Demystifying the Role of Feedback in GPT Self-Repair for Code Generation

Theo X. Olausson [1] **Jeevana Priya Inala** [2] **Chenglong Wang** [2] **Jianfeng Gao** [2] **Armando Solar-Lezama** [1]

Abstract

Large Language Models (LLMs) have shown remarkable aptitude in generating code from natural language specifications, but still struggle on challenging programming tasks. Self-repair—in which the user provides executable unit tests and the model uses these to debug and fix mistakes in its own code—may improve performance in these settings without significantly altering the way in which programmers interface with the system. However, existing studies on how and when self-repair works effectively have been limited in scope, and one might wonder how self-repair compares to keeping a software engineer in the loop to give feedback on the code model's outputs. In this paper, we analyze GPT-3.5 and GPT-4's ability to perform self-repair on APPS, a challenging dataset consisting of diverse coding challenges. We find that when the cost of generating both feedback and repaired code is taken into account, performance gains from self-repair are marginal and can only be seen with GPT-4. In contrast, when human programmers are used to provide feedback, the success rate of repair increases by as much as 57%. These findings suggest that self-repair still trails far behind what can be achieved with a feedback-giving human kept closely in the loop.

1. Introduction

Large language models (LLMs) have proven capable of generating code snippets from natural language specifications, but still struggle on complex coding challenges such as those found in competitions and professional software engineering interviews. Most attempts at improving this have centered around clustering, ranking and filtering model outputs before they are shown to the user (Li et al., 2022; Shi et al., 2022; Chen et al., 2021; Inala et al., 2022; Zhang et al., 2022). Recently, self-repair—in which the model finds explanations for and corrects mistakes in its own code after executing it against a set of user-provided tests (Figure 1)—has emerged as a popular alternative strategy (Gupta et al., 2020; Le et al., 2022; Chen et al., 2023b; Zhang et al., 2023). From the point of view of building productivity-enhancing, AI-powered programming assistants, self-repair is an attractive idea. Compared to a simple specify-sample-filter approach, the only additional work required by the user is to provide the unit tests, and yet—ideally—the performance gain might be comparable to that which would be obtained if the user was kept in the loop to provide feedback on the code at each step.

In this paper, we explore the potency of self-repair in competition-level programming by studying GPT-3.5's (Ouyang et al., 2022; OpenAI, 2022) and GPT-4's (OpenAI, 2023) ability to provide feedback on and repair their own code. We first propose a new evaluation strategy dubbed `pass@t` in which the likelihood of obtaining a correct program (with respect to the given unit tests) is weighed against the total number of tokens sampled from the model. Using this instead of the traditional `pass@k` (Chen et al., 2021; Kulal et al., 2019) metric (which weighs pass rate against the number of trials), we are able to accurately measure performance gained through self-repair against any additional work done by the model when generating feedback and carrying out the repair. We then move on to a set of experiments in which we begin by carefully studying the dynamics of the self-repair process under a range of hyper-parameters (Section 3.1); then investigate the impact of improving only the feedback stage by using a stronger feedback generation model than code generation model (Section 3.2); and, finally, carry out a brief user study in which human participants provide feedback on incorrect programs, allowing us to quantitatively and qualitatively compare model-generated self-feedback to that provided by human programmers (Section 3.3). In summary, we find that:

1. When taking the cost of doing inspection and repair into account, performance gains from self-repair can only be seen with GPT-4; for GPT-3.5, the pass rate with repair is lower than or equal to that of the baseline, no-repair approach at all budgets.

[1]MIT CSAIL [2]Microsoft Research. Correspondence to: Theo X. Olausson <theoxo@mit.edu>.

AI & HCI Workshop at the 40th International Conference on Machine Learning (ICML), Honolulu, Hawaii, USA. 2023.

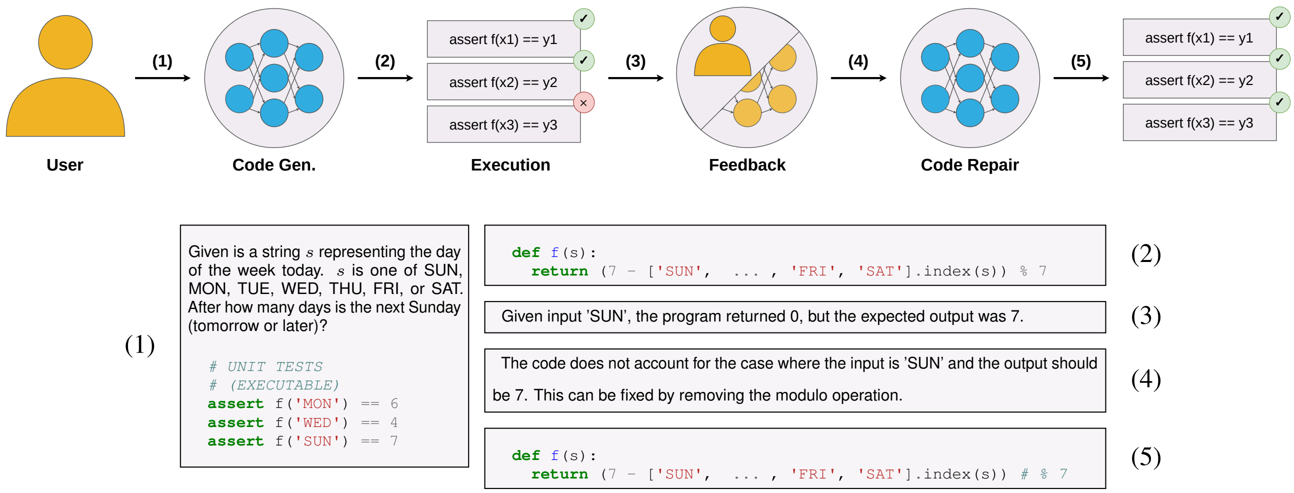


Figure 1: Self-repair with human or self-feedback. First, a user gives a specification in the form of text and a suite of unit tests (1). Then, a code model (blue) generates a program (2). The program is checked against the unit tests using a symbolic execution engine, and an error message is returned (3). In order to provide more signal to the code model, textual feedback as to *why* this happened is provided by a feedback model (yellow) *or* by the user (4). Finally, this feedback is used by the code model to repair its program (5).

2. Even for the GPT-4 model, performance gains are modest at best ($66\% \rightarrow 71\%$ pass rate with a budget of 7000 tokens, $\approx$ the cost of 45 i.i.d. GPT-4 samples) and depend on having sufficient diversity in the initial programs.

3. Replacing GPT-3.5's explanations of what is wrong with feedback produced by GPT-4 leads to better self-repair performance, even beating the baseline, no-repair GPT-3.5 approach ($50\% \rightarrow 54\%$ at 7000 tokens).

4. Replacing GPT-4's own explanations with those of a human programmer improves repair significantly, leading to a 57% increase in the number of repaired programs which pass the tests.

2. Methodology

2.1. Self-Repair Overview

As shown in Figure 1, our self-repair approach involves 4 stages: code generation, code execution, feedback generation, and code repair. We now formally define these four stages.

Code generation. Given a specification ψ, a programming model M_P first generates n_p samples i.i.d., which we denote

$$\{p_i\}_{i=1}^{n_p} \overset{i.i.d.}{\sim} M_P(\psi)$$

Code execution. Having obtained n_p initial programs, we execute all of them against the test bed. If any program passes all the tests, we stop, since a satisfying program has then been found. Otherwise, we collect the error messages $\{e_i\}_i$ returned by the execution environment. These error messages either contain the compile/runtime error information or an example input on which the program's output differs from the expected one. An example is shown in Figure 1 (component 3).

Feedback generation. The error messages returned by the execution environment are usually very high-level, providing little signal for repair. Therefore, as an intermediate step, textual feedback is generated by a feedback model (or a human-in-the-loop); Figure 1 (component 4) shows an example. Formally, in this stage, we generate n_f feedback strings, $\{f_{ij}\}_j$, for each wrong program, p_i, as follows:

$$\{f_{ij}\}_{j=1}^{n_f} \overset{i.i.d.}{\sim} M_F(\psi; p_i; e_i)$$

Having an explicit feedback generation step allows us to ablate this component so that we can study its significance in isolation.

Code repair. In the final step, for each initial program p_i and feedback f_{ij}, n_r candidate repaired programs are sampled from M_P [1]:

$$\{r_{ijk}\}_{k=1}^{n_r} \overset{i.i.d.}{\sim} M_P(\psi; p_i; e_i; f_{ij})$$

Repair tree. We call the tree of interleaved text and programs produced by this procedure—rooted in the specification ψ, then branching into initial programs p_i, each of

[1] We use the same model for both the initial code generation and the code repair, since these are fundamentally similar tasks.

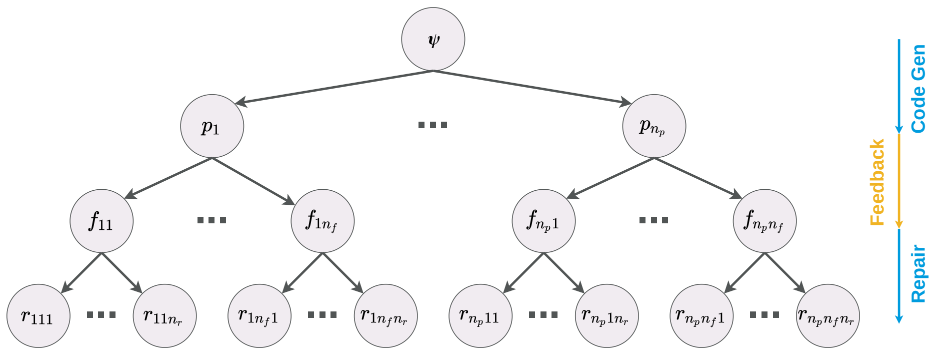


Figure 2: A repair tree begins with a specification ψ (root node), then grows into initial programs, feedback, and repairs.

which branches into feedback f_{ij} and then repairs r_{ijk}—a *repair tree* (Figure 2), T, and denote the full sampling procedure as $T \sim M(\psi)$.

Caveat: jointly sampling feedback and repair. The general framework presented above does not require the programming model and feedback model to be the same, thus allowing for the use of specialized models in the system. However, when $M_P = M_F$ we jointly generate both the feedback and the repaired program in a single API call, since both GPT-3.5 and GPT-4 have a natural tendency to interleave text and code in their responses. See Appendix E for a detailed look at how the prompt differs between this and the previous setting. Formally, we denote this as

$$\{(f_{ij}, r_{ij})\}_{j=1}^{n_{fr}} \stackrel{i.i.d.}{\sim} M_P(\psi; p_i; e_i)$$

2.2. `pass@t`: pass rate vs. token count

Since self-repair requires several dependent model invocations of non-uniform cost, this is a setting in which pass@k—the likelihood of obtaining a correct program in k i.i.d. samples—is not a suitable metric for comparing and evaluating various hyper-parameter choices of self-repair. Instead, we measure the pass rate as a function of the total number of tokens sampled from the model, a metric which we call `pass@t`.

Formally, suppose that you are given a dataset $D = \{\psi_d\}_d$ and a chosen set of values for the hyper-parameters $(M_P, M_F, n_p, n_f, n_r)$. Let $T_d^i \sim M(\psi_d)$ denote a repair tree that is sampled as described in Section 2.1 for the task ψ_d; let size(T_d^i) denote the total number of program and feedback tokens in the repair tree; and say that $T_d^i \models \psi_d$ is true if, and only if, T_d^i has at least one leaf program that satisfies the unit tests in the specification ψ_d. Then the `pass@t` metric of this choice of hyper-parameters is defined as the expected pass rate at the number of tokens which you would expect to generate with this choice of hyper-parameters:

$$\texttt{pass@t} \triangleq \mathop{\mathbb{E}}_{\substack{\psi_d \sim D \\ T_d^i \sim M(\psi_d)}} \left[T_d^i \models \psi_d \right]$$
$$\textbf{at} \quad t = \mathop{\mathbb{E}}_{\substack{\psi_d \sim D \\ T_d^i \sim M(\psi_d)}} \left[\text{size}(T_d^i) \right]$$

In our experiments, we plot bootstrapped estimates of these two quantities. To obtain these, we first generate a very large repair tree for each task specification, with: $N_p \geq n_p$ initial program samples; $N_f \geq n_f$ feedback strings per wrong program; and $N_r \geq n_r$ repair candidates per feedback string. Given a setting of (n_p, n_f, n_r), we then sub-sample (with replacement) N_t different repair trees from this frozen dataset. Finally, we compute the sample mean and standard deviation of the pass rate and the tree size over these N_t trees. Estimating the `pass@t` in this way greatly reduces the computational cost of our experiments, since we can reuse the same initial dataset to compute the estimates for all of the various choices of n_p, n_f, and n_r.

We use $N_p = 50$ for all experiments, and consider $n_p \leq 25$ for the self-repair approaches and $n_p \leq 50$ for the baseline, no-repair approach. Similarly, for the feedback strings, we use $N_f = 25$ and $n_f \leq 10$ (except for Section 3.2, in which we only consider $n_f = 1$ and therefore settle for $N_f = 10$ instead). For the repair candidates, since we do joint sampling of feedback and repair in most of our experiments, we set $N_r = n_r = 1$. Finally, we use $N_t = 1000$ for all settings.

3. Experiments

Having established the precise framework in which we will study self-repair with and without a human in the loop, we now move on to empirically answering the following research questions: (a) In the context of challenging programming puzzles, is self-repair better than i.i.d. sampling without repair for the models we consider? If so, under what hyper-parameters is self-repair most effective? (b)

Would a stronger feedback model boost the model's repair performance? (c) Would keeping a human in the loop to provide feedback unlock better repair performance even for the strongest model?

We evaluate these hypotheses on Python programming challenges from the APPS dataset (Hendrycks et al., 2021). The APPS dataset contains a diverse range of programming challenges paired with a suite of tests, making it a perfect (and challenging) setting to study self-repair in. To keep our experiments tractable, we evaluate on a subset of the APPS test set, consisting of 300 tasks. These tasks are proportionally sampled in accordance with the frequency of the different difficulty levels in the test set: 180 interview-level questions, 60 competition-level questions, and 60 introductory-level questions (listed in Appendix F). We use GPT-3.5 (Ouyang et al., 2022; OpenAI, 2022) and GPT-4 (OpenAI, 2023) as our models of choice, and implement self-repair using templated string concatenation with one-shot prompting; our prompts are given in Appendix E. When appropriate, we compare against a baseline without repair. This baseline, shown with a black line in the plots, is simply i.i.d. sampling from the corresponding model (e.g., GPT-4 when we explore whether GPT-4 is capable of self-repair). Based on preliminary experiments, we set the decoding temperature to 0.8 for all the models to encourage diverse samples.

3.1. Self-repair requires strong models and diverse initial samples

In this subsection, we consider fully self-contained repair, i.e., where one single model is used for both code/repair generation and feedback generation without any intervention from the user. To evaluate if self-repair leads to better `pass@t` than a non-repair, i.i.d. sampling-based baseline approach, we vary n_p and n_{fr}—that is, the number of initial i.i.d. base samples and joint feedback, repair samples drawn from M_P—in the range $(n_p, n_{fr}) \in \{1, 2, 5, 10, 25\} \times \{1, 3, 5, 10\}$.[2]

The results are shown in Figure 3 for GPT-3.5 and Figure 4 for GPT-4. In the left-hand subplots, the color of each dot indicates the number of initial samples (n_p), while its shape indicates the number of feedback-repair samples (n_{fr}). In the right hand plots, we show a heat-map with the two hyper-parameters along the axes, where the value in each cell indicates the pass rate with self-repair normalized by the pass rate of the baseline, non-repair approach when given the same token budget (i.e., `pass@t` at the same value of `t`). When the normalized mean pass rate is 1, this means that self-repair has the same pass rate as the non-repair, baseline approach at that same token budget; a higher value (≥ 1) means self-repair performs better than the baseline.

From the plots, we can see that for the GPT-3.5 model, the `pass@t` is lower than or equal to the corresponding baseline (black line) for all settings of n_p, n_{fr}, clearly showing that self-repair is not an effective strategy for GPT-3.5. On the other hand, for GPT-4, there are several values of n_p, n_{fr} for which the pass rate with self-repair is significantly better than that of the baseline. For example, with $n_p = 10, n_{fr} = 3$ the pass rate increases from 65% to 70%, and with $n_p = 25, n_{fr} = 1$ it increases from 65% to 71%.

Our experiments also show a clear trend with respect to the relationship between the hyper-parameters. Given a fixed number of feedback-repairs (n_{fr}), increasing the number of initial programs (n_p) (i.e., moving right along the x-axis on the heat maps) consistently leads to relative performance gains for both models. On the other hand, fixing n_p and increasing n_{fr} (i.e., moving up along the y-axis on the heat maps) does not appear to be worth the additional cost incurred, giving very marginal gains at higher budgets and even decreasing performance at lower budgets. This suggests that, given a fixed budget, the most important factor determining whether self-repair will lead to a correct program or not is the diversity of the base samples that are generated up-front, rather than the diversity of the repairs sampled. Having more initial samples increases the likelihood of there being at least one program which is close to the ideal program and, hence, can be successfully repaired.

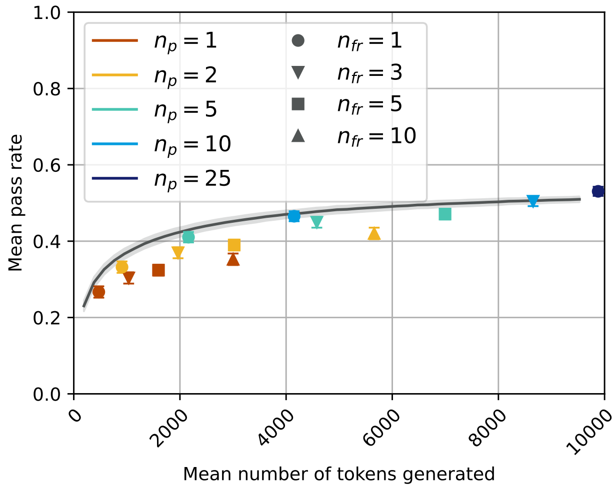
Since $n_{fr} = 1$ is the best choice for the hyper-parameter n_{fr}, we next isolate the effect of the number of initial programs, n_p, by exploring a denser set of possible values: $(n_p, n_{fr}) \in \{1, 2, ..., 24, 25\} \times \{1\}$. The plots are shown in Figure 5 for both $M_P = M_F \in \{\text{GPT-3.5}, \text{GPT-4}\}$ and the baseline, no-repair approaches. Note that since n_{fr} is fixed, in these plots, there is a direct correlation between n_p and the total number of tokens, t. Again, we see that self-repair is not an effective strategy for the GPT-3.5 model, but that it is effective for GPT-4—especially at higher values of n_p (≥ 5000), where it increases pass rate by over 5 points.

3.2. GPT-4 feedback improves GPT-3.5 repair

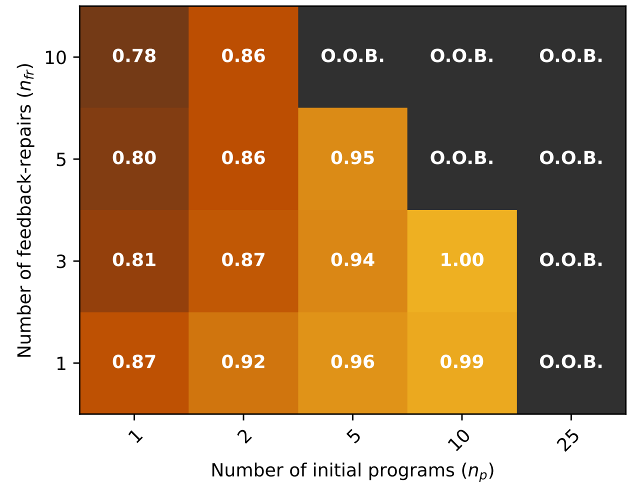
Next, we conduct an experiment in which we evaluate the impact of using a separate, stronger model to generate the feedback. This is to test the hypothesis that self-repair is held back (especially for GPT-3.5) by the model's inability to introspect and debug its own code.

For this experiment, we set M_P = GPT-3.5 and M_F = GPT-4 and vary the hyper-parameters as $(n_p, n_f, n_r) \in \{1, 2, ..., 24, 25\} \times \{1\} \times \{1\}$, similarly to the previous experiment. Note that since we are now operating in a setting in which the feedback and repair stages must be separated, we have three hyper-parameters—n_p, n_f, n_r—instead of

[2] Recall that when $M_P = M_F$, we jointly sample for n_{fr} pairs of feedback strings and repair programs instead of sampling them one after another (Section 2.1).



(a) Mean pass rate vs. number of tokens generated. Black line is i.i.d. sampling without repair from GPT-3.5. Note that the error bars are often smaller than the markers; all settings have a standard deviation of less than 1.5 absolute points on the y-axis. Results truncated at $t = 10,000$.



(b) Normalized mean pass rate relative to the (interpolated) baseline at an equivalent budget (number of tokens). Cells for which the number of tokens generated exceeds 50 samples from the GPT-3.5 baseline marked O.O.B. (out of bounds).

Figure 3: Pass rate versus number of tokens generated for various settings of n_p (number of initial programs) and n_{fr} (number of repairs sampled per program). GPT-3.5 is used for all samples, including the baseline.

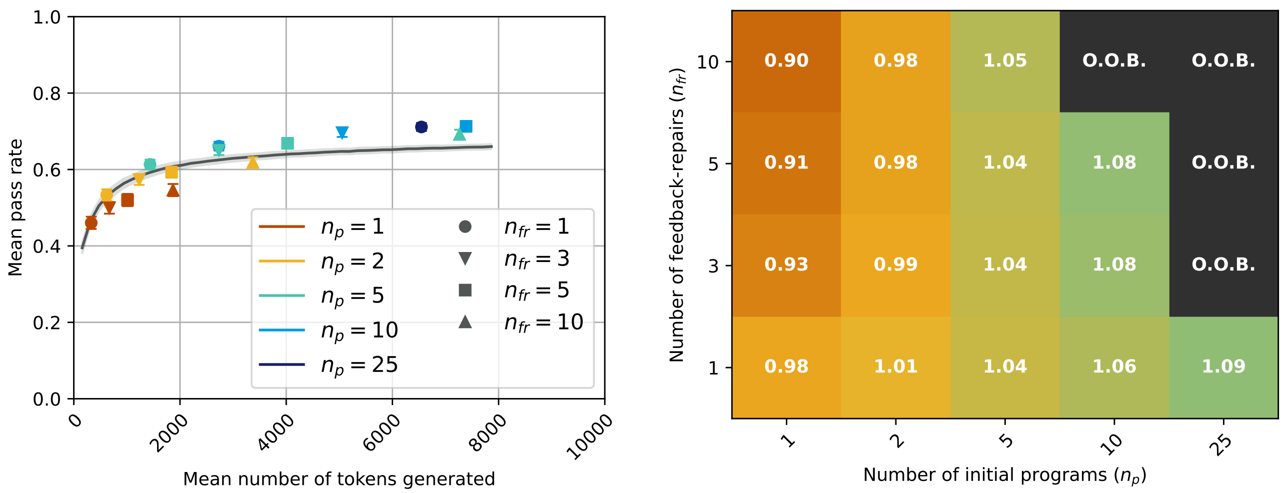
two—n_p, n_{fr} (Section 2.1). To keep the computational budget tractable, and since the variance was seen to be very low in the previous experiment, we use $N_f = 10$ instead of $N_f = 25$ for this experiment (see Section 2.2).

The results for this experiment are shown in Figure 5 (bright blue line). We observe that in terms of absolute performance, M_P = GPT-3.5, M_F = GPT-4 does break through the performance barrier and becomes marginally more efficient than i.i.d. sampling from GPT-3.5. This suggests that the textual feedback stage itself is of crucial importance, and that improving it relieves the bottleneck in GPT-3.5 self-repair.

3.3. Human feedback significantly improves the success rate of GPT-4 repair

For our final experiment, we investigate the effect of using an expert human programmer's feedback when performing repair with stronger models such as GPT-4. Note that the goal of this study is not to do a complete comparison between a human-in-the-loop approach vs. self-repair, since we do not study the additional costs involved in keeping the user in the loop at this stage. Instead, our goal is to understand how the model's ability to identify mistakes in the code compares to that of a human and how this affects downstream performance in self-repair. We thus conduct both qualitative and quantitative analyses of the impact of human feedback on self-repair, but leave it to future research to investigate how programming assistants should be designed to facilitate this style of interaction.

Data collection methodology. We first sample 20 tasks $\{\psi_i\}_{i=1}^{20}$ from the APPS test set; to make the data collection process less time-consuming for the participants of the study, we skew the distribution towards easier tasks (14 introductory; 3 interview; 3 competition). For each task ψ_i, we then sample two failing GPT-4 completions $p_{i,1}, p_{i,2}$, making for a total of $20 \cdot 2 = 40$ programs to refine. We recruit 16 participants, consisting of 15 graduate students and one professional machine learning engineer. Each participant is provided with five different base programs based on their level of experience with Python and competitive programming. Each program is taken from a distinct task; participants are never showed two different programs belonging to the same task. Participants are then asked to explain, in their own words, what the program is doing wrong. To reduce the cognitive load for participants, each program $p_{i,j}$ is accompanied by the error message $e_{i,j}$ and two feedback strings $f_{i,j,1}, f_{i,j,2}$ sampled from GPT-4. We obtain these feedback strings by randomly sampling from the feedback-repair pairs used in the previous experiments and removing the code block. Note that each of the 40 programs will be shown to two different participants, to reduce variance caused by participants' skill levels and writing style. Participants were told to spend approximately one hour on the study overall, and were compensated with a \$15 gift card. This human data collection was approved by our Institutional Review Board (IRB) and carried out



(a) Mean pass rate vs. number of tokens generated. Black line is i.i.d. sampling without repair from GPT-4. Note that the error bars are often smaller than the markers; all settings have a standard deviation of less than 1.5 absolute points on the y-axis. Results truncated at $t = 10,000$.

(b) Normalized mean pass rate relative to the (interpolated) baseline at an equivalent budget (number of tokens). Cells for which the number of tokens generated exceeds 50 samples from the GPT-4 baseline marked O.O.B. (out of bounds).

Figure 4: Pass rate versus number of tokens generated for various settings of n_p (number of initial programs) and n_{fr} (number of repairs per failing program). GPT-4 is used for all samples, including the baseline.

exclusively through an online survey. See Appendix B for a complete, concrete copy of the instructions which we provide to our participants. Five example participant responses are shown in Appendix D.

Quantitative Analysis. Having obtained two human-written pieces of feedback $h_{i,j,1}, h_{i,j,2}$ for each program $p_{i,j}$, we sample 25 repaired programs

$$\{r_l\}_{l=1}^{25} \overset{i.i.d.}{\sim} \text{GPT-4}(\psi_i; p_{i,j}; e_{i,j}; f)$$

for $f \in \{h_{i,j,1}, h_{i,j,2}, f_{i,j,1}, f_{i,j,2}\}$. That is: we ask GPT-4 to generate 25 candidate repairs for each program, conditioned on the specification, the initial program, and a feedback string which is either set to one of GPT-4's own feedback strings or to one provided by a participant. Finally, we execute all of these candidate repairs against the test bed, and take note of how often they pass.

The results are summarized in Table 1, with a complete task-by-task breakdown in Appendix C. We note first of all that the overall success rate is increased by over $1.57\times$ when we replace GPT-4's own debugging with that of our human participants. Perhaps unsurprisingly, the relative difference increases as the problems get harder, indicating that GPT-4's ability to produce accurate and useful feedback trails further behind our human participants' when the task (and code) becomes more complex.

Qualitative Analysis. In this section, we qualitatively analyze the difference between the feedback provided by the human participants and the feedback provided by GPT-4. We manually go through all of GPT-4's and the participants' feedback and note down whether the feedback: (a) seems, at a cursory glance, to be correct, or if it is obviously inaccurate; (b) explicitly suggests a small change to the code (e.g. "change the condition on line X"); (c) explicitly suggests a large change to the code (e.g. "frame the problem as min-cut instead of shortest-path"); (d) contains blocks of pseudocode or Python (which GPT-4's feedback never does, per our experiment design); or (e) expresses uncertainty (using phrases such as "unsure", "it appears", etc.).[3] Examples of each category are shown in Appendix D. We find that

- Only 2/80 human-contributed feedback strings include pseudocode or explicit Python; that is, almost all human feedback we obtain is natural language interleaved with occasional single-statement math/code expressions.
- GPT-4's feedback is much more likely to be obviously inaccurate (32/80 vs. 7/80 for human feedback).
- GPT-4 is more likely to explicitly suggest small changes (54/80 vs 42/80; 28/48 vs. 38/73 when seemingly correct), while our human participants show a slightly greater tendency to suggest high-level changes (23/80 vs. 18/80 for GPT-4; 21/73 vs. 13/48 when seemingly correct).

[3]We do not count individual single-line statements/expressions such as "$x = 5$" as pseudocode or Python.

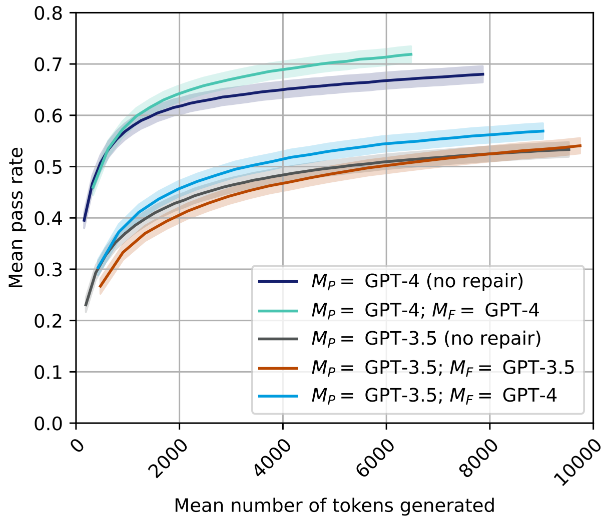


Figure 5: Mean pass rate for each model when n_{fr} (or n_f and n_r) = 1. Shaded region is ± 1 standard deviation. Complete breakdown per difficulty in Appendix A.

Difficulty	GPT-4 Feedback	Human Feedback
Introductory	42.64%	62.21%
Interview	19.33%	45.67%
Competition	3.67%	14.67%
Overall	33.30%	52.60%

Table 1: Success rate of repair with GPT-4's explanations vs. with those of our human participants.

- Our human participants sometimes express uncertainty (7/80); GPT-4 never does (0/80).

This further analysis suggests that the results in Table 1 are not due to artefacts such as our participants providing explicit code blocks which the model simply copies. Instead, the difference in performance appears to be caused by a combination of more accurate feedback, a greater ability to suggest high-level, large-scale changes to the code when needed, and our participants' ability to express their uncertainty (instead of confidently giving potentially inaccurate feedback).

4. Limitations

Firstly, to reduce computational cost, we pre-populate and then sub-sample from a single large repair tree to bootstrap our estimates of the `pass@t` rates. To minimize the risk of this introducing statistical bias, we bound n_p and n_{fr} far below N_p and N_{fr}, respectively, in our self-repair experiments. Furthermore, we note that the standard deviation is very small (≤ 1.5 points) across all values of (n_p, n_f, n_r), even those that are small enough that we have very many unique samples thereof in our pre-populated repair tree. While these measures do not completely eliminate the risks involved with bootstrapping, not performing this amortization would have required significantly larger amounts of compute.

Secondly, we assume access to an executable suite of unit tests for each task. We do not, for example, require the model to extract tests from textual specifications. While this assumption may seem out of place in the era of chat-style assistants like ChatGPT (OpenAI, 2022), it does align well with established software engineering practices like Test-Driven Development (Astels, 2003). Furthermore, techniques which automatically synthesize test cases given a specification (Li et al., 2022; Chen et al., 2023a) may relieve some of the user burden.

Finally, our study on human data did not track how much time the participants took to debug the programs. As a result, we can only evaluate the quality of the feedback (and the impact this has on repair). Although our results suggest that keeping a feedback-providing human in the loop is a very effective way of improving the quality of the generated code, further research at the intersection of HCI, AI, and program synthesis is needed to explore how programming assistants should be designed to best facilitate this style of interaction.

5. Related work

Program synthesis with large language models. The use of large language models for program synthesis has been studied extensively in the literature (Li et al., 2022; Austin et al., 2021; Chen et al., 2021; Le et al., 2022; Fried et al., 2023; Nijkamp et al., 2023; Chowdhery et al., 2022; Touvron et al., 2023; Li et al., 2023). This literature has predominantly focused on evaluating models in terms of either raw accuracy or the `pass@k` metric (Kulal et al., 2019; Chen et al., 2021), often leveraging filtering techniques based on execution (Li et al., 2022; Shi et al., 2022) or ranking (Chen et al., 2021; Inala et al., 2022; Zhang et al., 2022) to reduce the number of samples which are considered for the final answer. In contrast, our work focuses on evaluating the models from the point of view of minimizing the number of samples that need to be drawn from the model in the first place. Our work is also different in that we assume access to the full collection of input-output examples, as is typically done in inductive synthesis (Kitzelmann, 2010; Polozov & Gulwani, 2015; Gulwani et al., 2017; Chen et al., 2019a; Ellis et al., 2021). In particular, unlike some prior work (Li et al., 2022; Shi et al., 2022), we do not make a distinction between public tests used for filtering and private tests used to determine correctness, since our method does not involve filtering the outputs.

Code repair. Statistical and learning-based techniques for code repair have a rich history in both the programming languages and machine learning communities, although they have traditionally been used predominantly to repair human-written code (Long & Rinard, 2016; Bader et al., 2019; Le Goues et al., 2021; Yasunaga & Liang, 2021; Chen et al., 2019b; Mesbah et al., 2019; Wang et al., 2018). More recently, using repair as a post-processing step to improve code which was itself automatically synthesised has been used in the synthesis of both domain-specific languages (Gupta et al., 2020) and general-purpose code (Le et al., 2022; Yasunaga & Liang, 2021; 2020). Our contribution differs from most prior work in this literature in the use of textual feedback for repair, which is possible thanks to the above mentioned rise in the use of LLMs for program synthesis.

Contemporary work on LLM self-repair. There is much contemporary work seeking to self-repair with LLMs. Zhang et al. (2023) explore self-repair without natural language feedback on APPS (Hendrycks et al., 2021) using a diverse range of fine-tuned models. They also experiment with prompt-based repair using Codex (Chen et al., 2021), InCoder (Fried et al., 2023), and CodeGen (Nijkamp et al., 2023). Notably, their framework does not consider the cost associated with feedback and repair, which presents a significantly different perspective on self-repair. Similarly, Chen et al. (2023b) assess Codex's ability to self-repair across a variety of tasks, in a framework that closely resembles that which we study in this work. However, their study differs from ours in terms of the models considered, the evaluation strategy, and, most importantly, the research goal, as we specifically aim to investigate the significance of the textual feedback stage. Self-repair, or frameworks with other names that are conceptually very similar to it, has also been used in contexts outside of code generation. Peng et al. (2023) use self-repair to mitigate hallucinations and improve factual grounding in a ChatGPT-based web search assistant, in which the model revises its initial response based on self-generated feedback. Similarly, Madaan et al. (2023) present a framework in which a model iteratively provides feedback on and revises its output until a stopping criterion is reached; they apply this framework to a range of tasks, including dialogue and code optimization. Ultimately, we see our work, in which we use the novel evaluation metric `pass@t` to investigate the significance of the textual feedback stage in competition-level self-repair, as being complementary to contemporary research which uses traditional metrics to evaluate self-repair in a broader context. We are eager to see what the implications of our results will be in these other domains.

6. Conclusion

In this paper, we investigated the role of textual feedback in GPT self-repair for code generation. We presented `pass@t`, a new evaluation strategy which captures the cost of making the model provide feedback on and repair its own code, and then used this metric to show that (1) GPT-3.5 is not capable of carrying out self-repair on challenging coding tasks, and (2) while performance gains are seen with GPT-4, they are modest and rely on achieving sufficient diversity in the initial programs. Furthermore, by ablating the feedback stage we found that (3) substituting GPT-3.5's feedback with that of GPT-4 improved performance, even surpassing GPT-3.5's baseline. Finally, we carried out an experiment with human participants, in which we found that (4) replacing GPT-4's self-generated feedback with feedback provided by an experienced programmer increased the number of repaired programs which pass all unit tests by 57%. These findings suggest that while state-of-the-art code models may be showing some ability to self-repair, the performance gained therefrom still pales in comparison to that which can be achieved when a human is kept closely in the loop.

Acknowledgements

T.X. Olausson is supported by the Defense Advanced Research Projects Agency (DARPA) under the ASKEM program, award HR00112220042. T.X. Olausson was also supported through a position at Microsoft Research for part of the time period during which this work was carried out. A. Solar-Lezama is supported by the National Science Foundation (NSF) and Intel Corporation through NSF Grant CCF:2217064. This work benefited greatly from discussion with several colleagues at Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the Defense Advanced Research Projects Agency, Intel Corporation, or Microsoft Research.

References

Astels, D. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. ISBN 0131016490.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program Synthesis with Large Language Models, 2021. *arXiv preprint arXiv:2108.07732*. https://arxiv.org/abs/2108.07732.

Bader, J., Scott, A., Pradel, M., and Chandra, S. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct 2019. doi: 10.1145/3360585.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. CodeT: Code generation with generated tests. In *International Conference on Learning Representations*, 2023a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating Large Language Models Trained on Code, 2021. *arXiv preprint arXiv:2107.03374*. https://arxiv.org/abs/2107.03374.

Chen, X., Liu, C., and Song, D. Execution-Guided Neural Program Synthesis. In *International Conference on Learning Representations*, 2019a.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching Large Language Models to Self-Debug, 2023b. *arXiv preprint arXiv:2304.05128*. https://arxiv.org/abs/2304.05128.

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transaction on Software Engineering*, 2019b.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. PaLM: Scaling Language Modeling with Pathways, 2022. *arXiv preprint arXiv:2204.02311*. https://arxiv.org/abs/2204.02311.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., and Tenenbaum, J. B. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *The International Conference on Programming Language Design and Implementation*, 2021.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. InCoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*, 2023.

Gulwani, S., Polozov, O., and Singh, R. *Program Synthesis*. Foundations and Trends® in Programming Languages Series. Now Publishers, 2017. ISBN 9781680832921.

Gupta, K., Christensen, P. E., Chen, X., and Song, D. Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis. In *Advances in Neural Information Processing Systems*, 2020.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring Coding Challenge Competence With APPS. In *Advances in Neural Information Processing Systems*, 2021.

Inala, J. P., Wang, C., Yang, M., Codas, A., Encarnación, M., Lahiri, S., Musuvathi, M., and Gao, J. Fault-Aware Neural Code Rankers. In *Advances in Neural Information Processing Systems*, 2022.

Kitzelmann, E. Inductive Programming: A Survey of Program Synthesis Techniques. In *Approaches and Applications of Inductive Programming: Third International Workshop*, 2010.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. S. SPoC: Search-based Pseudocode to Code. In *Advances in Neural Information Processing Systems*, 2019.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2022.

Le Goues, C., Pradel, M., Roychoudhury, A., and Chandra, S. Automatic Program Repair. *IEEE Softw.*, 38(4): 22–27, jul 2021. ISSN 0740-7459. doi: 10.1109/MS.2021.3072577.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. StarCoder: may the source be with you!, 2023. *arXiv preprint arXiv:2305.06161*. https://arxiv.org/abs/2305.06161.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Long, F. and Rinard, M. Automatic Patch Generation by Learning Correct Code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-Refine: Iterative Refinement with Self-Feedback, 2023. *arXiv preprint arXiv:2303.17651*. https://arxiv.org/abs/2303.17651.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N., and Aftandilian, E. DeepDelta: Learning to Repair Compilation Errors. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations*, 2023.

OpenAI. Introducing ChatGPT, 2022. Blog post. https://openai.com/blog/chatgpt [Accessed 5/17/2023].

OpenAI. GPT-4 Technical Report, 2023. *arXiv preprint arXiv:2303.08774*. https://arxiv.org/abs/2303.08774.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022.

Peng, B., Galley, M., He, P., Cheng, H., Xie, Y., Hu, Y., Huang, Q., Liden, L., Yu, Z., Chen, W., and Gao, J. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*, 2023.

Polozov, O. and Gulwani, S. FlashMeta: A Framework for Inductive Program Synthesis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural Language to Code Translation with Execution. In *Empirical Methods in Natural Language Processing*, 2022.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models, 2023. *arXiv preprint arXiv:2302.13971*. https://arxiv.org/abs/2302.13971.

Wang, K., Singh, R., and Su, Z. Dynamic Neural Program Embedding for Program Repair. In *International Conference on Learning Representations*, 2018.

Yasunaga, M. and Liang, P. Graph-based, Self-supervised Program Repair from Diagnostic Feedback. In *International Conference on Machine Learning*, 2020.

Yasunaga, M. and Liang, P. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *International Conference on Machine Learning*, 2021.

Zhang, K., Li, Z., Li, J., Li, G., and Jin, Z. Self-Edit: Fault-Aware Code Editor for Code Generation, 2023. *arXiv preprint arXiv:2305.04087*. https://arxiv.org/abs/2305.04087.

Zhang, T., Yu, T., Hashimoto, T. B., Lewis, M., Yih, W.-t., Fried, D., and Wang, S. I. Coder Reviewer Reranking for Code Generation, 2022. *arXiv preprint arXiv:2211.16490*. https://arxiv.org/abs/2211.16490.

A. Self-Repair Results Per Difficulty

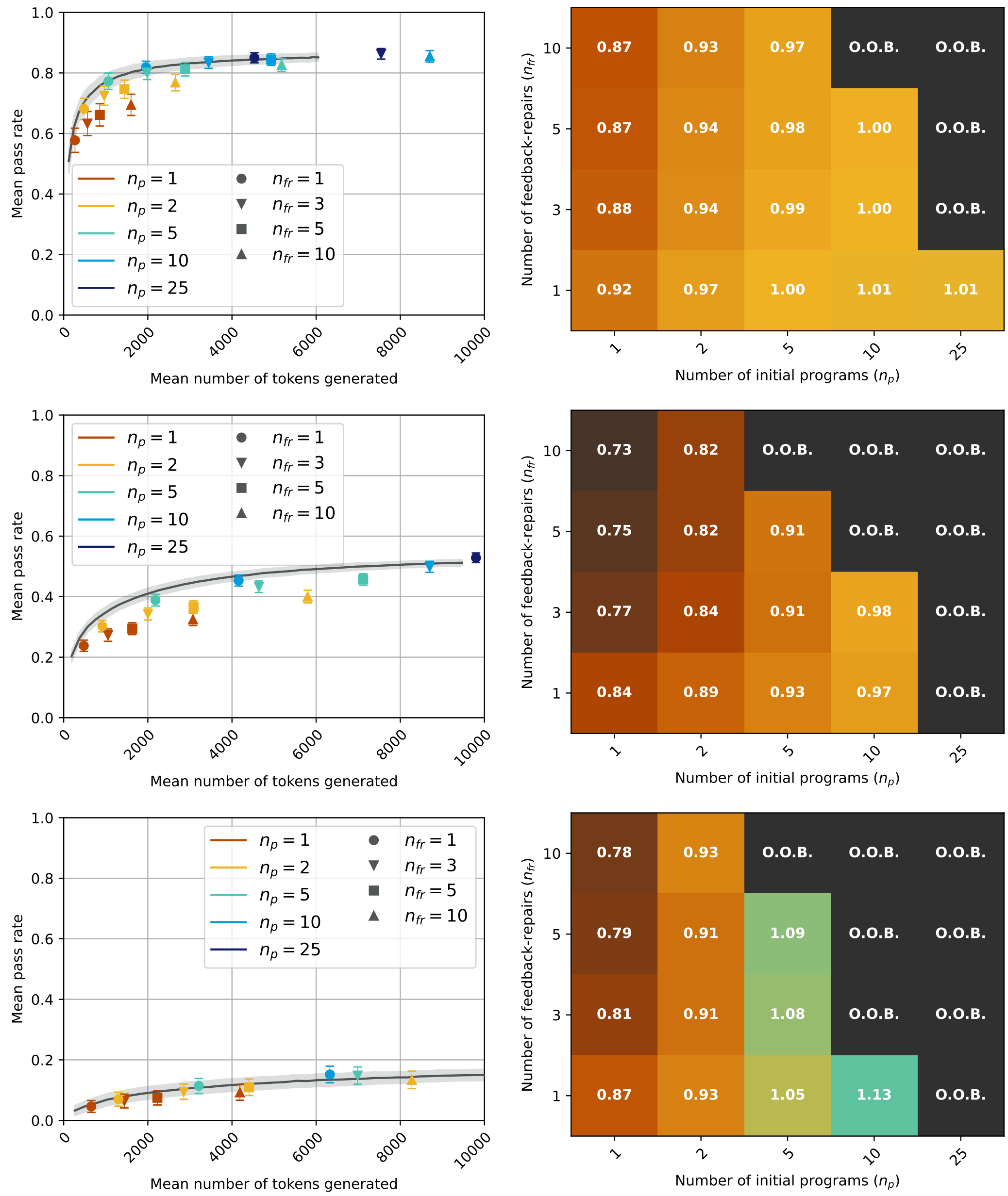


Figure 6: GPT-3.5 results from Figure 3 (Section 3.1) per difficulty (row), from top to bottom: `introductory`, `interview`, and `competition`.

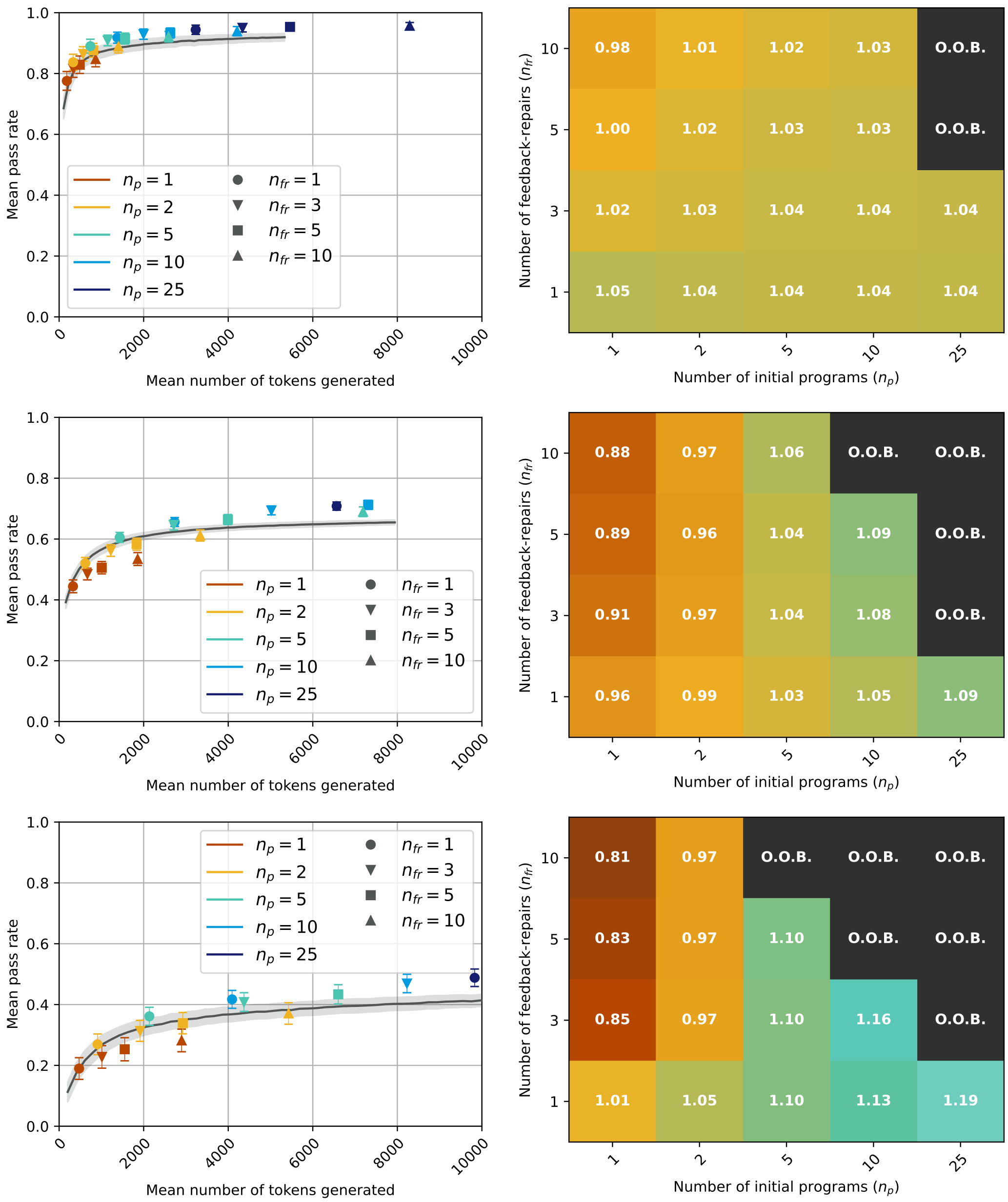


Figure 7: GPT-4 results from Figure 4 (Section 3.1) per difficulty (row), from top to bottom: `introductory`, `interview`, and `competition`.

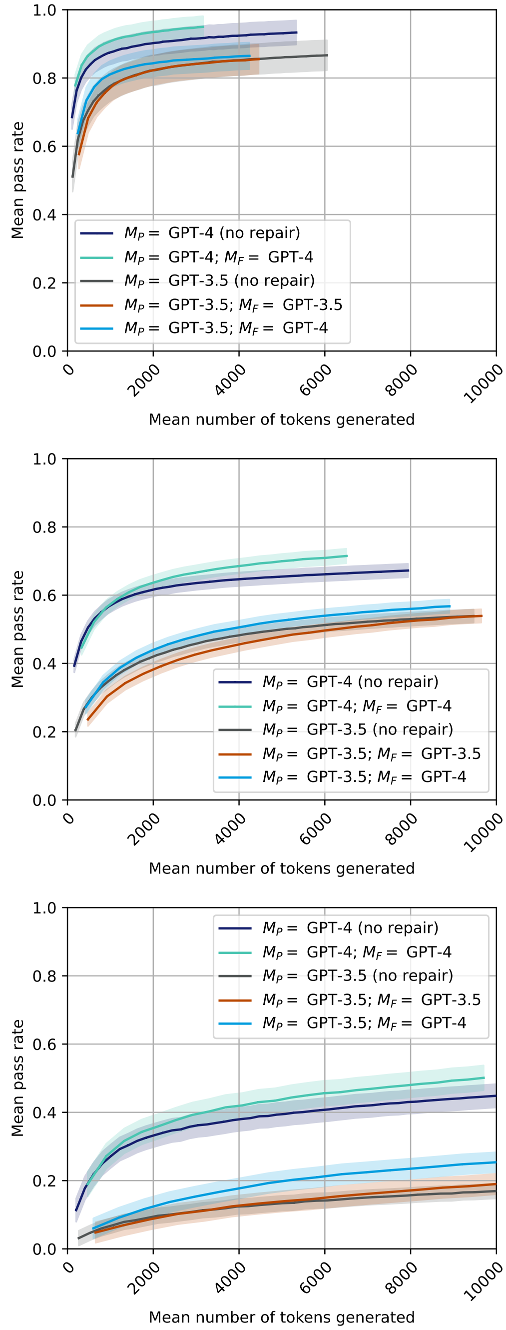
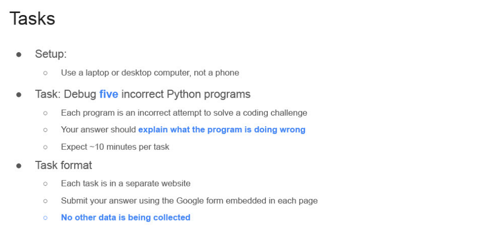
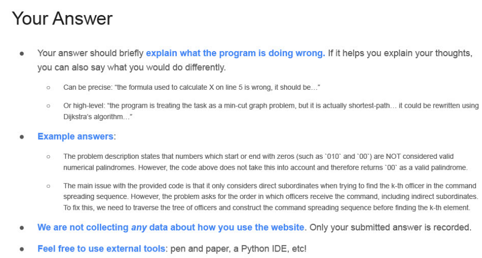


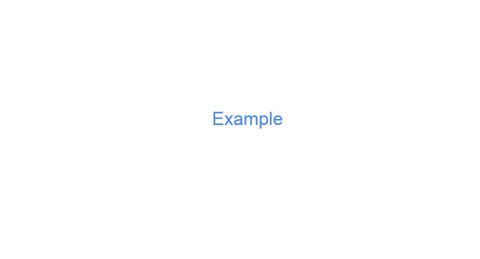
Figure 8: Results from Figure 5 (Section 3.2) per difficulty (row), from top to bottom: `introductory`, `interview`, and `competition`.

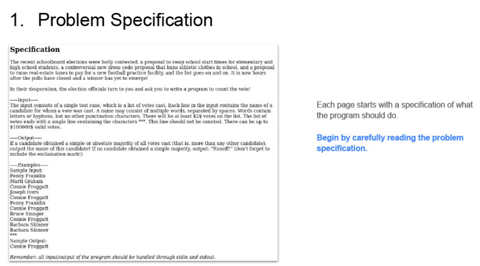
B. Human Experiment: Study Instructions

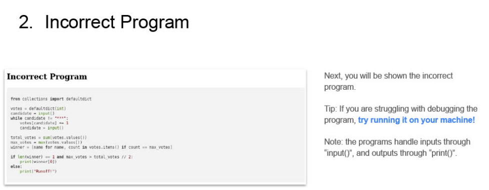
For our study on human data, participants were given a slide deck with instructions. The following ten images show the instructions, which include an example of a task shown to a participant:

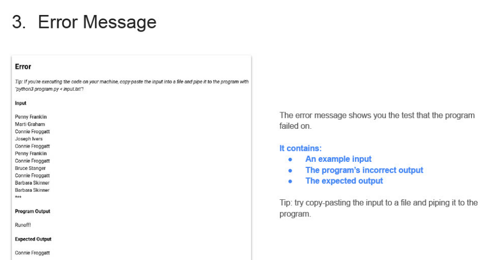


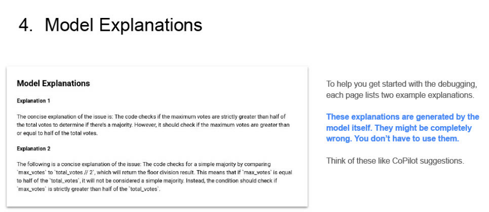


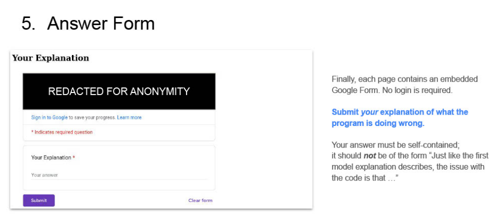


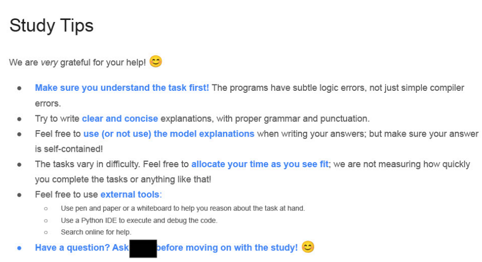


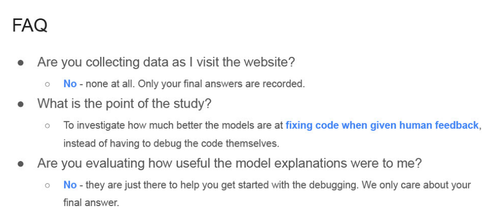












C. Human Experiment (Quantitative Analysis): Results Per Task

In the table below, we give a complete breakdown of the quantitative results presented in Section 3.3. Each cell is the number of repair candidates (out of 25) that passed all the unit tests.

Task	Difficulty	Program	GPT-4 #1	GPT-4 #2	Human #1	Human #2
2106	interview	A	7	10	10	0
		B	0	2	20	16
2673	interview	A	4	7	17	24
		B	3	25	25	25
2923	interview	A	0	0	0	0
		B	0	0	0	0
3070	competition	A	0	0	0	0
		B	3	0	5	0
3286	competition	A	2	6	10	25
		B	0	0	0	4
3754	competition	A	0	0	0	0
		B	0	0	0	0
4182	introductory	A	25	25	25	24
		B	25	0	25	25
4195	introductory	A	25	3	24	23
		B	23	25	25	25
4281	introductory	A	0	4	0	0
		B	0	0	0	0
4333	introductory	A	25	0	25	0
		B	23	24	24	25
4347	introductory	A	0	0	7	25
		B	0	0	25	25
4426	introductory	A	25	25	25	25
		B	25	25	25	25
4450	introductory	A	0	0	0	0
		B	24	0	22	24
4507	introductory	A	0	0	0	0
		B	0	0	1	0
4514	introductory	A	15	21	1	16
		B	0	0	25	0
4704	introductory	A	0	25	0	25
		B	25	25	24	23
4741	introductory	A	25	25	25	25
		B	25	25	25	25
4855	introductory	A	0	1	17	25
		B	0	2	3	23
4873	introductory	A	0	0	0	0
		B	0	0	0	18
4952	introductory	A	0	0	2	25
		B	24	8	24	21

D. Human Experiment (Qualitative Analysis): Examples

In this appendix, we give examples of each category used to classify the responses in the qualitative analysis of Section 3.3. For each category, we give one example from the human participants and one from GPT-4 (when possible). Each example contains: the textual specification of the task; the incorrect program; the error message (in the format it was shown to participants); the feedback in question; and a short comment motivating its classification.

D.1. Category (a): obviously incorrect feedback

D.1.1. GPT-4: Example

```
### SPECIFICATION
The recent schoolboard elections were hotly contested: a proposal to swap school start times for elementary and high school students,
      a controversial new dress code proposal that bans athletic clothes in school, and a proposal to raise real-estate taxes to pay
      for a new football practice facility, and the list goes on and on. It is now hours after the polls have closed and a winner has
      yet to emerge!

In their desperation, the election officials turn to you and ask you to write a program to count the vote!

-----Input-----
The input consists of a single test case, which is a list of votes cast. Each line in the input contains the name of a candidate for
      whom a vote was cast. A name may consist of multiple words, separated by spaces. Words contain letters or hyphens, but no other
      punctuation characters. There will be at least votes on the list. The list of votes ends with a single line containing the
      characters ***. This line should not be counted. There can be up to valid votes.

-----Output-----
If a candidate obtained a simple or absolute majority of all votes cast (that is, more than any other candidate), output the name of
      this candidate! If no candidate obtained a simple majority, output: "Runoff!" (don't forget to include the exclamation mark!)

-----Examples-----
Sample Input:
Penny Franklin
Marti Graham
Connie Froggatt
Joseph Ivers
Connie Froggatt
Penny Franklin
Connie Froggatt
Bruce Stanger
Connie Froggatt
Barbara Skinner
Barbara Skinner
***
Sample Output:
Connie Froggatt

### INCORRECT PROGRAM
from collections import defaultdict

votes = defaultdict(int)
candidate = input()
while candidate != "***":
    votes[candidate] += 1
    candidate = input()

total_votes = sum(votes.values())
max_votes = max(votes.values())
winner = [name for name, count in votes.items() if count == max_votes]

if len(winner) == 1 and max_votes > total_votes // 2:
    print(winner[0])
else:
    print("Runoff!")

### ERROR MESSAGE
===Input===
Penny Franklin
Marti Graham
Connie Froggatt
Joseph Ivers
Connie Froggatt
Penny Franklin
Connie Froggatt
Bruce Stanger
Connie Froggatt
Barbara Skinner
Barbara Skinner
***
===Program Output===
Runoff!
===Expected Output===
Connie Froggatt

### FEEDBACK
The concise explanation of the issue is: The code checks if the maximum votes are strictly greater than half of the total votes to
      determine if there's a majority. However, it should check if the maximum votes are greater than or equal to half of the total
      votes.
```

```
### COMMENT
The issue with the code is not that the ">" should have been a ">=", but rather that it is not allowing for a simple majority.
```

D.1.2. HUMAN: EXAMPLE

```
### SPECIFICATION
Snuke and Raccoon have a heap of N cards. The i-th card from the top has the integer a_i written on it.
They will share these cards.
First, Snuke will take some number of cards from the top of the heap, then Raccoon will take all the remaining cards.
Here, both Snuke and Raccoon have to take at least one card.
Let the sum of the integers on Snuke's cards and Raccoon's cards be x and y, respectively.
They would like to minimize |x-y|.
Find the minimum possible value of |x-y|.

-----Constraints-----
- 2 \leq N \leq 2 \times 10^5
- -10^{9} \leq a_i \leq 10^{9}
- a_i is an integer.

-----Input-----
Input is given from Standard Input in the following format:
N
a_1 a_2 ... a_{N}

-----Output-----
Print the answer.

-----Sample Input-----
6
1 2 3 4 5 6

-----Sample Output-----
1

If Snuke takes four cards from the top, and Raccoon takes the remaining two cards, x=10, y=11, and thus |x-y|=1. This is the minimum
      possible value.

### INCORRECT PROGRAM
def main():
    n = int(input())
    a = list(map(int, input().split()))

    Snuke_sum = 0
    Raccoon_sum = sum(a)

    min_diff = abs(Raccoon_sum)
    for i in range(n-1):
        Snuke_sum += a[i]
        Raccoon_sum -= a[i]
        diff = abs(Snuke_sum - Raccoon_sum)
        min_diff = min(min_diff, diff)

    print(min_diff)

main()

### ERROR MESSAGE
===Input===
2
10 -10
===Program Output===
0
===Expected Output===
20

### FEEDBACK
The code initializes the result `min_diff` to `abs(Raccoon_sum)` corresponding to the case where Raccoon takes all the cards, however
      this is incorrect because of the constraint "Snuke and Raccoon have to take at least one card". Instead just setting `min_diff`
      to negative infinity initially would fix it.

### COMMENT
This human participant was on the right track, but note that setting `min_diff` to *negative* infinity would cause it to never be
      updated inside of the for loop, and so the program would in fact always return negative infinity.
```

D.2. Category (b): feedback explicitly suggests a minor change to the code

D.2.1. GPT-4: EXAMPLE

```
### SPECIFICATION
There is a square in the xy-plane. The coordinates of its four vertices are (x_1,y_1),(x_2,y_2),(x_3,y_3) and (x_4,y_4) in
      counter-clockwise order.
(Assume that the positive x-axis points right, and the positive y-axis points up.)
Takahashi remembers (x_1,y_1) and (x_2,y_2), but he has forgot (x_3,y_3) and (x_4,y_4).
Given x_1,x_2,y_1,y_2, restore x_3,y_3,x_4,y_4. It can be shown that x_3,y_3,x_4 and y_4 uniquely exist and have integer values.
```

```
-----Constraints-----
- |x_1|,|y_1|,|x_2|,|y_2| \leq 100
- (x_1,y_1) != (x_2,y_2)
- All values in input are integers.

-----Input-----
Input is given from Standard Input in the following format:
x_1 y_1 x_2 y_2

-----Output-----
Print x_3,y_3,x_4 and y_4 as integers, in this order.

-----Sample Input-----
0 0 0 1

-----Sample Output-----
-1 1 -1 0

(0,0),(0,1),(-1,1),(-1,0) is the four vertices of a square in counter-clockwise order.
Note that (x_3,y_3)=(1,1),(x_4,y_4)=(1,0) is not accepted, as the vertices are in clockwise order.

### INCORRECT PROGRAM
x1, y1, x2, y2 = map(int, input().split())

x3 = x2 + (y2 - y1)
y3 = y2 - (x2 - x1)
x4 = x1 + (y2 - y1)
y4 = y1 - (x2 - x1)

print(x3, y3, x4, y4)

### ERROR MESSAGE
===Input===
0 0 0 1
===Program Output===
1 1 1 0
===Expected Output===
-1 1 -1 0

### FEEDBACK
The issue with the code is that it doesn't consider the counter-clockwise order of the vertices. To fix this, we need to swap the
      signs of (y2 - y1) and (x2 - x1) in the calculations.

### COMMENT
Swapping the listed signs is a concrete suggestion of a way to fix the code. Since the overall structure of the code would remain the
      same, we count this as a minor change.
```

D.2.2. HUMAN: EXAMPLE

```
### SPECIFICATION
Given is a string S representing the day of the week today.
S is SUN, MON, TUE, WED, THU, FRI, or SAT, for Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday, respectively.
After how many days is the next Sunday (tomorrow or later)?

-----Constraints-----
- S is SUN, MON, TUE, WED, THU, FRI, or SAT.

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

-----Output-----
Print the number of days before the next Sunday.

-----Sample Input-----
SAT

-----Sample Output-----
1

It is Saturday today, and tomorrow will be Sunday.

### INCORRECT PROGRAM
S = input()
days = ['SUN', 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT']
print((7 - days.index(S)) % 7)

### ERROR MESSAGE
===Input===
SUN
===Program Output===
0
===Expected Output===
7

### FEEDBACK
The logic is incorrect. There should be no modulo operator.

### COMMENT
Suggesting that the modulo operator should be removed is a relatively small, but explicit, change.
```

D.3. Category (c): feedback explicitly suggests a large change to the code

D.3.1. GPT-4: Example

```
### SPECIFICATION
Takahashi is about to assemble a character figure, consisting of N parts called Part 1, Part 2, ..., Part N and N-1 connecting
      components. Parts are distinguishable, but connecting components are not.
Part i has d_i holes, called Hole 1, Hole 2, ..., Hole d_i, into which a connecting component can be inserted. These holes in the
      parts are distinguishable.
Each connecting component will be inserted into two holes in different parts, connecting these two parts. It is impossible to insert
      multiple connecting components into a hole.
The character figure is said to be complete when it has the following properties:
- All of the N-1 components are used to connect parts.
- Consider a graph with N vertices corresponding to the parts and N-1 undirected edges corresponding to the pairs of vertices
      connected by a connecting component. Then, this graph is connected.
Two ways A and B to make the figure complete are considered the same when the following is satisfied: for every pair of holes, A uses
      a connecting component to connect these holes if and only if B uses one to connect them.
Find the number of ways to make the figure complete. Since the answer can be enormous, find the count modulo 998244353.

-----Constraints-----
- All values in input are integers.
- 2 \leq N \leq 2 \times 10^5
- 1 \leq d_i < 998244353

-----Input-----
Input is given from Standard Input in the following format:
N
d_1 d_2 \cdots d_N

-----Output-----
Print the answer.

-----Sample Input-----
3
1 1 3

-----Sample Output-----
6

One way to make the figure complete is to connect Hole 1 in Part 1 and Hole 3 in Part 3 and then connect Hole 1 in Part 2 and Hole 1
      in Part 3.

### INCORRECT PROGRAM
def main():
    N = int(input())
    d = list(map(int, input().split()))

    MOD = 998244353

    fact = [1] * (N+1)
    for i in range(1, N+1):
        fact[i] = fact[i-1] * i % MOD

    ans = 0
    for ki in range(N):
        ans += d[ki] * fact[ki] % MOD * fact[N-ki-1] % MOD
        ans %= MOD

    ans = ans * fact[N-1] % MOD

    print(ans)

main()

### ERROR MESSAGE
===Input===
3
1 1 3
===Program Output===
18
===Expected Output===
6

### FEEDBACK
The following is a concise explanation of the issue: The code above calculates the total number of ways to connect the parts, but it
      does not ensure that each connecting component is used only once. This results in overcounting the number of ways to make the
      figure complete. To fix the issue, we need to ensure that each connecting component is used exactly once.

### COMMENT
The feedback suggests changing the code so that each connecting component is used exactly once, which does not seem to be taken into
      account in the current approach.
Changing the code to satisfy this part of the specification seems to require a major rewrite.
```

D.3.2. Human: Example

```
### SPECIFICATION
Oh, New Year. The time to gather all your friends and reflect on the heartwarming events of the past year...

$n$ friends live in a city which can be represented as a number line. The $i$-th friend lives in a house with an integer coordinate
      $x_i$. The $i$-th friend can come celebrate the New Year to the house with coordinate $x_i-1$, $x_i+1$ or stay at $x_i$. Each
      friend is allowed to move no more than once.

For all friends $1 \le x_i \le n$ holds, however, they can come to houses with coordinates $0$ and $n+1$ (if their houses are at $1$
      or $n$, respectively).

For example, let the initial positions be $x = [1, 2, 4, 4]$. The final ones then can be $[1, 3, 3, 4]$, $[0, 2, 3, 3]$, $[2, 2, 5,
      5]$, $[2, 1, 3, 5]$ and so on. The number of occupied houses is the number of distinct positions among the final ones.

So all friends choose the moves they want to perform. After that the number of occupied houses is calculated. What is the minimum and
      the maximum number of occupied houses can there be?


-----Input-----

The first line contains a single integer $n$ ($1 \le n \le 2 \cdot 10^5$) - the number of friends.

The second line contains $n$ integers $x_1, x_2, \dots, x_n$ ($1 \le x_i \le n$) - the coordinates of the houses of the friends.


-----Output-----

Print two integers - the minimum and the maximum possible number of occupied houses after all moves are performed.

-----Examples-----
Input
4
1 2 4 4
Output
2 4
Input
9
1 1 8 8 8 4 4 4 4

Output
3 8

Input
7
4 3 7 1 4 3 3

Output
3 6



-----Note-----

In the first example friends can go to $[2, 2, 3, 3]$. So friend $1$ goes to $x_1+1$, friend $2$ stays at his house $x_2$, friend $3$
      goes to $x_3-1$ and friend $4$ goes to $x_4-1$. $[1, 1, 3, 3]$, $[2, 2, 3, 3]$ or $[2, 2, 4, 4]$ are also all valid options to
      obtain $2$ occupied houses.

For the maximum number of occupied houses friends can go to $[1, 2, 3, 4]$ or to $[0, 2, 4, 5]$, for example.

### INCORRECT PROGRAM
from collections import Counter

def main():
    n = int(input())
    x = list(map(int, input().split()))

    cnt = Counter(x)
    max_houses = 0
    for pos in range(1, n + 1):
        if cnt[pos]:
            max_houses += 1

    occupied = [0] * (n + 2)
    for pos in x:
        if not occupied[pos - 1]:
            occupied[pos - 1] = 1
        elif not occupied[pos]:
            occupied[pos] = 1
        elif not occupied[pos + 1]:
            occupied[pos + 1] = 1

    min_houses = sum(occupied)

    print(min_houses, max_houses)

main()

### ERROR MESSAGE
===Input===
4
1 2 4 4
===Program Output===
```

```
4 3
===Expected Output===
2 4

### FEEDBACK
The code calculates both the minimum and maximum count of houses incorrectly. The correct solution requires conducting a tree search
     to determine the true maximum and minimum. This can be further optimized by first sorting the list of occupied houses and using
     a sliding window to process independent segments of the number line.

### COMMENT
The feedback suggests conducting a tree search to determine the max and min, as well as a way to optimize this search.
These constitute major changes to the program as it is currently written.
```

D.4. Category (d): Feedback contains blocks of pseudocode or Python code

D.4.1. GPT-4: EXAMPLE

GPT-4's feedback never contains blocks of Python per our experiment design, in which the code block is stripped (Section 3.3). We find that GPT-4 never utilizes blocks of pseudocode, either.

D.4.2. HUMAN: EXAMPLE

```
### SPECIFICATION
Given is a string S representing the day of the week today.
S is SUN, MON, TUE, WED, THU, FRI, or SAT, for Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday, respectively.
After how many days is the next Sunday (tomorrow or later)?

-----Constraints-----
- S is SUN, MON, TUE, WED, THU, FRI, or SAT.

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

-----Output-----
Print the number of days before the next Sunday.

-----Sample Input-----
SAT

-----Sample Output-----
1

It is Saturday today, and tomorrow will be Sunday.

### INCORRECT PROGRAM
day = input()
days = ['SUN', 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT']
index = days.index(day)
result = (7 - index) % 7
print(result)

### ERROR MESSAGE
===Input===
SUN
===Program Output===
0
===Expected Output===
7

### FEEDBACK
before returning the result, we should just check to see if result ==0.
so we just need to add:
if result == 0:
    result = 7

### COMMENT
The feedback contains a block of Python code that the model can simply copy in order to fix the program.
```

D.5. Category (e): feedback which expresses uncertainty

D.5.1. GPT-4: EXAMPLE

In the $20 \cdot 2 \cdot 2 = 80$ *examples we study, we do not find that GPT-4 ever expresses uncertainty.*

D.5.2. HUMAN: EXAMPLE

```
### SPECIFICATION
Two kingdoms had been at war for a long time, until the emperor intervened to bring an end to the conflict. The territory in question
     comprises an $M$ by $N$ rectangular grid. At the emperor's insistence, the two kings have withdrawn their troops until no two
     opposing troops are in adjacent squares of the map (adjacent being horizontal or vertical - diagonal is not considered).
```

```
The emperor proposes to designate certain squares of the map as neutral territory. Neither king will be allowed to move troops into
      those squares, and the emperor's own forces will patrol them to be sure that both kings observe these rules.

The emperor is frugal and does not want to commit more soldiers to this effort than absolutely necessary. His generals have marked
      each square of the map with the number of soldiers required to secure that square. What remains is to choose which of those
      squares should be patrolled.

Write a program to determine the minimum number of soldiers that the emperor will need to be deploy to guarantee that the troops of
      one kingdom cannot move, in one or more steps, into squares occupied by the troops of the second kingdom (moving horizontally or
      vertically) without encountering the emperor's own soldiers.

-----Input-----
Input begins with a line containing $2$ integers, $w$ and $h$, denoting the width and height of the map, where $1 \leq w, h \leq 40$.

This is followed by $h$ lines. Each line contains $w$ characters, left justified. These characters will be 'A' or 'B', designating a
      position held by king A or king B, or a single numeric digit, designating a currently unoccupied position that can be secured by
      the use of that number of soldiers. For example, a '2' would indicate that two soldiers must be deployed to that square to
      secure it against passage of other troops. A '0' indicates terrain that is impassible - the emperor need not commit soldiers
      there because the kingdom troops cannot pass through that square.

No 'A' will be adjacent, horizontally or vertically, to any 'B'.

There will be at least one 'A' and one 'B' in the input.

-----Output-----
Print a single line containing an integer denoting the minimum number of soldiers that the emperor must deploy to guarantee that there
      is no open path between any 'A' position and any 'B' position, using any combination of horizontal or vertical moves.

-----Examples-----
Sample Input:
8 5
A11111AA
AA7B111A
111BB111
11BBB111
11BBB11B
Sample Output:
13

### INCORRECT PROGRAM
def main():
    w, h = map(int, input().split())
    grid = [input() for _ in range(h)]

    def bfs(x, y, z):
        dist = [[float("inf")] * w for _ in range(h)]
        dist[x][y] = 0
        q = [(x, y)]
        for x, y in q:
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < h and 0 <= ny < w and dist[nx][ny] == float("inf"):
                    if grid[nx][ny] == z:
                        dist[nx][ny] = dist[x][y]
                        q.append((nx, ny))
                    if "0" <= grid[nx][ny] <= "9":
                        dist[nx][ny] = dist[x][y] + int(grid[nx][ny])
                        q.append((nx, ny))
        return dist

    ans = float("inf")
    for i in range(h):
        for j in range(w):
            if grid[i][j] == "A":
                ans = min(ans, min(x for row in bfs(i, j, "B") for x in row))
    print(ans)

main()

### ERROR MESSAGE
===Input===
8 5
A11111AA
AA7B111A
111BB111
11BBB111
11BBB11B
===Program Output===
0
===Expected Output===
13

### FEEDBACK
This code appears to be finding the minimum "distance" between an A position and a B position, where distance is measured in the
      number of soldiers required to secure the intermediate positions. This isn't really addressing the question at all, because even
      occupying the positions along some path from a particular A -> B won't necessarily block the kings from reaching each other from
      those positions through a different, longer path. You probably need some sort of min-cut algorithm to divide the graph into two
      connected components, one each for A and B nodes.

### COMMENT
```

```
The feedback expresses mild uncertainty in two aspects. First, the user indicates that they are not completely certain in their
      understanding of the code's current behaviour by using the phrase "appears to be". Then, they express uncertainty in their
      suggestion for what to do instead, saying that one *probably* needs some sort of min-cut algorithm.
```

E. Prompts

In this appendix, we describe the prompting structure used for our experiments. All of our experiments use one-shot prompting, in which a single example is given in the prompt before the desired task.

For initial code generation (the first sample from M_P), we use different prompts for the two types of tasks in APPS: call-based tasks, in which the desired program should take the input as a parameter to a function and return the output in the function's return statement; and stdio-based tasks, in which inputs should be read from stdin and outputs should be written to stdout. These prompts are shown in Listing 1 and 2, respectively. The example tasks and programs were taken from APPS' training set.

For feedback samples (i.e., samples from M_F), we use the prompt in Listing 3. This prompt contains an example in which the user provides the textual specification, the incorrect program and the error message, and the assistant generates feedback. Similarly, for repair samples (i.e., samples from M_P which follow M_F) we use the prompt in Listing 4, in which the user also supplies the feedback, and the assistant returns only the fixed version of the program. Finally, for joint feedback-repair samples (i.e., when sampling $(f, r) \sim M_P$), we use the prompt in Listing 6. This prompt combines the prompts from Listing 4 and 5 into one prompt, in which the assistant returns both the feedback and the fixed program. In all of these prompts, the specification used was taken from APPS' training set, while the programs and the feedback were constructed manually.

Listing 1: Code generation prompt for call-based tasks.

````
=====system=====
You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program
      that matches the specification and passes all tests. You will NOT return anything except for the program. Put your fixed program
      within code delimiters, for example: ```python
# YOUR CODE HERE
```.
=====user=====
### QUESTION
A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward. Examples of
      numerical palindromes are: `2332, 110011, 54322345`

For a given number ```num```, write a function which returns an array of all the numerical palindromes contained within each number.
      The array should be sorted in ascending order and any duplicates should be removed.

In this kata, single digit numbers and numbers which start or end with zeros (such as `010` and `00`) are **NOT** considered valid
      numerical palindromes.

If `num` contains no valid palindromes, return `"No palindromes found"`.
Otherwise, return `"Not valid"` if the input is not an integer or is less than `0`.


## Examples

```
palindrome(1221)      =>  [22, 1221]
palindrome(34322122)  =>  [22, 212, 343, 22122]
palindrome(1001331)   =>  [33, 1001, 1331]
palindrome(1294)      =>  "No palindromes found"
palindrome("1221")    =>  "Not valid"
```

---

### Other Kata in this Series:

Numerical Palindrome #1
Numerical Palindrome #1.5
Numerical Palindrome #2
Numerical Palindrome #3
Numerical Palindrome #3.5
Numerical Palindrome #4
Numerical Palindrome #5
### Use Call-Based Format
=====assistant=====
```python
def is_pal(s):
    return s == s[::-1]

def palindrome(num):
    if not isinstance(num, int) or num < 0:
        return 'Not valid'
    s = str(num)
    pals = set()
    for i, ch in enumerate(s):
        if ch == '0':
            continue
````

```
        for j in range(i + 2, len(s) + 1):
            test = s[i:j]
            if is_pal(test):
                pals.add(test)
    return sorted(int(x) for x in pals) or 'No palindromes found'
```

Listing 2: Code generation prompt for stdio-based tasks.

````
=====system=====
You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program
      that matches the specification and passes all tests. You will NOT return anything except for the program. Put your fixed program
      within code delimiters, for example: ```python
# YOUR CODE HERE
```.
=====user=====
### QUESTION
Polycarp has $n$ different binary words. A word called binary if it contains only characters '0' and '1'. For example, these words are
      binary: "0001", "11", "0" and "0011100".

Polycarp wants to offer his set of $n$ binary words to play a game "words". In this game, players name words and each next word
      (starting from the second) must start with the last character of the previous word. The first word can be any. For example,
      these sequence of words can be named during the game: "0101", "1", "10", "00", "00001".

Word reversal is the operation of reversing the order of the characters. For example, the word "0111" after the reversal becomes
      "1110", the word "11010" after the reversal becomes "01011".

Probably, Polycarp has such a set of words that there is no way to put them in the order correspondent to the game rules. In this
      situation, he wants to reverse some words from his set so that:  the final set of $n$ words still contains different words (i.e.
      all words are unique);  there is a way to put all words of the final set of words in the order so that the final sequence of $n$
      words is consistent with the game rules.

Polycarp wants to reverse minimal number of words. Please, help him.


-----Input-----

The first line of the input contains one integer $t$ ($1 \le t \le 10\^4$) - the number of test cases in the input. Then $t$ test
      cases follow.

The first line of a test case contains one integer $n$ ($1 \le n \le 2\cdot10^5$) - the number of words in the Polycarp's set. Next
      $n$ lines contain these words. All of $n$ words aren't empty and contains only characters '0' and '1'. The sum of word lengths
      doesn't exceed $4\cdot10^6$. All words are different.

Guaranteed, that the sum of $n$ for all test cases in the input doesn't exceed $2\cdot10^5$. Also, guaranteed that the sum of word
      lengths for all test cases in the input doesn't exceed $4\cdot10^6$.


-----Output-----

Print answer for all of $t$ test cases in the order they appear.

If there is no answer for the test case, print -1. Otherwise, the first line of the output should contain $k$ ($0 \le k \le n$) - the
      minimal number of words in the set which should be reversed. The second line of the output should contain $k$ distinct integers
      - the indexes of the words in the set which should be reversed. Words are numerated from $1$ to $n$ in the order they appear. If
      $k=0$ you can skip this line (or you can print an empty line). If there are many answers you can print any of them.


-----Example-----
Input
4
4
0001
1000
0011
0111
3
010
101
0
2
00000
00001
4
01
001
0001
00001

Output
1
3
-1
0

2
1 2
### Use Standard Input Format (read inputs with `input()`, write results with `print()`)
=====assistant=====
```python
````

```
for _ in range(int(input())):
    n = int(input())
    mass = []
    zo = 0
    oz = 0
    zz = 0
    oo = 0
    ozs = []
    zos = []
    ozss = set()
    zoss = set()
    for j in range(n):
        k = input()
        mass.append(k)
        if k[0] == '0' and k[-1] == '1':
            zoss.add(k)
            zos.append(j + 1)
            zo += 1
        elif k[0] == '1' and k[-1] == '0':
            ozss.add(k)
            ozs.append(j + 1)
            oz += 1
        elif k[0] == '0' and k[-1] == '0':
            zz += 1
        else:
            oo += 1
    if zz and oo and not oz and not zo:
        print(-1)
        continue
    else:
        if zo > oz:
            print((zo - oz) // 2)
            ans = []
            need = (zo - oz) // 2
            i = 0
            while need:
                zzz = mass[zos[i] - 1][len(mass[zos[i] - 1]) - 1:: -1]
                if zzz not in ozss:
                    ans.append(zos[i])
                    need -= 1
                i += 1
            print(*ans)
        else:
            print((oz - zo) // 2)
            ans = []
            need = (oz - zo) // 2
            i = 0
            while need:
                zzz = mass[ozs[i] - 1][len(mass[ozs[i] - 1]) - 1:: -1]
                if zzz not in zoss:
                    ans.append(ozs[i])
                    need -= 1
                i += 1
            print(*ans)

```

```
```

Listing 3: Feedback-only prompt.

````
=====system=====
You are a helpful programming assistant and an expert Python programmer. You are helping a user debug a program. The user has written
      some code, but it has some errors and is not passing the tests. You will help the user by giving a concise (at most 2-3
      sentences) textual explanation of what is wrong with the code. You will *not* generate any code, because the user wants to fix
      the code themselves.
=====user=====
### QUESTION
A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward. Examples of
      numerical palindromes are: `2332, 110011, 54322345`

For a given number ```num```, write a function which returns an array of all the numerical palindromes contained within each number.
      The array should be sorted in ascending order and any duplicates should be removed.

In this kata, single digit numbers and numbers which start or end with zeros (such as `010` and `00`) are **NOT** considered valid
      numerical palindromes.

If `num` contains no valid palindromes, return `"No palindromes found"`.
Otherwise, return `"Not valid"` if the input is not an integer or is less than `0`.


## Examples

```
palindrome(1221)      =>  [22, 1221]
palindrome(34322122)  =>  [22, 212, 343, 22122]
palindrome(1001331)   =>  [33, 1001, 1331]
palindrome(1294)      =>  "No palindromes found"
palindrome("1221")    =>  "Not valid"
```

---
````

````
### Other Kata in this Series:

Numerical Palindrome #1
Numerical Palindrome #1.5
Numerical Palindrome #2
Numerical Palindrome #3
Numerical Palindrome #3.5
Numerical Palindrome #4
Numerical Palindrome #5
### Use Call-Based Format
### INCORRECT PYTHON CODE
```python
def is_pal(s):
    return s == s[::-1]

def palindrome(num):
    if not isinstance(num, int) or num < 0:
        return 'Not valid'
    s = str(num)
    pals = set()
    for i, ch in enumerate(s):
        for j in range(i + 2, len(s) + 1):
            test = s[i:j]
            if is_pal(test):
                pals.add(test)
    return sorted(int(x) for x in pals) or 'No palindromes found'
```
The code above is wrong and contains a bug. Given input "1001331" the output was "['0', '33', '1001', '1331']" but the expected output
      was "['33', '1001', '1331']".
=====assistant=====
The following is a concise explanation of the issue: The problem description states that numbers which start or end with zeros (such
      as `010` and `00`) are NOT considered valid numerical palindromes. However, the code above does not take this into account and
      therefore returns `00` as a valid palindrome. This can be fixed by checking if the first or last character is `0` before adding
      the string to the set of palindromes.
````

Listing 4: Repair-only prompt.

````
=====system=====
You are a helpful programming assistant and an expert Python programmer. You are helping a user write a program to solve a problem.
      The user has written some code, but it has some errors and is not passing the tests. The user has spent some time debugging the
      program and will provide you with a concise textual explanation of what is wrong with the code. You will use this explanation to
      generate a fixed version of the program. Put your fixed program within code delimiters, for example: ```python
# YOUR CODE HERE
```.
=====user=====
### QUESTION
A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward. Examples of
      numerical palindromes are: `2332, 110011, 54322345`

For a given number ```num```, write a function which returns an array of all the numerical palindromes contained within each number.
      The array should be sorted in ascending order and any duplicates should be removed.

In this kata, single digit numbers and numbers which start or end with zeros (such as `010` and `00`) are **NOT** considered valid
      numerical palindromes.

If `num` contains no valid palindromes, return `"No palindromes found"`.
Otherwise, return `"Not valid"` if the input is not an integer or is less than `0`.


## Examples

```
palindrome(1221)      =>  [22, 1221]
palindrome(34322122)  =>  [22, 212, 343, 22122]
palindrome(1001331)   =>  [33, 1001, 1331]
palindrome(1294)      =>  "No palindromes found"
palindrome("1221")    =>  "Not valid"
```

---

### Other Kata in this Series:

Numerical Palindrome #1
Numerical Palindrome #1.5
Numerical Palindrome #2
Numerical Palindrome #3
Numerical Palindrome #3.5
Numerical Palindrome #4
Numerical Palindrome #5
### Use Call-Based Format
### INCORRECT PYTHON CODE
```python
def is_pal(s):
    return s == s[::-1]

def palindrome(num):
    if not isinstance(num, int) or num < 0:
        return 'Not valid'
````

````
    s = str(num)
    pals = set()
    for i, ch in enumerate(s):
        for j in range(i + 2, len(s) + 1):
            test = s[i:j]
            if is_pal(test):
                pals.add(test)
    return sorted(int(x) for x in pals) or 'No palindromes found'
```
The code above is wrong and contains a bug. Given input "1001331" the output was "['0', '33', '1001', '1331']" but the expected output
      was "['33', '1001', '1331']".
The following is a concise explanation of the issue: The problem description states that numbers which start or end with zeros (such
      as `010` and `00`) are NOT considered valid numerical palindromes. However, the code above does not take this into account and
      therefore returns `00` as a valid palindrome. This can be fixed by checking if the first or last character is `0` before adding
      the string to the set of palindromes.
=====assistant=====
### FIXED PYTHON CODE
```python
def is_pal(s):
    return s == s[::-1]

def palindrome(num):
    if not isinstance(num, int) or num < 0:
        return 'Not valid'
    s = str(num)
    pals = set()
    for i, ch in enumerate(s):
        if ch == '0':
            continue
        for j in range(i + 2, len(s) + 1):
            test = s[i:j]
            if is_pal(test):
                pals.add(test)
    return sorted(int(x) for x in pals) or 'No palindromes found'
```
````

Listing 5: Joint feedback-repair prompt.

````
=====system=====
You are a helpful programming assistant and an expert Python programmer. You are helping a user write a program to solve a problem.
      The user has written some code, but it has some errors and is not passing the tests. You will help the user by first giving a
      concise (at most 2-3 sentences) textual explanation of what is wrong with the code. After you have pointed out what is wrong
      with the code, you will then generate a fixed version of the program. Put your fixed program within code delimiters, for
      example: ```python
# YOUR CODE HERE
```.
=====user=====
### QUESTION
A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward. Examples of
      numerical palindromes are: `2332, 110011, 54322345`

For a given number ```num```, write a function which returns an array of all the numerical palindromes contained within each number.
      The array should be sorted in ascending order and any duplicates should be removed.

In this kata, single digit numbers and numbers which start or end with zeros (such as `010` and `00`) are **NOT** considered valid
      numerical palindromes.

If `num` contains no valid palindromes, return `"No palindromes found"`.
Otherwise, return `"Not valid"` if the input is not an integer or is less than `0`.


## Examples

```
palindrome(1221)      =>  [22, 1221]
palindrome(34322122)  =>  [22, 212, 343, 22122]
palindrome(1001331)   =>  [33, 1001, 1331]
palindrome(1294)      =>  "No palindromes found"
palindrome("1221")    =>  "Not valid"
```

---

### Other Kata in this Series:

Numerical Palindrome #1
Numerical Palindrome #1.5
Numerical Palindrome #2
Numerical Palindrome #3
Numerical Palindrome #3.5
Numerical Palindrome #4
Numerical Palindrome #5
### Use Call-Based Format
### INCORRECT PYTHON CODE
```python
def is_pal(s):
    return s == s[::-1]

def palindrome(num):
    if not isinstance(num, int) or num < 0:
````
````

```
        return 'Not valid'
    s = str(num)
    pals = set()
    for i, ch in enumerate(s):
        for j in range(i + 2, len(s) + 1):
            test = s[i:j]
            if is_pal(test):
                pals.add(test)
    return sorted(int(x) for x in pals) or 'No palindromes found'
```
The code above is wrong and contains a bug. Given input "1001331" the output was "['0', '33', '1001', '1331']" but the expected output was "['33', '1001', '1331']".
=====assistant=====
The following is a concise explanation of the issue: The problem description states that numbers which start or end with zeros (such as `010` and `00`) are NOT considered valid numerical palindromes. However, the code above does not take this into account and therefore returns `00` as a valid palindrome. This can be fixed by checking if the first or last character is `0` before adding the string to the set of palindromes.
The code below is the correct version of the code above, where the issue has been fixed:
FIXED PYTHON CODE
```python
def is_pal(s):
    return s == s[::-1]

def palindrome(num):
    if not isinstance(num, int) or num < 0:
        return 'Not valid'
    s = str(num)
    pals = set()
    for i, ch in enumerate(s):
        if ch == '0':
            continue
        for j in range(i + 2, len(s) + 1):
            test = s[i:j]
            if is_pal(test):
                pals.add(test)
    return sorted(int(x) for x in pals) or 'No palindromes found'
```

F. Dataset: APPS Tasks Used For Our Evaluations

These tasks were randomly sampled from APPS' test set. To avoid distribution shift, we sampled according to the relative frequency of difficulties in the full dataset. We report the resulting list of tasks to aid reproducibility.

Difficulty	Tasks
introductory	'4004', '4058', '4063', '4065', '4100', '4108', '4117', '4155', '4164', '4182', '4193', '4195', '4211', '4217', '4241', '4249', '4270', '4275', '4281', '4293', '4333', '4347', '4350', '4356', '4409', '4426', '4431', '4450', '4465', '4484', '4498', '4505', '4507', '4514', '4544', '4553', '4586', '4610', '4662', '4663', '4667', '4677', '4681', '4704', '4716', '4741', '4750', '4786', '4787', '4801', '4855', '4862', '4864', '4870', '4873', '4890', '4897', '4952', '4966', '4984'
interview	'0004', '0013', '0033', '0056', '0073', '0074', '0089', '0091', '0124', '0131', '0139', '0162', '0166', '0183', '0186', '0191', '0199', '0205', '0249', '0253', '0268', '0274', '0300', '0304', '0341', '0342', '0413', '0427', '0434', '0466', '0467', '0496', '0501', '0511', '0537', '0564', '0571', '0575', '0579', '0592', '0597', '0626', '0637', '0676', '0704', '0728', '0757', '0765', '0788', '0794', '0804', '0805', '0811', '0829', '0879', '0904', '0915', '0925', '0937', '0948', '0954', '0955', '0972', '0985', '0989', '1018', '1019', '1033', '1046', '1076', '1133', '1140', '1141', '1145', '1146', '1149', '1168', '1185', '1221', '1232', '1256', '1257', '1280', '1285', '1299', '1317', '1347', '1380', '1392', '1393', '1418', '1444', '1448', '1458', '1489', '1517', '1533', '1573', '1635', '1653', '1668', '1672', '1721', '1736', '1748', '1756', '1759', '1775', '1777', '1825', '1850', '1863', '1865', '1870', '1875', '1906', '1917', '1956', '1962', '1967', '1976', '2024', '2049', '2062', '2092', '2093', '2097', '2106', '2172', '2176', '2203', '2231', '2246', '2264', '2266', '2295', '2326', '2328', '2332', '2342', '2361', '2369', '2407', '2408', '2418', '2455', '2463', '2511', '2515', '2516', '2535', '2585', '2623', '2629', '2642', '2651', '2662', '2668', '2673', '2698', '2701', '2709', '2735', '2742', '2752', '2759', '2765', '2787', '2802', '2832', '2835', '2844', '2858', '2885', '2897', '2923', '2932', '2945', '2973', '2980'
competition	'3017', '3019', '3054', '3062', '3063', '3066', '3070', '3077', '3083', '3097', '3117', '3135', '3161', '3186', '3209', '3220', '3286', '3287', '3323', '3335', '3353', '3355', '3371', '3375', '3376', '3388', '3404', '3411', '3433', '3441', '3445', '3470', '3481', '3484', '3548', '3557', '3605', '3609', '3634', '3635', '3671', '3679', '3709', '3754', '3769', '3792', '3798', '3799', '3804', '3810', '3819', '3823', '3836', '3843', '3849', '3876', '3913', '3934', '3972', '3974'

G. More Examples of Feedback from GPT-4 and our Human Participants

In this appendix, we give five more examples from the study in Section 3.3, so that the keen reader may themself compare and contrast the feedback generated by GPT-4 and by our human participants. Each listing (6-10) contains a textual specification, an incorrect program, an error message, both examples of GPT-4 feedback which were shown to the participants assigned to this task, and the responses from the participants. For each piece of feedback, we also list the number of resulting repaired programs which passed all the tests (c.f. the full results table in Appendix C).

Listing 6: Task 2106, Program A.

```
### SPECIFICATION
There are n cities in the country where the Old Peykan lives. These cities are located on a straight line, we'll denote them from left
      to right as c_1, c_2, ..., c_{n}. The Old Peykan wants to travel from city c_1 to c_{n} using roads. There are (n - 1) one way
      roads, the i-th road goes from city c_{i} to city c_{i} + 1 and is d_{i} kilometers long.

The Old Peykan travels 1 kilometer in 1 hour and consumes 1 liter of fuel during this time.

Each city c_{i} (except for the last city c_{n}) has a supply of s_{i} liters of fuel which immediately transfers to the Old Peykan if
      it passes the city or stays in it. This supply refreshes instantly k hours after it transfers. The Old Peykan can stay in a city
      for a while and fill its fuel tank many times.

Initially (at time zero) the Old Peykan is at city c_1 and s_1 liters of fuel is transferred to it's empty tank from c_1's supply. The
      Old Peykan's fuel tank capacity is unlimited. Old Peykan can not continue its travel if its tank is emptied strictly between two
      cities.

Find the minimum time the Old Peykan needs to reach city c_{n}.


-----Input-----

The first line of the input contains two space-separated integers m and k (1 <= m, k <= 1000). The value m specifies the number of
      roads between cities which is equal to n - 1.

The next line contains m space-separated integers d_1, d_2, ..., d_{m} (1 <= d_{i} <= 1000) and the following line contains m
      space-separated integers s_1, s_2, ..., s_{m} (1 <= s_{i} <= 1000).


-----Output-----

In the only line of the output print a single integer - the minimum time required for The Old Peykan to reach city c_{n} from city c_1.


-----Examples-----
Input
4 6
1 2 5 2
2 3 3 4

Output
10

Input
2 3
5 6
5 5

Output
14

### INCORRECT PROGRAM
def main():
    m, k = map(int, input().split())
    ds = list(map(int, input().split()))
    ss = list(map(int, input().split()))

    time = 0
    fuel = ss[0]
    min_time = 0
    max_s = ss[0]

    for i in range(m):
        fuel -= ds[i]
        max_s = max(max_s, ss[i])
        while fuel < 0:
            time += k
            fuel += max_s
        time += ds[i]

    print(time)


main()

### ERROR MESSAGE
===Input===
4 6
1 2 5 2
2 3 3 4
```

```
===Program Output===
28

===Expected Output===
10

### GPT-4 FEEDBACK #1
The issue with the code is that it does not consider the time spent in the current city before moving to the next city, resulting in
      an overestimation of the minimum time required. This can be fixed by updating the time spent in the current city before
      calculating the fuel used and accounting for the time it takes to travel to the next city.
# Number of passing repairs: 7/25

### GPT-4 FEEDBACK #2
The following is a concise explanation of the issue: The code incorrectly calculates the waiting time at each city. It should minimize
      the waiting time by considering the maximum fuel supply that can be collected during the waiting time. The code needs to be
      updated to account for this.
# Number of passing repairs: 10/25

### PARTICIPANT FEEDBACK #1
The program doesn't add the fuel for passing through cities, only if it needs to actually stay in the city to refuel. Whenever it
      passes through a city, it should add ss[i] to the fuel.
# Number of passing repairs: 10/25

### PARTICIPANT FEEDBACK #2
The code is incorrect because it doesn't take into account the fuel at each city that is immediately transferred after entering the
      city (aside for the fuel received at the first city). As a result, this program overestimates the time needed, because it
      assumes that the Old Peykan always has to wait for a refill (besides the fuel they are given in the first city).
# Number of passing repairs: 0/25
```

Listing 7: Task 3070, Program B.

```
### SPECIFICATION
In the free-market, ruthlessly capitalist world of train fares, only one thing matters: incentives.

Train companies are incentivised with bonuses for high throughput, successful journeys, and customer satisfaction. Conversely, the
      companies are disincentivised from failure via mandatory refunds for customers delayed by $30$ minutes or more.

Being a ruthless capitalist yourself, you have decided to take advantage of this generous delay compensation provision.

The refund is awarded provided that no matter the combination of trains you had taken (provided they followed the same route of
      stations as planned), you would still be unable to reach your destination in strictly less time than $30$ minutes (or $1800$
      seconds), of the time you would have arrived assuming your booked journey was exactly on time.

Armed with your printout of the day's delays, and the original timetable, you must ask yourself only one question: what is the
      earliest time you can book a train for from station $1$, in order to earn this restitutive reward?

-----Input-----
- One line containing two integers: $N$ ($1 \le N \le 100$), the number of stations, and $M$ ($1 \le M \le 10^5$), the number of
      scheduled trains.
- The next $M$ lines each contain 4 integers:
- $X$, the starting station ($1 \le X \le N-1$),
- $S$ and $T$ ($0 \le S \le T < 86400$), the planned departure and arrival times in seconds,
- and $L$ ($0 \le L < 86400$), the duration by which the train's departure and arrival times are delayed.

Stations are numbered from $1$ to $N$ in the order you will visit them. Each train goes between stations $X$ and $X+1$. It is possible
      to change between trains instantanesouly.

-----Output-----
- One line containing one integer: the start time of the earliest train journey you could book in order to earn your compensation, or
      impossible if no such journey is possible.

-----Examples-----
Sample Input 1:
2 3
1 1800 9000 1800
1 2000 9200 1600
1 2200 9400 1400
Sample Output 1:
1800

Sample Input 2:
2 2
1 1800 3600 1800
1 1900 3600 1600
Sample Output 2:
impossible


### INCORRECT PROGRAM
def main():
    N, M = map(int, input().split())
    train_data = [tuple(map(int, input().split())) for _ in range(M)]

    train_data.sort(key=lambda x: (x[0], x[1]))
    times = [1e9] * (N + 1)
    times[1] = 0

    for X, S, T, L in train_data:
        if times[X] <= S:
```

```
            times[X + 1] = min(times[X + 1], T + L)

    if times[N] >= 1800:
        print(int(times[1]))
    else:
        print("impossible")


main()

### ERROR MESSAGE
===Input===
2 3
1 1800 9000 1800
1 2000 9200 1600
1 2200 9400 1400
===Program Output===
0
===Expected Output===
1800

### GPT-4 FEEDBACK #1
The issue with the code is that it prints `times[1]` instead of the earliest possible train start time. The code needs to get the
      minimum start time from the train_data list that has a valid delay for compensation.
# Number of passing repairs: 3/25

### GPT-4 FEEDBACK #2
The issue with the code is that it directly prints the value of `times[1]`, which is initialized to `0` and never updated. The correct
      output should be the starting time of the earliest train journey that satisfies the condition. To fix this, we need to track the
      earliest possible start time and update it if a valid train journey is found.
# Number of passing repairs: 0/25

### PARTICIPANT FEEDBACK #1
This code is clearly wrong as times[1] is initialized to 0 then never updated and it is returned as the answer. Also, in the sorting
      line `int(x[0])` should be used in place of `x[0]` and likewise for `int(x[1])` in place of `x1` because we want numerical order
      sorting not string order sorting. The goal in this problem is calculate for each station what the earliest-arriving undelayed
      path to the station is and what the earliest-arriving delayed path to the station is, take the difference between the two, and
      see if it results in a total delay of >= 30 minutes (then to return the earliest departure from station 1 for this path. Given
      this, first the current code based around `times` can remain - this is the delayed time. However, the if-statement within the
      for-loop should be `<= S + L` since the departure time is also delayed. Then `undelayed_times` should be tracked as well,
      updated in the foor loop when  `undelayed_times[X] <= S` and set to a min() between the current undelayed time an `T` (not T+L).
      Finally the if-statement for time printing can use the difference between the delayed and undelayed times and see if any() are
      >= 1800. Finally we need to print the earliest arriving train for that path
# Number of passing repairs: 5/25

### PARTICIPANT FEEDBACK #2
One of the main issues is that the code prints 'times[1]', which is set to 0 and never updated. However, there are also some larger
      issues with the code. First, instead of checking if you reach your destination within 30 minutes of the expected time, it checks
      if you reach your destination in 30 minutes. To fix this, the program would need to keep track of the time you'd reach your
      destination if there were not delays, which it currently doesn't. Second, instead of printing times[1], it should be keeping
      track of the start time of the train that leads to a delay and printing this.
# Number of passing repairs: 0/25
```

Listing 8: Task 4333, Program A.

```
### SPECIFICATION
There is a square in the xy-plane. The coordinates of its four vertices are (x_1,y_1),(x_2,y_2),(x_3,y_3) and (x_4,y_4) in
      counter-clockwise order.
(Assume that the positive x-axis points right, and the positive y-axis points up.)
Takahashi remembers (x_1,y_1) and (x_2,y_2), but he has forgot (x_3,y_3) and (x_4,y_4).
Given x_1,x_2,y_1,y_2, restore x_3,y_3,x_4,y_4. It can be shown that x_3,y_3,x_4 and y_4 uniquely exist and have integer values.

-----Constraints-----
- |x_1|,|y_1|,|x_2|,|y_2| \leq 100
- (x_1,y_1) != (x_2,y_2)
- All values in input are integers.

-----Input-----
Input is given from Standard Input in the following format:
x_1 y_1 x_2 y_2

-----Output-----
Print x_3,y_3,x_4 and y_4 as integers, in this order.

-----Sample Input-----
0 0 0 1

-----Sample Output-----
-1 1 -1 0

(0,0),(0,1),(-1,1),(-1,0) is the four vertices of a square in counter-clockwise order.
Note that (x_3,y_3)=(1,1),(x_4,y_4)=(1,0) is not accepted, as the vertices are in clockwise order.

### INCORRECT PROGRAM
x1, y1, x2, y2 = map(int, input().split())

x3 = x2 + (y2 - y1)
y3 = y2 - (x2 - x1)
x4 = x1 + (y2 - y1)
y4 = y1 - (x2 - x1)
```

```
print(x3, y3, x4, y4)

### ERROR MESSAGE
===Input===
0 0 0 1

===Program Output===
1 1 1 0

===Expected Output===
-1 1 -1 0

### GPT-4 FEEDBACK #1
The issue with the code is that it doesn't consider the counter-clockwise order of the vertices. To fix this, we need to swap the
      signs of (y2 - y1) and (x2 - x1) in the calculations.
# Number of passing repairs: 25/25

### GPT-4 FEEDBACK #2
The following is a concise explanation of the issue: The code above does not ensure that the vertices are in counter-clockwise order.
      It calculates x3, y3, x4, and y4 in the wrong order, which results in the coordinates being in clockwise order instead. To fix
      this, we need to swap the calculations for x3 and x4, and y3 and y4.
# Number of passing repairs: 0/25

### PARTICIPANT FEEDBACK #1
The program calculates vertices for a clockwise-oriented square, not counter-clockwise. To fix this, (y2 - y1) and  (x2 - x1) should
      be negated in the equations.
# Number of passing repairs: 25/25

### PARTICIPANT FEEDBACK #2
The calculation for y3 and y4 incorrectly subtract the x direction vector to y0 and y1 respectively. In each of these cases, the
      calculation can be fixed by changing the first subtraction to addition.
# Number of passing repairs: 0/25
```

Listing 9: Task 4347, Program B.

```
### SPECIFICATION
One day, $n$ people ($n$ is an even number) met on a plaza and made two round dances, each round dance consists of exactly
      $\frac{n}{2}$ people. Your task is to find the number of ways $n$ people can make two round dances if each round dance consists
      of exactly $\frac{n}{2}$ people. Each person should belong to exactly one of these two round dances.

Round dance is a dance circle consisting of $1$ or more people. Two round dances are indistinguishable (equal) if one can be
      transformed to another by choosing the first participant. For example, round dances $[1, 3, 4, 2]$, $[4, 2, 1, 3]$ and $[2, 1,
      3, 4]$ are indistinguishable.

For example, if $n=2$ then the number of ways is $1$: one round dance consists of the first person and the second one of the second
      person.

For example, if $n=4$ then the number of ways is $3$. Possible options: one round dance - $[1,2]$, another - $[3,4]$; one round dance
      - $[2,4]$, another - $[3,1]$; one round dance - $[4,1]$, another - $[3,2]$.

Your task is to find the number of ways $n$ people can make two round dances if each round dance consists of exactly $\frac{n}{2}$
      people.


-----Input-----

The input contains one integer $n$ ($2 \le n \le 20$), $n$ is an even number.


-----Output-----

Print one integer - the number of ways to make two round dances. It is guaranteed that the answer fits in the $64$-bit integer data
      type.


-----Examples-----
Input
2

Output
1

Input
4

Output
3

Input
8

Output
1260

Input
20

Output
12164510040883200
```

```
### INCORRECT PROGRAM
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

n = int(input())

ans = (factorial(n) // (n // 2)**2) // (2 * n)
print(ans)

### ERROR MESSAGE
===Input===
2

===Program Output===
0

===Expected Output===
1

### GPT-4 FEEDBACK #1
The issue with the code is that it uses integer division when calculating the final answer, which may cause the result to be rounded
      down incorrectly. To fix this, we can use float division and then convert the result to an integer.
# Number of passing repairs: 0/25

### GPT-4 FEEDBACK #2
The following is a concise explanation of the issue: The code attempts to calculate the number of ways using integer division, which
      may result in a loss of precision and incorrect results. To fix the issue, change integer division to regular division, then
      round the result to the nearest integer using `round()`.
# Number of passing repairs: 0/25

### PARTICIPANT FEEDBACK #1
The problem is that the formula used is incorrect. For n <= 4, n! is less than n^3 / 2 and so this is why we see a 0. There is an
      erroneous additional division by n which is the true reason for the wrong answer.
# Number of passing repairs: 25/25


### PARTICIPANT FEEDBACK #2
On line 11, the program calculates a term with a correct numerator, but an incorrect denominator. It divides by (2*n), when it should
      just divide by 2.
# Number of passing repairs: 25/25
```

Listing 10: Task 4855, Program A.

```
### SPECIFICATION
The impact factor of a scientific journal is a measure reflecting the average number of citations to articles published in science
      journals. For this task we are using a simplified formula for calculating the impact factor:\[ \frac{\text {Total count of
      citations received by articles published in the journal}}{\text {Total number of articles published}} \]

Rounding is always performed upwards. For example the impact factor of the "Journal for ore research and time wasting" that published
      $38$ articles cited $894$ times is $894 / 38 = 23.53$ rounding up to $24$.

You are the editor of one scientific journal. You know how many articles you are going to publish and the owners are pushing you to
      reach a specific impact factor. You are wondering how many scientists you will have to bribe to cite your articles to meet the
      owners demands. Since money is tight you want to bribe the minimal amount of scientists. Each bribed scientist buys you a single
      citation.

-----Input-----
First and only line of input will contain $2$ integers, $A$ ($1 \leq A \leq 100$), the number of articles you plan to publish and $I$
      ($1 \leq I \leq 100$), the impact factor the owners require.

-----Output-----
The first and only line of output should contain one integer, the minimal number of scientists you need to bribe.

-----Examples-----
Sample Input 1:
38 24
Sample Output 1:
875

Sample Input 2:
1 100
Sample Output 2:
100

### INCORRECT PROGRAM
import math

A, I = map(int, input().split())
needed_citations = math.ceil(A * (I - 0.5))
scientists_to_bribe = needed_citations - A * (I - 1)
print(scientists_to_bribe)

### ERROR MESSAGE
===Input===
38 24
```

```
===Program Output===
19

===Expected Output===
875

### GPT-4 FEEDBACK #1
The following is a concise explanation of the issue: The calculation for the number of scientists to bribe is incorrect. It should be
      calculated by first finding the minimum number of citations needed: `A * I`, and then subtracting the current number of
      citations: `A * (I - 1)`.
# Number of passing repairs: 0/25

### GPT-4 FEEDBACK #2
The following is a concise explanation of the issue: The code calculates the required number of scientists to bribe by subtracting the
      current number of citations from the number of needed citations. However, it incorrectly calculates the current number of
      citations.
# Number of passing repairs: 1/25

### PARTICIPANT FEEDBACK #1
The scientists_to_bribe variable on line 3 is unnecessary., It's sufficient to just do A * (I - 1) + 1.
# Number of passing repairs: 17/25

### PARTICIPANT FEEDBACK #2
The program seems to have a conceptual misunderstanding. Specifically, the number of scientists needed to bribe is equal to the number
      of required citations (`citations_needed`). However, the calculation of `citation_needed` is also incorrect -- it implies
      rounding impact factor conventionally (e.g. 3.49 goes to 3, 3.51 goes to 4), whereas rounding here is a ceiling function on the
      impact factor. As such, the actual formula for `citation_needed` should be `A*(I-1) + 1`
# Number of passing repairs: 25/25
```