Enhancing Code Generation Performance of Smaller Models by Distilling the Reasoning Ability of LLMs

Zhihong Sun[1], **Chen Lyu**[1*†], **Bolun Li**[1], **Yao Wan**[2]
Hongyu Zhang[3], **Ge Li**[4], **Zhi Jin**[4*]
[1]School of Information Science and Engineering, Shandong Normal University, China
[2]Huazhong University of Science and Technology, China [3]Chongqing University, China
[4]Key Lab of HCST (PKU), MOE; SCS, Peking University, China
2022021002@stu.sdnu.edu.cn, lvchen@sdnu.edu.cn, libolun118@gmail.com
wanyao@hust.edu.cn, hyzhang@cqu.edu.cn
{lige, zhijin}@pku.edu.cn

Abstract

Large Language Models (LLMs) have recently made significant advances in code generation through the 'Chain-of-Thought' prompting technique. This technique empowers the model to autonomously devise "solution plans" to tackle intricate programming challenges, thereby improving its performance in code generation. Nevertheless, smaller models have been struggling to keep up with LLMs in deducing these plans, adversely affecting their code generation capabilities. Given the considerable size and associated deployment costs, along with concerns about data security, many teams opt for deploying smaller models for code generation. Consequently, there arises a compelling need for transferring LLMs' code generation reasoning abilities to the smaller models. In this paper, we propose the CodePLAN framework, which aims to transfer LLMs' reasoning capabilities to smaller models through distillation. We adopt a multi-task learning approach, jointly undertaking code generation and solution plan generation tasks, to enhance the code generation capabilities of the smaller model. To ensure the superior quality of the solution plans, we advocate for the utilization of backward reasoning and plan sampling strategies. Our experiments show that in comparison to the conventional fine-tuning approach, our approach improves the smaller model's code generation performance (measured in pass@1 metric) by over 130% on the challenging APPS benchmark.

1. Introduction

Automatic code generation has a history spanning decades, aiming to create executable programs from problem specifications (Backus et al., 1957; Waldinger and Lee, 1969; Manna and Waldinger, 1971). As artificial intelligence technology rapidly advances, the application of neural network techniques in intelligent code generation is increasingly gaining attention in the field of software engineering (Ling et al., 2016; Yin and Neubig, 2018; Lyu et al., 2021). Recently, large language models (LLMs) such as ChatGPT (OpenAI, 2022) have made significant advances in code generation owing to their superior reasoning capabilities. However, deploying these mammoth models comes with significant computational, time, and financial demands, coupled with data and security risks. Consequently, many enterprises and teams still prefer more manageable, smaller models.

In the realm of code generation, smaller models lag in reasoning capabilities compared to LLMs, leading to challenges with complex programming tasks. Our empirical studies highlight the exceptional in-context learning (ICL) of LLMs. By utilizing "Chain-of-Thought (CoT)" (Wei et al., 2022) as human-defined solution steps, LLMs can bolster their reasoning, allowing them to craft solution plans from these in-context examples. This methodology elevates LLMs' problem-solving accuracy and is notably effective in code generation (Jiang et al., 2023; Huang et al., 2024). However, while CoT strategies shine with massive-parameter models, smaller models, even after fine-tuning, struggle in deriving CoT-based solution plans due to ICL and reasoning constraints. Yet, we have observed that a smaller model, around 1B parameters in size, when fine-tuned and given both problem description and a precise CoT-based solution plan (labeled as "best plan"), sees a substantial boost in code generation capabilities, as shown in Figure 1.

This finding prompted us to further explore strategies for providing smaller models with a "best plan" when addressing programming tasks. However, we face two significant challenges: **1) Dependency on Large Language Models (LLMs) during inference.** Utilizing LLMs to generate solution plans for smaller models might be pragmatic, but it becomes impractical if the smaller model consistently relies on LLMs during inference. **2) Securing accurate, high-quality solution plans.** Solution plans are primarily procured either manually by experts or automatically by LLMs. While expert-curated plans

*Zhi Jin and Chen Lyu are the corresponding authors.

†This work was done when Chen Lyu was a visiting scholar at Peking University.

LREC-COLING 2024, pages 5878–5895
20-25 May, 2024.

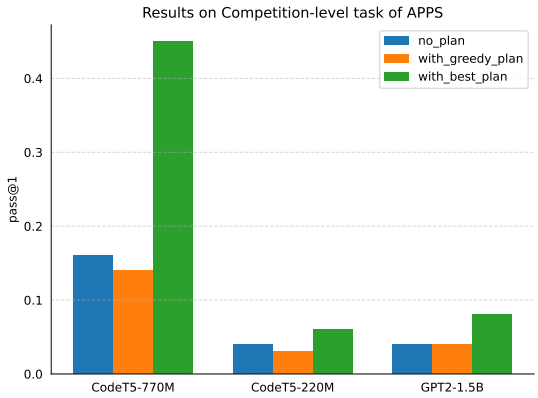

Figure 1: Comparison results of different models without solution plans, spliced LLM greedy generated solution plans and the best quality solution plans as prompt, where all models were fine-tuned on the APPS train dataset.

are typically more reliable, their high costs make them less feasible for automated code generation. In contrast, most LLM-generated plans, directly derived from problem descriptions, often do not meet the desired quality. Notably, even state-of-the-art LLMs like ChatGPT can produce plans that are not consistently accurate, leading not only to missed enhancements but potential performance regressions in smaller models, as depicted in Figure 1.

To address these challenges, we introduce “CodePLAN”, a novel multi-task plan-based framework designed to enhance the code generation for smaller models by distilling LLMs’ reasoning ability. Essentially, CodePLAN utilizes multi-task learning to imbue smaller models with LLMs’ reasoning capabilities, allowing them to autonomously develop solution plans and generate code. Central to CodePLAN’s effectiveness is the precision of these solution plans, both in training and inference. Thus, we innovate two techniques: "back reasoning" and "plan sampling", which respectively enhance the quality of plans during LLM distillation and during CodePLAN’s own inference.

Specifically, to tackle the first challenge, we conceptualize LLMs as "teachers" and smaller models as "students", with the objective of distilling the teacher’s reasoning capabilities into the student. We employ a multi-task training framework, using solution plans from LLMs and actual codes as supervisory signals. This framework emphasizes two tasks: 1) Code generation, which develops the smaller model’s coding skills, and 2) Plan generation, aiming to distill LLMs’ reasoning prowess. Leveraging this strategy, the smaller model’s code generation performance improves notably. While it leans on LLMs during training, it operates autonomously during inference. At this stage, capitalizing on its refined skill to generate solution plans, the model uses its plans to enhance the code generation process, optimizing its output potential.

For the second challenge, we guide LLMs to create solution plans based on actual codes using a "back reasoning" approach. This deviates from methods by Jiang et al. (2023) and Li et al. (2023c), who rely solely on problem descriptions. Our method prioritizes obtaining top-tier solution plans (refer to section 3.1), a claim supported by our empirical data (see Table 5). Nonetheless, smaller models still confront a similar obstacle during the inference phase - the inability to directly generate correct solution plans. To address this problem, we take inspiration from the process that programmers use to solve complex programming problems. The process of programmers in solving complex programming problems is actually a process of continuous trial and error of thinking, where the correctness of thinking has been verified by writing code according to the constructed thinking until the problem is solved. Consequently, in the inference phase, we introduce a technique called "plan sampling" to simulate a programmer’s problem-solving. The key was to ensure efficiency while targeting quality solutions. To this end, we craft a strategy using limited sampling and concise unit tests for each sampled solution plan. This makes "plan sampling" both lightweight and highly effective.

We executed a comprehensive series of experiments on two distinct streamed code generation datasets, namely APPS and MBPP. Our novel approach, in comparison to standard finetune methods, considerably enhances the code generation proficiency of the model, most notably improving the pass@1 metric on the APPS dataset by over 130%. To the best of our knowledge, this study is the first exploration of distilling the reasoning ability of LLMs to improve code generation in smaller models. Our codebase is publicly accessible at: https://github.com/sssszh/CodePLAN.

2. Related Work

Code Generation. With the advent of transformer (Vaswani et al., 2017) and the development of pre-training techniques (Devlin et al., 2018), more and more pre-training models are applied in the field of code generation. For instance, open-source code models like CodeT5 (Wang et al., 2021), CodeT5+ (Wang et al., 2023), CodeGen (Nijkamp et al., 2022), PolyCoder (Xu et al., 2022), InCoder (Fried et al., 2022), StarCoder (Li et al., 2023a), as well as general-purpose language models such as GPT-J (Wang and Komatsuzaki, 2021), GPT-Neo (Black et al., 2021) have demonstrated substantial performance in code generation tasks.

The dominant approaches in code generation mainly involve fine-tuning pre-trained code genera-

tion models using supervised learning (Hendrycks et al., 2021) or reinforcement learning (RL) (Li et al., 2022; Le et al., 2022; Shojaee et al., 2023; Li et al., 2024). However, neither supervised nor reinforcement learning fine-tuning allows the model to learn reasoning well. Moreover, RL-based approaches decompose code generation into sequences of token-generating actions, which may limit the model to learn reasoning ability due to the lack of high-level thinking. Different from these methods, we achieve high-level thinking in smaller models by distilling the reasoning abilities of LLMs into them.

Chain-of-Thought (CoT). With the advent of large language models, such as ChatGPT (OpenAI, 2022) and GPT4 (OpenAI, 2023), and the evolution of CoT prompting techniques (Wei et al., 2022; Wang et al., 2022), an increasing number of researchers have committed themselves to identify strategies that effectively augment the emergent capabilities of LLMs (Shum et al., 2023; Zhou et al., 2022). Jiang et al. (2023) proposed a "self-plan" approach, leveraging the inherent reasoning abilities of LLMs to sequentially decompose and solve problems. This methodology has yielded promising results for fundamental programming tasks. However, these CoT prompt-based techniques are predominantly applicable to models with many parameters (e.g., 100 B or more) and are less suitable for models with fewer parameters, which lack inferential solid interpretation abilities to decompose complex problems independently. Ho et al. (2022) and Hsieh et al. (2023) employed a novel strategy of using inference interpretations generated by LLMs as supervised signals to train smaller models, with the aim of enhancing their performance on simple natural language processing (NLP) tasks.

In summary, the "self-plan" approach proposed by Jiang et al. (2023) relies heavily on the inherent reasoning ability of LLMs. Different from methods that stimulate the inherent reasoning abilities of LLMs themselves, our methodology utilizes solution plans generated by LLMs as supervised signals for training smaller models, distilling the reasoning ability of LLMs into small models, it can reduce the expensive cost of deploying LLMs. Previous studies (Ho et al., 2022; Hsieh et al., 2023) have leveraged the reasoned interpretation of LLMs to enhance the performance of smaller models on simple NLP tasks. However, unlike these simple NLP tasks, code generation is a much more complex task, and the difficulty of obtaining high-quality solution plans prevents these methods from being directly applied to the field of code generation.

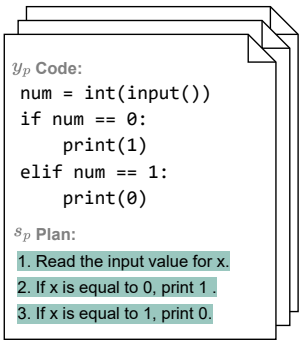

Figure 2: We use the prompt to allow LLM to reason backwards a solution plan from the code written by the programmer (highlighted in green).

3. CodePLAN

In this section, we provide a comprehensive exposition of the core principles underlying CodePlan. First, we outline how CodePlan extracts distilled knowledge from LLMs, with a specific emphasis on the key ingredient termed "solution plans". Following this, we demonstrate the detailed training process by which CodePlan leverages these "solution plans" within a multi-task learning framework. Lastly, we describe in-depth how CodePlan, during its inference phase, utilizes its self-generated "solution plans" to facilitate code generation.

3.1. Generating Plans through LLM

Contemporary research reveals that LLMs have the capacity to generate high-quality inference steps for certain rudimentary NLP tasks, thereby interpreting the solutions it produces (Wei et al., 2022). However, within the domain of code generation, LLM does not guarantee the generation of high-quality inference steps for intricate programming challenges. This necessitates the exploration of an effective methodology to uncover these high-quality solution plans.

Most LLM-based plan generation methods use a "forward reasoning" strategy, leveraging CoT examples to deduce plans from problem descriptions. However, our empirical studies for code generation tasks suggest that "backward reasoning" — deducing from the given solution/code — often produces higher-quality plans. To facilitate this, we establish a dataset D, comprised of paired elements (x_i, y_i), where x_i represents the problem description, and y_i represents the solution to x_i. For complex programming tasks, the LLM may not be able to generate high-quality solution plans directly from x_i. In contrast, as y_i represents the solution to x_i as authored by the programmer, it intrinsically encompasses the programmer's solution plan for addressing the problem. Consequently, we infer the solution plan s_i, written by the programmer when composing y_i to solve x_i, by reasoning backwards

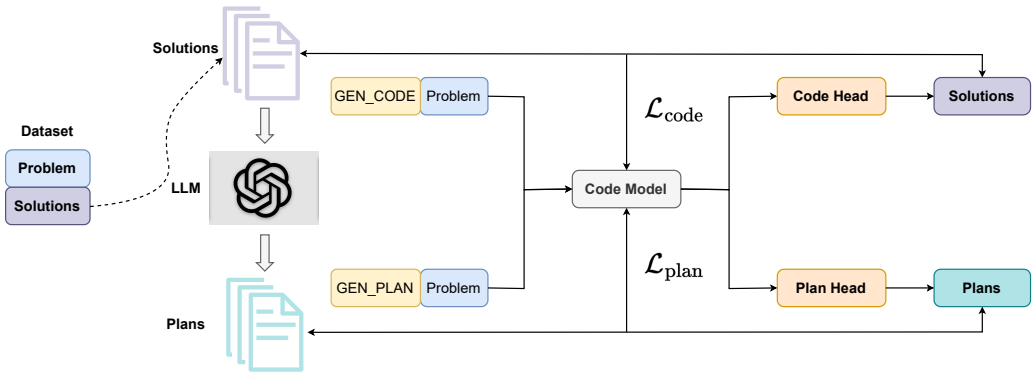

Figure 3: **Our framework for the training phase of CodePLAN:** backward reasoning from solutions via LLM about the programmer's solution plan at the time of solving this programming problem, and using these solution plans and solutions to fine-tune the code generation model in an alternating multi-task fashion.

from y_i. The solution plan we deduce backwards from y_i is typically superior to the solution plan obtained directly from x_i using the LLM (see section 4.5). We guide the LLM to generate solution plans based on provided prompt (y_p, s_p), utilizing the prompt template shown in Figure 2. For a new $y_i \in D$, the LLM emulates the prompt (y_p, s_p) to reason backwards a solution plan s_i for y_i.

3.2. Training Model with Plans

We initially outline the methodology for training the base model using the solution plans. In this procedure, we employ the intermediate solution plans, generated by LLMs, as a novel fine-tuning task assigned to the base model. The specifics of this training process are graphically depicted in Figure 3.

In conventional fine-tuning strategies, the base code generation model typically aims to minimize the cross-entropy loss between the generated code and the target code, serving as the primary training objective:

$$\mathcal{L}_{\text{code}}\ (\theta_1) = -\sum_t \log p_{\theta_1}(w_t \mid w_{1:t-1}, D) \qquad (1)$$

where D represents the problem description and $W = (w_1, ...w_t)$ represents the ground truth code.

Nonetheless, this conventional fine-tuning strategy fails to equip the model with inferential proficiency. To endow smaller models with the LLM's capability to decompose intricate problems, we add a training task -distilling reasoning ability from LLM - generating solution plans. This task is executed to minimize the cross-entropy loss between the solution plans generated by the model and those generated by the LLM:

$$\mathcal{L}_{\text{plan}}\ (\theta_2) = -\sum_t \log p_{\theta_2}(s_t \mid s_{1:t-1}, D) \qquad (2)$$

where D represents the problem description and $S = (s_1, ...s_t)$ represents the solution plan generated by LLM.

This approach not only equips the model with code generation capabilities but also enables it to generate intermediate solution plans. Within this training workflow, we utilize an alternating training strategy to fine-tune our model, distinguishing between the two tasks using two unique characters: $[GEN\_CODE]$ and $[GEN\_PLAN]$. Given the stark differences between program language and natural language, we modified our model by incorporating a new "plan head" at the end of the base model to generate solution plans. The total loss function optimized in our model is:

$$\mathcal{L} = (1-\lambda)\mathcal{L}_{\text{code}} + \lambda\mathcal{L}_{\text{plan}} \qquad (3)$$

where λ is a hyperparameter that regulates the weight assignment for the loss of the two tasks. In our experimental setup, λ=0.5.

3.3. Inferencing with Plans

Leveraging a multi-task fine-tuning approach, our framework enables the base model to generate both code and solution plans. In this context, we detail how, during the inference phase, the solution plans produced by the model enhance code generation, as illustrated in Figure 4.

Plan Sampling. As shown in Figure 4, the inference phase of CodePLAN is delineated into three specific stages. Initiated in the first stage, we use the fine-tuned model to generate the solution plans. The input of the model consists of the $[GEN\_PLAN]$ label and the problem description x_i, and its output is the solution plan s_i. However, the utilization of a greedy decoding strategy is insufficient to assure the precision of the solution plans. This deficiency prompted us to consider

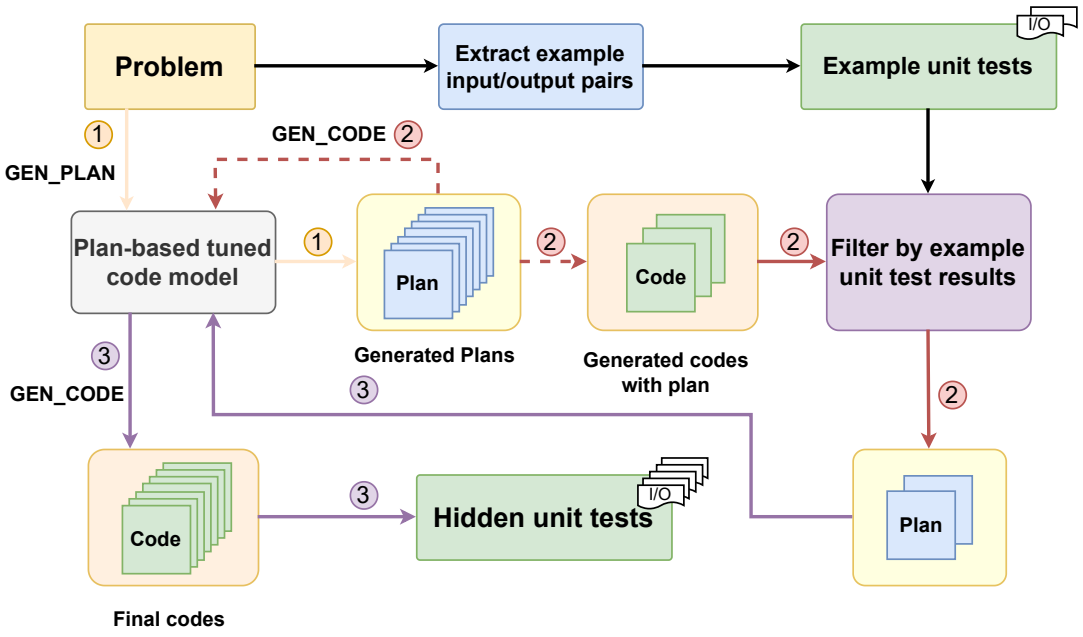

Figure 4: **The inference phase schematic comprises three stages:** ① Initially, the model formulates candidate solution plans based on the provided problem description. ② Subsequently, as indicated by the dashed line, solution plans generated in Stage ① are integrated with the problem description for code generation. Candidate solution plans are chosen based on the evaluation outcomes of the code generated through example unit tests. ③ Ultimately, the selected high-quality solution plan is used as a prompt, integrated within the problem description for a new cycle of code generation.

the methods by which programmers tackle complex competition problems: in such scenarios, a plethora of potential solution plans are conceived, with code being written and subsequently verified through unit tests.

Accordingly, we incorporated a novel strategy - "plan sampling" - to get the correct solution plans. This approach permits the sampling of multiple solution plans per problem, thereby encouraging the model to ideate akin to a programmer while acknowledging that complex programming problems may have multiple solutions. For the second stage, the model utilizes the $[GEN\_CODE]$ label, the problem description x_i, and the solution plan s_i as input, and then proceeds to generate a small volume of codes $y_i = \{y_{i_1}, y_{i_2}, ..., y_{i_n}\}$ in accordance with the solution plan, where n is limited to 10. This process can be formally defined as: $y_i \longleftarrow f(x_i, s_i)$, where f represents the base model. Considering time constraints, we sample up to 20 solution plans. The quality of a solution plan is indicated by the number of generated codes that successfully pass the example unit tests t_i — typically, only one or two as specified in the problem description. This evaluation metric is formalized as $score(s_i) = \sum_{y_{i_n} \in y_i} \delta(y_{i_n}, s_i, t_i)$. Here, $\delta(y_{i_n}, s_i, t_i)$ represents whether the code y_{i_n}, guided by the plan s_i, passes the example unit tests t_i, defined as: $\delta(y_{i_n}, s_i, t_i) := \begin{cases} 1, & \text{if } y_{i_n} \text{ passes } t_i \\ 0, & \text{otherwise} \end{cases}$. Consequently, the solution plan correlating with the highest number of successful code tests is deemed as the highest quality, formally captured by $s_i = argmax_{s_i \in S}(Score(s_i))$. Transitioning to the third stage, our framework elevates the model's code generation capability using the chosen top-tier solution plan. Codes generated within this enhanced framework are further assessed by hidden unit tests, which are more rigorous than the example unit tests, often capturing edge cases or extreme instances of the code's functionality.

4. Experiments

4.1. Experiment Setup

Dataset and Models. In this study, we evaluate our approach on two mainstream code generation datasets: (1) **APPS** (Hendrycks et al., 2021). (The dataset was collected from several programming competition platforms (e.g. Codeforces, LeteCode, etc.) with 10,000 problems, of which 5,000/5,000 problems were divided for training/testing and divided into three levels according to the difficulty of the problems, introductory level, interview level, and competition level. (2) **MBPP** (Austin et al., 2021). The dataset con-

Model	Size	*Pass@1*				*Pass@5*				*Pass@100*			
		Intro	Inter	Comp	All	Intro	Inter	Comp	All	Intro	Inter	Comp	All
Codex	12B	4.14	0.14	0.02	0.92	9.65	0.51	0.09	2.25	-	-	-	-
GPT2	1.5B	1.30	0.70	0.00	0.68	3.60	1.03	0.00	1.34	-	-	-	-
GPT-Neo	2.7B	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58	-	-	-	-
GPT-J	6B	5.60	1.00	0.50	1.82	9.20	1.73	1.00	3.08	-	-	-	-
StarCoder	164M	1.73	0.44	0.01	0.63	4.70	1.43	0.46	1.89	14.80	5.50	3.80	7.02
CodeGen	350M	1.54	0.38	0.08	0.56	4.91	1.26	0.37	1.82	17.40	5.51	4.10	7.62
CodeT5	220M	0.71	0.27	0.03	0.31	2.40	0.94	0.12	1.07	9.90	3.53	1.60	4.42
CodeT5+	770M	4.41	0.99	0.26	1.53	9.92	2.59	1.05	3.75	23.50	8.33	6.50	11.00
CodeT5	770M	3.30	0.68	0.15	1.10	8.12	1.89	0.74	2.91	21.30	6.53	6.00	9.38
CodeT5+CodePLAN	770M	**7.87**	**1.61**	**0.42**	**2.62**	**14.66**	**3.54**	**1.59**	**5.37**	**28.60**	**9.17**	**8.60**	**12.94**
Relative Improvement		138.5%	136.8%	162.5%	138.2%	80.5%	87.3%	114.9%	84.5%	34.3%	40.4%	43.3%	38.0%

Table 1: Performance by Pass@k on APPS: “Intro”: introductory, “Inter”: interview, “Comp”: competition.

sists of 974 programming problems constructed from crowdsourcing, with 374/90/500 problems divided for training/validation/testing and 10 reserved for few-shot prompt learning. We choose two of the most popular code generation models, CodeT5-770M (Wang et al., 2021) and CodeGen-350M (Nijkamp et al., 2022), to validate the effectiveness of our approach. And the solution plans we use for training are from OpenAI’s GPT-3.5-Turbo API (OpenAI, 2022).

Metric. To evaluate the functional correctness of generated codes, we followed the previous works (Hendrycks et al., 2021; Chen et al., 2021) using pass@k as the evaluation metric. This metric measures the functional correctness of the code by executing unit test cases. For each problem sampled to generate n>=k copies of code, the number of correct codes c<=n, pass@k metric is calculated as follows:

$$\text{pass}@k = \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (4)$$

In our experimental setup, we sample 100 copies of the code for each problem to compute pass@$\{1, 5, 100\}$

Training/Inference Setting. For the training phase associated with the APPS dataset, we adhered to the data preprocessing structure as delineated in the original paper (Hendrycks et al., 2021). The established maximum lengths for the source sequence and the target sequence were 600 and 512, respectively. The batch size was configured to 32, and the learning rate was specified at 2e-5, a learning rate decay of 0.05. The fine-tuning process was executed 10 epochs. When approaching the MBPP dataset, we remained consistent with the data preprocessing methodology laid out in the original paper (Austin et al., 2021). The respective maximum lengths for the source sequence and the target sequence were set at 350 and 300. Both the batch size and the learning rate mirrored the parameters established for the APPS dataset. Importantly, we implemented a total of 50 rounds of fine-tuning for the MBPP, to account for the more limited number of training sets within this dataset. In the inference stage, we employed temperature sampling for both APPS and MBPP, with respective temperature settings of 0.6 and 1.2. For each problem, we stipulated the generation of 100 instances of the code and 20 solution plans.

4.2. Experimental Results on APPS

We evaluated our models and compared them with several baseline models, which include GPT-2 (Radford et al., 2019), GPT-Neo (Black et al., 2021), GPT-3 (Brown et al., 2020), CodeX (Chen et al., 2021), CodeT5 (Wang et al., 2021), CodeT5+ (Wang et al., 2023), StarCoder (Li et al., 2023a) and CodeGen (Nijkamp et al., 2022). Note that all models except CodeX and GPT-3 are fine-tuned on APPS. As illustrated in Table 1, CodePLAN notably bolsters the code generation competency of the model, surpassing models equipped with several folds the number of parameters. Specifically, across all levels of the APPS benchmark, CodePLAN secures an impressive gain of over 130% in the pass@1, as compared to the standard fine-tuning process. Moreover, our method manifests considerable improvements in the pass@5 and pass@100. It’s worth emphasizing that the enhancement engendered by our method on the pass@1 metric is significantly more pronounced than on the pass@100. This denotes that CodePLAN significantly escalates the likelihood of the model generating correct code for the identical question. Furthermore, CodePLAN can generate a larger volume of correct codes than alternative methods, thereby proving advantageous for subsequent post-processing tasks like code ranking. Interestingly, the relative improvement of CodePLAN on complex, competition-level questions surpasses that on introductory-level and interview-level questions, indicating that CodePLAN empowers smaller models to solve complex programming problems with reasoning capabilities.

Method	*Pass@1*				*Pass@5*			
	Intro	Inter	Comp	All	Intro	Inter	Comp	All
CodeGen-350M								
standard finetune	1.54	0.38	0.08	0.56	4.91	1.26	0.37	1.82
CoT finetune	1.38	0.35	0.06	0.50	3.95	1.21	0.21	1.56
CodePLAN w/o PS	**2.06**	**0.56**	**0.09**	**0.77**	**5.27**	**1.62**	**0.40**	**2.11**
CodeT5-770M								
standard finetune	3.30	0.68	0.15	1.10	8.12	1.89	0.74	2.91
CoT finetune	3.22	0.78	0.14	1.15	7.65	1.99	0.36	2.84
CodeRL*	3.76	0.79	0.16	1.25	9.20	2.08	0.69	3.22
CodePLAN w/o PS	**3.90**	**0.80**	**0.20**	**1.30**	**9.25**	**2.17**	**0.78**	**3.31**

Table 2: Results with different training methods on APPS.

Method	*Pass@1*	*Pass@5*	*Pass@80*
CodeGen-350M			
Standard finetune	7.51	14.98	30.29
CoT finetune	7.95	15.01	30.12
CodePLAN w/o PS	**10.39**	**18.66**	**33.05**
CodeT5-770M			
Standard finetune	13.78	26.01	47.89
CoT finetune	12.06	24.03	47.01
CodePLAN w/o PS	**15.13**	**28.07**	**51.09**

Table 3: Results with different training methods on MBPP.

4.3. Comparative Analysis of Various Training Approaches

In this section, we compare various training techniques on APPS and MBPP, namely standard fine-tuning, CoT fine-tuning, RL-based fine-tuning, and CodePLAN without Plan Sampling (abbreviated as CodePLAN w/o PS). Across these methods, a consistent base model is employed. CoT fine-tuning, inspired by existing research (Ho et al., 2022), is not typically used for code generation. For this method, we merge the solution plan with the code to create a target sequence. During inference, the model produces a "CoT + Code" output, with the Code segment extracted for assessment. For the RL-based fine-tuning, our reference point is the CodeT5 checkpoint released by CodeRL (Le et al., 2022), a framework that harnesses RL training for code generation.

Result on APPS. On the APPS benchmark, we conducted this experiment using CodeT5 770M (Wang et al., 2021) and CodeGen 350M (Nijkamp et al., 2022) as base models. Table 2 presents the comparative results of CodePLAN w/o PS alongside various fine-tuning methodologies on the APPS benchmark. We can find that the code generation ability of smaller base models is improved by distilling the reasoning ability of the LLM, and that this approach outperforms other fine-tuning methods on all difficulty levels of the APPS benchmark. Compared to standard fine-tuning and RL-based fine-tuning, the inference ability of the smaller model is improved by distilling the inference ability of the LLM thus indirectly improving the code generation ability of the base model. In contrast, standard fine-tuning and RL-based fine-tuning methods lack high-level thinking as a supervisory signal and are not beneficial for improving the reasoning ability of smaller models. While CoT fine-tuning has proven its mettle in simpler NLP tasks (Ho et al., 2022), our experiments reveal its direct application to the intricate realm of code generation to be less impactful. In this context, CodePLAN demonstrates a significant edge.

Result on MBPP. We also conducted this experiment on MBPP, where we followed the experimental setup of the original paper (Austin et al., 2021) and also used the CodeT5-770M and CodeGen-350M as the base models. The outcomes are presented in Table 3. Mirroring findings from the APPS benchmark, CodePLAN (w/o PS) consistently outperforms both standard fine-tuning and CoT fine-tuning methods by leveraging the distilled reasoning capabilities of the LLM.

4.4. Impact Analysis of Varying Solution Plan Sample Sizes

Table 4 presents the results of ablation experiments, examining the effect of varying the number of sampled solution plans during inference. In this setup, the model-generated solution plan s_i is appended to the problem description x_i to guide the code generation process $y_i \longleftarrow f(x_i, s_i)$. Interestingly, with N=1, where the base model generates a single solution plan using greedy decoding, the code generation performance doesn't improve. It might even degrade compared to when no solution plan is used (N=0, represented by $y_i \longleftarrow f(x_i)$). This indicates that solution plans derived via greedy decoding may lack precision. However, employing multiple samplings to select quality solution plans can significantly enhance the model's code generation efficacy. Moreover,

Plan Sampling Number	*Pass@1*				*Pass@5*				*Pass@100*			
	Intro	Inter	Comp	All	Intro	Inter	Comp	All	Intro	Inter	Comp	All
N=0	3.82	0.78	0.15	1.26	9.25	2.07	0.68	3.22	21.30	6.53	6.00	9.38
N=1	2.40	0.57	0.09	0.84	5.49	1.50	0.43	2.08	15.10	5.43	3.90	7.06
N=5	5.33	1.04	0.24	1.74	11.02	2.59	1.04	3.96	24.70	7.57	7.10	10.90
N=10	6.61	1.33	0.35	2.19	12.92	3.03	1.33	4.67	26.30	8.43	8.10	11.94
N=20	**7.87**	**1.61**	**0.42**	**2.62**	**14.66**	**3.54**	**1.59**	**5.37**	**28.60**	**9.17**	**8.60**	**12.94**

Table 4: Results of ablation experiments with different number of sampling plans.

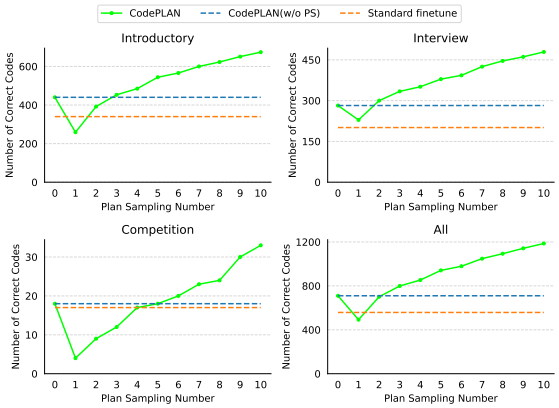

Figure 5: Results of different number of solution plans on the number of correct codes generated.

a greater sampling quantity increases the likelihood of identifying a more accurate solution plan. Figure 5 depicts how sampling solution plans of varying quantities and difficulties impacts the count of accurately generated codes, evaluated against the APPS dataset. Across all difficulty levels, it is evident that, with N=2, there's an uptick in correct codes relative to the "Standard Finetune". By N=3, the performance even eclipses that of "CodePLAN(w/o PS)". These findings underscore that a minimal sampling of solution plans can effectively yield high-quality selections. This confirms that our approach not only amplifies the code generation prowess of smaller models but also refines their inference accuracy.

4.5. Evaluation of LLM-Generated Training Data Quality

In this subsection, we delve into the quality analysis of training data formulated by the LLM. Assessing the quality of solution plans directly generated by the LLM poses challenges. Instead, we resort to an indirect method, evaluating the quality of codes generated under the guidance of these solution plans. For this assessment, we utilize CodeT5 770M, which underwent standard fine-tuning on the APPS dataset. The results, presented in Table 5, compare the quality of LLM-generated solution plans from problem descriptions ("Problem to Plan") and those derived from backward reasoning using ground truth codes ("Code to Plan"). The label "No-Plan" indicates scenarios where LLM-generated plans were not used as auxiliary guidance. Our findings reveal that solution plans derived from backward reasoning using ground truth codes surpass in quality those generated directly from intricate problem descriptions. This likewise indirectly ensures the quality of our distilled data.

Method	*Pass@1*	*Pass@5*	*Pass@10*
CodeT5-770M			
Without Plan	0.37	1.19	1.71
Problem to Plan	0.73	2.02	2.81
Code to Plan	**1.35**	**3.52**	**4.78**

Table 5: Quality results of solution plans generated from LLM using different approaches on APPS.

5. Discussion

How Does Solution Plan Quality Impact Model Performance in Code Generation? Based on the data in Table 4 and Figure 5, it is clear that the quality of solution plans significantly influences model performance. Instead of enhancing the model's code generation capabilities, subpar solution plans might actually degrade its performance. We believe that these lower-quality plans could be misconstrued by the model as noise, negatively affecting its foundational capabilities. On the other hand, high-quality solution plans can greatly boost the model's code generation, leading to a higher output of accurate codes. As such, devising a method to select high-quality solution plans becomes crucial in code generation,

What Distinguishes Our Approach from Conventional Code Post-Processing Methods in Code Generation? It's worth noting that various post-processing code methodologies (Chen et al., 2022; Zhang et al., 2022; Inala et al., 2022) employ a technique to rank potential codes. However, such a ranking strategy doesn't inherently enhance the model's code generation capabilities. In contrast, our approach actively encourages the model to produce more correct codes. Consider a scenario in a programming competition: a conventionally fine-tuned model might generate 100 code samples for a problem, yet only 1 or 2 of those might pass the unit test. This low accuracy complicates the task of code ranking. Conversely, our method drives the model to yield a much

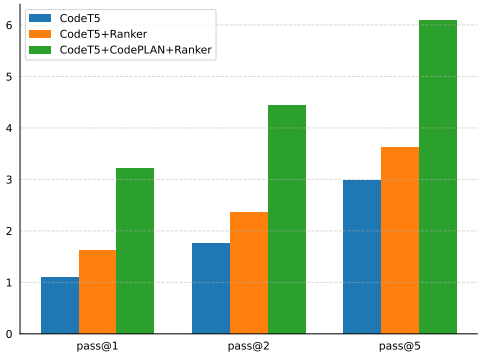

Figure 6: The complementarity between CodeRanker and our CodePLAN.

higher proportion of accurate codes—perhaps 50 to 90 out of 100. This surge in accurate code generation certainly aids in the ranking process. We adopted the CodeRanker (Inala et al., 2022) to train a Ranker to validate our assertions. Figure 6 presents the results of CodeT5, CodeT5+Ranker, and CodeT5+CodePLAN+Ranker on APPS. CodeT5+Ranker improves CodeT5's pass@{1,2,5} by an average of 54.5%, and in combination with CodePLAN, CodeT5+Ranker+CodePLAN brings an average of 91.3% improvement, it demonstrates the advantages of CodePLAN in code ranking tasks and it is also proved that the two are orthogonal.

6. Conclusion

We introduced an innovative code generation framework named CodePLAN. During its training phase, CodePLAN uniquely emphasizes the generation of solution plans, aiming to refine and optimize the overall code generation process. In the inference phase, the framework leverages autonomously produced solution plans, strategically enhancing the likelihood of producing accurate codes. Our extensive experimental evaluations provide compelling evidence of the efficacy of our approach, demonstrating a significant boost in the performance of smaller models in code generation.

Limitations

Here we summarize two main limitations:

Firstly, the first limitation is that due to the limited dataset we have not considered what CodePLAN's preferences are for different types of programming topics. Some code generation datasets are now starting to consider dividing the dataset based on different algorithms (Li et al., 2023b), so we may in the future integrate different algorithms in the solution plan to enhance CodePLAN's ability to learn different algorithms and evaluate CodePLAN at a fine-grained level (e.g., different algorithm types).

Secondly, we considered only one programming language. In our future work, we plan to explore the adaptability of this framework across different programming languages and more intricate coding scenarios. With the continuous advancement in automatic code generation techniques, we believe methods like CodePLAN will play a pivotal role in furthering the progress of this domain.

Acknowledgments

The work is supported in part by the Natural Science Foundation of Shandong Province, China (Grant No. ZR2021MF059), the National Natural Science Foundation of China (Grant Nos. 62192731, 62072007, 62192733, 61832009, 62192730), the National Key R&D Program (Grant No. 2023YFB4503801) and the Key Program of Hubei (Grant No. JD2023008).

7. Bibliographical References

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

John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, Lois M Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, Harold Stern, et al. 1957. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198.

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 Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared

Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

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. *arXiv preprint arXiv: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. *NeurIPS*.

Namgyu Ho, Laura Schmid, and Se-Young Yun. 2022. Large language models are reasoning teachers. *arXiv preprint arXiv:2212.10071*.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*.

Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. 2024. Knowledge-aware code generation with large language models. *arXiv preprint arXiv:2401.15940*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. *Advances in Neural Information Processing Systems*, 35:13419–13432.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*.

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. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Bolun Li, Zhihong Sun, Tao Huang, Hongyu Zhang, Yao Wan, Ge Li, Zhi Jin, and Chen Lyu. 2024. Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. *arXiv preprint arXiv:2401.16637*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023a. Starcoder: may the source be with you!

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023b. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. 2023c. Think outside the code: Brainstorming boosts large language models in code generation. *arXiv preprint arXiv:2305.10679*.

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

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.

Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. 2021. Embedding api dependency graph for neural code generation. *Empirical Software Engineering*, 26:1–51.

Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2022. ChatGPT. https://openai.com/blog/chatgpt/.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*.

KaShun Shum, Shizhe Diao, and Tong Zhang. 2023. Automatic prompt augmentation and selection with chain-of-thought from labeled data. *arXiv preprint arXiv:2302.12822*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Richard J Waldinger and Richard CT Lee. 1969. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252.

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

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

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*.

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. *arXiv preprint arXiv:2211.16490*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A. Early Exploration Experiments

In the course of our preliminary experimental investigations, we discerned that the solution plans, conjured by LLM(specifically refers to gpt-3.5-turbo) from the problem descriptions, were integrated with these descriptions to serve as prompts. This composite data was subsequently fed to the fine-tuned smaller models. We observed that this methodology marginally enhanced the smaller models' aptitude in addressing complex programming problems. Furthermore, by manually selecting the most superior solution plans, we were able to significantly amplify the smaller models' code generation capacities. We singled out these high-quality solution plans based on the substantial improvement they afforded to the CodeT5-770M model. Interestingly, as revealed from our analysis of Table 6, these high-quality solution plans demonstrated a degree of generalization, thereby leading to a substantial improvement in the performance of other smaller models. This crucial insight spurred us to explore a fresh approach aimed at tapping into the latent potential of smaller models in tackling intricate programming problems.

B. Examples

B.1. Examples of Generated Plans

Some examples of solution plans generated by LLM based on ground truth codes are shown in Figure 7 and Figure 8. The example in Figure 7 is an interview-level problem on the APPS benchmark, and the example in Figure 8 is a competition-level problem on the APPS benchmark.

B.2. Example Generated Programs

Figures 9 to 11 provide illustrative examples of code produced by CodeT5, under the influence of various fine-tuning methods. Code segments failing to pass the unit test are presented on a red background, whilst code that successfully navigates the unit test is outlined on a green background. As evidenced in Figure 9, CodeT5, when fine-tuned using our proposed approach, generates a significantly higher volume of correct code than when it is fine-tuned with the standard method. Figure 10 depicts how CodeT5, when fine-tuned using our methodology, can effectively address issues that remain unresolved when CodeT5 is fine-tuned using the standard method. This outcome stems from the standard fine-tuning method's limited success in enhancing the smaller model's reasoning abilities, whereas our method successfully amplifies the mini-model's cognitive capabilities by fostering mutual enhancement between the two tasks. Furthermore, our strategy of deploying high-quality solution plans as cues significantly augments the probability of correct code generation. Figure 11, however, reveals a failure scenario where despite our method yielding a greater number of accurate codes, the selection of an incorrect solution plan (generated by the greedy decoding of CodePLAN w/o PS) as a cue fails to produce correct codes. This is attributable to the incorrect solution plan being perceived as noise by the smaller model, thereby compromising the smaller model's innate abilities.

Method	Intro	Inter	Comp	All
CodeT5-770M		**Pass@1**		
No-Plan	3.30	0.68	0.16	1.10
With-Plan(greedy)	4.90 (↑ 48.5%)	1.03 (↑ 51.5%)	0.14 (↓ 14.3%)	1.62 (↑ 47.3%)
With-Plan(best)	**9.72** (↑ 194.5%)	**1.91** (↑ 180.9%)	**0.45** (↑ 181.3%)	**3.18** (↑ 189.1%)
CodeT5-770M		**Pass@5**		
No-Plan	8.12	2.01	0.74	2.98
With-Plan(greedy)	11.30 (↑ 39.2%)	2.69 (↑ 33.8%)	0.61 (↓ 17.6%)	4.00 (↑ 34.2%)
With-Plan(best)	**18.90** (↑ 132.8%)	**4.56** (↑ 126.9%)	**1.80** (↑ 143.2%)	**6.88** (↑ 130.9%)
CodeT5-220M		**Pass@1**		
No-Plan	0.71	0.27	0.03	0.31
With-Plan(greedy)	1.36 (↑ 91.5%)	0.29 (↑ 7.41%)	0.04 (↑ 33.3%)	0.45 (↑ 45.2%)
With-Plan(best)	**1.62** (↑ 128.2%)	**0.40** (↑ 48.2%)	**0.06** (↑ 100.0%)	**0.58** (↑ 87.1%)
CodeT5-220M		**Pass@5**		
No-Plan	2.40	0.94	0.12	1.07
With-Plan(greedy)	3.56 (↑ 48.3%)	1.02 (↑ 8.5%)	0.19 (↑ 58.3%)	1.36 (↑ 27.1%)
With-Plan(best)	**4.71** (↑ 96.3%)	**1.22** (↑ 29.8%)	**0.24** (↑ 100.0%)	**1.70** (↑ 58.9%)
GPT2-1.5B		**Pass@1**		
No-Plan	1.30	**0.70**	0.00	0.68
With-Plan(greedy)	3.37 (↑ 159.2%)	0.50 (↓ 40.0%)	0.04 (↑ ∞)	0.94 (↑ 38.2%)
With-Plan(best)	**4.13** (↑ 217.7%)	0.65 (↓ 7.1%)	**0.08** (↑ ∞)	**1.20** (↑ 76.5%)
GPT2-1.5B		**Pass@5**		
No-Plan	3.60	1.03	0.00	1.34
With-Plan(greedy)	8.20 (↑ 127.8%)	1.54 (↑ 49.5%)	0.22 (↑ ∞)	2.51 (↑ 87.3%)
With-Plan(best)	**9.66** (↑ 168.3%)	**1.79** (↑ 73.8%)	**0.38** (↑ ∞)	**3.02** (↑ 125.4%)

Table 6: Results of different smaller models on APPS with different solution plans generated from LLM.

Problem Specification

```
Numbers $1, 2, 3, \dots n$ (each integer from $1$ to $n$ once) are written on a board. In one operation you can erase any two numbers $a$ and $b$ from the board and write one integer $\frac{a + b}{2}$ rounded up instead.
You should perform the given operation $n - 1$ times and make the resulting number that will be left on the board as small as possible. For example, if $n = 4$, the following course of action is optimal:  choose $a = 4$ and $b = 2$, so the new number is $3$, and the whiteboard contains $[1, 3, 3]$;  choose $a = 3$ and $b = 3$, so the new number is $3$, and the whiteboard contains $[1, 3]$;choose $a = 1$ and $b = 3$, so the new number is $2$, and the whiteboard contains $[2]$. It's easy to see that after $n - 1$ operations,there will be left only one number.Your goal is to minimize it.
-----Input-----
The first line contains one integer $t$ ($1 \le t \le 1000$) — the number of test cases.
The only line of each test case contains one integer $n$ ($2 \le n \le 2 \cdot 10^5$) — the number of integers written on the board initially. It's guaranteed that the total sum of $n$ over test cases doesn't exceed $2 \cdot 10^5$.
-----Output-----
For each test case, in the first line, print the minimum possible number left on the board after $n - 1$ operations. Each of the next $n - 1$ lines should contain two integers — numbers $a$ and $b$ chosen and erased in each operation.
-----Example-----
Input 1\n4 Output 2\n2 4\n3 3\n3 1
```

Ground Truth Code

```
for _ in range(int(input())):
    n = int(input())
    k = n
    print(2)
    for i in range(n - 1, 0, -1):
        print(i, k)
        if (k + i) % 2 != 0:
            k = (k + i) // 2 + 1
        else:
            k = (k + i) // 2
```

Solution Plan

```
1. Read the input value for the number of
test cases.

2. For each test case, read the input value
for n.

3. Assign k the value n.

4. Print the number 2.

5. For values of i between n-1 and 1 (inclusive), perform the
following steps:

    - Print the values of i and k.

    - Check if the sum of k and i is odd. If so, update the
value of k to (k+i)//2 + 1. If not, update the value of k to
(k+i)//2.

6. Repeat steps 2-5 for all the test cases.
```

Figure 7: An example of an interview-level problem and ground truth code on the APPS benchmark, and a solution plan generated by LLM based on the ground truth code.

Problem Specification

```
You're playing a game called Osu! Here's a simplified version of it. There are n clicks in a game. For each click there are
two outcomes: correct or bad. Let us denote correct as "O", bad as "X", then the whole play can be encoded as a sequence of n
characters "O" and "X". Using the play sequence you can calculate the score for the play as follows: for every maximal
consecutive "O"s block, add the square of its length (the number of characters "O") to the score. For example, if your play
can be encoded as "OOXOOOXXOO", then there's three maximal consecutive "O"s block "OO", "OOO", "OO", so your score will be 2^2
+ 3^2 + 2^2 = 17. If there are no correct clicks in a play then the score for the play equals to 0. You know that the
probability to click the i-th (1 ≤ i ≤ n) click correctly is p_{i}. In other words, the i-th character in the play sequence
has p_{i} probability to be "O", 1 - p_{i} to be "X". You task is to calculate the expected score for your play.
-----Input-----
The first line contains an integer n (1 ≤ n ≤ 10^5) — the number of clicks. The second line contains n space-separated real
numbers p_1, p_2, ..., p_{n} (0 ≤ p_{i} ≤ 1).
There will be at most six digits after the decimal point in the given p_{i}.
-----Output-----
Print a single real number — the expected score for your play. Your answer will be considered correct if its absolute or
relative error does not exceed 10^{ - 6}.
-----Examples-----
Input 3\n0.5 0.5 0.5  Output 2.750000000000000
-----Note-----
For the first example. There are 8 possible outcomes. Each has a probability of 0.125.  "OOO"  →  3^2 = 9;  "OOX"  →  2^2 =
4;  "OXO"  →  1^2 + 1^2 = 2;  "OXX"  →  1^2 = 1;  "XOO"  →  2^2 = 4;  "XOX"  →  1^2 = 1;  "XXO"  →  1^2 = 1;  "XXX"  →  0. So
the expected score is $\frac{9 + 4 + 2 + 1 + 4 + 1 + 1}{8} = 2.75$
```

Ground Truth Code

```
(n, a, b) = (int(input()), 0, 0)

for i in map(float, input().split()):

    (a, b) = (a + i * (1 + b * 2), i * (b + 1))

print(a)
```

Solution Plan

```
1. Read the input value for n.

2. Initialize variable a to 0. Initialize
variable b to 0.

3. For each floating point number i in a list created by
  splitting the input(), do the following:

  a. Update the value of variable a by adding i multiplied by
  (1 plus b multiplied by 2) to the current value of a.

  b. Update the value of variable b by adding i multiplied by
  (b plus 1) to the current value of b.

4. Print the value of a.
```

Figure 8: An example of an competition-level problem and ground truth code on the APPS benchmark, and a solution plan generated by LLM based on the ground truth code.

Problem Specification

Given is a string S. Replace every character in S with x and print the result.

-----Constraints-----
- S is a string consisting of lowercase English letters.
- The length of S is between 1 and 100 (inclusive).

-----Input-----
Input is given from Standard Input in the following format:
S

-----Output-----
Replace every character in S with x and print the result.

-----Sample Input-----
sardine

-----Sample Output-----
xxxxxxx

Replacing every character in S with x results in xxxxxxx.

Standard Fine-tuning

```
s = input()
s = list(s)
for i in range(len(s)):
    if(s[i]!='x'):
        s[i] = 'x'
print(''.join(s))
```

```
s=input()
for i in s:
    if i=='x':
        print(i,end='')
    else:
        print(i,end='')
```

```
s = input()
for i in range(len(s)):
    s1 = s[i]
    s2 =
str(s1).replace(s1,
str(i+1))
    print(s2)
```

```
s = input()
ans = ""
for i in s:
    ans += i+"x"
print(ans)
```

CodePLAN (w/o PS)

```
S = str(input())
B = []
for i in S:
    b = str(i)
    B.append(b)
for i in S:
    x = B.index(i)
    B[x] = 'x'
print(''.join(B))
```

```
S = input()
s = []
for i in S:
    s.append(i)
s.sort()
for i in range(len(s)):
    s[i]=s[i].replace(s[i]
, 'x')
print(''.join(s))
```

```
s = input()
x = []
for i in s:
    z = i.replace(i, 'x')
    x.append(z)
print(''.join(x))
```

```
S = str(input())
ans = []
for i in S:
    ans.append(i.replace(i
, 'x'))
print(''.join(ans))
```

Figure 9: **An example of a problem on the APPS benchmark and code generated by CodeT5 with different ways of fine-tuning:** only one of the four copies of code generated by CodeT5 with standard finetune passes the unit test, while all four copies of code generated by CodeT5 with our CodePLAN (w/o PS) fine-tuning pass the unit test.

Problem Specification

Find the number of palindromic numbers among the integers between A and B (inclusive).
Here, a palindromic number is a positive integer whose string representation in base 10 (without leading zeros) reads the same forward and backward.

-----Constraints-----
- 10000 \leq A \leq B \leq 99999
- All input values are integers.

-----Input-----
Input is given from Standard Input in the following format:
A B

-----Output-----
Print the number of palindromic numbers among the integers between A and B (inclusive).

-----Sample Input-----
11009 11332

-----Sample Output-----
4

There are four integers that satisfy the conditions: 11011, 11111, 11211 and 11311.

Standard Fine-tuning

```
A,B = map(int,input().split())
print(sum(11*(10**(i-1))-
int(str(i)[::-1])) for i in
range(A,B+1))
```

```
a,b=list(map(int,input()
.split()))
c=0
for i in range(a,b+1):
    s=str(i)
    p=10**len(s)
    if p<=i:
        c+=int(s[::-1])
print(c)
```

```
a, b = map(int,
input().split())
c = []
for i in range(a, b + 1):
    for j in range(10):
        if j > 9:
            j -= 10
        r = int(str(i)[::-1])
        if r > 9:
            r -= 10
        c.append(r)
ans = sum(c)
print(ans)
```

```
def main():
    a,b = list(map(int,
        input().split()))
    ans = 0
    for i in range(a, b + 1):
        ans += int(str(i)
        [::-1]) < 10 ** 18
    print(ans)
def __starting_point():
    main()
__starting_point()
```

CodePLAN (w/o PS)

```
(a, b) = map(int,
input().split())
ans = 0
for i in range(a, b + 1):
    p = str(i)
    if p[::-1] == p:
        ans += 1
print(ans)
```

```
(A, B) = map(int,
input().split())
(A, B) = (A - 1, B - 1)
(N, C) = (0, set())
for i in range(A, B + 1):
    if i in C:
        continue
    if str(i)[::-1] == str(i):
        C.add(i)
    else:
        N += 1
print(N)
```

```
(a, b) = map(int,
input().split())
l = list(range(a, b + 1))
def f(x):
    s = str(x)
    if len(s) % 2 == 0:
        s = s[1:]
    return int(s, 10)
ans = 0
for x in l:
    if f(x) == x:
        ans += 1
print(ans)
```

```
(a, b) = map(int,
        input().split())
c = 0
for i in range(a, b + 1):
    (p, s) = (0, str(i))
    for j in range(len(s) -
1):
        p = 10 * p + int(s[j])
    if p < 10 and p == int(s):
        c += 1
print(c)
```

CodePLAN

1. Define a function called `is_palindrome` that takes in a number as input and checks if it is a palindrome.**2.** Read in two integers `A` and `B` as input.**3.** Initialize a variable called `ans` to 0.**4.** Loop through all numbers between `A` and `B` (inclusive) using the `range` function.**5.** Convert each number to a string using `str(i)`.**6.** Check if the string is equal to its reverse using `str(i)[::-1]`.**7.** If the string is a palindrome, increment `ans`.**8.** Print out the final value of `ans`.",

```
def is_palindrome(n):
    s = str(n)
    return s == s[::-1]
(a,b)=map(int,input().split())
ans = 0
for i in range(a, b + 1):
    s = str(i)
    if is_palindrome(i):
        ans += 1
print(ans)
```

```
def is_palindrome(n):
    return str(n) == str(n)[::-1]
(a,b)=map(int,input().split())
ans = 0
for i in range(a, b + 1):
    s = str(i)
    if is_palindrome(i):
        ans += 1
print(ans)
```

```
def is_palindrome(x):
    x = str(x)
    if x == x[::-1]:
        return True
    else:
        return False
(a,b)= map(int,input().split())
ans = 0
for i in range(a, b + 1):
    x = str(i)
    if is_palindrome(i):
        ans += 1
print(ans)
```

```
(a, b) =
map(int,input().split())
c = 0
for i in range(a, b + 1):
    x = str(i)
    if x == x[::-1]:
        c += 1
print(c)
```

Figure 10: **An example of a problem on the APPS benchmark and code generated by CodeT5 with different ways of fine-tuning:** CodeT5 with the standard fine-tune method generates four copies of code that do not pass the unit test, while CodeT5 with our CodePLAN (w/o PS) generates one of the four copies of code that pass the unit test, and we select a high-quality solution plan to be spliced after the problem description as a prompt, and CodeT5 with CodePLAN generates four copies of code that pass the unit test.

Problem Specification

In 2020, AtCoder Inc. with an annual sales of more than one billion yen (the currency of Japan) has started a business in programming education.
One day, there was an exam where a one-year-old child must write a program that prints Hello World, and a two-year-old child must write a program that receives integers A, B and prints A+B.
Takahashi, who is taking this exam, suddenly forgets his age.
He decides to write a program that first receives his age N (1 or 2) as input, then prints Hello World if N=1, and additionally receives integers A, B and prints A+B if N=2.
Write this program for him.
-----Constraints-----
- N is 1 or 2.
- A is an integer between 1 and 9 (inclusive).
- B is an integer between 1 and 9 (inclusive).
-----Input-----
Input is given from Standard Input in one of the following formats:
1
2
A
B
-----Output-----
If N=1, print Hello World; if N=2, print A+B.
-----Sample Input-----
1
-----Sample Output-----
Hello World
As N=1, Takahashi is one year old. Thus, we should print Hello World.

Standard Fine-tuning

```
n=int(input())
if n==1:
    print("Hello World")
elif n==2:
    print("Hello World")
else:
    print(n*a+n*b)
```

```
n = int(input())
if n==1:
    print("Hello World")
elif n==2:
    print("Hello World")
    print(n+n)
else:
    print("Hello World")
    print(n+n)
```

```
n=int(input())
if n%2==0:
    print("Hello World")
else:
    print("Hello World")
    a=n//2
    b=n%2
    print(a,b)
```

```
n=int(input())
if n==1:
    print("Hello World")
elif n==2:
    print("Hello World")
else:
    print("Hello World")
    print(n,n+1,n+2)
```

CodePLAN (w/o PS)

```
import sys
input = sys.stdin.readline
n = int(input())
if n == 1:
    print('Hello World')
else:
    a = int(input())
    b = int(input())
    print(a + b)
```

```
n = int(input())
if n == 1:
    print('Hello World')
else:
    a = int(input())
    b = int(input())
    if n == 2:
        print(a + b)
    else:
        print(a + b)
```

```
n = int(input())
if n == 1:
    print('Hello World')
else:
    print(int(input()) +
int(input()))
```

```
n = int(input())
if n == 1:
    print('Hello World')
else:
    (a, b) = map(int,
input().split())
    print(a + b)
```

CodePLAN

1. Read an input value that represents the number of test cases to be executed.**2.** For each test case, read in two integers a and b.**3.** If a is equal to 1, print \"Hello World\".**4.** If a is equal to 2, print a+b.\n5. If a is not equal to 1 or 2, continue to the next step.**6.** Repeat steps 2-5 for all test cases.

```
n = int(input())
for i in range(n):
    (a, b) =
 map(int,input().split())
    if a == 1:
        print('Hello
World')
    elif a == 2:
        print(a + b)
    else:
        continue
```

```
for i in range(int(input())):
    (a, b) = map(int,
input().split())
    if a == 1:
        print('Hello
World')
    elif a == 2:
        print(a + b)
    else:
        continue
```

```
import sys
input = sys.stdin.readline
n = int(input())
for i in range(n):
    (a, b) = list(map(int,
input().split()))
    if a == 1:
        print('Hello World')
    elif a == 2:
        print(a + b)
    else:
        pass
```

```
n = int(input())
while n:
    (a, b) = list(map(int,
input().split()))
    if a == 1:
        print('Hello World')
    elif a == 2:
        print(a + b)
    else:
        a -= 1
    n -= 1
```

Figure 11: **An example of a problem on the APPS benchmark and code generated by CodeT5 with different ways of fine-tuning:** All four copies of CodeT5 generated by the standard fine-tuning method fail the unit test, while all four copies of CodeT5 generated by our CodePLAN (w/o PS) pass the unit test. However, we select an incorrect solution plan and splice it after the problem description as a prompt that all four copies of CodeT5 with CodePLAN do not pass the unit test.