Code prompting: A Systematic Study of How to Improve Program-based Prompting for Large Language Model Reasoning

Anonymous ACL submission

Abstract

Large language models (LLMs) have scaled up to unlock a wide range of complex reasoning tasks with the aid of various prompting methods. However, previous prompting 005 methods generate natural language intermediate steps to help reasoning, which can cause imperfect task reduction and confusion due to 007 the ambiguity and sequential nature of natural language. To mitigate such limitations, Gao et al. (2023); Chen et al. (2022) have proposed program-based prompting, triggering code as 011 intermediate steps. In this paper, we perform a systematic study of the approach which we refer to as "code prompting". We conduct experiments on both symbolic and arithmetic reasoning datasets regarding both zero-shot/few-shot scenarios, whether to employ an external inter-017 preter for code execution or use the LLM itself 018 instead, and auxiliary prompting techniques to 019 facilitate reasoning including "self-debugging", "comments", "equation instruction" and "elimination of irrelevant information". To further understand the performance and limitations of code prompting, we perform extensive ablation studies and error analyses. We also consider the ensemble of code prompting and CoT prompting to combine the strengths of both. 027

1 Introduction

Recent years have seen huge revolutions in the field of Natural Language Processing regarding the shockingly fast development of large language models (LLMs) (Brown et al., 2020; OpenAI, 2023; Touvron et al., 2023; Zhang et al., 2022a; Thoppilan et al., 2022; Chowdhery et al., 2022; Raffel et al., 2020). According to Qin et al. (2023); OpenAI (2023), LLMs have shown certain levels of the ability to solve complex reasoning tasks, with the scaling up for the model size (Kaplan et al., 2020). Besides the size of LLMs, how to prompt is crucial to the reasoning ability. A large number of works have proposed different prompting methods to facilitate LLM reasoning by generating natural language intermediate steps before the final answer and have enhanced the reasoning ability of LLMs to a great extent (Wei et al., 2023; Zhang et al., 2022b; Kojima et al., 2023; Zhou et al., 2023; Fu et al., 2023; Khot et al., 2023; Press et al., 2022). However, there still remain limitations. For example, they may fail to plan the whole process of solution ahead, which is likely to result in unreasonable task reduction. In other words, the "step" can be too big for the LLM to stride over. In Figure 1 (top), we show a case where the LLM is supposed to concatenate the last letters of the given words. CoT prompting leads the LLM to (1) first extract the last letter of each word and (2) then concatenate all the letters together. However, the second step may be too difficult for the LLM with the number of letters increasing, resulting in wrong answers (Zhou et al., 2023).

041

042

043

044

045

047

049

052

053

055

059

060

061

062

063

064

065

066

067

068

069

070

071

072

073

074

075

077

078

081

To overcome the difficulties, recent works (Gao et al., 2023; Chen et al., 2022; Cheng et al., 2023) have tried to facilitate complex reasoning in LLMs with code as intermediate steps, which we refer to as "code prompting" in the rest of the paper. Code prompting is generally a two-stage method with the pipeline shown in the bottom of Figure 1, where LLMs first generate a piece of code, and then 1) offload the solution process to an external interpreter or 2) follow the code to generate the final answer by themselves. Gao et al. (2023); Chen et al. (2022) have pointed out that offloading the second stage to an interpreter greatly enhances the LLM reasoning performance. In this paper, we also study the potential of LLMs to "execute" the code generated in the second stage. As a generalpurpose tool, code is machine-executable, LLMproducible, and can describe flexible computation processes. Code can work as a mind map for the LLM and reduce the task into sub-tasks represented by separate operations in the code. As the separate



Figure 1: The pipelines of zero-shot CoT prompting (**above**) and zero-shot code prompting (**below**). Left: input of LLM (prompt + previous generation), right: generation by LLM. Texts highlighted in orange are instructions; texts highlighted in blue are the code generated by the LLM.

operations in code are often easy for LLMs, task reduction greatly facilitates LLMs to solve complex reasoning problems step by step by themselves.

Previous works have initially demonstrated the power of code prompting, but there are still some problems: for example, for which tasks it is more suitable, how to improve its performance on specific problems, etc. In this paper, we follow the works and further explore the code prompting method. We perform a systematic study of code prompting from the following perspectives: (1) tasks calling for different reasoning abilities, (2) zero-shot or few-shot prompting, (3) the potential of self-contained code executing ability of LLM, and (4) some auxiliary prompting tricks including self-debugging, writing comments, etc.

We experiment on both symbolic and arithmetic reasoning datasets and show that while code prompting generally outperforms CoT prompting, the performance gap in symbolic ones is much greater than in arithmetic ones. That is, code performs better on symbolic reasoning tasks. Besides, we investigate into both zero-shot and fewshot settings, showing that zero-shot code prompting with auxiliary prompting tricks is competitive with current few-shot methods, including few-shot CoT (Wei et al., 2023) and PAL (Gao et al., 2023). Moreover, besides calling an external Python Interpreter to execute the code like previous work, we provide another option of letting the LLM itself to generate the final answer according to the code step by step. We show that even without the executor, code prompting still matches or even exceeds CoT prompting in both symbolic and arithmetic tasks. For the feasibility of auxiliary prompting tricks, we dig into the failure cases of code prompting and identify several key limitations of code prompting, based on which we propose customized extensions to enhance its performance, including selfdebugging, comments, elimination of irrelevant information and equation instruction. We conduct detailed ablation study to show their effects. Moreover, we find out that code prompting and CoT prompting lead the LLM to think from different angles, suggesting a combination of both methods, which achieves 87.95% accuracy (+6.37% from few-shot CoT) on GSM8K.

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

Overall, in this paper, we systematically study the reasoning ability of code prompting across different settings and tasks. The key contributions over previous works can be summarized as follows:

LLM self-contained. Our work for the first

082

time explored the self-contained reasoning ability of LLMs based on code prompts, while previous works all demonstrate the power of code prompting based on code interpreters.

> **Performance.** We improve both zero- and fewshot abilities of code prompting. We first show that zero-shot code prompting could match few-shot methods like few-shot CoT (Wei et al., 2023) and PAL (Gao et al., 2023).

> Auxiliary prompting tricks. We identify general limitations of code prompting through detailed error analyses. Besides, we show that code prompting can be enhanced by simple auxiliary prompting techniques specific to the limitations.

> **Ensemble.** We explore the ensemble of code prompting and CoT prompting to combine the strengths of both, showing great performance gain.

2 Related Work

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

158

159

160

162

163

164

165

166

167

168

170

171

172

173

174

175

176

177

178

179

180

181

183

Prompting. Various prompting methods have emerged to boost LLM reasoning (Wei et al., 2023; Zhang et al., 2022b; Kojima et al., 2023; Zhou et al., 2023; Fu et al., 2023; Khot et al., 2023; Press et al., 2022). Classified based on whether specific exemplars are provided, prompting methods can be divided into zero-shot prompting and few-shot prompting. For example, CoT prompting has a zero-shot version (Kojima et al., 2023) and a few-shot version (Wei et al., 2023). Fewshot prompting calls for manually constructed taskspecific demonstrations, so it is more costly. In this paper, we investigate in both zero-shot and few-shot versions of code prompting. Existing prompting methods can also be divided into singlestep prompting and multi-step prompting methods. CoT prompting (Kojima et al., 2023; Wei et al., 2023) is a classic single-step prompting method. Least-to-most prompting introduced by Zhou et al. (2023) is a multi-step prompting method, which first divides a question into several sub-questions and then guides the LLM to solve each question sequentially. Despite its strong performance, leastto-most prompting is limited to a few-shot version and requires hand-crafting of task-specific demonstrations. Thus, it is less general than CoT and code prompting.

Program-aided reasoning. Prior works have shown that programs generated by LLMs may facilitate reasoning (Gao et al., 2023; Chen et al., 2022; Chowdhery et al., 2022). We follow the work and dig deeper into code prompting: 1) Gao

et al. (2023) (PAL) proposed a few-shot prompting 184 method, while we investigate into both zero-shot 185 and few-shot code prompting. We show that zero-186 shot code prompting matches or even exceeds PAL 187 in arithmetic reasoning tasks. Besides, we improve 188 the few-shot performance too. Although (Chen 189 et al., 2022) also studied the zero-shot setting, its 190 performance is still far from the few-shot method 191 without auxiliary prompting methods we propose 192 in §5. 2) We offer another option on the second 193 stage of final answer generation, namely directly 194 asking LLM to generate the final answer according 195 to the code, other than calling a Python interpreter 196 as in Gao et al. (2023); Chen et al. (2022). Our 197 experiments on using LLM in the second stage 198 further prove that code prompting indeed assists 199 reasoning in LLM even without executing the code. 200 Although Gao et al. (2023) also conducted experi-201 ments without an external interpreter, they instruct 202 the LLM to generate the answer directly after gen-203 erating the code. Instead, we instruct the LLM to 204 follow the code step by step to generate the final 205 answer and boost the performance to a large extent. 206 3) We discuss some key limitations and insights of 207 code prompting through error analysis, based on 208 which we propose several highly useful extensions. 209 Besides, they help us understand the power of code 210 prompting better and motivate us to combine CoT 211 and code prompting. 212

LLMs with external tools. Code prompting has an option to call a Python interpreter as an external tool to assist the LLM to complete the tasks. The concept of augmenting LLMs with external tools has drawn much attention (Khot et al., 2023; Cheng et al., 2023; Press et al., 2022). A similar work (Cheng et al., 2023) also uses programs to assist LLM reasoning. However, they focus on generating SQL or SQL-like programs to deal with questions of reasoning with tables. Besides, to improve the code generation, a recent work (Chen et al., 2023) introduces a method of instructing LLMs to debug their generated program with the help of feedback from a code interpreter. In our work, we equip code prompting with a similar technique, which we call "self-debug" in the rest of the paper. However, we consider reasoning tasks while Chen et al. (2023) focuses on tasks of text-to-code or code-to-code generation.

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

325

327

328

279

3 Code Prompting

235

239

240

242

244

245

246

247

248

249

253

254

255

258

259

263

269

270

271

272

273

275

276

Code prompting guides LLMs to solve a complex problem in two stages:

 Code Generation. In the first stage, the prompt asks the LLM to generate Python code to solve the problem. We test on both zero-shot and fewshot prompting. The zero-shot prompt is simply "Generate python code to { task description }.". The few-shot prompt consists of exemplars of questions followed by corresponding code.

 Based-on-code Solution. The second stage is for solving the problem given the code generated in the first stage. We compare two different methods: 1) LLM self-contained: we ask the LLM to generate step-by-step solution following the code. 2) Interpreter: we directly call a Python interpreter to execute the code.

The settings in these two stages are combinatorial so that there are four different pipelines, which we will all compare in §4. An example of the pipeline in the task of last letter concatenation (Wei et al., 2023) is shown in Figure 1, where the zeroshot code prompt and the LLM self-contained solution are used. In the first stage, we use the zero-shot prompt to guide the LLM to output a piece of code for the task. In the second stage, the LLM is given the generated code along with a concrete question. Here we ask the LLM to stimulate code execution step by step.

Self-debugging The "self-debug" technique can be used to improve code generation ability of LLMs (Chen et al., 2023). It is a general technique to improve code generation and has become a default setting in GPT-4. Here, if we use a Python interpreter in the second stage, we can feed the generated code (with bugs) and the bug report back to the LLM to fix the bugs by the LLM itself. We adapt the technique to help LLMs to generate executable code. The pipeline of the "self-debugging" module is shown in Figure 2.

4 Experiments

4.1 Setup

Tasks

We conduct experiments on 7 popular datasets involving both symbolic and arithmetic reasoning.

For **symbolic reasoning**, we consider two tasks introduced by Wei et al. (2023) and widely used

in Kojima et al. (2023); Zhou et al. (2023); Zhang et al. (2022b):

Last letter concatenation. The task asks LLMs to concatenate the last letters of given words. We follow Zhou et al. (2023) to construct word lists by randomly selecting words from five thousand words of the Wikipedia frequency list. We construct word lists of lengths 4, 8 and 12. For each length, we test the prompting methods on 500 word lists, which form a test dataset of 1,500 samples.

Coin flip. The task requires LLMs to answer whether a coin is still heads up after several people flipped or did not flip it. The number of people varies from 3 to 5. For each certain number of people, we construct 500 questions, which form a test dataset of 1,500 samples.

For **arithmetic reasoning**, we consider five commonly used datasets: (1) SingleEq (Koncel-Kedziorski et al., 2015), (2) AddSub (Hosseini et al., 2014), (3) MultiArith (Roy and Roth, 2016), (4) SVAMP (Patel et al., 2021), (5) GSM8K (Cobbe et al., 2021). Among the datasets, SingleEq and AddSub only take single-step calculation, while MultiArith, GSM8K and SVAMP contain harder math problems that require multi-step reasoning. See Appendix A.2 for more details of each dataset.

Baselines

In symbolic reasoning tasks, we consider zero-shot standard prompting (only the question), zero-shot and few-shot CoT prompting (Kojima et al., 2023; Wei et al., 2023) as baselines. For code prompting methods, we only consider zero-shot code prompting because the "groundtruth" code for each question is basically the same so that the exemplars in few-shot code prompting may leak the answer.

In arithmetic reasoning tasks, we consider zeroshot and few-shot CoT prompting (Kojima et al., 2023; Wei et al., 2023) as the baseline for zero-shot and few-shot code prompting.

For all the prompt methods, we use the model gpt-3.5-0301 and set the temperature to 0 unless otherwise specified.

Methods

As is shown in Figure 1, the method prompts the LLM to first generate task-specific code and then follow the code to generate the final answer. For the first stage, we study both the **zero-shot** and **few-shot** prompting. For the second stage, we provide two options for how to generate the final answer: + LLM Self-contained: we use the LLM itself



0	· F F ·		88	0	
	last le	tter coi	ncatenation		coir
	l=4	l=8	l=12	flips=3	flip

	last le	etter con	catenation		coin flip	
	l=4	l=8	l=12	flips=3	flips=4	flips=5
zero-shot standard	7.4	2.0	2.6	22.6	17.2	16.8
zero-shot CoT	71.0	27.8	2.8	86.2	71.8	67.8
few-shot CoT	94.6	69.6	39.8	99.8	99.8	<u>99.0</u>
zero-shot code + LLM self-contained	97.2	85.4	75.6	86.2	88.8	85.8
zero-shot code + interpreter	99.4	99.8	99.8	99.8	99.8	99.4

Table 1: The accuracy (%) of different prompting methods on symbolic reasoning tasks. The number of words to concatenate (ranging from 4 to 12 with an interval of 4) and the number of flips (ranging from 3 to 5) are listed.

to perform reasoning; + Interpreter: we employ a Python interpreter to execute the code and take the output as the answer. We list all the prompt mentioned above in Appendix B (symbolic tasks) and Appendix D (arithmetic tasks).

329

330

332

336

338

341

342

347

350

Symbolic reasoning or arithmetic 4.2 reasoning

Our first observation is that the **code prompting** performs better in tasks involving symbolic reasoning. For symbolic reasoning, as shown in Table 1, even without a Python interpreter, code prompting greatly outperforms CoT prompting in the zero-shot setting, while the interpreter can further enlarge the gap. Zero-shot code prompting performance can match or even surpass the fewshot CoT. On the contrary, code prompting and CoT prompting generally show comparable performance in arithmetic reasoning, as shown in Table 2.

Moreover, in the symbolic reasoning, the performance gain increases with the complexity of the questions, implying that code prompting generalizes better than CoT. The performance of CoT

prompting decreases rapidly as the task becomes harder (e.g., longer word lists or bigger number of flips), while code prompting helps the LLM to complete tasks of various difficulties. The performance gain grows from 26.2% to 72.8% in last letter concatenation as the length of word lists increases from 4 to 12, and in the task of coin flip, the performance gain rises from 0.0% to 18.0% with flip times increasing from 3 to 5. However, we do not observe the same phenomenon in arithmetic reasoning tasks. Instead, code prompting performs worse than CoT when the tasks become harder. Shown in Table 2, in more easier (single-step) datasets SingleEQ and AddSub, the code prompting can ourperform CoT slightly, while in harder datasets MultiArith and GSM8k, CoT prompting is better. We also experiment on questions in GSM8K with various difficulties in Figure 4, where the gap between CoT and code prompt is larger in harder questions.

351

352

353

354

356

357

358

360

361

362

363

364

366

367

369

370

We summarize the reason behind the outstanding 371 performance of code prompting in symbolic tasks as abstraction and simplification. In both symbolic 373

SingleEq	AddSub	SVAMP	MultiArith	GSM8K
93.11 97.64	86.08 89.62	78.20 79 40	96.67 96.67	81.58 77.40
95.47	90.63	77.10	98.50	82.11
95.07	88.86	75.60	96.00	73.84
97.64 97.64	89.11 90.13	79.50 79.60	97.00 97.00	79.45 79.90
	SingleEq 93.11 97.64 95.47 95.07 97.64 97.64	SingleEq AddSub 93.11 86.08 97.64 89.62 95.47 90.63 95.07 88.86 97.64 89.11 97.64 90.13	SingleEq AddSub SVAMP 93.11 86.08 78.20 97.64 89.62 79.40 95.47 90.63 77.10 95.07 88.86 75.60 97.64 89.11 79.50 97.64 90.13 79.60	SingleEq AddSub SVAMP MultiArith 93.11 86.08 78.20 96.67 97.64 89.62 79.40 96.67 95.47 90.63 77.10 98.50 95.07 88.86 75.60 96.00 97.64 89.11 79.50 97.00 97.64 90.13 79.60 97.00

Table 2: The accuracy (%) of different prompting methods on arithmetic reasoning.

374tasks, code prompting endows the LLM the abil-375ity to extract the "loop" nature of the question and376leverages it explicitly in the code using "for" or377"while" syntax in Python language. In other words,378code simplifies the solution. However, arithmetic379tasks do not have such features that may help simplify the solution.

4.3 Zero-shot or few-shot

381

390

396

400

401

402

403

404

405 406

407

408

409

410

In symbolic reasoning, zero-shot code (+ interpreter) outperforms few-shot CoT prompting in both tasks, and zero-shot code (+ LLM selfcontained) performs better in last letter concatenation. Overall, **zero-shot code prompting is highly competitive with few-shot prompting methods in symbolic reasoning**.

In arithmetic reasoning tasks, the performance gap between zero-shot and few-shot CoT prompting is wider than that between zero-shot and fewshot code prompting, reflecting that **code prompting is less sensitive to the number of exemplars**. This is crucial since few-shot scenarios call for the handcraft of task-specific demonstrations, and code prompting handles this with less sensitivity to zero-shot or few-shot settings.

4.4 With or without interpreter

In Gao et al. (2023), the authors state that LLMs (specifically, Codex (Chen et al., 2021)) show little ability to return correct results of a piece of code. However, as show in Table 3, we observe that the **self-contained code executing ability of LLM can be stimulated** by prompting the LLM "*run*" the code *step by step* instead of directly asking for the final answer right after the code as in Gao et al. (2023). Although LLMs show code executing ability in both symbolic and arithmetic reasoning tasks, the external Python interpreter greatly boosts the performance of code prompting.

Model Method	Codex	ChatGPT
LLM direct	23.2	43.8
LLM step-by-step	/	73.8
interpreter	72.0	79.5

Table 3: GSM8K accuracy (%) of few-shot prompting methods including *LLM direct* (instruct the LLM to directly generate the final answer after the code in one step), *LLM step-by-step* (i.e. code + LLM self-contained) and *code* + *interpreter* adapted on Codex (Chen et al., 2021) and ChatGPT. Data of Codex is from Gao et al. (2023).



Figure 3: Accuracy of zero-shot and few-shot code prompting w/ or w/o "self-debugging" on arithmetic datasets.

4.5 With or without self-debugging

We study the effects of "self-debugging" described in §3. Figure 3 shows the performance of code prompting with or without self-debugging. This technique can indeed improve the reasoning accuracy, especially for the harder questions. **The performance on harder benchmarks is more likely to benefit from self-debugging**, as more complex problems are more likely to trigger bugs in code. 411

412

413

414

415

416

417

418

419

420

5 Auxiliary Prompting Tricks

In this section, we point out some general limi-
tations of code prompting through detailed error
analyses, including sensitivity to irrelevant infor-
mation and lack of the ability to solve equations.422
423
424Based on the limitations, we propose several sim-
ple yet helpful auxiliary prompting tricks and show
their feasibility through ablation studies.421
422

Method	SingleEq	AddSub	MultiArith	SVAMP	GSM8K
W/o comments (PAL)	97.64	89.11	97.00	79.50	79.45
W/ comments (end)	97.44	89.87	97.33	79.60	80.21
W/ comments (beginning)	96.85	89.87	95.33	79.90	77.71

Table 4: Accuracy (%) of methods of *few-shot code prompting* w/o comments or w/ comments added at different locations of each line of the code. "Self-debugging" is not used here.



Figure 4: Accuracy of few-shot code prompting w/ or w/o comments and CoT prompting on questions with various difficulties. The left figure use the number of steps of the rationales as the metric; the right one use the number of words of the rationales as the metric.

5.1 Comments

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

Comments in code are informative natural language that may help to hint at the semantics and the role of a certain part of the code. So we investigate into the effects of comments in code prompting. As shown in Table 4, adding comments slightly improves the performance.

Does where we add comments matter? We experiment on code prompting with comments added either at the beginning or at the end of each line of code. As shown in Table 4, in most cases, adding comments at the end is a better choice. This may be due to humans' habit of writing code. We tend to add comments specifically describing each line of code at the end of each line, thus the training corpus contains more code samples with comments at the end of each line.

Do comments help harder or easier questions? We experiment on GSM8K to find out where the performance gain of adding comments comes from. We consider the number of steps and the number of words in the provided answer as two notions of difficulty. According to Figure 4, the performance gain on the hardest questions are the largest.

Method	AddSub
zero-shot code	89.62
zero-shot code+irr	91.65
few-shot code	89.87
few-shot code+irr	91.39

Table 5: Accuracy (%) of code prompting w/ or w/o "irrelevant information" on AddSub.

Method	GSM8K
zero-shot code	77.40
zero-shot code _{+equ(comments)}	78.09
zero-shot code _{+equ(sympy)}	78.92

Table 6: Accuracy (%) of code prompting w/ or w/o "equation instruction" on GSM8K.

Method	GSM8K
few-shot CoT	81.58
few-shot code	79.68
CoT vote	87.49
code vote	83.85
CoT + code vote	87.95

Table 7: Accuracy (%) of ensemble methods on GSM8K.

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

474

475

476

477

478

479

480

5.2 Elimination of irrelevant information

In AddSub, the generated code tend to define irrelevant variables that appear in the question but are not useful for the final result, thus disturbing the reasoning process. See Appendix E for examples. The observation suggests code generation by LLMs can be easily distracted by irrelevant information. . This motivates elimination of irrelevant information (+irr). We add an instruction "*There may be irrelevant information in the question. If you find it, ignore it.*" at the end of the prompt. Table 5 shows that despite the simplicity, it effectively improves the performance.

5.3 Equation instruction

In GSM8K, the LLM struggles to solve equations in code. When facing an equation, the LLM tends to solve it in the comments or directly lists the equation in the code, which may easily result in mistakes. See examples in Appendix E. To mitigate such limitations, we provide an instruction on solving equations in Python using the package sympy. See Appendix D for more details. We also experiment on directly asking the LLM to solve equations in the comments. Table 6 shows that the best choice is to teach the LLM to use certain Python packages to solve equations.

5.4 Ensemble of CoT and code prompting

In GSM8K, we find out through statistics that the error overlap of code prompting and CoT prompt-

ing is very small, suggesting that the two prompting methods lead LLMs to think from different angles (Appendix E). So we consider the ensemble of CoT prompting and code prompting based on voting. Here we also adapt the "comments" and "equation" tricks for code prompting. For each question, if both prompting methods generate the same answer, we accept the answer as the final answer; otherwise, we set the temperature to 0.7 and ask the LLM to generate n answers following each prompting method. Then we vote among the 2nanswers to give the final answer. We also compare voting methods whose 2n answers are generated from the same prompting method. As shown in Table 7, ensemble methods outperform the baselines significantly. Further, the ensemble of two prompting methods surpasses that of only one prompting method votes. See Appendix D for details.

481

482

483

484

485

486

487

488

489

490

491

492

493

494

495

496

497

498

499

503

504

507



Figure 5: Answer distribution on questions w/ and w/o ambiguity.

5.5 Conclusion

Above, we propose some simple auxiliary tricks which can improve the performance of code prompting further. The "comments" and "ensemble" tricks are independent of the specific task and are always available as an option. The "irrelevant information" and "equation" tricks are a kind of targeted remedies, requiring an analysis of the mistakes in specific tasks and the provision of targeted prompts accordingly. However, although the specific prompts vary, the effectiveness of this kind of feedback is generally moderate, demonstrating the room for further improvement in code prompting. 508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

6 Discussion

Disambiguation. One of the most different property between code and natural language is the ambiguity, where the semantic of code is generally clearer than natural language.

On one hand, the general semantic of words in CoT (which could be different from this current semantic usage of the word) sometimes mislead the steps, while the code can avoid the risk by giving clearer instructions. See examples in Appendix C.

On the other hand, code prompting can detect some confusing or imprecise expressions in questions. We find out that code prompting is more sensitive to ambiguity in the question. See Appendix E for examples in MultiArith. This indicates that code prompting has the potential to discover ambiguity in a question. Here we use 5 cases where the questions present ambiguities and we can manually fix them. We test few-shot code prompting and CoT prompting on both questions with and without ambiguity. For each question, we generate 15 answers by code prompting and CoT prompting respectively with the temperature of the LLM set to 0.7. Figure 5 shows the histogram of answers. It is evident that ambiguity disturbs code prompting, while CoT prompting is less sensitive to ambiguity. We may leverage this feature to detect ambiguity in questions.

7 Conclusion

We study code prompting systematically. We conduct comprehensive experiments on 7 benchmarks involving both symbolic and arithmetic reasoning. For tasks calling for different reasoning abilities, we show the advantage of code prompting lies mostly in symbolic tasks instead of arithmetic tasks. Besides, we first show that zero-shot code prompting matches few-shot methods with the aid of auxiliary prompting tricks. Moreover, we explore the potential of self-contained code executing ability of LLMs for the first time. Finally, extensive experiments and analyses verify the effectiveness of our auxiliary prompting tricks including selfdebugging, comments, equation instruction and elimination of irrelevant information. 556

57

8 Limitations

We only investigate in a specific programming
language Python and a specific language model
ChatGPT. The comparison between multiple programming languages and between various language
models are left for future work.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. 562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

595

596

597

598

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug.
- Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding language models in symbolic languages.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi,

698

699

700

701

702

703

704

705

706

707

708

709

710

711

712

713

714

715

716

717

718

719

720

721

722

723

724

725

726

727

728

729

David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

621

634

643

648

651

657

661

667

671

- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.
 - Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2023. Complexity-based prompting for multi-step reasoning.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models.
- Mohammad Javad Hosseini, Hannaneh Hajishirzi, Oren Etzioni, and Nate Kushman. 2014. Learning to solve arithmetic word problems with verb categorization. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 523–533, Doha, Qatar. Association for Computational Linguistics.
- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.
- Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal.
 2023. Decomposed prompting: A modular approach for solving complex tasks.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large language models are zero-shot reasoners.
- Rik Koncel-Kedziorski, Hannaneh Hajishirzi, Ashish Sabharwal, Oren Etzioni, and Siena Dumas Ang. 2015. Parsing algebraic word problems into equations. *Transactions of the Association for Computational Linguistics*, 3:585–597.
- Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. 2016. MAWPS: A math word problem repository. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1152–1157, San Diego, California. Association for Computational Linguistics.
- 673 OpenAI. 2023. Gpt-4 technical report.

- Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems?
- Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A. Smith, and Mike Lewis. 2022. Measuring and narrowing the compositionality gap in language models.
- Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. 2023. Is chatgpt a general-purpose natural language processing task solver?
- Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer.
- Subhro Roy and Dan Roth. 2016. Solving general arithmetic word problems.
- Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. 2022. Lamda: Language models for dialog applications.
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.
- Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022a. Opt: Open pre-trained transformer language models.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022b. Automatic chain of thought prompting in large language models.

730

731

732

733

734 735

736

737

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-most prompting enables complex reasoning in large language models.

A Datasets

A.1 Symbolic reasoning

dataset		#samples	example problem
last letter concatenation	1,500	500 (4 words) 500 (8 words)	"fully, drug, gut, agreement" "urge, participate, strategic, rise, win, through, now, within"
		500 (12 words)	"elementary, consensus, plaza, mes- sage, telescope, accessible, another, transport, bubble, bizarre, adviser, cow"
coin flip	1,500	500 (3 flips)	A coin is heads up. Taylor doesn't flip the coin. Harmon doesn't flip the coin. Dejesus doesn't flip the coin. Is the coin still heads up?
		500 (4 flips)	A coin is heads up. Nichols flips the coin. Mcbride flips the coin. Mathis doesn't flip the coin. Ruiz flips the coin. Is the coin still heads up?
		500 (5 flips)	A coin is heads up. Weber flips the coin. Contreras flips the coin. Carter flips the coin. Moran flips the coin. Colon doesn't flip the coin. Is the coin still heads up?

A.2 Arithmetic reasoning

dataset	#samples	example problem
SingleEq	508	Joan found 70 seashells on the beach. she gave Sam some of her seashells. She has 27 seashell left. How many seashells did she give to Sam ?
AddSub	395	There are 22 walnut trees currently in the park. Park work- ers will plant walnut trees today. When the workers are finished there will be 55 walnut trees in the park. How many walnut trees did the workers plant today?
MultiArith	600	For Halloween Debby and her sister combined the candy they received. Debby had 32 pieces of candy while her sister had 42. If they ate 35 pieces the first night, how many pieces do they have left?
SVAMP	1,000	Julia played tag with 18 kids on monday. She played tag with 10 kids on tuesday. How many more kids did she play with on monday than on tuesday ?
GSM8K	1,319	Janet's ducks lay 16 eggs per day. She eats three for break- fast every morning and bakes muffins for her friends every day with four. She sells the remainder at the farmers' mar- ket daily for \$2 per fresh duck egg. How much in dollars does she make every day at the farmers' market?

738 739

We list the details and licenses for arithmetic datasets we used. The symbolic datasets were newly created.	741
• SingleEq (Koncel-Kedziorski et al. (2015)); AddSub (Hosseini et al. (2014)): https://www.cs. washington.edu/nlp/arithmetic; MultiArith (Roy and Roth (2016)) from Math World Problem Repository (Koncel-Kedziorski et al. (2016)): https://github.com/sroy9/mawps, license: CC	742 743 744
BY 4.0	745 746
• SVAMP (Patel et al. (2021)): https://github.com/arkilpatel/SVAMP, MIT license: https://github.com/arkilpatel/SVAMP/blob/main/LICENSE	747 748 749
• GSM8K (Cobbe et al. (2021)): https://github.com/openai/grade-school-math, MIT li- cense: https://github.com/openai/grade-school-math/blob/master/LICENSE	750 751
Our use of the datasets is consistent with their intended use under the licenses.	752

B Experiment details for symbolic reasoning

754 B.1 Last letter concatenation

753

755

757

Here we show correct and incorrect examples for all the prompting methods we have mentioned in the paper on the task of last letter concatenation.

Zero-shot standard

Q: Concatenate the last letters of the given words: "imagine,admire,assume,equally" A: e, e, e, y. Therefore, the answer is "eeey" (correct)

Q: Concatenate the last letters of the given words: uh,employer,intelligent,straight" A: hrtt. Therefore, the answer is "hrtt". (correct)

Q: Concatenate the last letters of the given words: "motion,instant,act,chapter" A: n,t,t,r. Therefore, the answer is "nttr". (correct)

Q: Concatenate the last letters of the given words: "fully,drug,gut,agreement"

A: lygdnt. Therefore, the answer is "lygdnt" (incorrect)

Q: Concatenate the last letters of the given words: "evil,secret,expansion,experience" A: Ints. Therefore, the answer is Ints. (incorrect)

Q: Concatenate the last letters of the given words: "consist,wear,concede,considerable" A: t-r-e-e-l-e. Therefore, the answer is "treele". (incorrect)

Table 8: Examples of correct and incorrect results by zero-shot standard prompting on last letter concatenation.

Zero-shot CoT

Q: Concatenate the last letters of the given words: "fully,drug,gut,agreement"

A: Let's think step by step.

1. The last letter of "fully" is "y".

2. The last letter of "drug" is "g".

3. The last letter of "gut" is "t".

4. The last letter of "agreement" is "t".

5. Concatenating these letters together, we get "ygtt". Therefore, the answer is "ygtt". (correct)

Q: Concatenate the last letters of the given words: "imagine,admire,assume,equally"

A: Let's think step by step.1. The last letter of "imagine" is "e".

2. The last letter of "admire" is "e".

3. The last letter of "assume" is "e".

4. The last letter of "equally" is "y".

5. Concatenating these letters gives us "eeeY". Therefore, the answer is "eeeY". (correct)

Q: Concatenate the last letters of the given words: "consist,wear,concede,considerable"

A: Let's think step by step.1. The last letter of "consist" is "t".

2. The last letter of "wear" is "r".

3. The last letter of "concede" is "e".

4. The last letter of "considerable" is "e".

5. Concatenating these letters together, we get "tree". Therefore, the answer is "tree". (correct)

Q: Concatenate the last letters of the given words: "evil,secret,expansion,experience"

A: Let's think step by step.1. The last letter of "evil" is "l".

2. The last letter of "secret" is "t".

3. The last letter of "expansion" is "n".

4. The last letter of "experience" is "e".

5. Concatenating these letters together, we get "lten". Therefore, the answer is "lten". (incorrect)

Q: Concatenate the last letters of the given words: "weapon,reasonable,thin,loyalty"

A: Let's think step by step.1. The last letter of "weapon" is "n".

2. The last letter of "reasonable" is "e".

3. The last letter of "thin" is "n".

4. The last letter of "loyalty" is "y".

5. Concatenating these letters together, we get "ney". Therefore, the answer is "ney". (incorrect)

Q: Concatenate the last letters of the given words: "motion,instant,act,chapter"

A: Let's think step by step.1. The last letter of "motion" is "n".

2. The last letter of "instant" is "t".

3. The last letter of "act" is "t".

4. The last letter of "chapter" is "r".

5. Concatenating these letters together, we get "ntr". Therefore, the answer is "ntr". (incorrect)

Table 9: Examples of correct and incorrect results by zero-shot CoT prompting on last letter concatenation.

Few-shot CoT

Q: "think, machine"

A: The last letter of "think" is "k". The last letter of "machine" is "e". Concatenating "k", "e" leads to "ke". So, "think, machine" outputs "ke".

Q: "learning, reasoning, generalization"

A: The last letter of "learning" is "g". The last letter of "reasoning" is "g". The last letter of "generalization" is "n". Concatenating "g", "g", "n" leads to "ggn". So, "learning, reasoning, generalization" outputs "ggn".

Table 10: Exemplars for the full prompt of few-shot CoT prompting on last letter concatenation. We follow Zhou et al. (2023) to construct the exemplars.

Q: Concatenate the last letters of the given words: "fully,drug,gut,agreement"

A: The last letter of "fully" is "y". The last letter of "drug" is "g". The last letter of "gut" is "t". The last letter of "agreement" is "t". Concatenating "y", "g", "t", "t" leads to "ygtt". So, "fully,drug,gut,agreement" outputs "ygtt".(correct)

Q: Concatenate the last letters of the given words: "imagine,admire,assume,equally" A: The last letter of "imagine" is "e". The last letter of "admire" is "e". The last letter of "assume" is "e". The last letter of "equally" is "y". Concatenating "e", "e", "e", "y" leads to "eeey". So, "imagine, admire, assume, equally" outputs "eeey".(correct)

Q: Concatenate the last letters of the given words: "evil,secret,expansion,experience"

A: The last letter of "evil" is "l". The last letter of "secret" is "t". The last letter of "expansion" is "n". The last letter of "experience" is "e". Concatenating "l", "t", "n", "e" leads to "ltne". So, "evil,secret,expansion,experience" outputs "ltne".(correct)

Q: "vacation, diverse, argument, speculation"

A: The last letter of "vacation" is "n". The last letter of "diverse" is "e". The last letter of "argument" is "t". The last letter of "speculation" is "n". Concatenating "n", "e", "t", "n" leads to "nent". So, "vacation, diverse, argument, speculation" outputs "nent". (incorrect)

Q: "Christianity,arm,invention,dog"

A: There is no common letter at the end of all four words, so there is no output for "Christianity, arm, invention, dog".(incorrect)

Q: "German, establishment, accurate, initially"

A: The last letter of "German" is "n". The last letter of "establishment" is "t". The last letter of "accurate" is "e". The last letter of "initially" is "y". Concatenating "n", "t", "e", "y" leads to "nty". So, "German, establishment, accurate, initially" outputs "nty". (incorrect)

Table 11: Examples of correct and incorrect results by few-shot CoT prompting on last letter concatenation.

Zero-shot code

Generate python code to concatenate the last letters of the given words.

Here's the Python code to concatenate the last letters of the given words: words = ["apple", "banana", "cherry", "date", "elderberry"] result = "" for word in words: result += word[-1] print(result)

Table 12: Code generated in the first stage of code prompting for last letter concatenation.

```
Q: "fully,drug,gut,agreement"
A:
Let's think step by step. Print all the middle variables.
words = ["fully", "drug", "gut", "agreement"]
result = ""
for word in words:
last_letter = word[-1]
result += last_letter
print(f"Word: {word}, Last Letter: {last_letter}, Result: {result}")
print(result)
. . .
Output:
- - -
Word: fully, Last Letter: y, Result: y
Word: drug, Last Letter: g, Result: yg
Word: gut, Last Letter: t, Result: ygt
Word: agreement, Last Letter: t, Result: ygtt
ygtt
Therefore, the answer is "ygtt". (correct)
Q: "feed, characteristic, spray, anymore"
A:
Let's think step by step. Print all the middle variables.
. . .
words = ["feed", "characteristic", "spray", "anymore"]
result = ""
for word in words:
last_letter = word[-1]
result += last_letter
print(f"Word: {word}, Last Letter: {last_letter}, Result: {result}")
print(result)
. . .
Output:
Word: feed, Last Letter: d, Result: d
Word: characteristic, Last Letter: c, Result: dc
Word: spray, Last Letter: y, Result: dcy
Word: anymore, Last Letter: e, Result: dbye
dbye
Therefore, the answer is dbye. (incorrect)
```

Table 13: Examples of correct and incorrect results (in the second stage) by zero-shot code prompting on last letter concatenation.

B.2 Coin flip

Here we show correct and incorrect examples for all the prompting methods we have mentioned in the paper on the task of coin flip.

Zero-shot standard

Q: A coin is heads up. Taylor doesn't flip the coin. Harmon doesn't flip the coin. Dejesus doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Yes, the coin is still heads up. Therefore, the answer (Yes or No) is Yes. (correct)

Q: A coin is heads up. Tyler doesn't flip the coin. Gates doesn't flip the coin. Weber doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Yes, the coin is still heads up. Therefore, the answer (Yes or No) is Yes. (correct)

Q: A coin is heads up. Kline doesn't flip the coin. Underwood doesn't flip the coin. Yu doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Yes, the coin is still heads up. The fact that no one flipped the coin does not change its initial state. Therefore, the answer (Yes or No) is Yes. (correct)

Q: A coin is heads up. Valencia doesn't flip the coin. Ross flips the coin. Walter doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Yes, the coin is still heads up. The actions of Valencia, Ross, and Walter do not affect the initial state of the coin. Therefore, the answer (Yes or No) is Yes. (incorrect)

Q: A coin is heads up. Valentine flips the coin. Beck flips the coin. Molina doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: No, we don't know the result of Valentine and Beck's flips, so we can't assume the coin is still heads up. Therefore, the answer (Yes or No) is unknown. (incorrect)

Q: A coin is heads up. Stephens flips the coin. Cooper doesn't flip the coin. Olson flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: It depends on the outcome of Olson's flip. If Olson flips the coin and it lands heads up, then the coin is still heads up. If Olson flips the coin and it lands tails up, then the coin is no longer heads up.Therefore, the answer (Yes or No) is uncertain. (incorrect)

Table 14: Examples of correct and incorrect results by zero-shot standard prompting on coin flip.

Zero-shot CoT

Q: A coin is heads up. Taylor doesn't flip the coin. Harmon doesn't flip the coin. Dejesus doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Let's think step by step. Initially, the coin is heads up. None of the three people mentioned in the question flip the coin, which means they don't reverse it. Therefore, the coin remains in its initial state, which is heads up. So, yes, the coin is still heads up. Therefore, the answer (Yes or No) is Yes. (correct)

Q: A coin is heads up. Valencia doesn't flip the coin. Ross flips the coin. Walter doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Let's think step by step.

1. The coin is heads up.

2. Valencia doesn't flip the coin, so the coin remains heads up.

3. Ross flips the coin, which means the coin is now tails up.

4. Walter doesn't flip the coin, so the coin remains tails up.

Therefore, the answer is no, the coin is not still heads up. It is now tails up. Therefore, the answer (Yes or No) is No. (correct)

Q: A coin is heads up. Medrano doesn't flip the coin. Cannon doesn't flip the coin. Hutchinson flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Let's think step by step.

1. The coin is heads up.

2. Medrano doesn't flip the coin, so it remains heads up.

3. Cannon doesn't flip the coin, so it still remains heads up.

4. Hutchinson flips the coin. We don't know the result of the flip, so we can't say for sure whether the coin is still heads up or not.

Therefore, we can't determine whether the coin is still heads up or not after Hutchinson flips it. Therefore, the answer (Yes or No) is indeterminate. (incorrect)

Q: A coin is heads up. Farmer flips the coin. Chandler doesn't flip the coin. Lozano flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Let's think step by step.

1. The coin is heads up.

2. Farmer flips the coin, which means he reverses it. So now the coin is tails up.

3. Chandler doesn't flip the coin, so it remains tails up.

4. Lozano flips the coin, which means he reverses it again. So now the coin is heads up again.

Therefore, the coin is not still heads up, but it has gone through a series of flips and is now heads up again. Therefore, the answer (Yes or No) is No. (incorrect)

Table 15: Examples of correct and incorrect results by zero-shot CoT prompting on coin flip.

Q: A coin is heads up. Ka flips the coin. Sherrie flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Ka and Sherrie. So the coin was flipped 2 times, which is an even number. The coin started heads up, so after an even number of flips, it will still be heads up. So the answer is yes.

Q: A coin is heads up. Jamey flips the coin. Teressa flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Jamey and Teressa. So the coin was flipped 2 times, which is an even number. The coin started heads up, so after an even number of flips, it will still be heads up. So the answer is yes.

Q: A coin is heads up. Maybelle flips the coin. Shalonda does not flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Maybelle. So the coin was flipped 1 time, which is an odd number. The coin started heads up, so after an odd number of flips, it will be tails up. So the answer is no.

Q: A coin is heads up. Millicent does not flip the coin. Conception flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Conception. So the coin was flipped 1 time, which is an odd number. The coin started heads up, so after an odd number of flips, it will be tails up. So the answer is no.

Q: A coin is heads up. Sal flips the coin. Raymond does not flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Sal. So the coin was flipped 1 time, which is an odd number. The coin started heads up, so after an odd number of flips, it will be tails up. So the answer is no.

Q: A coin is heads up. Conception flips the coin. Kristian does not flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Conception. So the coin was flipped 1 time, which is an odd number. The coin started heads up, so after an odd number of flips, it will be tails up. So the answer is no.

Q: A coin is heads up. Inga does not flip the coin. Elanor does not flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by no one. So the coin was flipped 0 times. The coin started heads up, and it was not flipped, so it is still heads up. So the answer is yes.

Q: A coin is heads up. Ryan flips the coin. Shaunda flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Ryan and Shaunda. So the coin was flipped 2 times, which is an even number. The coin started heads up, so after an even number of flips, it will still be heads up. So the answer is yes.

Table 16: Exemplars for the full prompt of few-shot CoT prompting on coin flip. We follow Wei et al. (2023) to construct the exemplars.

Q: A coin is heads up. Taylor doesn't flip the coin. Harmon doesn't flip the coin. Dejesus doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was not flipped by anyone. So the coin was flipped 0 times. The coin started heads up, and it was not flipped, so it is still heads up. So the answer is yes. (correct)

Q: A coin is heads up. Valencia doesn't flip the coin. Ross flips the coin. Walter doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Ross. So the coin was flipped 1 time, which is an odd number. The coin started heads up, so after an odd number of flips, it will be tails up. So the answer is no. (correct)

Q: A coin is heads up. Valentine flips the coin. Beck flips the coin. Molina doesn't flip the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: The coin was flipped by Valentine and Beck. So the coin was flipped 2 times, which is an even number. The coin started heads up, so after an even number of flips, it will still be heads up. Molina did not flip the coin, so it does not affect the outcome. So the answer is yes. (correct)

Q: A coin is heads up. Park flips the coin. Cooper doesn't flip the coin. Wright doesn't flip the coin. Is the coin still heads up?

A: The coin was flipped by Park. So the coin was flipped 1 time, which is an odd number. The coin started heads up, so after an odd number of flips, it will be tails up. However, since Cooper and Wright did not flip the coin, we do not know if it was flipped again. So the answer is uncertain. (incorrect)

Table 17: Examples of correct and incorrect results by few-shot CoT prompting on coin flip.

Zero-shot code

A coin is heads up, there are some people, each one flipped or didn't flip the coin. Generate python code to determine whether the coin is still heads up.

Note that "flip" here means "reverse".

Assuming that the initial state of the coin is represented by a boolean variable called "heads_up" and the list of people's actions is represented by a list of boolean values called "flips", the following Python code can be used to determine whether the coin is still heads up:

heads_up = True # initial state of the coin
flips = [True, False, True, True, False] # list of people's actions
for flip in flips:
if flip:
heads_up = not heads_up
if the person didn't flip the coin, do nothing
if heads_up:
print("The coin is still heads up.")
else:
print("The coin is now tails up.")

Table 18: Code generated in the first stage of code prompting for coin flip.

the coin. Is the coin still heads up? Let's think step by step. Print all the middle variables. A:Yes, the coin is still heads up. To verify this, we can use the same Python code as before, but with a different list of flips: heads_up = True # initial state of the coin flips = [False, False, False] # list of people's actions for flip in flips: if flip: heads_up = not heads_up # reverse the state of the coin # if the person didn't flip the coin, do nothing print(heads_up) # print the current state of the coin if heads_up: print("The coin is still heads up.")

Q: A coin is heads up. Taylor doesn't flip the coin. Harmon doesn't flip the coin. Dejesus doesn't flip

else:

print("The coin is now tails up.")

• • •

When we run this code, we can see that the current state of the coin remains True (heads up) throughout the loop, and the final message confirms that the coin is still heads up:

True True True The coin is still heads up.

Therefore, the answer (Yes or No) is Yes. (correct)

Q: A coin is heads up. Valencia doesn't flip the coin. Ross flips the coin. Walter doesn't flip the coin. Is the coin still heads up?

Let's think step by step. Print all the middle variables.

A:Using the same code as before, we can modify the list of flips to match the scenario described in the question:

heads_up = True # initial state of the coin flips = [False, True, False] # list of people's actions for flip in flips: if flip: heads_up = not heads_up # reverse the state of the coin # if the person didn't flip the coin, do nothing print(heads_up) if heads_up: print("The coin is still heads up.") else: print("The coin is now tails up.") When we run this code, we get the following output: True False True The coin is still heads up.

This means that the coin is still heads up after Ross flips it, since the final state of the coin is still heads up. Therefore, the answer (Yes or No) is Yes, the coin is still heads up. (incorrect)

Table 19: Examples of correct and incorrect results by zero-shot code prompting on coin flip.

C Error analysis for symbolic reasoning

771

772

773

774

776

Here we show an example to illustrate how ambiguity disturbs CoT prompting.

Q: A coin is heads up. Christian doesn't flip the coin. Lewis doesn't flip the coin. Becker flips the coin. Is the coin still heads up? Note that "flip" here means "reverse".

A: Let's think step by step.

- 1. The coin is heads up.
- 2. Christian doesn't flip the coin, so it remains heads up.
- 3. Lewis doesn't flip the coin, so it still remains heads up.

4. Becker flips the coin. We don't know the result of the flip, so we can't say for sure whether the coin is still heads up or not.

Therefore, we can't determine whether the coin is still heads up or not without knowing the result of Becker's flip. Therefore, the answer (Yes or No) is unknown.

Table 20: Example of ambiguity in CoT prompting. In the specific example, in the fourth step, the LLM is confused by the meaning of "flip" and thus fail to give the correct answer.

D Experiment details for arithmetic reasoning

Here we show full prompts for zero-shot CoT prompting, zero-shot code prompting (+irr/+equ), few-shot CoT, PAL, few-shot code prompting (+irr/+equ) and few-shot code prompting + LLM self-contained. Furthermore, we show the results for few-shot code prompting + LLM self-contained.

Besides, we add system messages for all the prompting methods to align with Gao et al. (2023). For CoT prompting, we set the system message to "You will solve math problems."; for PAL and code prompting, we set the system message to "You will write python program to solve math problems. You will only write code blocks.".

Zero-shot CoT

Q: {question} A: Let's think step by step.

Table 21: Zero-shot CoT prompt for math world problems.

plain:

Generate python code to answer the question. Note that code should follow the format ```code```. Q: {question}

+irr:

Generate python code to answer the question. Note that code should follow the format ```code```. There may be irrelevant information in the question. If you find it, ignore it. Q: {question}

+equ:

Generate python code to answer the question. Note that code should follow the format ```code```. If you need to solve an equation, here's an instruction: ```python # to solve an equation, you can use python package sympy import sympy # for example, to solve $2^*x = 5$ # First, declarify your variable, in this case, 'x' x = sympy.symbols("x") # Second, transform the equation so that the right hand side of the equation is zero. #2*x - 5 = 0# Third, use 'sympy.solve' to solve the equation a = sympy.solve([2 * x - 5], [x])# Print the output as a float. Note that 'a' is a dict print(float(a[x])) Q: {question}

Table 22: Zero-shot code prompt (plain/+irr/+equ) for math world problems.

Few-shot CoT



Figure 6: The pipelines of few-shot CoT prompting and few-shot code prompting are shown in the figure.

Let's think step by step to solve math problems. Here are three examples how to do it,

Q: Olivia has \$23. She bought five bagels for \$3 each. How much money does she have left? A: Olivia had 23 dollars. 5 bagels for 3 dollars each will be $5 \ge 3 = 15$ dollars. So she has 23 - 15 dollars left. 23 - 15 is 8. The answer is 8.

Q: Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How many golf balls did he have at the end of wednesday?

A: Michael started with 58 golf balls. After losing 23 on tuesday, he had 58 - 23 = 35. After losing 2 more, he had 35 - 2 = 33 golf balls. The answer is 33.

Q: There were nine computers in the server room. Five more computers were installed each day, from monday to thursday. How many computers are now in the server room?

A: There were originally 9 computers. For each of 4 days, 5 more computers were added. So 5 * 4 = 20 computers were added. 9 + 20 is 29. The answer is 29.

How about this question? Q: {question}

Table 23: Few-shot CoT prompt for math world problems. We follow the code released by Gao et al. (2023) to choose the exemplars.

PAL

Let's use python to solve math problems. Here are three examples how to do it,

Q: Olivia has \$23. She bought five bagels for \$3 each. How much money does she have left?

```
def solution():
"""Olivia has $23. She bought five bagels for $3 each. How much money does she have left?"""
money_initial = 23
bagels = 5
bagel_cost = 3
money_spent = bagels * bagel_cost
money_left = money_initial - money_spent
result = money_left
return result
```

Q: Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How many golf balls did he have at the end of wednesday?

```
def solution():
"""Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How
many golf balls did he have at the end of wednesday?"""
golf_balls_initial = 58
golf_balls_lost_tuesday = 23
golf_balls_lost_wednesday = 2
golf_balls_left = golf_balls_initial - golf_balls_lost_tuesday - golf_balls_lost_wednesday
result = golf_balls_left
return result
```

Q: There were nine computers in the server room. Five more computers were installed each day, from monday to thursday. How many computers are now in the server room?

```
def solution():
```

"""There were nine computers in the server room. Five more computers were installed each day, from monday to thursday. How many computers are now in the server room?"""

```
computers_initial = 9
computers_per_day = 5
num_days = 4 # 4 days between monday and thursday
computers_added = computers_per_day * num_days
computers_total = computers_initial + computers_added
result = computers_total
return result
We about this question?
Q: {question}
```

Table 24: PAL prompt for math world problems. We use the demonstrations from the code released by Gao et al. (2023).

Few-shot code + comments

Let's use python to solve math problems. Here are three examples how to do it, Q: Olivia has \$23. She bought five bagels for \$3 each. How much money does she have left?

def solution():

"""Olivia has \$23. She bought five bagels for \$3 each. How much money does she have left?""" money_initial = 23 # Olivia has \$23 initially

bagels = 5 # Olivia bought 5 bagels

bagel_cost = 3 # Each bagel cost \$3

money_spent = bagels * bagel_cost # The total cost of 5 bagels is the product of the price of each bagel and the number of bagels

money_left = money_initial - money_spent # Money left is the difference between initial money and the total cost of 5 bagels

result = money_left

return result

• • •

Q: Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How many golf balls did he have at the end of wednesday?

def solution():

"""Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How many golf balls did he have at the end of wednesday?"""

golf_balls_initial = 58 # Michael had 58 golf balls initially

golf_balls_lost_tuesday = 23 # He lost 23 golf balls on Tuesday

golf_balls_lost_wednesday = 2 # He lost 2 more golf balls on Wednesday

golf_balls_left = golf_balls_initial - golf_balls_lost_tuesday - golf_balls_lost_wednesday # Number of golf balls remaining is the difference between initial golf balls and the golf balls lost on Tuesday and Wednesday

result = golf_balls_left return result

• • •

Q: There were nine computers in the server room. Five more computers were installed each day, from monday to thursday. How many computers are now in the server room?

def solution():

"""There were nine computers in the server room. Five more computers were installed each day, from monday to thursday. How many computers are now in the server room?"""

computers_initial = 9 # There were 9 computers in the server room initially

computers_per_day = 5 # 5 more computers were installed each day from Monday to Thursday

```
num_days = 4 # 4 days between monday and thursday
```

computers_added = computers_per_day * num_days # Additional computers are the product of the number of computers installed each day and the number of days

```
computers_total = computers_initial + computers_added # Total number of computers is the sum of initial computers and the additional computers installed on 4 days
```

result = computers_total

return result

• • •

How about this question? Q: {question}

Table 25: Few-shot code prompt for math world problems.

Few-shot code + LLM self-contained

 $bagel_cost = 3$

result = money_left = 8

So the answer is 8.

money_left = money_initial - money_spent = 23 - 15 = 8

Let's use python to solve math problems. Here are three examples how to do it,

Q: Olivia has \$23. She bought five bagels for \$3 each. How much money does she have left?

```
def solution():
"""Olivia has $23. She bought five bagels for $3 each. How much money does she have left?"""
money_initial = 23 # Olivia has $23 initially
bagels = 5 # Olivia bought 5 bagels
bagel_cost = 3 # Each bagel cost $3
money_spent = bagels * bagel_cost # The total cost of 5 bagels is the product of the price of each bagel
and the number of bagels
money_left = money_initial - money_spent # Money left is the difference between initial money and
the total cost of 5 bagels
result = money_left
return result
. . .
Let's think step by step according to the code:
. . .
money_initial = 23
bagels = 5
```

```
many golf balls did he have at the end of wednesday?
```

"""Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How many golf balls did he have at the end of wednesday?"""

Q: Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How

golf_balls_initial = 58 # Michael had 58 golf balls initially golf_balls_lost_tuesday = 23 # He lost 23 golf balls on Tuesday golf_balls_lost_wednesday = 2 # He lost 2 more golf balls on Wednesday golf_balls_left = golf_balls_initial - golf_balls_lost_tuesday - golf_balls_lost_wednesday # Number of golf balls remaining is the difference between initial golf balls and the golf balls lost on Tuesday and Wednesday result = golf_balls_left return result

```
Let's think step by step according to the code:

golf_balls_initial = 58

golf_balls_lost_tuesday = 23

golf_balls_lost_wednesday = 2

golf_balls_left = golf_balls_initial - golf_balls_lost_tuesday - golf_balls_lost_wednesday = 58 - 23 - 2

= 33

result = golf_balls_left = 33

...

So the answer is 33.

Q: There were nine computers in the server room. Five more computers were installed each day, from

monday to thursday. How many computers are now in the server room?

...

def solution():

"""There were nine computers in the server room. Five more computers were installed each day, from

monday to thursday. How many computers are now in the server room?

...
```

```
computers initial = 9 \# There were 9
```

computers in the server room initially computers_per_day = 5 # 5 more computers were installed each day from Monday to Thursday

num_days = 4 # 4 days between monday and thursday

computers_added = computers_per_day * num_days # Additional computers are the product of the number of computers installed each day and the number of days

computers_total = computers_initial + computers_added # Total number of computers is the sum of initial computers and the additional computers installed on 4 days

result = computers_total return result

• • •

Let's think step by step according to the code:

```
computers_initial = 9
computers_per_day = 5
num_days = 4
computers_added = computers_per_day * num_days = 5 * 4 = 20
computers_total = computers_initial + computers_added = 9 + 20 = 29
result = computers_total = 29
So the answer is 29.
```

How about this question? Q: {question}

Table 26: Prompt of few-shot code + LLM self-contained for math world problems.

785 E Error analysis for arithmetic reasoning

6 E.1 AddSub

As mentioned in §5.2, in AddSub, LLMs make errors mainly due to the distraction of irrelevant variables.
Here we show two examples of the LLMs distracted by irrelevant variables.

Q: Joan had 695 Pokemon cards, and 6 were torn. Sara bought 133 of Joan's Pokemon cards. How many Pokemon cards does Joan have now?	Q: Dan spent \$11.76 on a snake toy, and a cage cost him \$14.54. Dan also found a dollar bill on the ground. What was the total cost of Dan's purchases?
<pre># Joan had 695 Pokemon cards initially initial_cards = 695 # 6 of the cards were torn torn_cards = 6 # Sara bought 133 of Joan's Pokemon cards cards_sold = 133 # Total number of cards Joan has now is the difference between initial cards, torn cards, and cards sold remaining_cards = initial_cards - torn_cards - cards_sold # Print the final result print(remaining_cards)</pre>	<pre>snake_toy_cost = 11.76 # The cost of the snake toy is \$11.76 cage_cost = 14.54 # The cost of the cage is \$14.54 found_money = 1 # Dan found \$1 on the ground total_cost = snake_toy_cost + cage_cost - found_money # The total cost is the sum of the cost of the snake toy and the cage, minus the dollar bill Dan found print(total_cost) # Print the final result</pre>

Table 27: Error examples for AddSub. In the left example, torn_card is an irrelevant variable that should not appear in the equation. The correct answer should be remaining_cards = initial_cards - torn_cards, which gives 562. In the right example, found_money is an irrelevant variable. The correct answer should be total_cost = snake_toy_cost + cage_cost, which gives 26.3.

E.2 MultiArith

In MultiArith, the wrong answers from code prompting are mainly due to two reasons: 1) Code prompting is more sensitive to ambiguity in the question, which may leads to a wrong answer. 2) The groundtruth for some questions is wrong, while the predictions are correct.

Here we show examples for both cases.

Q: The school cafeteria ordered 43 red apples and 32 green apples for students lunches. But, if only 2 students wanted fruit, how many extra did the cafeteria end up with?

```
red_apples = 43 # The cafeteria ordered 43 red apples
green_apples = 32 # The cafeteria ordered 32 green apples
total_apples = red_apples + green_apples # Total number of apples ordered is the sum of red and green apples
students want fruit = 2 # Only 2 students wanted fruit
apples_needed = students_want_fruit * 2 # Each student needs 2 apples
extra_apples = total_apples - apples_needed # Extra apples are the difference between total apples and the
apples needed for the 2 students
print(extra_apples) # Print the final result
```

Table 28: An error example of few-shot code prompting for MultiArith. The question does not make it clear how many apple a student want, but the ground truth suggests that we should take it as default that each student wants 1 apple. Code prompting makes the LLM more sensitive to such ambiguity.

Q: Emily's team won their dodgeball game and scored 39 points total. If Emily scored 23 of the points and everyone else scored 2 points each, how many players were on her team?

```
emily_points = 23 # Emily scored 23 points
other_players_points = 2 # Each other player scored 2 points
total_points = 39 # The team scored 39 points in total
num_other_players = (total_points - emily_points) / other_players_points # Number of other players is the
difference between total points and Emily's points, divided by the points each other player scored
num_players = num_other_players + 1 # Total number of players is the number of other players plus Emily
print(num_players) # Print the final result
```

Table 29: An error example of few-shot code prompting for MultiArith. The ground truth fails to count Emily into the players, resulting in wrong answer (39 - 23)/2 = 8.

789 790 701

792

794 E.3 GSM8K

795

Error with equations. We have mentioned in §5.3 that the LLM struggles to solve equations in code.

When facing an equation, the LLM tends to solve it in the comments or directly lists the equation in the code, which may easily result in wrong answers or bugs. Here are two examples.

Q: Ruby is 6 times older than Sam. In 9 years, Ruby will be 3 times as old as Sam. How old is Sam now?

```
# Let's assume Sam's age is x
# According to the question, Ruby is 6 times older than Sam
# So, Ruby's age will be 6x
# In 9 years, Ruby will be 3 times as old as Sam
# So, (6x + 9) = 3(x + 9)
# Solving the above equation, we get x = 9
# Therefore, Sam's age now is 9 years old
sam_age = 9
print("Sam's age now is", sam_age, "years old.")
```

Table 30: An error example of code prompting for GSM8K.

Q: The three angles in a triangle add up to 180 degrees. One angle is twice the smallest angle, and one angle is three times the smallest angle. What is the measure of the largest angle in the triangle, in degrees?

```
smallest_angle = x
second_angle = 2 * x
third_angle = 3 * x
sum_of_angles = smallest_angle + second_angle + third_angle
largest_angle = third_angle
print(largest_angle)

Bug report:
NameError: name 'x' is not defined
```

Table 31: An error example of code prompting for GSM8K.

Error distribution of CoT prompting and code prompting. Here we show the error distribution of few-shot CoT prompting and few-shot code prompting(w/ comments at the end). Since here we only investigate into the error overlap of CoT prompting and code prompting, to eliminate the effects of system messages and instructions, we remove all the system messages and unnecessary instructions (They are added in the former experiments to align with the method PAL Gao et al. (2023)). Besides, we use the original 8 exemplars from Wei et al. (2023). In §5.4, we show the results of 8-shot CoT prompting, 8-shot code prompting and the ensemble of them.



Figure 7: Error distribution of few-shot code prompting and few-shot CoT prompting regarding dataset GSM8K.