Course Project: Part 1

Generative AI

Saarland University – Winter Semester 2024/25

Martínez
7057573
`cama00005@stud.uni-saarland.de`

Part 1.a: Basic Prompts

Table 1: Overall results for GPT-4o-mini and Phi-3-mini with the basic prompt to generate program repairs. For **(I.1)** and **(I.3)**.

Model	***RPass***	***REdit***
GPT-4o-mini	88.0	23.77
Phi-3-mini	36.0	18.11

Table 2: Average per problem results for GPT-4o-mini with the basic prompt to generate program repairs. The average is calculated without considering the cases with edit distance $= -1$. For **(I.1)**.

Problem	**Prog. 1**	**Prog. 2**	**Prog. 3**	**Prog. 4**	**Prog. 5**	**Avg.**
Problem 1	4	-1	15	19	24	15.50
Problem 2	1	63	16	85	54	43.80
Problem 3	7	19	11	36	10	16.60
Problem 4	53	5	5	39	2	20.80
Problem 5	-1	12	4	-1	39	18.33

Table 3: Average per problem results for Phi-3-mini with the basic prompt to generate program repairs. The average is calculated without considering the cases with edit distance $= -1$. For **(I.3)**.

Problem	**Prog. 1**	**Prog. 2**	**Prog. 3**	**Prog. 4**	**Prog. 5**	**Avg.**
Problem 1	23	-1	5	-1	-1	14.00
Problem 2	-1	-1	-1	-1	-1	0.00
Problem 3	-1	-1	-1	-1	10	10.00
Problem 4	51	4	5	-1	30	22.50
Problem 5	-1	-1	10	-1	25	17.50

Table 4: Hint Quality Metrics for GPT-4o-mini. This model tends to be very verbose, as well as directly pointing out the code solution, improving its *HInformative* metric, but at the expense of *HConceal*. For **(I.2)**.

Problem	**Program**	***HCorrect***	***HInformative***	***HConceal***	***HComprehensible***	***HGood***
Problem 1	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	0	1	0
	Prog. 3	1	1	0	1	0
	Prog. 4	1	1	0	1	0
	Prog. 5	1	1	0	1	0
Avg. Problem 1		1.0	1.0	0.0	1.0	0.0
Problem 2	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	1	1	1
	Prog. 3	1	1	0	0	0
	Prog. 4	1	0	1	1	0
	Prog. 5	1	1	1	1	1
Avg. Problem 2		1.0	0.8	0.6	0.8	0.4
Problem 3	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	0	1	0
	Prog. 3	1	1	1	1	1
	Prog. 4	1	1	1	0	0
	Prog. 5	1	1	0	1	0
Avg. Problem 3		1.0	1.0	0.4	0.8	0.2
Problem 4	Prog. 1	1	1	1	1	1
	Prog. 2	1	1	1	1	1
	Prog. 3	1	1	1	1	1
	Prog. 4	1	1	0	1	0
	Prog. 5	1	1	0	0	0
Avg. Problem 4		1.0	1.0	0.6	0.8	0.6
Problem 5	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	1	0	0
	Prog. 3	1	1	0	1	0
	Prog. 4	1	1	1	1	1
	Prog. 5	1	1	1	0	0
Avg. Problem 5		1.0	1.0	0.6	0.6	0.2
Overall Average		1.0	0.96	0.44	0.8	0.28

Table 5: Hint Quality Metrics for Phi-3-mini. Similar to GPT-4o-mini, this model tends to directly give answers, instead of actually hinting, affecting its *HInformative* metric. At the same time, it has readability and formatting issues, which affect its *HComprehensible*. For **(I.4)**.

Problem	**Program**	***HCorrect***	***HInformative***	***HConceal***	***HComprehensible***	***HGood***
Problem 1	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	0	1	0
	Prog. 3	1	1	0	1	0
	Prog. 4	1	0	1	1	0
	Prog. 5	1	0	1	1	0
Avg. Problem 1		1.0	0.6	0.4	1.0	0.0
Problem 2	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	0	1	0
	Prog. 3	1	1	1	0	0
	Prog. 4	0	1	1	1	0
	Prog. 5	1	1	1	1	1
Avg. Problem 2		0.8	1.0	0.6	0.8	0.2
Problem 3	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	0	1	0
	Prog. 3	1	1	0	0	0
	Prog. 4	1	1	1	0	0
	Prog. 5	1	1	0	1	0
Avg. Problem 3		1.0	1.0	0.2	0.6	0.0
Problem 4	Prog. 1	1	1	1	1	1
	Prog. 2	1	1	0	1	0
	Prog. 3	1	1	1	1	1
	Prog. 4	1	1	1	1	1
	Prog. 5	1	1	0	0	0
Avg. Problem 4		1.0	1.0	0.6	0.8	0.6
Problem 5	Prog. 1	1	1	0	1	0
	Prog. 2	1	1	1	0	0
	Prog. 3	0	1	1	0	0
	Prog. 4	1	1	1	0	0
	Prog. 5	0	0	0	0	0
Avg. Problem 5		0.6	0.8	0.6	0.2	0.0
Overall Average		0.88	0.88	0.48	0.68	0.16

Part 1.b: Improved Program Repairs

The modifications I did to the provided codebase to implement the *improved program repairs* are as follows:

1. The method `Repair.generate_repair()` now takes 3 additional positional parameters: `n=1`, `do_sample=None`, `temperature=None`. These are defined in the main method of the `project_part1_evaluate.py` and then propagated through `DatasetEvaluation.evaluate_programs_in_folder()`, which calls the method `ProgramEvaluation.get_and_evaluate_repair()`, which in turn calls the `Repair.generate_repair()`, defined in `project_part1_repair.py`. This allows to turn on/off sampling and set the temperature for the generation of repair candidates, without affecting the basic workflow of **Part 1.a**.
2. The temperature parameter, which was propagated through as explained above, is now accepted as a new parameter in the `Repair.call_llm_openai()` function. Namely, when querying GPT-4o-mini and Phi-3-mini, it is set to 0.7.
3. Similarly, the `Repair.call_llm_huggingface()` now receives these 2 additional parameters: `do_sample` and `temperature`, which are then passed to `self.model.generate()` as additional positional arguments. This allows to enable sampling and set the temperature, when querying Phi-3-mini.
4. Inside `Repair.generate_repair()`, the algorithm 1 was implemented.

Algorithm 1: Algorithm for *improved program repairs*.

```
Input  : Problem data $T$, buggy program $P_b$, test suite $\Omega_T$, sampling flag $s$, temperature $\tau$,
         number of repaired programs to query $n$
generateRepair $(T, P_b, \Omega_T, n, s, \tau)$
    $\mathcal{C} \leftarrow$ COMPILER()                                  // Compiler instantiation
    $\mathcal{D} \leftarrow$ DISTANCE()                                  // Distance metric initialization
    $\mathcal{S} \leftarrow \emptyset$                                   // Initialize repair statistics set
    Repeat
        $r \leftarrow$ QUERYLLM$(T, P_b, s, \tau)$                       // Generate repair candidate
        $f \leftarrow$ EXTRACTFIX$(r)$                                   // Extract fixed program

        // The result of the test suite are stored in a list of tuples $(\omega, o)$ where
           $\omega$ represents whether the test case passed or not, and $o$ is the output
           of the test case
        $\Omega_{\text{results}} \leftarrow \mathcal{C}(f, \Omega_T)$    // Execute test suite $\Omega_T$ on $f$

        // Count the number of passing test cases
        $\omega_c \leftarrow \sum_{(\omega,o)\in\Omega_{\text{results}}} \mathbf{1}(\omega)$    // $\mathbf{1}(\cdot)$: Indicator function for test success
        $\delta \leftarrow \mathcal{D}(f, P_b)$                          // Compute edit distance between $f$ and $P_b$
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{(f, \omega_c, d, \Omega_{\text{results}})\}$    // Store the statistics of the current repair
    until $n$ times

    // Total ordering: Maximize $\omega_c$, minimize $\delta$. If there are multiple correct $f$,
       choose the one with smallest $\delta$
    $\mathcal{S}_{\text{max}} \leftarrow \{(f, \omega, \delta, \Omega) \in \mathcal{S} \mid \omega = \max_{(f',\omega',\delta',\Omega')\in\mathcal{S}} \omega'\}$
    $(P_r, \_, \delta_{\text{opt}}, \Omega_{\text{results}}) \leftarrow \arg\min_{(f,\omega,\delta,\Omega)\in\mathcal{S}_{\text{max}}} \delta$
    return $(P_r, \Omega_{results})$
Output : The repaired program $P_r$, with the most correct testcases and closest to $P_b$, and the
         associated results of evaluating it with the testcases, $\Omega_{\text{results}}$.
```

Table 6: Overall results for GPT-4o-mini and Phi-3-mini with the basic prompt to generate program repairs. For **(I.5)** and **(I.6)**.

Model	***RPass***	***REdit***
GPT-4o-mini	96.0	22.71
Phi-3-mini	60.0	25.87

Part 1.c: Advanced Workflow

Table 7: Hint Quality Metrics for GPT-4o-mini with the *advanced workflow*. Note how *HGood* improved from 0.28 in Table 4 to 0.72. The major problem was *HConceal*, which was tackled by the CoT prompting effectively, at the expense of *HInformative*, which decreased compared to the basic prompt strategy. For **(I.7)**.

Problem	**Program**	***HCorrect***	***HInformative***	***HConceal***	***HComprehensible***	***HGood***
Problem 1	Prog. 1	1	1	1	1	1
	Prog. 2	1	1	1	1	1
	Prog. 3	1	1	1	1	1
	Prog. 4	1	0	1	1	0
	Prog. 5	1	0	1	1	0
Avg. Problem 1		1.0	0.6	1.0	1.0	0.6
Problem 2	Prog. 1	1	1	1	1	1
	Prog. 2	1	1	1	1	1
	Prog. 3	1	1	1	1	1
	Prog. 4	1	0	1	1	0
	Prog. 5	1	1	1	1	1
Avg. Problem 2		1.0	0.8	1.0	1.0	0.8
Problem 3	Prog. 1	1	0	1	1	0
	Prog. 2	1	1	1	1	1
	Prog. 3	1	1	1	1	1
	Prog. 4	1	1	1	1	1
	Prog. 5	1	1	1	1	1
Avg. Problem 3		1.0	0.8	1.0	1.0	0.8
Problem 4	Prog. 1	1	1	1	1	1
	Prog. 2	1	1	1	1	1
	Prog. 3	1	1	1	1	1
	Prog. 4	1	0	1	0	0
	Prog. 5	1	1	1	1	1
Avg. Problem 4		1.0	0.8	1.0	0.8	0.8
Problem 5	Prog. 1	1	1	1	1	1
	Prog. 2	1	0	1	1	0
	Prog. 3	1	1	1	1	1
	Prog. 4	1	1	1	1	1
	Prog. 5	1	0	1	1	0
Avg. Problem 5		1.0	0.6	1.0	1.0	0.6
Overall Average		1.0	0.72	1.0	0.96	0.72

Table 8: Hint Quality Metrics for Phi-3-mini with the *advanced workflow*. The CoT prompting seems to have addressed the formatting and readability issues but at the cost of some specificity in the hints, as the *HInformative* metric is lower than in the basic prompt approach. For **(I.8)**.

Problem	**Program**	***HCorrect***	***HInformative***	***HConceal***	***HComprehensible***	***HGood***
Problem 1	Prog. 1	1	0	1	1	0
	Prog. 2	1	1	1	1	1
	Prog. 3	1	0	1	1	0
	Prog. 4	1	0	1	1	0
	Prog. 5	1	1	1	1	1
Avg. Problem 1		1.0	0.4	1.0	1.0	0.4
Problem 2	Prog. 1	1	0	1	1	0
	Prog. 2	1	0	1	1	0
	Prog. 3	1	1	1	1	1
	Prog. 4	1	1	1	1	1
	Prog. 5	1	1	1	1	1
Avg. Problem 2		1.0	0.6	1.0	1.0	0.6
Problem 3	Prog. 1	1	1	1	1	1
	Prog. 2	1	1	1	1	1
	Prog. 3	1	0	1	1	0
	Prog. 4	1	1	1	1	1
	Prog. 5	1	1	1	1	1
Avg. Problem 3		1.0	0.8	1.0	1.0	0.8
Problem 4	Prog. 1	0	0	1	1	0
	Prog. 2	1	1	1	1	1
	Prog. 3	1	1	1	1	1
	Prog. 4	1	1	1	1	1
	Prog. 5	1	0	1	1	0
Avg. Problem 4		0.8	0.6	1.0	1.0	0.6
Problem 5	Prog. 1	1	1	1	1	1
	Prog. 2	1	1	1	1	1
	Prog. 3	1	0	1	1	0
	Prog. 4	1	1	1	1	1
	Prog. 5	1	1	1	1	1
Avg. Problem 5		1.0	0.8	1.0	1.0	0.8
Overall Average		0.96	0.64	1.0	1.0	0.64

(I.9) The approach I took to create the *advanced workflow* and improve the results of the models involved making the following modifications to the provided codebase:

1. A new parameter called `use_advanced_workflow` was introduced in the `Hint` class. This allows to turn on the **Part 1.c**’s *advanced workflow* by setting it to `True`, and the basic workflow used for **Part 1.a** and **Part 1.b** by setting it to `False`.

2. The function `Hint.generate_hint()` was modified to include a new positional argument, `repair_agent`. So that, if `use_advanced_workflow` is set to `True`, the `repair_agent.generate_repair()` function is called, obtaining the repaired program P_r and the results of the test cases Ω_{results}, as outlined by the algorithm 1.
3. The parameters `repaired_program` and `testcases_results` were added to the function `Hint.call_llm()`, so that `Hint.generate_hint()` can pass them to generate the hint, upon calling the latter. Moreover, if `use_advanced_workflow` is set to `True`, the `user_prompt` is formatted to include not only the problem data and the student's buggy program, but also the repaired program P_r and the results of the test cases Ω_{results}. See **Appendix: Advanced Prompt**, which shows the template of the advanced prompt used.

This approach allowed me to improve the basic prompt by adding the repaired program P_r to the model, as well as the evaluation of P_r on the test suite Ω_T. This provides more context to the model, whilst taking into account that, since P_r is also generated by the model and thus is not guaranteed to be correct, the model *knows* how correct P_r actually is, based on the results of the test cases. This allows the model to have a better understanding of the problem, how a potential solution should look like (the provided repaired program), how good the potential solution is (the testcases results), and uses this context to gain ideas on potential things to pinpoint at, in order to create the final hint to the student.

This approach created a 2-step look-ahead process for the model to generate hints, which can be summarized as follows:

1. The model is prompted with a buggy code P_b to generate a repair, based on the problem data T.
2. Using **Part 1.b**'s *improved program repairs*, it generates a repaired program P_r and evaluates it on the test suite Ω_T.
3. The repaired program P_r is added to a new prompt, which also includes the problem data T, the buggy code P_b, and the results of the test cases Ω_{results}. This new prompt is used to generate the hint.
4. Upon prompting the same model with this new prompt, we give the chance for the model to analyze the problem again, the student's buggy code and a potential solution P_r, which from the model's perspective should be correct, as it was prompted in the first step to specifically generated a *repaired* program given P_b and T. But now that we included the testcases results inside the given prompt, the model can see how wrong or correct the produced P_r was, and we give it the opportunity to rationalize on why the P_r that it produced on the first step was wrong.
5. The model has now seen the problem twice. It has also seen a P_r, which was produced by itself in the first step, and it knows how correct or incorrect it was, based on the testcases results. With this information and the constrain to make it create a Chain of Thought inside the `[EXP]` placeholders, the model can focus not only on P_b, but also on how a potential solution P_r, that a student could have made, may also be wrong and why. In the case that P_r is correct, the model can directly focus on providing a hint to the student that makes P_b closer to the P_r, that it now knows for certain that is correct, thanks to the testcases results.

The previous explanation is cleverly illustrated in the Figure 5, in which the Phi-3-mini model is not able to generate a correct repair, but given that it knows that it is not correct, the model in its Chain-of-Thought explanation pinpoints bugs in the repaired program, as well as in the student's buggy code. This steers the model off of potentially providing the student a hint that would make it go towards the wrong repaired program (since in the first step, when it was prompted to generate a repair, it thought that this particular repaired program was correct). This can be seen when the model says: *"(...) The problem with the repaired code is that it does not maintain the relative order of the elements in the original list, which is a key requirement of the problem statement (...)"*. The model now has to *think*[1] twice and figure out how the student's buggy code and the repaired program are wrong, and thus it can provide a hint that looks ahead on problems that the student may encounter. This can be seen in the hint that the model finally provided: *"(...) Consider using a data structure that maintains the order of elements as you iterate through the list and check for duplicates. (...)"*. There,

---

[1] In the sense of the *Chain of Thought* approach, as LLM's actually *thinking* is a topic of debate.

it took into account both problems, the one in the repaired program as well as one in the student's buggy code, and provided a hint that combined both. In the case of GPT-4o-mini, since it was already quite good at generating hints, as seen in Table 5, the effect of this approach was not as pronounced, but it still allowed the model to provide even better hints.

Limitations[2]**.** There are of course some limitations to this approach. The better the model is in generating repairs, the better the hints will be. This can be seen on Table 7 and Table 8, in which it is clear that GPT-4o-mini is better in terms of the *HGood* metric [1] compared to Phi-3-mini. If a model always provides correct repairs, the model does not have to analyze too much, as it can simply give a concise hint that directly makes the student's buggy code closer to the repaired program.

The model is also limited by the quality of the test cases, as it can only evaluate the repaired program based on the test cases results. If the test cases are not good, the model may generate a repaired program that is actually wrong, but the test cases may not be able to capture this, and thus the model may generate a hint that makes the student's buggy code closer to this wrong repaired program.

Further Improvements[3]**.** The approach could be further improved by adding even more context to the prompt. For example, the actual input and expected output of the testcases could be provided, instead of only saying whether the testcases passed or not. This would provide the model exactly what it's been evaluated on, and based on the problem data, it should deduce some logic out of the given testcases.

At the same time, few-shot learning could be used by providing the model with examples of problems, their corresponding buggy code, repaired code and testcases results, and a final hint that was given to the student. This hint would have already been externally evaluated and deemed as good. This would allow the model to learn from examples of good hints, learn what actually makes a good hint, and generate better hints in the future that resemble those examples. In practice and if it is not possible for a human to review all examples and hints, GPT-4o-mini or an even better (potentially larger and non-free) model could be used to generate these examples with some fixed budget, and then Phi-3-mini or another open-source model could be used in production to generate hints based on these examples. This would allow it to mimic the behavior of a better model, without having to pay for each query.

---

[2] Only model, data and algorithmic limitations are considered here. There is one obvious limitation to the whole approach and its the way we are evaluating the final performance, the *HGood* metric, which requires manual annotations, which are of course subjective.

[3] *Easy-to-consider* improvements such as prompting the model indefinitely until it generates a repaired program that passes all tests or having direct access to the problem data's expected solution are not considered in this section, since they might not reflect real-life scenarios with a fixed budget, which makes it infeasible to prompt the model indefinitely.

Acknowledgements

Week 10's slides and listed references.

References

[1] Nachiket Kotalwar, Alkis Gotovos, and Adish Singla. Hints-in-browser: Benchmarking language models for programming feedback generation, 2024.

Appendix: Template of the Advanced Prompt

```
Following is the setup of a problem in Python.  It contains the
description and a sample testcase.

[Problem Starts]
{problem_data}
[Problem Ends]

Following is the student's buggy code for this problem:

[Buggy Code Starts]
{buggy_program}
[Buggy Code Ends]

Previously, a Large Language Model was asked to review the student's
buggy code and generate a repaired program based on the problem
description.  The repaired program is as follows:

[Repaired Code Starts]
{repaired_program}
[Repaired Code Ends]

Nevertheless, this repaired program is not guaranteed to be correct.
For you to review whether this repaired program is correct or not,
the results of the test cases for this problem are provided below:

[Testcases results of repaired code Starts]
{testcases_results}
[Testcases results of repaired code Ends]

Based on all of the above information, create an explanation written
inside placeholders [EXP] and [/EXP] that analyzes the student's
buggy code, the provided repaired program and the testcases results,
to come up step-by-step with the best approach to provide a hint to
the student.  After that, provide a concise single-sentence hint to
the student about one bug in the student's buggy code, which allows
him to get closer to the correct program, but without directly giving
him the code or detailed explanations.  Output your hint between
[HINT] and [/HINT]. Make sure to always use the [HINT] and [/HINT]
placeholders with a hint inside.  Do not output anything else inside
the [HINT] placeholders that is not part of your chosen hint.  Do
not refer to the student as "the student", but with "you" if strictly
necessary.  Always start your hints with "Consider" or "Think about".
```

Figure 1: Advanced prompt assigned to the variable `user_message_nus_hint_advanced` in `project_part1_prompts.py`, which is used in **??** and **??**. Notice how CoT prompting is being used by constraining the model to generate a Chain of Thought inside the `[EXP]` placeholders. Moreover, in order to constrain the kind of hints it should give, the model is constrained to always start its hints with "Consider" or "Think about".

Appendix: Real Use-case of the Advanced Prompt by GPT-4o-mini

```
[Problem Starts]
Duplicate elimination - Write a function remove_extras(lst) that...
[Problem Ends]

[Buggy Code Starts]
def remove_extras(lst):
    new = []
    for x in lst:
        if lst.count(x) > 1:
            new += []
        else:
            new = new.append(x)
    return new
[Buggy Code Ends]

Previously, a Large Language Model was asked to review the student’s
buggy code and generate a repaired program based on the problem
description.  The repaired program is as follows:

[Repaired Code Starts]
def remove_extras(lst):
    new = []
    for x in lst:
        if x not in new:
            new.append(x)
    return new
[Repaired Code Ends]

Nevertheless, this repaired program is not guaranteed to be correct.
For you to review whether this repaired program is correct or not,
the results of the test cases for this problem are provided below:

[Testcases results of repaired code Starts]
* 6 out of 6 testcases passed.
[Testcases results of repaired code Ends]

Based on all of the above information, create an explanation written
inside placeholders [EXP] and [/EXP] that analyzes the student’s
buggy code, the provided repaired program and the testcases results,
to come up step-by-step with the best approach to provide a hint to
the student.  After that, provide a concise single-sentence hint to
the student about one bug in the student’s buggy code, which allows
him to get closer to the correct program, but without directly giving
him the code or detailed explanations.  Output your hint between
[HINT] and [/HINT]. Make sure to always use the [HINT] and [/HINT]
placeholders with a hint inside.  Do not output anything else inside
the [HINT] placeholders that is not part of your chosen hint.  Do
not refer to the student as "the student", but with "you" if strictly
necessary.  Always start your hints with "Consider" or "Think about".
```

Figure 2: An example of the advanced prompt being used to query GPT-4o-mini. Notice that the repaired program created by GPT-4o-mini passes 6 out of 6 test cases, thus it’s a correct repaired program that can be used as context to generate a hint. *Prompt was shortened for brevity.*

Appendix: Hint by GPT4-o-mini given prompt in Figure 2

```
[EXP]The student’s buggy code attempts to remove duplicates but has a
couple of fundamental issues.  First, it uses ‘lst.count(x) > 1‘ to
check if an element is duplicated, which is not necessary for the
goal of this function.  This approach can be inefficient because
‘count‘ iterates through the list every time it is called, leading
to a time complexity of O(n**2) in the worst case.  Additionally, in
the line ‘new = new.append(x)‘, ‘append‘ returns ‘None‘, so ‘new‘
ends up being ‘None‘ instead of the expected list, thereby causing an
error in the function.

The repaired code correctly uses a mechanism to build a new list
by checking if each element is already in the new list and only
appending it if it’s not present.  Since all provided test cases for
the repaired code passed, it indicates that this approach correctly
meets the problem’s requirements.

To help you create a functioning solution, focus on ensuring that
your program checks for the existence of elements in the new list
appropriately and maintains the order of the first occurrences
without excessive iterations.
[/EXP]

[HINT]Consider how you are currently checking for duplicates in your
code and whether this aligns with the goal of creating a new list
that preserves order while eliminating extras.
[/HINT]
```

Figure 3: Response given by GPT-4o-mini to the advanced prompt in Figure 2.

Appendix: Real Use-case of the Advanced Prompt by Phi-3-mini

```
[Problem Starts]
Duplicate elimination - Write a function remove_extras(lst) that...
[Problem Ends]

[Buggy Code Starts]
def remove_extras(lst):
    new = []
    for x in lst:
        if lst.count(x) > 1:
            new += []
        else:
            new = new.append(x)
    return new
[Buggy Code Ends]

Previously, a Large Language Model was asked to review the student’s
buggy code and generate a repaired program based on the problem
description.  The repaired program is as follows:

[Repaired Code Starts]
def remove_extras(lst):
    new = []
    for x in lst:
        if lst.count(x) == 1:
            new.append(x)
    return new
[Repaired Code Ends]

Nevertheless, this repaired program is not guaranteed to be correct.
For you to review whether this repaired program is correct or not,
the results of the test cases for this problem are provided below:

[Testcases results of repaired code Starts]
* 1 out of 6 testcases passed.
[Testcases results of repaired code Ends]

Based on all of the above information, create an explanation written
inside placeholders [EXP] and [/EXP] that analyzes the student’s
buggy code, the provided repaired program and the testcases results,
to come up step-by-step with the best approach to provide a hint to
the student.  After that, provide a concise single-sentence hint to
the student about one bug in the student’s buggy code, which allows
him to get closer to the correct program, but without directly giving
him the code or detailed explanations.  Output your hint between
[HINT] and [/HINT]. Make sure to always use the [HINT] and [/HINT]
placeholders with a hint inside.  Do not output anything else inside
the [HINT] placeholders that is not part of your chosen hint.  Do
not refer to the student as "the student", but with "you" if strictly
necessary.  Always start your hints with "Consider" or "Think about".
```

Figure 4: An example of the advanced prompt being used to query Phi-3-mini. Notice that the repaired program created by Phi-3-mini passes 1 out of 6 test cases, thus it’s not a completely correct repaired program. *Prompt was shortened for brevity.*

Appendix: Hint by Phi-3-mini given prompt in Figure 4

```
[EXP] The student’s buggy code attempts to remove duplicates by
checking if the count of an element in the list is greater than 1.
However, the use of ‘new += []‘ in the if condition is incorrect
because it tries to concatenate an empty list to ‘new‘, which does
not affect the list at all.  Moreover, the else clause should not
reassign ‘new‘ to the result of ‘new.append(x)‘, as this would
overwrite the list with a single element, which is not the intended
behavior.  The repaired code correctly appends unique elements to
‘new‘, but it does not preserve the relative order of the elements,
which is a requirement of the problem statement.
[/EXP]

[EXP] The test cases results indicate that the repaired program does
not pass all the test cases, suggesting that there is still an issue
with the logic that needs to be addressed.  The problem with the
repaired code is that it does not maintain the relative order of
the elements in the original list, which is a key requirement of the
problem statement.
[/EXP]

[HINT] Consider using a data structure that maintains the order of
elements as you iterate through the list and check for duplicates.
```

Figure 5: Response given by Phi-3-mini to the advanced prompt in Figure 4. Notice that Phi-3-mini fails to terminate the hint with the `[/HINT]` placeholder, eventhough it was asked for in the prompt. The provided hint however is of good quality.