HumanEval-V: Benchmarking High-Level Visual Reasoning with Complex Diagrams in Coding Tasks

Anonymous ACL submission

Abstract

Understanding and reasoning over diagrams is a fundamental aspect of human intelligence. While Large Multimodal Models (LMMs) have demonstrated impressive capabilities across various tasks, existing benchmarks lack comprehensive evaluation of their diagram interpretation and reasoning abilities, particularly in coding contexts. We present HumanEval-V, a rigorous benchmark of human-annotated coding tasks that spans six task types and evaluates diverse visual reasoning capabilities. Each task features carefully crafted diagrams paired with function signatures and test cases, employing novel code generation tasks to thoroughly assess models' diagram comprehension. Through extensive experiments with 22 LMMs, we find that even top-performing models achieve modest success rates, with Claude 3.5 Sonnet reaching only 36.8% pass@1, highlighting substantial room for improvement. Our analysis reveals that current LMMs struggle with spatial transformations, topological relationships, and dynamic patterns that humans find intuitive. These findings provide valuable insights for advancing LMMs' visual reasoning abilities.[1]

1 Introduction

High-level intelligence, whether in humans or advanced AI systems, requires the ability to understand and reason over visual information represented in diagrams. Diagrams are essential in many domains, including science, engineering, and mathematics, as they serve as a powerful medium for abstracting and communicating complex data, relationships, and processes, encoding rich information in a visual and structured format. The abilities required to comprehend diagrams extend beyond simple pattern recognition; they necessitate sophisticated cognitive capabilities, including interpreting transformation patterns, recognizing hierarchical structures, and integrating multiple visual cues

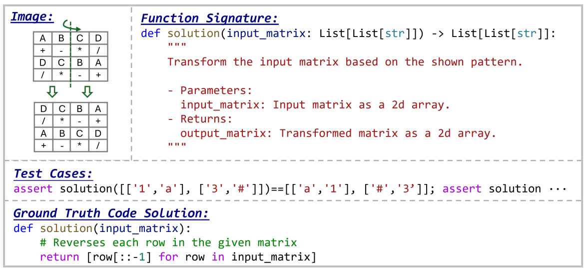


Figure 1: A task example from HumanEval-V. LMMs are required to figure out the facts and patterns in the diagram and complete the function body.

such as arrows, symbols, and their relative positions to perform spatial or logical reasoning. The rapid development of Large Multimodal Models (LMMs) has led to the creation of various benchmarks designed to assess the alignment between LMMs' capabilities and human intelligence. However, there remains a significant gap in benchmarks that specifically evaluate the ability to understand and reason over complex diagrams.

Popular multimodal benchmarks, such as MMMU (Yue et al., 2024), MathVista (Lu et al., 2023), and ChartQA (Masry et al., 2022), focus primarily on scientific, mathematical, and chart-based analytical questions over various domains of images, testing LMMs' multidiscipline knowledge rather than diagram understanding. While abstract visual reasoning tasks (Zhang et al., 2019; Nie et al., 2020) from IQ tests typically feature static patterns based on visual analogies or numerical inference over diagrams, they often lack the complexity and diversity in visual patterns and diagram types. This gap highlights the need for benchmarks that assess more intricate diagram reasoning abilities. The field of coding presents an underexplored opportunity, as developers frequently use various diagrams to illustrate data structures, algorithms, and problem constraints. A recent study, MMCode (Li et al., 2024b), evaluated LMMs on coding prob-

[1] Data and code available at this anonymous repository

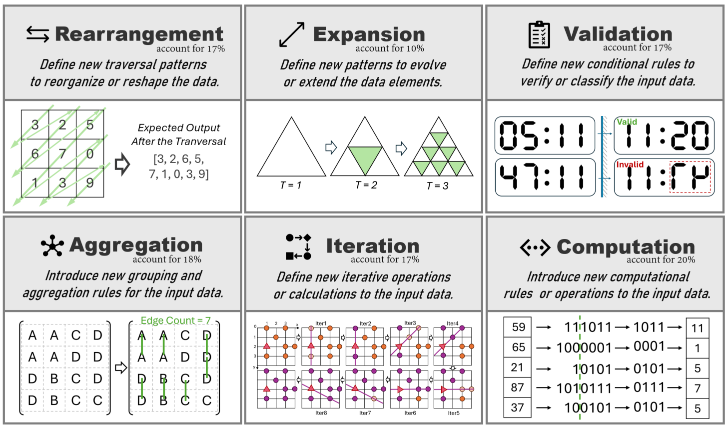


Figure 2: The six task categories in HumanEval-V, along with their quantitative distribution and representative diagram examples.

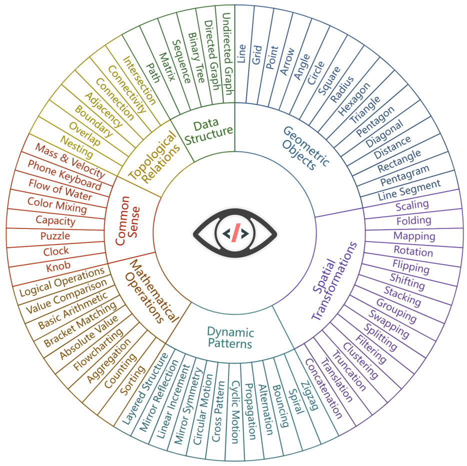


Figure 3: Core knowledge required for understanding diagrams in HumanEval-V.

lems with visual contexts directly crawled from competition platforms. However, competition coding problems often include comprehensive textual descriptions, making the visual information supplementary. Their results, which showed similar LMM performance with and without visual information, further underscore the gap in evaluating genuine diagram understanding abilities.

To address this gap, we introduce HumanEval-V, a novel benchmark designed to provide a focused evaluation of complex diagram understanding and reasoning abilities in programming contexts. Unlike MMCode, our benchmark is dedicated to assessing visual capabilities through a rigorous annotation pipeline that creates coding tasks capturing the essence of real-world problems. Each task features an *indispensable*, *self-explanatory* diagram with minimal textual clues, as demonstrated by our experiments where top LMMs failed all tasks without the provided diagrams. HumanEval-V consists of 253 human-annotated coding tasks. Each task features (1) a diagram encoding the problem context, (2) a function signature defining the task's input-output structures, and (3) test cases to verify solution correctness. Figure 1 provides an example task, where the diagram illustrates spatial transformation patterns, requiring the model to comprehend fine-grained visual elements such as matrices, arrow directions, and spatially ordered data points. This task aligns with the ARC-AGI (Chollet, 2019) benchmark in inferring transformation patterns from limited visual examples. However, unlike ARC's matrix-formatted diagrams, HumanEval-V offers a more diverse and complex set of diagrams spanning six task types (Figure 2), demanding versatile capabilities (Figure 3) for diagram understanding and reasoning. For a comparison of diagrams from existing benchmarks and ours, see Figures 21 and 22.

Another novelty of HumanEval-V lies in using code generation tasks for evaluation instead of the multiple-choice or short-answer questions commonly used in other multimodal benchmarks. This approach offers compelling benefits: code generation is more challenging, requiring comprehensive logical thinking and visual understanding with minimal chance of correct guesses, and test cases could rigorously verify whether the model captures all critical visual information, rather than relying on similarity matching with ground truth. Additionally, we utilize a two-stage evaluation pipeline that supports LMMs with limited coding abilities by first prompting them to generate a structured diagram description summarizing the visual context, then using a more capable coder model to implement the solution, ensuring the evaluation prioritizes visual understanding over coding proficiency.

Through extensive experiments with 22 LMMs, we observe the following key findings: (1) Our benchmark presents unique challenges not addressed by other multimodal benchmarks. The top-performing model, Claude 3.5 Sonnet, achieves 36.8% pass@1, while the best open-weight model, Pixtral 124B, reaches 21.3%. (2) Current LMMs exhibit stronger vision-to-language alignment than vision-to-code. Their best performance occurs when they serve as diagram describers, with GPT-4o acting as the coder model. (3) LMMs' performance can be further enhanced through sampling or iterative self-refinement. For instance, Claude

3.5 Sonnet achieves a 74.3% pass rate with 100 samples, and it can reach 55.3% pass@1 with four self-refining iterations based on test case execution feedback. (4) Current LMMs still have difficulty understanding diagrams that are trivial for humans, particularly understanding spatial transformations, topological relationships, and dynamic patterns.

2 Benchmark Construction

Task Definition: As shown in Figure 1, each coding task in `HumanEval-V` includes: (1) a single diagram D providing visual context, (2) a Python function signature σ with input parameters, return type, and brief instructions, and (3) a set of test cases $T = t_1, t_2, \ldots, t_n$ to validate the correctness of the generated output O, a complete Python function produced by the LMM.

Annotation Standards: We establish rigorous standards to ensure high-quality coding tasks: (1) The visual context must be essential for solving the task, with all relevant information contained in a single image; (2) Tasks should be designed around the visual context with minimal textual description; (3) Unnecessary programming complexities, such as recursion, intricate constraints, or complex data structures, should be avoided.

Task Annotation Pipeline: We define the task annotation pipeline with *four steps* as in Figure 4.

*In the **first** step*, we collect a large set of coding problems from prominent Q&A and coding challenge platforms that incorporate images, then screen them to exclude questions that (1) require specific programming frameworks or libraries, or contain images that (2) are not illustrative diagrams, (3) provide no useful information for solving the problem, or (4) require extensive textual context for interpretation. *For the **second** step*, we distill the screened problems to identify critical visual elements, along with the data structures, operations, transformations, or conditional rules involved, categorizing them into six task types (Figure 2) and outlining the key capabilities needed for diagram understanding. *For the **third** step*, we design new coding tasks based on these distilled ideas, eliminating unnecessary complexities, refining input/output structures and function signatures, and creating visual objects and layouts using basic shapes and a consistent color scheme in PowerPoint. We also generate tailored test cases, resulting in ***100*** newly crafted *seed tasks* after excluding tasks with design or formulation challenges. *For the **fourth** step*, we expand the task set by diversifying the seed tasks using a hybrid approach with GPT-4o. GPT-4o identifies relevant capability aspects for each seed task and suggests modifications involving new spatial transformations, mathematical operations, dynamic patterns, or variations in data structures and object attributes. Human annotators then refine and annotate these new tasks and diagrams, creating 0 to 2 diversified versions per seed task based on complexity, culminating in ***253*** tasks in `HumanEval-V` and finalizing the capability aspects shown in Figure 3. Further details and examples of the data collection and diversification processes are provided in Appendix B.1 and Appendix B.2.

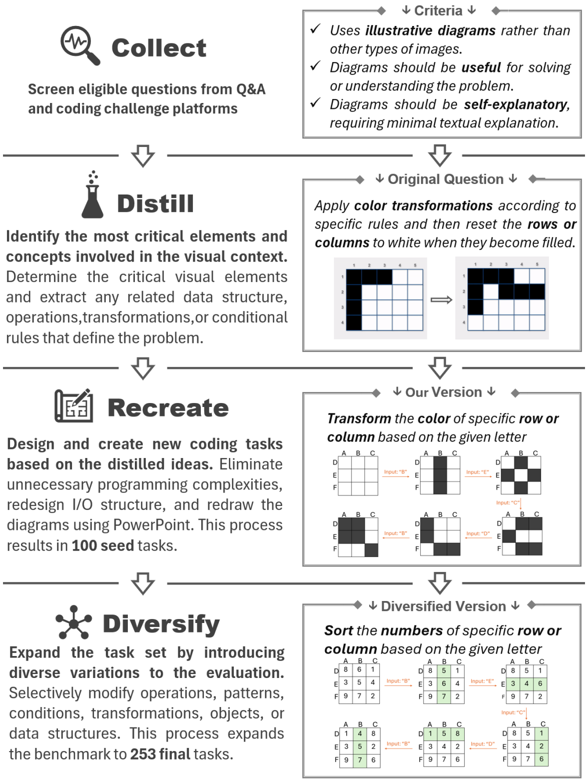
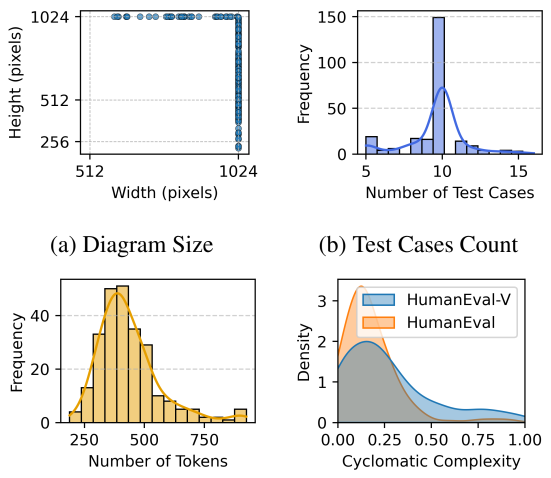


Figure 4: `HumanEval-V` task construction pipeline.

Quality Assurance: Our annotation team comprises four experienced programmers, each with over four years of Python programming experience. Initially, each annotator independently annotates their assigned tasks following pre-defined guidelines. Subsequently, all annotators review each other's work by annotating ground truth code solutions and diagram descriptions to ensure tasks are visually grounded, solvable with the provided information, and free of design or conceptual errors. Any identified issues are resolved collaboratively,



(a) Diagram Size (b) Test Cases Count

(c) Description Length (d) Coding Complexity

Figure 5: Distribution analysis of the benchmark data.

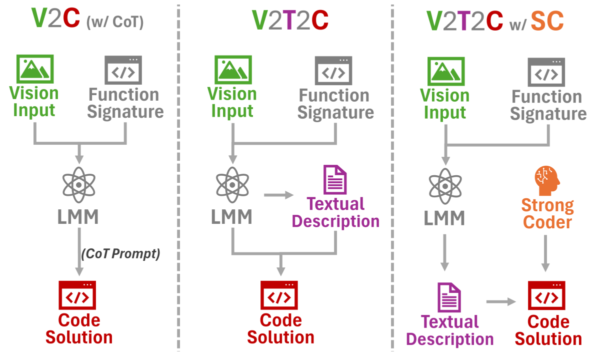


Figure 6: Evaluation pipelines employed in the experiments: (1) Direct translation of visual context into code (V2C), (2) with an optional Chain-of-Thought prompt (V2C w/ CoT); (3) Translation of visual context into a textual description, which is then processed to generate code (V2T2C); and (4) A variant of the third pipeline, where a stronger coder model is used to generate the code solution (V2T2C w/ SC).

with tasks finalized only after consensus is reached. Additionally, one annotator ensures consistent formatting and style across all visual representations and coding tasks. Each annotator contributes over 200 hours to the annotation process.

Benchmark Statistics: To further demonstrate the quality of our benchmark, we conduct statistical analyses on several key aspects and present the distribution of these statistics in Figure 5. First, we strictly control diagram sizes, capping the maximum width or height at 1024 pixels to eliminate the need for high-resolution perception. Second, each task includes at least five test cases, with the majority containing ten, ensuring full statement and branch coverage over human-annotated code solutions. Third, the token length of human-annotated diagram descriptions is predominantly around 400 (measured using tiktoken (OpenAI, 2024c)), demonstrating that our diagrams encapsulate rich visual context. Lastly, our human-annotated code solutions exhibit cyclomatic complexity (Gill and Kemerer, 1991) levels comparable to HumanEval (Chen et al., 2021), a widely used coding benchmark designed for entry-level programming tasks.

3 Benchmarking Setup

Models: We evaluate 22 state-of-the-art LMMs, including a representative mix of leading proprietary and open-weight models. Our evaluation covers five of the latest proprietary models: Claude 3.5 Sonnet, GPT-4o, GPT-4o-mini, Gemini 1.5 Pro, and Gemini 1.5 Flash. We also assess 17 top-performing open-weight models spanning various parameter sizes, including InternVL 2.5 (4/8/26/78B), Qwen2-VL (7/72B), Pixtral (12/124B), LLaVA-OV (7/72B), Llama-3.2-V (11/90B), Molmo-D (7/72B), Chameleon (7/30B), and Phi-3.5-V (4B). Further details are in Table 4.

Prompting: we employ multiple strategies for our evaluation pipelines as illustrated in Figure 6, where LMMs may encounter four different prompting scenarios: (1) *Direct Code Generation*. The model directly generates code based on the given diagram D and function signature σ, denoted as $P_{V2C}(D, \sigma)$; (2) *Chain of Thought (CoT)*. This variant enhances the V2C pipeline by incorporating a zero-shot CoT instruction I_{CoT} (Wei et al., 2022), prompting the model to outline its reasoning process before generating the code. This is denoted as $P_{V2C}(D, \sigma, I_{CoT})$; (3) *Intermediate Textual Representation*. The model first produces a structured textual ***problem specification*** PS based on D and σ, denoted as $P_{V2T}(D, \sigma)$. The problem specification consists of three key sections: *Problem Restatement*, *Visual Facts*, and *Visual Patterns*. This structured representation is derived from our benchmark annotation process, which we found to be effective in capturing a comprehensive description of the problem context; (4) *Code Generation from Text*. The model generates code based on PS rather than the original diagram D, denoted as $P_{T2C}(PS, \sigma)$. The corresponding prompt templates are shown in Figure 27, with further details on prompt design available in Appendix C.2.

Models	V2C		V2C w/ CoT		V2T2C		V2T2C w/ GPT-4o	
pass@k	k=1	k=3	k=1	k=3	k=1	k=3	k=1	k=3 🏅
Proprietary LMMs								
Claude 3.5 Sonnet	**28.1**	**37.9**	**36.8**$_{\uparrow 8.7}$	**47.9**$_{\uparrow 10.0}$	**33.2**$_{\uparrow 5.1}$	**43.6**$_{\uparrow 5.7}$	**31.6**$_{\uparrow 3.5}$	**43.7**$_{\uparrow 5.8}$
GPT-4o	24.1	33.8	27.7$_{\uparrow 3.6}$	40.0$_{\uparrow 6.2}$	26.5$_{\uparrow 2.4}$	40.5$_{\uparrow 6.7}$	26.5$_{\uparrow 2.4}$	40.5$_{\uparrow 6.7}$
Gemini 1.5 Pro	23.3	26.9	22.9$_{\downarrow 0.4}$	34.1$_{\uparrow 7.2}$	28.5$_{\uparrow 5.2}$	36.4$_{\uparrow 9.5}$	26.9$_{\uparrow 3.6}$	37.3$_{\uparrow 10.4}$
Gemini 1.5 Flash	15.4	20.5	17.4$_{\uparrow 2.0}$	24.9$_{\uparrow 4.4}$	15.8$_{\uparrow 0.4}$	22.0$_{\uparrow 1.5}$	18.6$_{\uparrow 3.2}$	27.2$_{\uparrow 6.7}$
GPT-4o-mini	9.90	16.0	15.8$_{\uparrow 5.9}$	21.1$_{\uparrow 5.1}$	14.2$_{\uparrow 4.3}$	22.7$_{\uparrow 6.7}$	18.2$_{\uparrow 8.3}$	24.6$_{\uparrow 8.6}$
Open-weight LMMs with more than 70B parameters								
Pixtral 124B	**12.6**	**20.3**	**16.6**$_{\uparrow 4}$	**28.1**$_{\uparrow 7.8}$	**21.3**$_{\uparrow 8.7}$	**29.9**$_{\uparrow 9.6}$	21.3$_{\uparrow 8.7}$	**31.6**$_{\uparrow 11.3}$
InternVL 2.5 78B	12.3	19.7	13.4$_{\uparrow 1.1}$	27.3$_{\uparrow 7.6}$	17.8$_{\uparrow 5.5}$	25.7$_{\uparrow 6}$	**21.7**$_{\uparrow 9.4}$	31.4$_{\uparrow 11.7}$
Qwen2 VL 72B	9.10	15.7	14.2$_{\uparrow 5.1}$	19.4$_{\uparrow 3.7}$	10.7$_{\uparrow 1.6}$	19.1$_{\uparrow 3.4}$	16.6$_{\uparrow 7.5}$	25.1$_{\uparrow 9.4}$
LLaVA-OV 72B	6.70	7.70	6.30$_{\downarrow 0.4}$	11.4$_{\uparrow 3.7}$	10.7$_{\uparrow 4}$	13.1$_{\uparrow 5.4}$	13.8$_{\uparrow 7.1}$	19.7$_{\uparrow 12}$
Molmo-D 72B	3.20	4.80	3.20$_{\uparrow 0.0}$	8.80$_{\uparrow 4.0}$	1.60$_{\downarrow 1.6}$	7.00$_{\uparrow 2.2}$	5.10$_{\uparrow 1.9}$	14.2$_{\uparrow 9.4}$
Llama-3.2-V 90B	4.30	6.10	4.00$_{\downarrow 0.3}$	8.20$_{\uparrow 2.1}$	5.90$_{\uparrow 1.6}$	10.9$_{\uparrow 4.8}$	4.70$_{\uparrow 0.4}$	11.0$_{\uparrow 4.9}$
Open-weight LMMs with fewer than 70B parameters								
Pixtral 12B	**4.0**	5.9	**6.3**$_{\uparrow 2.3}$	**12.2**$_{\uparrow 6.3}$	**5.5**$_{\uparrow 1.5}$	**12.6**$_{\uparrow 6.7}$	**13.8**$_{\uparrow 9.8}$	**21.3**$_{\uparrow 15.4}$
InternVL 2.5 26B	3.6	**6.6**	4.3$_{\uparrow 0.7}$	6.8$_{\uparrow 0.2}$	2.8$_{\downarrow 0.8}$	6.7$_{\uparrow 0.1}$	8.3$_{\uparrow 4.7}$	16.7$_{\uparrow 10.1}$
Qwen2 VL 7B	0.8	3.3	1.6$_{\uparrow 0.8}$	3.9$_{\uparrow 0.6}$	2.4$_{\uparrow 1.6}$	6.3$_{\uparrow 3.0}$	6.3$_{\uparrow 5.5}$	14.7$_{\uparrow 11.4}$
InternVL 2.5 8B	0.8	2.0	0.8$_{\uparrow 0.0}$	3.7$_{\uparrow 1.7}$	1.2$_{\uparrow 0.4}$	3.3$_{\uparrow 1.3}$	5.1$_{\uparrow 4.3}$	13.6$_{\uparrow 11.6}$
InternVL 2.5 4B	1.2	4.1	3.2$_{\uparrow 2.0}$	3.4$_{\downarrow 0.7}$	3.2$_{\uparrow 2.0}$	5.8$_{\uparrow 1.7}$	5.9$_{\uparrow 4.7}$	13.5$_{\uparrow 9.4}$
LLaVA-OV 7B	2.0	1.9	1.6$_{\downarrow 0.4}$	2.4$_{\uparrow 0.5}$	2.0$_{\uparrow 0.0}$	3.2$_{\uparrow 1.3}$	5.1$_{\uparrow 3.1}$	10.2$_{\uparrow 8.3}$
Phi-3.5-V 4B	0.0	0.0	0.0$_{\uparrow 0.0}$	0.0$_{\uparrow 0.0}$	0.0$_{\uparrow 0.0}$	0.0$_{\uparrow 0.0}$	5.9$_{\uparrow 5.9}$	9.40$_{\uparrow 9.4}$
Llama-3.2-V 11B	2.0	2.0	1.6$_{\downarrow 0.4}$	3.9$_{\uparrow 1.9}$	2.0$_{\uparrow 0.0}$	5.2$_{\uparrow 3.2}$	4.0$_{\uparrow 2.0}$	8.80$_{\uparrow 6.8}$
Molmo-D 7B	1.2	1.0	0.4$_{\downarrow 0.8}$	1.8$_{\uparrow 0.8}$	0.8$_{\downarrow 0.4}$	1.0$_{\uparrow 0.0}$	2.8$_{\uparrow 1.6}$	8.40$_{\uparrow 7.4}$
Chameleon 7B	0.0	0.0	0.0$_{\uparrow 0.0}$	0.2$_{\uparrow 0.2}$	0.0$_{\uparrow 0.0}$	0.0$_{\uparrow 0.0}$	1.2$_{\uparrow 1.2}$	2.20$_{\uparrow 2.2}$
Chameleon 30B	0.0	0.0	0.0$_{\uparrow 0.0}$	0.2$_{\uparrow 0.2}$	0.4$_{\uparrow 0.4}$	0.0$_{\uparrow 0.0}$	0.0$_{\uparrow 0.0}$	1.90$_{\uparrow 1.9}$

Table 1: Performance of LMMs across different settings. Models are ranked based on the 🏅 column. The **best** and second-best performances in each column are highlighted. The numerical values are color-coded to indicate performance changes relative to the corresponding pass@k values in the **V2C** column: green represents improvement, red indicates decline, and yellow denotes minimal change.

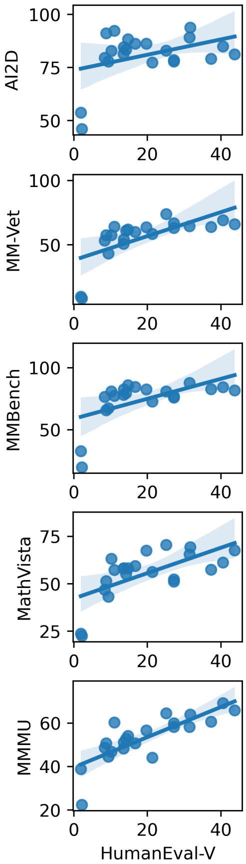


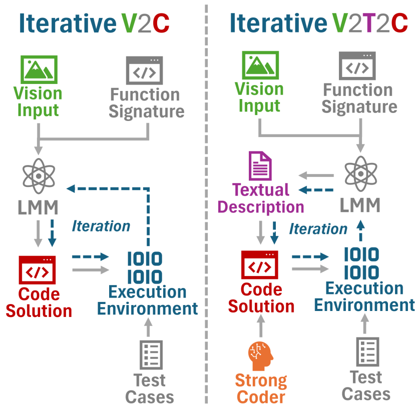
Figure 7: Comparison of LMM performance on `HumanEval-V` and other popular multimodal benchmarks.

Hyper-parameters & Post-processing: We apply two distinct decoding strategies for both code and description generation. First, we use greedy decoding to produce a single deterministic output, assessing model performance in a constrained setting. Additionally, we employ a sampling method with $Top\_p = 0.95$, $Top\_k = 20$, and a temperature of 0.8 to generate diverse outputs, allowing us to evaluate the models' ability to produce correct solutions when given multiple attempts. We set the maximum output length to 2048 tokens for both code and description generation. To facilitate extraction, we prompt the models to encapsulate their generated code within Markdown-style code blocks. We then apply an abstract syntax tree parser to detect and retrieve generated import statements, class definitions, and function definitions. These components are concatenated to form the final code solution. An additional ablation study on the temperature setting is presented in Appendix C.3.

Evaluation Metrics Following established practices in code generation evaluation (Chen et al., 2021, 2022), we use the pass@k metric to assess functional correctness. A task is considered solved if at least one of the k selected solutions passes all test cases, and pass@k is the percentage of solved tasks. We report results for $k = 1, 3$. In the *V2C* setting, we generate n code samples per task and randomly select k for evaluation. For greedy decoding, $n = 1$ for pass@1, while for sampling-based evaluation, $n = 6$ for pass@3. In the *V2T2C* setting, we first sample six problem specifications per task, then use greedy decoding to generate one code solution per *PS*, resulting in six solutions per task for pass@3 computation.

4 Benchmarking Results

Main Results: We present the benchmarking results of 22 LMMs in Table 1, covering the four evaluation pipelines introduced in Figure 6. Additionally, Figure 7 provides a correlation analy-



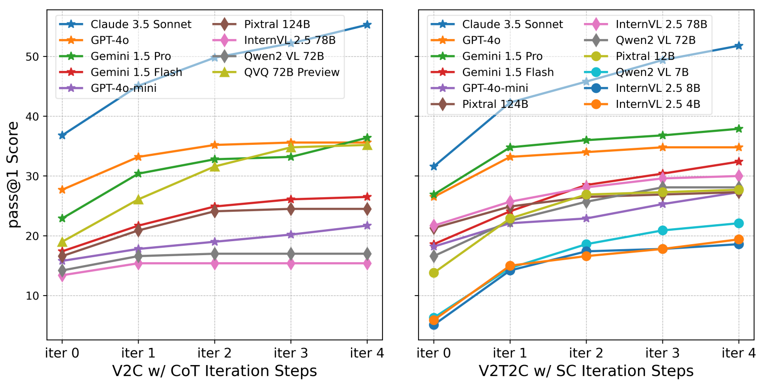


Figure 8: Iterative evaluation pipelines. Figure 9: Performance of LMMs under the iterative evaluation settings.

sis to illustrate the performance gap between the evaluated LMMs on `HumanEval-V` and other popular benchmarks (more details on the correlation analysis are in Appendix A.1). Based on these results, we highlight the following key findings: *(1) Our benchmark presents unique challenges not captured by other benchmarks.* As shown in Figure 7, most evaluated LMMs exhibit significantly larger performance gaps on `HumanEval-V` compared to other benchmarks. While MMMU demonstrates the highest correlation with our benchmark, its results still lack sufficient discrimination between models. *(2) LMMs generally achieve their best performance under the V2T2C w/ GPT-4o setting.* This is particularly evident for LMMs with fewer than 70B parameters, which struggle to complete tasks in the *V2C* setting. These findings validate the importance of decoupling visual understanding from coding abilities. Additionally, CoT prompting and the decoupled V2T2C pipeline show similar performance distributions, with more capable LMMs benefiting more from these enhancements than smaller models. *(3) Open-weight LMMs still lag behind top proprietary models.* Although high-capacity open-weight LMMs (e.g., Pixtral 124B) outperform the mini/flash versions of proprietary models, they still fall short of the most capable proprietary LMMs. For smaller-scale models, Pixtral 12B, Qwen2 VL 7B, and InternVL 2.5 4B demonstrate a high performance-to-size ratio. *(4) Certain LMMs exhibit anomalously poor performance.* Models such as Molmo-D, Llama-3.2-V, and the Chameleon series perform significantly worse than other LMMs of similar scale. Another case is Phi-3.5-V, which appears to lack coding ability, achieving a performance score of 0 in the *V2C* settings, compared to 9.4% pass@3 when assisted by GPT-4o for code generation. *(5) Additional Results of o1 and QVQ:* We also evaluated reasoning-enhanced LMMs that leverage test-time scaling by generating long chain-of-thought (CoT) reasoning. Specifically, we assessed OpenAI o1 (OpenAI, 2024b) and QVQ-72B-Preview (Team, 2024b) under the *V2C w/ CoT* setting, achieving pass@1 scores of 40.6% and 19.0%, respectively. Our case study reveals that both models still struggles with visual understanding, often failing to identify rules or patterns in the diagrams. Meanwhile, QVQ primarily fails due to excessively long CoT reasoning, with 35% of cases unable to generate a valid code solution within the 20k token limit. These results underscore the complexity of the diagrams in our benchmark. Example cases for o1 and QVQ are shown in Figures 34, 39, 44 and Figures 35, 40, 45.

Iterative Benchmarking: We introduce an iterative benchmarking pipeline to evaluate LMMs' ability to reason over environmental feedback and perform self-refinement—an essential skill for real-world problem-solving. Figure 8 illustrates two types of iterative pipelines derived from the *V2C* and *V2T2C w/ SC* settings. In these pipelines, LMMs must refine either their generated code solutions or textual descriptions based on feedback from the execution environment. To support this process, we design new prompt templates (Figure 28) that guide the refinement steps. Specifically, for each task, LMMs perform an additional iteration if the generated code contains syntax errors or fails to pass all test cases. The feedback includes detailed error messages or the failed test cases' inputs and expected outputs.

For the iterative evaluation, we select the most capable LMMs across different parameter scales,

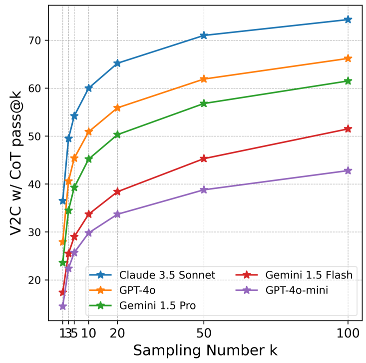


Figure 10: Performance with increased sample size.

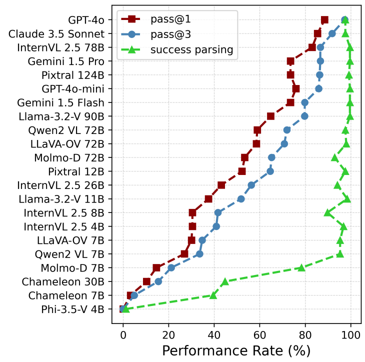


Figure 11: LMMs' performance with human problem specifications.

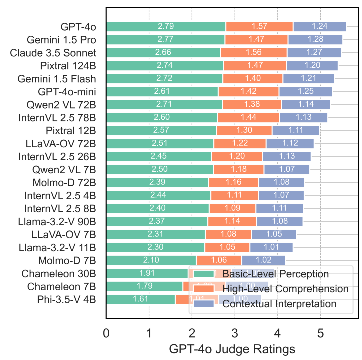


Figure 12: LLM-as-Judge ratings for LMMs in the *V2T2C* setting.

using greedy decoding and the pass@1 metric. Figure 9 presents the results, where *iter 0* represents the first round of generation without feedback. We observe that LMMs generally improve across iterations, with more capable models achieving larger performance gains. Notably, some models, such as Claude and QVQ, exhibit stronger self-refinement capabilities. We also investigate the cases that are corrected after iterations and find that approximately 90% of these cases are corrected due to the models' improved understanding of the diagram and task. The remaining 10% are cases that fix edge conditions highlighted by the test case feedback. None of these corrections result from hard-coding the exposed test cases into the code solutions.

5 Experimental Analysis

This section presents our analysis of model performance under various settings, including increased sampling sizes, human-annotated problem specifications, the use of GPT-4o judge for rating LMMs' diagram descriptions, and error pattern analysis. Our goal is to examine both the potential and limitations of the LMMs on `HumanEval-V`. Further analysis is provided in Appendix D, where we explore the co-occurrence of capability aspects required in our benchmark tasks, the stability of using QwenCoder-32B as a strong coder instead of GPT-4o, and experimental evidence supporting the value of tasks diversified from the seed tasks.

Performance with Increased Sample Size: We scale up the number of samples for five proprietary LMMs to explore their potential performance. We increase the sampling number n to 200, using the same Top_p, Top_k, and max token limitations outlined in Section 3 to calculate the pass@100 score under the *V2C w/ CoT* setting. As shown in Figure 10, we observe a consistent performance improvement across all models with larger sample sizes. Notably, Claude 3.5 Sonnet achieves a significant improvement, reaching 74.3% pass@100, underscoring the strong potential for these models when scaling up sample sizes.

Coding Performance with Human-Annotated Problem Specifications: We evaluate all LMMs on a new task where they *generate code based on human-annotated problem specifications*, without direct access to the diagrams. This setup isolates their ability to perform visual reasoning and generate code. We also calculate the *success parsing rate* using Pylint (Wikipedia, 2024), which measures the syntactic correctness of the generated code, independent of its functional accuracy. The results, presented in Figure 11, show that *most LMMs demonstrate strong coding capabilities*, generally outperforming their best results from Table 1. Notably, GPT-4o achieves 96.5% pass@3, a significant improvement over its 40.5% pass@3 in the *V2T2C* setting. Smaller models, such as InternVL 2.5 4B, also show substantial improvement.

We also evaluate a setting where LMMs generate code based solely on the function signature, without access to diagrams or descriptions, and find that *none of the five proprietary models are able to pass any tasks*. This underscores the necessity of visual context in our benchmark. These results suggest that current LMMs face more challenges in visual reasoning than coding on `HumanEval-V`.

LLM-as-Judge Ratings: We evaluate the problem specifications (PS) generated by LMMs in the *V2T2C* setting using GPT-4o as the judge. GPT-4o rates the PS in three dimensions: Basic-Level Perception (identifying basic visual elements), High-Level Comprehension (understanding objects, patterns, transformations, and operations), and Contextual Interpretation (clear description without vagueness or hallucinations) as outlined in the prompt template shown in Figure 29. Ratings are on a 1-3 scale, where 1 indicates severe errors and 3 reflects near perfection in the capability dimension. The results, shown in Figure 12, reveal that while LMMs generally excel in basic perceptual abilities, they struggle with high-level comprehension and clarity of expression. Notably, the performance gap between models is small. We also find the difference in ratings between passed and failed tasks is minimal. For example, GPT-4o scores 2.9, 2.0, and 1.3 across the three dimensions on passed tasks, compared to average ratings of 2.8, 1.6, and 1.2 across all tasks, highlighting the limitations of using LLM-as-judge as an evaluation tool. This lack of robustness may stem from rigid comparisons to human-annotated PS, further emphasizing the importance of pass rates as the evaluation metric.

Error Analysis: We conducted a comprehensive error analysis to understand the limitations of current LMMs in `HumanEval-V`, as detailed in Appendix E. Our analysis examined correlations between model performance and three key factors: task types, general capability dimensions, and specific capability aspects. The results reveal that LMMs particularly struggle with tasks involving *Transformation* and *Iterative Calculation*. And models show notable difficulties with specific capabilities such as understanding dynamic patterns (e.g., spirals, circular arrangements) and spatial transformations (e.g., stacking, translation). Interestingly, our investigation of task difficulty metrics shows that LMM performance correlates poorly with human-perceived difficulty measures, including both programming complexity (measured by cyclomatic complexity) and visual comprehension difficulty (measured by description length). This suggests a fundamental gap in LMMs' visual reasoning capabilities, where even tasks considered trivial by humans can prove challenging for state-of-the-art models. For concrete examples of these challenges, we present representative error cases in Figures 31 to 47.

6 Related Work

Benchmarks Involving Diagrams: Prior work on multimodal benchmarks can be categorized into several groups: (1) General-purpose multimodal evaluation benchmarks (Yue et al., 2024; Liu et al., 2023; Yu et al., 2023; Li et al., 2023; Ying et al., 2024; Chen et al., 2024a) that assess models' broad multidisciplinary capabilities; (2) Scientific diagram understanding (Lu et al., 2022; Kembhavi et al., 2016); (3) Mathematical visual reasoning (Lu et al., 2023; Wang et al., 2024a; Zhang et al., 2024); (4) Data visualization comprehension (Masry et al., 2022; Wang et al., 2024c; Chollet, 2019) that focus on plots and charts; (5) Abstract reasoning (Zhang et al., 2019; Jiang et al., 2024; Nie et al., 2020; Chia et al., 2024); and (6) Specialized diagram understanding including abstract symbol interpretation and geometric spatial reasoning (Lu et al., 2021; Rahmanzadehgervi et al., 2024). While these benchmarks cover various aspects of visual understanding, they do not address the complex diagrams in the coding context.

Multimodal Code Generation: Recent work in multimodal code generation has focused on two main Categories. In the first category, researchers have explored derendering web pages into functional code (Si et al., 2024; Laurençon et al., 2024) and converting scientific figures into their corresponding plotting code (Shi et al., 2024; Wu et al., 2024). The second category includes Program-based VQA approaches, where models leverage pre-defined modules to answer visual questions (Surís et al., 2023; Subramanian et al., 2023). MMCode (Li et al., 2024b) is the most related coding benchmark to ours, evaluating LMMs' coding abilities using problems with visual demonstrations from competition platforms. However, our benchmark differs in its dedicated focus on assessing the visual capabilities of LMMs. We provide a detailed discussion in Appendix F, highlighting the differences between `HumanEval-V` and MMCode in terms of visual indispensability, task complexity, and evaluation design.

7 Conclusion

In this paper, we introduced `HumanEval-V`, a novel benchmark designed to evaluate LMMs' capabilities in understanding and reasoning over diagrams in programming contexts. Through comprehensive experiments, we demonstrated that current LMMs, while showing promising performance, still face significant challenges in complex diagram understanding and reasoning. Our extensive experimental results and analysis provide valuable insights for the future development of more sophisticated visual reasoning abilities in AI systems.

8 Limitations

Despite the valuable contributions of our benchmark, several limitations remain that we aim to address in future work:

Limited Benchmark Size: The size of our benchmark is constrained by the significant cost of human annotation, as we prioritize high-quality task design to ensure meaningful insights, with each annotator dedicating over 200 hours to constructing `HumanEval-V`. Nevertheless, our benchmark includes 253 tasks, comparable to many well-established human-annotated benchmarks in academia and industry, such as HumanEval (Chen et al., 2021) with 164 tasks, MM-Vet (Yu et al., 2023) with 218, and Vibe-Eval (Padlewski et al., 2024) with 269. Notably, none of the current popular multimodal benchmarks feature manually drawn diagrams, further distinguishing `HumanEval-V`. Furthermore, `HumanEval-V` offers a diverse and balanced set of task types covering a wide range of capability aspects, enabling us to uncover unique insights into the limitations of current LMMs.

Limited Model Coverage: While our experiments evaluate a representative set of top-performing LMMs, the rapid pace of model development means newly released models may not be covered in our current evaluation. To address this, we plan to publicly release our evaluation toolkit and dataset, along with an up-to-date leaderboard to track ongoing advancements. This will enable benchmarking of new models as they become available, ensuring `HumanEval-V` remains relevant and continuously updated.

Limitations in Exploring Advanced Methods: While our experiments cover various evaluation settings, including chain-of-thought (CoT), iterative refinement, and long-CoT-enhanced LMMs, our exploration of more advanced CoT techniques is limited. Methods such as supervised fine-tuning (Chen et al., 2024b), reinforcement learning (Snell et al., 2024), or more complex CoT approaches (Yao et al., 2024; Mitra et al., 2024) could further enhance LMM reasoning capabilities. However, these techniques are challenging to apply to diagram reasoning due to the lack of high-quality training data in this domain. As our primary objective is to bridge the gap in diagram reasoning benchmarks, we leave the exploration of more sophisticated reasoning-enhancing methods to future work.

References

Pravesh Agrawal, Szymon Antoniak, Emma Bou Hanna, Baptiste Bout, Devendra Chaplot, Jessica Chudnovsky, Diogo Costa, Baudouin De Monicault, Saurabh Garg, Theophile Gervet, et al. 2024. Pixtral 12b. *arXiv preprint arXiv:2410.07073*.

Anthropic. 2024. Claude 3.5 sonnet.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Lin Chen, Jinsong Li, Xiaoyi Dong, Pan Zhang, Yuhang Zang, Zehui Chen, Haodong Duan, Jiaqi Wang, Yu Qiao, Dahua Lin, et al. 2024a. Are we on the right way for evaluating large vision-language models? *arXiv preprint arXiv:2403.20330*.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Qiguang Chen, Libo Qin, Jin Zhang, Zhi Chen, Xiao Xu, and Wanxiang Che. 2024b. M3cot: A novel benchmark for multi-domain multi-step multi-modal chain-of-thought. *arXiv preprint arXiv:2405.16473*.

Zhe Chen, Weiyun Wang, Yue Cao, Yangzhou Liu, Zhangwei Gao, Erfei Cui, Jinguo Zhu, Shenglong Ye, Hao Tian, Zhaoyang Liu, et al. 2024c. Expanding performance boundaries of open-source multimodal models with model, data, and test-time scaling. *arXiv preprint arXiv:2412.05271*.

Yew Ken Chia, Vernon Toh Yan Han, Deepanway Ghosal, Lidong Bing, and Soujanya Poria. 2024. Puzzlevqa: Diagnosing multimodal reasoning challenges of language models with abstract visual patterns. *arXiv preprint arXiv:2403.13315*.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, et al. 2024. Molmo and pixmo: Open weights and open data for state-of-the-art multimodal models. *arXiv preprint arXiv:2409.17146*.

Haodong Duan, Junming Yang, Yuxuan Qiao, Xinyu Fang, Lin Chen, Yuan Liu, Xiaoyi Dong, Yuhang Zang, Pan Zhang, Jiaqi Wang, Dahua Lin, and Kai Chen. 2024. Vlmevalkit: An open-source toolkit for evaluating large multi-modality models. *Preprint*, arXiv:2407.11691.

Geoffrey K Gill and Chris F Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284–1288.

Google. 2024a. Introducing gemini 1.5, google's next-generation ai model.

Google. 2024b. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models.

Tianrui Guan, Fuxiao Liu, Xiyang Wu, Ruiqi Xian, Zongxia Li, Xiaoyu Liu, Xijun Wang, Lichang Chen, Furong Huang, Yaser Yacoob, et al. 2023. Hallusionbench: An advanced diagnostic suite for entangled language hallucination and visual illusion in large vision-language models. *arXiv preprint arXiv:2310.14566*.

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

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

Yifan Jiang, Jiarui Zhang, Kexuan Sun, Zhivar Sourati, Kian Ahrabian, Kaixin Ma, Filip Ilievski, and Jay Pujara. 2024. Marvel: Multidimensional abstraction and reasoning through visual evaluation and learning. *arXiv preprint arXiv:2404.13591*.

Aniruddha Kembhavi, Mike Salvato, Eric Kolve, Minjoon Seo, Hannaneh Hajishirzi, and Ali Farhadi. 2016. A diagram is worth a dozen images. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 235–251. Springer.

Hugo Laurençon, Léo Tronchon, and Victor Sanh. 2024. Unlocking the conversion of web screenshots into html code with the websight dataset. *arXiv preprint arXiv:2403.09029*.

Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. 2024a. Llava-onevision: Easy visual task transfer. *arXiv preprint arXiv:2408.03326*.

Bohao Li, Rui Wang, Guangzhi Wang, Yuying Ge, Yixiao Ge, and Ying Shan. 2023. Seed-bench: Benchmarking multimodal llms with generative comprehension. *arXiv preprint arXiv:2307.16125*.

Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, Zhiyong Huang, and Jing Ma. 2024b. Mmcode: Benchmarking multimodal large language models for code generation with visually rich programming problems. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 736–783.

Yuan Liu, Haodong Duan, Yuanhan Zhang, Bo Li, Songyang Zhang, Wangbo Zhao, Yike Yuan, Jiaqi Wang, Conghui He, Ziwei Liu, et al. 2023. Mmbench: Is your multi-modal model an all-around player? *arXiv preprint arXiv:2307.06281*.

Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. 2023. Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts. *arXiv preprint arXiv:2310.02255*.

Pan Lu, Swaroop Mishra, Tanglin Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. 2022. Learn to explain: Multimodal reasoning via thought chains for science question answering. *Advances in Neural Information Processing Systems*, 35:2507–2521.

Pan Lu, Liang Qiu, Jiaqi Chen, Tony Xia, Yizhou Zhao, Wei Zhang, Zhou Yu, Xiaodan Liang, and Song-Chun Zhu. 2021. Iconqa: A new benchmark for abstract diagram understanding and visual language reasoning. *arXiv preprint arXiv:2110.13214*.

Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. 2022. Chartqa: A benchmark for question answering about charts with visual and logical reasoning. *arXiv preprint arXiv:2203.10244*.

Microsoft. 2024. Discover the new multi-lingual, high-quality phi-3.5 slms.

Chancharik Mitra, Brandon Huang, Trevor Darrell, and Roei Herzig. 2024. Compositional chain-of-thought prompting for large multimodal models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14420–14431.

Weili Nie, Zhiding Yu, Lei Mao, Ankit B Patel, Yuke Zhu, and Anima Anandkumar. 2020. Bongard-logo: A new benchmark for human-level concept learning and reasoning. *Advances in Neural Information Processing Systems*, 33:16468–16480.

OpenAI. 2024a. Hello gpt-4o.

OpenAI. 2024b. Introducing openai o1.

OpenAI. 2024c. tiktoken.

Piotr Padlewski, Max Bain, Matthew Henderson, Zhongkai Zhu, Nishant Relan, Hai Pham, Donovan Ong, Kaloyan Aleksiev, Aitor Ormazabal, Samuel Phua, et al. 2024. Vibe-eval: A hard evaluation suite for measuring progress of multimodal language models. *arXiv preprint arXiv:2405.02287*.

Pooyan Rahmanzadehgervi, Logan Bolton, Mohammad Reza Taesiri, and Anh Totti Nguyen. 2024. Vision language models are blind. In *Proceedings of the Asian Conference on Computer Vision*, pages 18–34.

Chufan Shi, Cheng Yang, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, et al. 2024. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. *arXiv preprint arXiv:2406.09961*.

Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2024. Design2code: How far are we from automating front-end engineering? *arXiv preprint arXiv:2403.03163*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. 2023. Modular visual question answering via code generation. *arXiv preprint arXiv:2306.05392*.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 11888–11898.

Chameleon Team. 2024a. Chameleon: Mixed-modal early-fusion foundation models. *arXiv preprint arXiv:2405.09818*.

Qwen Team. 2024b. Qvq: To see the world with wisdom.

Ke Wang, Junting Pan, Weikang Shi, Zimu Lu, Mingjie Zhan, and Hongsheng Li. 2024a. Measuring multimodal mathematical reasoning with math-vision dataset. *arXiv preprint arXiv:2402.14804*.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. 2024b. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*.

Zirui Wang, Mengzhou Xia, Luxi He, Howard Chen, Yitao Liu, Richard Zhu, Kaiqu Liang, Xindi Wu, Haotian Liu, Sadhika Malladi, et al. 2024c. Charxiv: Charting gaps in realistic chart understanding in multimodal llms. *arXiv preprint arXiv:2406.18521*.

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

Wikipedia. 2024. Pylint — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Pylint&oldid=1191495734. [Online; accessed 24-October-2024].

Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. 2024. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. *arXiv preprint arXiv:2405.07990*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Kaining Ying, Fanqing Meng, Jin Wang, Zhiqian Li, Han Lin, Yue Yang, Hao Zhang, Wenbo Zhang, Yuqi Lin, Shuo Liu, et al. 2024. Mmt-bench: A comprehensive multimodal benchmark for evaluating large vision-language models towards multitask agi. *arXiv preprint arXiv:2404.16006*.

Weihao Yu, Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Zicheng Liu, Xinchao Wang, and Lijuan Wang. 2023. Mm-vet: Evaluating large multimodal models for integrated capabilities. *arXiv preprint arXiv:2308.02490*.

Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, et al. 2024. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9556–9567.

Chi Zhang, Feng Gao, Baoxiong Jia, Yixin Zhu, and Song-Chun Zhu. 2019. Raven: A dataset for relational and analogical visual reasoning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5317–5327.

Renrui Zhang, Dongzhi Jiang, Yichi Zhang, Haokun Lin, Ziyu Guo, Pengshuo Qiu, Aojun Zhou, Pan Lu, Kai-Wei Chang, Yu Qiao, et al. 2024. Mathverse: Does your multi-modal llm truly see the diagrams in visual math problems? In *European Conference on Computer Vision*, pages 169–186. Springer.

Appendix

A Comparison with Other Benchmarks 12
- A.1 Correlation Analysis 12
- A.2 Diagrams in Other Benchmarks . . 12
- A.3 Diagrams in HumanEval-V 13

B More Details on Data Annotation 14
- B.1 Data Collection and Screening . . 14
- B.2 Recreation and Diversification . . 15

C More details on Experimental Setup 15
- C.1 Evaluated Models 15
- C.2 Prompt Templates 15
- C.3 Ablation on Temperature 16

D Deeper Analysis on HumanEval-V 16
- D.1 Co-occurrence of Capability Aspects 16
- D.2 Comparison of Strong Coder Models 16
- D.3 The Effect of Diversified Tasks . . 17

E More Detailed Error Analysis 19
- E.1 Error Patterns and Taxonomy . . . 19
- E.2 Error Analysis by Task Difficulty . 19

F More Discussion on MMCode 19

G Other Considerations 20

A Comparison with Other Benchmarks

A.1 Correlation Analysis

To assess whether `HumanEval-V` identifies specific weaknesses not captured by existing benchmarks, we select seven widely used multimodal benchmarks that cover a range of multidisciplinary abilities. These include AI2D (Kembhavi et al., 2016), MMVet (Yu et al., 2023), MMBench (Liu et al., 2023), MathVista (Lu et al., 2023), MMMU (Yue et al., 2024), MMStar (Chen et al., 2024a), and HallusionBench (Guan et al., 2023). Performance results for the 22 LMMs evaluated in this paper are collected from the OpenVLM Leaderboard (Duan et al., 2024), as well as corresponding papers and reports. These results are shown alongside the pass@3 scores for `HumanEval-V` under the *V2T2C w/ GPT-4o* setting in Table 2.

From the analysis, we observe that open-weight LMMs with more than 70B parameters generally perform well on the selected benchmarks, with models like Pixtral, InternVL 2.5, and Qwen2 VL even outperforming proprietary models such as GPT-4o and Claude 3.5 Sonnet in several cases. Llama-3.2-V also shows competitive performance. However, open-weight LMMs exhibit significantly lower performance on `HumanEval-V`, suggesting that our benchmark uncovers model weaknesses that may not be apparent in other benchmarks.

To quantify the relationship between `HumanEval-V` and the other five benchmarks, we visualize the performance of the 22 LMMs across all benchmarks using regression plots for each benchmark pair in Figure 20. The plots reveal low correlations between `HumanEval-V` and the other benchmarks, with notable differences in performance across models. Overall, the performance of all models remains lower on `HumanEval-V` compared to the other benchmarks.

A.2 Diagrams in Other Benchmarks

Figure 21 presents a comprehensive comparison of five distinct categories of diagrams commonly used in various benchmarks and coding platforms, showcasing the diverse range of visual reasoning challenges in the open world. The first category consists of real-world images from benchmarks such as MMMU, MMBench, and MM-Vet, encompassing everyday photographs of food, sports, architecture, art, and wildlife in both color and monochrome formats. These images test general visual recognition and understanding capabilities, contrasting sharply

Models	**AI2D**	**MM-Vet**	**MMBench**	**MathVista**	**MMMU**	**MMStar**	**HallusionBench**	**HumanEval-V** 🏅
				Proprietary LMMs				
Claude 3.5 Sonnet	81.2	66.0	81.7	67.7	**65.9**	65.1	55.1	**43.7**
GPT-4o	84.9	**69.1**	84.3	61.3	**69.2**	65.1	56.2	**40.5**
Gemini 1.5 Pro	79.1	64.0	82.8	57.5	60.6	59.1	45.6	37.3
Gemini 1.5 Flash	78.5	63.2	76.9	51.2	58.2	55.8	48.5	27.2
GPT-4o-mini	77.8	66.9	76.0	52.4	60.0	54.8	46.1	24.6
				Open-Weight LMMs				
Pixtral 124B	**93.8**	-	-	**69.4**	64.0	-	-	31.6
InternVL 2.5 78B	89.2	64.4	**87.7**	65.6	58.3	**72.1**	**57.4**	31.4
Qwen2 VL 72B	83.0	**74.0**	81.0	**70.5**	64.5	25.9	**58.7**	25.1
LLaVA-OV 72B	86.2	63.7	82.6	67.5	56.6	**65.8**	47.9	19.7
Molmo-D 72B	83.4	61.1	79.5	55.2	52.8	63.3	46.4	14.2
Llama-3.2-V 90B	**92.3**	64.1	77.3	57.3	60.3	55.3	44.1	11.0
Pixtral 12B	77.4	58.5	72.7	56.3	44.1	54.5	47.0	21.3
InternVL 2.5 26B	86.2	60.0	84.6	59.4	50.7	66.5	55.8	16.7
Qwen2 VL 7B	88.3	62.0	**85.9**	58.2	54.1	16.3	50.4	14.7
InternVL 2.5 8B	84.6	54.3	82.5	58.3	51.2	63.2	49.0	13.6
InternVL 2.5 4B	81.4	50.9	78.2	58.1	48.3	58.7	46.6	13.5
LLaVA-OV 7B	82.8	57.5	80.9	63.2	46.8	61.9	31.6	10.2
Phi-3.5-V 4B	77.8	43.2	67.4	43.2	44.6	47.5	40.5	9.4
Llama-3.2-V 11B	91.1	57.6	65.8	51.5	50.7	49.8	40.3	8.8
Molmo-D 7B	79.6	53.3	76.5	46.9	48.7	54.4	47.4	8.4
Chameleon 7B	46.0	8.3	19.8	22.5	22.4	31.1	17.1	2.2
Chameleon 30B	53.7	9.7	32.7	23.8	38.8	32.7	18.6	1.9

Table 2: A performance comparison of 22 LMMs across `HumanEval-V` and seven popular multimodal benchmarks. Models are ranked according to the 🏅 column. Results for `HumanEval-V` correspond to the *V2T2C w/ GPT-4o* setting from Table 1. The top two results for each column are highlighted in **bold**.

with the more structured representations found in other categories.

The second and third categories focus on analytical and scientific visualization. Analytical tables and charts, evaluated through benchmarks like ChartQA (Masry et al., 2022) and Charxiv (Wang et al., 2024c), comprise business and scientific data visualizations including bar charts, line graphs, and frequency tables. Scientific diagrams featured in MMMU (Yue et al., 2024), MMBench (Liu et al., 2023), and ScienceQA (Lu et al., 2022) present technical illustrations of molecular structures, particle dynamics, and ecosystem relationships. While both categories deal with data representation, they differ in their approach: analytical charts emphasize quantitative interpretation, whereas scientific diagrams focus on conceptual understanding.

Mathematical diagrams, assessed through benchmarks such as MathVista (Lu et al., 2023) and Math-Vision (Wang et al., 2024a), represent another crucial category that bridges pure mathematics with practical applications. These include function graphs, geometric constructions, and physics diagrams, demonstrating complex mathematical concepts through visual means. This category shares some common ground with programming-related diagrams, particularly in their emphasis on logical relationships and systematic thinking.

The fifth category encompasses visual abstract reasoning, evaluated through benchmarks like ARC-AGI (Chollet, 2019), RAVEN (Zhang et al., 2019), and Bongard (Nie et al., 2020). These tests feature grid-based patterns and geometric transformations that assess abstract thinking and pattern recognition skills. This category bears the closest resemblance to programming-related diagrams in terms of logical abstraction and systematic problem-solving approaches.

A.3 Diagrams in `HumanEval-V`

Figure 22 presents six fundamental task types in the `HumanEval-V` benchmark, each representing distinct cognitive challenges in visual reasoning. Our benchmark employs a rich variety of visual elements including geometric shapes, symbolic notations, matrices, and directed graphs. These representations are enhanced through connecting lines, arrows, color-coding, and numerical annotations

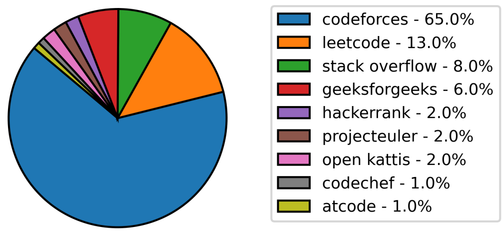


Figure 13: Sources of the screened tasks for annotation.

to effectively capture relationships and transformations between components. The visual representations maintain clarity across all categories while scaling in complexity to accommodate different difficulty levels. Through careful design of visual elements and systematic progression of patterns, each task type provides a clear framework for evaluating specific aspects of visual reasoning and problem-solving abilities.

The six task categories demonstrate diverse problem-solving requirements: Aggregation tasks (18% of the benchmark) introduce new grouping and aggregation rules for input data; Validation tasks (17%) define conditional rules to verify or classify input data; Expansion tasks (16%) focus on defining new patterns that evolve or extend data elements; Rearrangement tasks (17%) establish new traversal patterns to reorganize data; Iteration tasks (17%) define new iterative operations applied to input data; and Computation tasks (20%) introduce new computational rules and operations.

What distinguishes our benchmark is not only its balanced distribution across task types but also the wide interconnection between categories. While each category emphasizes specific problem-solving skills, real-world scenarios often require combining multiple approaches. For instance, computation tasks may incorporate iterative processes, while aggregation problems might require validation steps. This interconnected design reflects the complexity of practical problem-solving scenarios where multiple cognitive skills must be applied simultaneously.

B More Details on Data Annotation

B.1 Data Collection and Screening

Our data collection process involves two primary sources: coding challenge platforms, such as CodeForces, and the Q&A platform Stack Overflow (SO). Each coding problem undergoes a rigorous screening process to ensure it aligns with the standards of `HumanEval-V`. Annotators are instructed to exclude problems that: (1) require knowledge of specific programming frameworks or libraries, (2) contain images that are not abstract diagrams, (3) provide no useful information for solving the problem, or (4) require excessive textual context for interpretation.

The majority of our tasks are sourced from coding challenge platforms, especially CodeForces, as shown in Figure 13, where we display the distribution of screened tasks by platform. For coding challenge platforms, we use the open-source MMCode dataset (Li et al., 2024b), which includes coding problems from various platforms with visual elements in the problem descriptions. However, we find that most of these problems are unsuitable for `HumanEval-V`. Many images are non-essential, as they can be inferred from the textual problem descriptions. Some problems, though containing relevant visual information, are overly complex and require lengthy textual descriptions to interpret, violating our requirement for self-explanatory visual content. After careful screening, less than 5% of the viewed problems pass our standards.

We select SO for its extensive repository of real-world programming problems. To identify relevant posts, we first filter questions from 2020 that have non-negative votes and accepted answers. Then, we focus on posts that include images in the question body and code blocks in the corresponding answers, narrowing down further to those tagged with Python. After this automated filtering, we manually review the remaining posts, excluding topics related to front-end, mobile, or UI development, as these often require external frameworks and libraries that do not align with the goals of our benchmark. We also exclude posts where the images provide information in textual nature, such as code snippets, error messages, or execution outputs. Ultimately, we identified suitable questions primarily covering topics like geometry, plotting, and image processing.

To further illustrate our screening process, we present two negative examples that do not meet our standards in Figure 23: (1) The first example is a coding problem from CodeForces, where the task is to determine an optimal stacking method for a set of books with identical heights, given their thickness and width, in order to minimize the total thickness. While the provided image shows a possible stacking configuration, it lacks critical information, such as constraints on the stacking method and precise book dimensions. Moreover, the core problem-solving details are conveyed primarily through text, making the image non-essential for understanding the solution. (2) The second example is a coding problem from GeeksForGeeks, which involves traversing a 2D matrix according to a specified pattern, starting from the top-left corner and identifying the traversal endpoint. Although the image offers a basic representation of the matrix, the traversal pattern is too complex to be effectively captured visually and requires significant textual explanation. As a result, the textual description carries more problem-solving information than the image itself, violating our requirement for the visual context to be self-explanatory and serve as the primary source of information.

B.2 Recreation and Diversification

We present three examples in Figure 24, Figure 25, and Figure 26 to demonstrate our recreation and diversification process. Each figure is divided into three parts: the original problem that meets our screening criteria (top), the recreated coding task based on the distilled ideas (middle), and the diversified variant (bottom). Below are detailed explanations of each example:

Figure 24 showcases a Stack Overflow problem where a developer needs to draw a parallelogram using four specified points. The image illustrates the connection between these points, providing the essential information needed to solve the task. Since the text merely restates the geometric properties shown in the image, we significantly reduce the textual content without losing crucial details. For recreation, we transform this into a five-pointed star problem, enriching the visual information with four examples showing different point connection patterns. The new function signature clearly defines the implementation requirements, including objectives, input parameters, and return value constraints. Instead of generating a parallelogram image, our task focuses on determining whether two specific points should be connected, simplifying the implementation while maintaining emphasis on visual reasoning. For diversification, we modify the visual pattern from a five-pointed to a six-pointed star while maintaining the same function signature.

Figure 25 presents a CodeForces problem involving polygon folding and area calculation. The image demonstrates the folding process along dashed lines, showing both initial and final states. For recreation, we simplify this into a matrix folding task where overlapping sections produce color changes. The input matrix uses two initial colors (white and light blue), which can result in three distinct outcomes after folding (white, light blue, and dark blue). Three illustrative examples clarify the folding mechanics. For diversification, we replace the color addition rule with numeric addition, requiring models to process numerical changes before and after folding.

Figure 26, also from CodeForces, involves grid reduction following a specific pattern. The image effectively communicates the step-by-step transformation process. For recreation, we enhance the complexity by removing the reduction factor k as a parameter, requiring models to deduce that $k = 2$ from the provided examples. We transform the original binary scaling operation into a statistical pooling operation (e.g., minimum value computation), demanding both OCR capabilities and advanced visual reasoning. For diversification, we increase the pooling stride from 2 to 3, requiring models to analyze larger matrices. Test cases are adjusted accordingly to maintain consistency with the modified patterns.

In addition to the three examples above, we provide further examples of how we perform diversification across specific capability aspects in Table 3.

C More details on Experimental Setup

C.1 Evaluated Models

In Table 4, we provide a detailed list of Large Multimodal Models (LMMs) used in our experiments. For each model, we specify the number of parameters and include direct links to relevant reports or Huggingface repositories for further reference.

C.2 Prompt Templates

We designed three main sets of prompts for the experiments. The first set is used for the evaluation pipelines in the main benchmarking experiments, covering scenarios such as Vision-to-Code, Vision-to-Code with Chain-of-Thought (CoT), Vision-to-Text, and Text-to-Code, as described in Section 3. The corresponding prompts for these scenarios are listed in Figure 27. The second set of prompts is used in the iterative refinement experiments, introduced in Section 4. These prompts address scenarios where code or previously generated textual problem specifications are refined based on feedback from the execution environment. The relevant prompts for this scenario are provided in Figure 28. The third scenario involves using GPT-4o as a judge to rate the diagram descriptions gen-

Modification Aspects	Examples
Spatial Transformation	Adjust concatenation, swapping, grouping, or stacking order; modify the direction of translation, flipping, rotation, or folding.
Mathematical Operations	Alter arithmetic operations, sorting order, or aggregation rules.
Dynamic Patterns	Reverse alternation sequences; switch increments to decrements; change the direction of spirals or zigzags; adjust layer layouts.
Object Attributes	Change the color, size, shape, angle or position of objects.
Object Relations	Reverse nesting or overlap order; modify connection, intersection, or adjacency rules; adjust boundary interaction conditions.
Data Structure	Modify graph direction, data type in array, matrix dimensions.

Table 3: Examples of modifications applied to diversify seed tasks. Modifications of a task may span many aspects.

erated by LMMs. The prompt used is shown in Figure 29.

Our prompt design has undergone multiple rounds of optimization to address specific issues we encountered. For instance, we instruct the model to follow a markdown format code block for generation and avoid generating multiple code blocks to streamline post-processing and improve parsing success rates. We also provide detailed instructions for prompting LMMs to generate diagram descriptions in a structured problem specification format, including *Problem Restatement*, *Visual Facts*, and *Visual Patterns*, ensuring a comprehensive capture and expression of the visual context. In the iterative refinement setting, we specifically instruct LMMs not to hardcode test cases into their generated code to ensure that improvements stem from an enhanced understanding of the problem. Additionally, for LLM-as-Judge experiments, we list clear steps for rating the LMMs' outputs, promoting more robust and reliable rating results.

C.3 Ablation on Temperature

As described in Section 3, we set the sampling temperature to $T = 0.8$ for generating multiple predictions, following established practices in code generation benchmarking (Chen et al., 2021, 2022). Given that LMMs may exhibit varying performance at different temperatures, we conduct an ablation study to assess the rationale behind this choice. Specifically, we evaluated all 22 LMMs under the *V2T2C w/ GPT-4o* setting across a range of temperatures from 0.4 to 1.0. The results are presented in Table 5, which indicate that LMMs generally demonstrate consistent performance across these settings, with a few models showing slight variations, further validating the rationale for our chosen temperature setting.

D Deeper Analysis on `HumanEval-V`

D.1 Co-occurrence of Capability Aspects

As illustrated in Figure 3, the diagrams in `HumanEval-V` encompass a wide range of capability aspects that require human-level intelligence for interpretation. Each diagram typically involves multiple capability aspects to be fully understood. To explore the relationships between these capability aspects, we conduct a co-occurrence analysis based on the aspect labels assigned by human annotators during task annotation. The results, presented in Figure 30, show a heatmap where each value represents the number of tasks in `HumanEval-V` that involve both corresponding aspect labels. Our analysis reveals that the diagrams in `HumanEval-V` exhibit a diverse distribution of capability aspects. Among them, *adjacency*, *grid*, *matrix*, and *sequence* are the most frequently occurring labels. Common co-occurrences include *matrix-adjacency*, *grid-boundary*, *grid-path*, and *sequence-linear increment*, highlighting the fundamental spatial and structural relationships embedded in these diagrams.

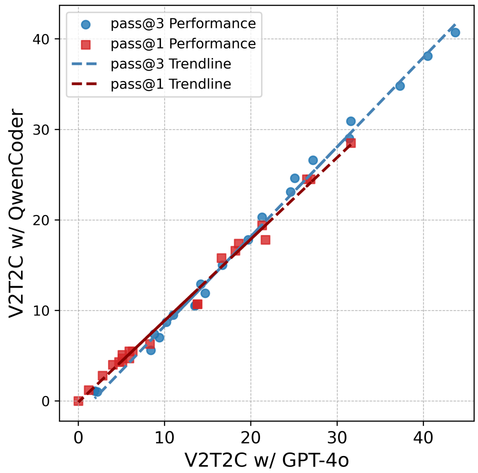


Figure 14: Performance comparison between GPT-4o and QwenCoder-32B as strong coders under the *V2T2C w/ SC* setting.

D.2 Comparison of Strong Coder Models

We use GPT-4o as the primary strong coder for our benchmarking experiments, leveraging its superior coding capabilities to translate problem specifications generated by LMMs into code. This allows us

Models	Params	Links
	Proprietary LMMs	
OpenAI o1 (OpenAI, 2024b)	-	OpenAI o1
GPT-4o (0806) (OpenAI, 2024a)	-	OpenAI GPT-4o
GPT-4o-mini (0718) (OpenAI, 2024a)	-	OpenAI GPT-4o-mini
Claude 3.5 Sonnet (1022) (Anthropic, 2024)	-	Anthropic Claude
Gemini 1.5 Pro (002) (Google, 2024a)	-	Google Gemini 1.5 Pro
Gemini 1.5 Flash (002) (Google, 2024a)	-	Google Gemini 1.5 Flash
	Open-weight LMMs with more than 70B parameters	
Pixtral (Agrawal et al., 2024)	124B	mistralai/Pixtral-Large-Instruct-2411
Llama-3.2-V 90B (Google, 2024b)	88.8B	meta-llama/Llama-3.2-90B-Vision-Instruct
InternVL 2.5 78B (Chen et al., 2024c)	78.4B	OpenGVLab/InternVL2-5-78B
Owen2 VL 72B (Wang et al., 2024b)	73.4B	Qwen/Qwen2-VL-72B-Instruct
QVQ-72B-Preview (Team, 2024b)	73.4B	Qwen/QVQ-72B-Preview
Molmo-D 72B (Deitke et al., 2024)	73.3B	allenai/Molmo-72B-0924
LLaVA-OV 72B (Li et al., 2024a)	73.2B	llava-hf/llava-onevision-qwen2-72b-ov-chat-hf
	Open-weight LMMs with fewer than 70B parameters	
Chameleon 30B (Team, 2024a)	34.3B	facebook/chameleon-30b
InternVL 2.5 26B (Chen et al., 2024c)	25.5B	OpenGVLab/InternVL2-5-26B
Pixtral 12B (Agrawal et al., 2024)	12.0B	mistralai/Pixtral-12B-2409
Llama-3.2-V 11B (Google, 2024b)	10.7B	meta-llama/Llama-3.2-11B-Vision-Instruct
Qwen2 VL 7B (Wang et al., 2024b)	8.3B	Qwen/Qwen2-VL-7B-Instruct
InternVl 2.5 8B (Chen et al., 2024c)	8.1B	OpenGVLab/InternVL2-5-8B
LLaVA-OV 7B (Li et al., 2024a)	8.03B	llava-hf/llava-onevision-qwen2-7b-ov-chat-hf
Molmo-D 7B (Deitke et al., 2024)	8.02B	allenai/Molmo-7B-D-0924
Chameleon 7B (Team, 2024a)	7.04B	facebook/chameleon-7b
Phi-3.5-V 4B (Microsoft, 2024)	4.2B	microsoft/Phi-3.5-vision-instruct
InternVL 2.5 4B (Chen et al., 2024c)	3.7B	OpenGVLab/InternVL2-5-4B
	Open-Weight Code LLM	
Qwen2.5 Coder 32B (Hui et al., 2024)	32.8B	Qwen/Qwen2.5-Coder-32B-Instruct

Table 4: List of LMMs with their parameter sizes and links to the official reports or Huggingface repositories.

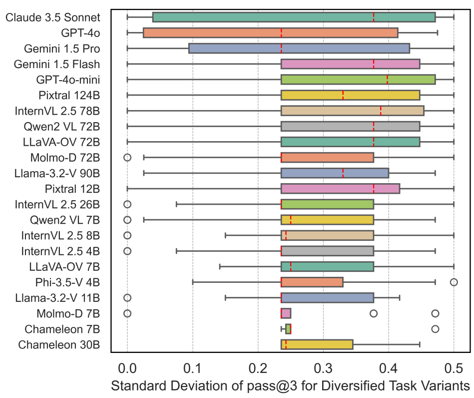


Figure 15: Effect of task diversification on performance variation across models in the *V2T2C w/ SC* setting.

to focus on the evaluation of LMMs' visual understanding ability in a more controllable manner. Our evaluation pipeline is designed to be robust, accommodating different strong coders. To test the stability of our results, we perform an ablation study by replacing GPT-4o with an open-weight LLM, Qwen2.5-Coder-32B-Instruct (Hui et al., 2024), referred to as QwenCoder-32B. QwenCoder-32B demonstrates comparable coding performance to GPT-4o, as evidenced by LiveCodeBench (Jain et al., 2024). This ablation allows us to explore whether switching to a different strong coder leads to deviations in our findings.

The ablation is conducted under the *V2T2C w/ SC* setting, where QwenCoder-32B replaces GPT-4o to generate code based on the problem specifications provided by LMMs. The results, shown in Figure 14, reveal that GPT-4o and QwenCoder-32B exhibit near-perfect correlations, demonstrating the strong stability of our evaluation methodology.

D.3 The Effect of Diversified Tasks

As outlined in Section 2 and Section B.2, our task annotation pipeline includes a crucial step to create diversified versions of the seed tasks, expanding both the volume and variety of tasks in `HumanEval-V`. To evaluate whether these diversi-

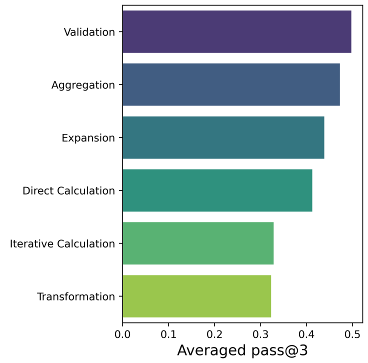


Figure 16: Pass rates of LMMs across different task types.

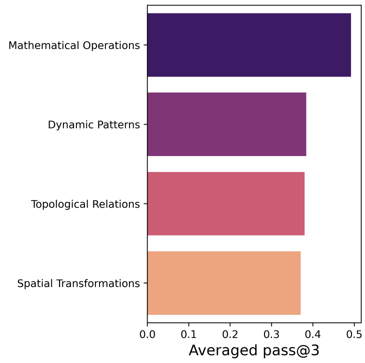


Figure 17: Pass rates of LMMs across main capability dimensions.

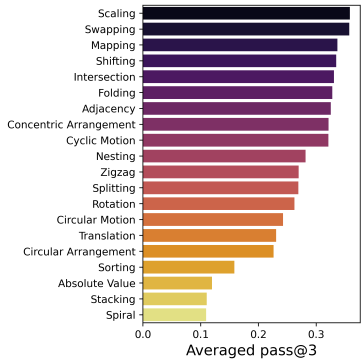


Figure 18: Pass rates of LMMs across specific capability aspects.

Models	**T=0.4**	**T=0.6**	**T=0.8**	**T=1**
Proprietary LMMs				
Claude 3.5 Sonnet	48.3	46.9	48.1	47.9
GPT-4o	44.9	42.2	43.6	44.9
Gemini 1.5 Pro	41.1	39.2	39.4	41.6
Gemini 1.5 Flash	27.1	30	28.4	29.1
GPT-4o-mini	27.9	29	29.9	32.1
Open-weight LMMs				
Pixtral 124B	35.6	39.8	34.2	37
InternVL 2.5 78B	37.2	36.1	35.9	36.2
Qwen2 VL 72B	31.1	29.1	28.7	31.5
LLaVA-OV 72B	25	22.7	24.1	22.2
Molmo-D 72B	16.9	13.2	14.4	14.5
Llama-3.2-V 90B	12.5	10.7	12.4	13.7
Pixtral 12B	21.2	23.4	23.2	21.1
InternVL 2.5 26B	20.9	21	20.7	21.8
Qwen2 VL 7B	13.3	17.2	16.6	18.5
InternVL 2.5 8B	13.6	16.6	17.3	16.3
InternVL 2.5 4B	16.9	17.6	15.2	20.5
LLaVA-OV 7B	15.1	15.2	13	15.2
Phi-3.5-V 4B	5.6	9.3	9.1	8.7
Llama-3.2-V 11B	7.5	10.2	9.6	9.9
Molmo-D 7B	9.2	10	11.2	11.2
Chameleon 7B	1	2.5	2.5	2.1
Chameleon 30B	2	0.5	3.3	2

Table 5: Ablation on LMMs' sampling temperature for the pass@3 results under the *V2T2C w/ GPT-4o* settings.

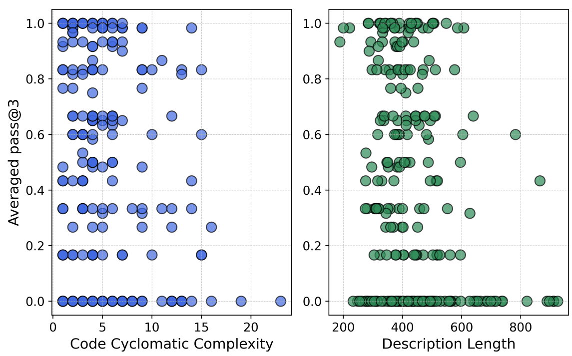


Figure 19: Correlation Between LMM Pass Rates and Task Difficulty in Coding and Diagram Descriptions.

fied tasks introduce different challenges compared to the original seed tasks, we analyze the standard deviation of pass rates across the seed tasks and their diversified versions.

Specifically, we use the results of the 22 LMMs under the V2T2C w/ SC setting. We group tasks based on whether they are seed tasks or their variants, resulting in 100 task groups (since we have 100 seed tasks). We then calculate the standard deviation of the pass@3 results within each group, excluding groups where all tasks have a pass@3 rate of 0. The standard deviation for each group is computed as:

$$\mathrm{SD}_{\text{group}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\text{pass@3}_i - \mu_{\text{group}})^2}$$

where N is the number of tasks in the group, pass@3_i is the pass rate of task i, and μ_{group} is the mean pass@3 for the group. The resulting distribution of pass@3 standard deviations (SD) is shown in Figure 15 as box plots.

The results reveal that the 25th percentile of most models has an SD greater than 0.2, and the median SD is around 0.4. This demonstrates a notable performance gap between tasks within the same group, validating the effectiveness of our task diversification process.

E More Detailed Error Analysis

E.1 Error Patterns and Taxonomy

To better understand where current LMMs fall short in solving the coding tasks in `HumanEval-V`, we conduct a statistical analysis examining the correlation between pass rates and three key factors: task type, general capability dimensions, and specific capability aspects (illustrated in Figure 3) required for understanding diagrams in `HumanEval-V`. The results are presented in Figure 16, Figure 17, and Figure 18. The pass rate in our analysis is the averaged pass@3 for the five proprietary models under the *V2T2C w/ SC* setting.

The analysis reveals that LMMs perform particularly poorly on tasks involving *Transformation* and *Iterative Calculation*, both achieving a pass@3 of approximately 32%. This suggests that these models struggle with understanding spatial transformations and tracking state changes over iterative steps. In terms of general capability dimensions, the difference in pass rates across various categories is minimal. Specifically, *Spatial Transformation*, *Topological Relations*, and *Dynamic Patterns* all yield an average pass@3 of 38%.

When examining specific capability aspects, we find that LMMs exhibit notable difficulty with diagrams involving dynamic patterns such as Spirals, Circular Arrangements, and Zigzags. Additionally, tasks requiring spatial transformations like Stacking, Translation, and Splitting, as well as mathematical operations such as Sorting and Absolute Value computations, pose significant challenges. We illustrate concrete error cases that highlight these challenges in Figure 31 to 45.

E.2 Error Analysis by Task Difficulty

We also investigate the correlation between LMM performance and task difficulty, using two key metrics. The first metric is the cyclomatic complexity (Gill and Kemerer, 1991) of the human-annotated solution code for each task, which reflects the complexity of programming logic. The second metric is the token length of the human-annotated diagram descriptions, which indicates the difficulty of understanding the diagram from a textual perspective. These two metrics represent human-perceived difficulty in both visual comprehension and programmatic reasoning.

For measuring LMM performance, we use the averaged pass@3 score of the top five proprietary models under the *V2T2C w/ SC* setting, with correlation results presented in Figure 19. Interestingly, the results suggest that LMM performance has little correlation with human-perceived difficulty, either in coding complexity or in visual description length, except for tasks with very high programming complexity or exceptionally long diagram descriptions.

Through a detailed case study, we find that many tasks in `HumanEval-V` are relatively easy for humans but remain challenging for LMMs, primarily because these models struggle to comprehend diagrams at a fundamental level. This limitation stems from their lack of basic visual perception and reasoning abilities, making it difficult to develop a precise metric that accurately captures LMM-perceived difficulty in our benchmark.

The error cases in Figures 46 and 47 further illustrate this challenge. Tasks that appear trivial to humans often prove insurmountable for even the top-performing LMMs, highlighting the fundamental gap in their ability to interpret diagrams and reason visually.

F More Discussion on MMCode

MMCode (Li et al., 2024b) presents a coding dataset for evaluating LMMs' algorithmic problem-solving capabilities in visual contexts, comprising 3.5k questions crawled from competitive programming platforms. However, as explained in Appendix B.1, the visual content in most coding challenges is redundant, with image information largely inferrable from textual descriptions. This redundancy is evident in MMCode's reported results, where performance on "language-only" inputs closely matches that of "vision + language" inputs. In contrast, `HumanEval-V` is specifically designed to evaluate visual understanding and reasoning capabilities rather than general coding proficiency. Our benchmark ensures that visual context is integral to problem-solving. Experiments with five proprietary LMMs demonstrate a striking contrast: while providing only function signatures without diagrams or diagram descriptions results in 0% pass rates across all models, the same models achieve over 90% pass rates (Figure 11) when given human-annotated diagram descriptions. This dramatic performance difference confirms the essential role of visual information in `HumanEval-V`. Furthermore, our difficulty analysis (Figure 5) shows that the coding tasks maintain moderate complexity, enabling a focused assessment of visual reasoning abilities. Our evaluation pipeline also introduces a two-stage code generation process, allowing LMMs with lower coding proficiency to generate diagram descriptions while delegating the coding implementation to more capable models. These deliberate design choices clearly distinguish `HumanEval-V` from MMCode by placing visual reasoning at the forefront of evaluation.

G Other Considerations

Environmental Considerations: Our benchmark's challenging tasks typically require larger-sized multimodal models, which raises environmental concerns regarding computational costs. However, we believe the solution lies in improving training efficiency rather than simply scaling up model size. Our future work will focus on developing resource-efficient training methods while maintaining performance on our benchmark.

Language Coverage: Currently, our benchmark primarily focuses on English and Python, which may appear limiting. This choice was deliberate, as these languages are most prevalent in LMMs' training data and best demonstrate their capabilities. While this focus allows for deeper analysis, we acknowledge the importance of linguistic diversity. Our annotation pipeline is language-agnostic and can be extended to other programming languages in future iterations.

License and Distribution: Our benchmark consists of manually created tasks, drawing inspiration from Stack Overflow discussions and the MMCode dataset (which includes problems from platforms like Codeforces and LeetCode). We intend to distribute our code and data under a research-only license *Creative Commons Non-Commercial (CC BY-NC)* to promote academic advancement.

Data Privacy and Protection: We can conclusively confirm that our dataset contains no personally identifiable information. All tasks were created from scratch by our team, with careful attention to privacy considerations. We maintained strict protocols during the creation process to ensure no sensitive information was included.

Computing Infrastructure: Our experimental setup utilized a computing node equipped with 8 NVIDIA A800 GPUs, primarily for LMM inference. Despite running multiple inference passes (1 greedy decode and 6 repeated sampling), the computational overhead remained manageable due to our focused dataset of 253 high-quality tasks.

Demographic of Annotators: The annotation team consisted of four highly qualified individuals – postgraduate and doctoral students specializing in computer science, each with over four years of Python programming experience. Their participation is entirely voluntary and research-motivated, with no monetary compensation involved. All annotators explicitly consented to participate in this academic endeavor. While this arrangement worked well for our academic setting, we acknowledge that paid annotation might be necessary for larger-scale or commercial projects.

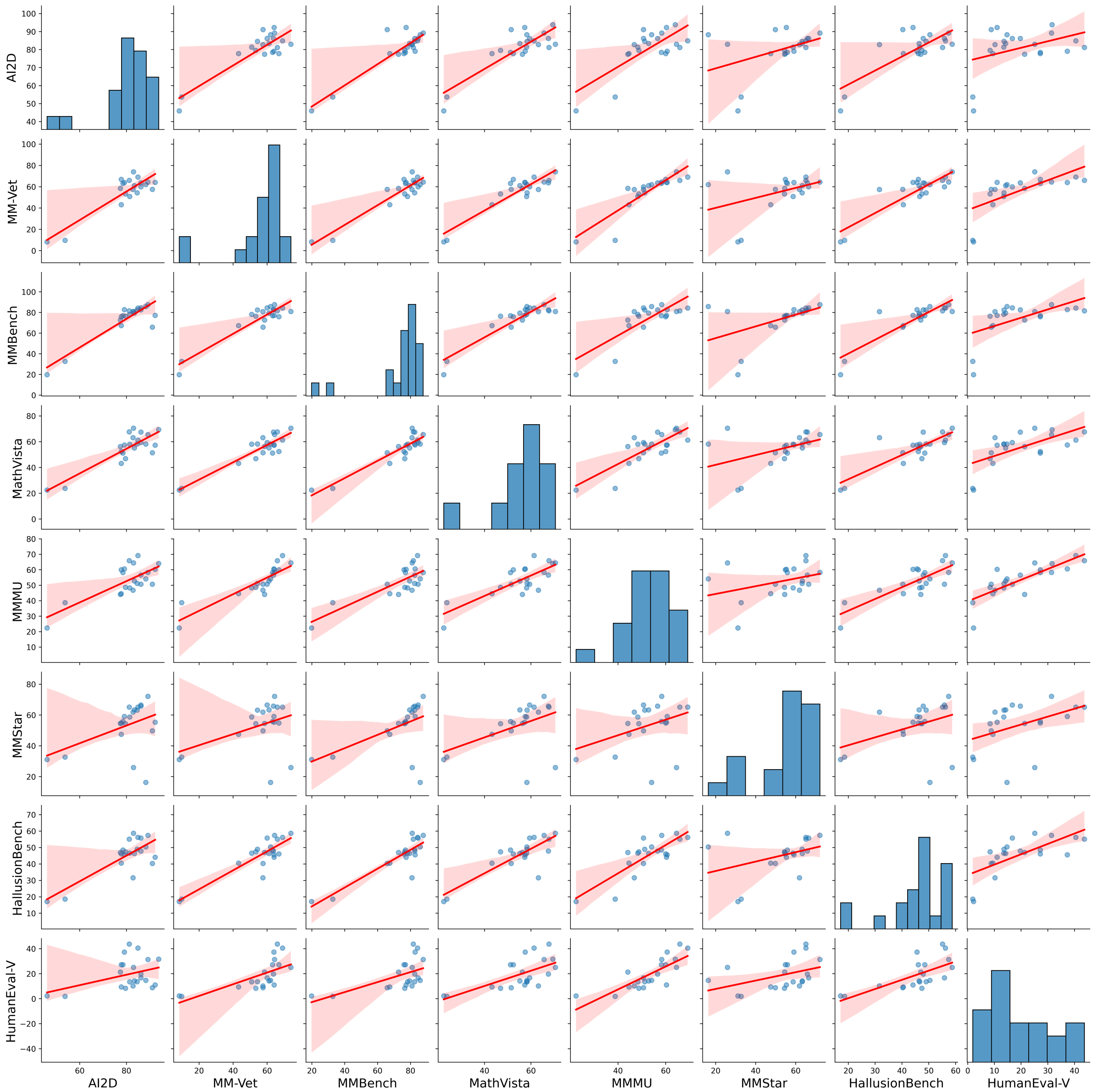


Figure 20: Correlations between eight multimodal benchmarks, including `HumanEval-V`. Each subplot displays the relationship between two benchmarks, while the diagonal subplots show the performance distribution for the corresponding benchmark.

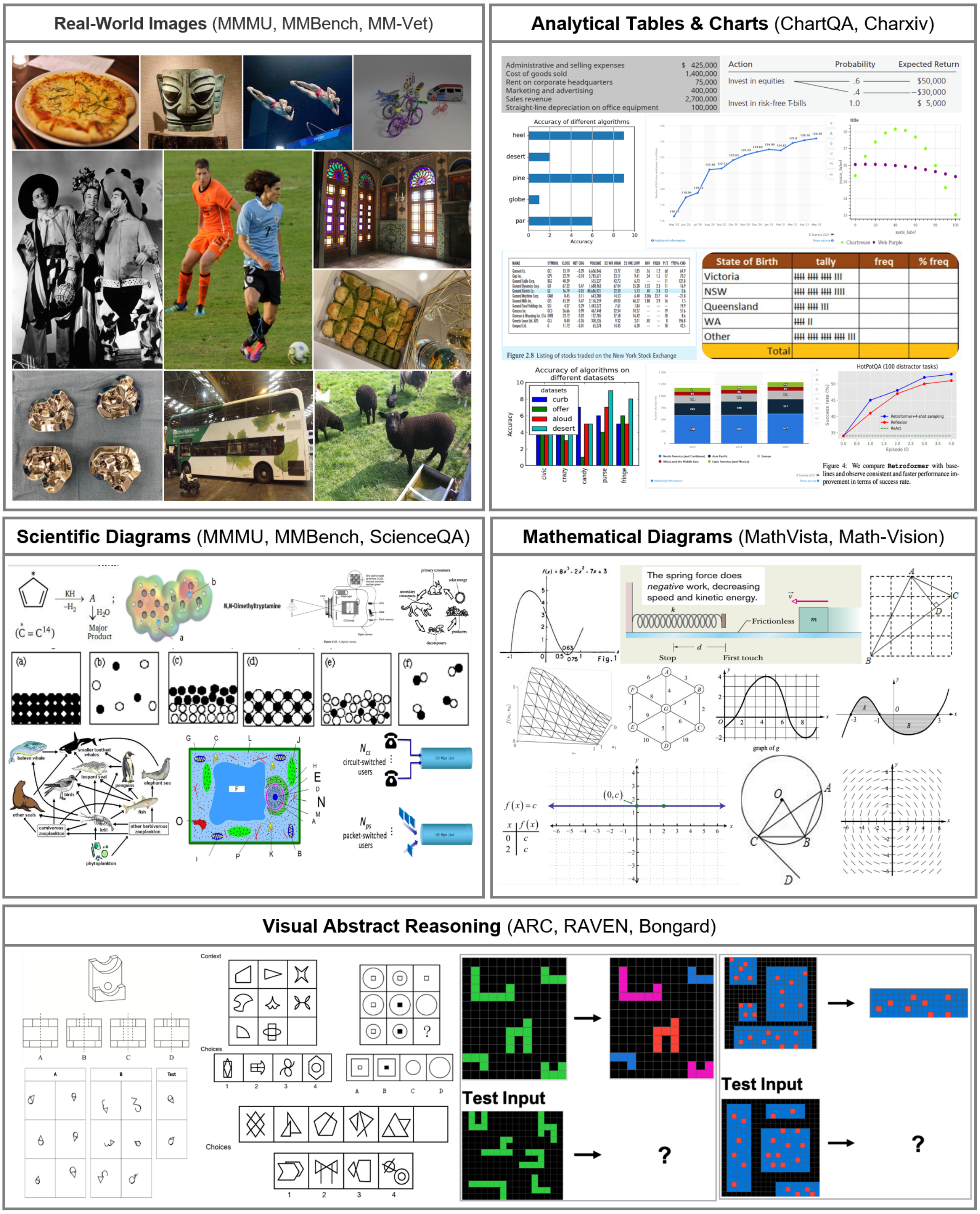


Figure 21: A comparison between diagrams covered in popular multimodal benchmarks.

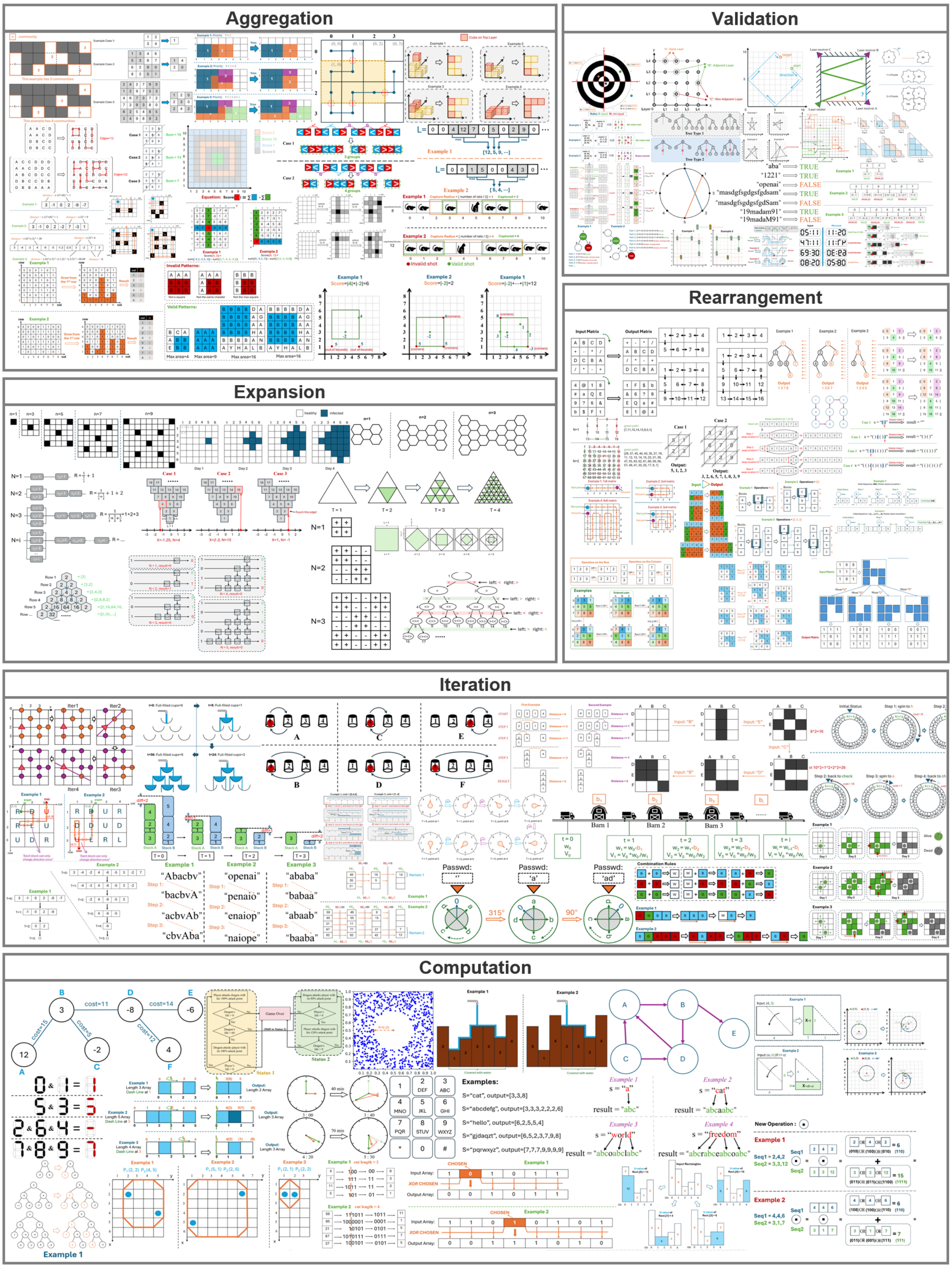


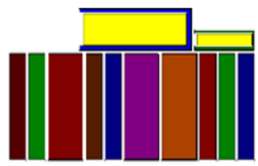
Figure 22: A curated selection of diagrams representing the six task types in HumanEval-V.

Problem Description:

Shaass has n books. He wants to make a bookshelf for all his books. He wants the bookshelf's dimensions to be as small as possible. The **thickness** of the i-th book is t_i and its pages' **width** is equal to w_i. The thickness of each book is either **1** or **2.** All books have the **same page heights.**
Shaass puts the books on the bookshelf in the following way. First he selects some of the books and put them vertically. Then he puts the rest of the books **horizontally above the vertical books.** The sum of the widths of the horizontal books must be **no more than** the total thickness of the vertical books. A sample arrangement of the books is depicted in the image.
Help Shaass to find the **minimum total thickness** of the vertical books that we can achieve.

Image:

This image lacks essential information.



Input:

The first line of the input contains an integer n, ($1 \le n \le 100$). Each of the next n lines contains two integers t_i and w_i denoting the **thickness** and **width** of the i-th book correspondingly, ($1 \le t_i \le 2$, $1 \le w_i \le 100$).

Output:

On the only line of the output print the **minimum total thickness** of the vertical books that we can achieve.

Problem Description:

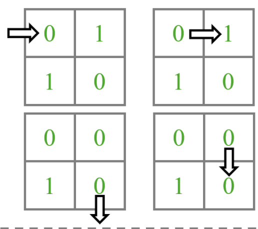
Given a binary matrix of dimensions with **R** rows and **C** columns. Start from **cell(0, 0)**, moving in the **right** direction. Perform the following operations:

- If the value of **matrix[i][j]** is **0**, then traverse in the same direction and check the next value.
- If the value of **matrix[i][j]** is **1**, then update **matrix[i][j]** to **0** and change the current direction clockwise. ie - **up**, **right**, **down**, or **left** directions change to **right**, **down**, **left**, and **up** respectively.

Find the index of the cell where you will be forced to exit the matrix while performing the given traversal.

Image:

This image requires substantial textual explanation to understand



Input:

A two-dimensional matrix **matrix[][]**, and the number of rows **R** and columns **C.**

Output:

The **index** of the cell from which you can exit the matrix.

Figure 23: Two negative examples in our data screening process: the first example is sourced from CodeForces (https://codeforces.com/problemset/problem/294/B), and the second from GeeksforGeeks (https://www.geeksforgeeks.org/problems/last-cell-in-a-matrix/1).

↓ Original Task ↓

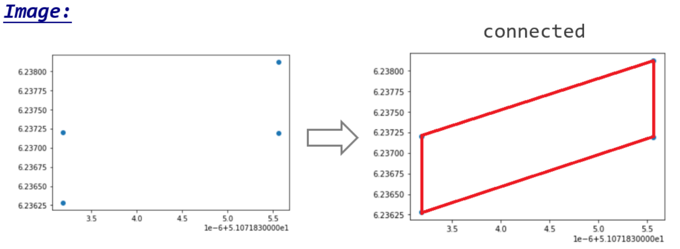
Problem Description:

I have four coordinates in my table:

lat1	lon1	lat2	lon2
51.07183	6.237204	51.07183	6.237195
lat3	**lon3**	**lat4**	**lon4**
51.07183	6.237195	51.07183	6.237204

Based on this data, I want to connect the points in such a way that a **parallelogram** is created.

Image:



Input:

An array of **DataFrame** type containing the coordinates of four points.

Output:

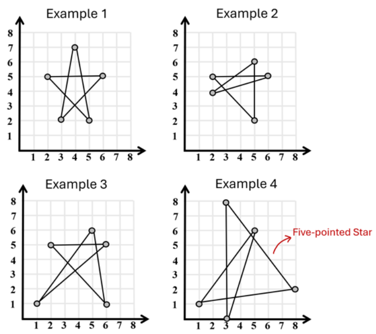
An **image** of a **parallelogram** in the coordinate system with the input four points as vertices.

↓ Our Recreated Version ↓

Function Signature:

```python
def solution(points: list[tuple[int, int]], point_a_index: int, point_b_index: int) -> bool:
    """
    Given a list of points and the index of two points, determine whether the two points should be connected to form the shape in the figure.

    Parameters:
        points (list[tuple[int, int]]): An list of tuples, where each tuple consists of two integers representing the x and y coordinates of a point. The points are not guaranteed to be in any particular order.
        point_a_index (int): The 0-based index of the first point.
        point_b_index (int): The 0-based index of the second point.

    Returns:
        bool: True if the two points should be connected, False otherwise.
    """
```

Image:



Test Cases:

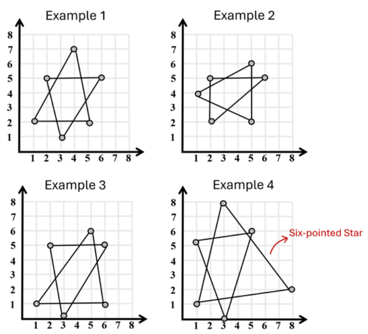
```python
assert solution([(2, 5), (4, 7), (3, 2), (5, 2), (6, 5)], 2, 4) == True; assert solution([(2, 5), (4, 7), (3, 2), (5, 2), (6, 5)], 2, 3) == False; assert solution(···
```

↓ Diversified Version ↓

Function Signature:

```python
def solution(points: list[tuple[int, int]], point_a_index: int, point_b_index: int) -> bool:
    """
    Given a list of points and the index of two points, determine whether the two points should be connected
    to form the shape in the figure.

    Parameters:
        points (list[tuple[int, int]]): An list of tuples, where each tuple consists of two integers repres-
        enting the x and y coordinates of a point. The points are not guaranteed to be in any particular order.
        point_a_index (int): The 0-based index of the first point.
        point_b_index (int): The 0-based index of the second point.

    Returns:
        bool: True if the two points should be connected, False otherwise.
    """
```

Image:



Test Cases:

```python
assert solution([(2, 5), (4, 7), (6, 5), (5, 2), (3, 1), (1, 2)], 2, 4) == True; assert solution([(2, 5), (4, 7), (6, 5), (5, 2), (3, 1), (1, 2)], 2, 3) == False; assert solution(···
```

Figure 24: Task annotation examples illustrating the recreation and diversification applied to the screened coding problem. The original problem is sourced from Stack Overflow (https://stackoverflow.com/questions/69163515).

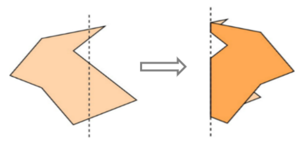
↓ Original Task ↓

Problem Description:

If you **fold the paper** along the vertical line $x = f$, what will be the **area** of the resulting shape? When you fold, the part of the paper to the left of the line is symmetrically reflected on the right side.

Your task is to answer q independent queries for values $f_1, \ldots, f_q$.

Image:



Input:

The first line contains two integers **n, q**– the number of polygon **vertices** and **queries**, respectively.
Each of the next n lines contains two integers x_i, y_i – the **coordinates** of the i-th point of the polygon.
Each of the next q lines contains a single integer f_i – the **x-coordinate** of the i-th fold query.

Output:

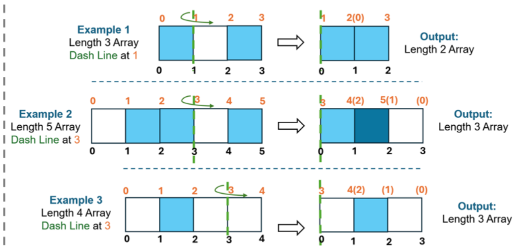
For each query, output the **area** A_i of the paper if you fold it along the line $x = f_i$.

↓ Our Recreated Version ↓

Function Signature:

```python
def solution(colors: list[list[int]], line_position: int) -> list[list[int]]:
    """
    Transforms a 1D array based on a given dashed line position.

    Args:
        colors (List[int]): A 1D array representing colors where:
           0 = white -1 = light blue  2 = dark blue
        line_position (int):
           The position of the dashed line used for transformation

    Returns:
        List[int]: A new 1D array with transformed colors where:
           0 = white -1 = light blue  2 = dark blue
    """
```

Image:



Test Cases:

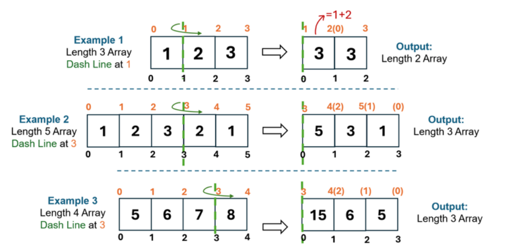
```python
assert solution([0, 1, 0, 1], 2) == [1, 1]; assert solution([0, 1, 0, 0], 3) == [0, 1, 0]; assert solution(···
```

↓ Diversified Version ↓

Function Signature:

```python
def solution(matrix: list[list[int]], line_position: int) -> list[list[int]]:
    """
    You are given a list of numbers. Your task is to generate a new list based on the given dashed line position.

    Input:
    - numbers: A 1D list of integers representing the initial state
        of the numbers.
    - line_position: An integer indicating the position of the
        dashed line in the transformation process.

    Output:
    - Return a 1D list of integers that represents the newly
        generated numbers after the transformation.
    """
```

Image:



Test Cases:

```python
assert solution([1, 2, 3], 1) == [3, 3]; assert solution([1, 2, 3, 4], 3) == [7, 2, 1]; assert solution(···
```

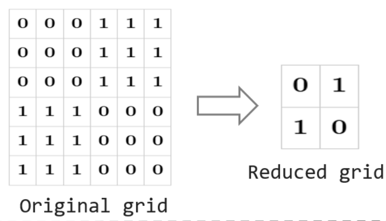
Figure 25: Task annotation examples illustrating the recreation and diversification applied to the screened coding problem. The original problem is sourced from CodeForces (`https://codeforces.com/problemset/problem/1381/E`).

↓ Original Task ↓

Problem Description:

Tina has a square grid with n rows and n columns. Each cell in the grid is either 0 or 1.
Tina wants to reduce the grid by a factor of k (k is a divisor of n). To do this, Tina splits the grid into $k \times k$ nonoverlapping blocks of cells such that every cell belongs to exactly one block.
Tina then replaces each block of cells with a single cell equal to the value of the cells in the block. **It is guaranteed that every cell in the same block has the same value.**
For example, the following demonstration shows a grid being reduced by factor of 3.

Image:



Input:

The first line contains t – the **number of test cases.**
The first line of each test case contains two integers n and k — the **number of rows and columns** of the grid, and the **factor** that Tina wants to reduce the grid by.
Each of the following **n** lines contain n characters describing the cells of the grid.
Each character is either **0 or 1.** It is guaranteed **every k by k block has the same value.**

Output:

For each test case, output the grid reduced by a factor of k on a new line.

↓ Our Recreated Version ↓

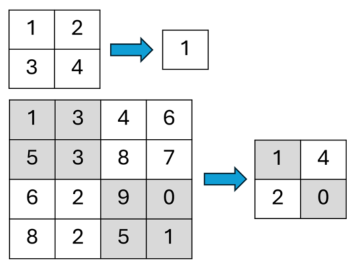
Function Signature:

```
def solution(matrix: list[list[int]]) -> list[list[int]]:
    """
    Refer to the example cases illustrated in the figure, identify and implement
the pooling operation on the matrix.

    Parameters:
        matrix: A 2d list representing the initial matrix.

    Returns:
        list[list[int]]: A 2d list representing the resulting matrix after the
pooling operation.
    """
```

Image:



Test Cases:

```
assert solution([[1, 3, 4, 2], [2, 1, 1, 3], [1, 2, 2, 4], [3, 2, 1, 0]]) == [[1, 1], [1, 0]]; assert solution(···
```

↓ Diversified Version ↓

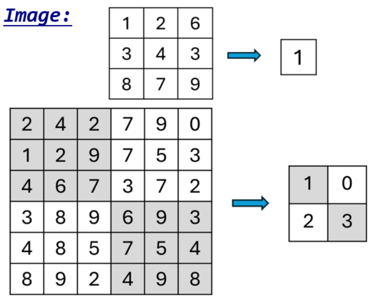
Function Signature:

```
def solution(matrix: list[list[int]]) -> list[list[int]]:
    """
    Refer to the example cases illustrated in the figure, identify and implement
the pooling operation on the
    matrix.

    Parameters:
        matrix: A 2d list representing the initial matrix.

    Returns:
        list[list[int]]: A 2d list representing the resulting matrix after the
pooling operation.
    """
```

Image:



Test Cases:

```
assert solution([[1, 3, 4, 2, 0, 3], [2, 1, 1, 3, 2, 6], [1, 2, 2, 4, 4, 7], [3, 2, 1, 0, 1, 0], [3, 2, 1, 0, 1, 0],
[1, 7, 5, 2, 2, 0], [2, 9, 1, 2, 3, 1]]) == [[1, 0], [1, 0]]; assert solution(···
```

Figure 26: Task annotation examples illustrating the recreation and diversification applied to the screened coding problem. The original problem is sourced from CodeForces (https://codeforces.com/problemset/problem/1996/B).

````
You are an exceptionally intelligent coding assistant with a deep understanding of Python programming and a keen ability to
interpret visual data. Your responses are consistently accurate, reliable, and thoughtful.

**Objective:**
You will be presented with a Python programming problem and an accompanying image. Please complete the function based on the
provided image and code context.

**Note**
- Remember, the signature by itself does not contain the entire problem; the image provides critical details.
- Observe the image closely and determine how its visual elements correspond to the problem's inputs, outputs, operations,
calculations, patterns (static/dynamic), and conditions.
- Please generate the complete code solution, including its function signature and body, formatted in a single Python code
block, **without any additional text or explanation**.

**Code Context:**
```python
{function_signature}
```
````

Scenario: $P_{V2C}(D, \sigma)$

````
You are an exceptionally intelligent coding assistant with a deep understanding of Python programming and a keen ability to
interpret visual data. Your responses are consistently accurate, reliable, and thoughtful.

**Objective:**
You will be presented with a Python programming problem and an accompanying image. Please complete the function based on the
provided image and code context.

**Note**
- {problem_category_specification}
- Remember, the signature by itself does not contain the entire problem; the image provides critical details.
- Observe the image closely and determine how its visual elements correspond to the problem's inputs, outputs, operations,
calculations, patterns (static/dynamic), and conditions.
- First summarize the important clues or findings and write a step-by-step analysis.
- Then generate the complete code solution,including the function signature and body,formatted in a single Python code block.

**Code Context:**
```python
{function_signature}
```
````

Scenario: $P_{V2C}(D, \sigma, I_{CoT})$

````
**Instructions:**
You will receive a Python programming problem and an accompanying image for analysis:

**Code Context:**
```python
{function_signature}
```

1. **Analyze the Function Signature**
   Examine the provided function signature (its input, output, and goal) and identify any missing context. Remember, the
signature by itself does not contain the entire problem; the image provides critical details.
2. **Examine the Image**
   Observe the image closely and determine how its visual elements correspond to the problem's inputs, outputs, operations,
calculations, patterns (static/dynamic), and conditions.
   - First, describe the visual elements you see.
   - Next, list the important facts from the image that are relevant for understanding the problem.
   - Finally, deduce any missing information from the problem based on the image.

**Response Format:**
Please structure your response in three main sections (use Markdown H1 headers):
1. **# Problem Restatement**
   Provide a concise restatement of the problem, including relevant background and requirements.
2. **# Visual Facts**
   List the facts directly observed from the image that are necessary for interpreting or solving the problem.
3. **# Visual Patterns**
   Summarize any objects, operations, transformations, patterns, conditions, and relationships inferred from these facts.
**Important Note:**
- Clearly separate facts (what you directly see in the image) from patterns (what you infer based on those facts).
- If complex visual information is difficult to express in plain language, use formal notation (mathematical or pseudo-code).
- State only what you are sure of; do not introduce assumptions not supported by the image or give vague conclusions.
- **Do not** include any code implementation in your response.
````

Scenario: $P_{V2T}(D, \sigma)$

````
**Instructions:**
You are an exceptionally intelligent coding assistant that consistently delivers accurate and reliable responses to user
instructions. Please complete the function based on the provided problem specification, code context, and accompanying image
(if provided). Return the complete solution, including the function signature, in a single response, formatted within a
Python code block.

**Problem Specification:**
```markdown
{problem_specification}
```

**Code Context:**
```python
{function_signature}
```
````

Scenario: $P_{T2C}(PS, \sigma)$

Figure 27: Prompting templates used for the four scenarios introduced in Section 3. {function_signature} and {problem_specification} serve as placeholders for the respective content.

````
You are an exceptionally intelligent coding assistant with a deep understanding of Python programming and a keen ability to
interpret visual data. Your responses are consistently accurate, reliable, and thoughtful.

**Objective:**
You will be presented with a Python programming problem, an accompanying image, and the problem analysis and code you
previously generated. Your task is to refine both the **problem analysis** and the **code solution** based on execution
feedback from the test cases.
                                                                                        Scenario: Iterative V2C
**Code Context:**
```python
{function_signature}
```
**Previous Problem Analysis and Solution:**
```markdown
{previous_prediction}
```
**Execution Feedback:**
```
{execution_feedback}
```

**Note**
- Remember, the signature by itself does not contain the entire problem; the image provides critical details.
- Observe the image closely and determine how its visual elements correspond to the problem's inputs, outputs, operations,
calculations, patterns (static/dynamic), and conditions.
- Carefully review the execution feedback and analyze any errors or issues that arose during testing.
- Based on the feedback, refine your understanding of the problem and make necessary corrections. Ensure you revisit the
previously neglected aspects from the image or problem analysis.
- !!You must NOT directly include the test cases from the feedback in your code. Doing so is considered cheating and
invalidates the solution.!! Instead, improve the logic to handle all potential scenarios correctly.

**Your task is to generate:**
1. A revised version of the step-by-step problem analysis with an improved understanding of the visual details, operations,
and conditions.
2. A refined Python code solution, formatted in a single code block, ensuring that it addresses the identified issues and
passes all test cases **without hardcoding specific values from the feedback**.
````

````
You are an exceptionally intelligent coding assistant with a deep understanding of Python programming and a keen ability to
interpret visual data. Your responses are consistently accurate, reliable, and thoughtful.

### Objective:
You will be presented with a Python programming problem, an accompanying image, and the **problem specification** you
previously generated. Your task is to refine and generate a **new version of the problem specification** based on execution
feedback from test cases.
                                                                                      Scenario: Iterative V2T2C
### Code Context:
```python
{function_signature}
```
### Your Previous Version Problem Specification:
```markdown
{previous_prediction}
```
### Execution Feedback:
```
{execution_feedback}
```

### Instruction for Refining Problem Specification:
1. **Analyze the Function Signature**
   - Examine the provided function signature (its input, output, and goal) and identify any missing context.
   - Remember, the signature by itself does not contain the entire problem; the image provides critical details.
2. **Examine the Image**
   Observe the image closely and determine how its visual elements correspond to the problem's inputs, outputs, operations,
calculations, patterns (static/dynamic), and conditions.
   - First, describe the visual elements you see.
   - Next, list the important facts from the image that are relevant for understanding the problem.
   - Finally, deduce any missing information from the problem based on the image.
3. **Execution Feedback Analysis**
   - Carefully review the execution feedback, especially error messages, or unexpected outputs with the expected results.
   - Analyze the issues that arose during testing, and consider how they may relate to aspects of the problem specification
that were previously unclear, overlooked, or incorrectly defined.
   - !!You must NOT directly include the test cases from the feedback into the refined problem specification. Doing so is
considered cheating and invalidates the refinement process.!!
   - Instead, generalize your understanding to address all possible cases comprehensively.
4. **Refine the Problem Specification**
   Based on the execution feedback, revise your understanding of the problem.
   - Clarify or update any ambiguous parts of the specification.
   - Address missing or incorrect details in the initial problem specification that were revealed by the test cases.

### Response Format:
Please structure your response in three main sections (use Markdown H1 headers):
- Problem Restatement, Visual Facts, and Visual Patterns

### Important Note:
- Focus on refining the **problem specification**. **Do not** include any code implementation in your response.
- Ensure your refinement is based on problem understanding. **Do not** hardcode the test case values in your response.
````

Figure 28: Prompting templates used for the iterative benchmarking scenarios introduced in Section 4. {function_signature}, {previous_prediction}, and {execution_feedback} serve as placeholders for the respective content.

```
**[Task Definition]**
Your task is to evaluate and identify the **Root-Cause Error** in a Model-Generated Problem Specification (Gen-PS) by
comparing it to a reference Human-Written Problem Specification (Ref-PS). Specifically, you will:
1. Identify the fundamental error in the Gen-PS that prevents it from accurately reflecting the visual context or programming
requirements.
2. Attribute the error to one or more of the following **capability dimensions**: `Basic-Level Perception`, `High-Level
Comprehension`, or `Contextual Interpretation`. Rate each dimension based on the severity of the error.
3. Label the specific **capability aspects** associated with the error.
4. Provide a clear explanation to justify your evaluation.

[Task Background & Key Terms]
- **Problem Specification**: A detailed analysis of an image's visual content and its alignment with a given Python function
signature to provide problem-solving instructions.
- Root-Cause Error: The most fundamental flaw in the Gen-PS that disrupts its ability to fully capture the problem's context.
- Capability Dimensions: The key areas for assessing the model's performance. Identify any conflicts between the Ref-PS and
Gen-PS in the following dimensions:
  - **Basic-Level Perceptin**:  Identifying basic visual elements like shapes, colors, text, numbers, layout, positions, etc.
  - **High-Level Comprehension**: Understanding the objects, relationships, constraints, patterns, and operations depicted in
the visual context.
  - **Contextual Interpretation**: The output contains three sections ("Problem Restatement", "Visual Facts", "Visual
Patterns"). The visual patterns are clearly described without vagueness or hallucinated details.

[Rating Scale]
Use a scale from **1** to **3**, where:
- **1**: Severe error in the specific capability dimension.
- **2**: Moderate error in the specific capability dimension.
- **3**: Nearly perfect in the specific capability dimension.

[Output Format]
Provide your evaluation in the following structure:

1. **Label**: A JSON object indicating the identified error and scores of three capability dimensions.
```json
{
    "ratings": {
        {
            "Basic-Level Perception": 1/2/3,
            "High-Level Comprehension": 1/2/3,
            "Contextual Interpretation": 1/2/3
        }
    }
}
```
2. **Explanation**: A concise justification for the identified root-cause error. Link it to the identified capability
dimensions and aspects.

[Inputs for Evaluation]

Reference Problem Specification (Ref-PS):
```markdown
{human_annotated_problem_specification}
```

Model-Generated Problem Specification (Gen-PS):
```markdown
{model_generated_problem_specification}
```

[Evaluation Notes]
1. Start by carefully comparing the Ref-PS and Gen-PS to identify discrepancies.
2. Determine the Root-Cause Error and assess which capability dimensions were not effectively addressed.
3. Rate each dimension from 1 to 3 based on the severity of the error.
```

Figure 29: Prompting templates used for the LLM-as-Judge rating experiment introduced in Section 5. {human_annotated_problem_specification} and {model_generated_problem_specification} serve as placeholders for the respective content.

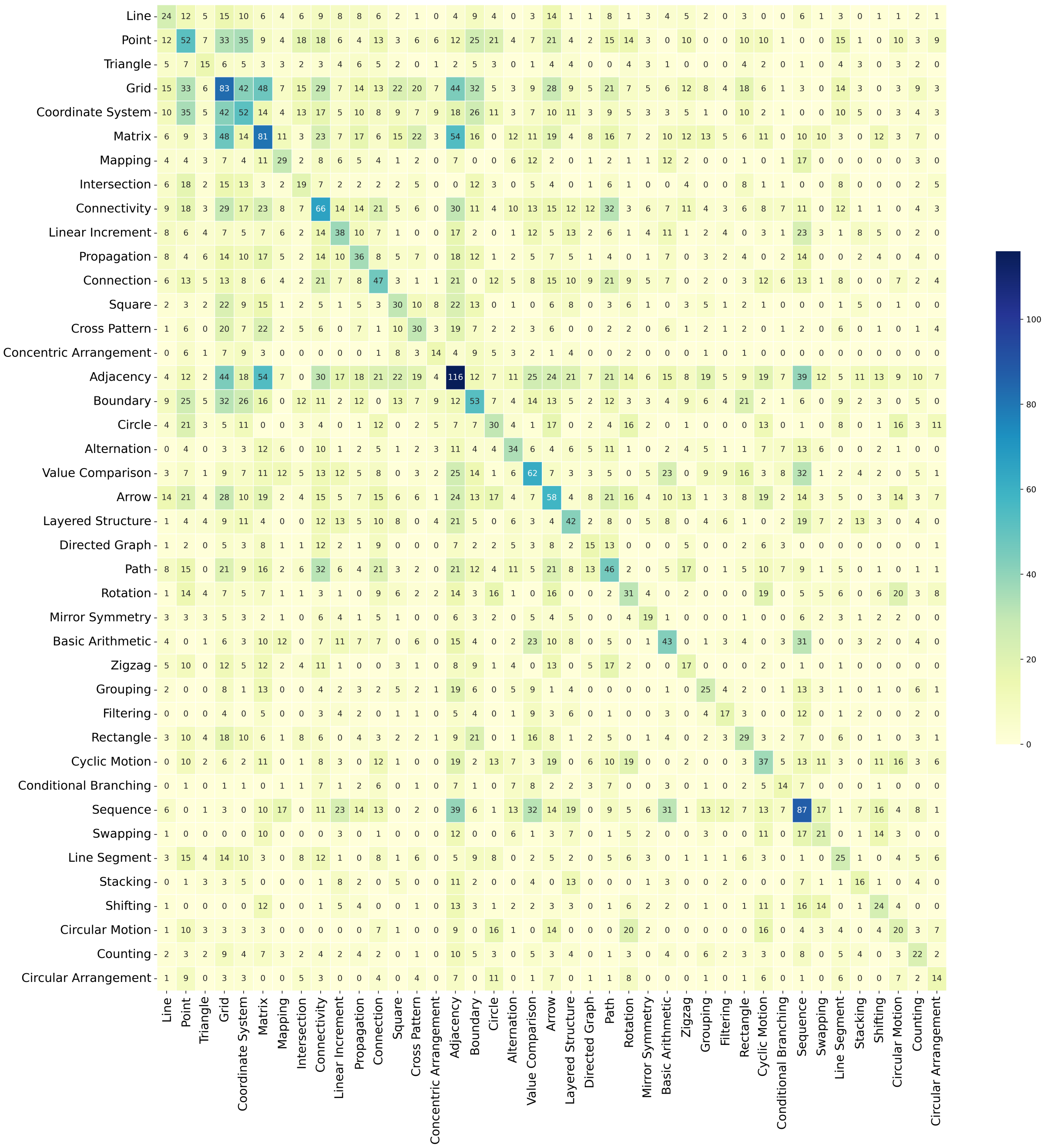
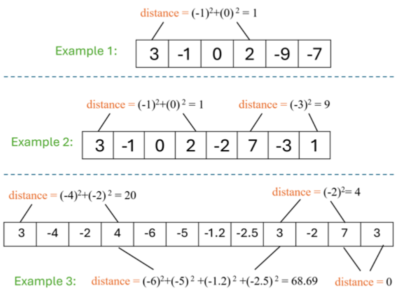


Figure 30: Analysis of the capability aspect co-occurrences in HumanEval-V tasks.

Image:



Function Signature:

```python
def solution(nums: list[float], idx1: int, idx2: int) -> float:
    """
    Calculate the distance between two elements in a list of numbers.
    Parameters:
        nums (list[float]): A list of numbers.
        idx1 (int): The first index position. (0-indexed)
        idx2 (int): The second index position. (0-indexed)
    Returns:
        float: The calculated distance between the elements at the two index positions. (round two decimal places)
    """
```

Test Cases:

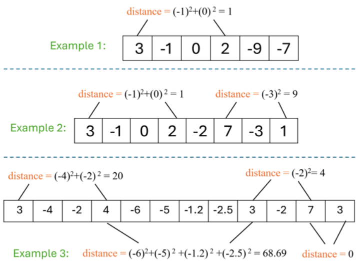
```python
assert solution([3,-1,0,2,-9,-7], 0, 3)==1; assert solution([2,-3,-1,-1,-2,-2,3,-3], 0, 6)==19; assert ···
```

Incorrect Diagram Description Generated by Claude 3.5 Sonnet:

```
# Problem Restatement
The problem requires implementing a function that calculates a "distance" between two elements in a list of numbers based on their index positions. The distance calculation follows specific rules shown in the examples, where elements between the two selected indices contribute to the total distance value. The result should be rounded to 2 decimal places.

# Visual Facts
1. Three examples are shown with arrays of different lengths:
   - Example 1: [3, -1, 0, 2, -9, -7]
   - Example 2: [3, -1, 0, 2, -2, 7, -3, 1]
   - Example 3: [3, -4, -2, 4, -6, -5, -1.2, -2.5, 3, -2, 7, 3]

2. Each example shows distance calculations between different pairs of indices:
   - Distance annotations show formulas like (-1)²+(0)² = 1
   - Multiple distances can be calculated within the same array
   - Some calculations involve multiple terms

3. Distance values shown include:
   - Single-step distances = 1 (from (-1)²+(0)²)
   - Larger distances = 9 (from (-3)²)
   - Complex distances = 68.69 (from (-6)²+(-5)²+(-1.2)²+(-2.5)²)
   - Zero distance in some cases

# Visual Patterns
1. Distance Calculation Rules:
   - When elements are adjacent, distance is calculated using the differences squared
   - For elements with other numbers between them, all elements in the range contribute
   - The calculation uses the square of each element in the range between (and including) the selected indices
                                   Incorrect: The elements shouldn't include both ends
   - Multiple terms are summed when there are multiple elements in the range

2. Mathematical Pattern:
   - Distance = Σ(x_i)² where x_i are the elements between and including the indexed positions
   - For adjacent elements or single elements, it simplifies to the square of the difference
   - When indices are the same, the distance is 0

3. Range Consideration:
   - The calculation includes both start and end indices
   - The order of indices doesn't matter (the distance is the same regardless of which index is first)
   - All elements between the two indices contribute to the final distance value
```

Figure 31: Example error case demonstrating LMMs' challenges with data structure manipulation and mathematical operations. The case shown for Claude 3.5 Sonnet is under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```
def solution(nums: list[float], idx1: int, idx2: int) -> float:
    """
    Calculate the distance between two elements in a list of numbers.
    Parameters:
        nums (list[float]): A list of numbers.
        idx1 (int): The first index position. (0-indexed)
        idx2 (int): The second index position. (0-indexed)
    Returns:
        float: The calculated distance between the elements at the two index
positions. (round two decimal places)
    """
```

Test Cases:

```
assert solution([3,-1,0,2,-9,-7], 0, 3)==1; assert solution([2,-3,-1,-1,-2,-2,3,-3], 0, 6)==19; assert ···
```

Incorrect Diagram Description Generated by Pixtral 124B:

```
## Problem Restatement
The problem requires calculating a "distance" between two elements in a list of floating-point numbers.
The function `solution` accepts a list of numbers and two index positions as inputs. The goal is to
compute and return this distance, rounded to two decimal places.

## Visual Facts
1. The image displays multiple examples of numerical lists with annotations explaining the calculations
of "distance."
2. Each example calculates the distance based on selected values from the list.
3. The distance formula used in the examples is the squared sum of selected elements from the list.
4. The index positions chosen for the examples are noted to be zero-indexed.
5. The results of the distance calculations are annotated and matched with specific parts of the list in
each example.
### Example 1:
- List: [3, -1, 0, 2, -9, -7]
- Selected indices: 1 and 2 (values: -1 and 0)
- Distance = (-1)^2 + (0)^2 = 1
### Example 2:
- List: [3, -1, 0, 2, -2, 7, -3, 1]
- Selected indices: 1 and 2 (values: -1 and 0)
  - Distance = (-1)^2 + (0)^2 = 1
- Selected indices: 4 and 5 (values: -2 and 7)
  - Distance = (-2)^2 + (7)^2 = 53
```

Hallucination

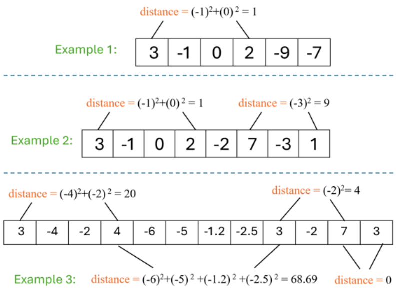
```
### Example 3:
- List: [3, -4, -2, 4, -6, -5, -1.2, -2.5, 3, -2, 7, 3]
- Selected indices: 1 and 2 (values: -4 and -2)
  - Distance = (-4)^2 + (-2)^2 = 20
- Selected indices: 9 and 10 (values: -2 and 7)
  - Distance = (-2)^2 + (7)^2 = 53
- Selected indices: 4 to 7 (values: -6, -5, -1.2, -1.5)
  - Distance = (-6)^2 + (-5)^2 + (-1.2)^2 + (-2.5)^2 = 68.69
- Selected indices: 11 and 11 (value: 3)
  - Distance = (3)^2 = 9

## Visual Patterns
1. The distance is calculated as the sum of the squares of selected elements from the list.
2. The selection of elements to calculate the distance depends on the provided indices.
3. The formula for distance between two indices `idx1` and `idx2` is the sum of squares of elements from
`idx1` to `idx2` (inclusive).
```

Incorrect: The elements shouldn't include both ends

````
4. If `idx1` equals `idx2`, the distance is simply the square of the element at that index.
5. The result should be formatted to two decimal places for precision.

The pattern is:
```
distance = sum(nums[i]^2 for i in range(idx1, idx2 + 1))
```
````

Figure 32: Example case on the same task in Figure 31, demonstrated by Pixtral 124B under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(nums: list[float], idx1: int, idx2: int) -> float:
    """
    Calculate the distance between two elements in a list of numbers.
    Parameters:
        nums (list[float]): A list of numbers.
        idx1 (int): The first index position. (0-indexed)
        idx2 (int): The second index position. (0-indexed)
    Returns:
        float: The calculated distance between the elements at the two index
positions. (round two decimal places)
    """
```

Test Cases:

```python
assert solution([3,-1,0,2,-9,-7], 0, 3)==1; assert solution([2,-3,-1,-1,-2,-2,3,-3], 0, 6)==19; assert ···
```

Incorrect Diagram Description Generated by Pixtral 12B:

Problem Restatement
The problem involves calculating a "distance" between two elements in a list of numbers. This distance is determined using specific mathematical operations defined in the image. The function `solution` takes three arguments:- `nums`: A list of floating-point numbers.
- `idx1`: The first index position (0-indexed).
- `idx2`: The second index position (0-indexed).

The function returns the calculated distance between the elements at the specified index positions, rounded to two decimal places.

Visual Facts
From the image, the following visual elements and facts are observed:

1. Each example consists of a list of numbers and a distance calculation.
2. The distance is calculated using squared differences between the elements at specific indices.
3. Each example provides a sequence of numbers and corresponding distances calculated using specific indices.
4. The calculations are shown in the image as:
   - Example 1: Distance between elements at indices 1 and 3.
   - Example 2: Distance between elements at indices 1, 3, and 6.
   - Example 3: Distance between elements at indices 1, 3, 5, 7, and 9.

Hallucination

Visual Patterns
From the visual facts, the following patterns and operations are inferred:

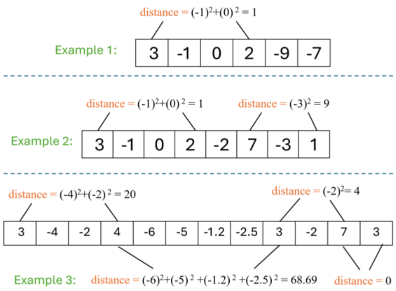
Vague Expression: Does not specify how to calculate the distance.

1. The distance is calculated using the sum of squared differences between the elements at the specified indices.
2. Each example shows the sum of squares for the differences:
   - Example 1: `distance = (-1)^2 + (0)^2 = 1`
   - Example 2: `distance = (-1)^2 + (0)^2 + (-3)^2 = 1 + 1 + 9 = 11`
   - Example 3: `distance = (-4)^2 + (-2)^2 = 16 + 4 = 20` and `distance = (-2)^2 = 4`
3. These distances appear to be cumulative sums of squared differences of elements at the specified indices.
4. For multiple indices, each squared difference is summed to produce the final distance.

This understanding of the operations and calculations helps in implementing the function `solution` as described in the provided function signature.

Figure 33: Example case on the same task in Figure 31, demonstrated by Pixtral 12B under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(nums: list[float], idx1: int, idx2: int) -> float:
    """
    Calculate the distance between two elements in a list of numbers.
    Parameters:
        nums (list[float]): A list of numbers.
        idx1 (int): The first index position. (0-indexed)
        idx2 (int): The second index position. (0-indexed)
    Returns:
        float: The calculated distance between the elements at the two index
positions. (round two decimal places)
    """
```

Test Cases:

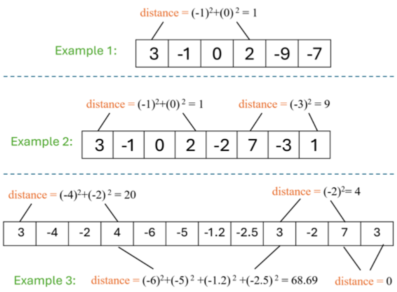
```python
assert solution([3,-1,0,2,-9,-7], 0, 3)==1; assert solution([2,-3,-1,-1,-2,-2,3,-3], 0, 6)==19; assert ···
```

Incorrect Code Solution Generated by OpenAI o1:

```
**Analysis**
From the examples in the image:

- They consistently show something like “distance = (x)² + (y)².”
- There is no visible square root; the result is simply the *sum of the squares* of the two chosen
elements.
- Finally, we round to two decimal places.

Hence, given a list `nums` and two indices `idx1`, `idx2`, we compute
\[
\text{distance} = \bigl(\text{nums[idx1]}\bigr)^2 + \bigl(\text{nums[idx2]}\bigr)^2
\]
and round it to two decimals.

```python
def solution(nums: list[float], idx1: int, idx2: int) -> float:
    """
    Calculate the 'distance' between two elements in a list of numbers,
    defined here as the sum of their squares (no square root).

    The result is rounded to 2 decimal places.

    Parameters:
        nums (list[float]): A list of numbers.
        idx1 (int): The first index position (0-based).
        idx2 (int): The second index position (0-based).

    Returns:
        float: The calculated distance, rounded to two decimals.
    """
    x = nums[idx1]
    y = nums[idx2]
    dist = x**2 + y**2
    return round(dist, 2)
```
```

Incorrect: Fail to understand the aggregation rules.

Incorrect: Fail to understand the aggregation rules.

Figure 34: Example case on the same task in Figure 31, demonstrated by OpenAI o1 under the textitV2C w/ CoT setting.

Image:



Function Signature:

```
def solution(nums: list[float], idx1: int, idx2: int) -> float:
    """
    Calculate the distance between two elements in a list of numbers.
    Parameters:
        nums (list[float]): A list of numbers.
        idx1 (int): The first index position. (0-indexed)
        idx2 (int): The second index position. (0-indexed)
    Returns:
        float: The calculated distance between the elements at the two index
positions. (round two decimal places)
    """
```

Test Cases:

```
assert solution([3,-1,0,2,-9,-7], 0, 3)==1; assert solution([2,-3,-1,-1,-2,-2,3,-3], 0, 6)==19; assert ···
```

Incorrect Code Solution Generated by QVQ 72B Preview:

```
**Final Solution**
To solve this problem, we need to calculate the distance between two elements in a list of numbers based
on the provided examples.

### Approach
1. **Understanding the Problem:**
   - We are given a list of floating-point numbers and two indices.
   - We need to calculate the "distance" between the elements at these indices.
   - The distance is defined based on the sum of squares of the elements at the given indices.
2. **Interpreting the Examples:**
   - Example 1: For indices 1 and 2 in `[3, -1, 0, 2, -9, -7]`, distance is `(-1)^2 + (0)^2 = 1 + 0 =
1`.
   - Example 2: For indices 1 and 2 in `[3, -1, 0, 2, -2, 7, -3, 1]`, distance is `(-1)^2 + (0)^2 = 1 +
0 = 1`; for indices 6 and 7, distance is `(-3)^2 + (1)^2 = 9 + 1 = 10`.
   - Example 3: For indices 4 and 7 in `[3, -4, -2, 4, -6, -5, -1.2, -2.5, 3, -2, 7, 3]`, distance is
`(-6)^2 + (-2.5)^2 = 36 + 2.25 = 38.25`.
```

Hallucination

```
3. **Function Signature:**
   - `def solution(nums: list[float], idx1: int, idx2: int) -> float:`

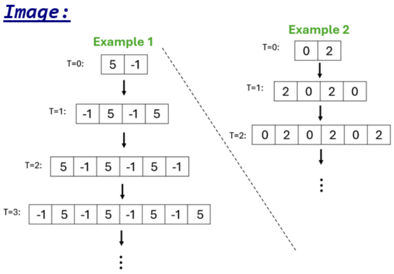
### Solution Code
```python
def solution(nums: list[float], idx1: int, idx2: int) -> float:
    # Ensure indices are within bounds
    if idx1 < 0 or idx1 >= len(nums) or idx2 < 0 or idx2 >= len(nums):
        raise IndexError("Indices out of bounds")
    # Calculate the sum of squares of elements at idx1 and idx2
    distance = nums[idx1]**2 + nums[idx2]**2
    # Round to two decimal places
    return round(distance, 2)
```
```

Incorrect: Fail to understand the aggregation rules.

```
### Explanation
1. **Bounds Check:**
   - We ensure that both indices are within the valid range for the list.
2. **Distance Calculation:**
   - Compute the square of each element at the given indices and sum them up.
3. **Rounding:**
   - Round the final distance to two decimal places to match the problem's requirement.
```

Figure 35: Example case on the same task in Figure 31, demonstrated by QVQ 72B Preview under the textitV2C w/ CoT setting.

Image:



Function Signature:

```python
def solution(nums: List[int], T: int) -> List[int]:
    """
    Returns the array at a specified time T.
    Parameters:
        nums (List[int]): The initial array of integers
        T (int): The target time point
    Returns:
        List[int]: The array at time T
    """
```

Test Cases:

```python
assert solution([5,-1], 1) == [-1, 5,-1, 5]; assert solution([0,2], 3) == [2,0,2,0,2,0,2,0]; assert ···
```

Incorrect Diagram Description Generated by Claude 3.5 Sonnet:

```
# Problem Restatement
Given an initial array of integers and a time point T, the problem requires transforming the array
according to a specific pattern over time steps until reaching time T. Each time step follows a
consistent transformation rule to generate the next
state of the array.

# Visual Facts
1. Two examples are shown, each starting with different initial arrays:
   - Example 1: Initial array [5, -1] at T=0
   - Example 2: Initial array [0, 2] at T=0
2. Time steps are labeled as T=0, T=1, T=2, T=3, etc.
3. Example 1 progression:
   - T=0: [5, -1]
   - T=1: [-1, 5, -1, 5]
   - T=2: [5, -1, 5, -1, 5, -1]
   - T=3: [-1, 5, -1, 5, -1, 5, -1, 5]
4. Example 2 progression:
   - T=0: [0, 2]
   - T=1: [2, 0, 2, 0]
   - T=2: [0, 2, 0, 2, 0, 2]

# Visual Patterns
1. Array Growth Pattern:
   - At each time step, the array size doubles
   - Length formula: Length(T) = 2^(T+1)
```

Incorrect: Length(T) should be 2(T+1)*

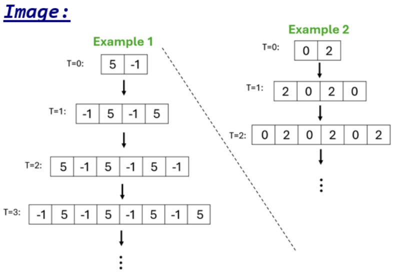
```
2. Element Arrangement Pattern:
   - Each element in the input array alternates in the output
   - For Example 1:
     * Elements 5 and -1 alternate throughout
```

Incorrect: The alternation order varies based on array size

```
     * Pattern repeats as: first element, second element, first element, second element...
   - For Example 2:
     * Elements 0 and 2 alternate throughout
     * Same alternating pattern applies
3. Value Transformation Rule:
   - No mathematical operations are performed on the values
   - Values are simply rearranged in an alternating pattern
   - The original values from T=0 are preserved throughout all time steps
4. Consistency:
   - The pattern is deterministic
   - The transformation rule is consistent across both examples
   - Each time step builds upon the previous step's result
```

Figure 36: Error case highlighting LMMs' limitations in understanding dynamic patterns with alternating elements and linear increments. The case shown for Claude 3.5 Sonnet is under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(nums: List[int], T: int) -> List[int]:
    """
    Returns the array at a specified time T.
    Parameters:
        nums (List[int]): The initial array of integers
        T (int): The target time point
    Returns:
        List[int]: The array at time T
    """
```

Test Cases:

```python
assert solution([5,-1], 1) == [-1, 5,-1, 5]; assert solution([0,2], 3) == [2,0,2,0,2,0,2,0]; assert ···
```

Incorrect Diagram Description Generated by Pixtral 124B:

```
# Problem Restatement
Given an initial array of integers and a time point T, the function is required to return the state of
the array at that specific time point T. The time evolution of the array follows a specific
transformation pattern demonstrated in the image, which needs to be identified and implemented.

# Visual Facts
1. **Initial States:**
   - Example 1 starts with the array \([5, -1]\) at T=0.
   - Example 2 starts with the array \([0, 2]\) at T=0.
2. **Transformation at T=1:**
   - In Example 1, the array \([5, -1]\) evolves into \([-1, 5, -1, 5]\).
   - In Example 2, the array \([0, 2]\) evolves into \([2, 0, 2, 0]\).
3. **Transformation at T=2:**
   - In Example 1, the array \([-1, 5, -1, 5]\) evolves into \([5, -1, 5, -1, 5, -1]\).
   - In Example 2, the array \([2, 0, 2, 0]\) evolves into \([0, 2, 0, 2, 0, 2]\).
4. **Continuation Pattern:**
   - The transformations continue for each subsequent time step, producing longer arrays.

# Visual Patterns
1. **Transformation Rule:**
   - At each time step, the array is transformed by duplicating the array and reversing the order of the
elements for the second half of the newly formed array.
   - Mathematically, for an array \(A_t\) at time \(t\), the array at time \(t+1\) is formed by
concatenating \(A_t\) with a reversed copy of \(A_t\).
```

Incorrect: Fail to understand the transformation rules.

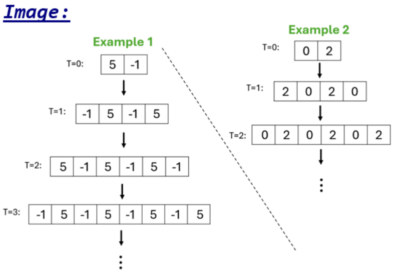
```
2. **Repeated Growth:**
   - The length of the array doubles at each transformation step.
   - For an initial array \(A_0\) of length \(n\), the length of the array at time \(t\) is \(n \times
2^t\).
```

Incorrect: Length(T) should be 2*(T+1)

```
3. **General Transformation Formula:**
   - If \(A_t\) is the array at time \(t\), then \(A_{t+1} = A_t + \text{reverse}(A_t)\).
   - Example: For \(A_0 = [a, b]\), \(A_1 = [a, b, b, a]\) and \(A_2 = [a, b, b, a, a, b, b, a]\).
4. **Binary Pattern Expansion:**
   - The expansion pattern observed is similar to creating a sequence where each element's position
follows a binary-like doubling pattern in its progression.

By carefully examining the image and understanding the transformation pattern, the problem can be solved
by implementing the transformation logic iteratively or recursively until the desired time point T is
reached.
```

Figure 37: Example case on the same task in Figure 36, demonstrated by Pixtral 124B under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(nums: List[int], T: int) -> List[int]:
    """
    Returns the array at a specified time T.
    Parameters:
        nums (List[int]): The initial array of integers
        T (int): The target time point
    Returns:
        List[int]: The array at time T
    """
```

Test Cases:

```python
assert solution([5,-1], 1) == [-1, 5,-1, 5]; assert solution([0,2], 3) == [2,0,2,0,2,0,2,0]; assert ···
```

Incorrect Diagram Description Generated by Pixtral 12B:

```
# Problem Restatement
The problem involves transforming an initial array of integers over a series of time steps, denoted by
\( T \). Given an initial array and a target time step \( T \), the goal is to return the array at that
specific time step. The transformation rules for each example are shown in the provided image, which
illustrates how the array evolves over time.

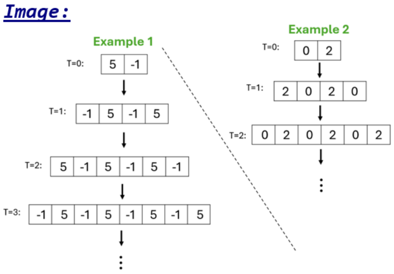
# Visual Facts
1. **Example 1:**
   - Initial array at \( T = 0 \): [5, -1]
   - Array at \( T = 1 \): [-1, 5, -1, 5]
   - Array at \( T = 2 \): [5, -1, 5, -1, 5, -1]
   - Array at \( T = 3 \): [-1, 5, -1, 5, -1, 5, -1]
2. **Example 2:**
   - Initial array at \( T = 0 \): [0, 2]
   - Array at \( T = 1 \): [2, 0, 2, 0]
   - Array at \( T = 2 \): [0, 2, 0, 2, 0, 2]
   - Arrays shown for \( T = 3 \) and beyond follow a similar pattern.
3. **Transformation Process:**
   - Each subsequent array is generated by duplicating and modifying the current array according to
specific rules.
   - Example 1 follows a pattern of adding the same value repeatedly.
   - Example 2 follows a pattern of shifting and duplicating values.

# Visual Patterns
1. **Example 1:**                              Incorrect: Fail to understand the transformation rules.
   - The transformation involves concatenating the initial array with itself and then shifting values
alternately.
   - Each subsequent array is formed by adding the initial array at the beginning and end.
   - The pattern involves shifting elements by 1 and inserting the corresponding values from the initial
array.
2. **Example 2:**                              Incorrect: Fail to understand the transformation rules.
   - The transformation involves shifting the array to the left and appending the initial array at the
end.
   - Each subsequent array is formed by moving all elements of the current array one position to the
left and appending the initial array at the end.
   - This pattern continues indefinitely, repeating the initial array's elements.

By observing these patterns, one can deduce the rules for transforming the array at each time step \( T
\). The solution involves implementing the identified transformation rules to generate the array at the
desired time point \( T \).
```

Figure 38: Example case on the same task in Figure 36, demonstrated by Pixtral 12B under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```
def solution(nums: List[int], T: int) -> List[int]:
    """
    Returns the array at a specified time T.
    Parameters:
        nums (List[int]): The initial array of integers
        T (int): The target time point
    Returns:
        List[int]: The array at time T
    """
```

Test Cases:

```
assert solution([5,-1], 1) == [-1, 5,-1, 5]; assert solution([0,2], 3) == [2,0,2,0,2,0,2,0]; assert ···
```

Correct Code Solution Generated by OpenAI o1:

```
**Analysis**
From the examples, each time step \(T\) corresponds to taking a “window” of length \(2 \times (T+1)\)
from an infinite repetition of the original two-element array, starting at index \(T\).  Concretely:

1. We have an initial array of length 2, say \(\bigl[x_0,x_1\bigr]\).
2. Imagine an infinite repeating sequence: \(\dots,x_0,x_1,x_0,x_1,x_0,x_1,\dots\).
3. At time \(T\), we skip the first \(T\) elements of this infinite sequence and then take the next
\(2\times (T+1)\) elements.

Visually, in **Example 1** with `nums=[5, -1]`:

- **\(T=0\)**: offset \(=0\), length \(=2\). We take the first 2 elements from \([5, -1, 5, -1,\dots]\),
yielding \([5, -1]\).
- **\(T=1\)**: offset \(=1\), length \(=4\). We start from index 1 (\(-1\)) and take 4 elements: \([-1,
5, -1, 5]\).
- **\(T=2\)**: offset \(=2\), length \(=6\). From index 2 (which is 5 again), the next 6 elements are
\([5, -1, 5, -1, 5, -1]\). (The figure shows a truncated 5-element snippet, but logically it continues.)

Similarly for **Example 2** (`nums=[0, 2]`), the pattern toggles every time step, with lengths 2, 4, 6,
8, etc.

```python
from typing import List

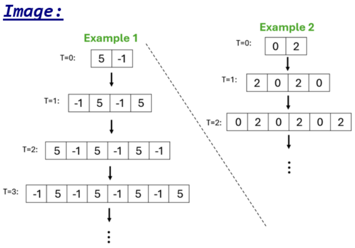
def solution(nums: List[int], T: int) -> List[int]:
    """
    Returns the array at a specified time T by viewing the original two-element
    array `nums` as an infinite repeating sequence and taking 2*(T+1) elements
    starting at index T.

    Parameters:
        nums (List[int]): The initial array of exactly two integers.
        T (int): The target time point.

    Returns:
        List[int]: The array of length 2*(T+1), starting from offset T in the
                   infinite repetition of nums.
    """
    # The length of the resulting array
    length = 2 * (T + 1)
    # Build the result by indexing into the repeating [nums[0], nums[1]] pattern
    result = [nums[(T + i) % 2] for i in range(length)]
    return result
```
```

Figure 39: Example case on the same task in Figure 36, demonstrated by OpenAI o1 under the *V2C w/ CoT* setting.

Image:



Function Signature:

```python
def solution(nums: List[int], T: int) -> List[int]:
    """
    Returns the array at a specified time T.
    Parameters:
        nums (List[int]): The initial array of integers
        T (int): The target time point
    Returns:
        List[int]: The array at time T
    """
```

Test Cases:

```python
assert solution([5,-1], 1) == [-1, 5,-1, 5]; assert solution([0,2], 3) == [2,0,2,0,2,0,2,0]; assert ···
```

Incorrect Code Solution Generated by QVQ 72B Preview:

```
## Final Solution
To solve this problem, we need to transform an initial array `nums` through a series of steps labeled by
a time parameter `T`. The transformation alternates between the original array and its rotation at each
step. The goal
is to generate the array at a specified time step `T`.

### Approach
1. **Initial Case (T=0):** Return the original array `nums`.
2. For subsequent time steps:
   - If `T` is even, use the original array `nums`.
   - If `T` is odd, use the array rotated by one position to the left.
3. Repeat this base array `(T + 1) // 2` times.
4. Extend this repeated array to the required length `len(nums) * (T + 1)` by duplicating the pattern.

### Solution Code

```python
from typing import List

def solution(nums: List[int], T: int) -> List[int]:
    if T == 0:
        return nums
    # Define a function to rotate the array to the left by one position
    def rotate_left(arr):
        return arr[1:] + arr[:1]
    # Determine the base array based on the parity of T
    if T % 2 == 0:
        base_array = nums
    else:
        base_array = rotate_left(nums)
    # Calculate how many times to repeat the base array
    repeats = (T + 1) // 2
    # Create the extended array
    extended_array = []
    for _ in range(repeats):
        extended_array.extend(base_array)
    # Ensure the array has the required length
    required_length = len(nums) * (T + 1)
    return extended_array[:required_length]
```

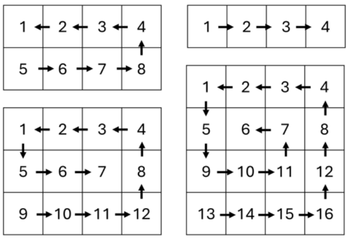
Explanation
- **Initial Case (T=0):** The array remains unchanged.
- **For Even T:** The base array is the original array `nums`.
- **For Odd T:** The base array is `nums` rotated by one position to the left.
- **Repetition:** The base array is repeated `(T + 1) // 2` times.
- **Extension:** The repeated array is extended to the required length `len(nums) * (T + 1)` by
duplicating the pattern.

This approach ensures that the array transformation follows the observed pattern in the examples
provided.
```

Implementation Error: The number of repeats should not be (T + 1) // 2, as this would produce an array of length 2 instead of 4 when T = 1 (as in Example 1).

Figure 40: Example case on the same task in Figure 36, demonstrated by QVQ 72B Preview under the *V2C w/ CoT* setting.

Image:



Function Signature:

```python
def solution(matrix: List[List[int]]) -> List[int]:
    """
    Given an M x N 2D matrix, traverse the matrix according to the spiral pattern
shown in the image
    Parameters:
    - matrix (List[List[int]]): A 2D list of integers representing the matrix.
    Returns:
    - A list of integers representing the elements of the matrix in the order as
shown in the image
    """
```

Test Cases:

```python
assert solution([[1,2]])==[1,2]; assert solution([[1,2],[3,4],[5,6],[7,8]])==[7,8,6,4,2,1,3,5]; assert ···
```

Incorrect Diagram Description Generated by Claude 3.5 Sonnet:

 # Problem Restatement

The problem requires implementing a function that traverses a matrix following specific arrow patterns shown in the image. The function takes a 2D matrix as input and should return a list of integers representing the elements in the order they are visited according to the arrow patterns demonstrated in the image.

 # Visual Facts

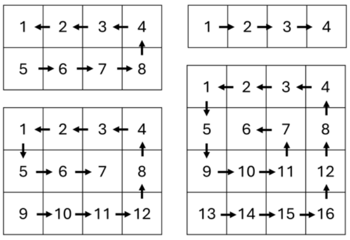
1. The image shows four different matrix examples with arrows indicating traversal paths:
   - Top left: 2x4 matrix
   - Top right: 1x4 matrix
   - Bottom left: 3x4 matrix
   - Bottom right: 4x4 matrix
2. Each cell contains a sequential number starting from 1
3. Arrows connect adjacent cells indicating the direction of traversal
4. The patterns show both horizontal and vertical movements
5. All examples demonstrate consistent numbering (sequential integers)

 # Visual Patterns

1. Traversal Pattern Rules:
   - Horizontal traversals alternate between left-to-right and right-to-left
   - Vertical traversals are consistently upward at the rightmost column
   - When multiple rows exist, the leftmost column shows downward traversal
   - The pattern creates a spiral-like path through the matrix
2. Common Characteristics: ***Incorrect: The traversal starts from the bottom-left cell***
   - The traversal always starts from the top-left cell (1)
   - The pattern follows a snake-like movement horizontally
   - Vertical movements occur at the edges of the matrix
   - The traversal continues until all cells are visited exactly once
3. Pattern Variations: ***Incorrect: Vertical movements may occur in any column***
   - The pattern adapts to different matrix dimensions (1x4, 2x4, 3x4, 4x4)
   - The basic rules remain consistent regardless of matrix size
   - The number of vertical transitions depends on the matrix height
   - The number of horizontal traversals depends on the matrix width
4. Order Properties:
   - Each number is visited exactly once
   - The traversal creates a continuous path through the matrix
   - The pattern ensures complete coverage of all matrix elements
   - The order of visited cells follows the arrow directions strictly

Figure 41: Error case illustrating LMMs' difficulties in recognizing and reasoning about complex geometric arrangements, particularly spiral and circular patterns. The case shown for Claude 3.5 Sonnet is under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(matrix: List[List[int]]) -> List[int]:
    """
    Given an M x N 2D matrix, traverse the matrix according to the spiral pattern shown in the image
    Parameters:
    - matrix (List[List[int]]): A 2D list of integers representing the matrix.
    Returns:
    - A list of integers representing the elements of the matrix in the order as shown in the image
    """
```

Test Cases:

```python
assert solution([[1,2]])==[1,2]; assert solution([[1,2],[3,4],[5,6],[7,8]])==[7,8,6,4,2,1,3,5]; assert ···
```

Incorrect Diagram Description Generated by Pixtral 124B:

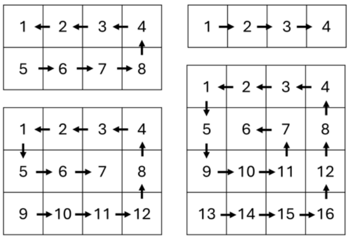
```
# Problem Restatement
Given a 2D matrix (a list of lists of integers), the problem asks us to traverse this matrix in a specific spiral pattern. The objective is to return a list of integers that represent the elements of the matrix in the order traversed by this spiral pattern as shown in the image.

# Visual Facts
The image provides several illustrations of matrices with arrows indicating the order of traversal in a spiral pattern. Here are the observed facts:
1. **Matrix Dimensions and Elements**:
    - The matrices shown are 2x4, 2x4, 3x4, and 4x4 in size.
    - Each matrix contains sequential integers starting from 1.
2. **Traversal Directions**:
    - The traversal starts from the top-left corner of the matrix.
    - Arrows indicate the following movements: right, down, left, and up.
    - The direction changes each time the edge of the matrix or the previously traversed path is reached.
3. **Traversal Paths**:
    - For the 2x4 matrix, the traversal first goes right across the top row, then down the last column, then left across the bottom row, and finally up the first column.
    - For the 3x4 matrix, the traversal includes the rightward movement across the top row, downward movement in the last column, leftward movement across the second row, upward movement in the first column, and so on.
    - For the 4x4 matrix, the pattern continues similarly, with additional inner rows and columns being traversed in a spiral manner.

# Visual Patterns
Based on the visual facts observed:
1. **Spiral Traversal Algorithm**:
    - Start from the top-left element.
    - Move right across the top row until the last element of the row is reached.
    - Move down the last column until the bottom-most element is reached.
    - Move left across the bottom row until the first element of the row is reached.
    - Move up the first column until the top-most element of the next inner row is reached.
    - Repeat this cycle for the inner sub-matrix until all elements are traversed.
2. **Dynamic Boundaries**:
    - The boundaries (top, bottom, left, right) of the current sub-matrix are reduced after each complete traversal of a layer.
    - The right boundary decreases after a rightward traversal.
    - The bottom boundary decreases after a downward traversal.
    - The left boundary increases after a leftward traversal.
    - The top boundary increases after an upward traversal.
3. **Edge Conditions**:
    - After traversing the outermost layer, the process continues with the next inner layer, reducing the matrix's effective size progressively.
    - The process ends when all elements have been visited, effectively reducing the matrix boundaries to converge at a point (if the matrix dimensions are odd) or traversing the last two central elements (if the matrix dimensions are even).
```

Incorrect: The traversal starts from the bottom-left cell

Hallucination

Figure 42: Example case on the same task in Figure 41, demonstrated by Pixtral 124B under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(matrix: List[List[int]]) -> List[int]:
    """
    Given an M x N 2D matrix, traverse the matrix according to the spiral pattern shown in the image
    Parameters:
    - matrix (List[List[int]]): A 2D list of integers representing the matrix.
    Returns:
    - A list of integers representing the elements of the matrix in the order as shown in the image
    """
```

Test Cases:

```python
assert solution([[1,2]])==[1,2]; assert solution([[1,2],[3,4],[5,6],[7,8]])==[7,8,6,4,2,1,3,5]; assert ···
```

Incorrect Diagram Description Generated by Pixtral 12B:

 # Problem Restatement

The problem involves traversing a given M x N 2D matrix according to a spiral pattern. The function `solution` takes a 2D list of integers (matrix) and returns a list of integers in the order specified by the spiral traversal as shown in the image.

 # Visual Facts

1. The image shows a matrix divided into a grid of cells with numbers inside each cell.
2. Arrows within each cell indicate the direction of traversal.
3. Each traversal moves from one cell to the next in a spiral pattern: starting from the top-left corner and moving outward.
4. The numbers inside the cells indicate the sequence in which they will be visited.
5. There are multiple matrices of different sizes shown:
   - A 2x2 matrix.
   - A 3x3 matrix. ***Hallucination***
   - A 4x4 matrix.
   - A 4x4 matrix with an extended spiral path.
6. The traversal pattern is consistent across all matrices: top-right, right-bottom, bottom-left, and left-top.

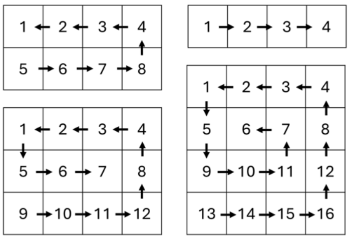
 # Visual Patterns ***Incorrect: The traversal starts from the bottom-left cell in counter-clock order***

1. The traversal pattern follows a clockwise spiral starting from the top-left corner.
2. The spiral path alternates between horizontal and vertical movements.
3. Each layer of the spiral is traversed in a specific order:
   - Top row from left to right.
   - Rightmost column from top to bottom.
   - Bottom row from right to left.
   - Leftmost column from bottom to top.
4. This pattern continues until all elements in the matrix have been visited.
5. The sequence of numbers collected from the traversal will follow the spiral pattern as depicted.

By understanding these visual facts and patterns, one can implement a function to traverse the matrix in the specified spiral order.

Figure 43: Example case on the same task in Figure 41, demonstrated by Pixtral 12B under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(matrix: List[List[int]]) -> List[int]:
    """
    Given an M x N 2D matrix, traverse the matrix according to the spiral pattern shown in the image
    Parameters:
    - matrix (List[List[int]]): A 2D list of integers representing the matrix.
    Returns:
    - A list of integers representing the elements of the matrix in the order as shown in the image
    """
```

Test Cases:

```python
assert solution([[1,2]])==[1,2]; assert solution([[1,2],[3,4],[5,6],[7,8]])==[7,8,6,4,2,1,3,5]; assert ···
```

Incorrect Code Solution Generated by OpenAI o1:

```
### Step-by-Step Analysis
From the image, the task is to traverse a matrix in a spiral order. The order in which the elements are traversed can be described as follows:
```

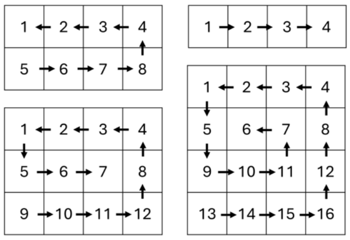
Incorrect: The traversal starts from the bottom-left cell

```
1. **Start from the top-left corner of the matrix** (row 0, column 0).
2. **Move right** across the topmost row.
3. **Move down** along the rightmost column.
4. **Move left** across the bottommost row (if more rows are left).
5. **Move up** along the leftmost column (if more columns are left).
6. **Repeat** this pattern, shrinking the boundary of traversal by excluding the outermost row and column until all elements are visited.
This spiral order traversal involves the following boundaries:
- `top` (initially 0),
- `bottom` (initially the last row index),
- `left` (initially 0),
- `right` (initially the last column index).
These boundaries are adjusted after each movement:
- After moving right, increment the `top` boundary.
- After moving down, decrement the `right` boundary.
- After moving left, decrement the `bottom` boundary.
- After moving up, increment the `left` boundary.
This loop continues until all elements are visited.
### Code Implementation
```

```python
def solution(matrix: List[List[int]]) -> List[int]:
    if not matrix:
        return []
    result = []
    top, bottom, left, right = 0, len(matrix) - 1, 0, len(matrix[0]) - 1
    while top <= bottom and left <= right:
        for i in range(left, right + 1):
            result.append(matrix[top][i])
        top += 1
        for i in range(top, bottom + 1):
            result.append(matrix[i][right])
        right -= 1
        if top <= bottom:
            for i in range(right, left - 1, -1):
                result.append(matrix[bottom][i])
            bottom -= 1
        if left <= right:
            for i in range(bottom, top - 1, -1):
                result.append(matrix[i][left])
            left += 1
    return result
```

Figure 44: Example case on the same task in Figure 41, demonstrated by OpenAI o1 under the *V2C w/ CoT* setting.

Image:



Function Signature:

```python
def solution(matrix: List[List[int]]) -> List[int]:
    """
    Given an M x N 2D matrix, traverse the matrix according to the spiral pattern
shown in the image
    Parameters:
    - matrix (List[List[int]]): A 2D list of integers representing the matrix.
    Returns:
    - A list of integers representing the elements of the matrix in the order as
shown in the image
    """
```

Test Cases:

```python
assert solution([[1,2]])==[1,2]; assert solution([[1,2],[3,4],[5,6],[7,8]])==[7,8,6,4,2,1,3,5]; assert ···
```

Unfinished Code Solution Generated by QVQ 72B Preview:

[...above is a very long CoT generation process...]

Alternatively, maybe the traversal is row by row for matrices where the number of rows is greater than or equal to the number of columns, and spiral otherwise.

But that also seems arbitrary.

Incorrect: The traversal pattern is irrelevant to row/column numbers

Given the confusion, perhaps the best approach is to implement a general spiral traversal algorithm, as initially thought.

Here's a more detailed implementation:

```python
def solution(matrix: List[List[int]]) -> List[int]:
    if not matrix or not matrix[0]:
        return []
    rows, cols = len(matrix), len(matrix[0])
    result = []
    top, bottom = 0, rows - 1
    left, right = 0, cols - 1
    while top <= bottom and left <= right:
          for col in range(left, right + 1):
            result.append(matrix[top][col])
        top += 1
        for row in range(top, bottom + 1):
            result.append(matrix[row][right])
        right -= 1
        if top <= bottom:
            for col in range(right, left - 1, -1):
                result.append(matrix[bottom][col])
            bottom -= 1
        if left <= right:
              for row in range(bottom, top - 1, -1):
                result.append(matrix[row][left])
            left += 1
    return result
```

But given that the examples show row by row traversal, perhaps the problem expects a row by row traversal.

Alternatively, maybe the spiral traversal is supposed to alternate directions based on the current layer.

I need to make a decision here.

Given the confusion between row by row and spiral traversal, and considering that the examples show row by row traversal, I will proceed with implementing a row by row traversal.

Here's the final code:

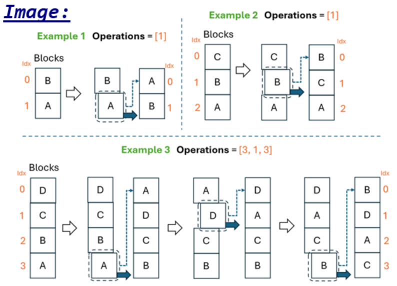
```python
def solution(matrix: List[List[int]]) -> List[int]:
    return [num for row in matrix for num in row]
```

This should satisfy the problem's requirements based on the examples provided.

[...unfinished CoT generation...]

Figure 45: Example case on the same task in Figure 41, demonstrated by QVQ 72B Preview under the *V2C w/ CoT* setting.

Image:



Function Signature:

```python
def solution(initial_blocks: List[str], operations: List[int]) -> List[str]:
    """
    return the transformed blocks after applying the operations.
    Args:
    - initial_blocks: List of strings representing the initial blocks
    - operations: List of integers representing the operations to be applied
    Returns:
    - List of strings representing the blocks after all operations
    """
```

Test Cases:

```python
assert solution(["C","B","D","A"],[1,3,2])==["C","A","B","D"];assert solution(["B","C"],[1])==["C","B"];assert ···
```

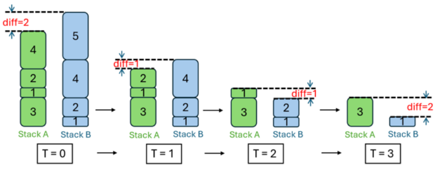
Incorrect Diagram Description Generated by Claude 3.5 Sonnet:

```
# Problem Restatement
The problem involves transforming a vertical arrangement of blocks (represented by letters) through a
series of operations. Each operation is specified by an index number that triggers a specific
transformation pattern. The
function needs to calculate the final arrangement of blocks after applying all specified operations in
sequence.

# Visual Facts
1. Each example shows blocks arranged vertically with index positions (0 to n-1)
2. Each block contains a single letter (A, B, C, D)
3. Example 1 shows:
   - Initial state: [B, A]
   - Operations = [1]
   - Final state: [A, B]
4. Example 2 shows:
   - Initial state: [C, B, A]
   - Operations = [1]
   - Final state: [B, C, A]
5. Example 3 shows:
   - Initial state: [D, C, B, A]
   - Operations = [3, 1, 3]
   - Final state: [B, D, A, C]
6. Each transformation is shown with:
   - Dotted boxes indicating moving blocks
   - Blue arrows showing movement direction
   - White arrows showing sequence of steps

# Visual Patterns
1. Operation Pattern:
   - When an index i is given in operations:
     - The block at index i moves up
     - The block at index i-1 moves down
     - They effectively swap positions
2. Sequential Processing:
   - Operations are applied one at a time
   - Each operation affects exactly two adjacent blocks
   - The result of one operation becomes the input for the next
3. Movement Rules:
   - Only adjacent blocks can be swapped
   - The operation index refers to the lower block in the swap
   - Blocks maintain their integrity (don't split or merge)
4. Transformation Properties:
   - Each operation preserves all blocks (no deletion/addition)
   - Multiple operations can affect the same blocks repeatedly
   - The final arrangement depends on the sequence of operations
```

Incorrect: All blocks with index < i moves down, not just swapping i and i-1

Figure 46: Error case highlighting the gap between human intuition and LMM performance on seemingly straightforward visual reasoning tasks. The case shown for Claude 3.5 Sonnet is under the *V2T2C w/ SC* setting.

Image:



Function Signature:

```python
def solution(a: List[int], b: List[int], t: int) -> int:
    """
    Calculate the `diff` value at a specific time.
    Parameters:
        a (List[int]): A stack, where index 0 represents the top element.
        b (List[int]): B stack, where index 0 represents the top element.
        t (int): Time at which to calculate the `diff`
    Returns:
        int: The `diff` value at time t
    """
```

Test Cases:

```python
assert solution([4,2,1,3], [5,5,2,1], 0) == 3; assert solution([4,2,1,3], [5,5,2,1], 3) == 2; assert ···
```

Incorrect Diagram Description Generated by Claude 3.5 Sonnet:

```
# Problem Restatement
The problem involves calculating the difference between two stacks (A and B) at a specific time point T.
The stacks contain integer values, and we need to track how they change over time and compute their
difference at a given moment. The difference appears to be measured between the topmost elements of both
stacks.

# Visual Facts
1. The image shows a sequence of four time steps (T=0 to T=3)
2. There are two stacks in each step:
   - Stack A (green blocks)
   - Stack B (blue blocks)
3. Initial state (T=0):
   - Stack A: [4,2,1,3] (top to bottom)
   - Stack B: [5,4,2,1] (top to bottom)
4. Each state shows a "diff=" value at the top
5. Elements disappear from the top of both stacks over time:
   - T=1: Stack A loses top elements, becomes [2,1,3]
   - T=1: Stack B loses top element, becomes [4,2,1]
   - T=2: Stack A loses more elements, becomes [1,3]
   - T=2: Stack B loses elements, becomes [2,1]
   - T=3: Stack A has only [3] remaining
   - T=3: Stack B has only [1] remaining

# Visual Patterns
1. The difference value ("diff") is calculated between the top elements of both stacks:
   - T=0: diff=2 (|4-5|=1)   Incorrect: diff is calculated between the total height of both stacks
   - T=1: diff=1 (|2-4|=2)
   - T=2: diff=1 (|1-2|=1)
   - T=3: diff=2 (|3-1|=2)
2. Stack Reduction Pattern:
   - Elements are removed from the top of both stacks over time
   - The removal rate may vary between stacks
   - The process continues until reaching time T or until stacks cannot be reduced further
3. Difference Calculation Rule:
   - At any time T, the difference is calculated as the absolute value between the top elements of both
stacks
   - The function needs to determine what the stacks look like at time T and calculate this difference
```

Figure 47: Another case highlighting the gap between human intuition and LMM performance on seemingly straightforward visual reasoning tasks. The case shown for Claude 3.5 Sonnet is under the *V2T2C w/ SC* setting.