Less is More: Summary of Long Instructions is Better for Program Synthesis

Kirby Kuznia* **Swaroop Mishra*** **Mihir Parmar** **Chitta Baral**

Arizona State University

Abstract

Despite the success of large pre-trained language models (LMs) such as Codex, they show below-par performance on the larger and more complicated programming related questions. We show that LMs benefit from the summarized version of complicated questions. Our findings show that superfluous information often present in problem description such as human characters, background stories, and names (which are included to help humans in understanding a task) does not help models in understanding a task. To this extent, we create a meta-dataset from the frequently used APPS dataset and the newly created CodeContests dataset for the program synthesis task. Our meta-dataset consists of human and synthesized summaries of the long and complicated programming questions. Experimental results on Codex show that our proposed approach outperforms baseline by 8.13% on the APPS dataset and 11.88% on the CodeContests dataset on average in terms of strict accuracy. Our analysis shows that summaries significantly improve performance for introductory (9.86%) and interview (11.48%) programming questions. However, it shows improvement by a small margin ($\sim$ 2%) for competitive programming questions, implying scope for future research in this direction.[1]

1 Introduction

Recently, large pre-trained LMs have been proven pivotal in programming-related tasks (Wang et al., 2021; Chen et al., 2021; Hendrycks et al., 2021; Lu et al., 2021; Papineni et al., 2002)[2]. Program synthesis aims to generate a code given the natural language description of a problem. Programming requirements in these problems vary in terms of complexity from a 3-5 line simple function to multiple functions that use advanced data structures. However, LMs such as Codex show below-par performance on the long and complicated programming questions. We observe that the natural language description of the program becomes long and complicated when there is superfluous information (see section 2.1.1). The goal of adding this information to the description is to make it more understandable to humans. However, we find that this information confuses the model in understanding a task[3]. We propose that removing the excess information and providing the model with the exact specifications of the problem can improve the performance of the LMs.

To remove excess information[4], we summarize the descriptions of the program in such a way that it does not lose important specifications. We use the APPS dataset (Hendrycks et al., 2021) and CodeContests dataset (Li et al., 2022) which are a collection of coding problems from different online sources and create a meta-dataset consisting of human and synthesized summaries.

We perform all experiments using the GPT-based Codex model (Chen et al., 2021) on the proposed meta-dataset and show that the summarized version of complicated questions improves strict accuracy by 8.13% on the APPS dataset and 11.85% on CodeContests. From our analysis, we can see significant improvement for introductory (9.86%) and interview (11.48%) related programming questions. However, it shows improvement by a small margin ($\sim$ 2%) for competitive programming questions. Considering that automatic evaluation of a program does not reward for partial correctness, we perform qualitative evaluation on our meta-dataset and find that original questions often confuse models in understanding the underlying problem, as models latch on to some spurious words in the text (e.g. the word 'list' in question makes the model

*Equal Contribution

[1]Code and data is available at https://github.com/kurbster/Prompt-Summarization

[2]Detailed related work is presented in Appendix A

[3]See example in Appendix C

[4]Instructions for creating summaries given in Appendix N

Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 4532 - 4552
December 7-11, 2022

design a list even though the underlying problem is on graphs). We further analyze model performance on different types of summaries (i.e., basic, expert, and synthetic) and provide instruction-design principles that can help future research on prompting in program synthesis.

2 Method

2.1 Dataset

We use the APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022) datasets to create summaries. We crowd-sourced the creation of human summaries. The result was 373 human summaries for APPS and 80 summaries for CodeContests along with and 8663 synthetic summaries using both datasets. Table 1 shows the statistics of the generated summaries.

Data Source	Difficulty	# of Problems
Human	Introductory	145
	Interview	123
	Competition	105
	CodeContests	80
	Total	453
Studio21	Introductory	1588
	Interview	4551
	Competition	1286
	CodeContests	80
	Total	7505
GPT-3	Introductory	194
	Interview	267
	Competition	244
	CodeContests	80
	Total	785
PEGASUS	Introductory	145
	Interview	123
	Competition	105
	Total	373

Table 1: Statistics of the proposed meta-dataset.

2.1.1 Human Generated Summaries

For the APPS and CodeContests human-generated summaries, the crowd worker reads and understands the original questions, then creates summaries in two steps[5]. First, we create a basic summary of the given problem and remove any information that is repeated and any hypothetical information without concrete instructions. For example, if the problem constructs a fake company or situation, we replace the fake situation with direct instructions. Full example is included in Appendix C. Second, we create an expert summary of the problem. To create this, we further summarize the first summary. This expert summary includes the absolute minimum information for an expert to understand the problem. We would not expect a novice to understand these prompts. An example of expert summaries is given in Appendix C.3.

2.1.2 Synthetic Summaries

We have generated synthetic summaries of program descriptions using jumbo (178B), large (7.5B) Studio21 model (Lieber et al., 2021), GPT-3 Davinci model (175B) (Brown et al., 2020) and PEGASUS model (Zhang et al., 2019). To generate a summary, we provide these models with a few examples in the in-context learning setup (Brown et al., 2020) from the human-generated summaries. For the few-shot examples, we use expert-level summaries.

Studio21 We use five examples with the large model, and three examples with the jumbo model[6]. For both models, we use a temperature of 0.3, and topP of 1. For the format of our prompt, we use DeJargonizer template[7] with a change to their header as shown in Appendix D. We create a total of 7,505 synthetic summaries using these models.

GPT-3 We use three examples for GPT-3 model. We empirically set temperature to 0.05, topP to 1, frequency penalty to 0.01, presence penalty to 0.05. To generate prompts, we followed their tl;dr template[8] as shown in Appendix D. We create 785 synthetic summaries using this model.

PEGASUS We use the PEGASUS model (Zhang et al., 2019) to create program summaries for the same set of problems that were summarized by humans. We choose this model because it was trained specifically for abstractive summarization.

2.2 Model

We use OpenAI Codex to build baselines and the proposed approach.

Baseline To create a baseline, we have used original program descriptions given in the datasets as prompts for the Codex model.

[5]Instructions for creating summaries are in Appendix N

[6]Examples are included in Appendix D

[7]https://studio.ai21.com/

[8]https://beta.openai.com/playground/p/default-tldr-summary?model=text-davinci-001

Difficulty	AP			EWPR			BWPR		
	Baseline	Basic	Expert	Baseline	Basic	Expert	Baseline	Basic	Expert
Introductory	42.96	**50.00**	**50.00**	44.20	51.45	**51.82**	43.23	**50.35**	**50.35**
Interview	37.70	41.80	**44.26**	36.52	45.54	**46.96**	37.70	41.80	**44.26**
Competition	4.76	**5.71**	**5.71**	4.00	**6.00**	**6.00**	4.76	**5.71**	**5.71**
Weighted Average	30.47	34.83	**35.64**	30.31	36.65	**37.22**	30.43	34.78	**35.59**
CodeContests	12.50	23.75	**25.00**	13.33	25.33	**26.66**	12.82	24.36	**25.64**

Table 2: Results of baseline and proposed model in terms of Strict Accuracy (SAcc). The first block is from the APPS dataset. The last block is from the CodeContests dataset. AP: All Problems, EWPR: Either Worst Problem Removal, BWPR: Both Worst Problem Removal (see explanation in section 3). All results are in %. Weighted Average is not shown for CodeContests because similar difficulties were not provided (see explanation in 4.1).

Difficulty	AP		EWPR	
	Baseline	Proposed	Baseline	Proposed
Introductory	42.96	**52.82**	44.53	**54.74**
Interview	37.70	**49.18**	38.66	**50.42**
Competition	4.76	**6.67**	4.81	**6.73**
Weighted Average	30.47	**38.48**	31.11	**39.44**

Table 3: Results when taking the best summary for each problem. The EWPR baseline is different from Table 2 because a different set of problems have been removed.

Proposed Approach We have used summaries of original program descriptions given in the datasets as prompts for the Codex model.

3 Experimental Setup

All the experiments are performed using the $davinci-codex$ (Chen et al., 2021) model provided through OpenAI[9]. At inference time, we use a modified version of the evaluation code[10] provided by Hendrycks et al. (2021). This evaluation code has four different outputs for each test case: (1) **-2**: the code has a syntax error and can not run, (2) **-1**: the code is syntactically correct but has a run time error, (3) **0**: the code runs without any errors but fails the test case, and (4) **1**: the code runs without any error and passes the test case. Similar to Chen et al. (2021), we implement a timeout for the code at inference time. If a test case takes more than 4 seconds to run then we throw an exception and count that test case as a -1.

Experiments To show effectiveness of the proposed approach, we have performed three different experiments using human generated summaries:

1. All problems from basic and expert summaries are used at inference time. We term this experiment All Problems (AP).
2. We eliminate problems that perform worse[11] for either basic or expert summaries. We term this experiment Either Worst Problem Removal (EWPR).
3. We eliminate problems that perform worse for both basic and expert summaries. We term this experiment Both Worst Problem Removal (BWPR).

Motivation behind EWPR and BWPR If a summary caused every test case to perform worse then it's likely the crowd worker produced a faulty summary. To mitigate the effect of outliers in the dataset, we use the EWPR method to remove such problems. Another hypothesis is that every problem benefits from some level of summarization (i.e., basic or expert). To measure this, we use the BWPR method. From Table 6 results, we identify that only 1 problem had both summaries (basic and expert) preform worse.

Metric In (Austin et al., 2021a), they show that the BLEU metric (Papineni et al., 2002) does not correlate well with synthesis performance. Thus, we use Strict Accuracy (SAcc) as our evaluation metric for all experiments (see Appendix E).

4 Results and Analysis

4.1 Human Generated Summaries

From Table 2, we can observe that both the summary-based models show on average superior performance compared to baseline. In particular, when calculating results for every problem,

[9] Implementation and parameters details in Appendix F

[10] https://github.com/hendrycks/apps/blob/main/eval/test_one_solution.py

[11] Definition of the worst problem is given in Appendix G

Model	Difficulty	AP		EWPR	
		Baseline	Proposed	Baseline	Proposed
GPT-3	Introductory	**41.75**	38.66	41.11	**41.67**
	Interview	**20.30**	18.80	18.18	**20.66**
	Competition	2.87	**3.28**	2.73	**3.64**
	Weighted Average	**20.17**	18.89	19.14	**20.55**
Studio21	Introductory	**39.53**	31.63	**39.04**	36.36
	Interview	**12.28**	11.00	10.57	**12.37**
	Competition	**1.67**	1.21	**1.38**	**1.38**
	Weighted Average	**11.53**	9.66	10.61	**10.98**
PEGASUS	Introductory	**42.96**	34.48	**44.26**	40.98
	Interview	**37.70**	10.57	14.29	**21.56**
	Competition	**4.76**	0.00	**2.76**	0.00
	Weighted Average	**30.47**	16.88	**25.50**	24.73

Table 4: Results of baseline and proposed approach (All results are in %). Summaries generated by GPT-3, Studio21, and PEGASUS used for inference from APPS.

Model	AP		EWPR	
	Baseline	Proposed	Baseline	Proposed
GPT-3	**12.50**	10.0	**18.75**	**18.75**
Studio21	**12.50**	8.75	**22.5**	20.0

Table 5: Results of baseline and proposed approach (All results are in %). 80 summaries generated by GPT-3 and Studio21 used for inference from CodeContests.

basic and expert summary-based models outperform baseline by 4.34% and 5.15% on average for APPS dataset, respectively. Further analysis shows that the expert summary-based model shows improved performance by $\sim$ 1% compared to the basic summary-based model.

On the CodeContests dataset (Li et al., 2022), we show an average improvement of 11.88% in terms of SAcc. For this dataset, we did not separate the problems by difficulty. This is because the problems come from different sources and have different scales of difficulty. Thus, we did not report the SAcc when weighted by difficulty in Table 2.

Our analysis shows that many problems where the basic summary would fail, however, the expert summary would succeed and vice-versa. Thus, we choose the best summary for each problem after evaluating both summaries and then calculate the results for the best summaries. Table 3 shows results when taking the best summary for each problem for APPS dataset. We observe a 9.86%, 11.48%, and 1.91% increase on SAcc for introductory, interview, and competition level problems, respectively.

4.2 Synthetic Summaries

Table 4 and 5 show the results for baseline, synthetic summaries generated by GPT-3, Studio21 and PEGASUS in terms of SAcc for two experiments. For the AP experiment, we can observe that the performance of the baseline outperforms synthetic summary-based models. However, the proposed model shows an average similar performance compared to the baseline for the EWPR experiment. Moreover, Appendix I shows the results for top 500 and top 1000 summaries from GPT-3 and Studio21, respectively.

4.3 Analysis

Why does eliminating the worst problems help? From Tables 2, we can observe that EWPR and BWPR have improved performance compared to AP for both human and synthetically generated summaries. By analyzing the summarized worst problems, we notice a difference in the summarization style which shows that these summaries are outliers and do not match the distribution of the other summaries. This can cause a problem in synthesizing a good program since the model loses important information. Hence, we believe that eliminating the worst problems improves model

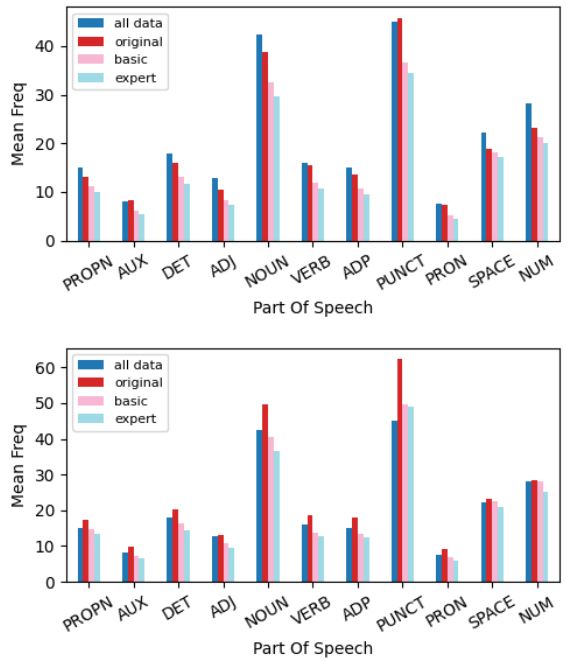


Figure 1: (Top plot) Mean frequency of POS for problems where programs where the generated by both the original and summarized prompt pass all test cases, and (Bottom plot) mean frequency of POS for problems where the summary passes all test cases and the original did not. The blue bar represents the mean of the entire dataset. analyzed only the top 11 most occurring POS. The plot shows that higher number of nouns degrade model performance.

performance.

Is there any possible bias in the meta-dataset? Recent studies shows that bias propagates in human-annotated datasets (Geva et al., 2019; Parmar et al., 2022a). Given that our summaries are also human-generated, there will be some bias in the dataset. Some details that are critical to one person can be trivial to others. In the context of generating expert summaries, assumptions about expert knowledge can vary. This bias causes drift in the dataset and hinders the model's performance. Similar to Mishra et al. (2021), we can provide a template for what is expected from the summary generator to reduce bias.

Why is competition accuracy low? We believe that these problems require multi-hop reasoning, even after summarization, which is still a challenge for language models.

Impact of POS on Accuracy In the top plot of Figure 1, we observe that frequency of *nouns* and *propernouns* for problems that passed all test cases is lower than the entire dataset. In the bottom plot, we observe that the frequency for *nouns* and *propernouns* is higher for the original question (which had < 100% accuracy on the test cases) and lower for the summary (which had 100% accuracy on the test cases). Thus, we can see that number of nouns degrades performance. We also see in the bottom chart that overuse of punctuation can be detrimental to performance. From the results in Figure 1 we see results of nouns affecting performance along with excessive punctuation. Additional detailed analysis is presented in Appendix B.

5 Conclusion

This paper introduces a summarization-based approach for efficient program synthesis. Experimental results show that the proposed approach improves the performance of the Codex model by on average $\sim 8\%$ across various levels of programming questions provided by the APPS and $\sim 11\%$ on the CodeContests. Further, this paper proposes a meta-dataset consisting of ~ 450 human-generated basic and expert-level summaries as well as $\sim 8k$ synthetically generated summaries by GPT-3 and Studio21; this can be helpful for future research on writing better instructions for the program synthesis. We show that program synthesis models benefit from concise prompts, hence, we believe that less number of high-quality instances are better than more low-quality data instances.

Future Extensions The decomposition of prompts has been shown to improve accuracy (Mishra et al., 2022; Patel et al., 2022); splitting up the summarization task the resulting summary can potentially result in higher accuracy for the Codex model in future. Additionally, the PEGASUS model could be used in conjunction with other models to perform the detailed algorithm outlined in Appendix N.

Limitations

Our summary-based approach shows improved performance on program synthesis models, however, it shows competitive performance on synthetic summaries. We believe that the generation of high-quality summaries can improve performance, hence, designing efficient prompts to improve synthetic summaries can be the scope of further research. Furthermore, human-generated summaries

show competitive performance on competition-level problems. These problems require reasoning with multiple logical leaps and knowledge of advanced algorithms and data structures. Hence, exploring new techniques for summarization can be a future research direction. In addition, this work only analyzes the codex model, hence, exploring the effect of summarization on other program synthesis models can be interesting.

References

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

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

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

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.

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, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR.

Ruifang Ge and Raymond Mooney. 2005. A statistical semantic parser that integrates syntax and semantics. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 9–16.

Mor Geva, Yoav Goldberg, and Jonathan Berant. 2019. Are we modeling the task or the annotator? an investigation of annotator bias in natural language understanding datasets. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1161–1166, Hong Kong, China. Association for Computational Linguistics.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

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

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. *arXiv preprint arXiv:2203.07814*.

Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. 2021. Jurassic-1: Technical details and evaluation. *White Paper. AI21 Labs*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Man Luo, Sharad Saxena, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. Biotabqa: Instruction learning for biomedical table question answering. *arXiv preprint arXiv:2207.02419*.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. 2022. Reframing instructional prompts to GPTk's language. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 589–612, Dublin, Ireland. Association for Computational Linguistics.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2021. Cross-task generalization via natural language crowdsourcing instructions. *arXiv preprint arXiv:2104.08773*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Mihir Parmar, Swaroop Mishra, Mor Geva, and Chitta Baral. 2022a. Don't blame the annotator: Bias already starts in the annotation instructions. *arXiv preprint arXiv:2205.00415*.

Mihir Parmar, Swaroop Mishra, Mirali Purohit, Man Luo, Murad Mohammad, and Chitta Baral. 2022b. In-BoXBART: Get instructions into biomedical multi-task learning. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 112–128, Seattle, United States. Association for Computational Linguistics.

Pruthvi Patel, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. Is a question decomposition unit all we need? *EMNLP 2022, Abu Dhabi*.

Ravsehaj Singh Puri, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. How many data samples is an additional instruction worth? *arXiv preprint arXiv:2203.09161*.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2021. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*.

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, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

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*.

Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. 2019. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization.

A Related Work

In the past, there are several methods including semantic parsing (Ge and Mooney, 2005), deductive approaches, enumerative and stochastic search, and constraint solving which have gained attention for program synthesis (Gulwani et al., 2017). With the advent of machine/deep learning, Balog et al. (2016) introduced a neural network based model for solving programming competition-style problems. Devlin et al. (2017) used sequence-to-sequence approach to do program synthesis. Furthermore, Hendrycks et al. (2021) introduced the APPS dataset for testing the accuracy of large LMs on program synthesis. Hendrycks et al. (2021) leveraged the *GPT-Neo* model (Black et al., 2021) which they fine-tune for this task using APPS dataset. CodeT5 model (Wang et al., 2021) utilizes many different training objectives. Recently, Austin et al. (2021b) explore limitations of large language models and propose two new benchmarks, MBPP and MathQA-Python. The Codex model (Chen et al., 2021) is an advanced code generation model that powers GitHub's Copilot. The state of the art model for program synthesis was introduced by Deepmind called AlphaCode (Li et al., 2022). They released their dataset CodeContests, which was used to fine-tune and test their model, and was used in this paper. Our approach suggesting smaller instructions compliments other approaches in improving model performance in instruction paradigm (Mishra et al., 2021; Wei et al., 2022; Parmar et al., 2022b; Nye et al., 2021; Puri et al., 2022; Luo et al., 2022; Wei et al., 2021; Sanh et al., 2021)

B Additional Analysis

Difficulty of CodeContests The accuracies for CodeContests is notably lower than the APPS dataset since this dataset is more challenging, e.g. the number and complexity of programming operations is relatively higher than APPS. From the baseline results in Table 2, we can observe that problems in CodeContests are harder than interview but easier than competition.

Impact of Entities on Accuracy In Figure 2, we can observe that the total number of entities *num_entities* is higher for problems that performed worse. Here, we can see that the original problems (which failed test cases) had a higher mean than the dataset and the summaries (which passed all test cases) had a lower number of entities.

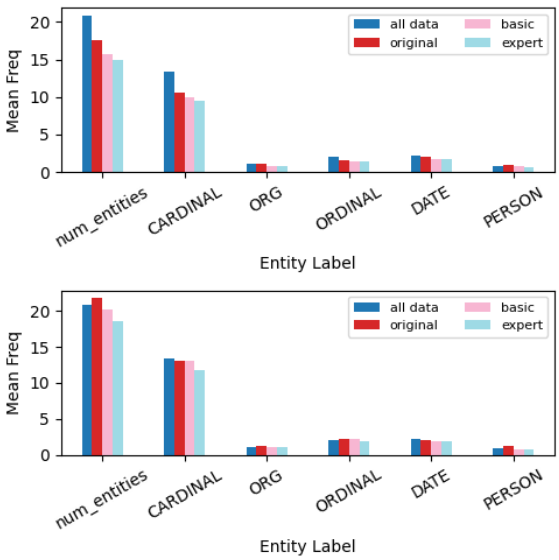


Figure 2: (Top plot) Mean frequency of the entity labels for problems where program generated by the original and summarized prompt pass all test cases, and (Bottom plot) mean frequency of entity labels for problems where the summary passes all test cases and the original did not. We analyzed only the top 5 most occurring entities among all entities we found.

C Example of removing fake information

To see the code produced by the model for this example, refer to Appendix J. There are more examples of superfluous information confusing the model in Appendix O and of made up information confusing the model in Appendix P.

C.1 Original Prompt

> Codefortia is a small island country located somewhere in the West Pacific. It consists of n settlements connected by m bidirectional gravel roads. Curiously enough, the beliefs of the inhabitants require the time needed to pass each road to be equal either to a or b seconds. It's guaranteed that one can go between any pair of settlements by following a sequence of roads.
>
> Codefortia was recently struck by the financial crisis. Therefore, the king decided to abandon some of the roads so that:
>
> it will be possible to travel between each pair of cities using the remaining roads only, the sum of times required to pass each remaining road will be minimum possible (in other words, remaining roads must form minimum spanning tree, using the time to pass the road as its weight), among all the plans minimizing the sum of times above, the time required to travel between the king's

residence (in settlement 1) and the parliament house (in settlement p) using the remaining roads only will be minimum possible.

The king, however, forgot where the parliament house was. For each settlement $p = 1, 2, \ldots, n$, can you tell what is the minimum time required to travel between the king's residence and the parliament house (located in settlement p) after some roads are abandoned?

——Input——

The first line of the input contains four integers n, m, a and b ($2 \leq n \leq 70$, $n - 1 \leq m \leq 200$, $1 \leq a < b \leq 10^7$) — the number of settlements and gravel roads in Codefortia, and two possible travel times. Each of the following lines contains three integers u, v, c ($1 \leq u, v \leq n$, $u \neq v$, $c \in \{a, b\}$) denoting a single gravel road between the settlements u and v, which requires c minutes to travel.

You can assume that the road network is connected and has no loops or multiedges.

——Output——

Output a single line containing n integers. The p-th of them should denote the minimum possible time required to travel from 1 to p after the selected roads are abandoned. Note that for each p you can abandon a different set of roads.

——Examples——

Input
5 5 20 25
1 2 25
2 3 25
3 4 20
4 5 20
5 1 20

Output
0 25 60 40 20

Input
6 7 13 22
1 2 13
2 3 13
1 4 22
3 4 13
4 5 13
5 6 13
6 1 13

Output
0 13 26 39 26 13

——Note——

The minimum possible sum of times required to pass each road in the first example is 85 — exactly one of the roads with passing time 25 must be abandoned. Note that after one of these roads is abandoned, it's now impossible to travel between settlements 1 and 3 in time 50.

We can see the author of the problem is trying to describe a fully-connected graph with n nodes and m edges each with a weight a or b. Thus, this paragraph can be summarized as:

C.2 Basic Summary

You are given a graph of n nodes and m bidirectional edges. The cost for each edge is either a or b. The graph is fully-connected, so you can travel between any pair of nodes using the edges.

For each node $p = 1, 2, \ldots, n$, you need to remove some edges so that: It will be possible to travel between each pair of nodes using the remaining edges only, and the sum of times required to pass each remaining road will be the minimum possible. You should output the minimum time required to travel between node 1 and node p.

——Input——

The first line of the input contains four integers n, m, a and b ($2 \leq n \leq 70$, $n - 1 \leq m \leq 200$, $1 \leq a < b \leq 10^7$) — the number of nodes and edges in the graph, and two possible travel times. Each of the following lines contains three integers u, v, c ($1 \leq u, v \leq n$, $u \neq v$, $c \in \{a, b\}$) denoting an edge between the nodes u and v, which has cost c.

You can assume that the graph is connected and has no loops or multiedges.

——Output——

Output a single line containing n integers. The p-th of them should denote the minimum possible post required to travel from 1 to p after the selected edges are abandoned. Note that for each p you can abandon a different set of edges.

——Examples——

Input
5 5 20 25
1 2 25
2 3 25
3 4 20
4 5 20
5 1 20

Output
0 25 60 40 20

Input
6 7 13 22
1 2 13
2 3 13
1 4 22
3 4 13
4 5 13
5 6 13
6 1 13

Output
0 13 26 39 26 13

C.3 Expert Summary

However, we can assume that an expert would already know what a minimum spanning tree is. Thus, we can remove this detailed description of an MST.

> You are given a connected graph of n nodes and m bidirectional edges. For each node $p = 1, 2, \ldots, n$, you need to find a minimum spanning tree. Then output the minimum cost required to travel between node 1 and node p.
>
> ——Input——
>
> The first line of the input contains four integers n, m, a and b ($2 \leq n \leq 70$, $n - 1 \leq m \leq 200$, $1 \leq a < b \leq 10^7$) — the number of nodes and edges in the graph, and two possible travel times. Each of the following lines contains three integers u, v, c ($1 \leq u, v \leq n$, $u \neq v$, $c \in \{a, b\}$) denoting an edge between the nodes u and v, which has cost c.
>
> You can assume that the graph is connected and has no loops or multiedges.
>
> ——Output——
>
> Output a single line containing n integers. The p-th of them should denote the minimum possible post required to travel from 1 to p after the selected edges are abandoned. Note that for each p you can abandon a different set of edges.
>
> ——Examples——
>
> Input
> 5 5 20 25
> 1 2 25
> 2 3 25
> 3 4 20
> 4 5 20
> 5 1 20
>
> Output
> 0 25 60 40 20
>
> Input
> 6 7 13 22
> 1 2 13
> 2 3 13
> 1 4 22
> 3 4 13
> 4 5 13
> 5 6 13
> 6 1 13
>
> Output
> 0 13 26 39 26 13

D Prompt templates

Studio21 Here is our template for Studio21. To see examples of summaries produced by Studio21AI's model along with the code generated for those summaries, refer to Appendix K and L.

> The following sentences contain computer science jargon. Rewrite them using simple words.
> Jargon: <ORIGINAL>
> Simple: <SUMMARY>
>
> Jargon: <ORIGINAL>
> Simple: <SUMMARY>
>
> Jargon: <ORIGINAL>
> Simple: <SUMMARY>
>
> Jargon: <ORIGINAL>
> Simple:

The few-shot examples were chosen randomly from the human generated expert summaries.

GPT3 Here is our template for GPT3. To see examples of summaries produced by GPT3 and the code generated for those summaries refer to Appendix M.

> Summarize the following paragraph: Original: <ORIGINAL>
> Summary: <SUMMARY>
>
> Original: <ORIGINAL>
> Summary: <SUMMARY>
>
> Original: <ORIGINAL>
> Summary: <SUMMARY>
>
> Original: <ORIGINAL> Summary:

Codex Here is our default template for Codex, which is used when there is no starter code provided. When there is starter code provided the docstring remains the same but the code after the doc string will be what is provided.

```
Python3
"""
<PROBLEM DESCRIPTION>
"""
def code():
```

E Strict Accuracy

Strict Accuracy (SAcc) is the percentage of problems that passed every test case. The formula to calculate SAcc is given below:

$$\textit{strict acc} := \frac{\textit{problems with 100\% accuracy}}{\textit{total number of problems}} \quad (1)$$

Given that, we are only generating one code solution for each problem our strict accuracy is comparable to (Chen et al., 2021)'s metric $raw\ pass@1$.

Summary	Difficulty	AP	EWPR	BWPR
Basic	Introductory	145	141	144
	Interview	123	113	123
	Competition	105	100	105
Expert	Introductory	145	140	144
	Interview	123	116	123
	Competition	105	100	105
StudioAI21	Introductory	215	187	-
	Interview	627	558	-
	Competition	659	578	-
GPT3	Introductory	194	180	-
	Interview	266	242	-
	Competition	244	220	-

Table 6: These are the numbers of problems in each split of the dataset. For GPT and Studio21 we did not look at problems that were worse or same for both experiments because there was insignificant overlap between the two experiments.

F Codex Configuration

We did a small test with 75 summaries to find our hyper-parameters for Codex. We set temperature to 0, topP to 1, frequency penalty to 0.2, and presence penalty to 0. We did not provide few-shot examples to Codex since we want to see if summarization only could improve the performance of the Codex model.

G Worst Problems and Statistics

Using the test case labels as defined in section 3 we defined a test case as getting worse if it's label (result) was lower. Then we defined a problem as worse if *every* test case had a lower label. Our methodology behind this was, if we removed problems that had a worse accuracy, then it would be a non-trivial result that accuracy improved. Also, if we removed problems with worse accuracy, then a problem that originally had all 0 labels (all False test cases) would score the same if the summary had all -1 labels (runtime error) or a -2 (syntax error). So, we removed problems which every test case performed worse, to see if removing these outliers would improve results. You can see the overall breakdown of each split in table 6.

H Average length of Problems and Solutions

Table 7 represents the statistics for average length of problems and solutions for original and summarized prompts.

I Abbreviated Synthetic Results

In table 8, we show the results for our synthetic summaries when taking the top 500 and 1000 summaries for GPT3 and StudioAI21, respectively. In our initial experiment, this was the amount of problems we tested for each model. However, in our final experiment we changed our configurations and generated more problems. For a comparison, we took the top performing summaries and and reported those results.

J Generated Code

In figure 3 is the code that was generated for the example mentioned in Appendix C and C.3. Given that the Codex model was prompted with the $def\ code()$: the model did not generate that function definition or the call to that function. That was added in afterwards, but everything inside that function was generated by Codex. The originally generated code (far left) fails with a -1 because it did not take in the input correctly. It added in another line *p = int(input())*, which most likely refers to the p mentioned in the original text. The expert summary generated code (middle) fails every test case. The basic summary generated code (right) passed $16/19\,(84\%)$ test cases and was the only code to pass at least 1 test case.

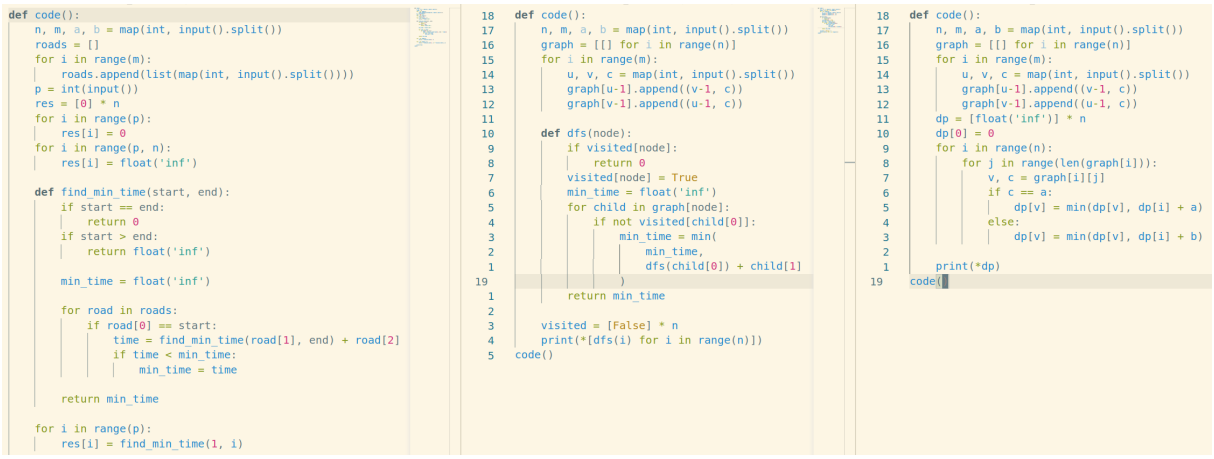
K StudioAI21 Generated Code

Below is an example of a competition problem where StudioAI21 summarized the prompt too much but Codex was still able to produce viable code. Here is the original prompt:

> Cengiz recently learned Fibonacci numbers and now he is studying different algorithms to find them. After getting bored of reading them, he came with his own new type of numbers that he named XORinacci numbers. He defined them as follows: $f(0) = a$; $f(1) = b$; $f(n) = f(n-1) \oplus f(n-2)$ when $n > 1$, where $\oplus$ denotes the bitwise XOR operation.
>
> You are given three integers a, b, and n, calculate $f(n)$.
>
> You have to answer for T independent test cases.
>
> ——Input——
>
> The input contains one or more independent test cases.
>
> The first line of input contains a single integer T ($1 \leq T \leq 10^3$), the number of test cases.
>
> Each of the T following lines contains three space-separated integers a, b, and n ($0 \leq a, b, n \leq 10^9$) respectively.

Experiment	Original Len	Summary Len	Orig Code Len	Summary Code Len	Code Solution Len
Summary	1147	937	339	349	671
Expert	1147	869	339	343	671
GPT	1386	1011	437	392	748
StudioAI21	1646	1114	602	473	721

Table 7: The average length of the original/summarized prompt and generated code. The average length of the code solutions is the average len of the solutions provided by the creators of the APPS dataset. A problem could have one or multiple solutions. The length is reported in characters.



```
def code():
    n, m, a, b = map(int, input().split())
    roads = []
    for i in range(m):
        roads.append(list(map(int, input().split())))
    p = int(input())
    res = [0] * n
    for i in range(p):
        res[i] = 0
    for i in range(p, n):
        res[i] = float('inf')

    def find_min_time(start, end):
        if start == end:
            return 0
        if start > end:
            return float('inf')

        min_time = float('inf')

        for road in roads:
            if road[0] == start:
                time = find_min_time(road[1], end) + road[2]
                if time < min_time:
                    min_time = time

        return min_time

    for i in range(p):
        res[i] = find_min_time(1, i)
```

```
def code():
    n, m, a, b = map(int, input().split())
    graph = [[] for i in range(n)]
    for i in range(m):
        u, v, c = map(int, input().split())
        graph[u-1].append((v-1, c))
        graph[v-1].append((u-1, c))

    def dfs(node):
        if visited[node]:
            return 0
        visited[node] = True
        min_time = float('inf')
        for child in graph[node]:
            if not visited[child[0]]:
                min_time = min(
                    min_time,
                    dfs(child[0]) + child[1]
                    )
        return min_time

    visited = [False] * n
    print(*[dfs(i) for i in range(n)])
code()
```

```
def code():
    n, m, a, b = map(int, input().split())
    graph = [[] for i in range(n)]
    for i in range(m):
        u, v, c = map(int, input().split())
        graph[u-1].append((v-1, c))
        graph[v-1].append((u-1, c))
    dp = [float('inf')] * n
    dp[0] = 0
    for i in range(n):
        for j in range(len(graph[i])):
            v, c = graph[i][j]
            if c == a:
                dp[v] = min(dp[v], dp[i] + a)
            else:
                dp[v] = min(dp[v], dp[i] + b)

    print(*dp)
code()
```

Figure 3: On the far left is the code generated by the original prompt. The middle is the code generated by the expert summary. The right is the code generated by the basic summary.

Model	Difficulty	AP		EWPR	
		Baseline	Proposed	Baseline	Proposed
GPT-3	Introductory	**41.97**	38.86	41.11	**41.67**
	Interview	25.27	**27.47**	24.86	**28.25**
	Competition	4.80	**6.40**	4.88	**6.50**
	Weighted Average	**26.60**	**26.60**	25.83	**27.71**
Studio21	Introductory	**39.91**	31.92	**39.25**	36.56
	Interview	**15.97**	14.50	13.23	**15.47**
	Competition	**2.57**	**2.57**	**2.71**	**2.71**
	Weighted Average	**16.90**	14.50	15.10	**15.64**

Table 8: Results when taking the top 500 GPT problems and top 1000 Studio problems

——Output——

For each test case, output $f(n)$.

——Example——
Input
3
3 4 2
4 5 0
325 265 1231232

Output
7
4
76

——Note——

In the first example, $f(2) = f(0) \oplus f(1) = 3 \oplus 4 = 7$.

Here is the summary that StudioAI21 generated:

You are given three integers a, b, and n. Calculate $f(n)$.

——Input——

The input contains one or more independent test cases.

The first line of input contains a single integer T ($1 \leq T \leq 10^3$), the number of test cases.

Each of the T following lines contains three space-separated integers a, b, and n ($0 \leq a, b, n \leq 10^9$) respectively.

——Output——

For each test case, output $f(n)$.

——Example——
Input
3
3 4 2
4 5 0
325 265 1231232

Output
7
4

76

——Note——

In the first example, $f(2) = f(0) \oplus f(1) = 3 \oplus 4 = 7$.

Because any input/output examples provided by the prompt are appended to the summary, Codex was able to figure out the pattern in the problem and generate code that was almost correct. In figure 4, the solution (left) used the pattern in the problem and simplify by taking $n \mod 3$. The Studio21 summary code (right) recognizes this pattern but erroneously does not take the modulus of the number. The original code (center) also makes the same mistake by not taking the modulus, but also brute forces the answer. This shows that the model did not recognize the pattern in this problem because of the superfluous details. Even though Studio21 might have summarized too much, the model was still able to make an improvement and understand the pattern in the problem more.

L StudioAI21 Generated Code

Here is an example of a summary made by StudioAI21 where the qualitative aspect of the code but it still failed. Here is the original prompt:

Given is a tree G with N vertices. The vertices are numbered 1 through N, and the i-th edge connects Vertex a_i and Vertex b_i. Consider painting the edges in G with some number of colors. We want to paint them so that, for each vertex, the colors of the edges incident to that vertex are all different. Among the colorings satisfying the condition above, construct one that uses the minimum number of colors.

——Constraints——
- $2 \leq N \leq 10^5$
- $1 \leq a_i < b_i \leq N$
- All values in input are integers.
- The given graph is a tree.

——Input——
Input is given from Standard Input in the following format:
N
a_1 b_1
a_2 b_2
$\vdots$
a_{N-1} b_{N-1}

——Output——
Print N lines.
The first line should contain K, the number of colors used.
The (i+1)-th line $(1 \leq i \leq N-1)$ should contain c_i, the integer representing the color of the i-th edge, where $1 \leq c_i \leq K$ must hold.
If there are multiple colorings with the minimum number of colors that satisfy the condition, printing any of them will be accepted.

——Sample Input——
3
1 2
2 3

——Sample Output——
2
1
2

Here is the summary that StudioAI21 generated:

You are given a tree with N vertices numbered from 1 to N. You need to find the minimum possible number of colors to paint the edges such that vertices are painted with different color.
——Constraints——
- $2 \leq N \leq 10^5$
- $1 \leq a_i < b_i \leq N$
- All values in input are integers.
- The given graph is a tree.

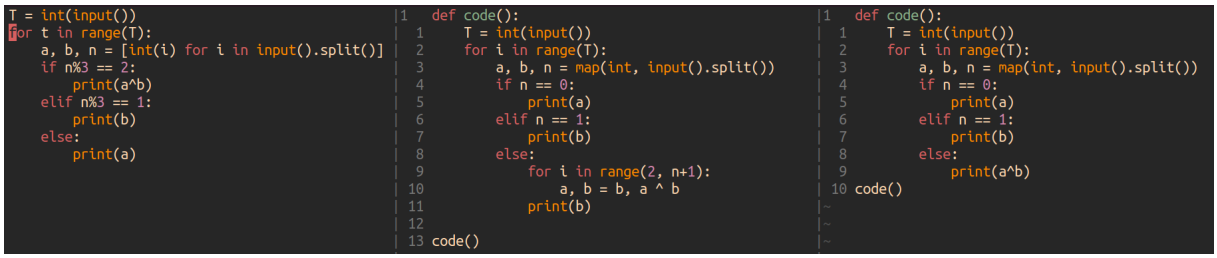
——Input——
Input is given from Standard Input in the following format:
N
a_1 b_1
a_2 b_2
$\vdots$
a_{N-1} b_{N-1}

——Output——
Print N lines.
The first line should contain K, the number of colors used.
The (i+1)-th line $(1 \leq i \leq N-1)$ should contain c_i, the integer representing the color of the i-th edge, where $1 \leq c_i \leq K$ must hold.
If there are multiple colorings with the minimum number of colors that satisfy the condition, printing any of them will be accepted.

——Sample Input——
3
1 2
2 3

——Sample Output——
2
1
2

In 5 the left is the original solution which fails with a -2 because the runtime of the algorithm is exponential. Note that it tries to create a list of all possible edge colorings which is $\mathcal{O}(2^N)$. The right is the code produced when using the StudioAI21 summary. You can see that this code is much closer



```
T = int(input())
for t in range(T):
    a, b, n = [int(i) for i in input().split()]
    if n%3 == 2:
        print(a^b)
    elif n%3 == 1:
        print(b)
    else:
        print(a)
```

```
def code():
    T = int(input())
    for i in range(T):
        a, b, n = map(int, input().split())
        if n == 0:
            print(a)
        elif n == 1:
            print(b)
        else:
            for i in range(2, n+1):
                a, b = b, a ^ b
            print(b)

code()
```

```
def code():
    T = int(input())
    for i in range(T):
        a, b, n = map(int, input().split())
        if n == 0:
            print(a)
        elif n == 1:
            print(b)
        else:
            print(a^b)
code()
```

Figure 4: The left is the code generated using the original prompt. The right is the code generated when using the StudioAI21 generated summary.

to solving the problem and produces an efficient algorithm. However, this fails with a -2 because it tries to print the sum of a boolean (near the end before the last for loop). Which fails in python because a bool is not iterable.

Here is a problem where StudioAI21's summary increased the accuracy to 100%. Here is the original prompt:

> Polycarpus has a sequence, consisting of n non-negative integers: $a_1, a_2, ..., a_n$.
>
> Let's define function f(l, r) (l, r are integer, $1 \leq l \leq r \leq n$) for sequence a as an operation of bitwise OR of all the sequence elements with indexes from l to r. Formally: f(l, r) = $a_l|a_l + 1|...|a_r$.
>
> Polycarpus took a piece of paper and wrote out the values of function f(l, r) for all l, r (l, r are integer, $1 \leq l \leq r \leq n$). Now he wants to know, how many distinct values he's got in the end.
>
> Help Polycarpus, count the number of distinct values of function f(l, r) for the given sequence a.
>
> Expression xly means applying the operation of bitwise OR to numbers x and y. This operation exists in all modern programming languages, for example, in language C++ and Java it is marked as "|", in Pascal — as "or".
>
> ——Input——
>
> The first line contains integer n ($1 \leq n \leq 10^5$) — the number of elements of sequence a. The second line contains n space-separated integers $a_1, a_2, ..., a_n$ ($0 \leq a_i \leq 10^6$) — the elements of sequence a.
>
> ——Output——
>
> Print a single integer — the number of distinct values of function f(l, r) for the given sequence a.
>
> Please, do not use the lld specifier to read or write 64-bit integers in C++. It is preferred to use cin, cout streams or the I64d specifier.
>
> ——Examples——
> Input
> 3
> 1 2 0
>
> Output
> 4
> Input
> 10
> 1 2 3 4 5 6 1 2 9 10
>
> Output
> 11
>
> ——Note——
>
> In the first test case Polycarpus will have 6 numbers written on the paper: f(1, 1) = 1, f(1, 2) = 3, f(1, 3) = 3, f(2, 2) = 2, f(2, 3) = 2, f(3, 3) = 0. There are exactly 4 distinct numbers among them: 0, 1, 2, 3.

Here is the summary that StudioAI21 generated. Not exactly as we would expect as the prompt still mentions the fictional *Polycarpus*.

> Polycarpus has a sequence, consisting of n non-negative integers: $a_1, a_2, ..., a_n$. Let's define function f(l, r) (l, r are integer, 1 l r n) for sequence a as an operation of bitwise OR of all the sequence elements with indexes from l to r. Formally: f(l, r) = $a_l|a_l + 1|...$
>
> ——Input——
>
> The first line contains integer n ($1 \leq n \leq 10^5$) — the number of elements of sequence a. The second line contains n space-separated integers $a_1, a_2, ..., a_n$ ($0 \leq a_i \leq 10^6$) — the elements of sequence a.
>
> ——Output——
>
> Print a single integer — the number of distinct values of function f(l, r) for the given sequence a.
>
> Please, do not use the lld specifier to read or write 64-bit integers in C++. It is preferred to use cin, cout streams or the I64d specifier.
>
> ——Examples——
> Input
> 3
> 1 2 0
>
> Output
> 4
> Input
> 10

```
import itertools
def code():
    n = int(input())
    edges = []
    for i in range(n-1):
        edges.append(list(map(int, input().split())))

    # Create a list of all possible colors
    colors = [i for i in range(1, n)]

    # Create a list of all possible colorings
    colorings = []
    for i in range(1, len(colors)+1):
        colorings += list(itertools.combinations(colors, i))

    # Create a list of all possible trees
    trees = []
    for i in range(len(colorings)):
        trees.append([])
        for j in range(len(edges)):
            if edges[j][0] == 1:
                trees[i].append([edges[j][0], edges[j][1], colorings[i][0]])
            elif edges[j][1] == 1:
                trees[i].append([edges[j][0], edges[j][1], colorings[i][0]])
```

```
def code():
    n = int(input())
    adj = [[] for i in range(n)]
    for i in range(n-1):
        a, b = map(int, input().split())
        adj[a-1].append(b-1)
        adj[b-1].append(a-1)
    colors = [0]*n
    for i in range(n):
        if colors[i] == 0:
            colors[i] = 1
            q = [i]
            while q:
                curr = q.pop()
                for j in adj[curr]:
                    if colors[j] == 0:
                        colors[j] = -colors[curr]
                        q.append(j)
                    elif colors[j] == colors[curr]:
                        print("Impossible")
                        return
    print(sum(colors == 1)+1)
    for i in range(1, n):
        print(colors[i])
code()
```

Figure 5: The left is the code generated using the original prompt. The right is the code generated when using the StudioAI21 generated summary.

1 2 3 4 5 6 1 2 9 10

Output
11

——Note——

In the first test case Polycarpus will have 6 numbers written on the paper: f(1, 1) = 1, f(1, 2) = 3, f(1, 3) = 3, f(2, 2) = 2, f(2, 3) = 2, f(3, 3) = 0. There are exactly 4 distinct numbers among them: 0, 1, 2, 3.

In 6 the left is the original solution which gets 77% accuracy. The right is the summary code which gets 100% accuracy.

M GPT Generated Summaries

Here are two summaries where GPT perfectly summarized the prompt and gave a concise description of what the task was. In both cases the original prompt did not have 100% accuracy but the summarized prompt did have 100% accuracy.

Here is the original prompt for the first question:

Božo is a strange little boy. Every day he tires his friends with strange questions. Today's question is: how many integers in the interval $[A, B]$ are there such that the sum of their digits is S, and which is the smallest such number?

Write a program that answers Božo's question so that he can get some sleep.

——Input——
The input contains three integers A, B and S ($1 \leq A \leq B < 10^{15}$, $1 \leq S \leq 135$).

——Output——
The first line should contain the number of integers in the interval with the digit sum equal to S.

The second line should contain the smallest such integer.

The input data will guarantee that the first number is at least 1.

——Examples——
Sample Input 1:
1 9 5
Sample Output 1:
1
5

Sample Input 2:
1 100 10
Sample Output 2:
9
19

Here is the summary that GPT Generated:

Write a program that calculates the smallest integer in the given interval whose sum of digits is equal to the given sum.

——Input——
The input contains three integers A, B and S ($1 \leq A \leq B < 10^{15}$, $1 \leq S \leq 135$).

——Output——
The first line should contain the number of integers in the interval with the digit sum equal to S.

The second line should contain the smallest such integer.

The input data will guarantee that the first number is at least 1.

——Examples——
Sample Input 1:
1 9 5
Sample Output 1:
1
5

```python
def code():
    n = int(input())
    a = list(map(int, input().split()))
    d = {}
    for i in range(n):
        for j in range(i, n):
            d[a[i] | a[j]] = 1
    print(len(d))
code()
```

```python
def code():
    n = int(input())
    a = list(map(int, input().split()))
    d = {}
    for i in range(n):
        for j in range(i, n):
            s = 0
            for k in range(i, j+1):
                s |= a[k]
            if s not in d:
                d[s] = 1
            else:
                d[s] += 1
    print(len(d))
code()
```

Figure 6: The left is the code generated using the original prompt. The right is the code generated when using the StudioAI21 generated summary.

Sample Input 2:
1 100 10
Sample Output 2:
9
19

In 7 you can see the original code on the left and the summary code on the right. There is a subtle difference but it's that difference that improved the problem from 33% accuracy to 100%.

Here is the original prompt for another question.

> Professor GukiZ makes a new robot. The robot are in the point with coordinates (x_1, y_1) and should go to the point (x_2, y_2). In a single step the robot can change any of its coordinates (maybe both of them) by one (decrease or increase). So the robot can move in one of the 8 directions. Find the minimal number of steps the robot should make to get the finish position.
>
> ——Input——
>
> The first line contains two integers x_1, y_1 $(-10^9 \leq x_1, y_1 \leq 10^9)$ — the start position of the robot.
>
> The second line contains two integers x_2, y_2 $(-10^9 \leq x_2, y_2 \leq 10^9)$ — the finish position of the robot.
>
> ——Output——
>
> Print the only integer d — the minimal number of steps to get the finish position.
>
> ——Examples——
> Input
> 0 0
> 4 5
>
> Output
> 5
>
> Input
> 3 4
> 6 1
>
> Output
> 3
>
> ——Note——
>
> In the first example robot should increase both of its coordinates by one four times, so it will be in position (4, 4). After that robot should simply increase its y coordinate and get the finish position.
>
> In the second example robot should simultaneously increase x coordinate and decrease y coordinate by one three times.

Here is the summary that GPT3 generated:

> The robot can move in one of the 8 directions. Find the minimal number of steps the robot should make to get the finish position.
>
> ——Input——
>
> The first line contains two integers x_1, y_1 $(-10^9 \leq x_1, y_1 \leq 10^9)$ — the start position of the robot.
>
> The second line contains two integers x_2, y_2 $(-10^9 \leq x_2, y_2 \leq 10^9)$ — the finish position of the robot.
>
> ——Output——
>
> Print the only integer d — the minimal number of steps to get the finish position.
>
> ——Examples——
> Input
> 0 0
> 4 5
>
> Output
> 5
>
> Input
> 3 4
> 6 1
>
> Output
> 3

```
def code():
    a, b, s = [int(x) for x in input().split()]
    count = 0
    for i in range(a, b+1):
        if sum([int(x) for x in str(i)]) == s:
            count += 1
            smallest = i
    print(count)
    print(smallest)
code()
```

```
def code():
    a, b, s = map(int, input().split())
    count = 0
    for i in range(a, b+1):
        if sum(map(int, str(i))) == s:
            count += 1
    print(count)
    for i in range(a, b+1):
        if sum(map(int, str(i))) == s:
            print(i)
            break
code()
```

Figure 7: The left is the code generated using the original prompt. The right is the code generated when using the GPT3 generated summary.

——Note——

In the first example robot should increase both of its coordinates by one four times, so it will be in position (4, 4). After that robot should simply increase its y coordinate and get the finish position.

In the second example robot should simultaneously increase x coordinate and decrease y coordinate by one three times.

In 8 you can see the original code on the left and the summary code on the right. There is a subtle difference but it's that difference that improved the problem from 20% accuracy to 100%.

N Human Generated Instructions

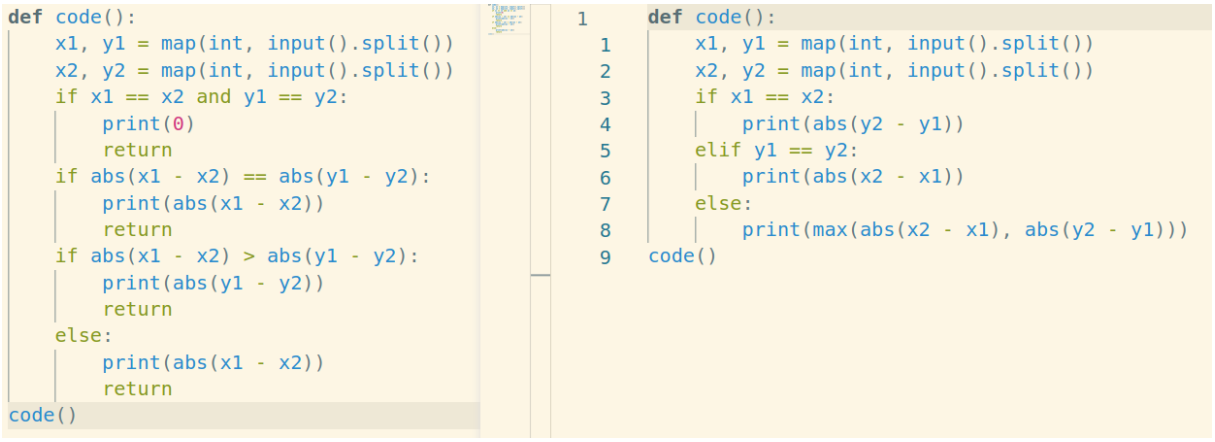
The section below was given to each crowd worker as instructions to follow when creating the regular and expert summaries.

N.1 Summarization

Create a file called $summary.txt$ this will contain your summary of the prompt. It's recommended that you copy the $question.txt$ file into the $summary.txt$ file then starting from the top of the prompt follow the steps and remove words/lines as necessary.

These are the rough steps for making a summary. Following these steps will create the most consistency in our dataset. However, you should summarize as you see fit. First, read through the prompt and understand what it's asking, then follow these steps to help create a summary.

1. Directly state what is given in the problem.
    - Most problems start by setting the scene, to help humans understand.
    - Start the problems by explicitly telling the model what the input is.
    - *You are given . . .*
2. Remove any $notes$ given in the prompt.
    - They are usually reemphasizing points, which is redundant and not needed in the summary.
    - This includes the $-Notes-$ section at the bottom of the file.
    - If there is pertinent information given from a note, include it in the prompt without describing it as a note.
3. Remove any text in parenthesis.
    - Most of the text in parenthesis is repeating the information that precede them.
    - If the text in parenthesis provides more context or information, then remove the preceding text.
    - Keep any parenthesis if it is describing constraints, such as the minimum and maximum values for the input etc...
4. Remove any made up people, places, things, etc...
    - These abstractions are made to help humans understand but confuse the model.
    - The prompts often mention things like $Codefortia$ or $Polycarp$, try to replace these with the word you.
    - Any text visualizing what the problem is asking, should be removed.
5. If the $Input$ or $Output$ section reference an abstraction they should be changed.



```
def code():
    x1, y1 = map(int, input().split())
    x2, y2 = map(int, input().split())
    if x1 == x2 and y1 == y2:
        print(0)
        return
    if abs(x1 - x2) == abs(y1 - y2):
        print(abs(x1 - x2))
        return
    if abs(x1 - x2) > abs(y1 - y2):
        print(abs(y1 - y2))
        return
    else:
        print(abs(x1 - x2))
        return
code()
```

```
def code():
    x1, y1 = map(int, input().split())
    x2, y2 = map(int, input().split())
    if x1 == x2:
        print(abs(y2 - y1))
    elif y1 == y2:
        print(abs(x2 - x1))
    else:
        print(max(abs(x2 - x1), abs(y2 - y1)))
code()
```

Figure 8: The left is the code generated using the original prompt. The right is the code generated when using the GPT3 generated summary.

- Overall, these sections are fine. However, if they mentioned something you removed in the previous steps, they should be changed to reflect that.
- If these sections repeat themselves remove any redundancies.
- In most cases these sections will be left alone.

N.2 Expert Summary

Create a file called $expert.txt$ this will contain an expert summary of the prompt. It's recommended that you copy the $summary.txt$ file into the $expert.txt$ file then starting from the top of the prompt remove words/lines as necessary. You should aim for the expert prompt to be $2-4$ lines.

Imagine you are describing the prompt to a senior software engineer. What else could you trim out? The difference between the original and expert summary, is the original summary may include something obvious, whereas the expert solution should be the absolute bare minimum. To create $summary.txt$ you want to remove superfluous details from the original prompt. To create $expert.txt$ you want to remove details that an expert would find obvious, from the summary.

For example, in problem 2000 (which is competitive difficulty) the summary mentions '*It will be possible to travel between each pair of nodes . . . , and the sum of times . . . will be the minimum possible*'. This process is describing a minimum spanning tree so you can just say '*Find a minimum spanning tree*'.

Also, if the prompt included an example and subsequent explanation, that should remain in the summary but should be removed from the expert summary. An expert already understands the problem and does not need any extra explanation. You should still keep the $-Examples-$ section.

Takeaways

- Removing made up people, places, and things from the prompt improved the quality of code generated.
- The optimal summarization depends on the difficulty of the problem.
- Synthetically generate summaries were close to maintaining accuracy.
- With more rigorous instructions, human summaries could be made with less noise which would further improve synthetic summary generation.

O Superfluous Information Confusing the Model

Here is an example of an interview level string problem where the original prompt got 0% and both human generated summaries got 100% accuracy. The question wants you to write code that will return the number of unique character in the given string.

O.1 Original Prompt

> You have initially a string of N characters, denoted by A1,A2...AN. You have to print the size of the largest subsequence of string A such that all the characters in that subsequence are distinct

ie. no two characters in that subsequence should be same.

A subsequence of string A is a sequence that can be derived from A by deleting some elements and without changing the order of the remaining elements.

——Input—— First line contains T, number of testcases. Each testcase consists of a single string in one line. Each character of the string will be a small alphabet(ie. 'a' to 'z').

——Output—— For each testcase, print the required answer in one line.

——Constraints——
- $1 \leq T \leq 10$
- Subtask 1 (20 points):$1 \leq N \leq 10$
- Subtask 2 (80 points):$1 \leq N \leq 105$

——Example——
Input:
2
abc
aba

Output: 3
2

——Explanation—— For first testcase, the whole string is a subsequence which has all distinct characters.

In second testcase, the we can delete last or first 'a' to get the required subsequence.

O.2 Basic Summary

You are given N string. You have to identify the duplicates and print the length of the new string as a combination of unique characters only.

——Input—— First line contains T, number of testcases. Each testcase consists of a single string in one line. Each character of the string will be a small alphabet(ie. 'a' to 'z').

——Output—— For each testcase, print the required answer in one line.

——Constraints——
- $1 \leq T \leq 10$
- Subtask 1 (20 points):$1 \leq N \leq 10$
- Subtask 2 (80 points):$1 \leq N \leq 105$

——Example——
Input:
2
abc
aba

Output: 3
2

O.3 Expert Summary

You have to remove duplicates and print the length of unique characters of the given string.

——Input—— First line contains T, number of testcases. Each testcase consists of a single string in one line. Each character of the string will be a small alphabet(ie. 'a' to 'z').

——Output—— For each testcase, print the required answer in one line.

——Constraints——
- $1 \leq T \leq 10$
- Subtask 1 (20 points):$1 \leq N \leq 10$
- Subtask 2 (80 points):$1 \leq N \leq 105$

——Example——
Input:
2
abc
aba

Output: 3
2

O.4 Generated Code

The original code (left) does not accomplish the task but rather prints the count of the most frequent character. The model was unable to distinguish what the task was given the verbose prompt. However, the basic and expert summaries make the task clear and the model produces the same code. Which properly solves the challenge.

P Made Up Information Confusing the Model

Here is an example of an interview level problem where the original prompt got 0% and the expert generated summary got 100% accuracy.

P.1 Original Prompt

The chef was searching for his pen in the garage but he found his old machine with a display and some numbers on it. If some numbers entered then some different output occurs on the display. Chef wants to crack the algorithm that the machine is following. Example to identify the pattern :

Input	Output
9	36
5	10
1	0
2	1

——Input:——
- First-line will contain T, the number of test cases.

```
def code():
    t=int(input())
    for i in range(t):
        s=input()
        l=len(s)
        if l==1:
            print(1)
            continue
        d={}
        for i in s:
            if i not in d:
                d[i]=1
            else:
                d[i]+=1

        max_val=0

        for i in d.values():
            if i>max_val:
                max_val=i

        print(max_val)
code()
```

```
def code():
    t = int(input())
    for i in range(t):
        s = input()
        l = []
        for j in s:
            if j not in l:
                l.append(j)
        print(len(l))
code()
```

```
def code():
    t = int(input())
    for i in range(t):
        s = input()
        l = []
        for j in s:
            if j not in l:
                l.append(j)
        print(len(l))
code()
```

Figure 9: The left is the code generated by the original prompt. The middle is the code generated by the expert summary. The right is the code generated by the basic summary.

Then the test cases follow. - Each test case contains a single line of input, N.

——Output:——
For each test case, output in a single line answer as displayed on the screen.

——Constraints——
- $1 \leq T \leq 10^6$
- $1 \leq N \leq 10^6$

——Sample Input:——
1
7

——Sample Output:——
21

P.2 Expert Summary

Write a code to print the average of the multiplication of a given number N with N-1 integer.
1

——Input:——
- First-line will contain T, the number of test cases. Then the test cases follow. - Each test case contains a single line of input, N.

——Output:——
For each test case, output in a single line answer as displayed on the screen.

——Constraints——
- $1 \leq T \leq 10^6$
- $1 \leq N \leq 10^6$

——Sample Input:——
1
7

——Sample Output:——
21

```
def code():
    T = int(input())
    for i in range(T):
        N = int(input())
        print((N*(N-1))//2)
code()
```

```
def code():
    T = int(input())
    for i in range(T):
        N = int(input())
        if N == 0:
            print(0)
        elif N == 1:
            print(1)
        elif N == 2:
            print(1)
        else:
            sum = 0
            for j in range(N):
                if j % 2 == 0:  # even numbers are 1's and odd numbers are 0's. So, the sum of all the digits is the answer.
                    sum += 1

                else:  # odd numbers are 1's and even numbers are 0's. So, the sum of all the digits is the answer.

                    sum += 0

            print(sum)
code()
```

Figure 10: The left is the code generated by the expert summary. The right is the code generated by the original prompt.