A Supplementary Material

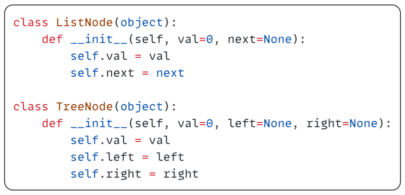
A.1 Dataset Nutrition Labels

Field Name	Definition
id	Task ID.
slug_name	Task name.
meta_info	The field accommodating the task description and submission statistics.
difficulty	The difficulty level of the task.
pretty_content	The field introduces the task description, examples, and constraints in pure text.
solutions	Samples of solutions extracted from actual past submissions.
prompt	The prompt of the solution.
entry_point	The nominative entry point of the solution.
generator_code	A function to generate test cases.
test_cases	A collection of generated test cases.
convert_online	A function to format test cases for online evaluation.
convert_offline	A function to format test cases for offline evaluation.
evaluate_offline	A function designed to evaluate solutions in an offline setting.

Table 5: Definitions of the fields within the Mercury dataset.

A.2 Mercury Data Distribution and Customized Data Structures

Except for all built-in Python data structures, Mercury imports another two structures to enhance the diversity and complexity as shown in Figure 4.



```
class ListNode(object):
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

Figure 4: Mercury supports two customized data structures: TreeNode and ListNode.

Splits	Easy	Medium	Hard	Sum
Mercury-train	446	968	219	1,633
Mercury-eval	88	81	87	256

Table 6: *Mercury-eval* encompasses 256 tasks, the difficulty of which has been balanced for model evaluation. *Mercury-train* comprises the remaining 1,633 tasks for model training.

A.3 Sandbox Details

Time and Memory Limitation. Each executed code within the sandbox is subject to certain constraints to ensure fair utilization of resources and to prevent any single code from monopolizing the system resource. Specifically, there are two primary constraints: a time limit and a memory limit. The time limit restricts how long the code can execute before being forcibly terminated, thereby ensuring that no infinite loops or excessively long computations negatively impact the availability of the sandbox. The memory limit caps the amount of RAM that a process can consume. This measure precludes a single code from exhausting the memory resources, which could lead to a denial of service for subsequent codes. In our experiment settings, the timeout limit is 30 seconds, and the memory limit is 2048 MB for each solution execution.

IO Restriction. To mitigate harmful activities such as unauthorized command execution or data exfiltration, the sandbox imposes strict Input/Output (IO) restrictions. These restrictions include limitations on reading from or writing to the disk and restrictions on the use of network sockets for sending or receiving data. By controlling the IO operations, the sandbox can prevent many common vulnerabilities and ensure that the code runs without interfering with other processes of the host system.

Isolated File System. The sandbox employs an isolated file system to provide a safe execution environment for the code. This means that the process running in the sandbox has its virtual file system, which is separated from the host's file system. The isolated nature of this file system ensures that even if a process within the sandbox attempts to modify or delete files, these changes will not affect the host system or other sandboxes. It acts as a security layer, protecting the host from potential threats and maintaining the integrity of the overall system.

System Libraries Redirection. To maintain a consistent and controlled environment, the sandbox redirects calls to system libraries to sandbox-specific versions. This is done to prevent code from using certain functions directly from the host's system libraries, which could result in unpredictable behavior or security vulnerabilities. The redirected libraries are often limited to a subset of functionalities deemed safe and necessary for executing programs within the sandbox, thus enforcing the security policies and ensuring that the running programs behave as expected.

Single-threaded Evaluation. Single-threaded evaluation refers to executing code using a sole thread of execution, thereby simplifying resource management and timing assessments, and mitigating the intricacies linked with multi-threaded execution, such as synchronization issues, race conditions, and potential deadlocks. This mode of operation is especially important in testing environments where reproducibility and fairness are paramount, ensuring that each piece of code is evaluated using identical computational resources.

Code Efficiency Measurement. Figure 5 shows the overview of the code execution pipeline. We gauge the *Solution Instantiation* and *Test Ease Evaluation* time spans as the execution runtime.

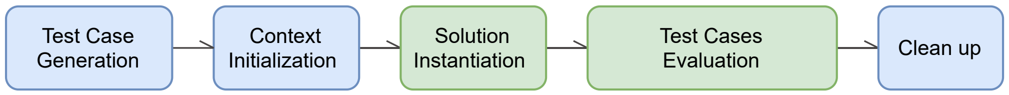


Figure 5: Sandbox Execution Pipeline. **1) Test Case Generation.** We first employ the corresponding test case generator for each task to produce a comprehensive set of test cases for the subsequent evaluation. **2) Context Initialization.** To prevent any unexpected code behavior, the sandbox environment is meticulously reinitialized for each new task. This phase ensures that all the common libraries required for executing the solution are loaded. **3) Solution Instantiation.** The solution under evaluation will be encapsulated as a *solution* class. **4) Test Case Evaluation.** Each test case the generator provides will be rigorously executed against the solution. A solution must successfully pass all the test cases to be deemed valid. **5) Clean up.** The final stage involves the sandbox dutifully clearing the namespace environment and the temporary directory. Mercury records the time consumed during the stage of Solution instantiation and Test Ease Evaluation as the primary metric for assessing code efficiency.

A.4 DPO Experiment Details

Dataset Construction. For every task problem T^i in Mercury, we randomly selected two solutions from the task solution set $\{s_w^i, s_l^i\} \sim T_{solution}^i$, to construct the preference dataset $D = \{P^i, s_w^i, s_l^i\}$, where p^i is the prompt, s_w^i has a faster runtime than s_l^i.

Model Initialization. RLHF [48] typically begins with a reference LLM π_{ref}. Here, we initialize π_{ref} by maximizing the likelihood of faster code completions $(p, s_w) \sim D$, so that $\pi_{ref} = \arg\max_\pi E_{(p,s_w)\sim D}[\log \pi(s_w|p)]$. This procedure helps mitigate the distribution shift between the *true reference distribution* and π_{ref}.

Optimization. We optimize the target LLM π_θ to minimize $\mathcal{L}_{DPO}$ for the given π_{ref} and D and desired hyperparameter β. The gradient with respect to the parameters θ can be written as $\nabla_\theta \mathcal{L}_{DPO}(\pi_\theta; \pi_{ref})$.

$$\mathcal{L}_{DPO}(\pi_\theta; \pi_{ref}) = -E_{(x,s_w,s_l)\sim D}\left[\log \alpha(\beta \log \frac{\pi_\theta(s_w|p)}{\pi_{ref}(s_w|p)}) - \log \frac{\pi_\theta(s_l|p)}{\pi_{ref}(s_l|p)})\right] \quad (2)$$

$$\nabla_\theta \mathcal{L}_{DPO}(\pi_\theta; \pi_{ref}) =$$

$$-\beta E_{(p,s_w,s_l)\sim D}\left[\underbrace{\alpha(\hat{r}_\theta(p,s_l) - \hat{r}_\theta(p,s_w))}_{\text{higher weight for wrong estimate}}\left[\underbrace{\nabla_\theta \log \pi(s_w|p)}_{\text{increase likelihood of } s_w} - \underbrace{\nabla_\theta \log \pi(s_l|p)}_{\text{decrease likelihood of } s_l}\right]\right] \quad (3)$$

Intuitively, the gradient of the loss function $\mathcal{L}_{DPO}$ increases the likelihood of the preferred completions s_w and decreases the likelihood of dis-preferred completions s_l, which are weighed by how much higher the implicit reward model $\hat{r}_\theta$ rates the dis-preferred completions, scaled by β, *i.e.*, how incorrectly the implicit reward model orders the completions, accounting for the strength of the KL constraint.

A.5 External Libraries Utilized in Mercury

Raw LeetCode solutions typically commence without importing shared libraries. To avoid solution failure due to absent libraries, we proactively import the libraries listed in Figure 6 during the sandbox *Context Initialization* phase. Note that all these libraries are imported in a temporary namespace of which the sandbox controls code behaviors.

```
exec('import re', namespace);
exec('import itertools', namespace);
exec('import collections', namespace);
exec('import heapq', namespace);
exec('import bisect', namespace);
exec('import string', namespace);
exec('import sys', namespace);
exec('import lctk', namespace);
exec('import functools', namespace);
exec('import math', namespace);
exec('import copy', namespace);
exec('import heapq', namespace);
exec('import sortedcontainers', namespace);

exec('from math import floor, ceil, factorial, sqrt, inf', namespace);
exec('from sys import maxsize, stdin', namespace);
exec('from bisect import bisect_left, bisect_right', namespace);
exec('from itertools import permutations, zip_longest', namespace);
exec('from heapq import heappush, heappop, heapify', namespace);
exec('from collections import deque, defaultdict, OrderedDict', namespace);
exec('from typing import List, Optional, Tuple', namespace);
exec('from functools import lru_cache, cache', namespace);
```

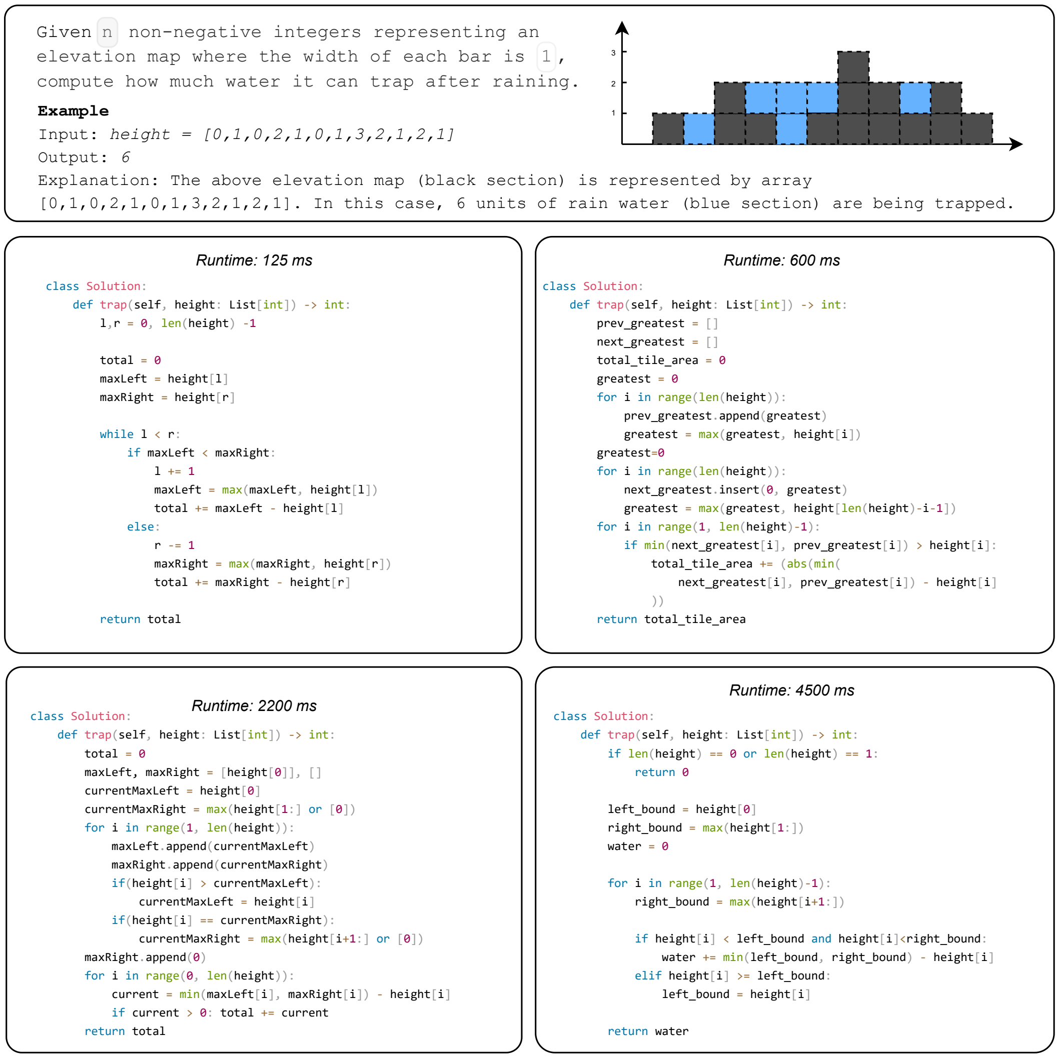
Figure 6: External Libraries Imported in Mercury Evaluate Framework.

A.6 Model Details

Model Name	Model Scale	Link
deepseek-coder-1.3b-base	1.3B	https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-base
starcoder2-3b	3B	https://huggingface.co/bigcode/starcoder2-3b
deepseek-coder-6.7b-base	6.7B	https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base
starcoder2-7b	7B	https://huggingface.co/bigcode/starcoder2-7b
CodeLlama-7b-hf	7B	https://huggingface.co/codellama/CodeLlama-7b-hf
CodeQwen1.5-7B	7B	https://huggingface.co/Qwen/CodeQwen1.5-7B
CodeLlama-13b-hf	13B	https://huggingface.co/codellama/CodeLlama-13b-hf
starcoder2-15b	15B	https://huggingface.co/bigcode/starcoder2-15b
deepseek-coder-33b-base	33B	https://huggingface.co/deepseek-ai/deepseek-coder-33b-base
CodeLlama-34b-hf	34B	https://huggingface.co/codellama/CodeLlama-34b-hf

Table 7: Model Details. We evaluated LLMs ranging from 1.3B to 34B.

A.7 A Mercury Example



Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example

Input: *height = [0,1,0,2,1,0,1,3,2,1,2,1]*

Output: *6*

Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Runtime: 125 ms

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        l,r = 0, len(height) -1

        total = 0
        maxLeft = height[l]
        maxRight = height[r]

        while l < r:
            if maxLeft < maxRight:
                l += 1
                maxLeft = max(maxLeft, height[l])
                total += maxLeft - height[l]
            else:
                r -= 1
                maxRight = max(maxRight, height[r])
                total += maxRight - height[r]

        return total
```

Runtime: 600 ms

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        prev_greatest = []
        next_greatest = []
        total_tile_area = 0
        greatest = 0
        for i in range(len(height)):
            prev_greatest.append(greatest)
            greatest = max(greatest, height[i])
        greatest=0
        for i in range(len(height)):
            next_greatest.insert(0, greatest)
            greatest = max(greatest, height[len(height)-i-1])
        for i in range(1, len(height)-1):
            if min(next_greatest[i], prev_greatest[i]) > height[i]:
                total_tile_area += (abs(min(
                    next_greatest[i], prev_greatest[i]) - height[i]
                ))
        return total_tile_area
```

Runtime: 2200 ms

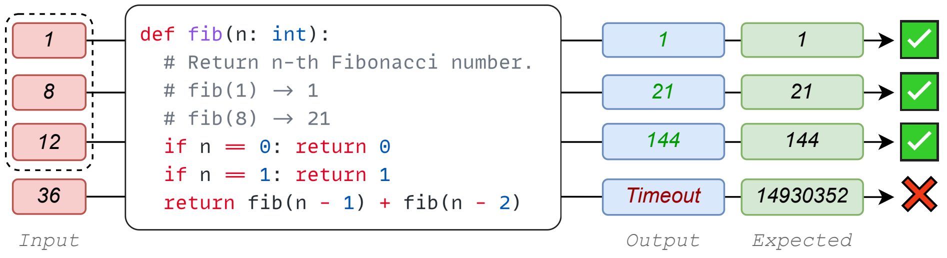
```python
class Solution:
    def trap(self, height: List[int]) -> int:
        total = 0
        maxLeft, maxRight = [height[0]], []
        currentMaxLeft = height[0]
        currentMaxRight = max(height[1:] or [0])
        for i in range(1, len(height)):
            maxLeft.append(currentMaxLeft)
            maxRight.append(currentMaxRight)
            if(height[i] > currentMaxLeft):
                currentMaxLeft = height[i]
            if(height[i] == currentMaxRight):
                currentMaxRight = max(height[i+1:] or [0])
        maxRight.append(0)
        for i in range(0, len(height)):
            current = min(maxLeft[i], maxRight[i]) - height[i]
            if current > 0: total += current
        return total
```

Runtime: 4500 ms

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        if len(height) == 0 or len(height) == 1:
            return 0

        left_bound = height[0]
        right_bound = max(height[1:])
        water = 0

        for i in range(1, len(height)-1):
            right_bound = max(height[i+1:])

            if height[i] < left_bound and height[i]<right_bound:
                water += min(left_bound, right_bound) - height[i]
            elif height[i] >= left_bound:
                left_bound = height[i]

        return water
```

Figure 7: This case is drawn from the *Mercury-eval* benchmark. The upper block presents the problem statement with its example, while the subsequent portion exhibits the corresponding solutions. Although all solutions are functionally correct, they exhibit significant differences in runtimes.

A.8 A HumanEval Example



```python
def fib(n: int):
  # Return n-th Fibonacci number.
  # fib(1) → 1
  # fib(8) → 21
  if n == 0: return 0
  if n == 1: return 1
  return fib(n - 1) + fib(n - 2)
```

Figure 8: An HumanEval example of insufficient test cases. Even though the code passed all test cases in the dashed-line box, it remains vulnerable to timeout or stack overflow when subjected to a larger input.

A.9 Prompts for Code Generation

To guarantee a fair comparison, we apply a unified one-shot prompt template for each pre-trained Code LLM. As displayed in Figure 9, the prompt template contains one shot example as well as three placeholders: <*task_content*>, <*code_starter*>, and <*code_completion*>.

```
 1  ### Question
 2  You are given a 0-indexed array of positive integers nums.
 3  Find the number of triplets (i, j, k) that meet these conditions:
 4      0 <= i < j < k < nums.length
 5      nums[i], nums[j], and nums[k] are pairwise distinct.
 6  Return the number of triplets that meet the conditions.
 7
 8  Example 1:
 9      Input: nums = [4,4,2,4,3]
10      Output: 3
11      Explanation: The following triplets meet the conditions:
12      - (0, 2, 4) because 4 != 2 != 3
13      - (1, 2, 4) because 4 != 2 != 3
14      - (2, 3, 4) because 2 != 4 != 3
15      Since there are 3 triplets, we return 3.
16      Note that (2, 0, 4) is not a valid triplet because 2 > 0.
17
18  Example 2:
19      Input: nums = [1,1,1,1,1]
20      Output: 0
21      Explanation: No triplets meet the conditions so we return 0.
22
23  Constraints:
24  3 <= nums.length <= 100
25  1 <= nums[i] <= 1000
26
27  ### Code Prompt
28  class Solution(object):
29      def unequalTriplets(self, nums: List[int]) -> int:
30
31  ### Completion
32  class Solution(object):
33      def unequalTriplets(self, nums: List[int]) -> int:
34          ans = 0
35          n = len(a)
36          for i in range(n):
37              for j in range(i + 1, n):
38                  for k in range(j + 1, n):
39                      ans += len({a[i], a[j], a[k]}) == 3
40          return ans
41
42  ### Question
43  <task_content>
44
45  ### Code Prompt
46  <code_starter>
47
48  ### Completion
49  <code_completion>
```

Figure 9: Code Generation Prompts. Lines 1 to 40 are the one-shot example. In Mercury experiments, we feed the *pretty_content* field to the placeholder <*task_content*>, the *prompt* field to the placeholder <*code_starter*>, and the *solution* field to the placeholder <*code_completion*>

A.10 Hardware-agnostic Evaluation

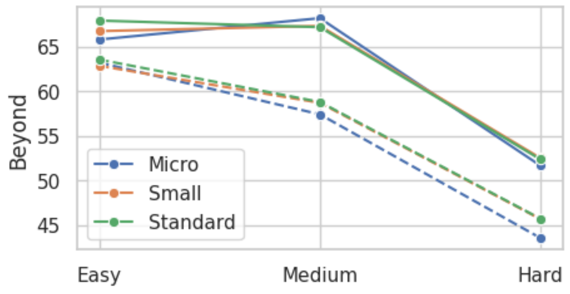


Figure 10: Beyond scores of 'deepseek-coder-33b' (solid line) and 'deepseek-coder-6.7b' (dashed line) across varied Intel Skylake CPU configurations. The results show that Beyond can remain consistent across different hardware configurations.

A.11 Distribution of Bootstrapped Beyond Scores

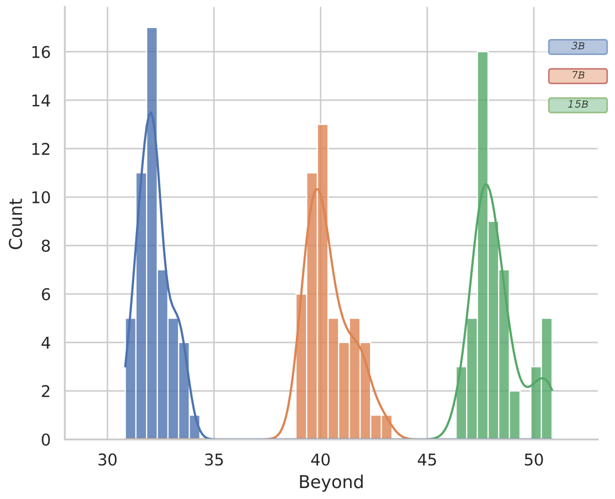


Figure 11: Bootstrapped Beyond Distribution. We evaluate 3B, 7B, and 15B Starcoder2[24] models using the *Mercury* benchmark. Each model was executed 50 times to ensure score robustness. The y-axis in the resulting histogram represents the frequency of observations within each bin.

A.12 Dataset Metadata

The Mercury dataset is hosted on Huggingface: `https://huggingface.co/datasets/Elfsong/Mercury`. The Croissant Metadata can be found at `https://huggingface.co/api/datasets/Elfsong/Mercury/croissant`.

A.13 Legal Compliance

In this study, we have curated a comprehensive dataset by gathering publicly accessible task descriptions and archived solutions from LeetCode (`https://leetcode.com/problemset/`). We have ensured that our collection process is strictly limited to tasks available in the free domain, intentionally excluding any content that falls under the paid services of the platform. We abide by Fair Use [33] (Section 107): *"the fair use of a copyrighted work, including such use by ... scholarship, or research, is not an infringement of copyright"*, where fair use is determined by *"the purpose and character of the use, including whether such use is of a commercial nature or is for nonprofit educational purposes"*. With the *Mercury* dataset, we emphasize its strictly non-commercial nature and underscore its purpose: to facilitate and advance academic research. The *Mercury* dataset is released under Creative Commons Attribution Non Commercial 4.0 [10].