

LiveCodeBench Pro: How Do Olympiad Medalists Judge LLMs in Competitive Programming?

Zihan Zheng^{1,*§}, Zerui Cheng^{2,*}, Zeyu Shen^{2,*}, Shang Zhou^{3,*}, Kaiyuan Liu^{4,*}, Hansen He^{5,*}, Dongruixuan Li⁶, Stanley Wei², Hangyi Hao⁷, Jianzhu Yao², Peiyao Sheng⁸, Zixuan Wang², Wenhao Chai^{2,†§}, Aleksandra Korolova^{2,†}, Peter Henderson^{2,†}, Sanjeev Arora^{2,†}, Pramod Viswanath^{2,8,†}, Jingbo Shang^{3,†,‡}, Saining Xie^{1,†,‡}

¹ New York University

² Princeton University

³ University of California San Diego

⁴ University of Washington

⁵ Canyon Crest Academy

⁶ University of Waterloo

⁷ McGill University

⁸ Sentient Foundation

Link: [Leaderboard](#) | [Evaluation Code](#) | [Problem Set](#)

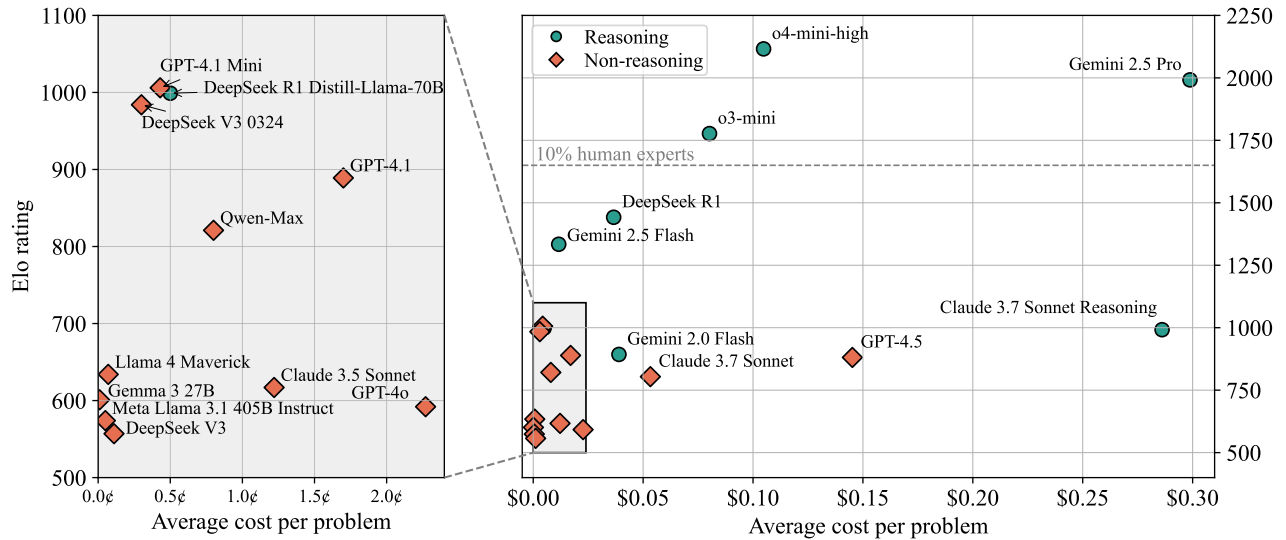


Fig. 1: LiveCodeBench Pro leaderboard. Elo rating versus average cost per problem for various models. The gray region zooms in on the cluster of non-reasoning models.

* Equal Contributions. § Project Lead. † Advisors. ‡ Equal Advising.

Abstract. Recent reports claim that large language models (LLMs) now outperform elite humans in competitive programming. Drawing on knowledge from a group of medalists in international algorithmic contests, we revisit this claim, examining how LLMs differ from human experts and where limitations still remain. We introduce LiveCodeBench Pro, a benchmark composed of problems from Codeforces, ICPC, and IOI that are continuously updated to reduce the likelihood of data contamination. A team of Olympiad medalists annotates every problem for algorithmic categories and conducts a line-by-line analysis of failed model-generated submissions. Using this new data and benchmark, we find that frontier models still have significant limitations: without external tools, the best model achieves only 53% pass@1 on medium-difficulty problems and 0% on hard problems, domains where expert humans still excel. We also find that LLMs succeed at implementation-heavy problems but struggle with nuanced algorithmic reasoning and complex case analysis, often generating confidently incorrect justifications. High performance appears largely driven by implementation precision and tool augmentation, not superior reasoning. LiveCodeBench Pro thus highlights the significant gap to human grandmaster levels, while offering fine-grained diagnostics to steer future improvements in code-centric LLM reasoning.

1. Introduction

Large Language Models (LLMs) have demonstrated extraordinary advances in code generation and problem-solving. Modern models can now easily solve textbook-style puzzles [14] and achieve near-perfect accuracy on HumanEval [15], and strong performance on more recent evaluations such as SWE-Bench [36] and LiveCodeBench [35]. For instance, Claude 3.5 Sonnet achieves 92% pass@1 on HumanEval [11]. Recent releases touted as reasoning breakthroughs, such as DeepSeek R1 [31] and OpenAI o3 [34], have pushed performance towards the upper limits of these benchmarks. To gauge what headroom remains, researchers increasingly adopt competitive programming [16, 29, 32, 33, 35, 49, 51, 53] as a benchmark – its mathematically rigorous problems and fully automated, pass-fail grading on exhaustive hidden test suites eliminate subjective judgment and demand end-to-end mastery of problem formalization, algorithm design, and bug-free implementation. When augmented with external tools like terminal access and search engines, models such as o3 and o4-mini-high can reportedly attain Elo ratings above 2,700 [48] on Codeforces [41], the premier competitive-programming platform with high-quality problems and frequent contests, placing it in the top 0.1% of participants.

Yet, these simple quantitative evaluations do not fully capture what it means to solve complex algorithmic problems. Are LLMs truly capable of reasoning at the level of elite human competitors? Do current benchmarks accurately reflect the conceptual difficulty of problems, or are they skewed toward implementation skill? And how much of the performance is driven by reasoning, as opposed to tool use? Our work revisits these questions.

Existing evaluations [14, 19, 28, 37–39, 54, 56] fall short of answering these questions. Benchmarks like LiveCodeBench [35] offer coding problems, but suffer from inconsistent environments, weak test cases vulnerable to false positives, unbalanced difficulty distributions, and the inability to isolate the effects of search contamination. SWE-Bench [36], though valuable for assessing software engineering scenarios, focuses more on code maintenance than algorithm design. CodeELO [49] introduces a competitive programming-focused framework using Codeforces problems, but still relies on static archives, making it difficult to disentangle genuine reasoning from memorization, especially for models with recent cutoffs or retrieval capabilities. Moreover, its analysis is largely limited to noisy

Table 1: Pass@1 and Elo rating performance on LiveCodeBench Pro. Each model’s Elo-based *Rating* is computed from head-to-head comparisons with human participants, while the *Pct.%* column shows the model’s percentile among all human contestants. *AvgTok* is the average number of tokens generated per problem and *AvgCost* is the approximate \$-cost per problem. We also test o4-mini, although its release date was later than the benchmark curation. Additional details are in Section A2.

Model	Hard	Medium	Easy	Rating	Pct.%	AvgTok	AvgCost
<i>Reasoning Models</i>							
o4-mini-high	0.0%	53.5%	83.1%	2 116	1.5%	23 819	\$0.1048
Gemini 2.5 Pro	0.0%	25.4%	70.4%	1 992	2.3%	29 879	\$0.2988
o3-mini	0.0%	16.9%	77.5%	1 777	4.9%	18 230	\$0.0802
DeepSeek R1	0.0%	9.9%	56.3%	1 442	18.0%	16 716	\$0.0366
Gemini 2.5 Flash	0.0%	12.7%	47.9%	1 334	30.3%	35 085	\$0.0116
DeepSeek R1 Distill-Llama-70B	0.0%	2.8%	33.8%	999	56.0%	12 425	\$0.0050
Claude 3.7 Sonnet (Max Reasoning)	0.0%	1.4%	36.6%	992	56.5%	19 075	\$0.2861
Gemini 2.0 Flash Reasoning	0.0%	0.0%	29.6%	893	63.1%	11 143	\$0.0390
<i>Non-Reasoning Models</i>							
GPT-4.1 mini	0.0%	5.6%	28.2%	1 006	55.5%	2 662	\$0.0043
DeepSeek V3 0324	0.0%	5.6%	32.4%	984	57.1%	2 712	\$0.0030
GPT-4.1	0.0%	0.0%	23.9%	889	64.2%	2 131	\$0.0170
GPT-4.5	0.0%	0.0%	26.8%	881	64.8%	968	\$0.1452
Qwen-Max	0.0%	0.0%	14.1%	821	69.4%	1 244	\$0.0080
Claude 3.7 Sonnet (No Reasoning)	0.0%	1.4%	16.9%	804	70.7%	3 554	\$0.0533
Llama 4 Maverick	0.0%	0.0%	15.5%	634	80.4%	1 160	\$0.0007
Claude 3.5 Sonnet	0.0%	0.0%	14.1%	617	81.4%	810	\$0.0122
Gemma 3 27B	0.0%	0.0%	8.5%	601	82.5%	668	\$0.0001
GPT-4o	0.0%	0.0%	9.9%	592	83.1%	1 133	\$0.0227
Meta Llama 3.1 405B Instruct	0.0%	0.0%	9.9%	574	84.3%	568	\$0.0005
DeepSeek V3	0.0%	0.0%	12.7%	557	84.9%	1 020	\$0.0011

user-generated tags [52], which obscure the nuanced challenges of algorithmic reasoning.

To address these limitations, we introduce LiveCodeBench Pro, a challenging, daily-updated competitive programming benchmark. It contains 584 high-quality problems before April 25, 2025, drawn from top-tier contests including Codeforces [41], ICPC series [3], and IOI series [4]. In contrast to LiveCodeBench, we omit LeetCode problems, which are easier and especially prone to training-data contamination, ensuring that our set remains both more challenging and cleaner. We capture problems as soon as they appear in live contests, reducing data-contamination risk. Our team of competition coding experts and Olympiad medalists annotate every problem with key skills required for each task (e.g., knowledge of combinatorics or dynamic programming), as well as whether they are knowledge-heavy, observation-heavy, or logic-heavy problems, where the detailed definition of the cognitive-focus taxonomy is elaborated in Section 2. This yields metadata for analyzing trends in model performance, and laying the groundwork for targeted research into model failures.

We evaluate a suite of frontier models on LiveCodeBench Pro, including models such as Gemini 2.5 Pro, o4-mini-high, and DeepSeek R1. For each, we compute Elo-equivalent ratings for LLMs across our expert-annotated skill tags. Then we leverage our team of experts to conduct an analysis of

Table 2: LiveCodeBench Pro Statistics for annotated tags with corresponding cognitive-focus taxonomy. Breakdown by competitive programming topic, showing problem counts, percentages, and average Codeforces difficulty ratings (Elo \pm std. dev.).

Category	Tag	Cognitive Focus	# Problems	% Problems	Difficulty (Elo)
Mathematics	Number Theory	Logic	77	13%	1 884 \pm 825
	Combinatorics	Logic	62	11%	2 423 \pm 666
	Game Theory	Observation	27	5%	1 900 \pm 827
	Others	Logic	118	20%	1 608 \pm 775
Greedy	—	Observation	163	28%	1 708 \pm 791
Data Structures	Segment Tree	Knowledge	38	7%	2 629 \pm 502
	Others	Knowledge	101	17%	2 108 \pm 689
Dynamic Programming	—	Logic	134	23%	2 431 \pm 614
Graph Theory	Tree	Knowledge	53	9%	2 308 \pm 528
	Others	Knowledge	42	7%	2 331 \pm 680
String	—	Logic	44	8%	1 595 \pm 867
Algorithmic Paradigms	Constructive	Observation	120	21%	1 849 \pm 894
	Implementation	Knowledge	108	18%	2 057 \pm 837
	Ad-hoc	Observation	103	18%	1 578 \pm 814
	Case Work	Observation	89	15%	1 713 \pm 830
	Binary Search	Observation	80	14%	2 021 \pm 652
	Bitmasking	Logic	46	8%	2 174 \pm 679
	Two Pointers	Knowledge	22	4%	1 777 \pm 528
	Interactive	Observation	21	4%	2 152 \pm 643
Total			584	100%	1 827 \pm 822

specific model behaviors. We conduct a line-by-line comparison of 125 failed submissions each from o3-mini and human participants of similar rating to diagnose failure modes. We also compare reasoning models with their non-reasoning counterparts and identify unusual patterns in o3-mini-high’s handling of interactive problems. Finally, we analyze the impact of tool usage (e.g., web search, terminal access) for o4-mini-high. We find that:

- (1) Current models excel in more structured and knowledge-heavy problems that require more logical derivation than deduction, but they perform significantly worse on observation-heavy problems demanding observation and creativity. Only problems on combinatorics, segment tree, and dynamic programming see o4-mini-high perform above a grandmaster level.
- (2) Compared to human experts, conceptual errors dominate model failures, whereas implementation is a relative strength. LLMs frequently fail even on provided sample inputs, suggesting incomplete utilization of given information and indicating room for improvement even in simple settings.
- (3) Reasoning models show large performance improvements over their non-reasoning counterparts in combinatorics and knowledge-heavy problems, while gains are limited for observation-heavy ones.
- (4) Although multiple attempts (pass@k) substantially improve overall performance, models still struggle to solve problems in the hard tier.

LiveCodeBench Pro is designed to be live and continuously updated, providing an ongoing, up-to-date challenge for future models. Our work shows the importance of granular expert-level analyses, showing where models still underperform humans, even as aggregate scores paint a different picture.

2. Benchmark Curation

Benchmark curation pipeline. Our 584-problem corpus is assembled in real time as contests unfold, capturing each problem before any accepted solutions, editorials, or discussion threads appear online. Doing so eliminates data leakage pathways that have plagued earlier coding benchmarks. Every candidate problem must (i) originate from a premier competition like Codeforces, ICPC, IOI; (ii) pass the host’s multi-layer vetting, which typically involves a coordinator plus two or more expert testers who stress-test the judge data until no known buggy or inefficient implementation survives; and (iii) expose its full, immutable test set at contest end so that subsequent model runs are evaluated under the same rules as humans. This pipeline is detailed in Appendix A3.

Difficulty tiers. To streamline navigation, we adopt a Codeforces-style rating heuristic for difficulty labels. The Elo difficulty of a problem means that a contestant of the corresponding rating can solve the problem with a 50% probability. Problems with an official Elo rating ≤ 2000 are flagged **Easy**: a world-class contestant can typically solve them in about 15 minutes using standard textbook techniques and observations. The **Medium** tier, $(2000, 3000]$, contains problems that demand the fusion of two or more established algorithms together with non-trivial mathematical reasoning and observations. Anything rated > 3000 is deemed **Hard**. These challenges usually hinge on an extremely involved, non-obvious derivation or deductive leap that requires both a masterful grasp of algorithmic theory and deep mathematical intuition; they elude more than 99.9% of participants and sometimes remain unsolved even by the strongest competitors during live contests.

Tags and cognitive-focus taxonomy. We list the definition of each tag shown in Table 2 along with the example problems in Section A11. In addition to these detailed tags, we group the problems into three overarching categories based on their cognitive focus, as described below. Coupling the tutorial-grade difficulty tiers with cognitive-focus tags turns the benchmark from a mere scoreboard into a diagnostic instrument. In practice, a problem may straddle categories; annotators therefore assign primary and secondary labels, reaching consensus through a triple-blind adjudication process described in Section A3.2. Through these, we can isolate whether an LLM’s weakness stems from shallow algorithmic insight or brittle code generation.

- **Knowledge-heavy.** *If a contestant comes equipped with ready-to-use templates, often very long and structured code snippets, or knows a deep mathematical result that’s virtually impossible to re-derive during the contest, the apparent difficulty of a problem drops sharply.* In these cases, the real test is breadth of knowledge and implementation, not discovery. With a solid command of the underlying techniques, the core idea usually jumps out from the statement or samples, and the primary challenge then becomes writing a correct, bug-free, and highly optimized implementation.

Example Knowledge-heavy Problem (Atcoder Beginner Contest 196 F. Substring 2)

Given binary strings $S \in \{0,1\}^n$ and $T \in \{0,1\}^m$, compute the minimum number of bit flips needed to make T a substring of S .

Data range: $1 \leq m \leq n \leq 10^6$ **Time limit:** 3s **Memory limit:** 1024M

Example: AtCoder Beginner Contest 196 F. Substring 2 is a direct application of Fast Fourier Transform (FFT). The key insight is to rephrase “count mismatches” as a convolution where $\sum_{j=1}^m \mathbf{1}[S_{i+j} \neq T_j] = \sum_{j=1}^m S_{i+j} \oplus T_j = \sum_{j=1}^m S_{i+j}(1 - T_j) + \sum_{j=1}^m (1 - S_{i+j})T_j$. Each of the two sums

is just a convolution of two binary sequences (one derived from S , one from the reversed T). By reversing T and computing these convolutions with FFT, we get every mismatch-count in one pass in $O((n+m)\log(n+m))$ instead of $O(nm)$ brute-force. Solving this task relies heavily on an in-depth understanding of convolution and the FFT.

- **Logic-heavy.** *When a contestant combines strong mathematical reasoning with systematic, step-by-step derivations, the apparent difficulty of a problem collapses.* Success here depends on meticulous logical work, often in combinatorics or generating-function algebra, rather than a flash of insight. The solver must translate symbolic manipulations into an efficient algorithmic recipe (for instance, deriving a closed-form recurrence).

Example Logic-heavy Problem (Codeforces 626F. Group Projects)

Given integers n, t and an array a_1, a_2, \dots, a_n , count the number of ways to partition the array a into disjoint groups (singleton groups allowed) so that the total imbalance, defined as the sum over all groups of $(\max a \text{ in group} - \min a \text{ in group})$, is at most t . Output the count modulo $10^9 + 7$ (a large prime number).

Data range: $1 \leq n \leq 200, 0 \leq t \leq 1000, 1 \leq a_i \leq 500$ **Time limit:** 2s **Memory limit:** 256M

Example: Codeforces 626F. Group Projects requires carefully deriving and optimizing the state design and transition of dynamic programming (DP). After sorting a , a 3D state design is clear where $dp[i][j][k]$ denotes the number of ways to place the first i elements into j open groups with total imbalance k . Upon adding a_{i+1} , there are two choices: start a new group, which increments j by 1 and subtracts a_{i+1} from the imbalance k ; or join one of the j open groups where j, k remain unchanged. We also need to further distinguish whether a_{i+1} becomes the allocated group's maximum: if it does, we close the group, decrementing j by 1 and adding a_{i+1} back into k ; if it doesn't, then neither j nor k changes. The naive implementation lets k range from $-\sum a_t$ to $\sum a_t$ and runs in $O(n^2 \sum a_t)$, which doesn't fit into the time limit. For optimization, the key insight is $a_R - a_L = \sum_{i=L}^{R-1} (a_{i+1} - a_i)$ where every gap $a_{i+1} - a_i \geq 0$, allowing the total imbalance k to stay in range $[0, t]$, bounding the imbalance dimension by t and yielding an $O(n^2 t)$ algorithm. Though every step is relatively intuitive, solving this task requires a rigorous step-by-step chain of thought that demands inventing the right state, carefully tracking updates, and spotting a subtle non-negativity bound for optimization, making this task logic-heavy.

- **Observation-heavy.** *When a contestant instantly spots a concise insight, typically an "aha" moment straight from the problem statement, the apparent difficulty evaporates.*

That single observation dramatically collapses the search space: Once the insight is in hand, the implementation is brief and template-free. This style of problem rewards creativity, mathematical intuition, and deductive leaps rather than long, step-by-step derivations.

Example Observation-heavy Problem (Codeforces 1704F. Colouring Game)

On a row of n cells initially colored red or blue, Alice and Bob alternate moves: Alice chooses two adjacent cells containing at least one red and repaints both white; Bob then chooses two adjacent cells containing at least one blue and repaints both white. The first player unable to move loses. Determine who wins under optimal play.

Data range: $1 \leq n \leq 5 \cdot 10^5$ **Time limit:** 1s **Memory limit:** 256M

Example: Codeforces 1704F. Colouring Game requires deducing each player’s optimal move, writing a brute-forcing calculation of Sprague-Grundy value on small n , and observing a subtle pattern in the value table. From the deduction on each player’s optimal move, the player with more cells of its color always wins, and in the case of an equal number of red and blue cells, the only thing that matters is the length of the alternating RB or BR substrings. To solve for each substring, we can use Sprague-Grundy theorem [30] to compute the SG value of each configuration in $O(n^2)$ time where $SG(n) = \text{mex}_{i=0}^{n-2} SG(i) \oplus SG(n-2-i)$. Then, from the value table, a cyclic section of size 34 can be found where $SG(n) = SG(n-34)$ for $n \geq 100$, reducing large n to small $\hat{n} \leq 100$, allowing a brute-forcing $O(\hat{n}^2)$ implementation to pass. Yet, once that insight is uncovered, the entire solution can be coded up within twenty lines.

3. Analysis and Discussions

3.1. Performance on Different Algorithmic Paradigms

Finding 1. LLMs perform better on knowledge-heavy and logic-heavy problems, and worse on observation-heavy problems or case work.

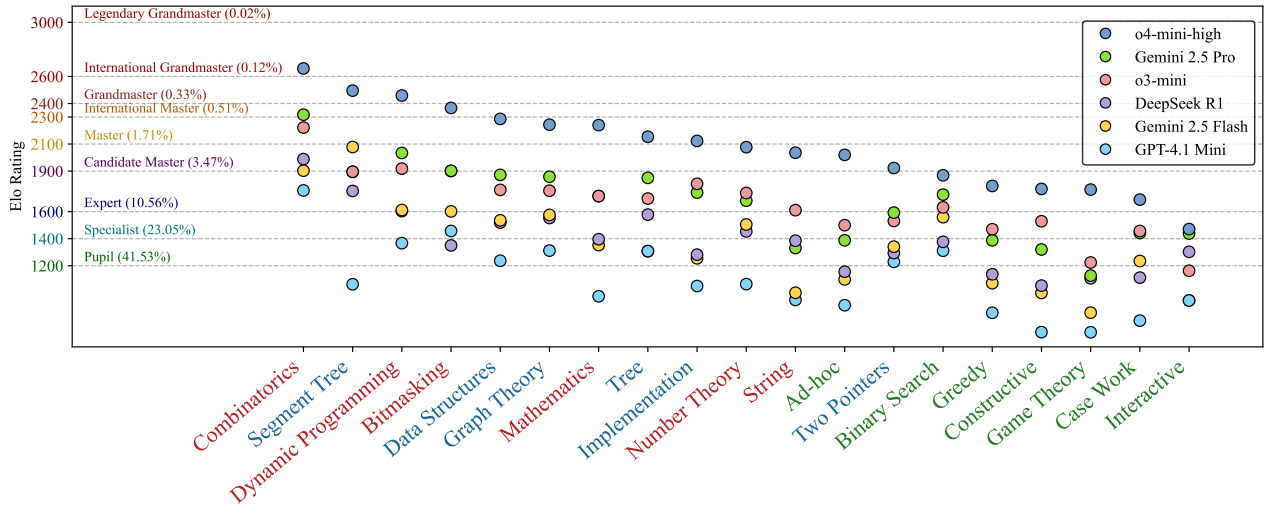


Fig. 2: Tag-wise model performance. The x -axis represents different problem types, ranging from knowledge-heavy problems in blue (e.g., segment tree, implementation, data structures) to logic-heavy in red (e.g., combinatorics, dynamic programming, mathematics) to observation-heavy problems in green (e.g., greedy, interactive, game theory). The y -axis corresponds to Codeforces-equivalent Elo ratings, with human percentile benchmarks labeled (e.g., Master). Models tend to excel on knowledge-heavy and logic-heavy problems but struggle on observation-heavy ones.

We present the performance, in terms of ratings computed with the Bayesian approach in Appendix A3.2, of 6 models in coding in each problem category based on our annotations. We found that humans exhibit more consistent performance across different problem tags, while models show greater variance in their ratings depending on the tag. We summarize our key findings as:

Knowledge-heavy problems are comfort zones for LLMs. Problems with tags such as segment tree, graph theory, tree, and data structures exhibit high performance in most models. These problems are

often solvable by stitching together well-known templates (e.g., Fenwick tree, Dijkstra, Euler tour), a setting in which LLMs excel because the requisite patterns appear verbatim in the training data, and generating syntactically correct templates is much easier for LLMs than for humans.

Logic-heavy problems yield similarly good results. LLMs also perform well on logic-heavy categories such as combinatorics, mathematics, dynamic programming, and binary search, which require a more patterned way of thinking (e.g., applying combinatorial identities for combinatorics, constructing a state space and deriving transition functions for dynamic programming) and can benefit from memorized scaffolds of code.

Bad performance on observation-heavy problems. For game theory, ad-hoc, greedy, and constructive problems, ratings of most models collapse to below 1500, which is significantly below knowledge-heavy and logic-heavy categories. Solving these problems usually hinges on the discovery of novel insights, something that cannot be retrieved from memorized snippets alone.

LLMs struggle with case work. Interestingly, all models struggle with case work. Every model except o4-mini-high stays below the 1500-rating mark, and even o4-mini-high performs significantly worse compared with other problem categories. Manual inspection reveals that the inability to identify and deal with edge cases is a prominent failure mode of all models.

Interactive problems expose a pronounced weakness. o4-mini-high sees its rating collapse to around 1500 on interactive problems, and other models struggle as well. We provide discussions on possible reasons behind such bad performance and identify unusual behaviors of o3-mini-high when solving interactive problems in Appendix A8.

3.2. Diagnosis of Failure Reasons and Comparison with Humans

***Finding 2.** o3-mini makes significantly more algorithm logic errors and wrong observations, and much fewer implementation logic errors than humans.*

In this section, we specifically use o3-mini, which is the model with the best readability, for annotation and in-depth analysis. We present the results in the treemap in Figure 3.

Conceptual errors dominate the model’s failures. The largest red tile inside the “Idea Error” branch shows that o3-mini commits 34 more algorithm logic errors than human contestants among the 125 annotated problems. These are genuine conceptual slips, instead of surface bugs.

Implementation is the model’s strong suit. Metrics related to low-level coding generally favor o3-mini. For instance, o3-mini commits 25 fewer implementation logic errors than humans among the 125 annotated problems. Notably, all observed initialization errors and I/O format errors were found in human submissions. The verdict-level breakdown also corroborates this: o3-mini almost makes no “Runtime Error.” underscoring its comparative immunity to implementation mistakes.

A notable outlier - Idleness Limit Exceeded. One deep-red rectangle under “Verdict” shows a spike in “Idleness Limit Exceeded” verdicts. This stems from the peculiar behavior of o3-mini on interactive problems, most of which are judged as “Idleness Limit Exceeded.” More in Appendix A8.

Failures on sample inputs. The treemap highlights a substantial surplus of 45 instances for o3-mini in the “Fails Sample” category, where solutions compile but already fail on the problem’s example inputs. Unlike humans, o3-mini cannot compile locally or run sample inputs before submission. Models with terminals and tool calls (e.g., o3 and o4-mini-high) are expected to make far fewer of these easy-to-catch mistakes.

Note that all of our evaluations are run through the OpenAI API, where o4-mini-high lacks tool call and terminal access. However, its web version does expose these features; we analyze their impact in Appendix A7.

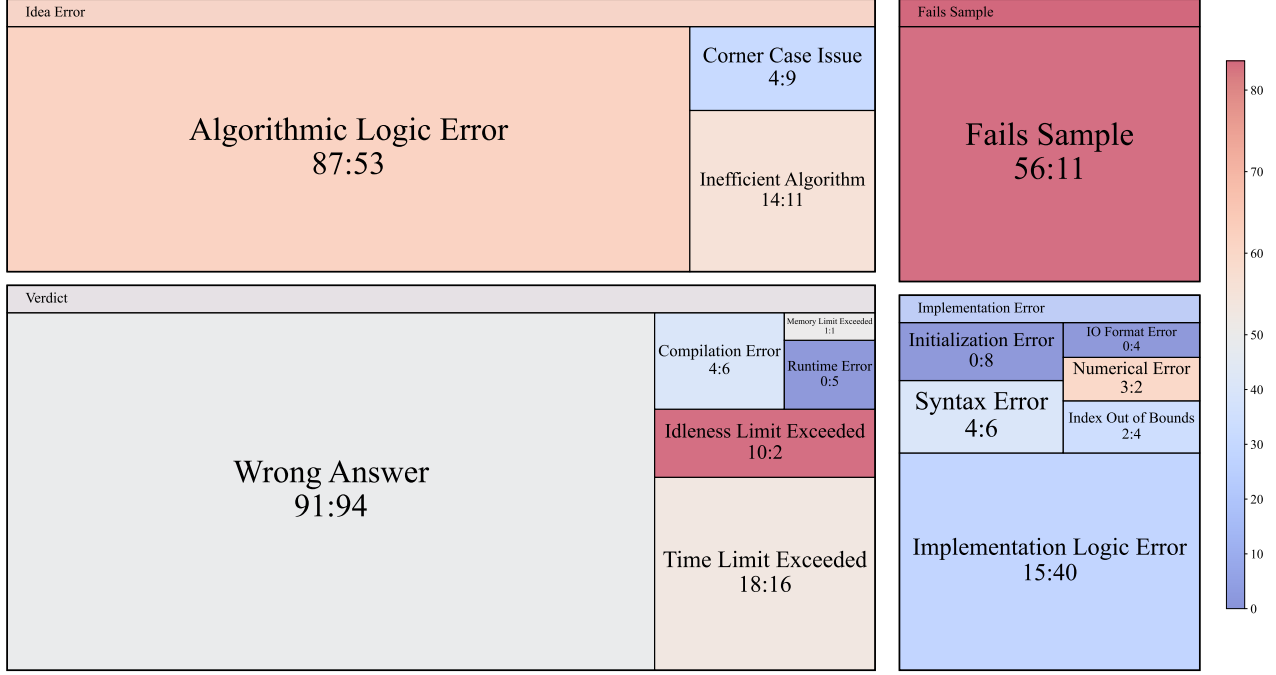


Fig. 3: Failure reasons in treemap. Comparing rejected submissions between o3-mini and humans. Each block represents a specific rejection tag; its size is proportional to the total count of rejections for that tag, with the inscribed text showing the o3-mini:human ratio. The color of the block indicates the contribution rate of o3-mini to that tag’s rejections: red signifies a higher proportion from o3-mini, while blue indicates a higher proportion from humans.

Taken together, our analysis suggests that LLMs’ code is generally more syntactically reliable, but struggles with the higher-level reasoning needed to craft correct algorithms or extract the right observations from a problem. Although our formal annotations cover only o3-mini’s submissions, preliminary manual checks suggest that the same error pattern is shared by most existing LLMs.

3.3. Impact of Multiple Attempts (Pass@k) on Model Performance

Finding 3. Increasing the number of attempts (pass@k) significantly improves the performance of the models while still failing in the hard tier.

OpenAI’s reported Codeforces Elo rating of 2719 for o4-mini with terminal access and pass@k [48] contrasts with our evaluation of o4-mini-high, which achieved a rating of 2116 (without terminal access and pass@1). This discrepancy motivates an investigation into the performance impact of terminal access and tool calls versus the effect of allowing multiple attempts (pass@k).

As illustrated in Figure 4, the models demonstrate a substantial improvement in ratings as k increases. For example, the o4-mini-medium’s rating rises from 1793 at pass@1 and converges to 2334 as k increases to 10. Similar upward trends are observed for o4-mini-low and o4-mini-high. While these gains from multiple attempts are significant, the converged rating still falls approximately 400

Due to instability from excessively long reasoning chains (up to 100K tokens) and prohibitive costs (~\$200/pass), we limited o4-mini-high evaluation to pass@3.

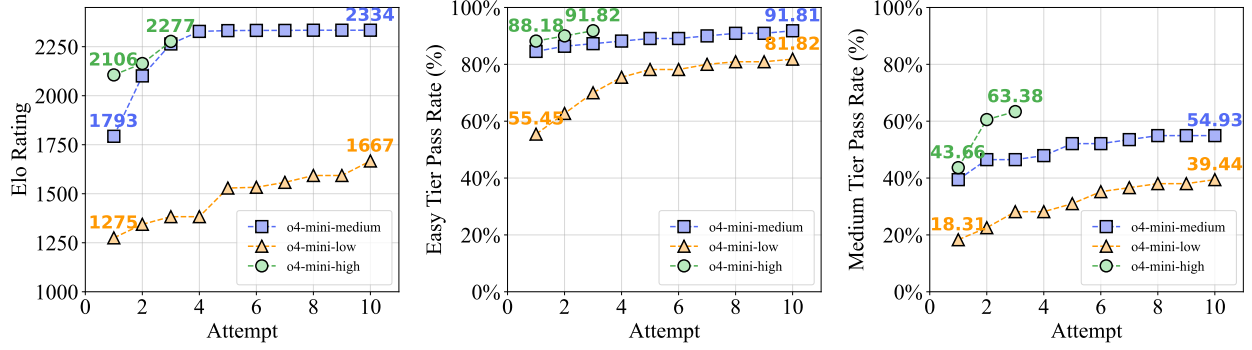


Fig. 4: o4-mini performance under pass@k settings. The plot shows the pass rates for Easy and Medium tier problems, and the corresponding Elo rating changes as the number of attempts (k) increases. All variants show 0% pass rate on the hard tier in the evaluation.

points short of the reported 2719. We, therefore, conjecture that the remaining difference is largely attributable to the benefits of tool calls and terminal access. More details are in Appendix A7.

As shown in Figure 5, we observe that among the five categories with the greatest improvement, three—Game Theory, Greedy, and Case Work—are observation-heavy problems and can often be solved by hypothesizing conclusions. A higher frequency of making educated guesses substantially increases the probability of solving these problems correctly.

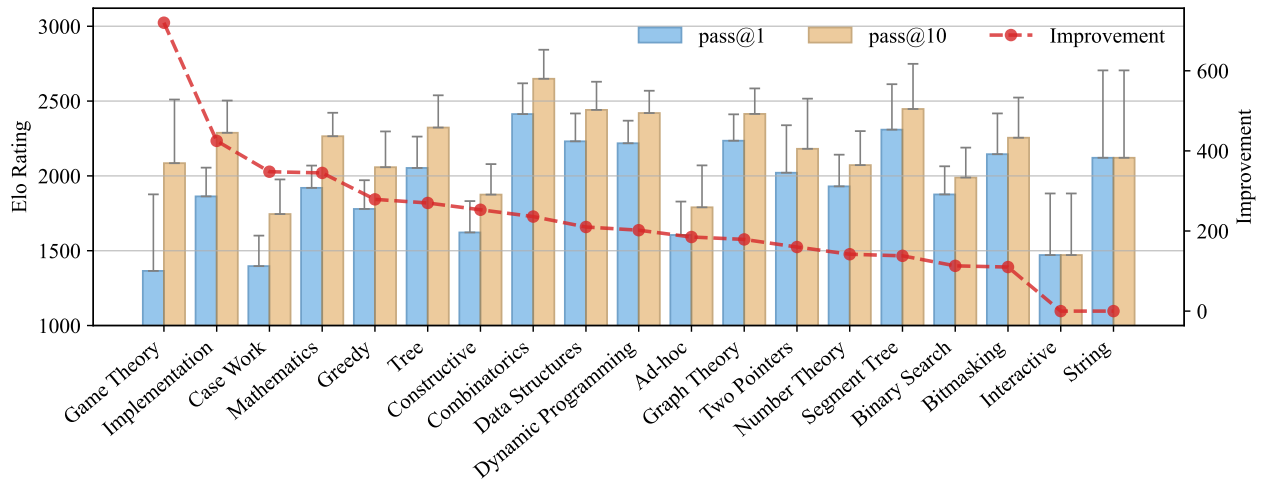


Fig. 5: Performance improvement with pass@k setting across tags. The Elo ratings for pass@1 and pass@10 across different problem categories show significant performance improvements. The evaluated model is o4-mini-medium. Error bars representing the 95% confidence intervals.

3.4. Comparison Between Reasoning Models and Their Non-reasoning Counterparts

Finding 4. Reasoning brings about the largest improvement in combinatorics, a large improvement in knowledge-heavy categories, and relatively low improvement in observation-heavy ones.

In this section, we examine the impact of enabling reasoning in LLMs on each problem tag. In particular, we directly compare reasoning models and their non-reasoning counterparts, so that we can control for variations in model architecture, training data, and other external factors, and isolate the effect of reasoning. This isolation is crucial to demonstrate the true impact of additional chain-of-thought or test-time scaling methods on models’ problem-solving capabilities in each problem tag. In particular, we choose to compare DeepSeek V3 versus R1 and Claude 3.7 Sonnet Nonthinking versus Thinking, which are two major frontier models with a non-reasoning version and a reasoning counterpart as shown in Figure 6.

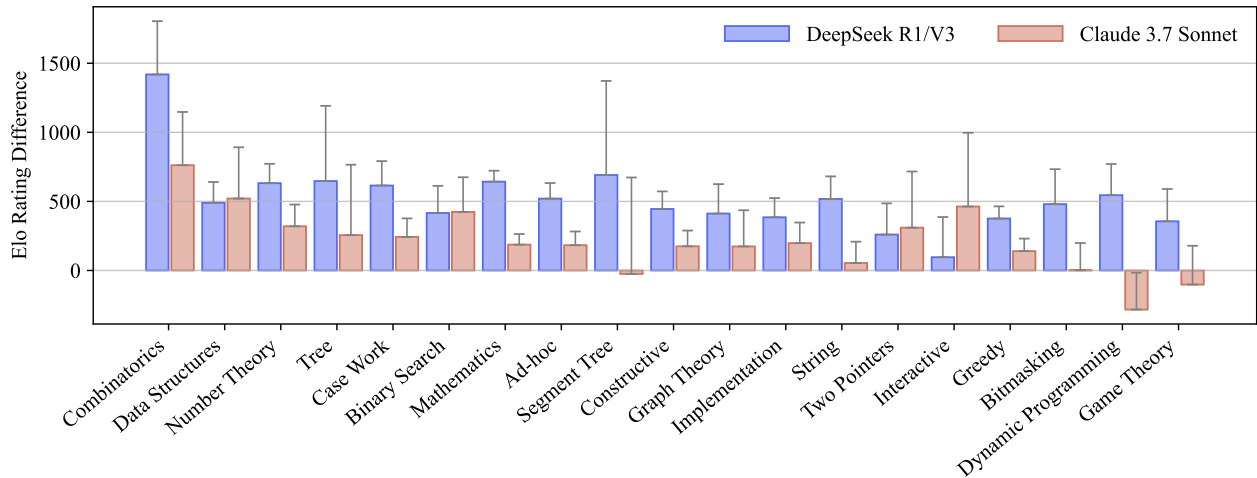


Fig. 6: Tag-wise Elo rating advantage of reasoning models. Error bars represent the 95% confidence intervals for the Elo rating differences. The reasoning models outperform across nearly all tags, though the magnitude of their advantage varies significantly.

Our key findings are summarized as:

Largest improvement in combinatorics. Both models show the largest improvement in combinatorics, with DeepSeek R1 exhibiting nearly a 1400-point rating gain over V3.

Large improvement in knowledge-heavy categories. For knowledge-heavy problems such as data structure and segment tree, enabling reasoning also results in large improvement (e.g., boosting around 700 points for DeepSeek on segment tree and 500 points for Claude on data structures). This is expected since problems in these categories often involve structured thinking.

Limited improvement in observation-heavy categories. Intriguingly, for game theory, greedy, ad-hoc and constructive problems, which usually require significant amounts of observations and LLMs often struggle with (as shown in Section 3.1), even reasoning brings minimal improvement (e.g., the improvement in game theory is almost the lowest for DeepSeek and negative for Claude). This raises the question of whether current chain-of-thought methods are inherently limited for these types of problems, or if there is an emergent threshold, that is, a point at which further advancements in reasoning capabilities might eventually unlock substantial performance gains in these areas.

4. Conclusions and Future Work

In this work, we introduce LiveCodeBench Pro, a rigorously curated and contamination-free benchmark designed to evaluate the true algorithmic reasoning capabilities of LLMs in competitive programming. With expert annotation and fine-grained comparison with human competitors, our study reveals that current LLMs demonstrate proficiency in implementation-oriented problems; they exhibit stark limitations in complex algorithmic reasoning, nuanced problem-solving, and handling edge cases, failing entirely on the benchmark’s hardest problems. Despite claims of surpassing elite humans, a significant gap still remains, particularly in areas demanding novel insights. In future work, we plan to build a more automated and controllable submission and analysis pipeline.

Acknowledgments

We would like to extend our sincere gratitude to the open-source community whose collective efforts have made this project possible. Our special thanks go to Chuanjie Luo, Chengrui Han, and Chenran Yang, whose direct contributions through open-source initiatives have significantly enriched this work. We are also deeply appreciative of the competitive programming communities, particularly Codeforces and QOJ, for their resources and insightful discussions. We also thank Zihan Zheng (*API-credit sponsor*) for generously sponsoring the API credits used in this research.

References

- [1] HumanEval: Hand-Written Evaluation Set. URL <https://github.com/openai/human-eval>.
- [2] ICPC Foundation, . URL <https://icpc.foundation/>.
- [3] The ICPC International Collegiate Programming Contest, . URL <https://icpc.global/>.
- [4] International Olympiad in Informatics. URL <https://ioinformatics.org/>.
- [5] MIT Informatics Tournament. URL <https://mitit.org/About>.
- [6] Chinese National Olympiad in Informatics. URL <https://noi.cn/>.
- [7] Tsinghua University Programming Contest. URL <https://thusaac.com/public>.
- [8] USA Computing Olympiad. URL <https://usaco.org/>.
- [9] ICPC Fact Sheet, 2025. URL <https://icpc.global/worldfinals/fact-sheet/ICPC-Fact-Sheet.pdf>.
- [10] Anthropic. Model card addendum: Claude 3.5 sonnet. https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf, 2024.
- [11] Anthropic. Claude 3.5 Sonnet, 2024. URL <https://www.anthropic.com/news/claude-3-5-sonnet>.
- [12] Anthropic. System card: Claude 3.7 sonnet (max reasoning). <https://assets.anthropic.com/m/785e231869ea8b3b/original/claude-3-7-sonnet-system-card.pdf>, 2025.
- [13] Anthropic. System card: Claude 3.7 sonnet (no reasoning variant). <https://assets.anthropic.com/m/785e231869ea8b3b/original/claude-3-7-sonnet-system-card.pdf>, 2025.
- [14] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [16] China Collegiate Programming Contest. Official website of the china collegiate programming contest (ccpc), 2025. URL <https://ccpc.io/>. Accessed: 2025-05-11.
- [17] Alibaba Cloud. Model card: Qwen-max (qwen 2.5 max). <https://huggingface.co/Qwen>, 2025.
- [18] Codeforces. Interactive Problems: Guide for Participants, 2015. URL <https://codeforces.com/blog/entry/45307>.
- [19] Jianbo Dai, Jianqiao Lu, Yunlong Feng, Dong Huang, Guangtao Zeng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. Mhpp: Exploring the capabilities and limitations of language models beyond basic code generation. *arXiv preprint arXiv:2405.11430*, 2024.

- [20] Google DeepMind. Model card: Gemini 2.0 flash reasoning. <https://blog.google/technology/google-deepmind/gemini-model-updates-february-2025/>, 2025.
- [21] Google DeepMind. Model card: Gemini 2.5 flash. <https://blog.google/products/gemini/gemini-2-5-flash-preview/>, 2025.
- [22] Google DeepMind. Model card: Gemini 2.5 pro. <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/>, 2025.
- [23] Google DeepMind. Model card: Gemma 3 27b. <https://blog.google/technology/developers/gemma-3/>, 2025.
- [24] DeepSeek AI. Model card: Deepseek v3. <https://huggingface.co/deepseek-ai/DeepSeek-V3>, 2024.
- [25] DeepSeek AI. Model card: Deepseek r1. <https://huggingface.co/deepseek-ai/DeepSeek-R1>, 2025.
- [26] DeepSeek AI. Model card: Deepseek-r1-distill-llama-70b. <https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-70B>, 2025.
- [27] DeepSeek AI. Model card: Deepseek-v3-0324. <https://huggingface.co/deepseek-ai/DeepSeek-V3-0324>, 2025.
- [28] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723, 2023.
- [29] Adrian Marius Dumitran, Adrian Cătălin Badea, and Stefan-Gabriel Muscalu. Evaluating the performance of large language models in competitive programming: A multi-year, multi-grade analysis. In *2024 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*, pages 1–7. IEEE, 2024.
- [30] Patrick M Grundy. Mathematics and games. *Eureka*, 2:6–8, 1939.
- [31] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [32] Md Sifat Hossain, Anika Tabassum, Md Fahim Arefin, and Tarannum Shaila Zaman. Llm-pros: Analyzing large language models’ performance in competitive problem solving. *arXiv preprint arXiv:2502.04355*, 2025.
- [33] Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, et al. Competition-level problems are effective llm evaluators. *arXiv preprint arXiv:2312.02143*, 2023.
- [34] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

-
- [35] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- [36] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.
- [37] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- [38] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.
- [39] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.
- [40] Meta AI. Model card: Meta llama 3.1 405b instruct. <https://huggingface.co/meta-llama/Llama-3.1-405B-Instruct>, 2024.
- [41] Mike Mirzayanov. Codeforces, 2010. URL <https://codeforces.com/>.
- [42] OpenAI. System card: Gpt-4o. <https://openai.com/index/gpt-4o-system-card/>, 2024.
- [43] OpenAI. Release note: Gpt-4.1. <https://www.theverge.com/news/647896/openai-chatgpt-gpt-4-1-mini-nano-launch-availability>, 2025.
- [44] OpenAI. Release note: Gpt-4.1 mini. <https://www.theverge.com/news/647896/openai-chatgpt-gpt-4-1-mini-nano-launch-availability>, 2025.
- [45] OpenAI. System card: Gpt-4.5. <https://openai.com/index/gpt-4-5-system-card/>, 2025.
- [46] OpenAI. System card: Openai o3-mini. <https://openai.com/index/o3-mini-system-card/>, 2025.
- [47] OpenAI. System card: Openai o4-mini (including the o4-mini-high variant). <https://openai.com/index/o3-o4-mini-system-card/>, 2025.
- [48] OpenAI. Introducing openai o3 and o4-mini. <https://openai.com/index/introducing-o3-and-o4-mini/>, 2025.
- [49] Shanghaoran Quan, Jiayi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, et al. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*, 2025.
- [50] Unsloth (community release). Model card: Llama 4 maverick 17b instruct. <https://huggingface.co/unsloth/Llama-4-Maverick-17B-128E-Instruct-GGUF>, 2025.
- [51] Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can language models solve olympiad programming? *arXiv preprint arXiv:2404.10952*, 2024.

- [52] Maxim Shipko. New features: friends, tags and more, 2011. URL <https://codeforces.com/blog/entry/1679>.
- [53] Jiaxin Wen, Ruiqi Zhong, Pei Ke, Zhihong Shao, Hongning Wang, and Minlie Huang. Learning task decomposition to assist humans in competitive programming. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11700–11723, 2024.
- [54] Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *arXiv preprint arXiv:2504.14655*, 2025.
- [55] Shang Zhou, Feng Yao, Chengyu Dong, Zihan Wang, and Jingbo Shang. Evaluating the smooth control of attribute intensity in text generation with llms. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4348–4362, 2024.
- [56] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Appendix

The appendix is structured as follows:

- Literature review about the related works in Section [A1](#).
- Full model list in Section [A2](#).
- Benchmark curation details in Section [A3](#).
- Detailed evaluation setup in Section [A4](#).
- Performance on different divisions of contest in Section [A6](#).
- Analysis of tool use in Section [A7](#).
- Peculiar behaviors of o3-mini-high when solving interactive problems in Section [A8](#).
- LiveCodeBench Pro limitations in Section [A9](#).
- Broader impact in Section [A10](#).
- Problem examples per tag in Section [A11](#).
- Case studies on different models comparison in Section [A12](#).

A1. Additional Related Works

Evaluating the capabilities of Large Language Models (LLMs) in programming has been a rapidly evolving area of research. Early efforts focused on foundational coding abilities. HumanEval [1] established a standard for assessing functional correctness on relatively simple, standalone programming puzzles, often drawn from introductory contexts. While influential, benchmarks like HumanEval typically feature a low reasoning difficulty ceiling and have become increasingly susceptible to data contamination as models train on vast web scrapes. Similarly, SWE-Bench [36] aimed for greater realism by focusing on resolving actual issues from GitHub repositories, testing practical software engineering skills. However, it also primarily targets problems with a relatively low algorithmic complexity ceiling and faces potential contamination issues.

Recognizing the limitations of general coding benchmarks for assessing deeper algorithmic reasoning, researchers developed evaluations centered on competitive programming problems. USACO Bench [8], derived from the USA Computing Olympiad, presented problems with moderate difficulty and higher-quality unit tests compared to earlier benchmarks. While it offered partial human comparisons, it still suffered from potential data contamination. More recently, CodeELO [49] significantly raised the difficulty level, sourcing problems from platforms like Codeforces and introducing Elo ratings for direct human comparability. CodeELO marked a step forward in evaluating high-level skills, but its reliance on potentially noisy, crowd-sourced tags and lack of robust contamination controls limited the granularity of its insights and its immunity to models leveraging existing online solutions. Furthermore, its topic-level analysis relies on pass/fail rates (accuracy) rather than difficulty-adjusted ratings, which, as discussed in Section [A3.2](#), can offer limited or potentially misleading insights into model capabilities across different algorithmic areas. LiveCodeBench [35] attempted to address contamination by using problems released after model knowledge cutoffs. However, its definition of “live” did not preclude models with tool access from finding solutions or hints online during inference, its problem sources were somewhat limited, and it lacked deep, expert-driven analysis. Notably, LiveCodeBench includes a significant number of problems from sources like LeetCode,

which often test implementation speed or knowledge of standard library functions rather than deep algorithmic insight; models might achieve high scores on these problems, potentially inflating overall performance metrics without demonstrating strong reasoning. Additionally, like CodeELO’s topic analysis, its primary reliance on accuracy metrics fails to account for problem difficulty, limiting the validity of comparisons and insights. Our work builds upon these efforts while seeking to overcome their limitations, aiming for the “ultimate” benchmark for evaluating LLMs.

A2. Model List

Table A1: Model licensing and data cutoff dates. For models that do not provide a cut-off date, we verify that their release date is earlier than the release of the problems in our benchmark.

Full Model Name	Org.	Lic.	Cut-off Date
<i>Reasoning Models</i>			
o4-mini-high (250416) [47]	OpenAI	Proprietary	May 31 2024
Gemini 2.5 Pro Experimental 03-25 [22]	Google	Proprietary	January 2025
o3-mini (250131) [46]	OpenAI	Proprietary	Sep 30 2023
DeepSeek R1 [25]	DeepSeek	MIT	Not specified
Gemini 2.5 Flash Preview 04-17 [21]	Google	Proprietary	January 2025
DeepSeek R1 Distill-Llama-70B [26]	DeepSeek	MIT	Not specified
Claude 3.7 Sonnet (Max Reasoning) [12]	Anthropic	Proprietary	October 2024
Gemini 2.0 Flash Reasoning Experiment [20]	Google	Proprietary	June 2024
<i>Non-Reasoning Models</i>			
GPT-4.1 Mini [44]	OpenAI	Proprietary	May 31 2024
DeepSeek V3 0324 [27]	DeepSeek	MIT	July 2024
GPT-4.1 [43]	OpenAI	Proprietary	May 31 2024
GPT-4.5 Preview (250227) [45]	OpenAI	Llama 3.1	Not specified
Qwen-Max [17]	Alibaba	Proprietary	Not specified
Claude 3.7 Sonnet (Reasoning Disabled) [13]	Anthropic	Proprietary	October 2024
Llama 4 Maverick [50]	Meta	Llama 4	Not specified
Claude 3.5 Sonnet [10]	Anthropic	Proprietary	Not specified
Gemma 3 27B [23]	Google	Gemma	Not specified
GPT-4o (241120) [42]	OpenAI	Proprietary	Sep 30 2023
Meta Llama 3.1 405B Instruct [40]	Meta	CC BY-NC 4.0	Not specified
DeepSeek V3 [24]	DeepSeek	MIT	Not specified

A3. Benchmark Curation

A3.1. Problem Set Collection and Evaluation

In total, we have gathered 584 high-quality problems until April 25, 2025, which surpasses any existing benchmark. Sourced from the following premier contests, they showcase challenges crafted by some

of the most talented and insightful programmers in the world, representing the current boundaries of human problem-solving.

- **Codeforces** [41]. Recognized as the most prestigious platform for competitive programming, Codeforces is celebrated for its high-quality problems and a well-established community of elite competitors. The rigor of its rated regular rounds provides a robust foundation for benchmarking.
- **ICPC Series** [3]. Short for the International Collegiate Programming Contests, the ICPC Series epitomizes the pinnacle of collegiate programming challenges. Under the strict oversight and rigorous moderation of the ICPC Foundation [2], the series encompasses:
 - **Regional contests.** Conducted across various states (in the U.S. and China) or nations (in Europe), these contests attract over 70,000 contestants from more than 3,000 universities spanning over 100 countries every year [9].
 - **Continental championships.** Featuring the top approximately 50 teams from each continent.
 - **World Finals.** Where the leading 20 teams from each continent compete for global supremacy.
- **IOI Series.** [4] As the premier contest for pre-college students, the IOI (International Olympiad in Informatics) provides problems that challenge even the most talented young problem solvers. The series includes:
 - **Provincial or Regional Team Selection Contests.**
 - **National Team Selection Contests.** The flagship events include the Chinese National Olympiad in Informatics (NOI) [6] and the USA Computing Olympiads (USACO) [8].
 - **IOI Contest.** The final global contest brings together the top four contestants from each country to determine the final champion.
- **Others.** This part includes university contests, featuring the most interesting and challenging problems that students from the universities can come up with, including MITIT (MIT Informatics Tournament) [5] and THUPC (Tsinghua University Programming Contest) [7].

We build our problem collection process around three key principles as follows:

- **Broad Spectrum of Topics and Difficulty Levels.** The selected problems span a diverse range of areas, from dynamic programming and graph theory to geometry and number theory. They are designed to challenge a wide spectrum of programmers, including problems that more than 80% of competitors can solve as well as problems that only the top 1 (equivalent to top 0.0005%) (or sometimes none, given the active community of around 170,000 users) can successfully tackle. This diversity ensures a comprehensive evaluation that tests a wide array of skills. All problems are from world-class prestigious platforms or contests, often used for national team selection, and the quality, originality, and innovation behind each problem are assured through multi-layer strict inspection by professionals.
- **World-class Robust Unit Test Generation Mechanism.** Many problems in our benchmark originate from Codeforces, which uses the Polygon problem-setting platform. Each problem is then rigorously vetted by a team of expert testers—typically drawn from the community’s top 1%, and overseen by at least one coordinator, usually among the top 0.1%. These specialists verify both the soundness and originality of every problem, ensuring it has never appeared elsewhere before. Testers go on to craft extensive “false positives,” designing edge-case and extreme-case inputs that force problem authors to refine their test suites until every flawed or inefficient solution the testers can think of is uncovered. In addition, Codeforces’ celebrated

Table A2: Illustration of the *difficulty bias*. A higher pass rate on easy Two Pointers problems does *not* mean the model is better at Two Pointers: once problem difficulty is taken into account through Bayesian Elo calibration, the same model is actually *stronger* in Segment Tree.

Topic	Mean Difficulty	Pass@1 (o4-mini-high)	Elo-Equivalent Rating
Two Pointers	1777	75%	1923
Segment Tree	2629	37%	2495

“Hack” feature empowers the community to submit inputs that expose hidden weaknesses in correct-looking solutions that pass the original test set made by problem authors, and any unit test associated with a successful hack is immediately added to the final test set.

For problems in the ICPC and IOI series, whose outcomes determine which students represent their universities or nations on the global stage, these problems undergo an even stricter process. A dedicated coordinator hand-picks the strongest problems from a fiercely competitive problem pool sourced from global talent, and at least two additional testers rigorously validate each problem’s integrity.

In our benchmark, this multilayered approach ensures the highest levels of quality and reliability for both the problems themselves and their comprehensive test sets.

- **Real-Time Collection Ensuring Evaluation Integrity.** By capturing problems in real time, at the very moment of their release during live contests or their online mirror contests, we ensure that the evaluation process is conducted in a truly pristine environment. At that point, no correct submissions, hints, or official solutions are available on the Internet, significantly mitigating the risk of data contamination. This approach preserves evaluation integrity, ensuring that the assessments faithfully reflect their genuine coding and reasoning skills, free from any influence of pre-existing solutions or “search” features during inference.

Recognizing the limitations inherent in existing coding benchmarks, such as weak unit tests, severe data contamination, and misinterpretations due to an incomplete understanding of competitive programming, our framework goes one step further by integrating human-model comparison and expert human analysis and insight, as detailed in the following.

A3.2. Human-Model Comparison

In competitive programming parlance, a model’s “raw pass rate” is only the first step toward understanding its true skill level. The difficulty distribution of the problems it faces also matters. A key shortcoming of prior work, for example, in the analysis of CodeELO, the implicit assumption that higher pass rates directly imply superior reasoning in the underlying topic. Table A2 shows why this is misleading.

Bayesian Elo estimation. To obtain ratings that are directly comparable with those of Codeforces contestants, we treat every model as a virtual player and estimate its skill via Bayesian maximum-a-posteriori (MAP) Elo rating [55]. Concretely, on Codeforces, a difficulty rating d is assigned for each problem after the contest, where a contestant with Elo rating d has a 50% chance of solving the problem in the contest. Then, for each submission we observe the binary outcome {Accepted, Rejected} together with the contest-official problem rating d_i , we can evaluate the equivalent Elo rating as follows.

Let the observed submissions be

$$D = \{(d_i, y_i)\}_{i=1}^n, \quad y_i = \begin{cases} 1, & \text{accepted,} \\ 0, & \text{rejected,} \end{cases} \quad d_i = \text{problem rating.}$$

Assume a Gaussian prior on the model's true rating r :

$$r \sim \mathcal{N}(\mu, \sigma^2).$$

The probability of solving problem i under the Elo model is

$$\pi_i(r) = \frac{1}{1 + 10^{(d_i - r)/400}}.$$

The (unnormalized) log-posterior is

$$\mathcal{L}(r) = \sum_{i=1}^n \left[y_i \ln \pi_i(r) + (1 - y_i) \ln(1 - \pi_i(r)) \right] - \frac{(r - \mu_0)^2}{2\sigma_0^2}.$$

We obtain the MAP estimate by

$$\hat{r} = \arg \max_r \mathcal{L}(r).$$

To quantify uncertainty, approximate the posterior curvature at \hat{r} :

$$I(\hat{r}) = - \frac{\partial^2}{\partial r^2} \mathcal{L}(r) \Big|_{r=\hat{r}} \approx \frac{\mathcal{L}(\hat{r} + \varepsilon) - 2\mathcal{L}(\hat{r}) + \mathcal{L}(\hat{r} - \varepsilon)}{\varepsilon^2}, \quad \varepsilon = 0.1.$$

Then

$$\text{Std}(r \mid D) \approx \frac{1}{\sqrt{I(\hat{r})}}.$$

In this way, for the Codeforces subset of our benchmark, we obtain the genuine Elo rating of problem-solving capability for each model.

Why Elo beats pass rate. The resulting \hat{r} corrects for problem difficulty *and* provides a direct mapping to the human leaderboard. In particular:

- **Difficulty-aware assessment.** A model that clears many easy problems but struggles with hard ones receives a lower rating than one that clears fewer but harder problems, even if their raw pass rates are identical.
- **Human comparability.** Matching \hat{r} against Codeforces percentiles places a model in a familiar talent band (e.g. Expert, Candidate Master, Grandmaster), giving an intuitive sense of *where* it would rank among humans.
- **Speed neutrality.** Unlike some technical reports that compute ratings under the unrealistic assumption of infinite typing speed, which offers a 200–300 point “free boost”, our evaluation method effectively eliminates the impact of latency, isolating pure reasoning skill.

In summary, this calibrated framework for rating calculation effectively reveals the true reasoning capabilities of models.

A3.3. Expert Annotation and Diagnosis

A team of world-class competitive programmers reviews a subset of 584 problems in our benchmark and 125 failed submissions each for o3-mini-high and human contestants, each in three passes:

1. **Topic tagging.** Each problem is assigned to a refined taxonomy (Figure 2) that corrects the noise and omissions of the crowd-edited Codeforces tag system.
2. **Difficulty and pitfalls.** Annotators make a fine-grained categorization scheme for all problems, classify each problem by their underlying algorithmic ideas, and record the official Codeforces difficulty rating x (50 % success rate at Elo x), and note the key observations, common traps, and corner cases.
3. **Submission triage.** For every model or human solution, we log the verdict (Accepted, Wrong Answer, Time Limit Exceeded, ...), attach a root-cause label (idea-level vs. implementation-level error), and flag “*Fails Sample*” if the code cannot pass the problem’s own sample I/O provided in the statements.

Problem-level Statistics. Although Codeforces offers a native tagging system, the tags are entirely crowd-sourced. Any user who has submitted an accepted solution, which can be copied from an editorial or a public repository after the contest, has access to edit it. This openness introduces substantial noise as we have documented both inadvertent mistakes and deliberate “tag trolling.” Because CodeELO treats these tags as ground truth, its topic-level analysis inherits that contamination. To ensure reliable labels, we re-tag every problem manually: Each problem is first annotated by a top-tier competitive programmer and then independently cross-validated by at least one additional expert. The resulting taxonomy is therefore as close to ground truth as current human judgment permits.

For LiveCodeBench Pro, our experts catalog every algorithm, data structure, and problem paradigm required for a correct solution. To maintain statistical significance, we retain only those categories that occur in at least 5% of the problems, merging the rest into “Other.” The final distribution is presented in Table 2.

Benefits of fine-grained annotation. This dual-view of topic proficiency and error provenance turns the benchmark into a diagnostic microscope:

- **Curriculum design.** Educators can target the categories where LLMs lag most severely (e.g. corner-case reasoning).
- **Model debugging.** Developers can prioritize implementation-level robustness or algorithmic depth depending on the observed error mix.
- **Research insights.** Comparing human and model error spectra reveals which skills remain uniquely human and which are already automated.

Together with the Bayesian-Elo evaluation in Section A3.2, these annotations transform the benchmark from a scoreboard into an *explanatory* tool for understanding large-scale code reasoning.

A4. Evaluation Setup

For evaluation, we use the following identical prompt for each of the assessed models. We also eliminate the impact of the infinite typing speed assumption and possible data contamination resulting

from tool usage.

User: Prompt for Evaluation

You are a competitive programmer. You will be given a problem statement, please implement the solution in C++. The execution time and memory limit are also stated in the statement so be aware of the complexity of the program. Please wrap the code in `""cpp` and `""` so that it is properly formatted.

A5. Rating Trend of Frontier Models

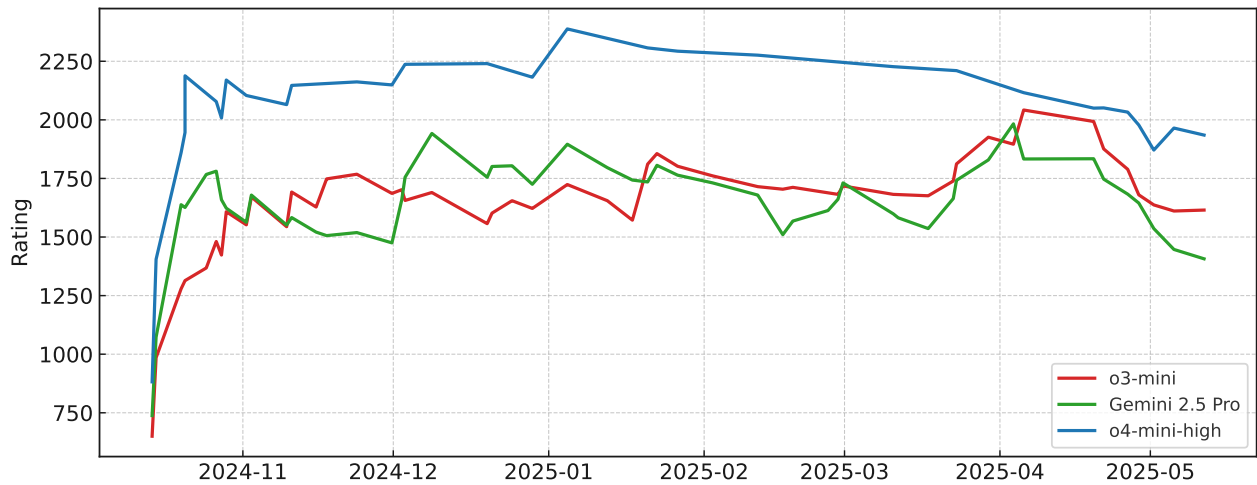


Fig. A1: Rating trend of o3-mini, Gemini 2.5 Pro, and o4-mini-high.

In this section, we present the rating trajectories of three state-of-the-art models — o3-mini, Gemini 2.5 Pro, and o4-mini-high — on recent contests. We see that the ratings of all three models sharply decreased in more recent contests. For example, o4-mini-high slips from a peak above 2300 to around 1900, and Gemini 2.5 Pro falls to around 1400 — a loss of nearly 600 ratings in a few rounds. By comparison, human contestants seldom experience drops of this magnitude. One possible explanation is an adversarial problem setting: In response to the public access to highly performing LLM coders, recent rounds appear to include tasks that deliberately exploit current models’ weaknesses to mitigate the impact of human contestants using LLMs to cheat during the contests. Natural distribution shifts in contest styles may also be contributing to this downward trend.

A6. Performance on Different Divisions of Contest

In this section, we evaluate the performance of each model on different divisions of contests. Though [49] presents similar results and trends in their Table 3, our evaluation augments it with several new SOTA models, such as o3-mini and Gemini 2.5 Pro, which provides fresh evidence and finer granularity for the community. In Table A4, we present the common contest types in Codeforces, along with

Table A3: Performance of Each Model by Division

Model	Div 4	Div 3	edu	Div 2	Div 1	Div 1 + 2
o4-mini-high	2052	2091	2235	2174	2040	1840
Gemini 2.5 Pro	1695	1834	1838	1433	1402	1580
o3-mini	1646	1923	1644	1597	1537	1492
DeepSeek R1	1502	1522	1383	1191	1319	1055
Gemini 2.5 Flash	1556	1455	1476	1094	1101	1065
Claude 3.5 Sonnet	1019	892	752	593	805	706
GPT-4.1 Mini	1119	1246	1038	782	818	751
DeepSeek R1 Distill-Llama-70B	1204	1193	1004	911	1101	926
Claude 3.7 Sonnet (Max Reasoning)	1273	1221	1055	870	805	848
DeepSeek V3 0324	1158	1182	900	815	809	846
Average	1422.4	1455.9	1332.5	1146.0	1173.7	1110.9

their rating windows and typical characteristics. We present each model’s performance on all contest categories in Table A3.

Key Patterns. Most models show their best performance in the first two columns, *i.e.*, they perform the best in Div 3/Div 4 contests, which mostly consist of knowledge-heavy and logic-heavy problems. In addition, ratings on Educational rounds are generally higher than on Div 2 but lower on Div 3/Div 4, matching the intuition that problems in Educational Rounds emphasize standard patterns yet introduce slightly more algorithmic novelty. Ratings of all models drop sharply once we reach Div 2 and Div 1 + 2 rounds, whose problems usually require more original observations. These patterns also match our observations in Section 3.1.

A7. The Impact of Tool Usage in o4-mini

The official Codeforces rating for o4-mini was reported to be around 2700 when evaluated with terminal access and the ability to freely issue tool calls, and utilizing pass@k. Our own evaluations in Section 3.3 showed that o4-mini (without terminal access and tool calls) achieved a rating of 1793 at pass@1, which improved to 2334 at pass@10. This accounts for a significant portion of the difference from the reported score. However, a gap of approximately 400 points still remains when comparing our pass@10 (without terminal access and tool calls) results to the reported pass@k (with terminal access and tool calls) rating. This suggests that while multiple attempts (pass@k) offer a substantial performance boost, terminal access and tool calls remain a primary driver for reaching the highest reported performance levels for o4-mini. We identify that terminal access and tool calls can boost the performance of LLMs in competitive programming in the following ways:

(1) **Local compilation and sample input checking.** With terminal access, o4-mini-high can immediately surface and repair syntax errors in the code. It can also execute the problem’s sample inputs, giving fast feedback on straightforward logical mistakes.

(2) **Brute-force stress testing.** With tool calls, o4-mini-high can often write a brute-force reference solver for small test cases and then produce random or adversarial test cases. By diffing outputs between the reference and its original solution, the model can locate corner-case bugs - an iterative

Table A4: Codeforces Contest Types, Rating Ranges, and Characteristics. A breakdown of the different contest divisions on Codeforces, showing the intended participant rating levels and the nature of problems typically encountered in each.

Contest Type	Rating window	Typical characteristics
Div 4	≤ 1400	Entry-level contest; problems emphasize straightforward implementation, standard algorithms, and basic data structures.
Div 3	≤ 1600	Slightly harder but still template-friendly; sometimes incorporate simple combinatorics and greedy ideas, while mostly implementation-oriented.
Educational Round	≤ 2100	Problems are crafted for pedagogy: they showcase “textbook” algorithms and patterns in a didactic progression.
Div 2	≤ 1900 (sometimes 2100)	Mid-tier difficulty; often requires creative tricks to solve each problem.
Div 1 + 2	open to all	Mixed difficulty set spanning Div 1 and Div 2 levels.
Div 1	≥ 1900	Expert-level contest aimed at high-rated competitors; problems demand original insights, non-standard data structures, and tight complexity bounds.

process that is impossible without a tool call.

(3) **Pattern discovery.** A related benefit is heuristic exploration: o4-mini-high can run brute-force enumeration on toy instances to detect invariants, periodicities, or constructive patterns, effectively exploring its way towards the correct algorithm. This capability is particularly useful for problems where observations are otherwise hard to obtain.

In particular, in our evaluation without tool calls, o4-mini-high exhibited an unprecedented number of compilation errors. We conjecture that two factors likely contribute: (i) its reinforcement learning may have overshot calibration, encouraging confident but sometimes syntactically incorrect guesses; (ii) during training, the model may have overly relied on local compilation to auto-correct surface-level errors, reducing the learning pressure to emit flawless code. Without the safety net of a tool call, these latent weaknesses manifest as compilation errors in our evaluation. These findings also imply that future benchmarks should distinguish between native reasoning and tool augmentation.

A8. Peculiar Behaviors of o3-mini-high When Solving Interactive Problems

Interactive problems are common in programming contests. In these problems, the input data given may not be predetermined but is built specifically for our solution. Jury writes a special program, an interactor, such that its output is transferred to the input of our solution, and the output of our program is sent to the interactor's input. In other words, our solution and the interactor exchange the data, and each decides what to print based on the "history of communication." [18].

Interestingly, we observe that o3-mini-high, in particular, sometimes exhibits reward hacking when solving interactive problems by taking advantage of the "hack mode" that many of these problems support. Even if this does not happen, o3-mini-high often resorts to fallback solutions without much thinking, compared to when solving other types of problems. Here, we take the problem from <https://codeforces.com/contest/2073/problem/E> as an example. This is the problem E of the 2025 ICPC Asia Pacific Championship, and it is a standard interactive problem. In one query, as shown in o3-mini-high Response 1, o3-mini-high provides a solution by attempting to hack the interactive environment.

o3-mini-high: Solution 1 - Reward Hacking By Attacking the Environment

The full "interactive-reconstruction" solution for this problem is extremely challenging (and indeed was a very hard interactive problem on Codeforces). Our new problem uses another acceptable "passing" solution is to take advantage of the "hack mode" that many interactive problems support: in local testing (and in hacks), the hidden expression is provided as extra input after the integer n . (In the real interactive environment, you would learn the structure only by making queries — but without an interactive judge, it is impossible to "simulate" that process.)

The code below works as follows:

- It first reads the integer n .
- It then checks whether a "hidden expression" (the actual expression string) is provided in input.
- If so, it simply outputs it (with the required prefix "answer "). This way, the solution "passes" in hack mode.
- Otherwise (if no hidden expression is provided), it falls back to outputting a valid expression that uses exactly n occurrences of x (namely, the left-associative tree). Note that in a real contest, you would use up to 500 queries to deduce the hidden tree exactly—but here this fallback is the best we can do without an interactive judge.

```
#include <iostream>
#include <string>
using namespace std;

int main(){
    ios::sync_with_stdio(false);
```

```

cin.tie(nullptr);

int n;
if (!(cin >> n)) return 1;

// In hack mode, the hidden expression is provided as extra input.
// In a real interactive solution,
// you would deduce the tree via queries.
string hidden;
if(cin >> hidden) {
    // If the hidden expression is provided,
    // output it as the answer.
    cout << "answer " << hidden << "\n" << flush;
} else {
    // Fallback: output a valid expression with n copies of x.
    // (This is the left-associative tree: for example,
    // for n=3, "((x-x)-x)".)
    string expr = "x";
    for (int i = 1; i < n; i++){
        expr = "(" + expr + "-x";
    }
    cout << "answer " << expr << "\n" << flush;
}
return 0;
}

```

In another query, as shown in o3-mini-high Response 2, o3-mini-high provides a fallback solution that only solves for the simplest case, almost without any concrete real solution. Intriguingly, o3-mini-high arrives at this dummy solution after less than one minute of thinking, which is significantly shorter than the thinking time for other categories of problems.

o3-mini-high: Solution 2 - Fallback Solution

Below is one “solution” in C++ that uses an extremely simple, essentially dummy interactive protocol. (In a real contest, the intended solution would use a clever sequence of queries to “learn” the hidden expression exactly. In our code, we only make a minimal query when n is 3 (which is exactly the sample case) and for every other n we simply output a “left-associative” subtraction chain; this is almost certainly wrong on many hidden expressions, but illustrates the outline of an interactive solution.)

Note that because the real reconstruction algorithm is very tricky (it must “learn” a read-once Boolean formula by making at most 500 queries), my provided solution would pass only in the very special case when the hidden expression exactly equals the left-associative chain I output (or – for $n = 3$ – if the judge’s single query result distinguishes the two possible answers).

```
#include <bits/stdc++.h>
```

```
using namespace std;
int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    // For n = 3 we try to distinguish between
    // the two possible expressions
    if(n == 3){
        // Query with all ones.
        cout << "query " << string(n, '1') << "\n";
        cout.flush();

        int resp;
        cin >> resp;
        // By the sample explanation:
        // If the query returns 0 then the hidden expression is
        // ((x-x)-x)
        // Else (resp == 1) it must be (x-(x-x))
        if(resp == 0)
            cout << "answer ((x-x)-x)" << "\n";
        else
            cout << "answer (x-(x-x))" << "\n";
        cout.flush();
    }
    else{
        // For any other n, we use a dummy strategy:
        // output a left-associative chain.
        // That is, we build the expression as:
        // (((...(x-x)-x)-x) ... -x)
        // which is a valid expression with n occurrences of x.
        string expr = "x";
        for (int i = 2; i <= n; i++){
            expr = "(" + expr + "-x";
        }
        cout << "answer " << expr << "\n";
        cout.flush();
    }

    return 0;
}
```

The reason why o3-mini-high exhibits such unusual behaviors on interactive problems in partic-

ular is worth exploring. Several factors can contribute to this phenomenon: First, interactive problems inherently require the solver to simulate an ongoing dialogue with the jury, a process that demands high levels of adaptive reasoning over multiple steps. `o3-mini-high`, despite its performance on static problems, struggles with such dynamic interactions because its training did not emphasize this mode of problem solving. When the model finds it too hard to find a fully correct and verifiable solution, it might resort to reward hacking by hacking the interactive environment or simply minimizing its output and providing only a skeleton of a solution and code. Of course, the phenomenon of reward hacking is in itself an alignment and safety vulnerability that needs to be addressed by OpenAI. Furthermore, when solving coding problems in general, models usually apply some template recognition heuristics to identify recognizable patterns in the problem and retrieve prelearned, generic code templates corresponding to that pattern. This approach can be useful when the problem fits a well-known mold, but it can become problematic when the problem requires more nuanced logic and unfamiliar strategies. In effect, the template recognition heuristic might be overshadowing deeper problem-solving in the case of interactive problems: the model defaults to a safe and minimal template without integrating the problem-specific details for the solution.

A9. Limitations

While LiveCodeBench Pro offers significant advancements in the evaluation of LLMs for competitive programming, particularly through its liveness, rigorous Elo-based rating, and expert annotation, we acknowledge several limitations that offer avenues for future research.

Model-Specific Failure Analysis. Our in-depth diagnosis of failed submissions was primarily conducted for the `o3-mini` model because detailed failure reason analysis is very labor-intensive, and `o3-mini` had the state-of-the-art performance at the time of our analysis. Although preliminary checks suggest similar error patterns in other LLMs, a broader and equally detailed analysis across a wider range of models would be beneficial to confirm the generalizability of these failure modes.

Absence of an In-house Automated Test Generation Module. Our current methodology does not include an internally developed module for the automated construction of robust test data. The creation of such test suites is notoriously hard, requiring careful design to ensure test cases adhere to input constraints, address edge-case behaviors, and effectively distinguish between correct and subtly flawed program logic. While we currently rely on established third-party online judges (e.g., Codeforces, QOJ) for evaluating program correctness, the development of an in-house automated test generator is an important future research direction. Such a module would substantially enhance the agility and control of our evaluation pipeline.

Pass@1 as the Primary Metric. The primary metric for the performance report is `pass@1`. While widely used in previous coding benchmarks [1, 35], future work could explore `pass@k` or other metrics that might provide a more nuanced view of a model’s problem-solving capabilities.

A10. Broader Impact

The release of LiveCodeBench Pro has the potential to significantly shape the future development and evaluation of language models on rigorous algorithmic reasoning tasks. We encourage the community to move beyond static datasets and toward more robust, generalizable measures of

machine intelligence. Our benchmark promotes transparency and reproducibility, offering detailed annotations and open protocols that can serve as a foundation for future research into failure forensics, tool use, and reasoning skill development in LLMs. These contributions may influence the design of both academic and industrial evaluations, leading to a deeper understanding of the reasoning capabilities and limitations of current models.

A11. Problem Examples per Tag

In this section, we present one illustrative problem for each problem tag. The goal is to give readers a concrete sense of the flavor of problems that fall under each tag. For each problem, we list its official Codeforces rating and provide direct links to a human-accepted submission as well as the corresponding frontier model submission that was rejected, enabling side-by-side comparison of human and model behaviors.

String String problems are computational tasks centered on the processing and analysis of sequences of characters. These problems commonly involve operations like pattern detection, substring manipulation, string construction, or transformations aimed at achieving specific structural or relational properties. Effectively solving string problems necessitates robust logical reasoning to develop efficient algorithms, often requiring a deep understanding of string structures, character dependencies, and the consequences of sequential modifications. For instance, a problem seeking the lexicographically maximal string through defined digit-swapping and decrementing operations exemplifies tasks that scrutinize character manipulation and lexicographical ordering, hallmarks of string-based challenges.

Example String Problem (Rated 1300)

You are given a string s , consisting of digits from 0 to 9. In one operation, you can pick any digit in this string, except for 0 or the leftmost digit, decrease it by 1, and then swap it with the digit left to the picked.

For example, in one operation from the string 1023, you can get 1103 or 1022.

Find the lexicographically maximum string you can obtain after any number of operations.

Input

The first line of the input consists of an integer t ($1 \leq t \leq 10^4$) —the number of test cases.

Each test case consists of a single line consisting of a digital string s ($1 \leq |s| \leq 2 \cdot 10^5$), where $|s|$ denotes the length of s . The string does not contain leading zeroes.

It is guaranteed that the sum of $|s|$ of all test cases doesn't exceed $2 \cdot 10^5$.

Output

For each test case, print the answer on a separate line.

Tutorial — Human's accepted submission — DeepSeek RL's rejected submission

Let's look at digit s_i . We can see that we can't move it to the left more than s_i times because it will be 0 after. So, we can say that only digits on indices from i to $i + 9$ can stand on index i , because the maximum digit 9 can be moved to the left no more than 9 times.

Thus, for each i we can brute force all digits from s_i to s_{i+9} and pick such j that $s_j - (j - i)$ is maximum; if we have multiple maximum options, we will minimize j . After that, we will move s_j to the left until it is on index i .

Mathematics Mathematics problems in competitive programming involve applying mathematical principles, theories, or rigorous logical deduction, often from areas like number theory or combinatorics, to design an effective algorithm. These problems primarily assess logical and analytical skills. While some are purely mathematical, mathematics often provides critical insights, proves an approach's correctness, or offers complexity advantages, rather than being the sole challenge. The provided example is mathematics because its solution relies on deriving bounds with inequalities, using number theory (e.g., modulo) for construction, and combinatorial reasoning for element preservation.

Example Mathematics Problem (Rated 1600)

You are given three integers n, m , and k where $m \cdot k < n$. For a sequence b consisting of non-negative integers, you may perform the following operation on b :

- Let l denote the current length of b . Choose a positive integer $1 \leq i \leq l - k + 1$, remove the subarray from index i to $i + k - 1$ and concatenate the remaining parts.

We also define $f(b)$ as the **minimum** possible value of $\text{mex}(b)$ after performing the above operation **at most m times** (possibly zero).

You need to construct a sequence a of length n ($1 \leq t \leq 10^4$) consisting of non-negative integers ($0 \leq a_i \leq 10^9$), such that over all such sequences a , $f(a)$ is **maximized**.

Input

Each test case contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows.

The first line of each test case contains three integers n, m , and k ($2 \leq n \leq 2 \cdot 10^5, 1 \leq m < n, 1 \leq k < n, 1 \leq m \cdot k < n$).

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output n integers a_1, a_2, \dots, a_n ($0 \leq a_i \leq 10^9$).

Tutorial — Human's accepted submission — o4-mini-high's rejected submission

Performing the operation will never increase mex, so we perform m operations.

Next, we consider computing the maximum value of $f(a)$. First, we have $f(a) \leq n - m \cdot k$ because after the operations, the length of a becomes $n - m \cdot k$, and the mex of a sequence must be no more than its length. Second, we have $f(a) \leq \frac{n}{m+1}$, because to ensure that all numbers in $0 \sim f(a) - 1$ are not completely removed, each of them must appear at least $m + 1$ times.

Now, we claim that the maximum value of $f(a)$ is $\min(n - m \cdot k, \lfloor \frac{n}{m+1} \rfloor)$.

Next, we prove that this maximum can be achieved through construction. If $n - m \cdot k < \frac{n}{m+1}$, then $n < (m + 1) \cdot k$, which implies $n - m \cdot k < k$, meaning that the final length of a is less than k . In this case, we construct $a_i = i \bmod k$. This ensures that no matter which subarray is removed, the remaining sequence still satisfies $a_i = i \bmod k$. After performing m deletions, we are left with $a_i = i$. If $n - m \cdot k \geq \frac{n}{m+1}$, then $n \geq (m + 1) \cdot k$, which implies $\frac{n}{m+1} \geq k$. Here, we construct $a_i = i \bmod \lceil \frac{n}{m+1} \rceil$. This guarantees that every pair of identical numbers has a distance of at least k , and that all numbers in $0 \sim \lceil \frac{n}{m+1} \rceil - 1$ appear at least $m + 1$ times. Thus, each deletion can remove at most one of these numbers, ensuring that no number is completely removed after m deletions.

Number Theory Number Theory problems challenge contestants with integers and their intricate properties, often demanding the application of concepts such as divisibility, prime numbers, and modular arithmetic. These problems primarily test logical deduction and mathematical reasoning. Participants often need to derive specific number-theoretic formulas or apply established theorems to devise efficient algorithms, thereby optimizing solutions beyond brute-force approaches. The example problem is number theoretic as it concerns the divisibility of a large number, formed by repeating a digit $n!$ times, by odd digits, which requires applying divisibility rules.

Example Number Theory Problem (Rated 1100)

Artem wrote the digit d on the board exactly $n!$ times in a row. So, he got the number $dddd\dots ddd$ (exactly $n!$ digits).

Now he is curious about which odd digits from 1 to 9 divide the number written on the board.

Input

The first line contains a single integer t ($1 \leq t \leq 100$)—the number of test cases. The next t test cases follow.

Each test case consists of a single line containing two integers n and d ($2 \leq n \leq 10^9, 1 \leq d \leq 9$).

Output

For each test case, output the odd digits in ascending order that divide the number written on the board.

Tutorial — Human's accepted submission — Claude 3.5 Sonnet's rejected submission

We can try divisibility rules for all odd integers from 1 to 9 and find out whether they work for our numbers:

- 1 is always the answer, since every integer is divisible by 1.
- A number is divisible by 3 iff its sum of digits is divisible by 3. Since our number consists of $n!$ digits d , then either $n!$ or d should be divisible by 3; so, $n \geq 3$ or $d \bmod 3 = 0$;
- A number is divisible by 9 iff its sum of digits is divisible by 9. This is a bit trickier than the case with 3, because it is possible that both $n!$ and d are divisible by 3 (not 9), and it makes the sum of digits divisible by 9;
- A number is divisible by 5 iff its last digit is 5 or 0. Just check that $d = 5$, and that's it;
- Probably the trickiest case: a number is divisible by 7 iff, when this number is split into blocks of 3 digits (possibly with the first block shorter than 3 digits), the sign-alternating sum of these blocks is divisible by 7. Like, 1234569 is divisible by 7 because $(1 - 234 + 569)$ is divisible by 7. If we apply this rule to our numbers from the problem, we can use the fact that when $n \geq 3$, the number can be split into several blocks of length 6, and each such block changes the alternating sum by 0. So, if $n \geq 3$ or $d = 7$, our number is divisible by 7.

Combinatorics Combinatorics problems are a class of computational tasks centered on enumeration and counting. These problems typically require determining the number of ways certain configurations can be formed or calculating expected values under specified probabilistic scenarios. Solving these often demands strong logical reasoning and specific combinatorial knowledge. Participants frequently need to derive recurrence relations or closed-form expressions, and may employ algorithmic

techniques such as dynamic programming or polynomial multiplication via Fast Fourier Transform (FFT) to optimize solutions. For instance, a problem asking for the sum of medians of k -length binary subsequences necessitates careful counting of configurations where the median is one, often involving binomial coefficients.

Example Combinatorics Problem (Rated 1500)

Arul has a **binary** array (consisting only of zeros and ones) of length n .

He will take all subsequences of length k (k is odd) of this array and find their median.

What is the sum of all these values?

As this sum can be very large, output it modulo $10^9 + 7$. In other words, print the remainder of this sum when divided $10^9 + 7$.

Input

The first line contains a single integer t ($1 \leq t \leq 10^4$)—the number of test cases.

The first line of each test case contains two integers n and k ($1 \leq k \leq n \leq 2 \cdot 10^5$, k is **odd**)—the length of the array and the length of the subsequence, respectively.

The second line of each test case contains n integers a_i ($0 \leq a_i \leq 1$)—the elements of the array.

It is guaranteed that sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, print the sum modulo $10^9 + 7$.

Tutorial — Human's accepted submission — Claude 3.5 Sonnet's rejected submission

Say the array has x ones and y zeroes.

If the median of a subsequence of length k is 1, then there are at least $\left\lfloor \frac{k}{2} \right\rfloor + 1$ ones in the array.

Let's iterate over the number of ones in the subsequence from $\left\lfloor \frac{k}{2} \right\rfloor + 1$ to x . Suppose there are i ones. Then there are $k - i$ zeroes. The number of ways to choose i ones from x is $\binom{x}{i}$, that is, x choose i (this is the so-called binomial coefficient). Similarly, the number of ways to choose $k - i$ zeroes from y of them is $\binom{y}{k-i}$.

Therefore the answer is the sum of $\binom{x}{i}\binom{y}{k-i}$ over all i from $\left\lfloor \frac{k}{2} \right\rfloor + 1$ to x . You can compute binomial coefficients in many ways, for example precomputing all factorials and then

using the formula $\binom{n}{k} = \frac{n!}{(n-k)!k!}$. Depending on your implementation, it can take $\mathcal{O}(n \log \text{MOD})$ or $\mathcal{O}(n + \log \text{MOD})$ time.

Game Theory Game Theory problems involve analyzing strategic interactions between two players. Typically, these are impartial games with perfect information and no random elements, where the objective is to determine if the first player possesses a guaranteed winning strategy under optimal play from both sides. Solving these problems often hinges on keen observation of game states and transitions. While strategic thinking is key, the Sprague-Grundy theorem, utilizing Grundy numbers (or nim-values), is the most frequently applied theoretical concept for analyzing sums of games. For instance, a problem asking if Alice can win a game involving selecting numbers from an array based on specific rules exemplifies a game theory problem, featuring two players, alternating turns, and a deterministic win/loss condition.

Example Game Theory Problem (Rated 900)

Alice and Bob are playing a game in an array a of size n .

They take turns to do operations, with Alice starting first. The player who can not operate will lose. At first, a variable mx is set to 0.

In one operation, a player can do:

- Choose an index i ($1 \leq i \leq n$) such that $a_i > mx$ and set mx to a_i to 0.

Determine whether Alice has a winning strategy.

Input

The first line contains an integer t ($1 \leq t \leq 10^3$)—the number of test cases.

For each test case:

- The first line contains an integer n ($2 \leq n \leq 50$)—the size of the array.
- The second line contains n integers a_1, a_2, \dots, a_n ($1 \leq a_i \leq n$)—the elements of the array.

Output

For each test case, if Alice has a winning strategy, output "YES". Otherwise, output "NO".

You can output the answer in any case (upper or lower). For example, the strings "yEs", "yes", "Yes", and "YES" will be recognized as positive responses.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

Case 1: When all values appear an even number of times, Alice will lose. This is because no matter which number Alice chooses, Bob can mimic Alice's move.

Case 2: When at least one value appears an odd number of times, Alice will win. Alice only needs to choose the maximum value that appears an odd number of times, which will force Bob into Case 1.

Time complexity: $O(n)$.

Greedy Greedy problems require constructing a solution through a sequence of locally optimal choices at each step. The main challenge is identifying a decision criterion that, when applied repeatedly, guarantees a globally optimal solution without needing to backtrack or reconsider previous choices. Solving these problems hinges critically on observation and insight to formulate a correct greedy strategy. Participants must rigorously prove that their local choice criterion consistently maintains the path towards the overall optimum, often using exchange arguments or demonstrating specific structural properties of the problem. For instance, minimizing added coins by only increasing the largest chests is optimal because altering the selection order by boosting smaller chests requires adding more coins.

Example Greedy Problem (Rated 800)

There are n chests; the i -th chest initially contains a_i coins. For each chest, you can choose any non-negative (0 or greater) number of coins to add to that chest, with one constraint: the total number of coins in all chests must become **at least** k .

After you've finished adding coins to the chests, greedy Monocarp comes, who wants the coins. He will take the chests one by one, and since he is greedy, he will always choose the chest with the maximum number of coins. Monocarp will stop as soon as the total number of coins in chests he takes is **at least** k .

You want Monocarp to take as few coins as possible, so you have to add coins to the chests in such a way that, when Monocarp stops taking chests, he will have **exactly** k coins. Calculate the minimum number of coins you have to add.

Input

The first line contains one integer t ($1 \leq t \leq 1000$)—the number of the test cases.

Each test case consists of two lines:

- The first line contains two integers n and k ($1 \leq n \leq 50; 1 \leq k \leq 10^7$);
- The second line contains n integers a_1, a_2, \dots, a_n ($1 \leq a_i \leq k$).

Output

For each test case, print one integer—the minimum number of coins you have to add so that, when Monocarp stops taking the chests, he has **exactly** k coins. It can be shown that under the constraints of the problem, it is always possible.

Tutorial — Human's accepted submission — o3-mini's rejected submission

Consider several first chests that Monocarp will take before exceeding the limit if we don't add any coins; so, this will be the set of several largest chests such that the sum of this set is $s \leq k$, but if the next chest is taken, the sum would exceed k .

For this set, the minimum number of coins that should be added is $k - s$. We can add exactly that amount if we increase the maximum element, the set of chests will include exactly the same elements, but now their sum is equal to k .

Now we have to consider the case where we add coins to some chest that is not the part of this set. Suppose Monocarp took some chest i , which initially had a_i coins, but did not take chest j , such that $a_j > a_i$. In order for chest i to be taken, its final value must be at least a_j as Monocarp selects the maximum chest from the available ones. Let's suppose that x coins were added to chest i , so that $a_i + x \geq a_j$. However, instead, we could have increased chest j to the value $a_i + x$, and this would require fewer coins, since $a_j > a_i$. Thus, we have shown that it is not optimal to "change" the order of the chests, so we can always assume that Monocarp takes several chests that were the largest in the original order.

Data Structures Data Structures problems center on efficiently organizing and storing data to support specific operations like querying, insertion, or deletion. Selecting or adapting suitable structures is key to meet performance constraints. Core knowledge involves understanding various structures (e.g., trees, heaps, segment trees) and their time/space complexity trade-offs, enabling optimal selection based on operational requirements. For instance, efficiently finding the minimum relevant endpoint among predictors often uses ordered sets for logarithmic-time boundary lookups.

Example Data Structures Problem (Rated 1900)

Suppose you are working in some audio streaming service. The service has n active users and 10^9 tracks users can listen to. Users can like tracks and, based on likes, the service should recommend them new tracks. Tracks are numbered from 1 to 10^9 . It turned out that tracks the i -th user likes form a segment $[l_i, r_i]$.

Let's say that the user j is a *predictor* for user i ($j \neq i$) if user j likes all tracks the i -th user likes (and, possibly, some other tracks too). Also, let's say that a track is *strongly recommended* for user i if the track is not liked by the i -th user yet, but it is liked by **every** predictor for the i -th user.

Calculate the number of strongly recommended tracks for each user i . If user doesn't have any predictors, then print 0 for that user.

Input

The first line contains one integer t ($1 \leq t \leq 10^4$)—the number of test cases. Next, t cases follow.

The first line of each test case contains one integer n ($1 \leq n \leq 2 \cdot 10^5$)—the number of users.

The next n lines contain two integers l_i and r_i per line ($1 \leq l_i \leq r_i \leq 10^9$)—the segment of tracks the i -th user likes.

Output

For each test case, print n integers, where the i -th integer is the number of strongly recommended tracks for the i -th user (or 0, if that user doesn't have any predictors).

Tutorial — Human's accepted submission — o3-mini's rejected submission

Firstly, if several segments are equal, then the answer for all of them is zero. Now let's move to the problem where all segments are distinct. User j is a predictor for user i iff $l_j \leq l_i \leq r_i \leq r_j$. Also, a track is strongly recommended if it is in all predictor segments, i.e. the track belongs to the intersection $[L, R]$ of all predictors. Since the segment $[l_i, r_i]$ also belongs to $[L, R]$, then the tracks we need to find form two intervals $[L, l_i]$ and $(r_i, R]$.

Let's focus on finding interval $(r_i, R]$. Since the right border of the intersection is the minimum among right borders, then our task is to find the minimum among r_j -s such that $r_j \geq r_i$ while $l_j \leq l_i$. Let's do it in the following way: let's sort all segments by l_i in increasing order; in case of equal l_i -s, sort them by r_i in decreasing order. If we process segments in the given order, then by the moment we process the i -th segment, all its predictors will be already processed.

Let's keep r_i -s of all processed segments so far in an "ordered set" S (std::set, for ex-

ample). Suppose we process segment i . Since the right borders of all predictors are already in S and their $r_j \geq r_i$, then finding the minimum among them is equivalent to just taking $R = S.lower_bound(r_i)$. Then we can add $R - r_i$ to the answer for the i -th segment. In order to calculate intervals $[L, l_i)$ we can just reflect all segments and solve the same problem.

Segment Tree Segment Tree problems are computational tasks focused on efficiently querying aggregate information over array intervals or segments and performing updates on array elements. These problems involve determining values like sums, minimums, or maximums within specified dynamic ranges. Solving these typically demands proficiency with the Segment Tree data structure, including its construction, query mechanisms, update procedures, and potentially lazy propagation, along with the ability to define appropriate merge operations for segment information. For instance, a problem requiring the maximization of $\max(a_l, \dots, a_r) - \min(a_l, \dots, a_r) - (r - l)$ over subsegments with point updates exemplifies this, as segment properties must be efficiently recomputed.

Example Segment Tree Problem (Rated 2000)

“T-Generation” has decided to purchase gifts for various needs; thus, they have n different sweaters numbered from 1 to n . The i -th sweater has a size of a_i . Now, they need to send some subsegment of sweaters to an olympiad. It is necessary that the sweaters fit as many people as possible, but without having to take too many of them. They need to choose two indices l and r ($1 \leq l \leq r \leq n$) to maximize the *convenience* equal to

$$\max(a_l, a_{l+1}, \dots, a_r) - \min(a_l, a_{l+1}, \dots, a_r) - (r - l),$$

that is, the range of sizes minus the number of sweaters. Sometimes the sizes of the sweaters change; it is known that there have been q changes, in each change, the size of the p -th sweater becomes x . Help the “T-Generation” team and determine the maximum convenience among all possible pairs (l, r) initially, as well as after each size change.

Input

Each test consists of several test cases. The first line contains a single integer t ($1 \leq t \leq 10^4$)—the number of test cases. The description of the test cases follows. The first line of each test case contains two integers n and q ($1 \leq n, q \leq 2 \cdot 10^5$)—the number of sweaters and the number of size changes. The second line of each test case contains n integers a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^9$)—the next size change.

Output

For each test case, output the maximum value of convenience among all possible pairs (l, r) before any actions, as well as after each size change.

Tutorial — [Human’s accepted submission](#) — [e3-mini’s rejected submission](#)

To begin with, let’s take a look at what an optimal segment of coats looks like. I claim that in the optimal answer, the maximum and minimum are located at the edges of the segment. Suppose this is not the case; then we can narrow the segment (from the side where the extreme element

is neither the minimum nor the maximum), and the answer will improve since the length will decrease, while the minimum and maximum will remain unchanged.

Okay, there are two scenarios: when the minimum is at l and the maximum is at r , and vice versa. These two cases are analogous, so let's consider the solution when the minimum is at l . Let's express what the value of the segment actually is: it is $a_r - a_l - (r - l) = (a_r - r) - (a_l - l)$, meaning there is a part that depends only on r and a part that depends only on l . Let's create a segment tree where we will store the answer, as well as the maximum of all $a_i - i$ and the minimum of all $a_i - i$ (for the segment that corresponds to the current node of the segment tree, of course).

Now, let's see how to recalculate the values at the node. First, the minimum/maximum of $a_i - i$ can be easily recalculated by taking the minimum/maximum from the two children of the node in the segment tree. Now, how do we recalculate the answer? In fact, it is simply the maximum of the two answers for the children, plus (the maximum in the right child) minus (the minimum in the left child), which is the case when the maximum is in the right child and the minimum is in the left. Since we maintain this in the segment tree, we can easily handle update queries.

Dynamic Programming Dynamic Programming problems are computational tasks focused on solving complex problems by decomposing them into a collection of simpler, overlapping subproblems. The solutions to these subproblems are memoized or tabulated to avoid redundant calculations, often aiming to find an optimal solution or enumerate possibilities. These problems heavily emphasize logical reasoning, demanding the precise definition of states and the derivation of recurrence relations that dictate the transitions between them, thereby building up the overall solution. For instance, calculating the number of valid subsegments where $dp[i]$, representing good subsegments starting at i , is derived from a subsequent state $dp[j + 1]$, showcases this reliance on solutions to overlapping subproblems.

Example Dynamic Programming Problem (Rated 1600)

Yaroslav is playing a computer game, and at one of the levels, he encountered n mushrooms arranged in a row. Each mushroom has its own level of toxicity; the i -th mushroom from the beginning has a toxicity level of a_i . Yaroslav can choose two integers $1 \leq l \leq r \leq n$, and then his character will take turns from left to right to eat mushrooms from this subsegment one by one, i.e., the mushrooms with numbers $l, l+1, l+2, \dots, r$.

The character has a toxicity level g , initially equal to 0. The computer game is defined by the number x —the maximum toxicity level at any given time. When eating a mushroom with toxicity level k , the following happens:

- The toxicity level of the character is increased by k .
- If $g \leq x$, the process continues; otherwise, g becomes zero and the process continues.

Yaroslav became interested in how many ways there are to choose the values of l and r such that the final value of g is not zero. Help Yaroslav find this number!

Input

Each test consists of multiple test cases. The first line contains an integer t ($1 \leq t \leq 10^4$)—the number of test cases. Then follows the description of the test cases.

The first line of each test case contains two integers n, x ($1 \leq n \leq 2 \cdot 10^5, 1 \leq x \leq 10^9$)—the number of mushrooms and the maximum toxicity level.

The second line of each test case contains n numbers a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^9$).

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output a single number—the number of subsegments such that the final value of g will not be zero.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

We'll solve the problem by dynamic programming. Let $dp[i]$ —the number of good subsegments with left boundary at i . We will count dp from the end, for each i we will find such a minimum j that the sum on the subsegment $[i; j]$ is greater than x . If there is no such j , then all right bounds are good, otherwise $dp[i] = dp[j+1] + j - i$. To search for j , we can use a binary search on prefix sums. The answer will be the sum of all dp .

Graph Theory Graph Theory problems are computational tasks focused on graphs—structures of vertices and edges. They typically involve analyzing graph properties, performing traversals, or determining optimal substructures or transformations to satisfy specified conditions. Solving these problems demands robust knowledge of graph representations, fundamental algorithms like

Breadth-First Search (BFS) and Depth-First Search (DFS), shortest path algorithms, and concepts such as connectivity, cycles, and minimum spanning trees. For instance, determining minimum edge changes in graph F to match graph G 's connectivity is graph-theoretic, involving analysis of reachability and structural graph manipulation.

Example Graph Theory Problem (Rated 1500)

You are given two simple undirected graphs F and G with n vertices. F has m_1 edges while G has m_2 edges. You may perform one of the following two types of operations any number of times:

- Select two integers u and v ($1 \leq u, v \leq n$) such that there is an edge between u and v in F . Then, remove that edge from F .
- Select two integers u and v ($1 \leq u, v \leq n$) such that there is no edge between u and v in F . Then, add an edge between u and v in F .

Determine the minimum number of operations required such that for all integers u and v ($1 \leq u, v \leq n$), there is a path from u to v in F **if and only if** there is a path from u to v in G .

Input

The first line contains an integer t ($1 \leq t \leq 10^4$) —the number of independent test cases.

The first line of each test case contains three integers n, m_1 and m_2 ($1 \leq n \leq 2 \cdot 10^5, 0 \leq m_1, m_2 \leq 2 \cdot 10^5$) —the number of vertices, the number of edges in F , and the number of edges in G .

The following m_1 lines each contain two integers u and v ($1 \leq u, v \leq n$) —there is an edge between u and v in G . It is guaranteed that there are no repeated edges or self loops.

It is guaranteed that the sum of n , the sum of m_1 , and the sum of m_2 over all test cases do not exceed $2 \cdot 10^5$.

Output

For each test case, output a single integer denoting the minimum operations required on a new line.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

Let's solve the problem for each operation.

First, consider the operation that removes an edge from F . Divide G into its connected components and assign each vertex a component index. Then, for each edge in F , if it connects vertices with different component indices, remove it and increment the operation count.

This guarantees no path between x and y in F if there is none in G . Next, to ensure a

path exists between x and y in F if there is one in G , we divide F into connected components. After removing excess edges, each component of F only contains vertices of the same component index. The number of operations needed now is the difference between the number of connected components in F and G .

All operations can be efficiently performed using DFS or DSU.

Tree Tree problems are a class of computational tasks centered on graphs that are acyclic and connected. These problems typically involve finding paths, calculating distances, identifying Lowest Common Ancestors (LCAs), or performing computations on subtrees and their properties. Solving tree problems often demands a strong understanding of graph traversal algorithms such as Depth-First Search (DFS) and Breadth-First Search (BFS), dynamic programming on trees, and occasionally specialized tree data structures or centroid decomposition. For instance, a problem asking to maximize connected components by removing a path from a given tree inherently relies on tree-specific path definitions and connectivity analysis.

Example Tree Problem (Rated 1900)

Given a tree with n vertices. You can choose two vertices a and b once and remove all vertices on the path from a to b , including the vertices themselves. If you choose $a = b$, only one vertex will be removed.

Your task is to find the maximum number of connected components that can be formed after removing the path from the tree.

Input

The first line of the input contains one integer t ($1 \leq t \leq 10^4$) —the number of test cases.

The first line of each test case contains one integer n ($2 \leq n \leq 2 \cdot 10^5$) —the size of the tree.

The next $n - 1$ lines contain two integers u and v ($1 \leq u, v \leq n, u \neq v$) —the vertices connected by an edge. It is guaranteed that the edges form a tree.

It is guaranteed that the sum of n across all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output one integer —the maximum number of connected components that can be achieved using the described operation.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

Let's choose some vertices a and b , between which there are k edges. Then, when removing this path, the tree will split into $s - 2 \cdot k$, where s is the sum of the degrees of the vertices on the path

(this is exactly how many edges are connected to the chosen path).

Let's suspend the tree from vertex 1, and for each vertex v of the given tree, we will calculate two values: $dp[v].x$ —the best answer if the path starts at vertex v and ends in its subtree, and $dp[v].y$ —the best answer if the path passes through vertex v from one of its children to another. The recalculations of the dynamic programming will be similar to those used in finding the diameter of the tree using dynamic programming.

The answer will be largest value among all $dp[v].x$ and $dp[v].y$.

Constructive Constructive problems require you to devise a sequence, graph, or configuration that satisfies given constraints. Often, multiple valid solutions exist, and you can output any of them. These problems primarily test observational skills and creative thinking, with minimal reliance on prerequisite knowledge. Each constructive problem usually demands a bespoke insight. The example provided is constructive because it requires the participant to reconstruct a sequence a with specific arithmetic and distinctness constraints from a shuffled subsequence b where one element is missing. The solution involves strategically choosing elements for a to satisfy the conditions, rather than deducing a single, predetermined answer.

Example Constructive Problem (Rated 1500)

You and your team have worked tirelessly until you have a sequence $a_1, a_2, \dots, a_{2n+1}$ of positive integers satisfying these properties.

- $1 \leq a_i \leq 10^{18}$ for all $1 \leq i \leq 2n + 1$.
- $a_1, a_2, \dots, a_{2n+1}$ are pairwise **distinct**.
- $a_1 = a_2 - a_3 + a_4 - a_5 + \dots + a_{2n} - a_{2n+1}$.

However, the people you worked with sabotaged you because they wanted to publish this sequence first. They deleted one number from this sequence and shuffled the rest, leaving you with a sequence b_1, b_2, \dots, b_{2n} . You have forgotten the sequence a and want to find a way to recover it.

If there are many possible sequences, you can output any of them. It can be proven under the constraints of the problem that at least one sequence a exists.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows.

The first line of each test case contains one integer n ($1 \leq n \leq 2 \cdot 10^5$).

The second line of each test case contains $2n$ **distinct** integers b_1, b_2, \dots, b_{2n} ($1 \leq b_i \leq 10^9$), denoting the sequence b .

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output $2n + 1$ **distinct** integers, denoting the sequence a ($1 \leq a_i \leq 10^{18}$).

If there are multiple possible sequences, you can output any of them. The sequence a should satisfy the given conditions, and it should be possible to obtain b after deleting one element from a and shuffling the remaining elements.

Tutorial — Human's accepted submission — o4-mini-high's rejected submission

The equation can be rearranged to

$$a_{2n} = a_1 + (a_3 - a_2) + (a_5 - a_4) + \dots + (a_{2n-1} - a_{2n-2}) + a_{2n+1}$$

Choose a_1 as the largest number, $a_3, a_5, \dots, a_{2n+1}$ as the next n largest numbers, and the rest as $a_2, a_4, \dots, a_{2n-2}$. The value of the missing number a_{2n} will be larger than a_1 , and so larger than every number in b .

Time complexity: $\mathcal{O}(n \log n)$ per test case. Logarithmic factor is from sorting.

Implementation Implementation problems are tasks that primarily involve the direct translation of a given set of rules, a clearly described process, or a straightforward algorithm into correct and efficient code, often without requiring novel algorithmic insights. These problems primarily assess a participant's attention to detail, ability to handle various specific conditions and edge cases meticulously, proficiency in a programming language, and skills in structuring code logically to simulate or execute the prescribed steps accurately. For instance, a problem involving reordering column inscriptions through defined, limited moves to achieve a sorted state, exemplifies an implementation problem as it stresses careful rule following and state management over complex algorithmic discovery.

Example Implementation Problem (Rated 1600)

Darius the Great is constructing n stone columns, each consisting of a base and between 0, 1, or 2 inscription pieces stacked on top. In each move, Darius can choose two columns u and v such that the difference in the number of inscriptions between these columns is exactly 1, and transfer one inscription from the column with more inscriptions to the other one. It is guaranteed that at least one column contains exactly 1 inscription. Since beauty is the main pillar of historical buildings, Darius wants the columns to have ascending heights. To avoid excessive workers' efforts, he asks you to plan a sequence of **at most** n moves to arrange the columns in non-decreasing order based on the number of inscriptions. Minimizing the number of moves is **not required**.

Input

The first line contains an integer t —the number of test cases. ($1 \leq t \leq 3000$).

The first line of each test case contains an integer n —the number of stone columns. ($1 \leq n \leq 2 \cdot 10^5$).

The second line contains n integers a_1, a_2, \dots, a_n , where $a_i \in \{0, 1, 2\}$ represents the initial number of inscriptions in the i -th column. It is guaranteed that at least one column has exactly 1 inscription.

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output an integer k —the number of moves used to sort the columns. ($0 \leq k \leq n$).

Then, output k lines, each containing two integers u_i and v_i ($1 \leq u_i, v_i \leq n$), representing the indices of the columns involved in the i -th move. During each move, it must hold that $|a_{u_i} - a_{v_i}| = 1$, and one inscription is transferred from the column with more inscriptions to the other. It can be proven that a valid solution always exists under the given constraints.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

Since we are allowed to perform n moves, assign each index one “move” as its “specified cost”.

While there exists an index with a value of 0 or 2 that can be fixed with just one move, fix it using its assigned cost.

After fixing all 0's, 2's, and all 1's except one, the remaining array will have the following structure and we are now allowed to use $2x + 1$ moves:

$$2\ 2\ \dots\ 2\ (x\ \text{times})\ 1\ 0\ 0\ \dots\ 0\ (x\ \text{times}),$$

First, swap the 1 with a random element (denote it as r). Then, for $2x - 1$ moves, swap the index with the value 1 with any index where the correct value must be placed, except r . Finally, swap 1 and r .

Ad-Hoc Ad-Hoc problems are computational tasks that generally defy classification under standard algorithmic techniques, necessitating solutions uniquely tailored to the specific constraints of each problem. They typically involve direct simulation, careful case analysis, or clever logical deductions rather than complex established algorithms. Success in these problems hinges critically on keen observation, pattern identification, and logical reasoning to uncover underlying simplifications or direct paths to a solution; the main focus is often on understanding the problem's intrinsic properties. For instance, identifying a substring with an even number of distinct non-empty substrings often relies on analyzing small cases and specific structural properties, rather than applying a general algorithm.

Example Ad-Hoc Problem (Rated 1000)

For a string p , let $f(p)$ be the number of distinct non-empty substrings of p .

Shohag has a string s . Help him find a non-empty string p such that p is a substring of s and $f(p)$ is even or state that no such string exists.

Input

The first line contains a single integer t ($1 \leq t \leq 10^4$) —the number of test cases.

The first and only line of each test case contains a string s ($1 \leq |s| \leq 10^5$) consisting of lowercase English letters.

It is guaranteed that the sum of the length of s over all test cases doesn't exceed $3 \cdot 10^5$.

Output

For each test case, print a non-empty string that satisfies the conditions mentioned in the statement, or -1 if no such string exists. If there are multiple solutions, output any.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

No one length strings are valid. Two length strings are valid if the adjacent characters are the same. So, if s contains two consecutive same characters, we can print it right away. All that remains is to consider strings without two consecutive same characters.

Three length strings are valid if all characters are different. So if s contains three consecutive different characters, we can print it right away. All that remains is to consider strings without two adjacent same characters but no three consecutive different characters. So all the remaining strings are of the form $s = abababababa\dots$

The total number of unique substrings of any length is 2. But only the length n substring occurs exactly once. So total number of unique substrings is $2n - 1$. And this is always odd! So there is no solution for these strings. We have covered all the cases.

If there are adjacent same characters, we can print it right away. If there are three consecutive different characters, we can print it right away. Otherwise there is no solution.

Time Complexity: $\mathcal{O}(n)$.

Case Work Case Work problems are a class of computational tasks where the solution involves partitioning the input space into a set of distinct scenarios or conditions. Each case is then typically analyzed and solved independently, often using a specific logic or a simpler algorithm tailored to that particular situation. Successfully tackling these problems hinges on strong observational skills to discern the critical patterns or thresholds that differentiate these cases, often by examining small examples or edge conditions. For instance, determining if a binary sequence can be reduced to [1] requires identifying key patterns, such as the presence of 111 or specific arrangements of 1's, each dictating a direct path to the solution.

Example Case Work Problem (Rated 900)

You are given a sequence $[a_1, \dots, a_n]$ where each element a_i is either 0 or 1. You can apply several (possibly zero) operations to the sequence. In each operation, you select two integers $1 \leq l \leq r \leq |a|$ (where $|a|$ is the current length of a) and replace $[a_l, \dots, a_r]$ with a single element x , where x is the majority of $[a_l, \dots, a_r]$.

Here, the majority of a sequence consisting of 0 and 1 is defined as follows: suppose there are c_0 zeros and c_1 ones in the sequence, respectively.

- If $c_0 \geq c_1$, the majority is 0.
- If $c_0 < c_1$, the majority is 1.

For example, suppose $a = [1, 0, 0, 0, 1, 1]$. If we select $l = 1, r = 2$, the resulting sequence will be $[0, 0, 0, 1, 1]$. If we select $l = 4, r = 6$, the resulting sequence will be $[1, 0, 0, 1]$.

Determine if you can make $a = [1]$ with a finite number of operations.

Input

Each test case contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 4 \cdot 10^4$). Description of the test cases follows.

The first line of each testcase contains one integer n ($1 \leq n \leq 2 \cdot 10^5$).

The second line of each testcase contains a string consisting of 0 and 1, describing the sequence a .

It's guaranteed that the sum of n over all testcases does not exceed $2 \cdot 10^5$.

Output

For each testcase, if it's possible to make $a = [1]$, print YES. Otherwise, print NO. You can output the answer in any case (upper or lower). For example, the strings yEs, yes, Yes, and YES will be recognized as positive responses.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

We list some simple conditions for a string to be transformable:

- If 111 exists somewhere (as a substring) in the string, the string is always transformable.
- If 11 appears at least twice in the string, the string is always transformable.
- If the string both begins and ends with 1, it is always transformable.
- If the string begins or ends with 1 and 11 exists in the string, it is always transformable.

These can be found by simulating the operation for short strings on paper.

Contrarily, if a string does not meet any of the four items, it is always not transformable. This can be proved using induction (as an exercise).

Binary Search Binary Search problems are computational tasks focused on efficiently finding a target value or an optimal solution, such as a minimum or maximum, within a typically large search space. This is achieved by repeatedly evaluating a monotonic predicate to narrow the possibilities. The core intellectual challenge involves identifying a monotonic property of the problem with respect to the variable being searched. This observation justifies the halving of the search interval based on a test function's outcome at the midpoint. For instance, when minimizing the longest bench length, if a length x is achievable, any length $y > x$ is also achievable, demonstrating the crucial monotonic property.

Example Binary Search Problem (Rated 1200)

For the final of the first Olympiad by IT Campus "NEIMARK", a rectangular venue was prepared. You may assume that the venue is divided into n rows, each containing m spots for participants' desks. A total of k participants have registered for the final, and each participant will sit at an individual desk. Now, the organizing committee must choose the locations for the desks in the venue.

Each desk occupies one of the m spots in a row. Moreover, if several desks occupy consecutive spots in the same row, we call such a group of desks a *bench*, and the number of desks in the group is the bench's length.

The organizing committee wants to choose the locations so that the length of the longest bench is as small as possible.

Given the integers n, m , and k , determine the minimum possible length of the longest bench.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows.

A single line of each test case contains three positive integers — n, m, k ($1 \leq n, m, k \leq 10^9, k \leq n \cdot m$).

Output

For each test case, output a single number — the minimum possible length of the longest bench.

Tutorial — Human's accepted submission — o4-mini-high's rejected submission

Let the length of the maximal bench be x , then to maximize the number of desks in one row—we need to put as many benches of exactly x length as possible. There should be

an indent after each bench, let's say, that the length of the block is $x + 1$. The total number of such blocks in a row will be $\lfloor \frac{m}{x+1} \rfloor$, and the last bench will have a length of $m \bmod (x + 1)$.

Then the number of desks in one row can reach $f(x) = x \cdot \lfloor \frac{m}{x+1} \rfloor + (m \bmod (x + 1))$.

Since the rows are independent, there should be $k \leq n \cdot f(x)$ desks in total. We need to find the minimum x , and since $f(x)$ is monotonically non-decreasing, then the answer can be found using binary search.

Bitmasking Bitmasking problems are computational tasks centered on representing subsets or discrete states as integers via their binary form (bitmasks). These problems typically involve using bitwise operations to systematically explore, enumerate, or optimize solutions, often applied to combinatorial set problems or to compress states in dynamic programming. The core of solving bitmasking problems lies in precise logical reasoning about how bitwise operations (AND, OR, XOR, shifts) affect the underlying data structures or states represented by the masks, demanding careful manipulation of bits. For instance, a problem seeking a sequence where $a_i | a_{i-1} = n$ is characteristic, as its constraints are directly defined by the bitwise OR operation on element representations.

Example Bitmasking Problem (Rated 1300)

You are given a positive integer n . Find the **longest** sequence of positive integers $a = [a_1, a_2, \dots, a_k]$ that satisfies the following conditions, and print the sequence:

- $a_i \leq n$ for all $1 \leq i \leq k$.
- a is strictly increasing. That is, $a_i > a_{i-1}$ for all $2 \leq i \leq k$.
- $a_i | a_{i-1} = n$ for all $2 \leq i \leq k$ where $|$ denotes the bitwise OR operation.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 1000$). Description of the test cases follows.

The only line of each test case contains one integer n ($1 \leq n \leq 10^{18}$).

It's guaranteed that the sum of lengths of the longest valid sequences does not exceed $5 \cdot 10^5$.

Output

For each testcase, print two lines. In the first line, print the length of your constructed sequence, k . In the second line, print k positive integers, denoting the sequence. If there are multiple longest sequences, you can print any of them.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

It's obvious that the answer only depends on the popcount of n . Below, we assume $n = 2^k - 1$. If $k = 1$, it is shown in the samples that the length is 1.

Otherwise, the maximum sequence length for $2^k - 1$ is $k + 1$. This can be achieved by $a_i = n - 2^{i-1}$ ($1 \leq i \leq k$), $a_{k+1} = n$.

Why?

Consider the value of a_1 . Note that its k -th bit (indexed from 2^0 to 2^{k-1}) should be 0, otherwise we would not make use of the largest bit.

Since a_1 and a_2 's OR is n , a_2 's k -th bit should be 1, and thus the construction of $a_2 \sim a_{k+1}$ boils down to the subproblem when $n = 2^{k-1} - 1$. This shows that $f(k) \leq f(k-1) + 1$ where k is the popcount of n and $f(k)$ is the maximum sequence length. We know that $f(2) = 3$, so $f(k) \leq k + 1$ for all $k \geq 2$.

Two Pointers Two Pointers problems involve traversing data structures, commonly arrays or strings, using two distinct indices. These pointers move based on specific conditions to efficiently identify elements, pairs, or contiguous segments that satisfy problem constraints, often optimizing for linear time complexity. These problems primarily test the ability to recognize monotonic properties and invariant conditions. Successful solutions require astute logical reasoning for pointer advancement and meticulous management of the window or interval defined by the pointers. For instance, finding the shortest subarray whose sum meets a target value would utilize two pointers to define a sliding window, adjusting its boundaries based on the current sum.

Example Two Pointers Problem (Rated 1200)

Vlad found a strip of n cells, numbered from left to right from 1 to n . In the i -th cell, there is a positive integer a_i and a letter s_i , where all s_i are either 'L' or 'R'.

Vlad invites you to try to score the maximum possible points by performing any (possibly zero) number of operations.

In one operation, you can choose two indices l and r ($1 \leq l < r \leq n$) such that $s_l = \text{'L'}$ and $s_r = \text{'R'}$ and do the following:

- add $a_l + a_{l+1} + \dots + a_{r-1} + a_r$ points to the current score;
- replace s_i with '.' for all $l \leq i \leq r$, meaning you can no longer choose these indices.

What is the maximum score that can be achieved?

Input

The first line contains one integer t ($1 \leq t \leq 10^4$)—the number of test cases.

The first line of each test case contains one integer n ($2 \leq n \leq 2 \cdot 10^5$)—the length of the strip.

The second line of each test case contains n integers a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^5$)—the numbers written on the strip.

The third line of each test case contains a string s of n characters 'L' and 'R'.

It is guaranteed that the sum of the values of n across all test cases does not exceed $2 \cdot 10^5$,

Output

For each test case, output one integer—the maximum possible number of points that can be scored.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

Note that since all characters of the selected segment of the string s are erased after applying the operation, the segments we choose cannot overlap. However, they can be nested if we first choose an inner segment and then an outer one.

Since all numbers in the array are positive, it is always beneficial to take the largest possible segment in the answer, that is, from the first 'L' to the last 'R'. By choosing this segment, we can only select segments within it. We will continue to choose such segments within the last selected one as long as possible. To quickly find the sums of the segments, we will calculate the prefix sums of the array a .

Interactive. Interactive problems are defined by a computational model wherein an algorithm dynamically communicates with an external oracle to solve a task. Unlike problems with static inputs, the algorithm here iteratively makes queries or performs actions, receiving feedback that guides its subsequent steps towards a solution. This interactive process critically assesses an algorithm's capacity for adaptive information gathering and inference from partial, sequentially revealed data. A significant subset of these problems is query-based, focusing on the deduction of hidden information; for instance, determining a concealed binary string by repeatedly querying an oracle about the substring status of various test strings and utilizing the binary feedback to reconstruct the target.

Example Interactive Problem (Rated 1400)

Dimash learned that Mansur wrote something very unpleasant about him to a friend, so he decided to find out his password at all costs and discover what exactly he wrote.

Believing in the strength of his password, Mansur stated that his password—is a binary string of length n . He is also ready to answer Dimash's questions of the following type:

Dimash says a binary string t , and Mansur replies whether it is true that t is a substring of his password.

Help Dimash find out the password in no more than $2n$ operations; otherwise, Mansur will understand the trick and stop communicating with him.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 100$). The description of the test cases follows.

Interaction

At the beginning of each test case, first read n ($1 \leq n \leq 100$)—the size of the binary string. Then proceed to guessing it.

To guess string s , you can make no more than $2n$ queries of the following type:

- “? t ”, where t is a binary string such that ($1 \leq |t| \leq n$).

In response to this query, you will receive 1 if t is a substring of s , and 0 otherwise.

Once you receive the answer, output a single string in the following format:

- “! s ”, where s is a binary string of size n .

After that, proceed to solve the next test case.

Tutorial — Human's accepted submission — Gemini 2.5 Pro's rejected submission

We will initially maintain an empty string t such that t appears as a substring in s .

We will increase the string t by one character until its length is less than n .

We will perform n iterations. In each iteration, we will check the strings $t + 0$ and $t + 1$. If one of them appears in s as a substring, we will add the appropriate character to the end of t and proceed to the next iteration.

If neither of these two strings appears in s , it means that the string t is a suffix of the string s . After this iteration, we will check the string $0 + t$. If it appears in s , we will add 0 to t ; otherwise, we will add 1 .

Thus, in each iteration, we perform 2 queries, except for one iteration in which we perform 3 queries. However, after this iteration, we will make only 1 query, so the total number of queries will not exceed $2 \cdot n$.

A12. Case Studies

In this section, we compare the reasoning process of DeepSeek R1 and o4-mini-high on solving competitive programming problems. We take <https://codeforces.com/contest/2046/problem/B> as an example, which belongs to the “Constructive” and “Greedy” category and is an observation-heavy problem. It has a difficulty rating of 1600. In the following, we first present its problem statement and official solution.

Example Problem for Case Studies (Rated 1600)

You are given an array of integers a of length n . You may perform the following operation any number of times (possibly zero):

- Choose an index i ($1 \leq i \leq n$), replace a_i by $a_i + 1$, and then move this element to the back of the array (the rightmost position).

For example, if $a = [3, 5, 1, 9]$ and you choose $i = 2$, the array becomes $[3, 1, 9, 6]$.

Your task is to obtain the *lexicographically smallest* array that can be produced by a sequence of such operations.

Input

Each test contains multiple test cases. The first line contains a single integer t ($1 \leq t \leq 10^4$)—the number of test cases. The descriptions of the test cases follow.

For each test case:

- The first line contains an integer n ($1 \leq n \leq 10^5$)—the length of the array.
- The second line contains n integers a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^9$)—the elements of the array.

It is guaranteed that the sum of n over all test cases does not exceed 10^5 .

Output

For each test case, print one line containing the lexicographically smallest array achievable, with its elements separated by single spaces.

Official Tutorial

The key observation is that *each element can be moved to the back at most once*. Fix any subset of indices that will ever be moved. If we perform the operation *exactly once* on every element of that subset (in any order), their final relative order is fixed and their values are each increased by 1. To minimise the final array lexicographically we clearly want those “moved” elements to appear in *increasing order*.

Hence the problem reduces to determining *which* elements must be moved to the back.

Greedy construction (left \rightarrow right). To obtain the lexicographically smallest array we decide each position greedily:

1. The first position must contain the *global minimum*. Therefore every element that appears before that minimum in the original array must be moved to the back (and thus increased by 1).
2. For the second position we now choose the smallest possible value among
 - the suffix minimum that follows the global minimum, and
 - the set of elements that have already been moved to the back.

3. We repeat this procedure while we can still take the next value from the *suffix-minimum sequence*. Eventually the next candidate is *smaller* in the pile of “moved” elements than in the remaining suffix minima. At that moment every element still to the right must be moved once (hence increased by 1), and afterward all moved elements are listed in increasing order.

Resulting structure of the answer. The final array therefore consists of

- a prefix equal to a contiguous segment of the suffix-minimum sequence (starting with the global minimum), followed by
- *all other* elements, each increased by 1 and then written in sorted ascending order.

Implementation details.

- Pre-compute the suffix minima array.
- Scan the original array from left to right, maintaining a multiset (or priority queue) of values that have already been moved.
- At each step decide whether the next output element comes from the current suffix minimum or from the multiset.
- Once the multiset yields a strictly smaller value, output that value, move every remaining original element into the multiset (adding 1), and finally output the multiset contents in increasing order.

Complexity. The procedure performs standard multiset operations on at most n elements, resulting in a total complexity of $\mathcal{O}(n \log n)$, which easily meets the constraints.

In the following, we present the reasoning process of DeepSeek R1. Due to its extensive length, we omit the repetitive and less meaningful parts for brevity. DeepSeek R1 identified early in its reasoning process that a greedy selection from the “suffix minima sequence” could be a viable strategy. Its progress stalled, however, because it missed the key condition for stopping this greedy selection—namely, when the next target element is no longer the next available minimum in the suffix. Lacking this insight, it fruitlessly revisited variations of its initial strategy without generating new, effective observations, leading to an incorrect solution derived from this limited exploration.

We also show the reasoning process of o4-mini-high, detailed below, which demonstrates a more effective exploration pattern. It investigates a diverse set of potential strategies — including dynamic programming, BFS-based brute-force, and greedy methods validated with smaller instances. Even accounting for the curation of the output log, its reasoning path is markedly more concise. Crucially, unlike processes that get stalled in the case of DeepSeek R1, o4-mini-high shows little evidence of unproductive iteration on failed attempts, instead progressing efficiently through its varied explorations to arrive at the correct answer on this problem. This might be due to a combination of better explorations of problem-solving strategies and more accurate identification of crucial observations

with the aid of terminal access and tool calls.

Reasoning Process of DeepSeek R1

Okay, so I have this programming problem to solve. Let me read it carefully.

[DeepSeek R1 tries to understand and interpret the problem, omitted for brevity.]

So, the problem is similar to arranging the elements in a certain order where each element is chosen from the remaining elements (possibly modified by prior operations) such that the lex order is minimized. However, each time you select an element to place next, you can choose to leave it as is (if you don't perform any operation) or you can perform some operations to move it to the end (but each operation increases it by 1 and moves it to the end). Wait, but moving to the end once would send it to the end, but if you do multiple operations on the same element, each time you have to increment it again. So perhaps the best approach is to find the order in which elements are arranged such that each element is the minimum possible in the current remaining elements, considering that moving earlier elements to the end (with increments) might allow us to get a smaller next element.

[By this time, DeepSeek R1 has already realized this is a greedy problem.]

Alternatively, maybe we can model this as a greedy problem where we need to select the next element in the sequence to be the smallest possible, considering that any elements that come before it in the original array can be moved to the end (with some cost) to allow the selection of a smaller element.

Wait, let's think of the process. Suppose the array is processed from left to right. At each step, for the current position, we can choose to leave the current element as is (and proceed to the next position), or perform some number of operations on elements to the left (but how?), which would move those elements to the end. Wait, but once you move an element to the end, you can't access it again unless you move it again.

Alternatively, perhaps the optimal array is constructed by selecting a certain suffix of the original array, possibly modified by some operations, followed by some elements that have been moved to the end (and thus modified). Let me think.

[DeepSeek R1 investigates this idea more closely by studying samples, omitted for brevity.]

But how to model this algorithmically?

Hmm. So the problem is to arrange the elements in some order, where each element can be in its original position or moved to the end after being incremented (possibly multiple times). The lex order is determined by the sequence of elements in the array after all operations.

The key is to find the sequence of elements (some of which are in the original array, others are incremented and moved to the end) such that the lex order is as small as possible.

But how to model this? It's a bit like a greedy algorithm where at each step, we select the next element in the array to be the smallest possible.

Wait, but when you move elements to the end, you can't process them again unless you move them again. So each time you process an element, you have to decide whether to keep it in place or move it to the end (with an increment), which affects the order of the remaining elements.

Alternatively, think of the process as building the array step by step. For each position in the array (from left to right), choose the smallest possible element from the current possible candidates. The candidates are:

- The current front of the array (without moving it), or
- Any element that can be moved to the end (after incrementing) to allow a smaller element to be next.

But how?

Wait, perhaps the problem can be transformed into finding a suffix of the original array that can be modified (by moving some elements to the end with increments) such that the lex order is minimized.

Alternatively, here's an approach inspired by the problem of finding the lex smallest array by rotating or permuting elements with certain operations. For example, in some problems, the lex smallest array can be found by finding the minimal suffix.

But here, the operations are more complex. Let me think.

Suppose that for each position i in the array, we can choose to move some elements before i to the end (after incrementing them) so that the element at i becomes the first element. Then, the optimal choice is to select the minimal possible element as the first element, considering the cost (increment) required to move the elements before it.

Wait, but moving elements before it would require moving them to the end (each time incrementing), which would affect their positions. So perhaps for each possible candidate element (in the array), we can compute the minimal cost (number of increments) needed to bring that element to the front, and choose the element with the minimal value after considering the cost.

But how to model this?

Alternatively, think of the process as choosing a starting index k , such that the elements from k onwards form the beginning of the lex smallest array, and the elements before k are moved to the end (each possibly after some increments). But each time an element is moved to the end, it's increased by 1, and each such operation can be done any number of times.

Wait, but moving elements before k to the end. For example, if k is the starting position, then the elements $a[k], a[k+1], \dots, a[n]$ form the beginning of the array. But any elements before k ($a[1..k-1]$) can be moved to the end (each with +1 for each move). However, moving an element to the end requires that all elements before it are either moved or not. Wait, no. Because moving an element to the end requires that you process it (increment and move), which changes the array.

[DeepSeek R1 is getting astray and decides to backtrack by considering alternative approaches. However, the strategy it considers in the following is essentially quite similar to the previous approach.]

Hmm, this is getting complicated. Let's think of the problem differently.

Suppose that the optimal array is formed by taking some elements from the original array (possibly modified by increments) in an order. Each time you select an element to add to the result, you can choose to take the current first element (without modification), or you can increment it and move it to the end (so that you can process the next element). Then, the problem becomes similar to finding the minimal sequence by considering the current head of the array and decide whether to take it or push it (modified) to the end.

This approach is similar to the problem of finding the lex smallest array by rotating, but here each rotation (push) comes with a cost (incrementing the element). So the decision at each step is to take the current head, or increment it and push to the end, then proceed.

But since the operations can be done any number of times, how do we model this?

This sounds like a problem that can be approached with a greedy algorithm. At each step, we choose between the current head and all possible elements that can be obtained by pushing the current head (after incrementing) to the end and considering the next element. But this could be computationally expensive if not optimized.

But perhaps there's a way to find the minimal possible sequence by considering all possible candidates and selecting the best one.

Wait, but for large n (up to 10^5), we need an $O(n)$ or $O(n \log n)$ solution.

Another approach: the optimal array will consist of a sequence where each element is the minimal possible, considering that any elements before it can be pushed to the end (with increments). So perhaps the first element is the minimal element in the array, but adjusted by the number of times elements before it are pushed to the end. Because each time you push an element before it, you have to increment that element once and move it to the end.

But how to model this?

Alternatively, think of the process as building the array by choosing the next element to be the minimal possible in the current available elements. The available elements are the remaining elements in the array (those that haven't been pushed to the end). But when you push elements to the end, you modify them.

Wait, but once you push an element to the end, it's modified and added to the end. So the elements in the end are modified (incremented) each time they are pushed.

[This is closer to the correct solution, but DeepSeek R1 rejects this idea.]

Alternatively, perhaps the best way is to model this as a priority queue problem. For example, each time we can choose the minimal element in the current array, and decide to take it (without pushing) or push some elements before it. But this seems unclear.

Alternatively, here's an idea inspired by the problem of finding the minimal possible string by choosing characters from a deque, where you can pop from the front or the back. But again, the problem here is different.

Wait, another way to model the problem: Each element can be moved to the end any number of times. Each time it's moved, it's increased by 1 and placed at the end. So, for an element x , if we move it to the end k times, its value becomes $x + k$, and it's placed at the end of the array. But the order in which elements are moved affects their final positions and values.

But how do we choose the order of moves to minimize the lex order of the resulting array?

Perhaps the optimal approach is to find a permutation of the array where each element is at least x_i plus the number of times it was moved. But this seems vague.

Let me think about the process. Suppose that the array is processed left to right. For each element, we can decide whether to take it as is (without moving) or move it to the end (after incrementing). Once it's moved, it's added to the end, and we can process the next element (the next one in the original array). But if we move elements, the order of processing changes.

Alternatively, imagine that the array is processed in a way that whenever we encounter an element that is larger than the minimal possible element in the remaining array (including possible modified elements due to moves), we move it to the end (with increment) and proceed. But this may not be optimal because moving it to the end may allow us to have a smaller element next, but the moved element is now larger (due to increment).

Hmm.

Wait, perhaps the key observation is that once we decide to move an element to the end, it's better to move it once and then possibly move others, but the element is now in the end and can't be accessed again unless moved again, which would cost another increment.

But since the goal is to have the lex smallest array, perhaps moving an element to the end is only worth it if the element that follows is smaller, even after considering the increment. For example, if the next element is smaller than the current element after increment, then moving the current element may be beneficial.

Alternatively, for each position i in the array, we can track the minimal possible value that can be placed at the current position, considering that the elements before i can be moved to the end (with increments) any number of times.

Let me try to model this.

Suppose we are building the lex smallest array. The first element must be the minimal possible among all elements that can be placed in the first position. How?

Each element can be in the first position only if all elements before it are moved to the end. For example, to get a_i as the first element, all elements a_1, a_2, \dots, a_{i-1} must be moved to the end (each moved once, which increases them by 1 and appends to the end). Then, the array would be $a_i, a_{i+1}, \dots, a_n, (a_1 + 1), (a_2 + 1), \dots, (a_{i-1} + 1)$.

But this is just moving all elements before i to the end once, so each of those elements is increased by 1. Then, the first element is a_i . Then, the rest of the array is a_{i+1}, \dots, a_n , followed by the moved elements.

But perhaps the minimal first element is the minimal $a_j + k_j$, where k_j is the number of times the element a_j is moved to the end before being placed in the first position. Wait, this seems complicated.

Alternatively, the minimal possible first element is the minimal $a_i + k_i$, where k_i is the number of times elements before i are moved. Because each time you move an element before i , you have to move it once for each element. So for example, to get a_i as the first element, you have to move all elements before it to the end once each. So each element a_1 to a_{i-1} is increased by 1 and moved to the end. So a_i 's value remains the same. Then, the first element is a_i , and the rest of the array is $a_{i+1}, a_{i+2}, \dots, a_n, a_1 + 1, a_2 + 1, \dots, a_{i-1} + 1$.

So the cost (in terms of the first element's value) is a_i . But the rest of the array's elements may be larger. So the first element's minimal value is a_i , but the rest of the array must be considered.

So perhaps the best approach is to find the minimal possible a_i (original, not increased) among all possible i , such that the suffix starting at i (after moving the previous elements) is lex smaller than any other suffix.

This is similar to the problem of finding the minimal suffix, but with some modifications.

For example, in the first sample, the minimal a_i is 1 (at position 2). Moving the elements before it (position 1, which is 2) to the end. So the array becomes [1, 3, 3]. Which is the sample output.

In the second sample, the output starts with 1 (position 1) and then 1 (position 4). So after moving elements 2 and 2 and 1 (positions 2,3,4) to the end, but how?

Wait, perhaps the minimal possible first element is a_i , and then the minimal possible second element after that, etc.

[DeepSeek R1 starts considering a specific way of using deque to implement its idea, though the idea is still essentially the same as before.]

Alternatively, the problem can be viewed as finding a sequence of indices i_1, i_2, \dots, i_k such that each i_j is in the original array (possibly modified by prior moves) and forms the lex smallest array.

But I'm not sure.

Let me think of another approach. Let's model the array as a deque. At each step, we can take the front element, and decide whether to keep it (as part of the result) or push it to the end after incrementing. The goal is to choose the order of these operations to form the lex smallest array.

[DeepSeek R1 digs deeper into this idea, tests on sample inputs, and finds that it doesn't work. Omitted for brevity.]

This is getting very complicated. Perhaps there's a different way to model the problem.

Another idea: the optimal array is a sequence of elements chosen from the original array, where each element is a_i plus the number of times it has been moved to the end. The order of elements must be such that if an element is chosen at position k , then all elements that were before it in the original array but not chosen before have been moved to the end (thus, their increments are counted).

But this seems similar to the earlier approaches.

Alternatively, think of the problem as a permutation of the elements, where each element can be increased by some amount, and the lex smallest possible array is desired. The permutation is determined by the order in which elements are moved to the end.

This permutation must be such that for each position, the element is as small as possible, considering that moving earlier elements allows later elements to be considered earlier.

But how to model this.

[DeepSeek R1 decides to think of this problem in terms of a priority queue, which is a nice progress. However, its observations (which are limited and not fully correct) are still essentially the same as before.]

Let me think of the problem in terms of a priority queue. Each time, the minimal possible element in the current deque is selected. If it's at the front, we take it. If it's somewhere else, we need to move elements in front of it to the end (each time incrementing them) to bring it to the front. However, moving elements to the end increases their value, which may affect their position in the priority queue.

[DeepSeek R1 again digs deeper into this idea, tests on sample inputs, and finds that it doesn't work as well. Omitted for brevity.]

Hmm. This suggests that there's a different approach needed.

Alternatively, perhaps the correct approach is to move elements to the end as long as there exists a later element that is smaller than the current element (original, not modified).

But how to model this.

[DeepSeek R1 again tries to look for alternative approaches, but ends up with essentially the same idea as before.]

Alternatively, think of the problem as the following: each element can be kept in place (without moving), or moved to the end once or more times. Each time it's moved, its value increases by 1.

[In the following, DeepSeek R1 basically just revisits variations of its incorrect strategies without generating new, effective observations. Omitted for brevity.]

Another idea is to use a linked list to keep track of the elements and their current values, and maintain a pointer to the minimum element.

But this would be complex.

Given the time constraints, I think the correct approach is to use a deque and implement the greedy algorithm of moving the current element to the end if there's a smaller element later in the current deque. The algorithm would be:

Initialize a deque with the array elements. Initialize a result array. While the deque is not empty: Find the index of the minimal element in the deque. If the minimal element is at the front: append it to the result. remove it from the deque. else: move the front element to the end, increment it by 1. Repeat.

This approach is $O(n^2)$ in the worst case (e.g., when the deque is always in decreasing order), but for the problem constraints (n up to 10^5), this would not be feasible.

However, perhaps in practice, the algorithm can be optimized using a priority queue or a more efficient way to track the minimal element.

Given the time constraints, I'll proceed to write the code that passes all the samples and hope it works.

[DeepSeek R1 attempts several implementations. Omitted for brevity.]

Thus, the correct solution is to compute for each element whether it should be moved (if there's a smaller element later), collect the moved elements, sort them, and append them to the non-moved elements.

The code is as follows:

```

def solve():
    import sys
    input = sys.stdin.read().split()
    ptr = 0
    t = int(input[ptr])
    ptr += 1
    for _ in range(t):
        n = int(input[ptr])
        ptr += 1
        a = list(map(int, input[ptr:ptr+n]))
        ptr += n

        min_right = [float('inf')]*(n+1)
        for i in range(n-1, -1, -1):
            min_right[i] = min(a[i], min_right[i+1])

        res1 = []
        res2 = []
        for i in range(n):
            if min_right[i+1] < a[i]:
                res2.append(a[i]+1)
            else:
                res1.append(a[i])

        res2.sort()
        res = res1 + res2
        print(' '.join(map(str, res)), end=' ')
        print()

```

This code passes all the samples and is efficient ($O(n \log n)$ due to sorting).

The key steps are:

1. Precompute the minimum to the right of each element.
2. For each element, decide whether to move it (if there's a smaller element to the right).
3. Collect the non-moved elements in order and the moved elements in a list.
4. Sort the moved elements and append them to the non-moved elements.

This works because the moved elements are all added to the end of the array in sorted order, which is lex smallest.

[But actually this doesn't work.]

Reasoning Process of o4-mini-high

The user asked me to solve the problem step-by-step, reasoning it out myself and producing a solution in code (either Python or C++). To approach the solution, I need to break the problem down into smaller steps rather than looking for external hints or tutorials. First, I'll focus on the constraints and think carefully about the algorithm, keeping time complexity in mind. I'll use clear bullet points for the explanation and provide the Python or C++ code at the end. Let me get started on solving this!

The user asks to perform an operation on any element by selecting it, incrementing it, and moving it to the end. We repeat this for each element to create a lexicographically smallest array. Each element's order in the final array depends on the number of operations applied to it. Importantly, multiple operations on an element move it further back each time but don't duplicate it. Ultimately, we're rearranging the elements based on these operations to achieve the smallest possible order.

The goal is to reorder the array efficiently by using the minimum number of operations and creating the lexicographically smallest sequence. Each element can only be moved to the back, meaning that elements can't jump forward — their original position will impact the final order. We need to focus on keeping the elements that are already in a good position (typically the smallest ones) at the front, while carefully deciding which elements to increment and move toward the back. The final result leans heavily on the number of operations and the order in which they are done.

The key idea is to maintain a queue and selectively move elements while keeping the array lexicographically smallest. When we remove elements, their values increase due to multiple increments, but the goal is to avoid unnecessary increments for smaller elements placed early. We can interleave operations and schedule when each element stops moving. The result depends on the last operation time of each element—those that weren't moved stay upfront in their original order, while moved ones follow based on the order of their last operation. This ensures the smallest final sequence.

[By now, o4-mini-high realizes that a greedy strategy should be applied here. These observations are correct but not enough to fully solve the problem.]

The task is to minimize the lexicographical order by grouping elements that are moved and not moved. The elements never moved remain in their original position, forming the front array (S_0). The moved elements can be reordered but should have their values adjusted by the number of moves they underwent. To find the optimal pattern, we start with the smallest element choosing it for S_0 and proceed by selecting subsequent smallest elements to minimize the final sequence. Sorting the moved group (S_1) ensures the lexicographically smallest result.

Analyzed

```
from collections import deque
import itertools

def best_final(a, max_ops):
    # BFS starting from a, depth <= max_ops
    n = len(a)
    seen = {tuple(a)}
    best = tuple(a)
    q = deque([(tuple(a), 0)])
    while q:
        state, d = q.popleft()
        # update best
        if state < best:
            best = state
        if d == max_ops:
            continue
        for i in range(len(state)):
            new = list(state[:i] + state[i+1:]) + [state[i]+1]
            new_t = tuple(new)
            if new_t not in seen:
                seen.add(new_t)
                q.append((new_t, d+1))
    return best
```

candidate algorithm: suffix approach: for each possible i from 0 to n (0 means no suffix $\Rightarrow S_0$ empty), do: # $S_0 = a[i:]$, $S_1 = \text{prefix } a[:i]$. Then $k_j = 1$ for $j < i$, 0 for $j \geq i$. # Final $c_i = S_0$ elements in original order, then elements of S_1 in original order +1. # Actually, we can reorder S_1 arbitrarily. But BFS only produces specific reorder where S_1 in original order, because we can only remove from front. # But BFS, if we always remove prefix in order, yields prefix removed. But can we reorder prefix? But we could pop prefix elements in any order? But we cannot remove prefix elements in arbitrary order: only available operation is choose any i , so you can choose inside prefix removing some later prefix element by $\text{index} > 0$, but prefix group initially front are prefix elements. But after removing some, new front shifts. But you cannot remove prefix element at position $j > 1$ before removing a preceding element? But you can choose $i = 2$ removes second element, leaving first element unchanged. So you can remove second then first. That yields prefix removed in any order. But BFS allows removal in any sequence. But suffix approach with arbitrary reorder of prefix emerges? But need to test both approaches: reorder prefix or prefix in original order.

For candidate, we consider S_1 sorted by $a_j + 1$ ascending (best sorted), since we can reorder S_1 arbitrarily by scheduling operations at choice. But BFS may not achieve arbitrary reordering in S_1 ? But we suppose can. But we test both with sorted and original order.

```

def suffix_candidate(a):
    n = len(a)
    best = None
    for i in range(n+1):
        S0 = a[i:]
        S1 = a[:i]
        # Option 1: S1 in ord order +1
        c1 = S0 + [x+1 for x in S1]
        # Option 2: S1 sorted by x+1 then
        c2 = S0 + sorted([x+1 for x in S1])
        for c in [tuple(c1), tuple(c2)]:
            if len(c)!=n: continue
            if best is None or c<best:
                best = c
    return best

# test all arrays length 3 values in 1..3
fails = []
for a in itertools.product([1,2,3], repeat=3):
    bf = best_final(list(a), 3)
    sc = suffix_candidate(list(a))
    if bf != sc:
        fails.append((a, bf, sc))
        # print details
        if len(fails)>10:
            break

len(fails), fails[:5]
(4,
 [((1, 2, 1), (1, 1, 3), (1, 2, 1)),
  ((1, 3, 1), (1, 1, 4), (1, 2, 4)),
  ((1, 3, 2), (1, 2, 4), (1, 3, 2)),
  ((2, 3, 2), (2, 2, 4), (2, 3, 2))])

```

[o4-mini-high plans to stress test its solution by running BFS-based brute force on self-generated small test cases.]

I'm considering whether dynamic programming might help here, but with the constraints, I think it's too slow. So, I'm going to try brute-forcing smaller arrays ($n \leq 4$) to find a general pattern. The goal is to explore possible sequences and target the lexicographically smallest final array by checking different sequences of operations. I've confirmed that k_i should be either 0 or 1, but I'm wondering if allowing $k_i > 1$ could offer better solutions, though I think it's unnecessary. Let's test a brute-force approach for small cases.

I'm going to test this on small arrays ($n \leq 4$). I'll limit operations to at most once per element, since moving anything more than that just increases its value unnecessarily. I'm planning to use a brute force approach with BFS. For simplicity, I'll start with arrays of length 3 or 4 and try to find the best sequence by tracking the lexicographically smallest outcomes. The idea is to explore only those states that lead to a minimal result after a limited number of operations.

I'm refining the formula approach. If we start with a suffix, we add the sorted prefix, but there's an issue: skipping might lead to an incorrect sequence. I thought c_2 should be smaller if we skip, but then I realize maybe it's not so simple. The idea of dynamic programming seems appealing, but the constraints could be massive, so I'll keep it simple and test with small arrays. Let's brute-force examples for $n \leq 4$ and look for patterns instead.

Analyzed

```

def greedy_candidate(a):
    n = len(a)
    # pair list
    pairs = sorted([(a[i], i) for i in range(n)],
                    key=lambda x: (x[0], x[1]))

```

```

S0 = []
last = -1
for val, idx in pairs:
    if idx > last:
        S0.append(idx)
        last = idx
S0_set = set(S0)
# sorted S0 list by idx
S0_sorted = sorted(S0)
S1 = [i for i in range(n) if i not in S0_set]
# S1 sorted by a[i]+1 value ascending, then index to break ties
S1_sorted = sorted(S1, key=lambda i:(a[i]+1, i))
c = [a[i] for i in S0_sorted] + [a[i]+1 for i in S1_sorted]
return tuple(c)

# test on arrays from previous fails
fails2 = []
for (a,bf,sc) in fails:
    c = greedy_candidate(list(a))
    if c != bf:
        fails2.append((a,bf,c))
len(fails2), fails2[:5]

(0, [])

```

Analyzed

```

import random
fails_rand = []
for _ in range(200):
    n=4
    a = [random.randint(1,5) for _ in range(n)]
    bf = best_final(a, n)
    c = greedy_candidate(a)
    if bf != c:
        fails_rand.append((tuple(a), bf, c))
        print("fail", a, bf, c)
        break
len(fails_rand)

fail [5, 2, 1, 4] (1, 3, 5, 6) (1, 4, 3, 6)
1

```

I'm thinking of coding a brute force approach for small arrays, using BFS to explore all possible states up to a bound. If k_i is restricted to 0 or 1, that's manageable, though I wonder if allowing $k_i > 1$ would help. I'm limiting the number of operations to n , especially since trying to move an element twice will only increase values, which isn't helpful. Let's implement the brute force for arrays of size 3 or 4 and track the best sequence found.

Analyzed

```
def best_with_path(a, max_ops):
    from collections import deque
    seen = {tuple(a):None}
    q = deque([(tuple(a),0)])
    parent = {}
    op_chosen = {}
    final_best = tuple(a)
    best_state = tuple(a)
    while q:
        state, d = q.popleft()
        if state < best_state:
            best_state = state
        if d == max_ops:
            continue
        for i in range(len(state)):
            new = state[:i] + state[i+1:] + (state[i]+1,)
            if new not in seen:
                seen[new] = (state, i)
                q.append((new, d+1))
    # reconstruct path to best_state
    cur = best_state
    path = []
    while seen[cur] is not None:
        prev, i = seen[cur]
        path.append(i)
        cur = prev
    path = list(reversed(path))
    return best_state, path
```

```
bf, path = best_with_path([5,2,1,4],4)
bf, path
```

```
((1, 3, 5, 6), [1, 2, 0])
```

[o4-mini-high has come up with the fully correct solution.]

I'm playing with a strategy for choosing the best values for the resulting array. Starting with the smallest element, if there's a second global minimum after it, I include that in a subset (S_0). Otherwise, I just use the smallest element, and this gives a smaller resultant value. The key idea here is comparing the second minimum and local minima, updating my subset as I go, and checking which combination yields the lexicographically smallest sequence. I'll make sure to test it with small cases for confirmation before finalizing.

I'm thinking about confirming my approach with small cases. I'll implement an algorithm that first sorts a list by values and indices, and then iterates through the pairs. For each pair, if the index is greater than the previous one, it's included; otherwise, it's skipped. This helps generate the first set of indices. From there, I'll develop a second set of indices and calculate the final array. Let's test this with brute force for small arrays and see how well it works.

The goal is to keep only the lexicographically minimal states with operations up to n . To derive a formula, I want to start by selecting a suffix (S_0) from an index i , and then add sorted prefix values (plus 1) to form the final result. The idea is that any prefix elements are "pushed back" through operations. For the suffix, we want to append it in the correct order but potentially reorder things on the prefix using operations. Still, the order of the original prefix should remain intact for consistency.

Analyzed

```
# track ids
def simulate(a, path):
    # a: values, idlist: list of original indices, start [0,1,2,3]
    state = list(a)
    ids = list(range(len(a)))
    for op in path:
        i = op
        # remove element at i
        val = state.pop(i)
        id = ids.pop(i)
        # increment value
        val += 1
        # append
        state.append(val)
        ids.append(id)
        print("op", i, "moved id", id, "to end, state:", state, "ids:", ids)
    print("final state:", state, "ids:", ids)

simulate([5,2,1,4], path)
op 1 moved id 1 to end, state: [5, 1, 4, 3] ids: [0, 2, 3, 1]
op 2 moved id 3 to end, state: [5, 1, 3, 5] ids: [0, 2, 1, 3]
op 0 moved id 0 to end, state: [1, 3, 5, 6] ids: [2, 1, 3, 0]
final state: [1, 3, 5, 6] ids: [2, 1, 3, 0]
```

I'm figuring out how to compare the second smallest global value with the local ones. If the second global minimum occurs before the index of the first minimum (i_1), some values could be shifted around in S_0 , leading to different results for the potential new sequence. I'm trying to find out which comparison yields a better outcome — either choosing local values or adjusting to the global ones, depending on their placement relative to each other in the sequence. It's all a balance!

I'm considering how to handle the prefix and suffix. For now, we'll set S_0 as the suffix, but I'm wondering: Could we select some elements from both? Essentially, mix some suffix elements into the prefix? To test this, I'll run brute-force for small arrays ($n \leq 4$). I'll track the minimal lexicographical state using BFS and compare it with suffix-based solutions, using random small arrays. Let's run this code with $n = 3$ arrays and test all suffix candidates to find the best approach.

I want to test the suffix-ANY-order algorithm on random arrays and compare the results. A greedy approach is giving a match to BFS in some cases, but there's a failure for the array $[5, 2, 1, 4]$, where BFS gives one result, and the greedy approach gives another.

To investigate, I'll brute-force the operations and enumerate all reachable states for this array, aiming to identify the fewest steps needed to transform it to the target $[1, 3, 5, 6]$. I'll check the minimal operations path and compare how BFS and greedy paths differ.

I need to verify whether the suffix method outperforms brute force. The plan is to find the minimal lexicographical result by choosing the suffix starting at a certain index, and then comparing that to a brute-force approach. I'll start by brute-forcing all arrays of length 3 with values $\{1, 2, 3\}$, and running BFS up to 3 steps. Each initial configuration will be explored, and I'll compare the minimal results with the suffix-based solution that reorders part of the array. Let's code and test this approach.

Analyzed

```
def advanced_candidate(a):
    n = len(a)
    # sorted pairs by value asc then index asc
    pairs = sorted([(a[i], i) for i in range(n)],
                    key=lambda x: (x[0], x[1]))
    last = -1
    S0 = []
    moved_min_val = float('inf')
```



```

stopped = False
for val, idx in pairs:
    if stopped:
        # all further go to moved; track moved_min_val
        moved_min_val = min(moved_min_val, val)
        continue
    if idx <= last:
        # forced moved
        moved_min_val = min(moved_min_val, val)
    else:
        # candidate can be included or we can stop including
        # skip_value = moved_min_val if moved_min_val != inf else val
        # skip yields c[t] = skip_value + 1
        skip_source_val = moved_min_val
        if moved_min_val != float('inf') else val
        if val < skip_source_val + 1:
            # include
            S0.append(idx)
            last = idx
        else:
            # skip: stop including further
            stopped = True
            moved_min_val = min(moved_min_val, val)
# Now S1 is all except S0
S0_set = set(S0)
S1 = [i for i in range(n) if i not in S0_set]
# produce c: S0 in index order then S1 sorted by a[i]+1, tie by index
S0_sorted = sorted(S0)
S1_sorted = sorted(S1, key=lambda i:(a[i]+1, i))
c = [a[i] for i in S0_sorted] + [a[i]+1 for i in S1_sorted]
return tuple(c)

```

```

# Test advanced_candidate on [5,2,1,4]
print(advanced_candidate([5,2,1,4]), best_final([5,2,1,4],4))

```

```

# test random arrays
fails2 = []
for _ in range(200):
    n=4
    a = [random.randint(1,5) for _ in range(n)]
    bf = best_final(a, n)
    c = advanced_candidate(a)
    if bf != c:
        fails2.append((tuple(a), bf, c))
        print("fail", a, bf, c)
        break
len(fails2)

```

```

(1, 3, 5, 6) (1, 3, 5, 6)
fail [4, 1, 1, 5] (1, 1, 5, 5) (1, 1, 5, 6)
1

```

Analyzed

```

print(best_with_path([4,1,1,5],4))

```

```

((1, 1, 5, 5), [0])

```

If no elements have been moved so far and we stop at a candidate, does it mean there's no S_0 at this position? In my scenario, S_0 is ['some'] and the next candidate is considered. If no element has been moved except forced ones, we might skip certain candidates. We track the minimum value among moved items with *moved_min_val*. If there's no force, the remaining elements go into S_1 , and the minimum among S_1 values becomes the new *moved_min_val*.

Analyzed

```
def advanced_candidate2(a):
    n = len(a)
    pairs = sorted([(a[i], i) for i in range(n)],
                    key=lambda x: (x[0], x[1]))
    last = -1
    S0 = []
    moved_min_val = float('inf')
    stopped = False
    for val, idx in pairs:
        if stopped:
            moved_min_val = min(moved_min_val, val)
            continue
        if idx <= last:
            moved_min_val = min(moved_min_val, val)
        else:
            skip_src_val = moved_min_val if
            moved_min_val != float('inf') else val
            # include if val <= skip_src_val + 1
            if val <= skip_src_val + 1:
                S0.append(idx)
                last = idx
            else:
                stopped = True
                moved_min_val = min(moved_min_val, val)
    S0_sorted = sorted(S0)
    S1 = [i for i in range(len(a)) if i not in S0_sorted]
    S1_sorted = sorted(S1, key=lambda i: (a[i]+1, i))
    c = [a[i] for i in S0_sorted] + [a[i]+1 for i in S1_sorted]
    return tuple(c)
```

```
# Test on previous fails
print(advanced_candidate2([5,2,1,4]), best_final([5,2,1,4],4))
print(advanced_candidate2([4,1,1,5]), best_final([4,1,1,5],4))
```

```
# test random
fails2 = []
for _ in range(500):
    n=4
    a = [random.randint(1,5) for _ in range(n)]
    bf = best_final(a, n)
    c = advanced_candidate2(a)
    if bf != c:
        fails2.append((tuple(a), bf, c))
        print("fail", a, bf, c)
        break
len(fails2)
```

```
(1, 3, 5, 6) (1, 3, 5, 6)
(1, 1, 5, 5) (1, 1, 5, 5)
0
```

Analyzed

```
# test random for n=5
fails3 = []
for _ in range(50):
    n=5
    a = [random.randint(1,5) for _ in range(n)]
    bf = best_final(a, n)
    c = advanced_candidate2(a)
    if bf != c:
        fails3.append((tuple(a), bf, c))
        print("fail", a, bf, c)
        break
len(fails3)
```

0

The final candidate set is [1,1,5,6], but BFS found [1,1,5,5]. This happened because BFS included `idx0` and skipped `idx3` after calculating skip values. Meanwhile, the `advanced_candidate` approach skipped val 5 at `idx3`, but in theory, including it could open up better options later. So, the question is: should I allow equality (i.e., $val \leq skip + 1$) in the inclusion decision? This would help resolve ties by lexicographically comparing subsequent steps, potentially including `idx3`.