A Overview of Critic Model

Figure 1 shows an overview of our critic model. In our CodeRL framework, besides the actor LM network $\theta$, we introduce a critic model that is trained as an error predictor and parameterized as a neural network with parameters $\phi$. The critic receives as inputs a problem description $D$ and a corresponding synthetic program $W^s$ sampled from the actor. The critic is required to output a prediction of the unit test outcome of the input program. We define 4 possible outcomes $u$: CompileError, RuntimeError, FailedTest, and PassedTest. The critic model is trained by minimizing the following loss:

$$L_{critic}(\phi) = -\log p_\phi(u|W^s, D)$$ (1)

The ground-truth outcome of a synthetic sample is obtained by passing it to the unit tests corresponding to the problem. Note that since our critic model is applied in a supervised learning environment with available ground truth, we also use the training samples from the original dataset with ground truth output $u = PassedTest$ to train the critic.

The learned hidden state representations of program tokens when passed through the critic are then used to measure their return estimates for our RL optimization objective. The return estimates are incorporated as intermediate returns at decoding steps to compute the expected gradient of the actor network $\nabla_\theta L_{rl}(\theta)$.

B Critic Sampling Procedure

Refer to Algorithm 1 for a step-by-step explanation of our critic sampling procedure.

C Additional Experimental Setup Details

Pretraining Setup. For CodeT5, we adopt the code-specific tokenizer as described by [Wang et al., 2021]. Note that we employ 6 programming languages (PLs) in CodeSearchNet [Husain et al., 2019] (CSN) instead of 8 PLs in CodeT5 as C/C# datasets are not publicly available. We employ only the pretraining task of masked span prediction (MSP) in CodeT5 and hence, we do not have to parse programs into abstract syntax trees (ASTs) to obtain the identifier information. This preprocessing step was required in other original pretraining tasks like masked identifier prediction [Wang et al., 2021]. To further speed up training, we concatenate data samples to batch size 512 for pretraining with MSP and the resulting number of tokens is 1.1B.

**APPs Benchmark.** We follow the same preprocessing step as in [Hendrycks et al., 2021] to formulate the input sequences from problem descriptions. APPS consists of 10,000 coding problems with a 50-50 train-test split. Each problem is accompanied by 23.2 correct Python programs and 21.2 unit tests on average. The average length per problem is 293.2 words and the average length per program is 18.0 lines. The dataset is categorized into three levels of difficulty: Introductory (3639, train/test=2639/1000), Interview (5000, train/test=2000/3000), and Competition (1361, train/test=361/1000). Similarly to [Hendrycks et al., 2021], we employ the strict accuracy metric to evaluate the functional correctness of a program, where it is counted as correct if it can pass all the unit tests corresponding to the problem.

**MBPP Benchmark.** We additionally include another smaller and simpler Python program synthesis dataset called MBPP [Austin et al., 2021] (Mostly Basic Programming Problems) for evaluation. The dataset contains 974 instances with 374/90/500 instances for training/validation/testing respectively and 10 reserved for few-shot learning. The problems are typically short, usually one sentence of natural language descriptions each. Each problem is accompanied by 1 correct solution (6.8 lines of code on average) and 3 unit tests in the form of `assert` statements for validating the functional correctness. Unlike APPS, unit tests in MBPP are not hidden and are explicitly incorporated into the source sequences for program synthesis models. This might encourage models to be overfitting to these `assert` statements via hard-coding an if-expression very occasionally. However, for a fair comparison with the baselines, we construct the source sequences in the same way as in prior work. Specifically, we adopt the same prompt format as in [Austin et al., 2021] to prepare the input sequence as: problem descriptions + “Your code should satisfy these tests:” + 3 `assert` statements.

**Finetuning Setup.** Following [Bahdanau et al., 2016], since our RL method is applied in a supervised learning task, in addition to synthetic programs, we also use the ground-truth programs of training samples to train the critic. These samples are considered perfect programs and always have a label of PassedTest. To optimize the LM actor network, in practice, following previous work [Bahdanau et al., 2016, Rennie et al., 2017, Wang et al., 2018], in each training optimization step, we can simply approximate the expected gradient with a single sample $W_s \sim p_0$:

$$\nabla_\theta \mathcal{L}_{rl}(\theta) \approx -(r(W^a) - r(W^b)) \sum_t \hat{q}_\phi(w_t^a) \nabla_\theta \log p_0(w_t^a|w_{1:t-1}, D)$$

(2)

**Configurations.** For pretraining, we perform our experiments on a Kubernetes with 16 A100-40G GPUs on Google Cloud Platform and the total pretraining duration is around 21 days. In the first pretraining stage with MSP, we employ a corruption rate of 15%, a peak learning rate (LR) of 2e-4,
Algorithm 1: Critic sampling procedure to generate programs

**Input:** Problem \( D \), Language model \( \theta \), Critic model \( \phi \)

**Output:** A set of \( N \) generated solution programs

begin
   /* step 1: generate program solutions by LM */
   a set of output solutions \( S \leftarrow \emptyset \)
   for \( N \) times do
      /* use sampling decoding to generate programs */
      Program \( \hat{W}_i \leftarrow \text{Sampling } p_\theta(W_i|D) \) ←− Sampling \( p_\theta(w_i^t|w_i^{t-1}, D) \)
      \( S \leftarrow S \cup \{ \hat{W}_i \} \)
   /* step 2: extract example unit tests from problem */
   example unit tests \( I \leftarrow \text{ExtractExampleInputOutput}(D) \)
   /* step 3: filter for passed programs and pass to the critic model to select sub-sequences */
   a set of filtered programs \( F \leftarrow \emptyset \)
   for each generated program \( \hat{W}_i \in S \) do
      Testoutcomes \( u_i \leftarrow \text{RunTests}(\hat{W}_i, I) \)
      if \( u_i = \text{PassedTest} \) then
         \( F \leftarrow F \cup \{ \hat{W}_i \} \)
   /* If no passed samples, return all current programs for evaluation */
   if \( |F| = 0 \) then return \( S \)
   upsampling param \( N' = N/|F| \)
   a set of output solutions \( S' \leftarrow \emptyset \)
   for each filtered program \( \hat{W}_i \in F \) do
      subsequence \( W_{i\text{sub}} \leftarrow \text{Sampling } p_\phi(u_i = \text{PassedTest}|\hat{W}_i, D) \)
      ←− Sampling \( p_\phi(u_i = \text{PassedTest}|\hat{w}_{1:t-1}, D) \)
      /* step 4: subsequence as seeds to regenerate programs */
      length of subsequence \( m \leftarrow |W_{i\text{sub}}| \)
      for \( N' \) times do
         \( \hat{W}_j \leftarrow \text{Sampling } p_\theta(W_j|W_{i\text{sub}}, D) \) ←− Sampling \( p_\theta(w_j^t|w_{m:t-1}^j, W_{i\text{sub}}, D) \)
         \( S' \leftarrow S' \cup \{ \hat{W}_j \} \)
   /* return the regenerated programs for evaluation */
   return \( S' \)
end
Table 1: **CodeT5 results on CodeXGLUE**: Code-to-Text generation results (smoothed BLEU-4) of CodeT5 pretrained with larger data, improved learning objectives, and larger model size

<table>
<thead>
<tr>
<th>Model</th>
<th>Ruby</th>
<th>JavaScript</th>
<th>Go</th>
<th>Python</th>
<th>Java</th>
<th>PHP</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoBERTa</td>
<td>11.17</td>
<td>11.90</td>
<td>17.72</td>
<td>18.14</td>
<td>16.47</td>
<td>24.02</td>
<td>16.57</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>12.16</td>
<td>14.90</td>
<td>18.07</td>
<td>19.06</td>
<td>17.65</td>
<td>25.16</td>
<td>17.83</td>
</tr>
<tr>
<td>DOBF</td>
<td>-</td>
<td>-</td>
<td>18.24</td>
<td>19.05</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PLBART</td>
<td>14.11</td>
<td>15.56</td>
<td>18.91</td>
<td>19.30</td>
<td>18.45</td>
<td>23.58</td>
<td>18.32</td>
</tr>
<tr>
<td>CoTextT</td>
<td>14.02</td>
<td>14.96</td>
<td>18.86</td>
<td>19.73</td>
<td>19.06</td>
<td>24.58</td>
<td>18.55</td>
</tr>
<tr>
<td>CodeT5-base</td>
<td>15.24</td>
<td>16.16</td>
<td>19.56</td>
<td>20.01</td>
<td>20.31</td>
<td>26.03</td>
<td>19.55</td>
</tr>
<tr>
<td>CodeT5-large</td>
<td><strong>15.58</strong></td>
<td><strong>16.17</strong></td>
<td><strong>19.69</strong></td>
<td><strong>20.57</strong></td>
<td><strong>20.74</strong></td>
<td><strong>26.49</strong></td>
<td><strong>19.87</strong></td>
</tr>
</tbody>
</table>

Table 2: **CodeT5 results on CodeXGLUE**: Text-to-Code generation results of CodeT5 pretrained with larger data, improved learning objectives, and larger model size

<table>
<thead>
<tr>
<th>Model</th>
<th>EM</th>
<th>BLEU-4</th>
<th>CodeBLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-2</td>
<td>17.35</td>
<td>25.37</td>
<td>29.69</td>
</tr>
<tr>
<td>CodeGPT-2</td>
<td>18.25</td>
<td>28.69</td>
<td>32.71</td>
</tr>
<tr>
<td>CodeGPT-adapted</td>
<td>20.10</td>
<td>32.79</td>
<td>35.98</td>
</tr>
<tr>
<td>PLBART</td>
<td>18.75</td>
<td>36.69</td>
<td>38.52</td>
</tr>
<tr>
<td>CoTextT</td>
<td>20.10</td>
<td>37.40</td>
<td>40.14</td>
</tr>
<tr>
<td>UniXcoder</td>
<td>22.60</td>
<td>38.23</td>
<td>-</td>
</tr>
<tr>
<td>CodeT5-small</td>
<td>21.55</td>
<td>38.13</td>
<td>41.39</td>
</tr>
<tr>
<td>CodeT5-base</td>
<td>22.30</td>
<td>40.73</td>
<td>43.20</td>
</tr>
<tr>
<td>CodeT5-large</td>
<td><strong>22.65</strong></td>
<td><strong>42.66</strong></td>
<td><strong>45.08</strong></td>
</tr>
</tbody>
</table>

Table 3: **CodeT5 results on CodeXGLUE**: Code-to-Code generation results of CodeT5 pretrained with larger data, improved learning objectives, and larger model size

<table>
<thead>
<tr>
<th>Model</th>
<th>Java to C#</th>
<th>C# to Java</th>
<th>Refine Small</th>
<th>Refine Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLEU-4</td>
<td>EM</td>
<td>BLEU-4</td>
<td>EM</td>
</tr>
<tr>
<td>Naive copy</td>
<td>18.54</td>
<td>0.00</td>
<td>18.69</td>
<td>0.00</td>
</tr>
<tr>
<td>Roberta (code)</td>
<td>77.46</td>
<td>56.10</td>
<td>71.99</td>
<td>57.90</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>79.92</td>
<td>59.00</td>
<td>72.14</td>
<td>58.00</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>80.58</td>
<td>59.40</td>
<td>72.64</td>
<td>58.80</td>
</tr>
<tr>
<td>PLBART</td>
<td>83.02</td>
<td>64.60</td>
<td>78.35</td>
<td>65.00</td>
</tr>
<tr>
<td>CoTextT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NSEdit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CodeT5-small</td>
<td>82.98</td>
<td>64.10</td>
<td>79.10</td>
<td>65.60</td>
</tr>
<tr>
<td>CodeT5-base</td>
<td><strong>84.03</strong></td>
<td><strong>65.90</strong></td>
<td><strong>79.87</strong></td>
<td><strong>66.90</strong></td>
</tr>
<tr>
<td>CodeT5-large</td>
<td>83.56</td>
<td><strong>66.00</strong></td>
<td><strong>79.77</strong></td>
<td><strong>67.00</strong></td>
</tr>
</tbody>
</table>

the actor language model for 10 more epochs using both the conventional NTP loss and RL-based loss.

For MBPP, in finetuning experiments with CodeT5, due to its small training set, we finetune the models for 60 epochs with a constant LR of 2e-5 and a batch size of 32, which takes less than 30 mins on one A100. We set its maximum source and target length to 382 and 306 respectively.

**Critic Sampling.** Note that in CS, many rounds of regeneration can be run to keep improving the output programs. However, in practice, we found that one round of regeneration is sufficient to obtain good programs. In the initial generation, if no samples pass example unit tests, we do not apply critic sampling and simply use all available programs for evaluation.
Table 4: Ablation results on MBPP: Ablation results of CodeRL with different CodeT5 model variants with different sizes, pretraining data and objectives on MBPP. CodeT5† is finetuned on APPS and evaluated on MBPP in a zero-shot setting.

<table>
<thead>
<tr>
<th>Model Size</th>
<th>Data</th>
<th>Objective</th>
<th>pass@80</th>
<th>pass@1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT 224M</td>
<td>Web Doc</td>
<td>LM</td>
<td>7.2</td>
<td>-</td>
</tr>
<tr>
<td>GPT 422M</td>
<td>Web Doc</td>
<td>LM</td>
<td>12.6</td>
<td>-</td>
</tr>
<tr>
<td>GPT 1B</td>
<td>Web Doc</td>
<td>LM</td>
<td>22.4</td>
<td>-</td>
</tr>
<tr>
<td>GPT 4B</td>
<td>Web Doc</td>
<td>LM</td>
<td>33.0</td>
<td>-</td>
</tr>
<tr>
<td>GPT 8B</td>
<td>Web Doc</td>
<td>LM</td>
<td>40.6</td>
<td>-</td>
</tr>
<tr>
<td>GPT 68B</td>
<td>Web Doc</td>
<td>LM</td>
<td>53.6</td>
<td>-</td>
</tr>
<tr>
<td>GPT 137B</td>
<td>Web Doc</td>
<td>LM</td>
<td>61.4</td>
<td>-</td>
</tr>
<tr>
<td>CodeT5 60M</td>
<td>CSN</td>
<td>MSP</td>
<td>19.2</td>
<td>36.2</td>
</tr>
<tr>
<td>CodeT5 220M</td>
<td>CSN</td>
<td>MSP</td>
<td>24.0</td>
<td>42.8</td>
</tr>
<tr>
<td>CodeT5 770M</td>
<td>CSN</td>
<td>MSP</td>
<td>32.4</td>
<td>47.8</td>
</tr>
<tr>
<td>CodeT5 770M+GCPY</td>
<td>MSP</td>
<td>34.6</td>
<td>51.6</td>
<td></td>
</tr>
<tr>
<td>CodeT5 770M+GCPY+NTP</td>
<td>MSP</td>
<td>46.8</td>
<td>66.2</td>
<td></td>
</tr>
<tr>
<td>CodeT5† 770M+GCPY+NTP</td>
<td>MSP</td>
<td>60.2</td>
<td>78.4</td>
<td></td>
</tr>
<tr>
<td>CodeRL 770M+GCPY+NTP</td>
<td>MSP</td>
<td>63.0</td>
<td>81.8</td>
<td></td>
</tr>
</tbody>
</table>

D Additional Experimental Results

D.1 CodeXGLUE Benchmark Results

To validate the effectiveness of our simplified pretraining strategies of CodeT5-large, we extensively evaluate it on a variety of generation tasks in CodeXGLUE [Lu et al., 2021], including code-to-text generation (i.e. summarization, see Table 1), text-to-code generation (see Table 2), and code-to-code generation (i.e., code translation and code refinement, see Table 3). Different from APPS [Hendrycks et al., 2021] and MBPP [Austin et al., 2021], we follow the default similarity-based evaluation metrics in the CodeXGLUE benchmark, including BLEU [Papineni et al., 2002] and CodeBLEU [Ren et al., 2020], and exact match (EM) scores. Table 1, 2, and 3 show that our simplified pretrained CodeT5-large sets new SOTA results on a large majority of the tasks, and hence, can be served as a better foundation model for other code-related generation tasks. Note that in these experiments, we employ the conventional finetuning objective with $L_{ce}$ and there might be potential to improve the performance further with our CodeRL framework.

D.2 MBPP Benchmark Results

Following [Austin et al., 2021], we adopt temperature sampling to generate multiple candidate solutions. We empirically find that CodeT5 benefits from a higher temperature of 1.2 (less greedy decoding or more diverse) than their GPT’s temperature of 0.5 on this benchmark.

Table 4 reports the pass@80 and pass@1000 results for both finetuned and zero-shot settings. For baselines, we compared with GPT models with sizes ranging from 224M to 137B [Austin et al., 2021], which are pretrained on 2.93B web documents (13.8M containing source code) using standard language modeling objective. Results of GPT models are obtained from the original authors. From the comparison among various CodeT5 variants, we again confirm that larger model sizes and pretraining data, and better pretraining objective of NTP all lead to a performance boost. Particularly, our CodeT5-770M yields a pass@80 of 46.8%, surpassing GPT-8B’s 40.6% with a much smaller model size. In addition, we find CodeT5 models finetuned on APPS can achieve a surprisingly good zero-shot performance on MBPP with a pass@80 of 60.2% and further improved to 63.0% with the help of CodeRL, which even outperforms the largest GPT-137B’s performance of 61.4%. This indicates APPS is a comprehensive program synthesis benchmark and CodeT5+CodeRL models trained on it are able to generalize to other simpler coding tasks. If we further increase the budget of
Analysis of duplicated lines between MBPP and APPS: The overlap of data between APPS and MBPP appears to be minimal, with only 12.6% MBPP programs with > 50% lines duplicated in APPS training data.

Attempts up to 1000, all models witness a consistent and significant boost in solving rate, especially our CodeT5+CodeRL yielding a new SOTA result of 81.8% pass@1000.

A common concern about transfer learning is that the source (APPS) and target (MBPP) tasks might overlap in their training data, which could result in the source model tending to memorize these substantially similar data when applied to the target task. To address this concern, we analyze how many lines of code appear in both the training set of APPS and the programs of MBPP following [Austin et al. 2021]. For this analysis, we discard code comments and normalize the whitespaces for each line, and then exclude lines that appear more than twice anywhere in MBPP, as these are likely to be common Python keywords such as return and break.

Figure 2 illustrates the number of absolute duplicated lines (Left) and the relative fraction of duplicated lines (Right) in the MBPP programs. As can be seen, the overlap between APPS and MBPP seems to be minimal. Only 12.6% MBPP programs have more than half of their lines matched somewhere in the APPS training data. Besides, more than half (514 out of 974) of programs have a zero overlap and 90.9% have only no more than 3 lines overlapping with the APPS training set. If we further require the lines to be consecutive, there are no more than 2 consecutive duplicated lines.

D.3 APPS Benchmark Results on Competition-level Tasks

Figure 3 shows the results of pass@k and n@k with k ranging from 1 to 200 and n = {1, 5}, for CodeRL+CodeT5 and CodeT5 only. We choose to investigate a subset of the APPS test split, which contains the test samples of the highest difficulty level (i.e. competition programming tasks). Since CodeRL is model-agnostic, we also integrate it with GPT-J [Wang and Komatsuzaki 2021] and report the results. To focus on the impact of our RL optimization, during test time, we compare models using only nucleus sampling and without the CS procedure. Figure 3 shows that the performance gains are quite consistent on both GPT-J and CodeT5. In particular, as k increases, the performance gain of CodeRL is more significant on both GPT-J and CodeT5 models. We attribute these gains to the CodeRL learning objective Lrl that encourages models to explore code solutions drawn from the model’s sampling distribution. During test time with an increasing k sampling budget, models are allowed to generate diverse code solutions and the impact of Lrl becomes more significant.

D.4 CodeT5 Ablation by Training Epochs

Figure 4 shows the performance of CodeT5 model variants by finetuning epochs and by difficulty levels of programming tasks. Note that in these experiments, we only need to compare among CodeT5 model variants by pretraining strategies, and hence, only engage Lcc (imitation learning) in
Figure 3: Results on APPS competition-level test samples: We investigate the most challenging programming problem tasks, i.e. competition level, in the APPS benchmark. Integrating CodeRL with both CodeT5 and GPT-J, we observe good performance improvement across $\text{pass@k}$ and $\text{n@k}$ metrics, with increasing performance gains as $k$ increases.

Figure 4: Ablation results by finetuning epochs: We report the finetuning progress of CodeT5-770M models that are pretrained on different configurations by pretraining data and pretraining tasks. CSN: CodeSearchNet, GCPY: Github Code Python, MSP: Masked Span Prediction, NTP: Next Token Prediction. All models are finetuned only with $\mathcal{L}_{\text{ce}}$ on APPS. Enhancing both pretraining data (with larger data of GCPY) and pretraining objectives (with NTP objective) improves model performance across training epochs in general. Moreover, as noted by our analysis of learning objectives, only using $\mathcal{L}_{\text{ce}}$ often leads to overfitting performance, typically after epoch 10 in our case. Hence, to further finetune large-scale LMs, we recommend adopting our RL objective $\mathcal{L}_{\text{rl}}$ to utilize synthetic training samples and avoid overfitting models.

D.5 Impacts of CodeRL on Program Quality by Unit Test Signals

Figure 5 demonstrates the average percentages of generated programs by their test signals. Specifically, we use CodeT5 or CodeRL+CodeT5 to generate programs and randomly select 200 generated programs per test sample in the APPS test split. We pass programs to either example unit tests or hidden unit tests corresponding to the problem and group the output programs by their output signals, including CompileError, RuntimeError, FailedTest, and PassedTest. We observe that integrating CodeRL can increase the likelihood that a program can pass unit tests, and reduces the likelihood that it fails one or more unit tests, or whether it contains compiling or runtime errors. However, we note that there are significant gaps in performance by test signals between example unit tests and hidden unit tests. This observation suggests that example tests are not as comprehensive as hidden
tests and hence, applying our CS procedure might lead to false positive samples for regeneration. We recommend additional methods to improve example unit tests, such as through data augmentation by mutating input/output pairs [Austin et al., 2021].

D.6 Example Generated Programs and Qualitative Analysis

Figure 6 to 8 show examples of programming problems from the APPS benchmark and corresponding programs generated by CodeT5 variants. Specifically, based on the same foundation pretrained CodeT5 (pretrained with GCPY data and NTP objective), we compare the CodeT5 model that is finetuned by $L_{ce}$ only and another model that follows our CodeRL framework. In CodeRL+CodeT5, we further show programs before and after applying the CS procedure. The example programs show that applying CodeRL can improve the quality of generated programs and the CS procedure further improves the functional correctness of the programs. For instance, in Figure 8, CodeT5 model misunderstands the problem and focuses on finding the greatest common divisor between $a$ and $b$ only. Instead, the CodeRL model avoids this mistake and tackles the problem based on the factorials of $a$ and $b$. In Figure 7, we note that the CS procedure improves the program by reordering the `elif` code blocks. The resulting program is more correct and is able to pass all hidden unit tests.

We also found that CodeRL can improve the efficiency of the programs, an important quality in complex programming problems. For instance, in the interview-level programs in Figure 8, we note that without applying CS, the generated program is functionally correct but fails during execution due to a timeout error. Applying the CS procedure can condition models on parts of the prior program and (re)generates new tokens to produce a more efficient program. Hence, the resulting final program is able to pass all hidden unit tests (including tests with extremely large values) without timeout errors.

References


**QUESTION:**
There is a right triangle ABC with $\angle ABC=90^\circ$.
Given the lengths of the three sides, $|AB|, |BC|$ and $|CA|$, find the area of the right triangle ABC.
It is guaranteed that the area of the triangle ABC is an integer.

--- Constraints ---
- $1 \leq |AB|, |BC|, |CA| \leq 100$
- All values in input are integers.
- The area of the triangle ABC is an integer.

--- Input ---
Input is given from Standard Input in the following format:
$|AB| \ |BC| \ |CA|$

--- Output ---
Print the area of the triangle ABC.

--- Sample Input ---
$3 \ 4 \ 5$

--- Sample Output ---
$6$

This triangle has an area of $6$. Use Standard Input format

**ANSWER:**

--- Example Unit Tests ---
<table>
<thead>
<tr>
<th>Input:</th>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4 5</td>
<td>6</td>
</tr>
</tbody>
</table>

--- Hidden Unit Tests ---
<table>
<thead>
<tr>
<th>Input:</th>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 12 13</td>
<td>30</td>
</tr>
<tr>
<td>24 7 25</td>
<td>84</td>
</tr>
</tbody>
</table>

--- Ground-truth Program ---
1. $L = \text{list(map(int, input().split()))}$
2. $\text{sortL = sorted(L)}$
3. $\text{ans = L[0] \times L[1] / 2}$
4. $\text{print(int(ans))}$

--- CodeT5 Output Program ---
1. $ab, bc, ca = \text{map(int, input().split())}$
2. $\text{print(ab \times bc \times ca)}$

--- Example tests: Failed
Hidden tests: Failed

--- CodeRL+CodeT5 Output Program (before CS) ---
1. $a, b, c = \text{map(int, input().split())}$
2. $x = (a+b+c)/2$
3. $\text{print(x)}$

--- Example tests: Passed
Hidden tests: Failed

--- CodeRL+CodeT5 Output Program (after CS) ---
1. $a, b, c = \text{map(int, input().split())}$
2. $x = (a+b+c)/2$
3. $y = (a+b-c)/2$
4. $\text{print((xx+y))}$

--- Example tests: Passed
Hidden tests: Passed

**Figure 6:** An example synthesis task from the APPS benchmark and corresponding programs generated by CodeT5 variants: Without the CS generation procedure, CodeRL+CodeT5 model can generate programs that pass all example tests but fail hidden tests. With the CS generation procedure, the model can condition on prior programs and generate a better program that passes all hidden tests.
Input Problem

QUESTION:
Allen has a LOT of money. He has $n$ dollars in the bank. For security reasons, he wants to withdraw it in cash (we will not disclose the reasons here). The denominations for dollar bills are $1$, $5$, $10$, $20$, $100$. What is the minimum number of bills Allen could receive after withdrawing his entire balance?

-----Input-----
The first and only line of input contains a single integer $n$ ($1 \leq n \leq 10^9$).

-----Output-----
Output the minimum number of bills that Allen could receive.

-----Examples-----
Input
125
Output
3

Input
43
Output
5

-----Input-----

Ground-truth Program

1  n = int(input())
2  ans = 0
3  ans += n//100
4  n %= 100
5  ans += n // 20
6  n %= 20
7  ans += n // 10
8  n %= 10
9  ans += n//5
10  n %= 5
11  ans += n//1
12  print(ans)

-----Examples-----

Example tests: Passed
Hidden tests: Passed

CodeRL+CodeT5 Output Program

(before CS)

1  def main():
2      n = int(input())
3      cnt = 0
4      while n > 0:
5          if n % 100 == 0:
6              n //= 100
7              cnt += 1
8          elif n % 20 == 0:
9              n //= 20
10             cnt += 1
11          elif n % 5 == 0:
12              n //= 5
13              cnt += 1
14          elif n % 10 == 0:
15              n //= 10
16              cnt += 1
17          else:
18              n = n - 1
19              cnt += 1
20      print(cnt)
21 22 23  _starting_point():
24      main()
25 26  _starting_point()

Example tests: Passed
Hidden tests: Failed

CodeT5 Output Program

1  n = int(input())
2  ans = 0
3  ans += n//100
4  n %= 100
5  ans += n // 20
6  n %= 20
7  ans += n // 10
8  n %= 10
9  ans += n//5
10  n %= 5
11  ans += n//1
12  print(ans)

-----Examples-----

Example tests: Failed
Hidden tests: Passed

-----Note-----

In the first sample case, Allen can withdraw this with a $100$ dollar bill, a $20$ dollar bill, and a $5$ dollar bill. There is no way for Allen to receive $125$ dollars in one or two bills.

In the second sample case, Allen can withdraw two $20$ dollar bills and three $5$ dollar bills.

In the third sample case, Allen can withdraw $100000000$ ($10$ million) $100$ dollar bills. Use Standard Input format

ANSWER:

Figure 7: An example synthesis task from the APPS benchmark and corresponding programs generated by CodeT5 variants: Without the CS generation procedure, CodeRL+CodeT5 model can generate programs that pass all example tests but fail hidden tests, especially those of corner cases. With the CS generation procedure, the model can condition on prior programs and refine the code. Specifically, we observe the model can simply reorder the `elif` blocks between line 11 and 15 to fix the error. The resulting program is functionally correct and passes all hidden tests.


**QUESTION:**
Holidays have finished. Thanks to the help of the hacker Leha, Noora managed to enter the university of her dreams which is located in a town Pavlopolis. It's well known that universities provide students with dormitory for the period of university studies. Consequently Noora had to leave Vičkopolis and move to Pavlopolis. Thus Leha was left completely alone in a quiet town Vičkopolis. He almost even fell into a depression from boredom!

Leha came up with a task for himself to relax a little. He chooses two integers A and B and then calculates the greatest common divisor of integers “A factorial” and “B factorial”. Formally the hacker wants to find out GCD(A!, B!). It's well known that the factorial of an integer x is a product of all positive integers less than or equal to x. Thus x! = 1·2·3·…·(x - 1)·x. For example 4! = 1·2·3·4 = 24. Recall that GCD(x, y) is the largest positive integer q that divides (without a remainder) both x and y.

Leha has learned how to solve this task very effective. You are able to cope with it not worse, aren't you?

---Input---
The first and single line contains two integers A and B (1 ≤ A, B ≤ 10^9, min(A, B) ≤ 12).

---Output---
Print a single integer denoting the greatest common divisor of integers A! and B!.

---Example---
Input
4 3
Output
6

---Note---
Consider the sample.
4! = 1·2·3·4 = 24, 3! = 1·2·3 = 6. The greatest common divisor of integers 24 and 6 is exactly 6.

Use Standard Input format

**ANSWER:**

<table>
<thead>
<tr>
<th>Input Problem</th>
<th>Example Unit Tests</th>
<th>Hidden Unit Tests</th>
</tr>
</thead>
</table>
| **QUESTION:** Holidays have finished. Thanks to the help of the hacker Leha, Noora managed to enter the university of her dreams which is located in a town Pavlopolis. It's well known that universities provide students with dormitory for the period of university studies. Consequently Noora had to leave Vičkopolis and move to Pavlopolis. Thus Leha was left completely alone in a quiet town Vičkopolis. He almost even fell into a depression from boredom! Leha came up with a task for himself to relax a little. He chooses two integers A and B and then calculates the greatest common divisor of integers “A factorial” and “B factorial”. Formally the hacker wants to find out GCD(A!, B!). It's well known that the factorial of an integer x is a product of all positive integers less than or equal to x. Thus x! = 1·2·3·…·(x - 1)·x. For example 4! = 1·2·3·4 = 24. Recall that GCD(x, y) is the largest positive integer q that divides (without a remainder) both x and y. Leha has learned how to solve this task very effective. You are able to cope with it not worse, aren't you? | **Input:**
4 3
**Output:**
6 | **Input:**
10 399603090
**Output:**
3628800
**Input:**
5 5
**Output:**
120

---Ground-truth Program---

```python
1 a, b = map(int, input().split())
2 a = min(a, b)
3 ans = 1
4 for i in range(1, a + 1):
5 ans *= i
6 print(ans)
```

---CodeT5 Output Program---

```python
1 from math import gcd
2 a, b = map(int, input().split())
3 c = math.gcd(a, b)
4 print(c)
```

---Example tests: Failed
Hidden tests: Failed

---CodeRL+CodeT5 Output Program (before CS)---

```python
1 from math import gcd
2 a, b = list(map(int, input().split()))
3 gcd = lambda x, y: x if y == 0 else gcd(y, x % y)
4 5 a_prod = 1
6 b_prod = 1
7 for i in range(1, a + 1):
8 a_prod *= i
9 for i in range(1, b + 1):
10 b_prod *= i
11 12 print(gcd(a_prod, b_prod))
```

---Example tests: Passed
Hidden tests: Runtime Error (TimeOut)

---CodeRL+CodeT5 Output Program (after CS)---

```python
1 from math import gcd
2 a, b = list(map(int, input().split()))
3 gcd = lambda x, y: x if y == 0 else gcd(y, x % y)
4 5 a_prod = 1
6 b_prod = 1
7 for i in range(1, min(a, b) + 1):
8 a_prod *= i
9 b_prod *= i
10 11 print(gcd(a_prod, b_prod))
```

---Example tests: Passed
Hidden tests: Passed

Figure 8: An example interview-level APPS programming task and programs generated by CodeT5 variants: The program generated by CodeT5 model fails all unit tests while CodeRL+CodeT5 (without CS generation) can generate a functionally correct program. However, this program leads to runtime errors in extreme test cases. After applying the CS generation procedure, the program is improved and able to pass all hidden unit tests.

Association for Computational Linguistics, 2021.