Self-Edit: Fault-Aware Code Editor for Code Generation

Kechi Zhang, Zhuo Li, Jia Li, Ge Li[*], Zhi Jin[*]
Key Lab of High Confidence Software Technology (PKU), Ministry of Education
School of Computer Science, Peking University, China
{zhangkechi,lizhmq}@pku.edu.cn, lijia@stu.pku.edu.cn,
{lige,zhijin}@pku.edu.cn

Abstract

Large language models (LLMs) have demonstrated an impressive ability to generate codes on competitive programming tasks. However, with limited sample numbers, LLMs still suffer from poor accuracy. Inspired by the process of human programming, we propose a generate-and-edit approach named Self-Edit that utilizes execution results of the generated code from LLMs to improve the code quality on the competitive programming task. We execute the generated code on the example test case provided in the question and wrap execution results into a supplementary comment. Utilizing this comment as guidance, our fault-aware code editor is employed to correct errors in the generated code. We perform extensive evaluations across two competitive programming datasets with nine different LLMs. Compared to directly generating from LLMs, our approach can improve the average of pass@1 by 89% on APPS-dev, 31% on APPS-test, and 48% on HumanEval over nine popular code generation LLMs with parameter sizes ranging from 110M to 175B. Compared to other post-processing methods, our method demonstrates superior accuracy and efficiency.

1 Introduction

Large language models (LLMs) have recently been applied to the competitive programming task. This task requires understanding a complex natural language description of a problem with example test cases and correctly implementing solutions that can span hundreds of lines. Solutions are evaluated by executing them on hidden test cases. However, existing LLMs often have low accuracy and pass rates in this task. For example, on a popular competitive programming benchmark *APPS-test* (Hendrycks et al., 2021), the nearly most powerful model GPT3 (Brown et al., 2020) achieves only 7% accuracy when allowed to submit only one program per task (referred to as *pass@1*).

[*]Corresponding authors

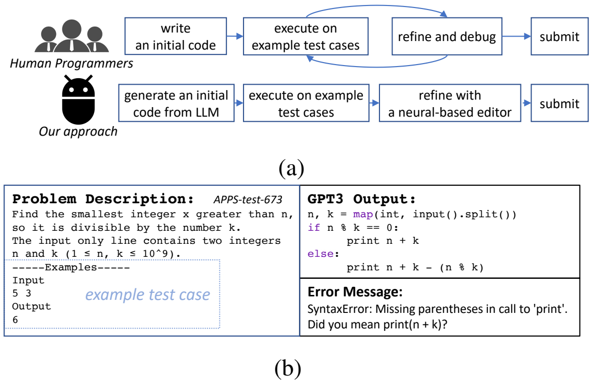


Figure 1: (a) Our approach is inspired by the problem-solving process of human programmers. (b) Output from *GPT3* model on APPS-test dataset and its corresponding error message, which is obtained by running on the example test case.

To improve the performance of LLMs on the competitive programming task, we take inspiration from the process of human programming. When solving competitive programming problems, programmers usually write an initial program, execute some example test cases, and refine the code based on the test results. In this process, a programmer can take key information (e.g, program outputs or compile/runtime error message) from the test results, which helps them debug the program. We instantiate this idea by adopting a similar pipeline with a neural-based editor (in Figure 1(a)). Analyzing the code generated by a pre-trained LLM, we have found that some of the generated codes can be improved with minor modifications. Figure 1(b) shows an example of generated code by GPT3 on the APPS-test dataset. GPT3 generates code that is inconsistent with the problem description. We notice that the error message directly points out the bug in the code, with which we can quickly fix the error. It motivates us to investigate approaches to edit and improve the quality of the code generated by LLMs with the help of execution results.

In this work, we propose a novel generate-and-edit approach to augment LLMs on the competitive

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 1: Long Papers, pages 769–787
July 9-14, 2023 ©2023 Association for Computational Linguistics

programming task, named Self-Edit. To mimic the above human programmers' behavior, our approach incorporates the ability of LLMs in three steps: ❶ *Generation with LLMs*. We use large language models as black-box generators and generate the program based on the problem description. ❷ *Execution*. Given a generated code from LLMs, we execute it on the example test case to get the execution results. We further wrap the execution results with templates as supplementary comments to include additional helpful information for editing. ❸ *Edit*. We develop a fault-aware neural code editor that takes the generated code and supplementary comment as input and refines the code. Our code editor aims to improve the quality and accuracy of code generation using LLMs.

We conduct extensive experiments on two public competitive programming benchmarks, including APPS (Hendrycks et al., 2021) and HumanEval (Chen et al., 2021). We apply our approach to 9 popular LLMs with parameter sizes ranging from 110M to 175B to show the universality. Compared to directly generating from LLMs, we have several findings: ❶ Our approach significantly improves the performance of LLMs. In particular, our approach improves the average of pass@1 by 89% on APPS-dev and 31% on APPS-test. Even for the chosen largest language model GPT3-175B, our relatively small editor model can improve pass@1 from 26.6% to 32.4% on the APPS-dev benchmark. ❷ Our approach is generalizable on a different style of dataset HumanEval, improving the average of pass@1 by 48%, showing the transfer ability on the out-of-distribution benchmark.

Recently some approaches are also proposed to post-process programs generated by LLMs (Shi et al., 2022; Inala et al., 2022; Chen et al., 2022; Zhang et al., 2022). These approaches do large-scale sampling from LLMs, rerank these sampled programs, and output the final program. In comparison, our self-edit framework has two advantages: ❶ Our approach maintains a constant sample budget and significantly reduces the computational overhead for LLMs. ❷ Our editor directly modifies the programs and outperforms these reranking-based methods, especially with a limited sample budget such as pass@1. ***To our knowledge, we are the first to adopt an editing-based post-processing method for competitive programming tasks.***

The contributions are listed as follows:

- We propose a generate-and-edit approach named Self-Edit for large language models (LLMs) to generate high-quality code for competitive programming tasks.
- We develop a fault-aware neural code editor that takes the generated code and error messages as input and uses them to refine the code, improving its quality and accuracy.
- We conduct experiments on two popular datasets and nine LLMs to demonstrate the effectiveness and universality of our approach.

2 Related Work

2.1 Code Generation

Code generation is a process in which source code is automatically generated based on functional requirements such as natural language descriptions (Iyer et al., 2018; Yin and Neubig, 2018; Li et al., 2023a,b,c) or pseudo code algorithms (Kulal et al., 2019; Oda et al., 2015) or a old version of code (Li et al., 2022a) or a response from programming tools (Zhang et al., 2023). One particularly challenging type of code generation task is competitive programming (Li et al., 2022c), in which models must solve problems at the level of programming competitions. This task often involves natural language descriptions and example input-output pairs. The performance of a code generation model on competitive programming tasks can serve as a measure of its ability to create complete solutions to problems. In recent years, large pre-trained language models such as AlphaCode (Li et al., 2022c) and the GPT3 (Brown et al., 2020) series have demonstrated impressive capabilities in code generation and competitive programming. Other open-source code generation models include GPT-Neo (Black et al., 2021), GPT-J (Wang and Komatsuzaki, 2021), CodeParrot (Wolf et al., 2020), PolyCoder (Xu et al., 2022), CodeGen (Nijkamp et al., 2022) and InCoder (Fried et al., 2022). We utilize the *text-davinci-002* API from OpenAI and various competitive code generation models in this work.

2.2 Post-processing of LLMs for code generation

To find the correct code solutions based on LLMs, researchers adopt various post-processing methods to filter/rerank the original outputs from LLMs. In the domain of solving math problems, Cobbe et al. (2021) and Shen et al. (2021) chose the one

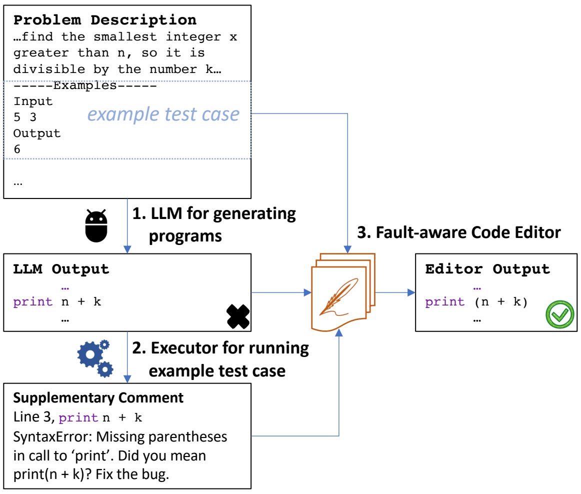
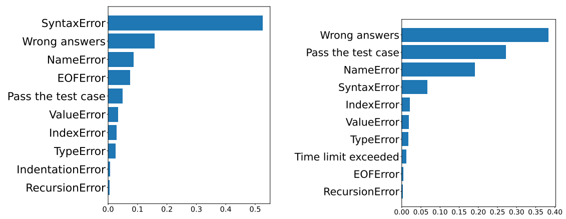


Figure 2: Pipeline of our self-edit approach.



(a) *PyCodeGPT-110M-finetuned* (b) *GPT3*

Figure 3: Distribution of the top 10 classes of supplementary comments in the APPS-train dataset when using the *PyCodeGPT-110M-finetuned* and *GPT3* models, expressed as a percentage of the total number of generated programs for each class.

with the highest rank by a trained ranker. Similar ranking methods are also used in the field of cross-domain adaptation (Li et al., 2022b). In the domain of code generation, post-processing techniques are also often used (Lahiri et al., 2022; Le et al., 2022). AlphaCode (Li et al., 2022c) and Shi et al. (2022) adopted the clustering and filtering methods based on the execution output of the generated programs. Inala et al. (2022) trained a fault-aware neural ranker to rerank the outputs with a large sample budget. Chen et al. (2022) use the large models to generate test cases for themselves and automatically rank the solutions based on the test-driven dual execution agreement. Zhang et al. (2022) reranked the LLM outputs with the generation probability of back translation.

However, these existing methods require large-scale sampling. They need to generate a large number of programs for post-processing. For example, AlphaCode (Li et al., 2022c) needs 1 million samples per problem, costing 10^5 TPU-seconds. In the real world, computing resources are precious and limited, and existing methods are ineffective in practical applications. Our self-edit approach addresses this issue by maintaining a constant sample budget and improving computational efficiency, described in Section 4.3.

3 Methodology

We provide an overview of the self-edit pipeline in Figure 2. Given the problem description, We first generate the initial code with LLM. Then we execute the example test case to obtain test results and construct the supplementary comment. Finally, we train a fault-aware code editor model to refine the code based on the problem description, generated code, and supplementary comment.

3.1 LLMs as Black-box Generator

We use large language models as black-box generators with fixed parameters in our design. This design choice is motivated by the fact that training LLMs is costly, and access to LLMs is often restricted. (*E.g.*, OpenAI only offers paid API to infer GPT3.) Using LLM as a black-box generator makes our approach flexible for using different LLMs. We investigate nine LLMs for code generation with sizes ranging from 110M to 175B. A detailed comparison is described in Table 2.

3.2 Executor and Supplementary Comments

After we generate the code using LLMs, we use an executor to run the example test case. We classify the execution results into three types: ❶ Passed: The program passes the test case. ❷ Wrong Answer: The program runs normally but gives incorrect outputs. ❸ Error: The program terminates abnormally due to syntax error, runtime exceptions, or exceeding time limit.

We analyze the distribution of test results on APPS-train dataset for code generated by a relatively small model PyCodeGPT-110M and a large model GPT3-175B as shown in Figure 3. We observe that programs produced by different models yield different test result distributions. Code generated by smaller models (PyCodeGPT) tends to encounter SyntaxError issues more frequently, while large models (GPT3) show fewer SyntaxErrors, fewer RuntimeErrors, but more normally executed cases.

In order to construct meaningful supplementary comments for the code editor model to understand

```
Comment 1:
Pass the example test case.
```

```
Comment 2:
Template: Wrong Answer with input: <input>. Expected
output is <output_1>, but generated output is <output_2>.
Rewrite the code.
Example: Wrong Answer with input: 2 5 3. Expected output
is 1, but generated output is 0. Rewrite the code.
```

```
Comment 3:
Template: Line <lineno>, <line_content>, <error_msg>.
Fix the bug.
Example: Line 2,
return len([i for i in str(i**2) for i in range(n+1) if
i == str(d)])
NameError: name 'i' is not defined. Fix the bug.
```

Figure 4: Example Supplementary Comments in different situations.

various execution results, we design the comment templates (Fig. 4) for the three types of test results. The comment template can wrap potential error messages with additional helpful information for editing. ❶ For the code passing the example test case, we use *Comment 1*: "Pass the example test case.". ❷ For the code producing incorrect outputs, we use *Comment 2* to include the relevant input, expected output, and the actual output. We also append the instruction "Rewrite the code" to guide the editor model to reimplement the algorithm to produce correct outputs. ❸ For the code that terminates with errors, we use *Comment 3* to include the error line number, line context, and full error message. These supplementary comments provide additional context and clarity for the generated code and are used to guide editing the code.

3.3 Fault-aware Code Editor

Once we have constructed the supplementary comments, we train a fault-aware editor that takes the natural language description, generated code, and supplementary comments as input and produces higher-quality refined code.

3.3.1 Code Editor Models

The fault-aware code edit task is formally defined as a sequence-to-sequence task: given a natural language description N, a program generated by LLM S, and accompanied supplementary comments C (Sec. 3.2), the model is required to generate higher-quality code $\hat{C}$ that implements the natural language description and passes test cases. In our experiments, the input pair (N, S, C) is segmented into three parts and concatenated using special separator tokens, represented as $[SOS], n_1, n_2, \ldots, n_{|N|}, [CODE], s_1, \ldots, s_{|S|}, , [CMNT], c_1, \ldots, c_{|C|}, [EOS]$, where the lowercase letters represent the token of the corresponding content in the input pair (N, S, C). We train a decoder-only model to complete the code edit task. Concretely, we implement the code editor by fine-tuning *PyCodeGPT-110M* on this task.

At inference time, we first generate multiple programs from LLMs using natural language description as input. For each generated program, we feed the example test case provided in the description into the executor to obtain a fault-aware comment. We then use the editor to generate a new program, which is the final version for further evaluation. This inference approach maintains a small sample budget compared with existing large-scale sampling and filter/reranking methods.

3.3.2 Dataset Construction for Code Editor

To train a fault-aware code editor, we need datasets that contain the generated program and the corresponding supplementary comments. To collect such datasets, we use different LLMs (Sec. 4.1) to generate candidate programs for problems in the APPS-train dataset. For each problem, we sample 10 programs from the LLM and then execute the example test case to get the test results and construct supplementary comments. At this point, we get the datasets of triplets (N, S, C) for different LLMs. To further obtain the ground truth program $\hat{C}$, we collect the standard ground truth programs in the original APPS training dataset and the generated programs that pass all hidden test cases. For each LLM, we create an individual editor dataset with nearly 4.5k generated programs with comments. For each generated program, we set at most 15 ground truth programs. As we described in Figure 3, the generated programs from different LLMs have different distributions of the corresponding comments. To optimize the performance of the fault-aware code editor for each LLM, it is necessary to use training datasets specific to the corresponding LLM.

3.3.3 Training Objective of Code Editor

Editing for a high-quality program based on the input pair (N, S, C) is a one-of-many task because multiple correct target programs satisfy the requirements. Standard maximum likelihood objectives aim to minimize loss by considering all of the solutions in the training set (like recall), while we focus on a model's ability to edit a single correct solution based on the existing generated code within a limited budget of attempts (like precision). To address this discrepancy, we follow previous work and adopt a variation of GOLD (Pang and He, 2021; Li et al., 2022c), which incorporates an off-policy importance weight into the standard maximum likelihood objective gradient:

$$\nabla\mathcal{L}(\theta) = -\sum_{t\in\hat{C}} P_\theta(t)\nabla log P_\theta(t) \qquad (1)$$

where θ represents the model parameters and $log P_\theta(t)$ is the standard log-likelihood objective for next token prediction. The additional weight $P_\theta(t)$ allows the model to focus on the tokens that already have a high likelihood, so the model can concentrate on these easier-to-learn ground truth solutions and increase the chance of getting at least one correct output. Such a loss setting allows editors to learn to copy part of the content from existing generated programs to obtain better outputs.

4 Experiment

We present extensive experiments that span two representative datasets and nine different LLMs for code generation, whose parameter counts range across four orders of magnitude. The details of the adopted LLMs are described in Section 3.1. We aim to investigate four research questions: (1) how much can fault-aware code editors improve various code generation models on competitive programming (Sec. 4.2), (2) the advantages of editor-based methods over existing ranking methods (Sec. 4.3), (3) to what extent does the supplementary comments help to refine the program (Sec. 4.4), (4) how does the number of editing rounds affect the final result (Sec. 4.5).

4.1 Experiment Setup

Dataset. We consider evaluating our approach on two existing code generation datasets: (1) **APPS** (Hendrycks et al., 2021): a collection of 5000 training and 5000 test tasks collected from coding competitions and interview problems. The test set has three different difficulty levels: Introductory, Interview, and Competition. (2) **HumanEval** (Chen et al., 2021): a set of 164 test programming problems with a function signature, docstring, body, and several unit tests. Our experiments only use the APPS-train dataset to finetune the code generation models and the code editor models since it is the largest training dataset. Following previous studies (Inala et al., 2022), we adopted the same division and used a set of 598 tasks excluded from the APPS training dataset for validation[1]. The detailed statistic of the datasets is shown in Table 1. The hidden test cases are those test cases for evaluation. They are not included in the problem description, so they are distinguished from the example test case used to obtain supplementary comments.

			Problems	Hidden Tests
Training dataset	APPS-train		4207	5.56
Testing benchmark	APPS-dev		598	4.03
	APPS-test	Introductory	1000	21.19
		Interview	3000	
		Competition	1000	
	HumanEval		164	8.08

Table 1: Statistics of training dataset and testing benchmarks: the total number of problems in datasets (*Problems*), the average number of hidden test cases per problem (*Hidden Tests*).

Base LLMs. In this paper, we investigate the effectiveness of several widely used language models for code generation, including text-davinci-002 (175B) (Brown et al., 2020), CodeGen (2B, 350M) (Nijkamp et al., 2022), InCoder (1B) (Fried et al., 2022), GPT-Neo (1.3B, 125M) (Black et al., 2021), GPT-J (6B) (Wang and Komatsuzaki, 2021) and PycodeGPT (110M) (Zan et al., 2022). These models are evaluated under zero-shot or finetune experimental conditions, with additional descriptions provided as a part of Table 2. [2]

Editor Model. We implement the code editor by fine-tuning *PyCodeGPT-110M*. We choose this model because of its relatively small parameter size and high performance. We also tried the *CodeGen-350M* model in early experiments but found that the training speed and final performance were not as good as the model we chose.

Considering that LLMs shows strong in-context learning abilities that do not need training process, we also explore to design a variant of our self-edit method with in-context learning. We use the *text-davinci-002* as both base model and editor model. The in-context learning self-edit performances are discussed in Section 5.2.

Metrics. We use the metric pass rate ***pass@k*** for performance evaluation and take advantage of hidden test cases to determine the functional correctness of code solutions. For each problem, we submit k code solutions for evaluation. If any of the

[1]https://github.com/microsoft/CodeRanker

[2]We do not use the *CodeX* model as it was in closed beta and was not available during our experiments. We choose *text-davinci-002* with equal parameter size as an alternative.

k code solutions passes all ground truth test cases, the problem is considered solved. Then *pass@k* is the percentage of solved problems. In our experiments, we set $k = \{1, 5, 10\}$.

To show the number of programs corrected by our editor, we design a new metric ***sol@k***, which means the total number of correct programs given k samples per problem. For example, for the 5000 problems in APPS-test, we will generate $5000 * k$ code solutions, from which we will count the number of correct solutions as *sol@k*. In our experiments, we set $k = 10$. We show the performance of the base model and the performance after editing (denoted as ***edit-pass@k*** and ***edit-sol@k***).

Training/Inference Settings. For each finetuned LLM, we limit the maximum epochs to 10 with a learning rate of 1e-5, and choose the best checkpoint based on the validation loss on APPS-dev. We adopt the same training strategy to train fault-aware code editors on each corresponding editor dataset. We set the maximum input length to 1024 and output length to 512 for our editors. To extract the supplementary comment, we choose only one example test case contained in the problem description even if it contains multiple. At inference time, we use temperature sampling with T = 0.8 both for LLM and editor outputs. We limit the sample budget of LLMs to 10. For each LLM output code, we only generate one code as the final version with our editor. Thus the usage of the editor maintains a constant sample budget. All experiments are conducted with 4 Tesla V100-32GB GPUs.

4.2 Comparison with Base LLMs

APPS-dev & APPS-test. We first compare with directly generating from LLMs to analyze how fault-aware code editors can improve nine popular code generation models. Table 2 shows the primary results on the APPS-dev dataset for nine different code generation models. The fault-aware editor improves all code generation models despite their different sizes and training settings. The average pass@1 value across nine models increases from 6.17% to 11.67%, representing an impressive 89% improvement. For those LLMs with a particularly large number of parameters, our editor can also achieve a significant improvement. For *GPT3* with 175B parameters, the improvement of our editor also achieves 5.9%, 5.0%, 8.4% on pass@{1,5,10}.

Results on the APPS-test dataset are shown in Table 3. The test problems are more challenging than APPS-dev, which we can see by the smaller pass@k numbers. Our editors maintain significant improvement for models of different sizes. The absolute improvement of *pass@1* covers from 0.12% to 0.7%, showing that the editor can solve 6 to 35 more problems on this challenging benchmark. As for *sol@10*, our editors can additionally correct hundreds of generated codes from LLMs.

In some cases, we observe that the *edit-pass@1* outperforms the *pass@5*. It demonstrates that editing the candidate code is very sample efficient. With the editor model, the number of required programs sampled from the LLM can be reduced.

Another interesting observation is that a smaller LLM equipped with our editor can achieve comparable performance as the super large models. For example, the *GPT-Neo-125M*, *GPT-Neo-1.3B*, and *GPT-J* are pretrained and finetuned with the same dataset. Using the editor can fill in the gaps in the parameter sizes of this series of models. The 125M pretrained model with a 110M editor can significantly outperform a 1.3B pretrained model and even outperform the 6B pretrained model in some cases. This finding can also be observed in other experiments, showing that our editor can offer a boost approximately equivalent to a tens of times pretrained model size increase.

On Different Difficulty-Level Problems. Considering that the APPS-test dataset has three difficulty levels, we further analyze the improvement on problems of different difficulty in Table 5. We choose *GPT-J-6B-finetuned* as the base model because it has shown promising results on this challenging benchmark and has certain representativeness. The editor can improve the base model on problems of all difficulty levels but has a relatively high pass rate improvement on simple *"Introductory"* problems. We find that the output of LLMs is poor on very difficult problems, making it too difficult for the editor to correct these solutions. Even so, our method slightly improves the *"Competition"* problems when enlarging the sample budget from 1 to 10.

HumanEval. We also measure the transfer ability of our editor on HumanEval, a dataset of different styles, in Table 4. The HumanEval dataset requires the model to give the function body based on the function signature, comments, and example test cases. Following the executability filter in previous work (Zhang et al., 2022), in this dataset, we only edit the outputs that can not pass the example test

Code Gen. Model	Para.	pass@1	edit pass@1	pass@5	edit pass@5	pass@10	edit pass@10	sol@10	edit sol@10
finetuned									
PyCodeGPT	110M	4.8	11.4	7.9	15.1	8.9	17.1	286	659
GPT-Neo 125M	125M	1.5	8.5	6.7	10.2	10.2	17.2	102	501
CodeGen-350M	350M	1.7	5.7	2.5	9.2	3.2	13.5	103	339
GPT-Neo 1.3B	1.3B	4.0	10.5	10.9	18.6	17.2	25.4	200	663
InCoder-1B	1.3B	9.4	12.4	12.5	16.2	13.5	18.1	568	730
GPT-J	6B	6.0	12.0	17.9	27.8	24.6	37.8	365	750
zero-shot									
InCoder-1B	1.3B	0.2	4.7	0.8	7.7	1.2	9.9	13	270
CodeGen-2B	2.7B	1.3	7.4	5.9	14.0	9.7	19.7	92	438
text-davinci-002	175B	26.6	32.4	43.8	48.8	49.7	58.0	1626	1948

Table 2: Results on the APPS-dev dataset on how our fault-aware editors can improve the pass rates for different sample budgets with various code generation models. *"finetuned"* indicates we finetune those models on APPS-train dataset. *"zero-shot"* indicates we use those models in the zero-shot setting. We will use the best checkpoints of LLMs and editor models based on this validation set in other experiments.

Code Gen. Model	pass@1	edit pass@1	pass@5	edit pass@5	pass@10	edit pass@10	sol@10	edit sol@10
finetuned								
PyCodeGPT	0.20	0.64	0.38	0.98	0.44	1.24	126	308
GPT-Neo 125M	0.08	0.22	0.40	0.70	0.70	1.12	45	135
CodeGen 350M	0.20	0.32	0.30	0.56	0.32	0.84	92	149
GPT-Neo 1.3B	0.14	0.68	0.74	1.38	1.40	2.10	106	340
InCoder 1B	0.66	0.86	1.18	1.62	1.44	2.10	344	421
GPT-J	0.70	1.40	2.46	3.34	3.52	4.76	404	738
zero-shot								
InCoder 1B	0.00	0.24	0.02	0.50	0.02	0.76	1	121
CodeGen 2B	0.12	0.28	0.34	0.66	0.66	1.08	41	131
text-davinci-002	7.48	7.94	15.94	16.66	-	-	1876 †	1983 †

† As we access *GPT3* through a paid API, we limit the sample budget of *GPT3* as 5 for this large benchmark and evaluate *sol@5*.

Table 3: Results on the APPS-test dataset.

case. We also modify the input format to be similar to the format in the APPS dataset. We select several representative LLMs for evaluation within our computational capabilities. We can again see that the editor improves the performance of all code generation models on all metrics. We notice that under larger sample budget conditions, even if the pass@10 does not increase for *CodeGen-2B*, our editor can still correct more generated solutions. Thus the *sol@10* increases significantly. These results demonstrate the ability and generality of our editor to correct out-of-distribution output codes.

4.3 Comparison with Post-processing Baseline

This experiment compares our self-edit approach with existing post-processing methods for code generation. We choose to compare with CodeRanker (Inala et al., 2022), a state-of-the-art reranking method on the APPS dataset. CodeRanker finetuned CodeBERT (125M) to classify the potential error type and use this classification prediction to rerank the generated codes from LLMs. The supervised training task makes this method more efficient than previous filtering and reranking methods. However, our experiments (Table 6) prove that our editor outperforms this state-of-the-art method in terms of accuracy and efficiency.

We choose the *GPT-Neo-1.3B-finetuned* as the base model and finetune on the APPS-train dataset, keeping the same experimental settings as CodeRanker for a fair comparison. Our method (*"+ editor"*) significantly outperforms CodeRanker (*"+ ranker"*). In particular, on APPS-test, our method can improve pass@1 from 0.14% to 0.68%, while their method can only improve from 0.14% to 0.3%. It means our method can solve 19 more problems on this challenging dataset. We also provide the performance of other reproduced base models in Table 9, where our method generally outperforms.

More importantly, existing post-processing

Code Gen. Model	pass@1	edit pass@1	pass@5	edit pass@5	pass@10	edit pass@10	sol@10	edit sol@10
finetuned on APPS								
PyCodeGPT	6.10	8.54	7.32	10.98	7.93	13.41	100	159
GPT-Neo 125M	0.61	3.05	3.05	7.32	6.10	9.76	21	76
CodeGen-350M	6.10	7.93	7.32	9.15	7.32	10.37	100	140
GPT-Neo 1.3B	2.44	5.49	8.54	10.98	11.59	14.63	66	132
Incoder-1B	6.71	10.37	8.54	13.41	9.76	14.63	112	169
GPT-J	7.32	9.76	17.07	19.51	25.00	25.61	133	183
zero-shot								
Incoder-1B	1.22	3.66	2.44	7.93	5.49	10.98	13	87
CodeGen-2B	14.02	17.07	29.27	29.88	34.15	34.15	226	255

Table 4: Results on the HumanEval dataset.

Difficulty level	pass@1	pass@5	pass@10
Introductory	2.10	7.40	10.10
	4.90 133%	10.40 40.5%	14.20 40.6%
Interview	0.43	1.53	2.37
	0.67 53.5%	1.97 28.1%	3.03 28.3%
Competition	0.10	0.30	0.40
	0.10	0.40 33.3%	0.50 25.0%
Average	0.70	2.46	3.52
	1.40 100%	3.34 35.8%	4.76 35.2%

Table 5: Results on the APPS-test dataset with 3 difficulty levels. We use the *GPTJ-6B-finetuned* as the base model. We show the base model results (the first row) and edited results (shaded row below). The numbers in red indicate the improvements of our editor.

		APPS-dev		APPS-test	
Setting	Samples	@1	@5	@1	@5
base model		4.0	10.9	0.14	0.74
+ ranker†	100	8.0	15.1	0.3	1.1
+ editor	{1,5}	**10.5**	**18.6**	**0.68**	**1.38**

† The results are copied from the original paper.

Table 6: Pass Rate Results compared with CodeRanker on the APPS dataset. *"+ ranker"* numbers are cited from Inala et al. (2022). We use the *GPT-Neo-1.3B-finetuned* as the base model. Our method outperforms CodeRanker with an extremely small sample budget.

	APPS-dev			APPS-test		
Setting	@1	@5	@10	@1	@5	@10
base model	4.8	7.9	8.9	0.2	0.4	0.4
after edit	11.4	**15.1**	**17.1**	**0.6**	**1.0**	**1.2**
- comments	9.4	11.5	13.5	0.3	0.3	0.4
+ edit round	**11.7**	**15.2**	**17.1**	0.4	0.7	0.9

Table 7: Pass Rate Results of ablation studies. We use the *PyCodeGPT-110M-finetuned* as the base model. The column *"after edit"* means the performance of our editor in original settings. We experiment with additional editing rounds or without supplemental comment.

methods rely on sampling many outputs from LLMs. For instance, the CodeRanker requires 100 outputs for each problem and then selects k samples with their ranker model to evaluate *pass@k* metric. In contrast, our method only requires $k = \{1, 5\}$ outputs per problem and then utilizes these outputs to generate a final solution through editing. Our approach is more efficient and effective, especially when obtaining outputs from large language models is costly. As a result, our method has greater practical significance and is more suitable for use with limited sample budgets.

4.4 Ablation on Supplementary Comments

To investigate the influence of supplementary comments, we remove the supplementary comments from the editor input and only use problem description and generated code to train a new editor. Other settings are kept the same. Results on APPS validation and test datasets are shown in Table 7.

We find that the pass rate of the modified editor decreases significantly on both datasets compared with the original editor. The modified editor can improve the APPS-dev dataset compared to the base model. However, on the more difficult APPS-test dataset, the editor model without comments shows no performance improvement. The results indicate that losing the guidance of the supplementary comment will hurt the performance of the editor model. Our experiments show that using error messages as supplementary comments for the code editor is crucial for achieving remarkable performances.

4.5 Ablation on the Number of Edit Rounds

In our self-edit approach, we make edits to the output of LLMs to produce the final program. It leads to a question: what if we make additional edits to the program after the first edit? We add an additional editing step to answer this question using our original editor. Concretely, the edited program is executed on an example test case to obtain comments and then refined by the editor model again. The results of this approach are presented in Table 7, with the column labeled *"+ edit round"* indicating the two-round editing approach.

The results show the two-round editing leads to a slight increase in pass@1 on APPS-dev. However, the additional edit round hurts the performance on APPS-test. We guess the reason is the gap between training and test time in the second editing round. The editor is trained to edit LLM outputs but used to edit its own output in the second edit round. In this setting, an additional editing round is not very helpful in generating better programs.

5 Discussion

5.1 Time Cost compared with Post-processing Baseline

For the specific issue of time cost, we use *Google Colab* [3] with a Tesla T4 GPU to build a demo and conduct evaluations over APPS-test dataset. We use *text-davinci-002* as the base model and the average time cost is nearly 8.4s to obtain 1 sample for each question. The executor costs <0.01s, and our editor costs 3.7s to get the final output, which is acceptable in our actual experience using the demo. By contrast, the state-of-the-art reranking method CodeRanker requires >110s to obtain candidate lists and 0.53s for the following ranker. As a result, our framework achieves better performance with less total time cost and fewer LLM calls.

5.2 Performances of In-Context Learning Self-Edit

Given that LLMs have demonstrated strong in-context learning abilities without requiring any specific training, we leverage the capabilities of the *text-davinci-002* model as both the base and editor models to develop a variant of our self-edit method that utilizes in-context learning. Specifically, we utilize in-context learning abilities of the model to self-edit its output using the supplementary comments we construct (detailed in Section 3.2) as input prompts for zero-shot inference. This approach allows the large model to edit its output program without additional training, offering a promising solution for optimizing the potential of LLMs.

[3] https://colab.research.google.com

Benchmark		pass@1	pass@5	sol@5
APPS-test	before	7.48	15.94	1876
	after	**8.94**	**17.12**	**2214**
HumanEval	before	34.76	60.98	288
	after	**39.63**	**64.63**	**331**

Table 8: Results of the in-context learning self-edit on APPS-test and HumanEval benchmarks. We use the *text-davinci-002* as the base model and editor model. We use the in-context learning ability of *GPT3* to self-edit the model output. The constructed supplementary comments are used as input prompts for the editor. We show the base model results (the first row) and edited results (shaded row below).

Our experiments on APPS-test and HumanEval are presented in Table 8. Results demonstrate that our self-edit framework can be extended using in-context learning, achieving significantly better performance than smaller editors across various benchmarks. However, it is important to note that this in-context learning self-edit method still incurs a relatively large number of LLM calls. Therefore, optimizing resource requirements while exploiting the potential of LLMs remains critical. To this end, we will explore strategies to efficiently utilize the in-context learning capabilities of LLMs in our self-edit framework in future work.

6 Conclusion

We propose a generate-and-edit approach named Self-Edit that utilizes execution results of the generated code from LLMs to improve the code quality on the competitive programming task. The central component of our approach is the fault-aware code editor, which can edit and optimize the generated code. In-depth evaluations demonstrate our approach significantly improves the quality of LLMs' output code.

7 Acknowledgement

This research is supported by the National Natural Science Foundation of China under Grant Nos. 62072007, 62192731, 62192733, 62192730, 61832009. The AI training platform supporting this work were provided by High-Flyer AI. (Hangzhou High-Flyer AI Fundamental Research Co., Ltd.) We also would like to thank all the anonymous reviewers for constructive comments and suggestions to this paper.

Limitations

Our work has several limitations, which we aim to address in our future work:

Firstly, we implement our editor with relatively small pretrained models within our computational capabilities. Our in-depth evaluations have preliminarily demostrated the effectiveness of the generate-and-edit approach. We hope to further understand the performance when using different pretrained models and architectures for the editor.

Secondly, the editor datasets we constructed are relatively small due to our computational capabilities. In our experiment, we only sample 10 programs from the LLM for each problem for dataset construction. Compared with existing post-editing methods, the dataset we use is quite small. It would be meaningful to do a detailed analysis of the impact of editor dataset size, or to experiment with other dataset construction methods. We leave this as future work.

Thirdly, We do not have strict comparison about computing resources with other post-editing methods. In Section 4.3 we compare with a state-of-the-art re-reaking baseline. We both use an additional model with a similar amount of parameters, but our approach outperforms using very few samples from LLMs. As accessing LLMs is costing, our approach demonstrates both superior accuracy and efficiency.

Finally, in our ablation study on the number of edit rounds, we faced with a gap between training and test time in the second editing round. Our existing implementation is not designed for this multiple-round editor. We hope to further try new specially designed model to implement the editor model. As large language models continue to advance, the need for effective strategies to interact with LLMs will be an important area of future research.

References

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Shuvendu K. Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2022. Interactive code generation via test-driven user-intent formalization. *CoRR*, abs/2208.05950.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Enabling programming thinking in large language models toward code generation. *arXiv preprint arXiv:2305.06599*.

Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2022a. Codeeditor: Learning to edit source code with pre-trained models. *arXiv preprint arXiv:2210.17040*.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023b. Skcoder: A sketch-based approach for automatic code generation. *arXiv preprint arXiv:2302.06144*.

Jia Li, Chongyang Tao, Huang Hu, Can Xu, Yining Chen, and Daxin Jiang. 2022b. Unsupervised cross-domain adaptation for response selection using self-supervised and adversarial training. In *WSDM '22: The Fifteenth ACM International Conference on Web Search and Data Mining, Virtual Event / Tempe, AZ, USA, February 21 - 25, 2022*, pages 562–570. ACM.

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023c. Towards enhancing in-context learning for code generation. *arXiv preprint arXiv:2303.17780*.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022c. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *CoRR*, abs/2203.13474.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE.

Richard Yuanzhe Pang and He He. 2021. Text generation by learning from demonstrations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Jianhao Shen, Yichun Yin, Lin Li, Lifeng Shang, Xin Jiang, Ming Zhang, and Qun Liu. 2021. Generate & rank: A multi-task framework for math word problems. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2269–2279. Association for Computational Linguistics.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. *CoRR*, abs/2204.11454.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, pages 1–10. ACM.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720*.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual pre-training on sketches for library-oriented code generation. In *The 2022 International Joint Conference on Artificial Intelligence*.

Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023. Toolcoder: Teach code generation models to use api search tools. *ArXiv*, abs/2305.04032.

Tianyi Zhang, Tao Yu, Tatsunori B. Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I. Wang. 2022. Coder reviewer reranking for code generation. *CoRR*, abs/2211.16490.

A Compared with CodeRanker

We compare with CodeRanker (Inala et al., 2022) using *GPT-Neo-125M-finetuned*, *GPT-Neo-1.3B-finetuned* and *GPT-J-6B-finetuned* as the base model. For fair comparison, we choose the same base model, training dataset and test benchmark as the CodeRanker. We choose the above three base models and finetune on the APPS-train dataset to reproduce their results. The purpose of this step is to make our base model results similar to their reported base model results, so as to fairly compare the post-processing performance. In the experiments, the base model performance in our results is similar to the base model reported by CodeRanker. Full details of results are shown in Table 9. With a very small number of samples output by LLMs, our method significantly exceeds this state-of-the-art baseline.

B Qualitative analysis of Code Editor

In Figure 5 and 6 we show various programs generated by the *GPT3*, its corresponding problem description (contains example test case) and the supplementary comment. Our fault-aware code editor concatenates these as input, and generate the edited code as the final output. We find that the edited code is simialr to the *GPT3* output. In particular, the first few lines of the edited output are exactly the same as the output of *GPT3*, and the subsequent code is also partially based on the content in *GPT3* output. Through statistical analysis, we find that the common prefix between the two sequences accounted for 19.10% of the edited output on the APPS-dev and APPS-test datasets. While this does not account for similarities in the intermediate content, it is sufficient evidence to demonstrate the impact of the LLM output on the edited code. As for the HumanEval benchmark, we also show case studies in Figure 7.

GPT-Neo-125M-finetuned			APPS-dev		APPS-test	
	Setting	Samples	@1	@5	@1	@5
Reported in (Inala et al., 2022)	base model †		1.4	5.2	0.04	0.17
	+ ranker	100	6.5	**11.4**	0.1	0.5
Our results	base model		1.5	6.7	0.08	0.40
	+ editor	**{1,5}**	**8.5**	10.2	**0.22**	**0.70**
GPT-Neo-1.3B-finetuned			APPS-dev		APPS-test	
	Setting	Samples	@1	@5	@1	@5
Reported in (Inala et al., 2022)	base model †		2.6	9.1	0.14	0.53
	+ ranker	100	8.0	15.1	0.3	1.1
Our results	base model		4.0	10.9	0.14	0.74
	+ editor	**{1,5}**	**10.5**	**18.6**	**0.68**	**1.38**
GPT-J-6B-finetuned			APPS-dev		APPS-test	
	Setting	Samples	@1	@5	@1	@5
Reported in (Inala et al., 2022)	base model †		5.1	15.6	0.5	1.6
	+ ranker	100	11.0	21.7	0.8	2.6
Our results	base model		6.0	17.9	0.7	2.46
	+ editor	**{1,5}**	**12.0**	**27.8**	**1.4**	**3.34**

† As CodeRanker does not release the weights of base models, we cite their results from Inala et al. (2022) and reproduce finetuned base models shown in the *"Our results - base model"* row below.

Table 9: Full details of Pass Rate Results compared with the CodeRanker on the APPS dataset. We use *GPT-Neo-125M-finetuned*, *GPT-Neo-1.3B-finetune* and *GPT-J-6B-finetuned* as the base model.

Problem Description: *Question id: APPS-dev-305*

Return the number of distinct non-empty substrings of text that can be written as the concatenation of some string with itself (i.e. it can be written as a + a where a is some string).

Example 1:
Input: text = "abcabcabc"
Output: 3
Explanation: The 3 substrings are "abcabc", "bcabca" and "cabcab".

Example 2:
Input: text = "leetcodeleetcode"
Output: 2
Explanation: The 2 substrings are "ee" and "leetcodeleetcode".

Constraints:
1 <= text.length <= 2000
text has only lowercase English letters.

GPT3:

```
class Solution:
    def distinctEchoSubstrings(self, text: str) -> int:
        n = len(text)
        res = set()
        for i in range(n):
            for j in range(i+1, n+1):
                if text[i] == text[j]:
                    k = 1
                    while i+k < j and j+k < n and
text[i+k] == text[j+k]:
                        k += 1
                    if k == j-i:
                        res.add(text[i:j])
        return len(res)
```

Supplementary Comment:
Line 7, if text[i] == text[j]:
IndexError: string index out of range. Fix the bug.

After Edit:

```
class Solution:
    def distinctEchoSubstrings(self, text: str) -> int:
        n = len(text)
        res = set()
        for i in range(n):
            for j in range(i+1, n):
                if text[i:j] == text[j:j+j-i]:
                    res.add(text[i:j])
        return len(res)
```

(a)

Problem Description: *Question id: APPS-dev-4615*

Our AAA company is in need of some software to help with logistics: you will be given the width and height of a map, a list of x coordinates and a list of y coordinates of the supply points, starting to count from the top left corner of the map as 0.
Your goal is to return a two dimensional array/list with every item having the value of the distance of the square itself from the closest supply point expressed as a simple integer.
Quick examples:
```python
logistic_map(3,3,[0],[0])
#returns
#[[0,1,2],
# [1,2,3],
# [2,3,4]]
logistic_map(5,2,[0,4],[0,0])
#returns
#[[0,1,2,1,0],
# [1,2,3,2,1]]
```
Remember that our company is operating with trucks, not drones, so you can simply use Manhattan distance. If supply points are present, they are going to be within the boundaries of the map; if no supply point is present on the map, just return `None`/`nil`/`null` in every cell.
```python
logistic_map(2,2,[],[])
#returns
#[[None,None],
# [None,None]]
```
Note: this one is taken (and a bit complicated) from a problem a real world AAA company [whose name I won't tell here] used in their interview. It was done by a friend of mine. It is nothing that difficult and I assume it is their own version of the FizzBuzz problem, but consider candidates were given about 30 mins to solve it.

GPT3:

```
def logistic_map(width, height, x, y):
    return [[min(abs(x - i) + abs(y - j) for x, y in zip(x, y))
for i in range(width)] for j in range(height)]
```

Supplementary Comment:
Line 2, return [[min(abs(x - i) + abs(y - j) for x, y in zip(x, y)) for i in range(width)] for j in range(height)]
ValueError: min() arg is an empty sequence. Fix the bug.

After Edit:

```
def logistic_map(width, height, supply_x, supply_y):
    if not supply_x or not supply_y:
        return [[None] * width for _ in range(height)]
    return [[min(abs(x - sx) + abs(y - sy) for sx, sy in
zip(supply_x, supply_y)) for x in range(width)] for y in
range(height)]
```

(b)

Figure 5: Case Study on APPS-dev dataset using *GPT3* model.

Problem Description: *Question id: APPS-test-2629*

Given a positive integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.

Example:

Input: 3
Output:
[
[1, 2, 3],
[8, 9, 4],
[7, 6, 5]
]

GPT3:

```
class Solution:
    def generateMatrix(self, n):
        if n == 0:
            return []
        if n == 1:
            return [[1]]

        res = [[0 for i in range(n)] for j in range(n)]
        i = 0
        j = 0
        count = 1
        while i < n and j < n:
            for k in range(j, n):
                res[i][k] = count
                count += 1
            i += 1
            for k in range(i, n):
                res[k][n - 1] = count
                count += 1
            n -= 1
        return res
```

Supplementary Comment:
Wrong Answer with input 3. Expected output is [[1, 2, 3], [8, 9, 4], [7, 6, 5]], but generated output is [[1, 2, 3], [6, 7, 4], [0, 0, 5]]. Rewrite the code.

After Edit:

```
class Solution:
    def generateMatrix(self, n):
        if n == 0:
            return []
        res = [[0 for i in range(n)] for j in range(n)]
        i, j, di, dj = 0, 0, 0, 1
        for k in range(1, n * n + 1):
            res[i][j] = k
            if res[(i + di) % n][(j + dj) % n]:
                di, dj = dj, -di
            i += di
            j += dj
        return res
```

(a)

Problem Description: *Question id: APPS-test-4854*

Mirko is a great code breaker. He intercepted an enemy message. The message consists of \$N\$ numbers, smaller than or equal to \$C\$. Mirko belives freqency analysis consists of sorting this sequence so that more frequent numbers appear before less frequent ones.

Formally, the sequence must be sorted so that given any two numbers \$X\$ and \$Y\$, \$X\$ appears before \$Y\$ if the number of times \$X\$ appears in the original sequence is larger than the number of time \$Y\$ does. If the number of appearances is equal, the number whose value appears sooner in the input should appear sooner in the sorted sequence.

Help Mirko by creating a "frequency sorter".

-----Input-----
First line of input contains two integers, \$N\$ (\$1 \le N \le 1000\$), the length of the message, and \$C\$ (\$1 \le C \le 1000000000\$), the number from the task description above.

The next line contains \$N\$ positive integers smaller than or equal to \$C\$, the message itself.

-----Output-----
The first and only line of output should contain \$N\$ numbers, the sorted sequence.

-----Examples-----
Sample Input 1:
5 2
2 1 2 1 2
Sample Output 1:
2 2 2 1 1

Sample Input 2:
9 3
1 3 3 3 2 2 2 1 1
Sample Output 2:
1 1 1 3 3 3 2 2 2

GPT3:

```
n, c = map(int, input().split())
arr = list(map(int, input().split()))
d = {}
for i in arr:
    if i not in d:
        d[i] = 1
    else:
        d[i] += 1
for key, value in sorted(d.items(), key=lambda item: (item[1], -
key)):
    print((key) * value, end=" ")
```

Supplementary Comment:
Line 9, for key, value in sorted(d.items(), key=lambda item: (item[1], -key)):
NameError: name 'key' is not defined. Fix the bug.

After Edit:

```
n, c = map(int, input().split())
a = list(map(int, input().split()))
cnt = Counter(a)
for i in range(n):
    for j in range(i+1, n):
        if cnt[a[i]] < cnt[a[j]]:
            a[i], a[j] = a[j], a[i]
        elif cnt[a[i]] == cnt[a[j]]:
            if a.index(a[i]) > a.index(a[j]):
                a[i], a[j] = a[j], a[i]
print(*a)
```

(b)

Figure 6: Case Study on APPS-test dataset using *GPT3* model.

Problem Description: *Question id: HumanEval-116*

```
def sort_array(arr):
  """
  In this Kata, you have to sort an array of non-negative integers according to
  number of ones in their binary representation in ascending order.
  For similar number of ones, sort based on decimal value.

  It must be implemented like this:
  >>> sort_array([1, 5, 2, 3, 4]) == [1, 2, 3, 4, 5]
  >>> sort_array([-2, -3, -4, -5, -6]) == [-6, -5, -4, -3, -2]
  >>> sort_array([1, 0, 2, 3, 4]) [0, 1, 2, 3, 4]
  """
```

CodeGen-2B:

```
def sort_array(arr):
    return sorted(arr, key = lambda x: count_ones(x))
```

Supplementary Comment:
Line 1, return sorted(arr, key = lambda x: count_ones(x))
NameError: name 'count_ones' is not defined. Fix the bug.

After Edit:

```
def sort_array(arr):
    return sorted(arr, key=lambda x: (bin(x).count('1'),
x))
```

(a)

Problem Description: *Question id: HumanEval-157*

```
def right_angle_triangle(a, b, c):
  '''
  Given the lengths of the three sides of a triangle. Return True if the three
  sides form a right-angled triangle, False otherwise.
  A right-angled triangle is a triangle in which one angle is right angle or
  90 degree.
  Example:
  right_angle_triangle(3, 4, 5) == True
  right_angle_triangle(1, 2, 3) == False
  '''
```

CodeGen-2B:

```
def right_angle_triangle(a, b, c):
    if (a**2 + b**2) == c**2 or (b**2 + c**2) == a**2 or
(a**2 + c**2) == b**2:
        print("Right angled triangle")
    else:
        print("Not Right angled triangle")
```

Supplementary Comment:
Wrong Answer with input 3, 4, 5. Expected output is True, but generated output is None. Rewrite the code.

After Edit:

```
def right_angle_triangle(a, b, c):
    if a == 0 or b == 0 or c == 0:
        return False
    else:
        if (a**2 + b**2) == c**2 or (b**2 + c**2) == a**2
or (a**2 + c**2) == b**2:
            return True
        else:
            return False
```

(b)

Figure 7: Case Study on HumanEval dataset using *CodeGen-2B* model.

ACL 2023 Responsible NLP Checklist

A For every submission:

☑ A1. Did you describe the limitations of your work?
Yes, we include a section named "Limitations", following the "Conclusion" section.

☑ A2. Did you discuss any potential risks of your work?
Yes, we include a detailed discussion in "Limitations" section. We also discuss the baseline risks in Appendix Section A.

☑ A3. Do the abstract and introduction summarize the paper's main claims?
Yes, we summarize our paper in Section 1."Introduction".

☒ A4. Have you used AI writing assistants when working on this paper?
No, we only use "grammarly" to polish our paper.

B ☑ Did you use or create scientific artifacts?

Yes, we use open-source.large language models in Section 4.1

☑ B1. Did you cite the creators of artifacts you used?
Yes, we cite the open-source.large language models in Section 4.1

☑ B2. Did you discuss the license or terms for use and / or distribution of any artifacts?
Yes, the models we use are open-source and free for education in Section 4.1

☑ B3. Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions (in particular, derivatives of data accessed for research purposes should not be used outside of research contexts)?
Yes, the models we use are open-source and free for education in Section 4.1

☒ B4. Did you discuss the steps taken to check whether the data that was collected / used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect / anonymize it?
We use the open-source popular datasets that do not contains any information that names or uniquely identifies individual people or offensive content.

☑ B5. Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.?
We describe the specific domain we focus on in Section 2.1

☑ B6. Did you report relevant statistics like the number of examples, details of train / test / dev splits, etc. for the data that you used / created? Even for commonly-used benchmark datasets, include the number of examples in train / validation / test splits, as these provide necessary context for a reader to understand experimental results. For example, small differences in accuracy on large test sets may be significant, while on small test sets they may not be.
We report in Table 1.

C ☑ Did you run computational experiments?

Yes, we provide a experiment about samples from LLMs in Section 4.3 and Appendix Section A.

☑ C1. Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?
Yes, we provide a detailed description in Table 2 and Section 4.1

The Responsible NLP Checklist used at ACL 2023 is adopted from NAACL 2022, with the addition of a question on AI writing assistance.

☑ C2. Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?
Yes, we provide a detailed description in Section 4.1

☑ C3. Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run?
Yes, we adopt our experiment with 3 times and summary statistics are described in Section 4.1 and Table 1.

☑ C4. If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation), did you report the implementation, model, and parameter settings used (e.g., NLTK, Spacy, ROUGE, etc.)?
Yes, we list the settings in Table 2. Our comparision with baseline is in a fair setting, described in Section 4.3 and Appendix Section A.

D ☒ Did you use human annotators (e.g., crowdworkers) or research with human participants?

Our evaluation is based on the execution results of the generated code with pass@k metrics. Thus it is a reliable indicator and we do not need evaluation with human annotations.

☐ D1. Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.?
Not applicable. Left blank.

☐ D2. Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)?
Not applicable. Left blank.

☐ D3. Did you discuss whether and how consent was obtained from people whose data you're using/curating? For example, if you collected data via crowdsourcing, did your instructions to crowdworkers explain how the data would be used?
Not applicable. Left blank.

☐ D4. Was the data collection protocol approved (or determined exempt) by an ethics review board?
Not applicable. Left blank.

☐ D5. Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data?
Not applicable. Left blank.