Learning Task Decomposition to Assist Humans in Competitive Programming

Jiaxin Wen[1,2], **Ruiqi Zhong**[3], **Pei Ke**[1,2,†], **Zhihong Shao**[1,2], **Hongning Wang**[1,2] , **Minlie Huang**[1,2,†]
[1]The CoAI group, Tsinghua University, Beijing, China
[2]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[3] University of California, Berkeley
wenjx22@mails.tsinghua.edu.cn, aihuang@tsinghua.edu.cn

Abstract

When using language models (LMs) to solve complex problems, humans might struggle to understand the LM-generated solutions and repair the flawed ones. To assist humans in repairing them, we propose to automatically decompose complex solutions into multiple simpler pieces that correspond to specific subtasks. We introduce a novel objective for learning task decomposition, termed *assistive value* (`AssistV`), which measures the feasibility and speed for humans to repair the decomposed solution. We collect a dataset of human repair experiences on different decomposed solutions. Utilizing the collected data as in-context examples, we then learn to critique, refine, and rank decomposed solutions to improve `AssistV`. We validate our method under competitive programming problems: under 177 hours of human study, our method enables non-experts to solve 33.3% more problems, speeds them up by 3.3x, and empowers them to match unassisted experts.

1 Introduction

With their increased capabilities, language models (LMs) are used to perform increasingly complex and high-impact problems (Trinh et al., 2024; Li et al., 2022; Huang et al., 2023b). The scalable oversight challenge emerges (Amodei et al., 2016): LMs might fail to provide reliable solutions for these problems, but it is also difficult for humans to evaluate and improve LMs' solutions due to therequired significant time and expertise. One strategy to assist humans is task decomposition: as shown in Figure 1, humans can more easily understand and repair complex solutions after they are decomposed into simpler pieces that correspond to specific subtasks (Lee and Anderson, 2001).

However, not all decompositions are helpful, and it is challenging for humans to design an effective one(Connolly and Dean, 1997; Selby, 2015; Correa et al., 2023). For example, Charitsis et al. (2023) shows that improper decomposition designed by novice programmers can impede human debugging performance. To decompose better, we need methods beyond using fixed heuristic rules (Wu et al., 2021) or learning from author-crafted demonstrations (Yao et al., 2022; Zelikman et al., 2023).

In this paper, we introduce a novel objective for learning task decomposition: *assistive value* (`AssistV`, Equation 1), which measures the feasibility and speed of humans to repair a decomposed solution in the actual annotation process (Figure 1 right). To improve `AssistV`, we first collect a dataset of decompositions, measure their `AssistV`, and ask human annotators to provide a natural language critique on what makes a decomposition (not) helpful. Then we design a three-stage process to generate high-`AssistV` decompositions, where each stage is implemented by an LLM that learns from our dataset in context: 1) learn a critique model π_{critique} for predicting human critique on how to improve the initial decomposition for higher `AssistV`, 2) learn a refine model π_{refine} to incorporate the critique to refine the initial decomposition, and 3) learn a rank model π_{rank} to select a decomposition with high `AssistV`.

We chose competitive programming as a testbed to validate our method for scalable oversight, since it is a challenging task that both LMs and humans alone struggle to solve. We recruit 30 Python programmers, including 11 experts [1] and 19 non-experts, to repair model-generated program solutions for competitive coding problems, resulting in a total of 177 worker hours. Experiment results show that our method enables humans to solve 33.3% more problems, speeds up non-experts and experts by 3.3 and 2.4 times, and assists non-experts to match the performance of non-assisted

[†] Corresponding author

[1]This group includes medalists in National or International Olympiad in Informatics.

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 11700–11723

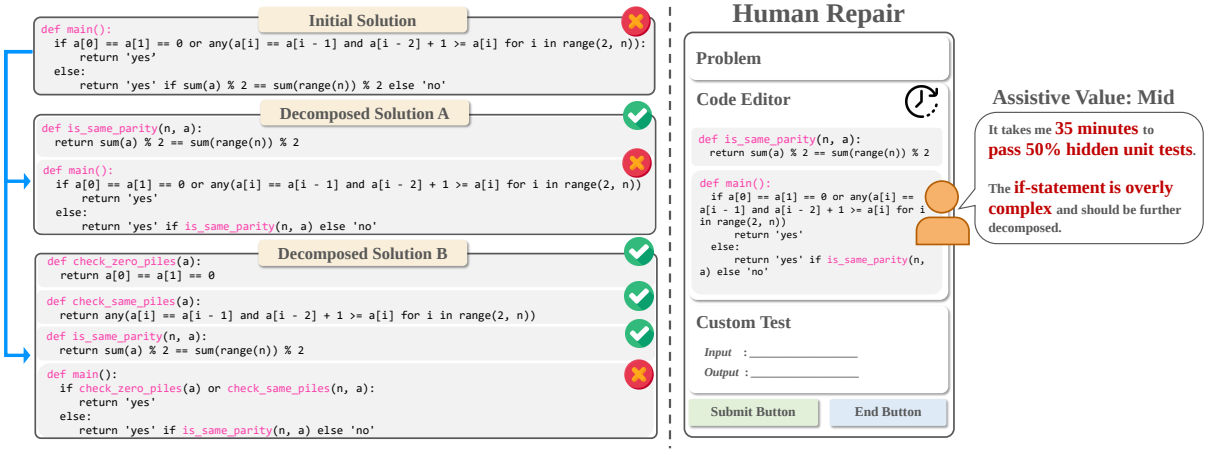

Figure 1: Decompositions can assist humans in supervising models to solve complex problems. **Left**: To solve a problem, an LM would first propose an initial solution; our goal is to decompose the initial solution into multiple simpler pieces such that humans can repair it more easily. (Sub)Task descriptions are truncated for brevity. **Right**: The *assistive value* (`AssistV`) of a decomposition measures the feasibility and speed of humans to repair the decomposed solution in the actual problem-solving process. For example, Decomposition B has a higher `AssistV` value than A in practice, as it further decomposes the complex if-statement into two simpler subtasks, which effectively assists humans in identifying a missing condition.

experts. We then analyze LMs' ability to select decompositions with higher `AssistV`: while humans' intuitive judgment is not better than random (49.5%), GPT-3.5-Turbo achieves 62.5% accuracy by learning from human repair experiences, and GPT-4 is 15.6% better. This result indicates that LMs could learn to perform better than humans at predicting what is more helpful for humans.

Our core contributions are:

- We assist humans with scalable oversight via automated task decomposition.
- We introduce a novel objective for learning task decomposition: `AssistV`, and we propose a three-stage method to produce a high-`AssistV` decomposition by learning to critique, refine, and rank decompositions.
- We show that our method is effective in competitive programming, improving human supervision performance and bridging the expertise gap.

Overall, even when LMs cannot solve the problem themselves, they can still learn to assist humans. By learning from human experiences to repair solutions, more capable models can predict assistive values more accurately and sometimes more accurately than humans, highlighting the potential of learning-based methods to assist humans.

2 Methodology

2.1 Task Definition

Our objective is to assist human labelers in repairing model-generated solutions to complex tasks by learning a decomposition model. This model decomposes input solutions into multiple easier-to-repair pieces corresponding to specific subtasks.

Formally, given a problem P and an initial model solution A, we aim to transform A into a decomposed solution A_d to improve its assistive value η, as defined by:

$$\eta(A_d) = \int_0^T \texttt{eval}(A_d^t)\mathrm{d}t \tag{1}$$

where A_d^t denotes the solution repaired by humans after spending time t from its initial solution A_d, and $\texttt{eval}(\cdot)$ is a metric of solution quality. In competitive programming, we set $\texttt{eval}(\cdot)$ as the passing rates on unit test cases. Assistive value $\eta(A_d)$ summarizes the solution quality over the human repairing process starting from A_d (Bradley, 1997), with example trajectories shown in Figure 4. A higher value of $\eta(A_d)$ indicates that assisted humans can more easily repair A_d, thus efficiently providing a high-quality label to the problem P.

While there are diverse ways to decompose one single solution as shown in Figure 1, effective decomposition that improves human problem-solving performance is non-trivial and can even be challenging for humans to devise (Charitsis et al., 2023).

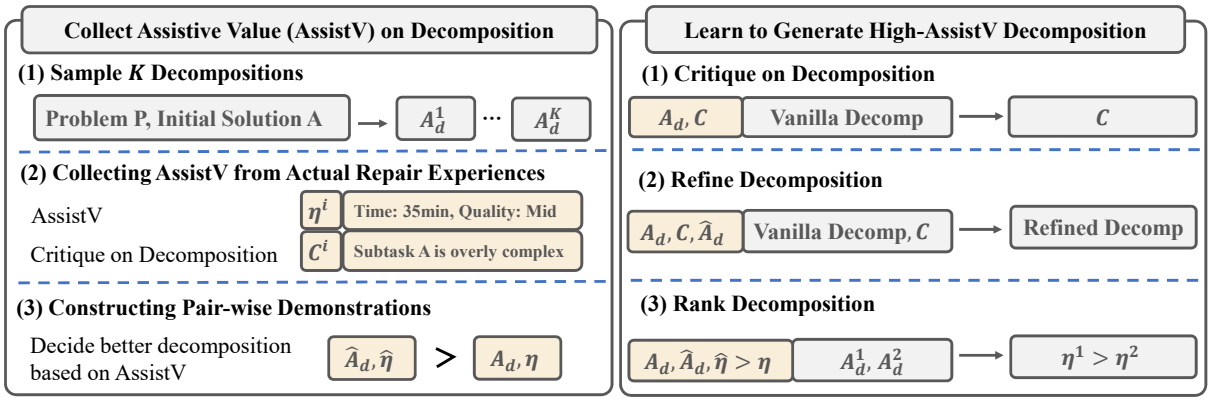

Figure 2: Method overview. **Left**: we sample multiple decompositions from LMs and evaluate them based on assistive value η and critique C. We then construct pair-wise decompositions to demonstrate the difference between low- and high-`AssistV` decompositions. **Right**: Starting from a vanilla decomposition generated by naively prompting LMs, we use the collected pair-wise data as in-context demonstrations to learn three models to critique, refine, and rank decompositions to better assist humans.

To decompose better, we need methods beyond prior methods that rely on fixed heuristic rules or learn from author-crafted demonstrations. In this paper, we propose to improve the assistive value of task decomposition by learning from human problem-solving experiences.

Specifically, we first construct a training set of human repair experiences between different decomposed solutions $D_{train} = \{(\hat{A}_d, A_d, C, P)\}$, where $\eta(\hat{A}_d) > \eta(A_d)$ and C is a natural language critique that explains why the decomposition A_d leads to a lower assistive value than $\hat{A}_d$. Utilizing the collected training set, we learn to critique, refine, and rank decompositions to improve assistive value. Figure 2 presents the overview of our framework.

2.2 Data Collection

Data Preparation For each problem P and an initial solution A, we obtain K different decomposed solutions $\{A_d^1, \cdots, A_d^K\}$ by sampling from various LMs with few-shot prompting. See more details about data preparation in Appendix A.2.

Recruiting Human Annotators We recruit annotators from college students. We conduct a pre-survey about their background and divide them into two groups based on their programming skills. We recruit 11 expert annotators who can solve Leetcode hard-level problems. Especially, 6 of them are medalists in the National or International Olympiad in Informatics. We recruit 19 non-expert annotators who can solve Leetcode medium-level problems but hardly solve hard-level problems. We conduct a warm-up test to train annotators to use our experiment environment. We further verify annotators' programming skills based on their performance in the warm-up test.

Collecting Assistive Value Following our definition of assistive value in Equation 1, we collect assistive value labels of different decompositions in the actual human annotation process. Specifically, for each sampled decomposed solution A_d^i, we collect the following labels:

- **Assistive Value η^i:** We calculate the assistive value of A_d^i following the definition described in Equation 1.
- **Critique on decomposition C^i:** After repairing, we ask annotators to provide natural language critique C^i on how decomposition assists or hinders their debugging.

Constructing Pair-wise Demonstrations We make (A_d^j, A_d^i) a comparison pair if they meet two requirements: (1) There is a substantial difference in assistive value between η^j and η^i. (2) The advantages in critique C^i match the disadvantages in C^j[2], and hence C^i and C^j explain why A_d^j leads to more efficient human repair than A_d^i.

2.3 Models

Starting with a vanilla decomposition generated by naively prompting LMs, we predict human critiques on it with π_{critic}, refine the decomposition according to the critique with π_{refine}, and rank

[2]In our experiments, we manually inspect human critiques and perform matching, as we only need a few examples for in-context learning. See Appendix A.2 for more discussions.

candidate decompositions to select the final high-AssistV output with π_{rank}. Figure 2 presents the overview of our framework. Notably, we introduce critique as an integral step instead of directly generating refined decomposition since it provides enriched information for models to learn how decomposition can achieve improved assistive value. Additionally, it also enables controllable decomposition as humans can manually edit critiques (e.g., requiring specific decomposition on certain complex subtasks).

We next describe the detailed inputs and outputs of each model:

- **Critic Model** $\pi_{\textbf{critic}}$: It takes a problem P and a decomposed solution A_d as inputs, and outputs critique on how to improve A_d for higher assistive value.
- **Refine Model** $\pi_{\textbf{refine}}$: It takes a problem P, a decomposed solution A_d, and a critique C as inputs, and outputs a refined decomposed solution $\hat{A}_d$.
- **Ranking Model** $\pi_{\textbf{rank}}$: It takes a problem P and two decomposed solutions A_d^1, A_d^2 as inputs, and outputs a ranking that predicts which decomposition leads to higher assistive value.

For model training, inspired by recent works showing that modern LMs can learn to critique and refine model outputs via in-context learning (Bai et al., 2022; Sun et al., 2023), we use this approach to learn our three models, where the collected training data $D_{train} = \{(\hat{A}_d, A_d, C, P)\}$ are formatted into in-context examples for each model. Example prompts are shown in Appendix D.

At inference time, we apply π_{critic}, followed by π_{refine}, to produce a refined decomposition set $\{\hat{A}_d\}$, and then use π_{rank} to select a decomposition A_d^* among them as the final decomposition.

3 Experiments

3.1 Setup

Benchmark We conduct experiments with the problems from two widely adopted competition-level code generation benchmarks, namely APPS (Hendrycks et al., 2021) and Code-Contests (Li et al., 2022). For human evaluation, we filter those problems where the model-generated program directly passes all test cases and randomly sample 30 problems as the test data.

Metric We measure the supervision quality in competitive programming by programs' passing rates on unit test cases. Specifically, we aggregate the program's performance on test cases with two metrics. **Test Case Average Accuracy** computes the average fraction of test cases passed among all the test cases. **Strict Accuracy** computes the average fraction of programs that pass all test cases.

Baselines We consider three baselines:

- **Initial**: It prompts a code language model $\mathcal{M}$ with the problem P to generate a solution A without explicit instructions for decomposition.
- **Heuristic Decomposition**: Inspired by McCabe's cyclomatic complexity (McCabe, 1976), a widely adopted metric for measuring code complexity in software engineering, we implement a heuristic baseline to decompose a complex program into simpler pieces. Specifically, we decompose each if-statement as well as for or while loops into a separate function since these code structures contribute to higher cyclomatic complexity. We then generate post-hoc subtask descriptions using GPT-4.
- **Vanilla Decomposition**: It prompts a code language model $\mathcal{M}$ to perform decomposition with basic author-crafted demonstrations without learning from human feedback.

Annotation Procedure We conduct experiments based on an internal Online Judge system. Problems are randomly assigned to each annotator while ensuring that they have not seen the assigned problem before (if they have, the problem will be reassigned) and never repeatedly debug the same problem. Next, given a competition-level problem, several exemplified public test cases, and a solution, labelers are required to perform debugging. See Appendix C for more annotation details.

Implementation Details We use GPT-4 (OpenAI, 2023) as our default backbone model $\mathcal{M}$ for code generation and decomposition due to its superior in-context learning capabilities. To focus our evaluation on the impact of decomposition, we ensure the consistency between initial and decomposed solutions based on their outputs on test cases. If the consistency check fails, we reject the decomposition and retry within a sampling budget. The consistency check is applied in all baselines for fair comparisons. See Appendix A.1 for more implementation details.

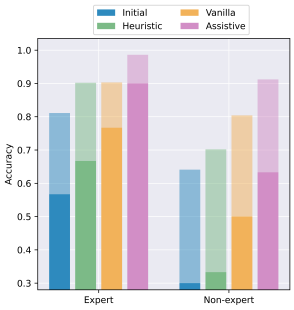

Figure 3: Humans provide higher-quality labels with decomposition. Dark color denotes strict accuracy of human-repaired programs; light color denotes test case average accuracy.

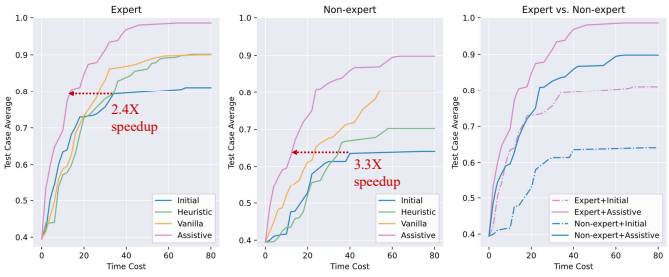

Figure 4: Decomposition improves human efficiency. We plot the relationship between the human-repaired program's test case average accuracy and human time cost.

3.2 Assisting Human Supervision

We evaluate the effectiveness of decomposition in aiding humans to repair programs and provide reliable supervision signals.

3.2.1 Quantitative Analysis

Supervising competitive programming is extremely challenging In our experiments, highly experienced experts spend at least 68 minutes repairing 56.7% of the model-generated programs, and non-experts spend at least 74 minutes repairing 30% of the model-generated programs. These results indicate that competitive programming serves as a good testbed for studying scalable oversight.

Decomposition improves human efficiency Figure 4 demonstrates that humans assisted with our decomposition model significantly outperform those without the assistance in terms of the speed to repair model-generated programs. For instance, the time required to collect repaired programs with 68% test case accuracy from non-experts is reduced from 40 minutes to 12 minutes, achieving a 3.3× speedup. Paired t-tests confirm that these efficiency gains are statistically significant ($p < 0.005$) for both experts and non-experts. These results indicate that our decomposition model can effectively improve the efficiency of human labelers.

Humans provide higher-quality labels with decomposition Beyond efficiency, we evaluate how decomposition impacts the final quality of human labels. As shown in Figure 3, our decomposition model enables humans to repair more model-generated programs. For instance, decomposition improves the strict accuracy of repaired programs from 56.7% to 90% for experts and from 30.0% to 63.3% for non-experts. Paired t-tests confirm the significance ($p < 0.01$) of these improvements.

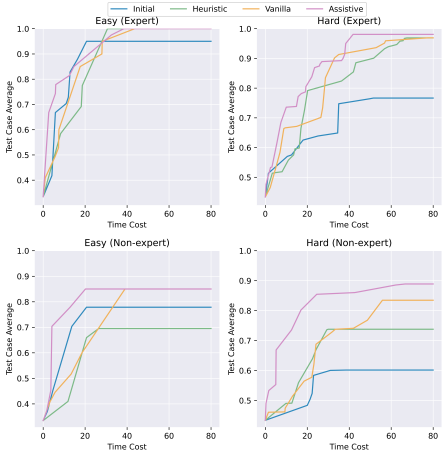

Figure 5: Decomposition brings more benefits to human labelers on hard problems.

This verifies the effectiveness of decomposition in aiding humans to tackle complex tasks by breaking them down into more manageable subtasks.

Decomposition enables non-experts to be comparable with experts Remarkably, when assisted with our decomposition model, non-experts achieve performance comparable to non-assisted experts in labeling efficiency and quality. This aligns with the overarching goal of scalable oversight, empowering human labelers to oversee models in superhuman tasks.

Decomposition benefits more on hard problems To further understand the impact of decomposition, we extract two subsets from our test data based on the time spent by non-experts to repair the model-generated programs: (1) Easy: problems that take less than 25 minutes. (2) Hard: problems that take more than 40 minutes. The results shown in Figure 5 reveal that easy problems benefit minimally from decomposition. However, when it comes to hard

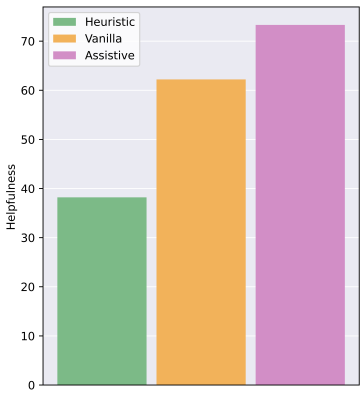

Figure 6: Helpfulness of decompositions evaluated by human labelers after repairing.

problems, decomposition significantly improves the labeling efficiency and final quality of human labelers. These results further underscore our decomposition's utility in complex tasks.

Humans prefer our decomposition model As shown in Figure 3 and Figure 4, our decomposition model surpasses other decomposition baselines in enhancing human labeling efficiency and quality. We also ask labelers to consider whether the decomposition is helpful for them to repair the program in the post-survey. As shown in Figure 6, labelers rate our decomposition results substantially more helpful than the baselines'. These results verify that LMs can learn to produce high-`AssistV` decompositions by learning from human repair experiences.

3.2.2 Qualitative Analysis

To further gain insight into what our decomposition model has learned from human feedback to enhance its utility, we analyze the collected human repairing traces alongside the corresponding critique, from which we draw the following qualitative observations.

Assisting humans by highlighting boundary conditions We observe that decomposing complex programs around boundary conditions effectively aids humans. As boundary conditions typically involve more straightforward logic than the core problem, humans can more easily understand and repair them. For instance, in one training demonstration (Figure 12), `check_validity` highlights all boundary conditions, earning positive human feedback for enabling them to quickly identify that problems are located in `calculate_minimum_moves`. This principle is learned by our decomposition model, as evidenced in Figure 13, where it highlights two boundary conditions in `check_zero_piles` and `check_same_piles`, leading humans to quickly realize the missing third boundary condition.

Assisting humans by creating simpler subtasks Decomposition reduces human workload by breaking down the initial solution into multiple simpler pieces, each corresponding to a specific subtask. For instance, in one training demonstration (Figure 14), `find_cycle` simplifies the complex `find_cycles`, and `generate_permutation` helps humans locate the actual bugs in a simple subtask. Similarly, in Figure 15, our decomposition model creates a simple task `calculate_participants` which effectively isolates bugs. In addition, our decomposition model learns to decompose code pieces that humans typically struggle with (e.g., nested loops, binary search, and dynamic programming), as exemplified in Figure 14 and Figure 16.

Assisting humans by presenting clear high-level logic Decomposition's ability to offer clear high-level logic can accelerate comprehension and bug identification before delving into low-level details, as demonstrated in Figure 16. Next, Figure 17 illustrates our decomposition model's integration of this principle by creating two subtasks `toggle_doors` and `toggle_single_door`. This high-level logic indicates that the current solution addresses each door independently, thereby enabling humans to locate bugs directly since doors are interrelated in the problem context.

3.3 Assisting AI Supervision

With the verified effectiveness of decomposition in assisting human supervision, we further explore its potential to assist AI supervision. We suspect that some patterns of decompositions shown in Section 3.2.2 (e.g., creating simpler subtasks and presenting clear high-level logic) can also help AI to evaluate (i.e., discriminate programs' correctness) and repair programs. We achieve AI supervision by prompting a code language model $\mathcal{M}$ to perform discrimination and repair (Saunders et al., 2022; Madaan et al., 2023; Chen et al., 2023).

Decomposition facilitates accurate self-supervision We first investigate decomposition's impact on self-supervision, where we use GPT-4 to supervise itself. Results in Table 1 reveal

Program	Discrimination Acc	Repair Acc (Strict)	Repair Acc (Test Case)
APPS			
Initial	10.2	18.3 (+0.0)	41.5 (-0.3)
Heuristic Decomp	30.0	16.7 (-1.6)	39.2 (-2.6)
Vanilla Decomp	32.7	19.2 (+0.9)	42.7 (+0.9)
Assistive Decomp	**42.9**	**21.7 (+2.9)**	**47.4 (+5.6)**
Code-Contests			
Initial	26.7	6.3 (+0.0)	20.1 (+1.3)
Heuristic Decomp	46.7	7.3 (+1.0)	19.3 (+0.5)
Vanilla Decomp	53.1	8.3 (+2.0)	21.4 (+2.6)
Assistive Decomp	**62.5**	**12.5 (+6.2)**	**26.3 (+7.5)**

Table 1: Decomposition aids AI systems to discriminate and repair programs, improving discrimination accuracy and repaired solutions' accuracy. Parenthetical values denote the accuracy variation from non-repair programs.

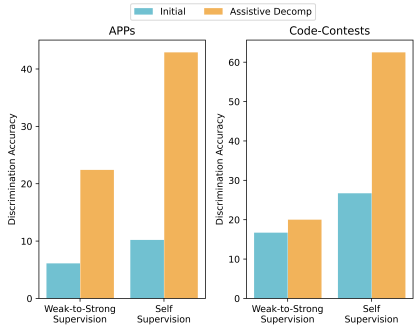

Figure 7: Decomposition aids AI systems to provide self-supervision, where GPT-4 discriminates its own outputs, and weak-to-strong supervision, where GPT-3.5-turbo discriminates GPT-4-generated programs.

that GPT-4 struggles to discriminate and repair programs generated by itself, echoing recent findings in LLMs' self-correction abilities (Huang et al., 2023a). However, decomposing the complex programs with our model improves both discrimination accuracy and repaired solution quality by a large margin. For instance, discrimination accuracy improves from 10.2% to 42.9% on APPS and from 26.7% to 62.5% on Code-Contests.

Decomposition facilitates accurate weak-to-strong supervision Having verified the effectiveness of decomposition in aiding non-expert human labelers, we now investigate its potential to assist weak models in supervising strong models. We evaluate GPT-3.5-turbo's ability to discriminate the programs generated by GPT-4. The results in Figure 7 reveal that our method leads to more accurate weak-to-strong AI supervision. For instance, it improves GPT-3.5-turbo's discrimination accuracy from 6.1% to 22.4% on APPS, surpassing the non-assisted accuracy of GPT-4 (10.2%).

Method	Accuracy
Intuitive Human Preference	49.5
Cyclomatic Complexity	48.4
GPT-3.5-turbo Zero-shot	54.3
GPT-3.5-turbo Few-shot	62.5
GPT-4 Zero-shot	66.7
GPT-4 Few-shot (π_{rank})	**78.1**

Table 2: Accuracy of the rank model in predicting the assistive value of decompositions in actual human repair processes.

4 Analysis

4.1 Validity of the Rank Model

We evaluate the effectiveness of our ranking model π_{rank} in predicting the assistive value of decompositions. We construct paired decompositions annotated with real assistive value labels as the test data. The results shown in Table 2 indicate that LLMs can learn to select higher-`AssistV` decomposition for assisting humans via in-context learning, surpassing prior heuristic or zero-shot baselines. Notably, we ask humans to provide preferences on paired decompositions without repairing them, which is shown to align poorly with their actual assistive value for human repair. These results highlight the importance of measuring assistive value based on actual human supervision experiences. See Appendix B.4 for more details about these ranking baselines.

4.2 Ablations on the Critic Model

We perform an ablation to explore the impact of our critic model π_{critic}. In the ablated version, we directly generate refined decompositions without incorporating predicted critiques. We use π_{rank} as a proxy to evaluate the assistive value of decompositions due to its reasonable performance (Section 4.1). The win rates between models with and without π_{critic} are 58.8% and 35.3%, respectively. These results indicate the effectiveness of learning from informative natural language human feedback.

4.3 Distilled Decomposition Model

In this section, we aim to study whether moderate-size open-source LLMs can also learn to generate better decompositions to assist human or AI supervision while reducing API costs and enhancing reproducibility. We hence distill the knowledge of assistive decomposition from proprietary LLMs into an in-house model π_θ. Specifically, we create

supervised data $D = \{(P, A, A^*_d)\}$ with LLMs as illustrated in Section 2.3, and optimizing π_θ over D via the standard MLE loss:

$$\mathcal{L}_{MLE} = -\sum_{t=1}^{|A^*_d|} \log \pi_\theta(A^*_{d_t} | A^*_{d_{<t}}, A, P)$$

where P, A, and A^*_d denote the problem, the initial model-generated solution, and the decomposed solution, respectively.

We evaluate the performance of the Code-LLaMA-based (Roziere et al., 2023) distilled decomposition model in aiding human and AI supervision, finding it outperforms the vanilla decomposition based on Code-LLaMA and even GPT-4. These results reveal the effectiveness of distilling the knowledge of assistive decomposition from GPT-4 to Code-LLaMA. See Appendix B.5 for detailed results.

5 Related Work

5.1 Code Generation

Language models have achieved impressive performance in generating simple code pieces (Chen et al., 2021; Austin et al., 2021) by learning from human demonstrations (Li et al., 2023), and are being used to generate increasingly complex programs (Hendrycks et al., 2021; Li et al., 2022). In this context, our paper studies how to aid humans in efficiently providing reliable supervision to train models that don't write buggy programs in complex and high-impact scenarios.

5.2 AI-assisted Programming

Prior works have explored various forms to assist human programmers, including code completion (Github, 2021), bug location (Xie et al., 2016), and program repair (Joshi et al., 2023). However, these methods rely on assistant models' capacity for accurate code generation or repair—a requirement unmet in our setup, where models face challenges in performing either task. We thus explore an alternative form of assistance via task decomposition, which alleviates the burden on assistant models.

5.3 Task Decomposition

Task decomposition has been extensively studied for tackling complex tasks, primarily focusing on enhancing model performance (Khot et al., 2022; Dua et al., 2022; Yao et al., 2022) and enabling efficient searching over the subtask space (Zelikman et al., 2023). In contrast, our paper studies decomposition from a human-centric perspective, aiming to facilitate human supervision over complex tasks, akin to Wu et al. (2021)'s work in aiding human supervision over book summarization. Concerning the implementation of decomposition, prior works mainly rely on fixed heuristic rules (e.g., decomposing each N-line of codes) or author-crafted demonstrations (Wu et al., 2021; Khot et al., 2022; Zelikman et al., 2023). In this paper, we take the first step towards advanced decomposition by learning to improve a novel objective: assistive value, which exactly measures the feasibility and speed of human repair in the actual annotation process.

5.4 Learning from Human Feedback

Learning from human feedback has been widely adopted in recent works for developing human-friendly general language models (Ouyang et al., 2022). Our work can also be understood as introducing this concept to the development of assistant models. Specifically, our introduced objective—assistive value–could also be introduced as a unique type of human feedback, which is gathered from actual human problem-solving experiences. Therefore, the collected human feedback could be more aligned with the ultimate goal of AI assistants: enhancing human problem-solving performance.

5.5 Scalable Oversight

Advancements in LLMs have intensified the need for scalable, reliable human oversight on complex tasks that reach or even exceed the capabilities of human experts. Addressing this, prior research has explored various assistance methods, such as self-critiquing (Saunders et al., 2022), AI debate (Parrish et al., 2022), and decomposition (Christiano et al., 2018; Wu et al., 2021). However, previous works overlook the feedback from actual assisted humans, potentially leading to reduced helpfulness of the assistance to humans (Xie et al., 2016; Parrish et al., 2022). To bridge this gap, we propose to learn assistance models from human feedback.

6 Conclusion

This paper focuses on assisting humans in supervising LMs on complex problems by automated task decomposition. We introduce a novel objective for learning task decomposition: assistive value (`AssistV`), which measures the feasibility

and speed of humans to repair a decomposed solution. We collect a dataset of decompositions, measure their AssistV, and ask human annotators to provide a natural language critique on what makes a decomposition (not) helpful. We then learn to critique, refine, and rank decompositions to generate high-AssistV decompositions. Experiment results demonstrate that our method can effectively assist humans in providing higher-quality supervision with significantly less time. Notably, our method assists non-experts in matching unassisted experts. Overall, we show that even when LMs cannot solve the problem themselves, they can still learn to assist humans by learning from human repair experiences. These results highlight the potential of learning-based methods to assist humans with scalable oversight.

Limitations and Future Work

One limitation of our work is that our non-expert participants, who could solve 30% of the problems while spending 74 minutes, are still more skilled than novice programmers (e.g., programmers who can only solve Leetcode easy-level problems) or non-programmers. Nonetheless, we demonstrate the effectiveness of our decomposition model, which successfully enables assisted non-experts to be comparable with non-assisted experts. We leave the investigation of leveraging decomposition to assist novice programmers or even non-programmers as future work.

Another limitation of our paper is that we did not conduct experiments on misleading benchmarks, which mainly consist of subtle errors that humans tend to overlook. The effectiveness of decomposition in aiding humans in detecting and fixing these subtle errors is worth studying in future work.

Moreover, we only conduct experiments based on in-context learning and supervised learning. Given the reasonable performance of the rank model (Section 4.1), fine-tuning the decomposition model with reinforcement learning is worth studying in the future.

Finally, our introduced objective AssistV, which highlights the possibility of learning an effective assistance model from human problem-solving experiences, can be further extended to other assistance forms beyond decomposition. For example, recent works explore using model-generated explanations to assist humans in complex decision-making tasks (e.g., complex question answering (Rein et al., 2023)). However, the practical benefits of model explanations are mixed as they may introduce additional human workload for understanding the explanation or even mislead humans to incorrect results (Parrish et al., 2022). Therefore, future works can explore what kind of explanations can better assist humans by measuring the AssistV of different explanations and learning to improve AssistV.

Ethics Statement

We recruit annotators mainly from college students. We inform annotators in advance how their annotation data will be collected and used. We pay them 30 USD per hour, which is higher than the average wage of the local residents.

Acknowledgements

This work was supported by the National Science Foundation for Distinguished Young Scholars (with No. 62125604) and the NSFC projects (with No. 62306160). This work was also supported by China National Postdoctoral Program for Innovative Talents (No. BX20230194) and China Postdoctoral Science Foundation (No. 2023M731952).

References

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*.

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

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Andrew P Bradley. 1997. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159.

Charis Charitsis, Chris Piech, and John C Mitchell. 2023. Detecting the reasons for program decomposition in cs1 and evaluating their impact. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 1014–1020.

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*.

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

Paul Christiano, Buck Shlegeris, and Dario Amodei. 2018. Supervising strong learners by amplifying weak experts. *arXiv preprint arXiv:1810.08575*.

Terry Connolly and Doug Dean. 1997. Decomposed versus holistic estimates of effort required for software writing tasks. *Management Science*, 43(7):1029–1045.

Carlos G Correa, Mark K Ho, Frederick Callaway, Nathaniel D Daw, and Thomas L Griffiths. 2023. Humans decompose tasks by trading off utility and computational cost. *PLOS Computational Biology*, 19(6):e1011087.

Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. 2022. Successive prompting for decomposing complex questions. *arXiv preprint arXiv:2212.04092*.

Github. 2021. Introducing github copilot.

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

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023a. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023b. Benchmarking large language models as ai research agents. *arXiv preprint arXiv:2310.03302*.

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*.

Frank J Lee and John R Anderson. 2001. Does learning a complex task have to be complex?: A study in learning decomposition. *Cognitive psychology*, 42(3):267–316.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

OpenAI. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Alicia Parrish, Harsh Trivedi, Nikita Nangia, Vishakh Padmakumar, Jason Phang, Amanpreet Singh Saimbhi, and Samuel R Bowman. 2022. Two-turn debate doesn't help humans answer hard reading comprehension questions. *arXiv preprint arXiv:2210.10860*.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2023. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators. *arXiv preprint arXiv:2206.05802*.

Cynthia C Selby. 2015. Relationships: computational thinking, pedagogy of programming, and bloom's taxonomy. In *Proceedings of the workshop in primary and secondary computing education*, pages 80–87.

Zhiqing Sun, Yikang Shen, Qinhong Zhou, Hongxin Zhang, Zhenfang Chen, David Cox, Yiming Yang, and Chuang Gan. 2023. Principle-driven self-alignment of language models from scratch with minimal human supervision. *arXiv preprint arXiv:2305.03047*.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7.

Jeff Wu, Long Ouyang, Daniel M Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. 2021. Recursively summarizing books with human feedback. *arXiv preprint arXiv:2109.10862*.

Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 808–819.

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

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic reasoning with language models by composing decompositions. In *Thirty-seventh Conference on Neural Information Processing Systems*.

A Implementation Details

A.1 Consistency of Decomposition

To ensure consistency between initial and decomposed solutions, we set the max retry time $N = 8$. If the consistency check fails in the end, we directly use the initial solution as the final output. Empirically, we find the consistency ratio of the decomposition generated by few-shot prompting with Code-LLaMA-13B, GPT-3.5-turbo, and GPT-4 is 92.9%, 97.6%, and 98.8%, respectively.

A.2 Data Collection

Data Preparation We use vanilla decomposition to sample $K = 5$ different decomposed solutions from various code generation models, including Code-LLaMA-13B, GPT-3.5-turbo, and GPT-4. We use top-p sampling with a default temperature 0.5.

Constructing Pair-wise Demonstrations For constructing pair-wise decompositions, since we adopt a few-shot learning setup in our experiments, we manually inspect the collected human critiques and perform matching. And we finally collect five pair-wise decompositions as the training data for in-context learning.

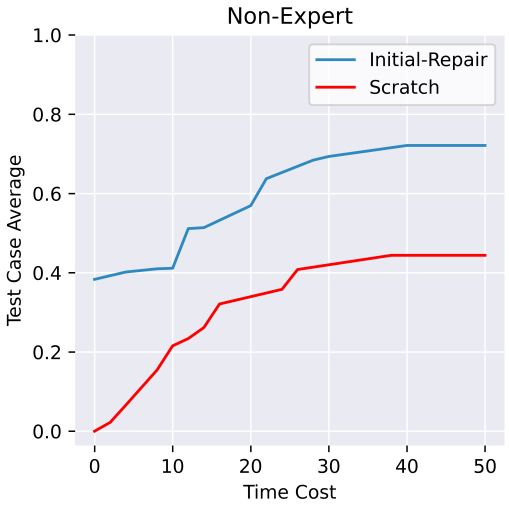

Figure 8: Comparing the performance of human labelers when asking them to write programs from scratch or repairing model-generated programs.

Considering that matching critiques based on semantic similarity is not challenging for modern LMs, we further explore automatically matching decompositions based on language models. Specifically, given two decomposed solutions with the corresponding human critiques: (A_d^1, C^1), (A_d^2, C^2), we prompt GPT-3.5-turbo to evaluate if C^1 matches C^2. We construct a test set that contains 50 decomposition pairs for evaluating the automatic matching performance, where the golden labels are derived from manual matching results. We find that GPT-3.5-turbo achieves a perfect accuracy of 100%. These results indicate that it is promising to construct pair-wise demonstrations automatically.

A.3 Benchmark

We obtain competition-level problems from Code-Contests and APPS. We use the whole test set of Code-Contests that consists of 96 problems and randomly sample 120 problems from the competition-level subset of APPs. When conducting human experiments, we first filter those problems where the generated solution directly passes all the hidden test cases and randomly sample 30 problems as the test data.

B Additional Results

B.1 Comparing Human Propose with Human Repair

We adopt a propose-and-repair pipeline to obtain labels for code generation. Instead of burdening humans with creating programs from scratch, we leverage the coding capabilities of modern LLMs to propose initial programs and then ask humans to repair them. This pipeline is motivated by the recognition that modern LLMs can effectively solve a considerable portion of test cases, making them a practical starting point for reducing the human workload and streamlining the data collection process (Vaithilingam et al., 2022).

We also empirically verify the effectiveness of this pipeline by comparing it with a baseline where human labelers are required to write programs from scratch. We construct two test sets for competitive programming, consisting of 10 and 18 problems that code generation models (e.g., GPT-4 in our experiments) succeed or fail to solve, respectively. We then study human labelers' efficiency on these two test sets. For problems that models can directly solve, non-expert humans also achieve 100% accuracy while still spending on average 24.8 minutes. For problems that models fail to solve, the results are shown in Figure 8. We can see that human labelers can also more easily collect a high-quality program label based on an initial model-generated program, yielding higher labeling quality and efficiency.

B.2 Directly Generating Decomposed Solutions

In our experiments, we follow a two-stage framework to first generate an initial solution and then decompose it. We choose not to generate decomposed solutions for these two reasons. First, the two-stage framework enables us to focus on the impact of decomposition. Otherwise, directly generating decomposed solutions may introduce differences in solution contents, which can also impact human repair experiences. Second, we observe that code generation models might perform worse when prompted to generate decomposed solutions directly. Table 3 presents the results, from which we can observe a substantial decrease in accuracy when directly prompting GPT-4 to generate decomposed solutions. We conjecture this might be due to the fact that most codes in these language models' pre-training corpus are not well modularized and decomposed.

Prompt	Strict Accuracy	Test Case Average
Non-decomposed	**18.3**	**41.5**
Decomposed	16.7	37.4

Table 3: Prompting GPT-4 to generate non-decomposed solutions and decomposed solutions. We evaluate the accuracy of the generated solutions on APPS.

B.3 Evaluating Code Decomposition with Software Engineering Metrics

We adopt the following four metrics to evaluate decomposition, which are widely adopted in software engineering:

- Func Number: It calculates the average number of functions (i.e., subtasks)
- Avg Complexity: It first calculates the average cyclomatic complexity among all pieces in a single program and then takes an average across all programs.
- Max Complexity: It first calculates the maximum cyclomatic complexity among all pieces in a single program and then takes an average across all programs.
- Global Max Complexity: It calculates the maximum cyclomatic complexity among all pieces across all programs.

From the results shown in Table 4, we can see that by learning from human feedback, our decomposition model produces more subtasks and lower complexity than the vanilla decomposition baseline. In addition, while heuristic decomposition results in the largest number of subtasks and the lowest cyclomatic complexity, it does not effectively assist humans in practice. These results indicate that these naive objectives for code decomposition (e.g., function number, cyclomatic complexity) are not well aligned with the assistive value in the actual human supervision process.

B.4 Analysis of the Rank Model

Given a question Q, and a pair of decomposed solutions A_d^1 and A_d^2, we adopt the following four methods to predict which decomposed solution leads to higher assistive value:

- **Intuitive Human Preference**: ask human programmers to give an intuitive preference for which decomposition can lead to higher human repair performance (i.e., higher assistive value). For each decomposition pair, we ask three anno-

Program	Func Number	Complexity		
		Avg	Max	Global Max
APPS				
Initial	0.4	5.5	5.7	17.0
Heuristic Decomp	4.5	2.0	2.3	4.0
Vanilla Decomp	2.8	2.5	4.0	12.0
Human-Centric Decomp	3.1	2.4	3.8	10.0
Code-Contests				
Initial	0.4	5.1	5.7	18.0
Heuristic Decomp	5.2	2.0	2.3	5.0
Vanilla Decomp	2.9	2.7	4.2	12.0
Human-Centric Decomp	4.3	2.2	3.7	10.0

Table 4: Statistics of initial programs and decomposed programs. "Func number" denotes the average number of functions (i.e., subtasks), "Complexity" denotes the value of cyclomatic complexity.

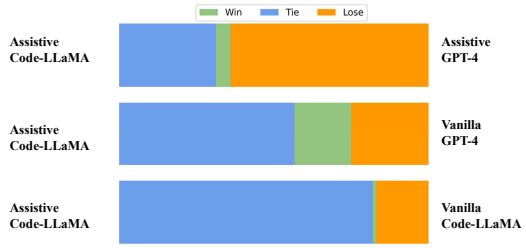

Figure 9: Comparison of the distilled assistive Code-LLaMA model against three other decomposition models. We use our rank model as a proxy evaluator.

tators to give a preference. We adopt majority voting to make final decisions among three annotators.

- **Cyclomatic Complexity**: consider the solution with a lower cyclomatic complexity as the better one.
- **Zero-shot**: prompt a code language model $\mathcal{M}$ to select a more effective decomposition in a zero-shot setting.
- **Few-shot (π_{rank})**: prompt a code language model $\mathcal{M}$ to select a more effective decomposition in a few-shot setting, where the few-shot demonstrations are collected from the feedback of real human labelers as illustrated in Section 2.

B.5 Analysis of the Distilled Decomposition Model

We compare the distilled assistive decomposition model based on Code-LLaMA-13B with the vanilla decomposition based on Code-LLaMA-13B and GPT-4, as well as its GPT-4-based teacher model. We conduct experiments to assist both human supervision and AI supervision.

Assisting Human Supervision Based on the reasonable accuracy of our rank model in predicting

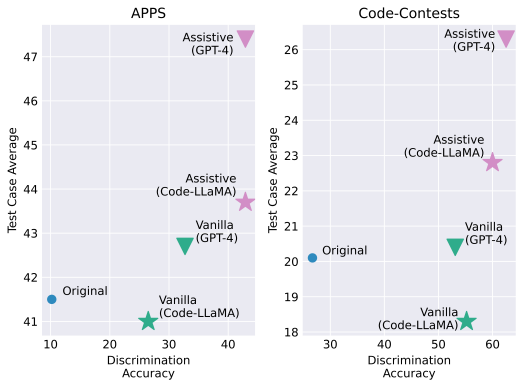

Figure 10: Assisting AI supervision with the distilled decomposition model.

the actual assistive value of decompositions (Section 4.1), we use the prediction of our rank model as a proxy to evaluate the distilled decomposition model. The results shown in Figure 9 demonstrate that the distilled assistive Code-LLaMA model substantially outperforms vanilla Code-LLaMA and moderately outperforms vanilla GPT-4, thanks to the internalized high-`AssistV` decomposition knowledge in the generated data from its teacher model.

Assisting AI Supervision As illustrated in Section 3.3, we evaluate the distilled decomposition model's performance in assisting AI systems in providing two forms of supervision: discrimination and repair. As shown in Figure 10, the distilled decomposition model leads to more accurate AI supervision than the vanilla Code-LLaMA-13B model and even the vanilla GPT-4.

C Additional Human Annotation Details

Labeler selection Our labelers are mainly hired from college students. To ensure the honesty of labelers, we track their debugging trajectories and filter those who plagiarize online golden solutions. As for group slicing, besides conducting a pre-survey to collect their self-evaluation on the programming level, we further examine their level based on their performance during the warm-up test.

Labeling instruction We present the summary of our labeling instructions in Table 5.

Interface In Figure 11, we present screenshots of our interface.

Instruction:
You are given an algorithmic coding problem and a model-generated solution. Your job is to debug the solution and improve its accuracy.

During debugging, you can actively submit your code and run your custom test cases. If the solution has a modular structure with multiple subfunctions, leverage it to accelerate your debugging. For example, you can check the presented high-level logic before inspecting low-level implementations, and you can perform function-level debugging.

There are two criteria to stop debugging: (1) The repaired solution passes all hidden test cases. (2) The debugging time already exceeds 30 minutes. This timeframe is imposed to avoid endless debugging.

Survey:
Review your debugging process and answer the following questions.

- Fixed Bugs: What bugs have you fixed during debugging?
- Critique on Decomposition: how does the current decomposition impede or improve your debugging efficiency?
- Other Assistance Forms: what other assistance forms do you need based on your debugging experience?

Table 5: Summary of our labeling instruction and survey.

D Example Prompts

D.1 Generating Initial Solution

For generating the Initial solution, we adopt the prompt in Table 6.

D.2 Vanilla Decomposition

For vanilla decomposition, we adopt the prompt in Table 7.

D.3 Critique

For generating critique on decomposition, we adopt the prompt in Table 8.

D.4 Refine

For generating the refined decomposed solution, we adopt the prompt in Table 9.

D.5 Rank

For ranking which decomposition leads to higher human efficiency, we adopt the prompt in Table 10.

E Case Study

We present several cases to illustrate how decomposition assists humans in practice by highlighting boundary conditions (Figure 12, Figure 13), creating simpler and more managable subtasks (Figure 14, Figure 15), and presenting clear high-level logic (Figure 16, Figure 17).

Competition-level Coding

Welcome，test_121！ exit

Annotation Procedure

- Step 1: Click on a question in the list, and the timer starts counting.
- Step 2: Perform debugging.
  - you can actively submit your code during debugging
- Step 3: End the annotation based on the following two criterias
  - The repaired solution passes all test cases
  - The debugging time already exceeds 30 minutes.
- Step 4: Review the debugging process and complete the post-survey

Evaluation

- Environment： `Python 3.8.10 [GCC 9.4.0]`； `Ubuntu 20.04`
- First use `py_compile` to complie the source code. Then use `python3` to execute the byte-code file.
- Apart from the built-in system libraries, no support is provided for any third-party libraries.
- time limit per test: 2s, memory limit per test: 512MB

Problem List

Idx	Status
test-0	Unlabeled
test-1	Unlabeled

;

(a)

;

(b)

Figure 11: Screenshots of our labeling interface. (a) After entering the OJ system, labelers can see a brief labeling instruction and a problem list. (b) For each problem, annotators are required to perform the debugging task. During debugging, our OJ system supports annotators to run their custom test cases.

Problem
You are given a tree consisting of n vertices. A number is written on each vertex; the number on vertex i is equal to a_i. Let's denote the function $g(x, y)$ as the greatest common divisor of the numbers written on the vertices belonging to the simple path from vertex x to vertex y (including these two vertices). Also let's denote $dist(x, y)$ as the number of vertices on the simple path between vertices x and y, including the endpoints. $dist(x, x) = 1$ for every vertex x.
Your task is to calculate the maximum value of $dist(x, y)$ among such pairs of vertices that $g(x, y) > 1$.
—–Input—–
The first line contains one integer n — the number of vertices ($1 \leq n \leq 2 \cdot 10^5$).
The second line contains n integers $a_1, a_2, ..., a_n$ ($1 \leq a_i \leq 2 \cdot 10^5$) — the numbers written on vertices.
Then $n - 1$ lines follow, each containing two integers x and y ($1 \leq x, y \leq n, x \neq y$) denoting an edge connecting vertex x with vertex y. It is guaranteed that these edges form a tree.
—–Output—–
If there is no pair of vertices x, y such that $g(x, y) > 1$, print 0. Otherwise print the maximum value of $dist(x, y)$ among such pairs...[TRUNCATED]
Solution

```
import sys

answer = 1
z = True
primes = []
for i in range (2, 5 * 10 ** 2):
    v = True
    for p in primes:
        if i \% p == 0:
            v = False
    if v == True:
        primes.append(i)
n = int(sys.stdin.readline().strip())
a = list(map(int, sys.stdin.readline().strip().split()))
if sum(a) == n:
    z = False
for i in range (0, n):
    x = a[i]
    a[i] = []
    for p in primes:
        if x \% p == 0:
            a[i].append([p, 1]) \\
            x = x // p
            while x \% p == 0:
                x = x // p
    if x != 1:
        a[i].append([x, 1])
neighbours = [[] for i in range (0, n)]
for i in range (0, n - 1):
    line = sys.stdin.readline().strip().split()
    neighbours[int(line[0]) - 1].append(int(line[1]) - 1)
    neighbours[int(line[1]) - 1].append(int(line[0]) - 1)
leaves = []
for i in range (0, n):
    if len(neighbours[i]) == 1:
        leaves.append(i)
while len(leaves) > 1:
    x = leaves.pop()
    y = neighbours[x][0]
    neighbours[y].remove(x)
    if len(neighbours[y]) == 1:
        leaves.append(y)
    for p in a[x]:
        for q in a[y]:
            if p[0] == q[0]:
                answer = max([answer, p[1] + q[1]])
                q[1] = max([q[1],p[1]+1])
if z == False:
    print(0)
else:
    print(answer)
```

Problem
{{Problem}}

Table 6: Excerpt of the input prompt for generating initial solutions.

Problem
You are given a tree consisting of n vertices. A number is written on each vertex; the number on vertex i is equal to a_i.
Let's denote the function $g(x, y)$ as the greatest common divisor of the numbers written on the vertices belonging to the simple path from vertex x to vertex y (including these two vertices). Also let's denote $dist(x, y)$ as the number of vertices on the simple path between vertices x and y, including the endpoints. $dist(x, x) = 1$ for every vertex x.
Your task is to calculate the maximum value of $dist(x, y)$ among such pairs of vertices that $g(x, y) > 1$...[TRUNCATED]
Initial Solution

```
# ...[TRUNCATED]
for i in range (2, 5 * 10 ** 2):
    v = True
    for p in primes:
        if i \% p == 0:
            v = False
    if v == True:
        primes.append(i)
n = int(sys.stdin.readline().strip())
a = list(map(int, sys.stdin.readline().strip().split()))
if sum(a) == n:
    z = False
for i in range (0, n):
    x = a[i]
    a[i] = []
    for p in primes:
        if x \% p == 0:
            a[i].append([p, 1])
            x = x // p
            while x \% p == 0:
                x = x // p
    if x != 1:
        a[i].append([x, 1])
# ...[TRUNCATED]
```

Your task is to decompose the initial solution into separate functions that handle specific tasks. This will make the code more modular and easier to understand and debug.
Decomposed Solution

```
def sieve_of_eratosthenes(limit):
    """
    Generate a list of prime numbers up to the specified limit.
    Parameters:
    - limit: The upper bound up to which prime numbers are to be found.
    Returns:
    - primes: A list of prime numbers up to the specified limit.
    """
    primes = []
    is_prime = [True] * (limit + 1)
    for p in range(2, limit + 1):
        if is_prime[p]:
            primes.append(p)
            for i in range(p * p, limit + 1, p):
                is_prime[i] = False
    return primes

def factorize_numbers(a, primes):
    """
    Factorize each number in the list 'a' using the list of known primes.
    Parameters:
    - a: The list of numbers to be factorized.
    - primes: The list of known prime numbers.
    Returns:
    - factorized: A list of factorized numbers, where each number is represented as a list of prime factors and their
      exponents.
    """
    factorized = []
    for x in a:
        factors = []
        for p in primes:
            if x \% p == 0:
                while x \% p == 0:
                    x //= p
                factors.append((p, 1))
        if x != 1:
            factors.append((x, 1))
        factorized.append(factors)
    return factorized

def find_leaves(neighbours):
    """
    Find the leaf nodes (vertices with only one adjacent vertex) of a tree.
    Parameters:
    - neighbours: The adjacency list representing the tree.
    Returns:
    - leaves: A list of leaf nodes in the tree.
    """
    leaves = []
    for i, neighbours_list in enumerate(neighbours):
        if len(neighbours_list) == 1:
            leaves.append(i)
    return leaves
# ...[TRUNCATED]
```

Problem
{{Problem}}
Initial Solution
{{Initial Solution}}

Table 7: Excerpt of the input prompt for generating vanilla decomposition.

Problem
Two pirates Polycarpus and Vasily play a very interesting game. They have n chests with coins, the chests are numbered with integers from 1 to n. Chest number i has a_i coins.
Polycarpus and Vasily move in turns. Polycarpus moves first. During a move a player is allowed to choose a positive integer $x(2x + 1 \leq n)$ and take a coin from each chest with numbers x, $2x$, $2x + 1$. It may turn out that some chest has no coins, in this case the player doesn't take a coin from this chest. The game finishes when all chests get emptied...[TRUNCATED]
Initial Solution

```
def is_valid_input(n, a):
    """
    Check if the input is valid for the given constraints.

    Parameters:
    - n: The number of chests with coins.
    - a: The list of number of coins in each chest.

    Returns:
    - True if the input is valid, False otherwise.
    """
    return n % 2 == 1

def initialize_chests(a):
    """
    Initialize the list of chests with coins, adding extra elements at the beginning and end.

    Parameters:
    - a: The list of number of coins in each chest.

    Returns:
    - The modified list of chests.
    """
    return [0] + a + [0, 0]

def make_move(a, i):
    """
    Perform a move in the game, updating the number of coins in the chests accordingly.

    Parameters:
    - a: The list of number of coins in each chest.
    - i: The index of the chest to perform the move on.
    """
    m = min(a[i], a[i - 1])
    a[i // 2] = max(0, a[i // 2] - m)
    a[i] -= m
    a[i - 1] -= m

def calculate_minimum_moves(n, a):
    """
    Calculate the minimum number of moves needed to finish the game.

    Parameters:
    - n: The number of chests with coins.
    - a: The list of number of coins in each chest.

    Returns:
    - The minimum number of moves needed to finish the game.
    """
    if n == 1 or not is_valid_input(n, a):
        return -1
    a = initialize_chests(a)
    for i in range(n, 0, -2):
        make_move(a, i)
    return sum(a)
# ... [TRUNCATED]
```

Critique Request
Identify specific ways in which the decomposition strategy of the initial solution is less effective in assisting humans to debug the solution.
Critique
1. Decomposition should be around logical breakpoints, but not syntax. However, `make_move` and `initialize_chests` are not the main logic steps of this problem. We should present a clear high-level logic that aligns with how humans reason about programs conceptually. In this problem, the main logic steps are input reading, checking game finishing conditions, calculating minimum steps, and output printing.
2. Boundary conditions (i.e., `if n == 1 and n % 2 == 1` should be explicitly highlighted as individual functions in the main logic, instead of in the function `calculate_minimum_moves`.
(...TRUNCATED)
Problem
{{Problem}}
Initial Solution
{{Initial Solution}}

Table 8: Excerpt of the input prompt for generating critique on decomposition

Problem
Two pirates Polycarpus and Vasily play a very interesting game. They have n chests with coins, the chests are numbered with integers from 1 to n. Chest number i has a_i coins.
Polycarpus and Vasily move in turns. Polycarpus moves first. During a move a player is allowed to choose a positive integer $x(2x + 1 \leq n)$ and take a coin from each chest with numbers x, $2x$, $2x + 1$. It may turn out that some chest has no coins, in this case the player doesn't take a coin from this chest. The game finishes when all chests get emptied...[TRUNCATED]
Initial Solution

```
def is_valid_input(n, a):
    """Check if the input is valid for the given constraints...(TRUNCATED)"""
    return n % 2 == 1

def initialize_chests(a):
    """Initialize the list of chests with coins, adding extra elements at the beginning and end...(TRUNCATED)"""
    return [0] + a + [0, 0]

def make_move(a, i):
    """Perform a move in the game, updating the number of coins in the chests accordingly...(TRUNCATED)"""
    m = min(a[i], a[i - 1])
    a[i // 2] = max(0, a[i // 2] - m)
    a[i] -= m
    a[i - 1] -= m

def calculate_minimum_moves(n, a):
    """Calculate the minimum number of moves needed to finish the game...(TRUNCATED)"""
    if n == 1 or not is_valid_input(n, a):
        return -1
    a = initialize_chests(a)
    for i in range(n, 0, -2):
        make_move(a, i)
    return sum(a)

def main():
    n = int(input())
    a = list(map(int, input().split()))
    result = calculate_minimum_moves(n, a)
    print(result)
# ...[TRUNCATED]
```

Critique
1. Decomposition should be around logical breakpoints, but not syntax. However, `make_move` and `initialize_chests` are not the main logic steps of this problem. We should present a clear high-level logic that aligns with how humans reason about programs conceptually. In this problem, the main logic steps are input reading, checking game finishing conditions, calculating minimum steps, and output printing.
2. Boundary conditions (i.e., `if n == 1 and n % 2 == 1` should be explicitly highlighted as individual functions in the main logic, instead of in the function `calculate_minimum_moves`.

Refine the initial solution based on the above critique. Do not introduce modification unrelated to the critique.
Refine

```
...(TRUNCATED)
def check_validity(n):
    """Check if the game can be finished...(TRUNCATED)"""
    return n != 1 and n % 2 != 0

def calculate_minimum_moves(n, a):
    """Calculate the minimum number of moves needed to finish the game...(TRUNCATED)"""
    a = [0] + a + [0, 0]
    for i in range(n, 0, -2):
        m = min(a[i], a[i - 1])
        a[i // 2] = max(0, a[i // 2] - m)
        a[i] -= m
        a[i - 1] -= m
    return sum(a)

def main():
    n, a = read_input()
    if not check_validity(n):
        print(-1)
    else:
        print(calculate_minimum_moves(n, a))
# ...[TRUNCATED]
```

(...TRUNCATED)
Problem
{{Problem}}
Initial Solution
{{Initial Solution}}
Critique
{{Critique}}

Table 9: Excerpt of the input prompt for generating refined decomposition

Problem
Two pirates Polycarpus and Vasily play a very interesting game. They have n chests with coins, the chests are numbered with integers from 1 to n. Chest number i has a_i coins.
Polycarpus and Vasily move in turns. Polycarpus moves first. During a move a player is allowed to choose a positive integer $x(2x + 1 \leq n)$ and take a coin from each chest with numbers x, $2x$, $2x + 1$. It may turn out that some chest has no coins, in this case the player doesn't take a coin from this chest. The game finishes when all chests get emptied...[TRUNCATED]
Decomposed Solution A

```
def is_valid_input(n, a):
    """Check if the input is valid for the given constraints...(TRUNCATED)"""
    return n % 2 == 1

def initialize_chests(a):
    """Initialize the list of chests with coins, adding extra elements at the beginning and end...(TRUNCATED)"""
    return [0] + a + [0, 0]

def make_move(a, i):
    """Perform a move in the game, updating the number of coins in the chests accordingly...(TRUNCATED)"""
    m = min(a[i], a[i - 1])
    a[i // 2] = max(0, a[i // 2] - m)
    a[i] -= m
    a[i - 1] -= m

def calculate_minimum_moves(n, a):
    """Calculate the minimum number of moves needed to finish the game...(TRUNCATED)"""
    if n == 1 or not is_valid_input(n, a):
        return -1
    a = initialize_chests(a)
    for i in range(n, 0, -2):
        make_move(a, i)
    return sum(a)

def main():
    n = int(input())
    a = list(map(int, input().split()))
    result = calculate_minimum_moves(n, a)
    print(result)
# ...[TRUNCATED]
```

Decomposed Solution B

```
...(TRUNCATED)
def check_validity(n):
    """Check if the game can be finished...(TRUNCATED)"""
    return n != 1 and n % 2 != 0

def calculate_minimum_moves(n, a):
    """Calculate the minimum number of moves needed to finish the game...(TRUNCATED)"""
    a = [0] + a + [0, 0]
    for i in range(n, 0, -2):
        m = min(a[i], a[i - 1])
        a[i // 2] = max(0, a[i // 2] - m)
        a[i] -= m
        a[i - 1] -= m
    return sum(a)

def main():
    n, a = read_input()
    if not check_validity(n):
        print(-1)
    else:
        print(calculate_minimum_moves(n, a))
# ...[TRUNCATED]
```

Evaluation Request Given two decomposed solutions A and B, determine which one of them is more effective in assisting humans to debug the solution.
Evaluation Results Decomposed Solution B is better.
1. Decomposition should be around logical breakpoints, but not syntax. However, `make_move` and `initialize_chests` in Decomposed Solution A are not the main logic steps of this problem. We should present a clear high-level logic that aligns with how humans reason about programs conceptually. In this problem, the main logic steps are input reading, checking game finishing conditions, calculating minimum steps, and output printing, as shown in Decomposed Solution B.
2. Boundary conditions (i.e., `if n == 1 and n % 2 == 1`) should be explicitly highlighted as individual functions in the main logic, while Decomposed Solution A places it in the `calculate_minimum_moves` function.
(...TRUNCATED)
Problem
{{Problem}}
Decomposed Solution A
{{Decomposed Solution A}}
Decomposed Solution B
{{Decomposed Solution B}}

Table 10: Excerpt of the input prompt for ranking different decompositions.

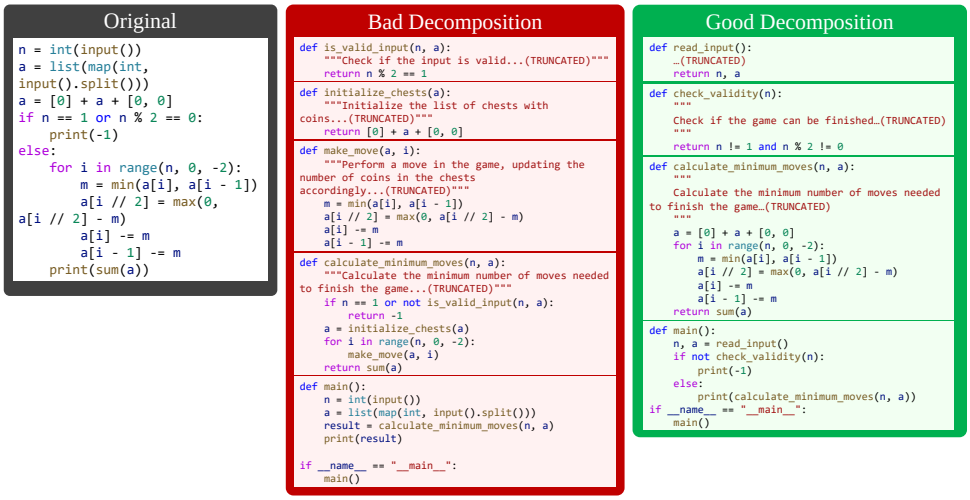

Figure 12: Excerpt of the decomposition that highlights boundary conditions in the training data.

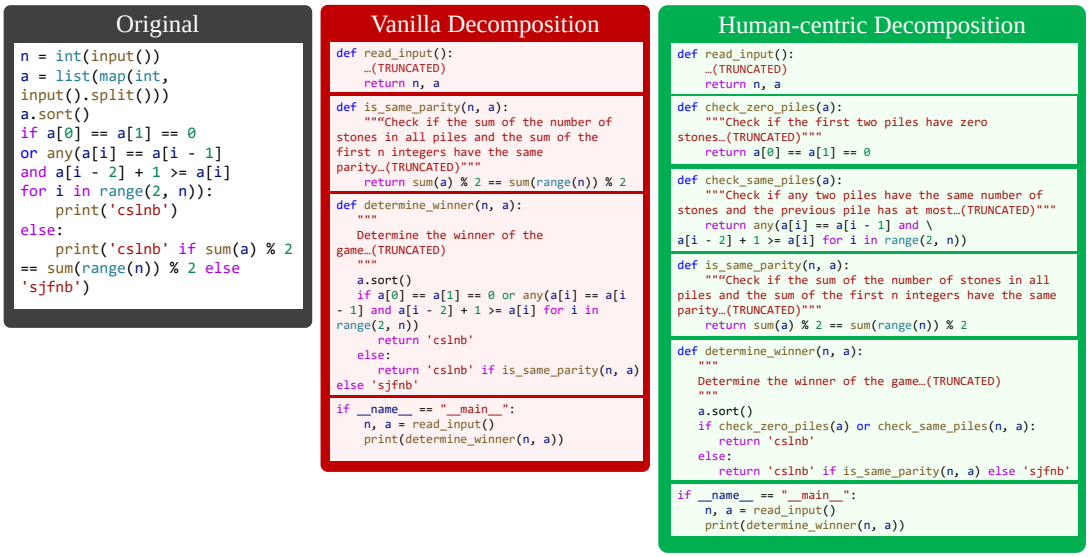

Figure 13: Excerpt of the decomposition that highlights boundary conditions in the test data.

Original

```
n = int(input())
a = list(map(int,
input().split()))
b = sorted([(a[i], i + 1) for i
in range(n)])
p = [0] * (n + 1)
for i in range(n):
    p[b[i][1]] = b[(i + 1) %
n][1]
used = [0] * (n + 1)
cycles = []
for i in range(1, n + 1):
    if not used[i]:
        cycle = [i]
        v = p[i]
        while v != i:
            cycle.append(v)
            v = p[v]
        cycles.append(cycle)
        for v in cycle:
            used[v] = 1
print(len(cycles))
for cycle in cycles:
    print(len(cycle), *cycle)
```

Bad Decomposition

```
def read_input():
    …(TRUNCATED)
    return n, a
```

```
def sort_sequence(n, a):
    """Sort the sequence and…(TRUNCATED)…"""
    b = sorted([(a[i], i + 1) for i in
range(n)])
    return b
```

```
def find_cycles(b, p):
    """Find the cycles of the elements in
the sorted sequence…"""
    used = [0] * (len(a) + 1)
    cycles = []
    for i in range(1, len(a) + 1):
        if not used[i]:
            cycle = [i]
            v = p[i]
            while v != i:
                cycle.append(v)
                v = p[v]
            cycles.append(cycle)
            for v in cycle:
                used[v] = 1
    return cycles
```

```
if __name__ == "__main__":
    # Read input values from standard input
    n = int(input())
    a = list(map(int, input().split()))
    # Initialize variables
    b = sort_sequence(n, a)
    p = [0] * (len(a) + 1)
    for i in range(len(a)):
        p[b[i][1]] = b[(i + 1) % len(a)][1]
    # Find the indices of the elements in
the sorted sequence
    cycles = find_cycles(b, p)
    # Print the output
    print(len(cycles))
    for cycle in cycles:
        print(len(cycle), *cycle)
```

Good Decomposition

```
def read_input():
    …(TRUNCATED)
    return n, a
```

```
def sort_sequence(n, a):
    """Sort the sequence and…(TRUNCATED)"""
    b = sorted([(a[i], i + 1) for i in range(n)])
    return b
```

```
def generate_permutation(n, b):
    """Generate a permutation of the sequence…(TRUNCATED)"""
    p = [0] * (n + 1)
    for i in range(n):
        p[b[i][1]] = b[(i + 1) % n][1]
    return p
```

```
def find_cycles(n, p):
    """Find the cycles in the permutation…(TRUNCATED)"""
    used = [0] * (n + 1)
    cycles = []
    for i in range(1, n + 1):
        if not used[i]:
            cycle = find_cycle(i, p, used)
            cycles.append(cycle)
    return cycles
```

```
def find_cycle(i, p, used):
    """Find a cycle in the permutation starting from
i…(TRUNCATED)"""
    cycle = [i]
    v = p[i]
    while v != i:
        cycle.append(v)
        v = p[v]
    for v in cycle:
        used[v] = 1
    return cycle
```

```
def print_output(cycles):
    …(TRUNCATED)
```

```
if __name__ == "__main__":
    n, a = read_input()
    p = generate_permutation(n, a)
    cycles = find_cycles(n, p)
    print_output(cycles)
```

Figure 14: Excerpt of the decomposition that creates simpler subtasks in the training data.

Original Solution

```
import math

Q = int(input())
for _ in range(Q):
    A, B = map(int,
input().split())
    if A > B:
        A, B = B, A
    if A * B == 1:
        print(0)
    else:
        x = int(math.sqrt(A * B))
        if x * (x + 1) < A * B:
            print(2 * x - 1)
        elif x * x < A * B:
            print(2 * x - 2)
        else:
            print(2 * x - 3)
```

Vanilla Decomposition

```
def calculate_max_participants(A, B):
    """Calculate the maximum possible number of
participants whose…(TRUNCATED)"""
    if A > B:
        A, B = B, A
    if A * B == 1:
        return 0
    else:
        x = int(math.sqrt(A * B))
        if x * (x + 1) < A * B:
            return 2 * x - 1
        elif x * x < A * B:
            return 2 * x - 2
        else:
            return 2 * x - 3
```

```
def process_queries(Q, queries):
    """
    Process each query and print the maximum
possible number of participants…(TRUNCATED)"""
    for i in range(Q):
        A, B = queries[i]
        max_participants =
calculate_max_participants(A, B)
        print(max_participants)
```

```
if __name__ == "__main__":
    Q = int(input())
    queries = [tuple(map(int, input().split()))
for _ in range(Q)]
    process_queries(Q, queries)
```

Human-centric Decomposition

```
def calculate_score(A, B):
    """Calculate the score based on Takahashi's ranks in the
two contests…(TRUNCATED)"""
    return A * B
```

```
def calculate_participants(score):
    """
    Calculate the maximum possible number…(TRUNCATED)"""
    x = int(math.sqrt(score))
    if x * (x + 1) < score:
        return 2 * x - 1
    elif x * x < score:
        return 2 * x - 2
    else:
        return 2 * x - 3
```

```
def calculate_max_participants(A, B):
    """Calculate the maximum possible number…(TRUNCATED)"""
    if A > B:
        A, B = B, A
    score = calculate_score(A, B)
    if score == 1:
        return 0
    else:
        return calculate_participants(score)
```

```
def process_queries(Q, queries):
    """
    Process each query and print the maximum possible number
of participants…(TRUNCATED)"""
    for i in range(Q):
        A, B = queries[i]
        max_participants = calculate_max_participants(A, B)
        print(max_participants)
```

```
if __name__ == "__main__":
    Q = int(input())
    queries = [tuple(map(int, input().split())) for _ in
range(Q)]
    process_queries(Q, queries)
```

Figure 15: Excerpt of the decomposition that creates simpler subtasks in the test data

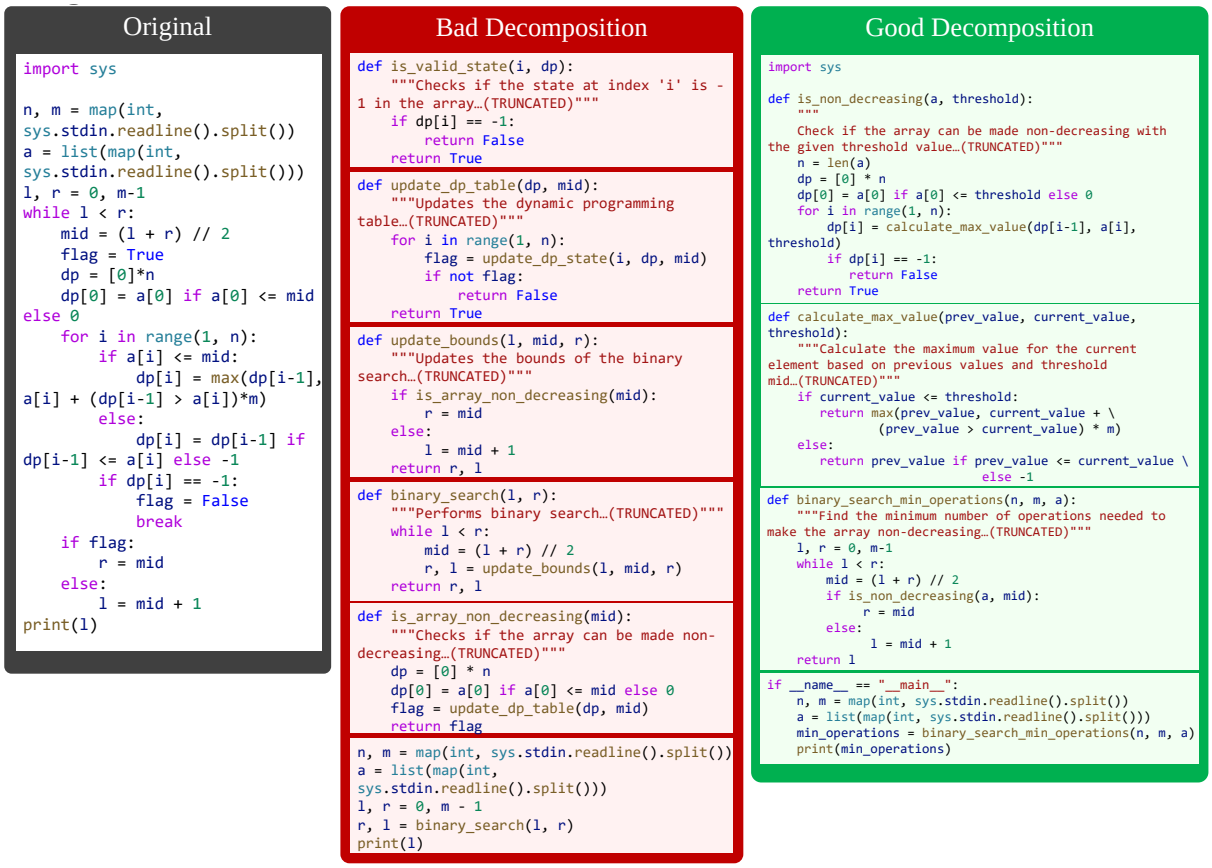

Figure 16: Excerpt of the decomposition that presents clear high-level logic in the training data.

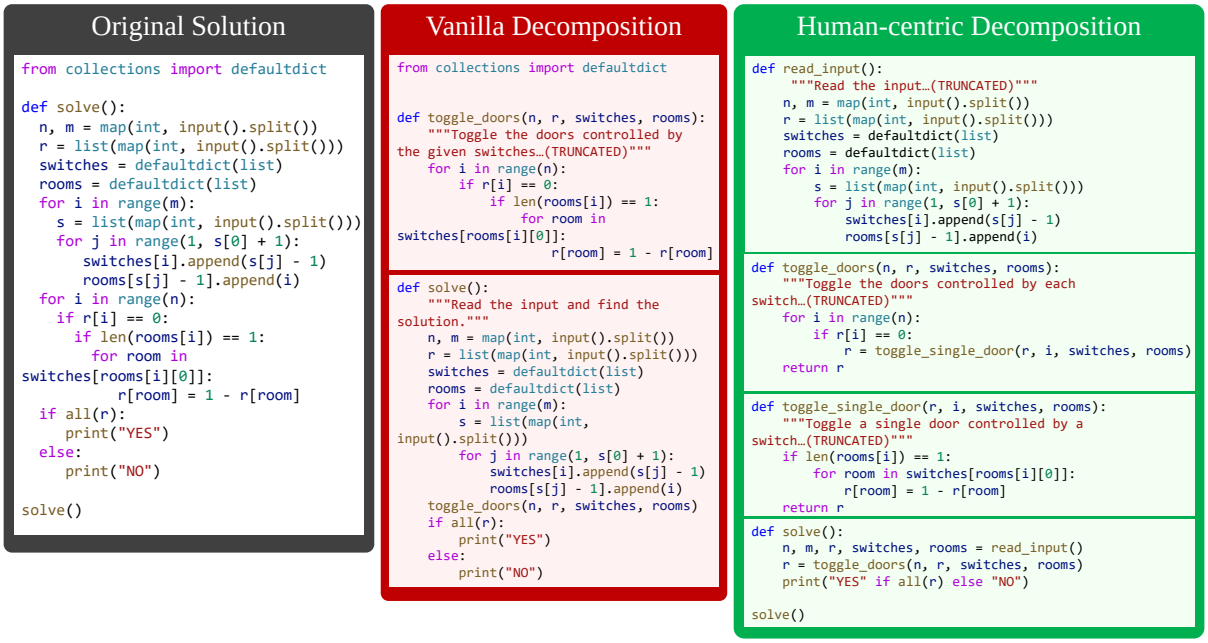

Figure 17: Excerpt of the decomposition that presents clear high-level logic in the test data.