Improving Code Style for Accurate Code Generation

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph Gonzalez, Koushik Sen, Ion Stoica
University of California, Berkeley
{naman_jain,tianjunz,weichiang,jegonzal,ksen,istoica}@berkeley.edu

Abstract

Natural language to code generation is an important application area of LLMs and has received wide attention from the community. The majority of relevant studies have exclusively concentrated on increasing the quantity and *correctness* of the training sets while disregarding other stylistic elements of programs. More recently, data quality has garnered a lot of interest and multiple works have showcased its importance for improving performance. In this work, we investigate data quality for code in terms of lexical properties, control-flow properties, and program explanations. We find that making the code more structured and readable leads to improved code generation performance of the system. We build a novel data-cleaning pipeline that uses these principles to transform existing programs by 1.) renaming variables, 2.) modularizing and decomposing complex code into smaller helper sub-functions, and 3.) inserting natural-language based planning annotations. We evaluate our approach on two challenging algorithmic code generation benchmarks and find that fine-tuning CODELLAMA-7B on our transformed programs improves the performance by up to **30%** compared to fine-tuning on the original dataset. Additionally, we demonstrate improved performance from using a smaller amount of higher-quality data, finding that a model fine-tuned on the entire original dataset is outperformed by a model trained on one-eighth of our cleaned dataset.

1 Introduction

Natural language to code generation has witnessed considerable advances in recent years with the advent of large language models (LLMs for brevity). These advances primarily arise from training on extensive web-scale data and are measured based on the functional correctness of the programs. Thus, other aspects like readability, structuring, and styling and how they affect training and data quality are primarily ignored by these works. On the flip side, many recent works have demonstrated the effectiveness of training on higher quality data during both pre-training Li et al. (2023d) and fine-tuning Zhou et al. (2023); Cao et al. (2023) phases. Even within the code-generation domain, Gunasekar et al. (2023) proposed training on "textbook" quality dataset, generated synthetically using the GPT-3.5-TURBO model Ouyang et al. (2022). However, these techniques do not provide an understanding of the factors improving data quality.

In this work, we show that using programs following good programming practices and allowing for more readability leads to improved code generation performance compared to using programs that do not follow these practices. We use these insights to build a novel automated code data-cleaning pipeline that transforms programs while maintaining functional correctness using input-output examples. In contrast to prior works that curate *high quality* datasets by directly generating *new* data using LLMs, here we translate existing datasets into their *parallel cleaned* versions while identifying attributes that improve data quality.

We use LLMs to perform the transformations in our data-cleaning approach. We demonstrate that instruction-tuned models can take a user-identified attribute of data quality, such as using meaning-

NeurIPS 2023 Workshop on Synthetic Data Generation with Generative AI.

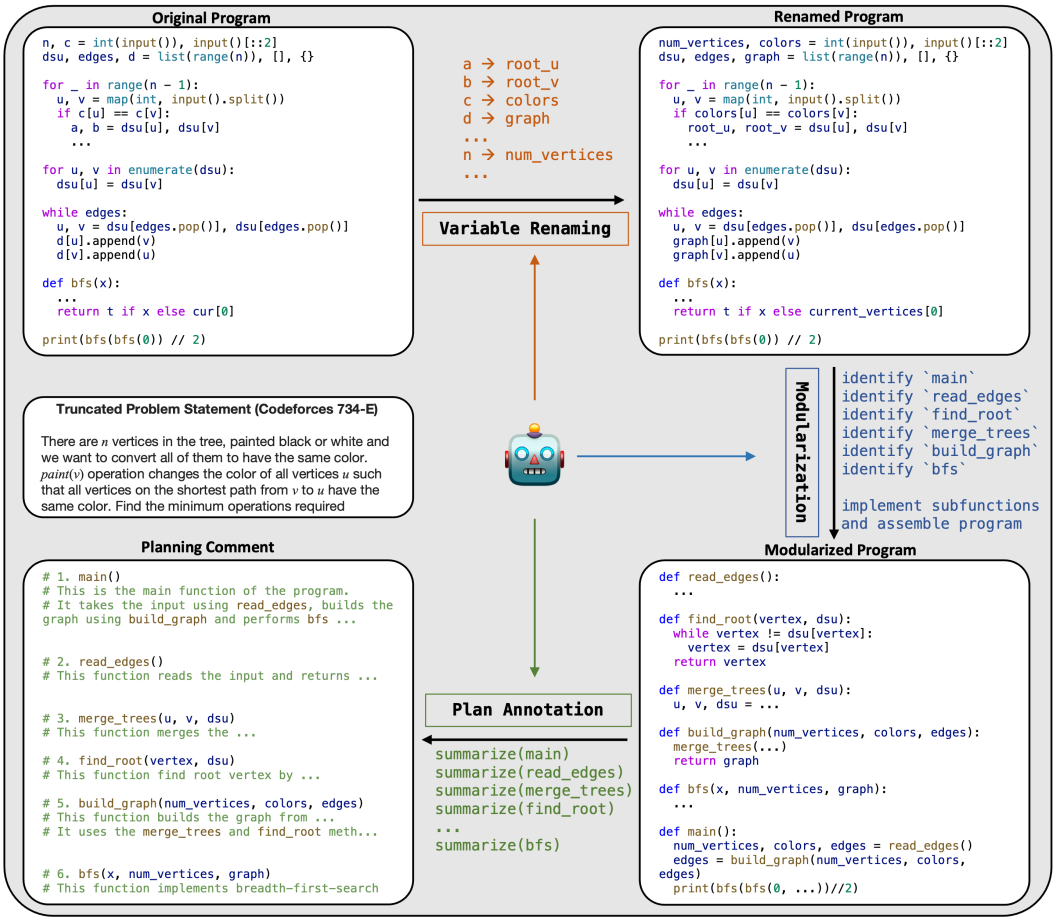

Figure 1: The overview of our code cleaning approach. We use instruction-tuned LLMs to transform existing datasets by providing natural language prompts and use input-output examples to maintain function equivalence. Our cleaning approach works in three steps. The top-left figure depicts the original program from our dataset. This program undergoes variable renaming (top-right figure). Next, the renamed program is decomposed into constituent sub-functions (bottom-right figure). Finally, we generate a *plan* from the modularized program by summarizing the functions called in the (bottom-left figure). The middle-left figure presents the truncated problem statement

ful variable names, as a natural language instruction and perform the transformation accurately. Our approach leverages the disparity in difficulty between generating a solution and editing an existing one. Therefore, it is particularly effective in domains where the existing model struggles to generate a correct solution but can effectively edit a given solution. We perform our data-cleaning transformations in three iterations: 1) renaming variables, 2) modularizing complex code into subfunctions, and 3) adding planning annotations.

Figure 1 provides an overview of our approach. Notice that the variable renaming step at the top adjusts the variable names to be contextually relevant (e.g., `a` to `root_u` and `d` to `graph`). The modularization step (depicted on the right) identifies and decomposes the original program into several smaller subfunctions such as `find_root`, `merge_trees`, `build_graph`, etc. It then implements these subroutines and assembles the modular program. Finally, our planning step (depicted at the bottom) constructs a plan by summarizing functions in a top-down fashion (starting from the `main`).

We evaluate our approach in a niche yet challenging domain of algorithmic code generation. The goal is to generate a program for a given problem statement. The task is challenging because it requires high-level algorithmic reasoning and low-level coding and is evaluated using a strict functional correctness metric. We use two well-known algorithmic code generation benchmarks, namely APPS Hendrycks et al. (2021) and CODE-CONTESTS Li et al. (2022). We transform the corresponding programs in the training sets and obtain *parallel* datasets from our cleaning approach. Additionally, we utilize input-output examples to maintain functional equivalence between the original and

transformed programs. We qualitatively analyze the generated dataset and find that it uses smaller helper sub-functions, each often implementing a standard algorithm or key program functionality. We analyze the generated dataset in more depth in Section 4.1. We further assess the impact of the transformed datasets on the performance of our downstream code generation task. We fine-tune the CODELLAMA-7B model on the collected datasets. Our findings reveal that the model fine-tuned on our modularized dataset outperforms the model fine-tuned on the functionally equivalent original dataset by up to **30%**. Beyond performance improvement, we also demonstrate that improving data quality improves data efficiency. In particular, a model fine-tuned on the original dataset outperforms a model trained on just one-eighth of our cleaned dataset.

2 Methodology

In this section, we present our general data transformation approach and then instantiate it for performing code data cleaning.

2.1 Transformations for data cleaning

Given a dataset $\mathcal{D}$ consisting of N instances $\mathbf{d}_i$, such that, $\mathcal{D} = \{\mathbf{d}_i\}_{i=1}^{N}$. To achieve a desired data cleaning specification, the user additionally provides a data-cleaning instruction $\mathcal{I}$, which highlights an attribute that needs to be modified. Optionally, we also use an oracle equivalence checker ($\mathcal{O}$) which ensures that the transformed data instance $\tilde{\mathbf{d}}_i$ is consistent with the original input based on some desired metric. For example, we can use edit-distance or functional equivalence based on input-output examples as our oracle checker.

We use a pre-trained language model (denoted by $\mathcal{M}$) to generate the transformed instance ($\tilde{\mathbf{d}}_i$) by prompting the model with the transformation instruction ($\mathcal{I}$) and the original answer ($\mathbf{y}$). We perform zero-shot prompting for performing the data cleaning operations. Finally, we extract the instance $\tilde{\mathbf{d}}_i$ generated by $\mathcal{M}$, and apply our oracle equivalence checker ($\mathcal{O}$) to ensure consistency with the original data. If $\mathcal{O}(\tilde{\mathbf{d}}_i, \mathbf{d}_i) = 0$, i.e., the oracle reports a failure, we reject the generated output and retry the example within a sampling budget

While our transformation approach does not provide any guarantees about the quality of the performed transformation and relies on LLMs, we empirically observe that instruction-tuned LLMs can perform various unstructured data cleaning steps quite effectively. We provide a detailed analysis of the generated outputs for our algorithmic code generation setting in Section 4.1. Finally, in accordance with existing literature on prompting LLMs, we found that using simple and precise, low-level instructions improves the performance and accuracy of the models in performing the operations. Thus, for complex data cleaning operations, we find improvements by breaking it down and performing multiple operations iteratively.

2.2 Code Data-Cleaning

We apply our transformations-based data cleaning approach to programming data. Coding requires both – low-level programming and high-level reasoning or planning skills. Therefore, we propose a three-step cleaning pipeline that improves the readability and program structuring targeting the low-level coding skills and inserts natural-language-based plans data targeting the high-level reasoning skills. Our steps are detailed below.

1. **Rename variables.** This step renames the variables in the program, making them descriptive and easier to follow. Figure 1 top provides an example of this transformation.
2. **Modularize functions.** Problem decomposition has been identified as a key approach for improving the reasoning capabilities of models Zhou et al. (2022); Wang et al. (2023). We identify program decompositions and transform the program by extracting their functionality into smaller helper functions. Figure 1 right provides an example of this transformation.
3. **Plan annotations.** This step summarizes the helper functions in the already modularized program and prepends it to the programs in the form of a natural language plan. These natural language descriptions are analogous to prompting approaches that are used for solving reasoning problems like chain-of-thought prompting Wei et al. (2022), parsel Zelikman et al. (2023), etc. Figure 1 bottom provides an example of this transformation.

Dataset	Notation	Applied On	Transformation Instruction ($\mathcal{I}$)
Base	$\mathcal{D}_{original}$	-	-
Rename	$\mathcal{D}_{rename}$	$\mathcal{D}_{original}$	*Rename the variables in the program to be descriptive, meaningful, and consistent*
Modularize	$\mathcal{D}_{modular}$	$\mathcal{D}_{rename}$	*Refactor the above program making it more modular with smaller and meaningful helper functions with good descriptive names for the helper functions*
Plan	$\mathcal{D}_{planning}$	$\mathcal{D}_{modular}$	*Generate a natural language description for the following functions in the program*

Table 1: Transformed datasets generated by our code cleaning approach. For each transformation, we have provided the corresponding notation, the transformation instruction used to perform the cleaning step and the dataset the transformation was applied on.

Additionally, while performing these transformations, we use the test cases provided in the dataset to construct our oracle equivalence checker ($\mathcal{O}$). It ensures that our transformed programs maintain functional equivalence to the original program.

3 Experimental Setup

We next detail our experimental setup and implementation. Section 3.1 outlines the benchmarks and metrics used for the algorithmic code generation task, while Sections 3.2 and 3.3 delve into the specifics of our code cleaning approach and fine-tuning experiments respectively.

3.1 Benchmarks

We use two standard algorithmic code generation benchmarks, APPS and CODE-CONTESTS. The benchmarks provide a collection of problem statements described in natural language and corresponding test cases. The goal is to generate a program that successfully solves the problem. The evaluation is performed using a functional-correctness-based metric.

APPS Hendrycks et al. (2021). This benchmark includes 10,000 problems, evenly split between training and test sets. It is sourced from multiple open-access competitive programming websites. It is further divided into APPS-INTRODUCTORY, APPS-INTERVIEW, and APPS-COMPETITION subsets based on problem difficulty. In this study, we only consider problems sourced from a subset of the competition websites based on the number of test cases provided.

CODE-CONTESTS Li et al. (2022). This benchmark includes 13,328 problems in the training set and 165 problems in the test set. We only use a subset of the training split that includes `python` solutions satisfying the provided test cases. Additionally, since the training set provides over a hundred solutions per problem, we perform near-deduplication on the solutions and limit them to a maximum of 25 solutions per problem.

Metrics. We assess the code generation performance of the models using the PASS@K metric Kulal et al. (2019); Chen et al. (2021), which evaluates the functional correctness of generated programs. For each problem, we generate N solutions (where $N \geq 2K$) and compute the expected number of scenarios in which the problem is solved at least once when sub-selecting a random sample of K solutions. We vary K in $\{1, 10, 25\}$ for APPS dataset and $\{1, 10, 100\}$ for the CODE-CONTESTS benchmark. We present more details about sampling hyperparameters in Appendix A.

3.2 Data Transformations

We apply our data transformation approach on the APPS and CODE-CONTESTS datasets. Unless specified otherwise, we use GPT-3.5-TURBO as our default language model $\mathcal{M}$ to perform the transformations and use a default temperature 0.3. In case of failure, we retry up to 5 iterations. We obtain three *parallel* datasets at the end of our cleaning process, one for each of renaming, modularization, and planning. Table 1 provides a summary of the generated datasets along with the instructions used to generate them. We provide complete details about the transformations in Appendix B.

We also simulate a simple distillation baseline, analogous to direct synthetic data generation approaches similar to Gunasekar et al. (2023). Specifically, we generate solutions for the training problems using the GPT-3.5-TURBO model. We use in-context learning with the two-shot prompt examples selected from our $\mathcal{D}_{modular}$ dataset. To ensure diverse solutions, we use three distinct few-shot examples and generate eight solutions for every prompt at a temperature of 0.5. Additionally, we filter the solutions that do not pass the test cases and refer to this dataset as $\mathcal{D}_{distill}$.

	APPS-INTRODUCTORY			APPS-INTERVIEW		
	PASS@1	PASS@10	PASS@25	PASS@1	PASS@10	PASS@25
In-context Learning						
CL-7B + $\mathcal{D}_{original}$	14.2	29.2	38.4	1.8	7.3	10.4
CL-7B + $\mathcal{D}_{modular}$	17.5	30.1	39.7	2.2	8.6	12.3
	+3.3	+0.9	+1.3	+0.4	+1.3	+1.9
Fine-tuning						
CL-7B + $\mathcal{D}_{original}$	18.7	34.4	40.2	3.4	9.7	13.6
CL-7B + $\mathcal{D}_{modular}$	**22.7**	**36.9**	42.6	**4.2**	**11.0**	**15.0**
	+4.0	**+2.5**	**+2.4**	**+0.8**	**+1.3**	**+1.4**
CL-7B + $\mathcal{D}_{planning}$	22.1	**37.1**	**43.8**	3.7	10.5	14.8
CL-7B + $\mathcal{D}_{rename}$	19.2	36.6	42.9	4.0	10.7	14.6
CL-7B + $\mathcal{D}_{distill}$	21.1	35.3	40.5	-	10.8	14.5
Closed models						
CODE-DAVINCI-002 [1]	22.1	50.2	58.7	4.1	16.8	23.8

Table 2: **Results on APPS dataset.** We use the CODELLAMA-7B model (referred to as CL-7B) and evaluate it under both fine-tuning and in-context learning settings. We use samples from the original and our transformed datasets in these settings and find that our cleaned datasets improve the performance of the model by up to **30%**. The green highlighted numbers depict the improvements obtained from using our $\mathcal{D}_{modular}$ dataset over the $\mathcal{D}_{original}$ dataset.

3.3 Experiment Details

To evaluate the *quality* of the transformed datasets, we measure how they impact the test benchmark accuracy. We study both in-context learning and fine-tuning using examples from our datasets.

Models. We use the CODELLAMA-7B model Rozière et al. (2023) in all our experiments (referred as CL-7B ahead).

In-context learning. We select two question-answer pairs from the $\mathcal{D}_{original}$ and $\mathcal{D}_{modular}$ training sets as our in-context learning example.

Fine-Tuning. We perform full fine-tuning over the base CL-7B model on the different datasets. We train the models for two epochs on the APPS dataset and one epoch on the CODE-CONTESTS dataset using a $5e^{-5}$ learning rate and an effective batch size of 256 on 4 A6000 GPUs.

4 Experimental Results

We present our experimental results in this section. Section 4.1 first provides a qualitative overview of the transformed programs and the remaining section presents the code generation results.

4.1 Analysis of the transformed programs

Data statistics. For the CODE-CONTESTS dataset, out of 98,582 programs extracted from the original dataset ($\mathcal{D}_{original}$), we can successfully transform 92,675 (94.0%) into our modularized dataset ($\mathcal{D}_{modular}$). We obtain similar success rates for the APPS dataset (details deferred to the appendix). On the contrary, the distilled dataset ($\mathcal{D}_{distill}$), which is constructed by generating solutions directly using GPT-3.5-TURBO only finds a correct solution for about 50% of the problems.

Analysis of the transformed programs. We find that our transformation approach decomposes the original programs by inserting three new functions on a median ($\sim$2.6 functions on average).To get a better understanding of the decomposition, we cluster the functions using their function names and signatures. We find that these helper functions often implement key program logic, standard algorithms, and utilities like handling inputs, outputs, and orchestrating the main function. Interestingly, we also find that the helper functions are often reused across problems, with small variations in implementations. For example, the top five most frequent helper functions, `dfs`, `build_graph`, `gcd`, `dp`, and `binary_search` occur in about 3-8% of the problems. Additionally, we qualitatively analyze a hundred random samples from $\mathcal{D}_{original}$ and $\mathcal{D}_{modular}$ datasets to determine the quality of performed transformations. Figures 4 to 11 in the appendix provide examples of such transformations. We find that most of the transformations are meaningful. They improve the readability of the programs and also find suitable decomposition for the program logic encoded in the control flow (see

[1] Model generations were obtained from Chen et al. (2022a)

Figure 4 as an example). However, in some cases, the generated helper functions can have improper names (`calculate_max_colors` in Figure 11) or complex implementations copied directly from the original program (`count_sequences` in Figure 12). Additionally, for simpler programs (Figure 13), the entire program functionality can be implemented in a single function and the *decomposition* does not provide any extra information.

Unlike, generated code, we cannot constrain or check the generated natural language plans. Thus, we find that sometimes the plans can be imprecise and vary in detail. While using a stronger pretrained model like GPT-4 could alleviate some of these issues, we believe this will be a good avenue for applying process supervision Lightman et al. (2023).

4.2 Main Result

Tables 2 and 3a provide our primary results on APPS and CODE-CONTESTS datasets respectively. We present our key results here and defer other results to the appendix.

We find that our data-cleaning approach improves the performance of the model on both APPS and CODE-CONTESTS datasets in both in-context learning and fine-tuning settings.

In-context Learning. We first evaluate the performance of the model when provided with *parallel* two-shot in-context learning examples from $\mathcal{D}_{original}$ and $\mathcal{D}_{modular}$ datasets each. We find that the PASS@1 improves from 14.2 to 17.5 (a 23% relative improvement) on the APPS-INTRODUCTORY dataset and PASS@100 improves from 7.2 to 9.3 (a 29% relative improvement) on the CODE-CONTESTS dataset. These results indicate that more readable and better-structured code helps the model in solving more problems.

Fine-tuning. Next, we fine-tune the model on the $\mathcal{D}_{original}$ and $\mathcal{D}_{modular}$ datasets and again find strong performance improvements from our transformation approach. Specifically, on the APPS-INTRODUCTORY dataset, the PASS@1 improves from 18.7 to 22.7 (a 23% relative improvement). Similarly, the CODE-CONTESTS dataset PASS@25 metric improves from 6.4 to 8.4 (30% relative improvement). These results cement our above findings about the effect of cleaning the data.

Interestingly, we also note that fine-tuning only provides modest improvements over the in-context learning performance. We hypothesize that this is due to the challenging nature of our task. Additionally, the in-context learning examples insert over 2,000 additional tokens to the sequence prefix and provide a much slower generation speed compared to the fine-tuned models.

5 Related Work

Data quality has been receiving increasingly more attention in the LLM literature, both in terms of improving the model performance and also for reducing the data requirements.

Instruction tuning. Instruction tuning refers to the process of finetuning a base pretrained LLM to perform general-purpose tasks and follow instructions. Recent works, Zhou et al. (2023); Cao et al. (2023); Chen et al. (2023) have demonstrated that a small high-quality instruction corpus is sufficient for achieving good instruction tuning performance. Here, we perform task-specific fine-tuning of LLMs and observe similar performance improvements.

Synthetic data for LLMs. Recent works have explored using synthetic datasets for general-purpose or task-specific finetuning of LLMs. These approaches work by generating synthetic datasets from a strong LLM (like GPT-3.5-TURBO or GPT-4) using a set of existing tasks Taori et al. (2023); Chiang et al. (2023) or generating new tasks using self-instruct Wang et al. (2022) or evol-instruct Xu et al. (2023) approaches. This has been also applied for task-specific finetuning – in common-sense reasoning West et al. (2022), text-summarization Sclar et al. (2022), mathematical reasoning Luo et al. (2023a); Yue et al. (2023), tool use Patil et al. (2023), coding Luo et al. (2023b), and general-purpose reasoning Li et al. (2023b).

More specifically, Yue et al. (2023) curates diverse corpus of mathematics problems with chain-of-thought Wei et al. (2022) or program-of-thought Chen et al. (2022b) annotations for mathematical reasoning analogous to our plans. Gunasekar et al. (2023) proposed pre-training models on programming "textbooks" generated synthetically from GPT-3.5-TURBO followed by finetuning the model on programming exercises generated similarly. Our work also studies curating synthetic data

for code-generation space. However, instead of directly generating data using LLMs, we identify better programming patterns and use our transformation approach to clean existing datasets.

6 Conclusion

Traditionally, data quality has primarily been linked to the correctness of programs, ignoring the rich stylistic aspects differing across programs. In this work, we demonstrate that stylistic aspects like readability, and program structuring actually impact the performance of the trained model on downstream tasks and thus also contribute to *data quality*. Next, we propose a LLM based data-cleaning pipeline that can be used for transforming programs to use better variable names, control flow structures, and function explanations in a scalable manner. While our evaluations focused on the algorithmic code generation task, we believe that these findings would also improve the general-purpose code generation capabilities of language models and be useful for software-engineering use cases.

Acknowledgement This work was supported in part by NSF grants CCF-1900968, CCF1908870 and SKY Lab industrial sponsors and affiliates Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

References

Cao, Y., Kang, Y., and Sun, L. (2023). Instruction mining: High-quality instruction data selection for large language models. *arXiv preprint arXiv:2307.06290*.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. (2022a). Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Chen, H., Zhang, Y., Zhang, Q., Yang, H., Hu, X., Ma, X., Yanggong, Y., and Zhao, J. (2023). Maybe only 0.5% data is needed: A preliminary exploration of low training data instruction tuning. *arXiv preprint arXiv:2305.09246*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. (2022b). Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., Stoica, I., and Xing, E. P. (2023). Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Fu, Y., Peng, H., Ou, L., Sabharwal, A., and Khot, T. (2023). Specializing smaller language models towards multi-step reasoning. *arXiv preprint arXiv:2301.12726*.

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Del Giorno, A., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., et al. (2023). Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. (2021). Measuring coding challenge competence with apps. *NeurIPS*.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. S. (2019). Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Li, J., Tworkowski, S., Wu, Y., and Mooney, R. (2023a). Explaining competitive-level programming solutions using llms. *arXiv preprint arXiv:2307.05337*.

Li, L. H., Hessel, J., Yu, Y., Ren, X., Chang, K.-W., and Choi, Y. (2023b). Symbolic chain-of-thought distillation: Small models can also "think" step-by-step. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2665–2679, Toronto, Canada. Association for Computational Linguistics.

Li, X.-Y., Xue, J.-T., Xie, Z., and Li, M. (2023c). Think outside the code: Brainstorming boosts large language models in code generation. *arXiv preprint arXiv:2305.10679*.

Li, Y., Bubeck, S., Eldan, R., Del Giorno, A., Gunasekar, S., and Lee, Y. T. (2023d). Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. (2022). Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. (2023). Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Luo, H., Sun, Q., Xu, C., Zhao, P., Lou, J., Tao, C., Geng, X., Lin, Q., Chen, S., and Zhang, D. (2023a). Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. *arXiv preprint arXiv:2308.09583*.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. (2023b). Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Magister, L. C., Mallinson, J., Adamek, J., Malmi, E., and Severyn, A. (2022). Teaching small language models to reason. *arXiv preprint arXiv:2212.08410*.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. (2023). Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. (2023). Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Sclar, M., West, P., Kumar, S., Tsvetkov, Y., and Choi, Y. (2022). Referee: Reference-free sentence summarization with sharper controllability through symbolic knowledge distillation. *arXiv preprint arXiv:2210.13800*.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., and Hashimoto, T. B. (2023). Stanford alpaca: An instruction-following llama model.

Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K.-W., and Lim, E.-P. (2023). Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. (2022). Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

West, P., Bhagavatula, C., Hessel, J., Hwang, J., Jiang, L., Le Bras, R., Lu, X., Welleck, S., and Choi, Y. (2022). Symbolic knowledge distillation: from general language models to commonsense models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4602–4625, Seattle, United States. Association for Computational Linguistics.

Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., and Jiang, D. (2023). Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Yue, X., Qu, X., Zhang, G., Fu, Y., Huang, W., Sun, H., Su, Y., and Chen, W. (2023). Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*.

Zelikman, E., Huang, Q., Poesia, G., Goodman, N. D., and Haber, N. (2023). Parsel: A (de-) compositional framework for algorithmic reasoning with language models. *arXiv preprint arXiv:2212.10561*.

Zhang, K., Wang, D., Xia, J., Wang, W. Y., and Li, L. (2023a). Algo: Synthesizing algorithmic programs with generated oracle verifiers. *arXiv preprint arXiv:2305.14591*.

Zhang, S., Chen, Z., Shen, Y., Ding, M., Tenenbaum, J. B., and Gan, C. (2023b). Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*.

Zhou, C., Liu, P., Xu, P., Iyer, S., Sun, J., Mao, Y., Ma, X., Efrat, A., Yu, P., Yu, L., et al. (2023). Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., et al. (2022). Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A Experimental Setup

APPS benchmark. We only retain problems from the `codeforces`, `codechef`, and `atcoder` competition websites. Specifically, the other websites or platforms provide very few (or no test cases) and use separate formats. While we considerably reduced the size of our training set, our test set is still quite close to the test set containing around 3800 problems instead of the default 5000.

Additionally, we note that some of the provided solutions in both APPS and CODE-CONTESTS datasets do not pass the test cases. These cases are sometimes due to incorrect programs but more often come from the fact that there can be multiple correct solutions are possible for the problem while only a single test solution is provided. We retain such samples for the APPS dataset and instead check whether the transformed program behavior is similar to the original program.

Metrics We use the PASS@K to perform our evaluations. We perform nucleus sampling using VLLM with $p = 0.95$. We outline the default sampling configurations used for computing the metrics

1. PASS@1 - We use a sampling budget (N) = 10 and temperature = 0.1.
2. PASS@10 - We use a sampling budget (N) = 50 and temperature = 0.6.
3. PASS@25 - We use a sampling budget (N) = 50 and temperature = 0.6.
4. PASS@100 - We use a sampling budget (N) = 200 and temperature = 0.8.

Finetuning details We finetune the CODELLAMA-7B model using deepspeed huggingface trainer. We use the following training configuration for our main experiments -

Training Parameters	**Values**
LR	$5e^{-5}$
Epochs	1 or 2 depending on the dataset
Batch Size	256 (combing grad. accumulation)
Dtype	bf16

B Code Transformations Implementation

We implement our code transformation approach using zero-shot prompting with GPT-3.5-TURBO model. After transformation, we extract the generated code and evaluate its functional correctness using the provided test cases. In case the program does not pass, we retry the process with up to a maximum of 5 attempts. In our experience, instruction-tuned models can follow precise commands and transform programs very well.

B.1 Renaming

We use the following prompt to perform renaming.

```
QUESTION:
{problem_statement}

ANSWER:
```python
{solution}
```
Rename the variables in the program to be descriptive, meaningful, and consistent. Do not
change the original semantics of the program. Enclose the program within backticks as shown
above and remember to use descriptive variable names.
```

B.2 Modularization

Unlike renaming, we perform two rounds of modularization in case the generated program consists of long function implementations (hinting that the function can be decomposed further). We use the following prompt to perform the first round of modularization

```
QUESTION:
{problem_statement}

ANSWER:
```python
{renamed_solution}
```
Refactor the above program. Follow the guidelines
* make the program more modular with smaller and meaningful helper functions
* good descriptive names for the helper functions
* have an entry function called `main()`
* `main()` is called inside `if __name__ == '__main__'`

Do not change the original semantics of the program significantly and no need to perform
optimizations. Enclose the program within backticks as shown above
```

Next, in case the modularized program contains a function with the number of lines greater than 20, we further prompt the model while signaling which functions to further decompose. This occurs in about 20-40% of modularized solutions and we use the following prompt.

```
QUESTION:
{problem_statement}

ANSWER:
```python
{modularized_solution}
```
Refactor the above program by modularizing it and breaking down long and complex functions
into smaller meaningful helper functions. Particularly refactor and decompose the following
function(s) into smaller helper functions - {function_names_string}
Only return the refactored program enclosed in backticks as shown above."""
```

B.3 Planning

We use the following prompt to generate natural language plans

```
QUESTION:
{problem_statement}

ANSWER:
```python
{modularized_solution}
```
Generate a summary for the following functions and classes in the program within four lines
each. The summaries should be descriptive and helpful for understanding the program (however
yet concise in four lines).
The functions and classes are -
{list_of_function_names}
Follow the provided format for the summaries while being informative and concise. Enclose the
 signatures in backticks as shown above.
```

	CODE-CONTESTS		
	PASS@10	PASS@25	PASS@100
In-context Learning			
CL-7B + $\mathcal{D}_{original}$	5.1	6.5	7.2
CL-7B + $\mathcal{D}_{modular}$	4.9	6.6	9.3
	-0.2	+0.1	+2.1
Fine-tuning			
CL-7B + $\mathcal{D}_{original}$	5	6.4	10.9
CL-7B + $\mathcal{D}_{modular}$	**6.1**	**8.3**	12.4
	+1.1	**+1.9**	**+1.5**
CL-7B + $\mathcal{D}_{planning}$	5.3	7.0	10.8
CL-7B + $\mathcal{D}_{rename}$	4.7	6.3	10.5
Closed models			
ALPHACODE-9B [2]	5.0	7.0	10.0
ALPHACODE-41B[3]	5.0	7.0	10.0
CODE-DAVINCI-002 [3]	3.0	-	7.5
GPT-3.5-TURBO[4]	-	-	18.2
+ BRAINSTORM[5]	-	-	29.3

(a) **Results on the CODE-CONTESTS dataset.** Similar to findings on the APPS dataset, we find that our data cleaning improves the performance.

	CODE-CONTESTS-PLAN		
	PASS@10	PASS@25	PASS@100
CL-7B + $\mathcal{D}_{original}$	6.5	9.5	15.0
CL-7B + $\mathcal{D}_{modular}$	8.8	11.8	17.8
CL-7B + $\mathcal{D}_{planning}$	6.9	10.5	15.4
CL-7B + $\mathcal{D}^{GT}_{plan}$	17.9	22.3	28.1
	+9.1	**+10.5**	**+11.3**

(b) **Effect of using ground-truth plans.** We disentangle the high-level reasoning vs coding capabilities by extracting ground-truth plans from solutions corresponding to the test problems. We find significant improvement in the performance on the CODE-CONTESTS-PLAN dataset, indicating that the model trained on the $\mathcal{D}_{planning}$ dataset while incapable of building correct plans, can follow such plans accurately

C Additional Results

C.1 Other results

C.1.1 Effect of planning

We next measure the effect of fine-tuning on the $\mathcal{D}_{planning}$ dataset which contains the natural language plans generated by our approach. Surprisingly, we find that planning only provides a modest improvement over the $\mathcal{D}_{modular}$ dataset (PASS@25 improved from 42.6 to 43.9 on the APPS-INTRODUCTORY dataset) or no improvements at all. These results are in sharp contrast to the literature on improving mathematical reasoning Yue et al. (2023); Magister et al. (2022); Fu et al. (2023) where training on chain-of-thought reasoning has been shown to be particularly effective. We hypothesize that this is an artifact of the difficult nature of our task and the diverse *reasoning* required for solving these problems. We leave a thorough comparison of these tasks for future work.

Upon inspection of the generated solutions, we find that often the generated plans are imprecise or incorrect, highlighting that planning still remains a bottleneck. To disentangle the high-level planning from the coding component, we analyze the performance of the model when provided with ground-truth plans on the CODE-CONTESTS dataset. We extract these ground-truth plans by applying our data transformation approach on the test set (similar to how $\mathcal{D}_{planning}$ training set was created). Since some problems in the test set do not have a corresponding correct `python` solution, we only consider a subset of 109 problems (instead of the original 165 problems). Table 3b provides results on this subset of the CODE-CONTESTS dataset. We find that while our model trained on the $\mathcal{D}_{planning}$ dataset is incapable of synthesizing new plans, it is capable of following the generated plans correctly. Specifically, all metrics showcase significant improvement, e.g. PASS@100 improving from 17.8 to 28.1, approaching the performance of GPT-3.5-TURBO, a much larger model!

C.1.2 Ablations

Effect of data size. Beyond improving the quality of the resulting model, data quality is also attributed to improving the data efficiency. We evaluate this aspect by fine-tuning our model on different fractions of $\mathcal{D}_{original}$ and $\mathcal{D}_{modular}$ datasets and find similar results. Figure 3 presents the performance of the model as a function of training set size. As shown in the figure, training on just one-eighth of $\mathcal{D}_{modular}$ dataset achieves similar PASS@1 as fine-tuning on the entire $\mathcal{D}_{original}$.

[2]Results sourced from Li et al. (2022)

[3]Results sourced from Zhang et al. (2023a)

[4]Results sourced from Li et al. (2023c)

```
def read_grid():
  n,m = input().split()
  ...

def remove_white_rows(grid):
  row_indices = []
  ...
  return grid

def remove_white_columns(grid):
  column_indices = []
  ...
  return grid

def main():
  grid = read_grid()
  grid = remove_white_rows(grid)
  grid = remove_white_columns(grid)
  print_grid(grid)
  ...
```

Figure 2: Example of a program generated by our model trained on the $\mathcal{D}_{modular}$ dataset. It correctly solves the problem by constructing smaller helper functions `remove_white_rows`, `remove_white_columns`

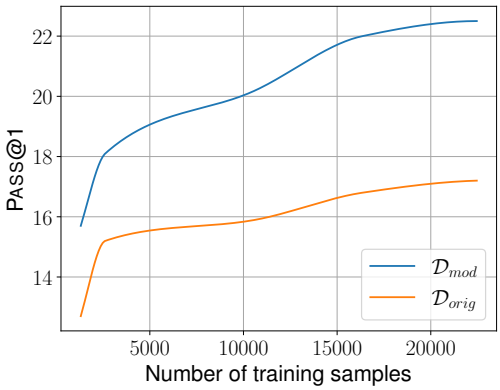

Figure 3: **Effect of quality on data-efficiency of the model**. Finetuning on 12.5% of clean $\mathcal{D}_{modular}$ dataset results in similar performance as finetuning on the entire $\mathcal{D}_{original}$ dataset.

Effect of renaming. We use variable renaming as an intermediate step in our cleaning process. We evaluate the performance of the model fine-tuned only on the $\mathcal{D}_{rename}$ dataset and find that renaming alone improves the performance when compared to fine-tuning on $\mathcal{D}_{original}$ dataset. For example, PASS@1 improved from 17.2 to 19.1. However, renaming still performs worse in comparison to fine-tuning on the $\mathcal{D}_{modular}$. This highlights that beyond just readable code, functional decomposition is also a key aspect of improving our performance.

Transformations vs Distillation. We compare our transformation approach with a direct distillation baseline where we directly generate solutions using GPT-3.5-TURBO, referred to as the $\mathcal{D}_{distill}$ dataset[5]. On the APPS-INTRODUCTORY dataset, we find that fine-tuning on the $\mathcal{D}_{modular}$ dataset achieves better performance compared to the $\mathcal{D}_{distill}$ dataset demonstrating the advantage of cleaning over the generation baseline.

C.1.3 Comparison to Other Baselines

We also provide a comparison to various In addition to CL-7B fine-tuned results, we also provide reference numbers from various closed-source models. Interestingly, our fine-tuned models outperform strong baselines like ALPHACODE on the CODE-CONTESTS dataset but still lag behind much larger CODE-DAVINCI-002 and GPT-3.5-TURBO models.

C.1.4 Case study of generated modularized program

Figure 2 provides an example of a program correctly generated by a model fine-tuned on our $\mathcal{D}_{modular}$ dataset. The problem requires removing rows and columns containing cells with certain attributes (i.e., if the cell is white) The modularized solution correctly identifies the steps required to solve the problem and implements them as separate helper functions, providing readable code.

We present the results on APPS-COMPETITION dataset here.

[5]Note that we generate these solutions using in-context examples from the $\mathcal{D}_{modular}$ dataset

	APPS-COMPETITION		
	PASS@1	PASS@10	PASS@100
Fine-tuning			
CL-7B + $\mathcal{D}_{original}$	0.2	1.7	3.1
CL-7B + $\mathcal{D}_{modular}$	0.5	2.3	3.2
	+0.3	**+0.6**	**+0.1**
CODE-DAVINCI-002	0.3	2.9	5.7

Table 4: Results on the APPS-COMPETITION datasets

D Examples of Transformed Program

```
def main():
    import sys
    input=sys.stdin.readline

    n,k=map(int,input().split())

    ab=[list(map(int,input().split())) for _ in
     [0]*n]

    g=[[] for _ in [0]*10]
    [g[b-1].append(a) for a,b in ab]
    [g[c].sort(reverse=True) for c in range(10)]

    for c in range(10):
        g[c]=[0]+g[c]

    for c in range(10):
        for i in range(2,len(g[c])):
            g[c][i]+=g[c][i-1]+2*(i-1)

            dp=[0]*(k+1)

    for c in range(10):
        dp2=[0]*(k+1)
        for i in range(len(g[c])):
            for j in range(k+1-i):
                dp2[i+j]=max(dp2[i+j],dp[j]+g[c
     ][i])
        dp=dp2

        print(max(dp))

        if __name__=='__main__':
    main()
```

(a) Original program

```
def read_input():
    ... (TRUNCATED)
    return num_books, num_sell, book_info

def group_books_by_genre(num_books, book_info):
    genre_books = [[] for _ in range(10)]
    for price, genre in book_info:
        genre_books[genre-1].append(price)
    return genre_books

def sort_books_by_price(genre_books):
    for genre in genre_books:
        genre.sort(reverse=True)
    return genre_books

def calculate_purchase_prices(genre_books):
    for genre in genre_books:
        genre.insert(0, 0)
    for genre in genre_books:
        for i in range(2, len(genre)):
            genre[i] += genre[i-1] + 2*(i-1)
    return genre_books

def calculate_max_purchase_price(num_sell,
     genre_books):
    dp = [0] * (num_sell+1)
    for genre in genre_books:
        dp2 = [0] * (num_sell+1)
        for i in range(len(genre)):
            for j in range(num_sell+1-i):
                dp2[i+j] = max(dp2[i+j], dp[j] +
      genre[i])
        dp = dp2
    return max(dp)

def main():
    num_books, num_sell, book_info = read_input
     ()
    genre_books = group_books_by_genre(num_books
     , book_info)
    genre_books = sort_books_by_price(
     genre_books)
    genre_books = calculate_purchase_prices(
     genre_books)
    max_purchase_price =
     calculate_max_purchase_price(num_sell,
     genre_books)
    print(max_purchase_price)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 4: Original and transformed programs

```
for _ in range(int(input())):
    p, q = map(int,input().split())
    c = q
    d = p

i = 1
factor = []
while i*i <= q:
    if q % i == 0:
        factor.append(i)
        if q//i != i:
            factor.append(q//i)
    i += 1

factor.sort(reverse=True)
factor.pop()
m = 1
for i in factor:
    d = p
    while d % c == 0:
        d //= i
    m = max(m, d)
print(m)
```

(a) Original program

```
def get_factors(q):
    factors = []
    i = 1
    while i*i <= q:
        if q % i == 0:
            factors.append(i)
            if q//i != i:
                factors.append(q//i)
        i += 1
    factors.sort(reverse=True)
    factors.pop()
    return factors

def get_largest_divisor(p, q, factors):
    largest_divisor = 1
    for i in factors:
        dividend_copy = p
        while dividend_copy % q == 0:
            dividend_copy //= i
        largest_divisor = max(largest_divisor,
     dividend_copy)
    return largest_divisor

def main():
    t = int(input())
    for _ in range(t):
        p, q = map(int, input().split())
        factors = get_factors(q)
        largest_divisor = get_largest_divisor(p,
      q, factors)
        print(largest_divisor)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 5: Original and transformed programs

```
def ncr(n, r, p):
    # initialize numerator
    # and denominator
    num = den = 1
    for i in range(r):
        num = (num * (n - i)) % p
        den = (den * (i + 1)) % p
    return (num * pow(den,
            p - 2, p)) % p
a,b,n=map(int,input().split())
ans=0
p=10**9+7
num=1
den=1
for i in range(n+1):
    s=a*i+b*(n-i)
    if i!=0:
        num=(num*(n-i+1))%p
        den=(den*(i))%p
    am=True
    while s!=0:
        if s%10!=a and s%10!=b:
            am=False
            break
        s//=10
    if am:
        ans=(ans+(num*pow(den,p-2,p))%p)%p
print(ans%p)
```

(a) Original program

```
def ncr(n, r, p):
    # calculate nCr modulo p
    # initialize numerator and denominator
    numerator = denominator = 1
    for i in range(r):
        numerator = (numerator * (n - i)) % p
        denominator = (denominator * (i + 1)) %
     p
    return (numerator * pow(denominator, p - 2,
     p)) % p

def is_good_number(number, a, b):
    # check if a number is a good number
    while number != 0:
        if number % 10 != a and number % 10 != b
     :
            return False
        number //= 10
    return True

def count_excellent_numbers(a, b, n):
    ans = 0
    p = 10**9 + 7
    numerator = 1
    denominator = 1

    for i in range(n + 1):
        sum_of_digits = a * i + b * (n - i)

        if i != 0:
            numerator = (numerator * (n - i + 1)
     ) % p
            denominator = (denominator * i) % p

        if is_good_number(sum_of_digits, a, b):
            ans = (ans + (numerator * pow(
     denominator, p - 2, p)) % p) % p

    return ans % p

def main():
    a, b, n = map(int, input().split())
    result = count_excellent_numbers(a, b, n)
    print(result)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 6: Original and transformed programs

```
import bisect
rev=[]
for i in range(1,10002):
    if str(i)==str(i)[::-1]:rev.append(i)
n=int(input())
ind= bisect.bisect_left(rev,n)
if abs(n-rev[ind-1])<=abs(n-rev[ind]):
    print(rev[ind-1])
else:
    print(rev[ind])
```

(a) Original program

```
import bisect

def generate_palindromes():
    palindromes = []
    for num in range(1, 10002):
        if str(num) == str(num)[::-1]:
            palindromes.append(num)
    return palindromes

def find_closest_palindrome(palindromes, n):
    index = bisect.bisect_left(palindromes, n)
    if abs(n - palindromes[index - 1]) <= abs(n
     - palindromes[index]):
        return palindromes[index - 1]
    else:
        return palindromes[index]

def main():
    palindromes = generate_palindromes()
    n = int(input())
    closest_palindrome = find_closest_palindrome
     (palindromes, n)
    print(closest_palindrome)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 7: Original and transformed programs

```
N = int(input())
T = [int(input()) for i in range(N)]
T.sort()
if N == 1:
    ans = 0
else:
    y = T[-1]
    R = []
    for x in range(1, y+1):
        if y % x == 0:
            R.append(x)
    cur = ans = 0
    for t in T:
        while R[cur] < t:
            cur += 1
        ans += R[cur] - t
print(ans)
```

(a) Original program

```
def find_divisors(max_interval):
    divisors = []
    for divisor in range(1, max_interval+1):
        if max_interval % divisor == 0:
            divisors.append(divisor)
    return divisors

def calculate_min_sum(intervals, divisors):
    current_index = min_sum = 0
    for interval in intervals:
        while divisors[current_index] < interval
         :
            current_index += 1
        min_sum += divisors[current_index] -
         interval
    return min_sum

def main():
    num_metronomes = int(input())
    intervals = [int(input()) for i in range(
     num_metronomes)]
    intervals.sort()
    if num_metronomes == 1:
        min_sum = 0
    else:
        max_interval = intervals[-1]
        divisors = find_divisors(max_interval)
        min_sum = calculate_min_sum(intervals,
     divisors)
    print(min_sum)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 8: Original and transformed programs

```
def main():
    from collections import defaultdict
    n, colors = int(input()), input()[::2]
    dsu, edges, d = list(range(n)), [],
     defaultdict(list)
    for _ in range(n - 1):
        u, v = map(int, input().split())
        u -= 1
        v -= 1
        if colors[u] == colors[v]:
            a, b = dsu[u], dsu[v]
            while a != dsu[a]:
                a = dsu[a]
            while b != dsu[b]:
                b = dsu[b]
            if a < b:
                dsu[b] = dsu[v] = a
            else:
                dsu[a] = dsu[u] = b
        else:
            edges.append(u)
            edges.append(v)
    for u, v in enumerate(dsu):
        dsu[u] = dsu[v]
    while edges:
        u, v = dsu[edges.pop()], dsu[edges.pop()
     ]
        d[u].append(v)
        d[v].append(u)

    def bfs(x):
        nxt, avail, t = [x], [True] * n, 0
        while nxt:
            t += 1
            cur, nxt = nxt, []
            for y in cur:
                avail[y] = False
                for y in d[y]:
                    if avail[y]:
                        nxt.append(y)
        return t if x else cur[0]

    print(bfs(bfs(0)) // 2)

if __name__ == '__main__':
    main()
```

(a) Original program

```
from collections import defaultdict

def find_root(vertex, dsu):
    while vertex != dsu[vertex]:
        vertex = dsu[vertex]
    return vertex

def merge_trees(u, v, dsu):
    root_u = find_root(u, dsu)
    root_v = find_root(v, dsu)
    if root_u < root_v:
        dsu[root_v] = dsu[v] = root_u
    else:
        dsu[root_u] = dsu[u] = root_v

def build_graph(num_vertices, colors, edges):
    dsu = list(range(num_vertices))
    graph = defaultdict(list)
    for u, v in edges:
        if colors[u] == colors[v]:
            merge_trees(u, v, dsu)
        else:
            graph[dsu[u]].append(dsu[v])
            graph[dsu[v]].append(dsu[u])
    return dsu, graph

def bfs(x, num_vertices, graph):
    next_vertices = [x]
    available = [True] * num_vertices
    t = 0
    while next_vertices:
        t += 1
        current_vertices, next_vertices =
     next_vertices, []
        for y in current_vertices:
            available[y] = False
            for neighbor in graph[y]:
                if available[neighbor]:
                    next_vertices.append(
     neighbor)
    return t if x else current_vertices[0]

def main():
    num_vertices = int(input())
    colors = input()[::2]
    edges = []
    for _ in range(num_vertices - 1):
        u, v = map(int, input().split())
        u -= 1
        v -= 1
        edges.append((u, v))
    dsu, graph = build_graph(num_vertices,
     colors, edges)
    print(bfs(bfs(0, num_vertices, graph),
     num_vertices, graph) // 2)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 9: Original and transformed programs

```
import heapq

def dfs(graph, start):
    n = len(graph)
    dist = [-0 for i in range(n + 1)]
    visited = [False for i in range(n + 1)]
    visited[start] = True
    stack = []
    dist[start] = 0
    heapq.heappush(stack, start)
    while stack:
        u = heapq.heappop(stack)
        for v in graph[u]:
            if not visited[v]:
                visited[v] = True
                dist[v] = dist[u] + 1
                heapq.heappush(stack, v)
    return dist

def solution():
    n, m, d = map(int, input().strip().split())
    p = list(map(int, input().strip().split()))
    graph = [[] for i in range(n + 1)]
    for i in range(n - 1):
        a, b = map(int, input().strip().split())
        graph[a].append(b)
        graph[b].append(a)
    dist = dfs(graph, 1)

    max_distance = -1
    u = -1
    v = -1
    for i in p:
        if dist[i] > max_distance:
            max_distance = dist[i]
            u = i

    distu = dfs(graph, u)

    max_distance = -1
    for i in p:
        if distu[i] > max_distance:
            max_distance = distu[i]
            v = i

    distv = dfs(graph, v)

    affected = 0
    for i in range(1, n + 1):
        if 0 <= distu[i] <= d and 0 <= distv[i]
     <= d:
            affected += 1

    print(affected)

solution()
```

(a) Original program

```
import heapq

def calculate_distances(graph, start):
    n = len(graph)
    distances = [-0 for i in range(n + 1)]
    visited = [False for i in range(n + 1)]
    visited[start] = True
    stack = []
    distances[start] = 0
    heapq.heappush(stack, start)
    while stack:
        current_node = heapq.heappop(stack)
        for neighbor in graph[current_node]:
            if not visited[neighbor]:
                visited[neighbor] = True
                distances[neighbor] = distances[
     current_node] + 1
                heapq.heappush(stack, neighbor)
    return distances

def find_possible_book_locations():
    n, m, d = map(int, input().strip().split())
    affected_settlements = list(map(int, input()
     .strip().split()))
    graph = [[] for i in range(n + 1)]
    for i in range(n - 1):
        a, b = map(int, input().strip().split())
        graph[a].append(b)
        graph[b].append(a)
    return calculate_possible_locations(n, m, d,
      affected_settlements, graph)

def calculate_possible_locations(n, m, d,
     affected_settlements, graph):
    distances = calculate_distances(graph, 1)
    max_distance, u = find_max_distance(
     distances, affected_settlements)
    distances_u = calculate_distances(graph, u)
    max_distance, v = find_max_distance(
     distances_u, affected_settlements)
    distances_v = calculate_distances(graph, v)
    return count_possible_locations(n, d,
     distances_u, distances_v)

def find_max_distance(distances,
     affected_settlements):
    max_distance = -1
    u = -1
    for settlement in affected_settlements:
        if distances[settlement] > max_distance:
            max_distance = distances[settlement]
            u = settlement
    return max_distance, u

def count_possible_locations(n, d, distances_u,
     distances_v):
    possible_locations = 0
    for i in range(1, n + 1):
        if 0 <= distances_u[i] <= d and 0 <=
     distances_v[i] <= d:
            possible_locations += 1
    return possible_locations

def main():
    possible_locations =
     find_possible_book_locations()
    print(possible_locations)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 10: Original and transformed programs

```python
n = int(input())
dividers = list()
i = 2

while i * i <= n:
    if n % i == 0:
        dividers.append(i)

        while n % i == 0:
            n /= i

    i += 1

if n > 1:
    dividers.append(n)

if len(dividers) == 1:
    print(dividers[0])
else:
    print(1)
```

(a) Original program

```python
def find_dividers(path_length):
    dividers = []
    divisor = 2

    while divisor * divisor <= path_length:
        if path_length % divisor == 0:
            dividers.append(divisor)
            while path_length % divisor == 0:
                path_length //= divisor
        divisor += 1

    if path_length > 1:
        dividers.append(path_length)

    return dividers

def calculate_max_colors(path_length):
    dividers = find_dividers(path_length)
    if len(dividers) == 1:
        return dividers[0]
    else:
        return 1

def main():
    path_length = int(input())
    max_colors = calculate_max_colors(
     path_length)
    print(max_colors)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 11: Original and transformed programs

```
import sys
readline = sys.stdin.readline

N, M = map(int, readline().split())
mod = 10**9+7
dpscc = [[0]*(N+1) for _ in range(N+1)]
dpus = [[0]*(N+1) for _ in range(N+1)]
dpscc[1][0] = 1

for m in range(M):
    dpscc2 = [[0]*(N+1) for _ in range(N+1)]
    dpus2 = [[0]*(N+1) for _ in range(N+1)]
    for i in range(1, N+1):
        for j in range(N+1-i):
            kscc = dpscc[i][j]
            kus = dpus[i][j]
            dpscc2[i][j] = (dpscc2[i][j] + i*
     kscc) % mod
            dpus2[i][j] = (dpus2[i][j] + j*(kus+
     kscc)) % mod
            dpscc2[i+j][0] = (dpscc2[i+j][0] + i
     *kus) % mod
            if N-i-j:
                dpus2[i][j+1] = (dpus2[i][j+1] +
      (N-i-j)*(kus+kscc)) % mod

    dpscc = [d[:] for d in dpscc2]
    dpus = [d[:] for d in dpus2]
print(dpscc[N][0])
```

(a) Original program

```
import sys

def count_sequences(num_towns, num_days):
    mod = 10**9+7
    dp_same_city_count = [[0]*(num_towns+1) for
     _ in range(num_towns+1)]
    dp_unique_city_count = [[0]*(num_towns+1)
     for _ in range(num_towns+1)]
    dp_same_city_count[1][0] = 1

    for day in range(num_days):
        dp_same_city_count2 = [[0]*(num_towns+1)
      for _ in range(num_towns+1)]
        dp_unique_city_count2 = [[0]*(num_towns
     +1) for _ in range(num_towns+1)]
        for i in range(1, num_towns+1):
            for j in range(num_towns+1-i):
                same_city_count =
     dp_same_city_count[i][j]
                unique_city_count =
     dp_unique_city_count[i][j]
                dp_same_city_count2[i][j] = (
     dp_same_city_count2[i][j] + i*
     same_city_count) % mod
                dp_unique_city_count2[i][j] = (
     dp_unique_city_count2[i][j] + j*(
     unique_city_count+same_city_count)) % mod
                dp_same_city_count2[i+j][0] = (
     dp_same_city_count2[i+j][0] + i*
     unique_city_count) % mod
                if num_towns-i-j:
                    dp_unique_city_count2[i][j
     +1] = (dp_unique_city_count2[i][j+1] + (
     num_towns-i-j)*(unique_city_count+
     same_city_count)) % mod

        dp_same_city_count = [d[:] for d in
     dp_same_city_count2]
        dp_unique_city_count = [d[:] for d in
     dp_unique_city_count2]

    return dp_same_city_count[num_towns][0]

def main():
    num_towns, num_days = map(int, input().split
     ())
    result = count_sequences(num_towns, num_days
     )
    print(result)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 12: Original and transformed programs

```
n,m = [int(i) for i in input().split()]
seg = {i:[] for i in range(1,n+1)}
for j in range(m):
    a,b = [int(i) for i in input().split()]
    seg[a].append(b)
    seg[b].append(a)
tail = [0]*(n+1)
tail[1] = 1
for i in range(2,n+1):
    temp = [tail[j] for j in seg[i]]+[0]
    tail[i] = max(temp)+1
temp = [len(seg[i])*tail[i] for i in range(1,n
    +1)]
print(max(temp))
```

(a) Original program

```
def calculate_beauty(num_points, num_segments,
     segments):
    tail_length = [0]*(num_points+1)
    tail_length[1] = 1
    for i in range(2, num_points+1):
        temp = [tail_length[j] for j in segments
     [i]]+[0]
        tail_length[i] = max(temp)+1
    spine_length = [len(segments[i])*tail_length
     [i] for i in range(1, num_points+1)]
    return max(spine_length)

def main():
    num_points, num_segments = [int(i) for i in
     input().split()]
    segments = {i:[] for i in range(1,
     num_points+1)}
    for j in range(num_segments):
        point1, point2 = [int(i) for i in input
     ().split()]
        segments[point1].append(point2)
        segments[point2].append(point1)
    result = calculate_beauty(num_points,
     num_segments, segments)
    print(result)

if __name__ == '__main__':
    main()
```

(b) Transformed program

Figure 13: Original and transformed programs

E Related Work

Algorithmic Code Generation. Code generation is a broad domain and we only focus on algorithmic code generation literature. Hendrycks et al. (2021) released the APPS dataset and evaluate the performance of small LLMs on it. Li et al. (2022) released the CODE-CONTESTS dataset and also developed the ALPHACODE models which depicted strong performance on the algorithmic code generation tasks using massive pretraining and fine-tuning. Chen et al. (2021) introduced earlier version of CODE-DAVINCI-002 models and applied it the APPS dataset. Zhang et al. (2023b) proposed a lookahead-search-based decoding algorithm for improving *reasoning* in LLMs and is orthogonal to our work. Zhang et al. (2023a) proposed the ALGO, that first generates a *brute-force* solution for the problem and then uses it as an oracle for generating further solutions. More recently, Zelikman et al. (2023) proposed the PARSEL approach which used the CODE-DAVINCI-002 model to first generate a plan in their high-level problem-specification language and then generate a program using it. We instead finetune smaller LLMs on natural language based plan descriptions. Li et al. (2023a) also study disentangling the high-level planning and low-level code generation capabilities of LLMs, similar to our experiments in Section C.1.1. However, the extract strong pre-training closed-source models while evaluate locally fine-tuned LLMs in our work.