseudocode Creation\*\***: Write down the steps you  
1934 will follow in pseudocode.  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943

1944 7. **\*\*Code Generation\*\***: Translate your pseudocode into  
 1945 annotated, executable Python code, leaving  
 1946 comments explaining each step in the code.  
 1947

1948 **\*\*Code Formatting\*\***: Please write code in  
 1949 ````python`  
 1950 `[REPLACE THIS WITH YOUR CODE]`  
 1951 `````  
 1952 format.

1953 # For example:  
 1954

1955 ## Example Prompt:  
 1956 Output the sum of a list of numbers.  
 1957 The first line is the length of the list, the next  
 1958 line is the space-delimited list of numbers.  
 1959

1960 ## Example Completion:  
 1961 ````python`  
 1962 `import sys`  
 1963 `n = eval(sys.stdin.readline().rstrip('\r\n'))`  
 1964 `lst = [int(i) for i in sys.stdin.readline().rstrip('\r`  
 1965 `\n').split()]`  
 1966 `output = 0`  
 1967 `for i in range(n):`  
 1968 `output += lst[i]`  
 1969 `sys.stdout.write(f'{output}\n')`  
 1970 `````

1971

1972 **\*\*Problem\*\***:  
 1973 **PROBLEM DESCRIPTION**  
 1974  
 1975  
 1976

1977 **\*\*Existing Incorrect Code\*\***:  
 1978 The following code was tried on the problem.  
 1979 **GENERATED CODE**  
 1980

1981 However, the code fails on:  
 1982 **TEST INPUTS**  
 1983

1984 The existing code does not produce the expected output  
 1985 on the above inputs.

1986 A possible reason that the code provided does not work  
 1987 is that:  
 1988 **REFLECTIONS**  
 1989

1990 You may choose to modify the existing code, or to come  
 1991 up with an entirely new code and algorithm. Pick  
 1992 whichever approach is more appropriate.

1993 Use a Chain-of-Thought approach to break down the  
 1994 problem, find a correct and efficient algorithm,  
 1995 create pseudocode, and then write a correct code  
 1996 in Python language.

1997 **\*\*Instructions\*\***:

- 1998
- 1999
- 2000
- 2001
- 2002
- 2003
- 2004
- 2005
- 2006
- 2007
- 2008
- 2009
- 2010
- 2011
- 2012
- 2013
- 2014
- 2015
1. **\*\*Understand and Clarify\*\***: Make sure you understand the task.
  2. **\*\*Brainstorm\*\***: Relate the task to similar problems, and identify any relevant data structures and algorithms.
  3. **\*\*Analyze\*\***: Using examples, edge cases, and extreme cases, analyze various methods to solve the problem, with a focus on correctness and efficiency.
  4. **\*\*Time Complexity\*\***: Analyze the time complexity of the algorithms.
  5. **\*\*Algorithm/Method Selection\*\***: Decide on the most efficient way to solve the task correctly.
  6. **\*\*Pseudocode Creation\*\***: Write down the steps you will follow in pseudocode.
  7. **\*\*Code Generation\*\***: Translate your pseudocode into annotated, executable Python code, leaving comments explaining each step in the code.

2016

2017

2018

In the time complexity analysis, keep in mind that operations on big integers in Python are not always constant-time. Optimize accordingly.

2019

2020

2021

2022

2023

2024

```
**Code Formatting**: Please write code in
```python
[REPLACE THIS WITH YOUR CODE]
```
format.
```

2025

2026

2027

2028

2029

Remember to read the input from STDIN using ``sys.stdin.read`` and write the output to STDOUT using ``sys.stdin.write``.

Include all necessary import statements but **WRITE ONLY THE SOLUTION AND DO NOT PROVIDE TEST CASES.**

2030

## 2031 H.2 TEST GENERATION

2032

### 2033 **Basic Test Cases**

2034

2035

2036

2037

```
**Role**: As a tester, your task is to create
comprehensive test cases for a given problem.
```

2038

2039

2040

```
**Instructions**:
- Implement an extensive and comprehensive set of test
cases which conform to the problem constraints.
```

2041

2042

The format of each test case should be:

2043

```
Inputs:
```
```

2044

2045

2046

2047

```
Input line 1
Input line 2
Input line 3
```

2048

```
...
```

2049

```
Outputs:
```
```

2050

2051

```
Output line 1
Output line 2
```

```

2052     Output line 3
2053     ```
2054
2055
2056     For large test cases, you may choose to provide a
2057     python expression instead. If you choose to
2058     provide a python expression, label the code block
2059     appropriately as in:
2060     ```python
2061     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2062     ```
2063
2064     Provide either the plaintext or the python expression,
2065     NOT both.
2066
2067     # For example:
2068
2069     ## Example Prompt:
2070     Given list of numbers, determine if any two numbers
2071     are closer to each other than given threshold.
2072     The first line of the input is the length of the list
2073     n, and the threshold t.
2074     The next n lines of the input are the numbers in the
2075     list.
2076     The output is a single line. which is True if any two
2077     numbers in the list are closer than the threshold.
2078
2079     ## Example Completion:
2080     Inputs:
2081     ```
2082     3 1
2083     1.0
2084     2.0
2085     3
2086     ```
2087     Outputs:
2088     ```
2089     False
2090     ```
2091
2092     Inputs:
2093     ```python
2094     "10000 0.5\n" + " ".join(range(10000))
2095     ```
2096     Outputs:
2097     ```
2098     False
2099     ```
2100
2101     **Problem**:
2102
2103     PROBLEM DESCRIPTION
2104
2105     Remember to follow the format described and ENCLOSE
2106     THE TEST INPUTS AND EXPECTED OUTPUTS IN CODE
2107     BLOCKS as in:
2108
2109     Inputs:
2110     ```

```

```

2106     Input line 1
2107     Input line 2
2108     Input line 3
2109     ...
2110
2111     Outputs:
2112     ```
2113     Output line 1
2114     Output line 2
2115     Output line 3
2116     ...
2117
2118     If you choose to provide a python expression instead ,
2119     label the code block appropriately as in:
2120     ```python
2121     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2122     ```
2123
2124     DO NOT provide both the plaintext and python
2125     expression for the same test case.
2126     Only the test case inputs/outputs should be in the
2127     code blocks. DO NOT WRITE COMMENTS IN THE CODE
2128     BLOCKS.
2129     Ensure that the generated tests conform to the input
2130     constraints and format.
2131     Write the test cases in FULL and do not shorten them
2132     with ellipses.
2133     DO NOT GENERATE A SOLUTION TO THE PROBLEM.
2134
2135 Edge Scenarios
2136
2137
2138     **Role**: As a tester , your task is to think of a
2139     comprehensive list of edge scenarios for a given
2140     problem. These edge scenarios should test the code
2141     's robustness and reliability under extreme or
2142     unusual conditions.
2143
2144     **Instructions**:
2145     - Come up with an extensive and comprehensive list of
2146     edge scenarios for the given problem.
2147     - Pay special attention to edge cases as they often
2148     reveal hidden bugs.
2149
2150     The format of edge cases should be:
2151     - Edge Case 1
2152     - Edge Case 2
2153     - Edge Case 3
2154
2155     # For example:
2156
2157     ## Prompt 1:
2158     Given list of numbers , determine if any two numbers
2159     are closer to each other than given threshold.

```

```

## Completion 1:

```

2160           – No two numbers are closer to each other than the  
2161           threshold  
2162           – Only one pair of numbers are closer to each other  
2163           than the threshold  
2164           – There are negative numbers  
2165           – The difference between each pair of numbers is  
2166           exactly the threshold  
2167           – The threshold is 0  
2168           – The threshold is infinite  
2169  
2170       ## Prompt 2:  
2171       Given a string containing multiple groups of nested  
2172       parentheses, your goal is to separate each group  
2173       with a space.  
2174       Separate groups are balanced (each open brace is  
2175       properly closed) and not nested within each other.  
2176  
2177       ## Completion 2:  
2178       – There is a single large group of nested parentheses  
2179       – There are many small groups of nested parentheses  
2180       – There are many layers of nesting of parentheses  
2181  
2182       \*\*Problem\*\*:  
2183       **PROBLEM DESCRIPTION**  
2184  
2185  
2186       The edge scenarios should be unique but still able to  
2187       conform to the input format and constraints.  
2188       That is, the generated edge scenario should not be “  
2189       ensure that invalid inputs are flagged”, or  
2190       anything similar.  
2191  
2192       Remember to format the edge cases as separate bullet  
2193       points with “-” like:  
2194       – Edge Case 1  
2195       – Edge Case 2  
2196       – Edge Case 3  
2197  
2198       **Edge Test Cases**  
2199  
2200       \*\*Role\*\*:  
2201       As a tester, your task is to create  
2202       comprehensive test cases for a given scenario in a  
2203       given problem.  
2204  
2205       \*\*Instructions\*\*:  
2206       – Implement an extensive and comprehensive set of test  
2207       cases which conform to the problem constraints.  
2208       – Pay special attention to edge cases as they often  
2209       reveal hidden bugs.  
2210       The format of each test case should be:  
2211       Inputs:  
2212       ```  
2213       Input line 1  
2214       Input line 2  
2215       Input line 3

```
2214     ...
2215     ...
2216     Outputs:
2217     ...
2218     Output line 1
2219     Output line 2
2220     Output line 3
2221     ...
2222     ...
2223
2224     For large test cases, you may choose to provide a
2225     python expression instead. If you choose to
2226     provide a python expression, label the code block
2227     appropriately as in:
2228     ```python
2229     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2230     ```
2231
2232     Provide either the plaintext or the python expression,
2233     NOT both.
2234
2235     # For example:
2236
2237     ## Example Prompt:
2238     Given list of numbers, determine if any two numbers
2239     are closer to each other than given threshold.
2240     The first line of the input is the length of the list
2241     n, and the threshold t.
2242     The next n lines of the input are the numbers in the
2243     list.
2244     The output is a single line. which is True if any two
2245     numbers in the list are closer than the threshold.
2246
2247     Test for scenarios where The difference between each
2248     pair of numbers is exactly the threshold
2249
2250     ## Example Completion:
2251     Inputs:
2252     ```
2253     3 0.5
2254     1.5
2255     2
2256     2.5
2257     ```
2258
2259     Outputs:
2260     ...
2261     False
2262     ...
2263
2264     Inputs:
2265     ```
2266     6 0.3
2267     1.3
2268     2.5
2269     1.9
2270     2.8
2271     2.2
2272     1.6
2273     ```
```

```

2268     Outputs:
2269     ```
2270     False
2271     ```
2272
2273     Inputs:
2274     ```python
2275     "10000 1\n" + " ".join(range(10000))
2276     ```
2277     Outputs:
2278     ```
2279     False
2280     ```
2281
2282     **Problem**:
2283
2284     PROBLEM DESCRIPTION
2285
2286     Specifically, create a list of comprehensive tests for
2287     scenarios where EDGE SCENARIO
2288
2289     Remember to follow the format described and ENCLOSE
2290     THE TEST INPUTS AND EXPECTED OUTPUTS IN CODE
2291     BLOCKS as in:
2292     Inputs:
2293     ```
2294     Input line 1
2295     Input line 2
2296     Input line 3
2297     ...
2298     Outputs:
2299     ```
2300     Output line 1
2301     Output line 2
2302     Output line 3
2303     ...
2304
2305     If you choose to provide a python expression instead,
2306     label the code block appropriately as in:
2307     ```python
2308     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2309     ```
2310
2311     DO NOT provide both the plaintext and python
2312     expression for the same test case.
2313     Only the test case inputs/outputs should be in the
2314     code blocks. DO NOT WRITE COMMENTS IN THE CODE
2315     BLOCKS.
2316     Ensure that the generated tests conform to the input
2317     constraints and format.
2318     Write the test cases in FULL and do not shorten them
2319     with ellipses.
2320     DO NOT GENERATE A SOLUTION TO THE PROBLEM.
2321
2321     Large Test Cases

```

```

2322     **Role**: As a tester , your task is to create
2323         comprehensive test cases for a given problem.
2324         These test cases should encompass Large-Scale
2325         scenarios to ensure the code's performance ,
2326         reliability , and scalability with large data
2327         samples .
2328
2329     **Instructions**:
2330     - Implement a comprehensive set of test cases which
2331       conform to the problem constraints .
2332     - Ensure each test case is well-documented with
2333       comments explaining the scenario it covers .
2334     - Pay special attention to edge cases as they often
2335       reveal hidden bugs .
2336     - For large-scale tests , focus on the function 's
2337       efficiency and performance under heavy loads .
2338
2339     The format of each test case should be :
2340     Inputs :
2341     ```
2342     Input line 1
2343     Input line 2
2344     Input line 3
2345     ```
2346
2347     For large test cases , you may choose to provide a
2348     python expression instead . If you choose to
2349     provide a python expression , label the code block
2350     appropriately as in :
2351     ```python
2352     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2353     ```
2354
2355     Provide either the plaintext or the python expression ,
2356     NOT both .
2357
2358     You do not have to generate expected outputs .
2359
2360     # For example :
2361
2362     ## Example Prompt :
2363     Given list of numbers , determine if any two numbers
2364     are closer to each other than given threshold .
2365     The first line of the input is the length of the list
2366     n , and the threshold t .
2367     The next n lines of the input are the numbers in the
2368     list .
2369     The output is a single line . which is True if any two
2370     numbers in the list are closer than the threshold .
2371
2372     ## Example Completion :
2373     Inputs :
2374     ```
2375     3 1
2376     1.0
2377     2.0
2378     3
2379     ```

```

```

2376
2377     Inputs :
2378     ``` python
2379     "10000 0.5\n" + " ".join(range(10000))
2380     ```
2381
2382
2383     **Problem**:
```

**PROBLEM DESCRIPTION**

```

2386     Remember to follow the format described and ENCLOSE
2387     THE TEST INPUTS IN CODE BLOCKS as in:
2388     Inputs :
2389     ```
2390     Input line 1
2391     Input line 2
2392     Input line 3
2393     :::
2394
2395     Outputs :
2396     ```
2397     Output line 1
2398     Output line 2
2399     Output line 3
2400     :::
2401
2402     If you choose to provide a python expression instead ,
2403     label the code block appropriately as in:
2404     ``` python
2405     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2406     ```
2407
2408     DO NOT provide both the plaintext and python
2409     expression for the same test case.
2410     Only the test case inputs should be in the code blocks
2411     . DO NOT WRITE COMMENTS IN THE CODE BLOCKS.
2412     Ensure that the generated tests conform to the input
2413     constraints and format.
2414     GENERATE ONLY THE LARGE TEST INPUTS AND NOT THE
2415     EXPECTED OUTPUTS.
2416     Write the test cases in FULL and DO NOT SHORTEN THEM
2417     WITH ELLIPSES.
2418     DO NOT GENERATE A SOLUTION TO THE PROBLEM.
2419
2420 Coverage-guided Test Cases
2421
2422     **Role**:
```

As a tester , your task is to create a test case for a piece of code for a given problem.

```

2423
2424     **Instructions**:
```

The format of each test case should be:

```

2427     Inputs :
2428     ```
2429     Input line 1
2429     Input line 2
```

```

2430     Input line 3
2431     ...
2432
2433     Outputs:
2434     ...
2435     Output line 1
2436     Output line 2
2437     Output line 3
2438     ...
2439
2440     For large test cases, you may choose to provide a
2441     python expression instead. If you choose to
2442     provide a python expression, label the code block
2443     appropriately as in:
2444     ```python
2445     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2446     ```
2447
2448     Provide either the plaintext or the python expression,
2449     NOT both.
2450
2451     # For example:
2452
2453     ## Example Prompt:
2454     Given list of numbers, determine if any two numbers
2455     are closer to each other than given threshold.
2456     The first line of the input is the length of the list
2457     n, and the threshold t.
2458     The next n lines of the input are the numbers in the
2459     list.
2460     The output is a single line. which is True if any two
2461     numbers in the list are closer than the threshold.
2462
2463     ## Example Completion:
2464     Inputs:
2465     ...
2466     3 1
2467     1.0
2468     2.0
2469     3
2470     ...
2471     Outputs:
2472     ...
2473     False
2474     ...
2475     Inputs:
2476     ```python
2477     "10000 0.5\n" + " ".join(range(10000))
2478     ```
2479     Outputs:
2480     ...
2481     False
2482     ...
2483
2484     **Problem**:

```

2484  
2485  
2486  
2487  
2488  
2489  
2490  
2491  
2492  
2493  
2494  
2495  
2496  
2497  
2498  
2499  
2500  
2501  
2502  
2503  
2504  
2505  
2506  
2507  
2508  
2509  
2510  
2511  
2512  
2513  
2514  
2515  
2516  
2517  
2518  
2519  
2520  
2521  
2522  
2523  
2524  
2525  
2526  
2527  
2528  
2529  
2530  
2531  
2532  
2533  
2534  
2535  
2536  
2537

**PROBLEM DESCRIPTION**

**\*\*Code\*\*:**  
**GENERATED\_CODE**

Write a test case such that line **LINE\_NUMBER** of the above code will be executed.  
Use a Chain-of-Thought approach to break down the code and formulate a test case such that line **LINE\_NUMBER** will be executed.

Remember to follow the format described and ENCLOSE THE TEST INPUTS AND EXPECTED OUTPUTS IN CODE BLOCKS as in:

```
Inputs:
```
Input line 1
Input line 2
Input line 3
...
Outputs:
```
Output line 1
Output line 2
Output line 3
...

```

If you choose to provide a python expression instead, label the code block appropriately as in:  
```python  
REPLACE THIS LINE WITH THE PYTHON EXPRESSION  
```

DO NOT provide both the plaintext and python expression for the same test case.  
Only the test case inputs/outputs should be in the code blocks. DO NOT WRITE COMMENTS IN THE CODE BLOCKS.  
Ensure that the generated tests conform to the input constraints and format.  
Write the test cases in FULL and do not shorten them with ellipses.  
DO NOT GENERATE A SOLUTION TO THE PROBLEM.

**Coverage-guided Test Cases**

**\*\*Role\*\*:** As a tester, your task is to create a test case to expose a bug in a piece of code for a given problem.

**\*\*Instructions\*\*:**  
The format of each test case should be:  
Inputs:  
```\n

```

2538     Input line 1
2539     Input line 2
2540     Input line 3
2541     ...
2542
2543     Outputs:
2544     ...
2545     Output line 1
2546     Output line 2
2547     Output line 3
2548     ...
2549
2550     For large test cases, you may choose to provide a
2551     python expression instead. If you choose to
2552     provide a python expression, label the code block
2553     appropriately as in:
2554     ```python
2555     REPLACE THIS LINE WITH THE PYTHON EXPRESSION
2556     ```
2557
2558     Provide either the plaintext or the python expression,
2559     NOT both.
2560
2561     # For example:
2562
2563     ## Example Prompt:
2564     Given list of numbers, determine if any two numbers
2565     closer to each other than given threshold.
2566     The first line of the input is the length of the list
2567     n, and the threshold t.
2568     The next n lines of the input are the numbers in the
2569     list.
2570     The output is a single line. which is True if any two
2571     numbers in the list are closer than the threshold.
2572
2573     ## Example Completion:
2574     Inputs:
2575     ```
2576     3 1
2577     1.0
2578     2.0
2579     3
2580     ```
2581     Outputs:
2582     ```
2583     False
2584     ```
2585
2586     Inputs:
2587     ```python
2588     "10000 0.5\n" + " ".join(range(10000))
2589     ```
2590     Outputs:
2591     ```
2592     False
2593     ```

```

2592  
 2593  
 2594  
 2595  
 2596  
 2597  
 2598  
 2599  
 2600  
 2601  
 2602  
 2603  
 2604  
 2605  
 2606  
 2607  
 2608  
 2609  
 2610  
 2611  
 2612  
 2613  
 2614  
 2615  
 2616  
 2617  
 2618  
 2619  
 2620  
 2621  
 2622  
 2623  
 2624  
 2625  
 2626  
 2627  
 2628  
 2629  
 2630  
 2631  
 2632  
 2633  
 2634  
 2635  
 2636  
 2637  
 2638  
 2639  
 2640  
 2641  
 2642  
 2643  
 2644  
 2645

**\*\*Problem\*\*:**

**PROBLEM DESCRIPTION**

**\*\*Existing Code\*\*:**

**GENERATED CODE**

It appears that the existing code is incorrect. Find a valid, constraint-adhering test case for which the existing code fails to output the correct answer.

Remember to follow the format described and ENCLOSE THE TEST INPUTS AND EXPECTED OUTPUTS IN CODE BLOCKS as in:

Inputs:

```

Input line 1  
 Input line 2  
 Input line 3

```

Outputs:

```

Output line 1  
 Output line 2  
 Output line 3

```

If you choose to provide a python expression instead, label the code block appropriately as in:

```python

REPLACE THIS LINE WITH THE PYTHON EXPRESSION

```

DO NOT provide both the plaintext and python expression for the same test case.

Only the test case inputs/outputs should be in the code blocks. DO NOT WRITE COMMENTS IN THE CODE BLOCKS.

Ensure that the generated tests conform to the input constraints and format. The code is NOT REQUIRED to handle inputs that violate the input constraints and format.

Write the test cases in FULL and do not shorten them with ellipses.

DO NOT GENERATE A SOLUTION TO THE PROBLEM.

### H.3 ERROR DIAGNOSIS

#### Input Validation

**\*\*Task\*\*:** As a tester, you are required to figure out why a code runs into an error on a given input.

2646 Determine if the input is valid. Use a Chain-of-  
 2647 Thought approach to identify the issue.  
 2648

2649 **\*\*Instructions\*\*:**  
 2650 1. **\*\*Understand and Clarify\*\*:** Make sure you  
 2651 understand the problem.  
 2652 2. **\*\*Input Parsing\*\*:** Interpret what the input means.  
 2653 3. **\*\*Format Verification\*\*:** Compare the format of the  
 2654 input to that of the examples.  
 2655 4. **\*\*Parameter Verification\*\*:** Ensure that the input  
 2656 is self-consistent.  
 2657 5. **\*\*Logical Verification\*\*:** Ensure that the input has  
 2658 a corresponds to a valid problem instance with a  
 2659 valid solution.  
 2660 6. **\*\*Answer\*\*:** Answer YES if the input is valid , else  
 2661 NO

2662 **\*\*Answer Formatting\*\*:** Please write your answer on a  
 2663 single line after "ANSWER: " as  
 2664 ANSWER: YES  
 2665 or  
 2666 ANSWER: NO

2667 # For example:  
 2668

2669 ## Prompt 1:  
 2670 Output the sum of a list of numbers.  
 2671 The first line is the length of the list , the next  
 2672 line is the space-delimited list of numbers.  
 2673 Example:  
 2674 ```  
 2675 4  
 2676 1 2 3 4  
 2677 ```

2678 Actual Input:  
 2679 ```  
 2680 3  
 2681 1 2  
 2682 ```

2683

2684 ## Response 1:  
 2685 ANSWER: NO

2686 Reason: There are only 2 items in the list , not 3  
 2687

2688 ## Prompt 2:  
 2689 Output the sum of a list of numbers.  
 2690 The first line is the length of the list , the next  
 2691 line is the space-delimited list of numbers.  
 2692 Example:  
 2693 ```  
 2694 4  
 2695 "1" "2" "3" "4"  
 2696 ```

2697 Input:  
 2698 ```  
 2699 2

2700 "1" "2"  
2701 ` ` ` `

2702

2703 ## Response 2:  
2704 ANSWER: YES

2705

2706 Reason: The input format has quotation marks like in  
2707 the example; there are two numbers in the list as  
2708 stated in the paramters

2709

2710

2711

2712 \*\*Problem\*\*:

2713 **PROBLEM DESCRIPTION**

2714

2715

2716 \*\*Actual Input\*\*:

2717

2718 **TEST INPUTS**

2719

2720

2721 \*\*Instructions\*\*:

2722 1. \*\*Understand and Clarify\*\*:

2723 understand the problem.

2724 2. \*\*Input Parsing\*\*:

2725 Interpret what the input means.

2726 3. \*\*Format Verification\*\*:

2727 Compare the format of the input to that of the examples.

2728 4. \*\*Parameter Verification\*\*:

2729 Ensure that the input is self-consistent.

2730 5. \*\*Logical Verification\*\*:

2731 Ensure that the input has a corresponds to a valid problem instance with a valid solution.

2732 6. \*\*Answer\*\*:

2733 Answer YES if the input is valid, else NO

2734

2735 The input is provided as is to stdin (if there are ellipses and descriptions in the input, they are not expanded and the input thus is invalid).

2736 Apart from trailing whitespaces, input format should match the examples given EXACTLY (if there are spaces/brackets/quotation marks in the format of the examples, the input should have them too).

2737 Remember to write your answer on a single line after "

2738 ANSWER: " as

2739 ANSWER: YES

2740 or

2741 ANSWER: NO

2742

2743

2744

2745

2746

2747 **Correct Output Judgement**

2748

2749

2750 \*\*Role\*\*:

2751 As a mathematician and logician, your task is to explain the correct answer to a problem. Use a Chain-of-Thought approach to break down the problem and explain the answer step-by-step.

2752

2753

2754

2755

2756

2757

2758

2759

2760

2761

2762

2763

2764

2765

2766

2767

2768

2769

2770

2771

2772

2773

2774

2775

2776

2777

2778

2779

2780

2781

2782

2783

2784

2785

2786

2787

2788

2789

2790

2791

2792

2793

2794

2795

2796

2797

2798

2799

2800

2801

2802

2803

2804

2805

2806

2807

2808

2809

2810

2811

2812

2813

2814

2815

2816

2817

2818

2819

2820

2821

2822

2823

2824

2825

2826

2827

2828

2829

2830

2831

2832

2833

2834

2835

2836

2837

2838

2839

2840

2841

2842

2843

2844

2845

2846

2847

2848

2849

2850

2851

2852

2853

2854

2855

2856

2857

2858

2859

2860

2861

2862

2863

2864

2865

2866

2867

2868

2869

2870

2871

2872

2873

2874

2875

2876

2877

2878

2879

2880

2881

2882

2883

2884

2885

2886

2887

2888

2889

2890

2891

2892

2893

2894

2895

2896

2897

2898

2899

2900

2901

2902

2903

2904

2905

2906

2907

2908

2909

2910

2911

2912

2913

2914

2915

2916

2917

2918

2919

2920

2921

2922

2923

2924

2925

2926

2927

2928

2929

2930

2931

2932

2933

2934

2935

2936

2937

2938

2939

2940

2941

2942

2943

2944

2945

2946

2947

2948

2949

2950

2951

2952

2953

2954

2955

2956

2957

2958

2959

2960

2961

2962

2963

2964

2965

2966

2967

2968

2969

2970

2971

2972

2973

2974

2975

2976

2977

2978

2979

2980

2981

2982

2983

2984

2985

2986

2987

2988

2989

2990

2991

2992

2993

2994

2995

2996

2997

2998

2999

3000

3001

3002

3003

3004

3005

3006

3007

3008

3009

3010

3011

3012

3013

3014

3015

3016

3017

3018

3019

3020

3021

3022

3023

3024

3025

3026

3027

3028

3029

3030

3031

3032

3033

3034

3035

3036

3037

3038

3039

3040

3041

3042

3043

3044

3045

3046

3047

3048

3049

3050

3051

3052

3053

3054

3055

3056

3057

3058

3059

3060

3061

3062

3063

3064

3065

3066

3067

3068

3069

3070

3071

3072

3073

3074

3075

3076

3077

3078

3079

3080

3081

3082

3083

3084

3085

3086

3087

3088

3089

3090

3091

3092

3093

3094

3095

3096

3097

3098

3099

3100

3101

3102

3103

3104

3105

3106

3107

3108

3109

3110

3111

3112

3113

3114

3115

3116

3117

3118

3119

3120

3121

3122

3123

3124

3125

3126

3127

3128

3129

3130

3131

3132

3133

3134

3135

3136

3137

3138

3139

3140

3141

3142

3143

3144

3145

3146

3147

3148

3149

3150

3151

3152

3153

3154

3155

3156

3157

3158

3159

3160

3161

3162

3163

3164

3165

3166

3167

3168

3169

3170

3171

3172

3173

3174

3175

3176

3177

3178

3179

3180

3181

3182

3183

3184

3185

3186

3187

3188

3189

3190

3191

3192

3193

3194

3195

3196

3197

3198

3199

3200

3201

3202

3203

3204

3205

3206

3207

3208

3209

3210

3211

3212

3213

3214

3215

3216

3217

3218

3219

3220

3221

3222

3223

3224

3225

3226

3227

3228

3229

3230

3231

3232

3233

3234

3235

3236

3237

3238

3239

3240

3241

3242

3243

3244

3245

3246

3247

3248

3249

3250

3251

3252

3253

3254

3255

3256

3257

3258

3259

3260

3261

3262

3263

3264

3265

3266

3267

3268

3269

3270

3271

3272

3273

3274

3275

3276

3277

3278

3279

3280

3281

3282

3283

3284

3285

3286

3287

3288

3289

3290

3291

3292

3293

3294

3295

3296

3297

3298

3299

3300

3301

3302

3303

3304

3305

3306

3307

3308

3309

3310

3311

3312

3313

3314

3315

3316

3317

3318

3319

3320

3321

3322

3323

3324

3325

3326

3327

3328

3329

3330

3331

3332

3333

3334

3335

3336

3337

3338

3339

3340

3341

3342

3343

3344

3345

3346

3347

3348

3349

3350

3351

3352

3353

3354

3355

3356

3357

3358

3359

3360

3361

3362

3363

3364

3365

3366

3367

3368

3369

3370

3371

3372

3373

3374

3375

3376

3377

3378

3379

3380

3381

3382

3383

3384

3385

3386

3387

3388

3389

3390

3391

3392

3393

3394

3395

3396

3397

3398

3399

3400

3401

3402

3403

3404

3405

3406

3407

3408

3409

3410

3411

3412

3413

3414

3415

3416

3417

3418

3419

3420

3421

3422

3423

3424

3425

3426

3427

3428

3429

3430

3431

3432

3433

3434

3435

3436

3437

3438

3439

3440

3441

3442

3443

3444

3445

3446

3447

3448

3449

3450

3451

3452

3453

3454

3455

3456

3457

3458

3459

3460

3461

3462

3463

3464

3465

3466

3467

3468

3469

3470

3471

3472

3473

3474

3475

3476

3477

3478

3479

3480

3481

3482

3483

3484

3485

3486

3487

3488

3489

3490

3491

3492

3493

3494

3495

3496

3497

3498

3499

3500

3501

3502

3503

3504

3505

3506

3507

3508

3509

3510

3511

3512

3513

3514

3515

3516

3517

3518

3519

3520

3521

3522

3523

3524

3525

3526

3527

3528

3529

3530

3531

3532

3533

3534

3535

3536

3537

3538

3539

3540

3541

3542

3543

3544

3545

3546

3547

3548

3549

3550

3551

3552

3553

3554

3555

3556

3557

3558

3559

3560

3561

3562

3563

3564

3565

3566

3567

3568

3569

3570

3571

3572

3573

3574

3575

3576

3577

3578

3579

3580

3581

3582

3583

3584

3585

3586

3587

3588

3589

3590

3591

3592

3593

3594

3595

3596

3597

3598

3599

3600

3601

3602

3603

3604

3605

3606

3607

3608

3609

3610

3611

3612

3613

3614

3615

3616

3617

3618

3619

3620

3621

3622

3623

3624

3625

3626

3627

3628

3629

3630

3631

3632

3633

3634

3635

3636

3637

3638

3639

3640

3641

3642

3643

3644

3645

3646

3647

3648

3649

3650

3651

3652

3653

3654

3655

3656

3657

3658

3659

3660

3661

3662

3663

3664

3665

3666

3667

3668

3669

3670

3671

3672

3673

3674

3675

3676

3677

3678

3679

3680

3681

3682

3683

3684

3685

3686

3687

3688

3689

3690

3691

3692

3693

3694

3695

3696

3697

3698

3699

3700

3701

3702

3703

3704

3705

3706

3707

3708

3709

3710

3711

3712

3713

3714

3715

3716

3717

3718

3719

3720

3721

3722

3723

3724

3725

3726

3727

3728

3729

3730

3731

3732

3733

3734

3735

3736

3737

3738

3739

3740

3741

3742

3743

3744

3745

3746

3747

3748

3749

3750

3751

3752

3753

3754

3755

3756

3757

3758

3759

3760

3761

3762

3763

3764

3765

3766

3767

3768

3769

3770

3771

3772

3773

3774

3775

3776

3777

3778

3779

3780

3781

3782

3783

3784

3785

3786

3787

3788

3789

3790

3791

3792

3793

3794

3795

3796

3797

3798

3799

3800

3801

3802

3803

3804

3805

3806

3807

3808

3809

3810

3811

3812

3813

3814

3815

3816

3817

3818

3819

3820

3821

3822

3823

3824

3825

3826

3827

3828

3829

3830

3831

3832

3833

3834

3835

3836

3837

3838

3839

3840

3841

3842

3843

3844

3845

3846

3847

3848

3849

3850

3851

3852

3853

3854

3855

3856

3857

3858

3859

3860

3861

3862

3863

3864

3865

3866

3867

3868

3869

3870

3871

3872

3873

3874

3875

3876

3877

3878

3879

3880

3881

3882

3883

3884

3885

3886

3887

3888

3889

3890

3891

3892

3893

3894

3895

3896

3897

3898

3899

3900

3901

3902

3903

3904

3905

3906

3907

3908

3909

3910

3911

3912

3913

3914

3915

3916

3917

3918

3919

3920

3921

3922

3923

3924

3925

3926

3927

3928

3929

3930

3931

3932

3933

3934

3935

3936

3937

3938

3939

3940

3941

3942

3943

3944

3945

3946

3947

3948

3949

3950

3951

3952

3953

3954

3955

3956

3957

3958

3959

3960

3961

3962

3963

3964

3965

3966

3967

3968

3969

3970

3971

3972

3973

3974

3975

3976

3977

3978

3979

3980

3981

3982

3983

3984

3985

3986

3987

3988

3989

3990

3991

3992

3993

3994

3995

3996

3997

3998

3999

4000

4001

4002

4003

4004

4005

4006

4007

4008

4009

4010

4011

4012

4013

4014

4015

4016

4017

4018

4019

4020

4021

4022

4023

4024

4025

4026

4027

4028

4029

4030

4031

4032

4033

```

2754 1. **Understand and Clarify**: Make sure you
2755     understand the problem.
2756 2. **Parsing**: Interpret the inputs in the context of
2757     the problem.
2758 3. **Reasoning**: Reason about the problem and inputs
2759     rigorously and work out the answer step-by-step.
2760 4. **Deliberation**: Critically consider the arguments
2761     for the possible answers.
2762 5. **Answer the question**: Decide on the correct
2763     answer based on careful and extensive analysis.
2764
2765 **Answer Formatting**: Please write your answer as in
2766     a code block after the word 'ANSWER:' as
2767 ANSWER:
2768     ```
2769     [REPLACE THIS WITH THE CORRECT ANSWER]
2770     ```
2771
2772 # For example:
2773
2774 ## Example Prompt:
2775 Output the sum of a list of numbers.
2776 The first line is the length of the list, the next
2777     line is the space-delimited list of numbers.
2778
2779 What is the correct output on input:
2780     ```
2781     2
2782     1 2
2783     ```
2784
2785 ## Example Completion:
2786 ANSWER:
2787     ```
2788     3
2789     ```
2790
2791 **Problem**:
2792
2793 PROBLEM DESCRIPTION
2794
2795 TEST INPUTS
2796
2797 Should the output be
2798     ```
2799 EXPECTED OUTPUT
2800     ```
2801
2802 ACTUAL OUTPUT
2803     ```
2804
2805 or something else?
2806
2807 (The correct answer is likely one of the listed
2808     options. But if you think that another answer is
2809     correct, output that as the answer instead.)

```

2808           \*\*Instructions\*\*:  
2809           1. **Understand and Clarify**: Make sure you  
2810                 understand the problem.  
2811           2. **Parsing**: Interpret the inputs in the context of  
2812                 the problem.  
2813           3. **Reasoning**: Reason about the problem and inputs  
2814                 rigorously and work out the answer step-by-step.  
2815           4. **Deliberation**: Critically consider the arguments  
2816                 for the possible answers.  
2817           5. **Answer the question**: Decide on the correct  
2818                 answer based on careful and extensive analysis.  
  
2819           Analyze step-by-step, using a Chain-of-Thought  
2820                 approach.  
2821           Remember to write your answer in a code block after  
2822                 the word 'ANSWER:' as  
2823           ANSWER:  
2824                 ```\n2825                 [REPLACE THIS WITH THE CORRECT ANSWER]\n2826                 ```\n  
2827  
2828  
2829  
2830  
2831  
2832  
2833  
2834  
2835  
2836  
2837  
2838  
2839  
2840  
2841  
2842  
2843  
2844  
2845  
2846  
2847  
2848  
2849  
2850  
2851  
2852  
2853  
2854  
2855  
2856  
2857  
2858  
2859  
2860  
2861