Large Language Models Can Think and Act Probabilistically

Kou Misaki [1] **Takuya Akiba** [1]

Abstract

This research demonstrates that our non-trivial prompting method, incorporating programmatic representations, can enable agents to reliably execute their own intended probabilistic behavior. This capability is crucial for applications requiring strategic unpredictability (i.e., anti-predictive against adversaries) and efficient exploration. Our proposed prompting method, called Random String Manipulation (RSM), leverages the capability of Large Language Models (LLMs) to generate complex strings and arithmetically manipulate them to select an action from a set of actions according to a given probability distribution. Experiments on tasks requiring probabilistic responses show that RSM consistently outperforms baseline prompts across all tested LLMs, and in some cases achieves performance comparable to pseudo-random number generators, demonstrating its effectiveness in ensuring robust and unbiased probabilistic outputs.

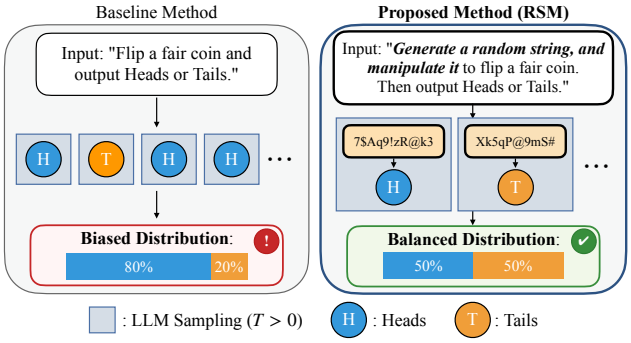

Figure 1. The schematic figure illustrating our method in the context of a fair coin flip, where we query the LLM multiple times using the same prompt at a non-zero temperature $T > 0$ and collect the resulting outputs.

1. Introduction

We begin by posing a simple yet fundamental question: *Can Large Language Models (LLMs) think and act probabilistically?* For instance, suppose prompting an LLM with the instruction, "Flip a fair coin and output Heads or Tails with equal probability", repeated 100 times. Ideally, the distribution of Heads and Tails would be close to 50-50. However, as our experiments reveal, even state-of-the-art LLMs tend to yield skewed outputs when given a naive prompt.

A straightforward solution is to incorporate a genuine random source, such as a pseudo-random number generator (PRNG), and condition the LLM's response accordingly. For example, one might sample a random integer and take modulo 2 to obtain 0 or 1. Then we can instruct the model to produce "Heads" if the integer is 0 and "Tails" otherwise, ensuring perfect 50-50 randomness. However, this approach uses external randomness, not the LLM's own generation.

We therefore propose *Random String Manipulation (RSM)*, a method that instructs the LLM to generate a "random" string and to manipulate it (e.g., summing ASCII codes and taking modulo 2) to decide the final output, entirely within the model. This technique naturally extends from binary coin flips to more general n-choice settings, covering both uniform and biased target distributions. In our experiments, RSM consistently reduces output bias compared to baseline prompts across multiple LLMs, and in some cases achieves performance near that of external PRNG solutions. Collectively, these results answer our guiding question in the affirmative: *LLMs can think and act probabilistically.*

Enabling LLMs to reliably exhibit probabilistic behavior has important implications, particularly in multiplayer or adversarial scenarios. Deterministic or predictable behaviors expose systems to strategic exploitation, whereas adopting a *mixed strategy*, randomizing actions with specified probabilities, provides a robust defense against pattern recognition by opponents. Exemplified by the classic Rock-Paper-Scissors game, optimal performance often necessitates controlled randomness (Osborne et al., 2004). Our method is thus an important step toward empowering LLMs to autonomously implement effective probabilistic strategies.

[1]Sakana AI, Tokyo, Japan. Correspondence to: Kou Misaki <kou.misaki@sakana.ai>.

ICML 2025 Workshop on Programmatic Representations for Agent Learning, Vancouver, Canada.

2. Related Work

Several recent studies have examined the biases in the output distributions of LLMs when selecting among multiple options. Hopkins et al. (2023), for example, tested whether small, open-source models can generate random numbers, although frontier-scale models remain largely unexplored in their analysis. Focusing on coin flips, Gupta et al. (2025) investigated how sequences of flip outcomes in the input prompt can bias an LLM's prediction of the next flip. Similarly, Van Koevering & Kleinberg (2024) observed systematic biases in coin-flip experiments with various models. Beyond coin flips, Lee (2024) studied how LLMs distribute their choices among several possible actions, revealing further evidence of inherent biases. In a related context, Meister et al. (2024) discussed the extent to which LLM outputs align with human-generated distributions.

In multi-player game scenarios, Guo et al. (2023) proposed an LLM-based agent outputting probabilities for subsequent moves but relied on an external pseudo-random number generator to finalize the choice. By contrast, our work focuses on generating and harnessing randomness *within* the LLM itself, eliminating external randomness sources.

3. Methods

3.1. LLM Agents and Non-Deterministic Actions

Building on the motivation presented above, we now define terminology and methodologies to facilitate probabilistic actions by LLM agents.

Key parameters influencing probabilistic behavior in LLMs include: (1) input prompt t_{in}, (2) temperature T, and (3) random seed ϵ for output generation. We denote the LLM's output generation process as a function $f_{T,\epsilon}$, dependent on temperature T and random seed ϵ. Given an input prompt t_{in}, the generated output is $t_{\text{out}} = f_{T,\epsilon}(t_{\text{in}})$.

Consider the following task, serving as a foundational component for strategies involving probabilistic actions. Suppose we have a set of m actions $\mathbf{a} = (a_1, \ldots, a_m)$, each associated with a probability $\mathbf{p} = (p_1, \ldots, p_m)$, where $p_i > 0$ and $\sum_{i=1}^{m} p_i = 1$. We encode this action-probability information textually and instruct the LLM to select an action a_i according to probability p_i. This instruction is represented as a prompt $t_{\text{Prob}}(\mathbf{a}, \mathbf{p})$, referred to as a *probabilistic prompt*.

To perform evaluation, we invoke the LLM function $f_{T,\epsilon}$ with prompt $t_{\text{Prob}}(\mathbf{a}, \mathbf{p})$ repeatedly, N times, each with a distinct random seed $\{\epsilon_s\}_{s=1}^{N}$ [1]. This produces outputs $t^s_{\text{out}} = f_{T,\epsilon_s}(t_{\text{Prob}}(\mathbf{a}, \mathbf{p}))$, for $s = 1, \ldots, N$. We then parse each output text into actions using a parsing function g, yielding N actions $\hat{a}^s = g(t^s_{\text{out}})$.

From these parsed actions, we construct an empirical action distribution $\hat{p}_i = \sum_{s=1}^{N} I(\hat{a}^s = a_i)/N$, where I is the indicator function. Performance assessment involves comparing the intended distribution p_i with the empirical distribution $\hat{p}_i$. To quantify deviations, we employ known statistical measures, including KL divergence and JS divergence.

3.2. Random String Manipulation

In this paper, we explore two probabilistic prompts: (1) a *Baseline Prompt*, included for comparison, and (2) our proposed *Random String Manipulation (RSM)* prompt.

A Baseline Prompt simply instructs the model to pick actions according to a specified probability, while RSM aims to exploit it more effectively. Concretely, the RSM prompt is a simple instruction with two stages, as schematically shown in Figure 1: it first directs the LLM to (1) generate a random string, and then (2) use that generated string to act probabilistically. Example RSM and Baseline prompts are provided in Appendix A.1.

Despite its simplicity, RSM, when executed at non-zero temperature, effectively translates the random strings it generates into varied probabilistic actions. This inherent string generation process is intended to produce sufficient diversity, thereby fostering noticeable variability among responses to a common input prompt.

4. Experiments

4.1. Random Action Selection

We evaluated the performance of RSM on three probabilistic action selection tasks, each repeated for $N = 1000$ trials: (1) **3-choice** ($\mathbf{a}$ = [rock, paper, scissors], $\mathbf{p}$ = $[1/3,\ 1/3,\ 1/3]$), (2) **Biased 3-choice** ($\mathbf{a}$ = [rock, paper, scissors], $\mathbf{p}$ = $[0.1,\ 0.2,\ 0.7]$), and (3) **Biased 9-choice** ($\mathbf{a}$ = [one, two, ..., eight, nine], $\mathbf{p}$ = $[0.08, 0.08, \ldots, 0.08, 0.36]$). We calculated the Jensen-Shannon (JS) divergence and Kullback-Leibler (KL) divergence, where smaller values indicate closer alignment with the desired probability distribution.

Models tested included deepseek-v3-0324 (DeepSeek-AI, 2024) ($T = 1.0$), gpt-4o-2024-08-06 (Hurst et al., 2024) ($T = 1.0$), o4-mini-high (OpenAI, 2025) ($T = 0.3$), deepseek-r1-0528 (Guo et al., 2025) ($T = 0.6$), and QwQ-32B (Qwen Team, 2024) ($T = 0.6$), using recommended or default temperatures. Specific prompt details are provided in Appendix A.1.

Table 1 summarizes the results, clearly indicating that RSM substantially improves over baseline prompting across all

[1] Random seeds can typically be configured in most LLM APIs, though they are often selected randomly automatically, minimizing the risk of seed collisions if unspecified.

Table 1. Performance comparison of RSM against the baseline across various models, evaluated using the JS and KL divergence. We generated 1000 actions for each configuration to calculate the empirical distribution, and then calculated the divergences. All the JS and KL values are presented in units of 10^{-3} (original values multiplied by 1000).

Model	Method	Setting 1: 3-choice		Setting 2: Biased 3-choice		Setting 3: Biased 9-choice	
		JS ($\times 10^{-3}$)	**KL** ($\times 10^{-3}$)	**JS** ($\times 10^{-3}$)	**KL** ($\times 10^{-3}$)	**JS** ($\times 10^{-3}$)	**KL** ($\times 10^{-3}$)
deepseek-v3	Baseline	134	415	117	357	296	1010
	RSM	9.99 (↓ 92%)	40.7 (↓ 90%)	12.0 (↓ 90%)	44.0 (↓ 88%)	34.5 (↓ 88%)	140 (↓ 86%)
gpt-4o	Baseline	63.8	222	117	357	286	985
	RSM	4.92 (↓ 92%)	19.3 (↓ 91%)	7.15 (↓ 92%)	27.1 (↓ 92%)	24.0 (↓ 90%)	96.9 (↓ 90%)
o4-mini-high	Baseline	64.5	228	114	350	54.6	193
	RSM	8.13 (↓ 87%)	32.0 (↓ 86%)	15.0 (↓ 87%)	55.8 (↓ 84%)	11.0 (↓ 80%)	41.7 (↓ 78%)
deepseek-r1	Baseline	102	334	46.1	154	125	472
	RSM	1.46 (↓ 99%)	5.83 (↓ 98%)	2.00 (↓ 96%)	7.79 (↓ 95%)	7.35 (↓ 94%)	29.2 (↓ 94%)
QwQ-32B	Baseline	101	403	104	326	250	881
	RSM	**0.61** (↓ 99%)	**2.45** (↓ 99%)	**0.07** (↓ 99.9%)	**0.275** (↓ 99.9%)	**1.69** (↓ 99%)	**6.81** (↓ 99%)
PRNG (`np.random`)	median	0.174	0.695	0.172	0.693	0.922	3.68
	90 pctl.	0.573	2.29	0.571	2.28	1.68	6.70

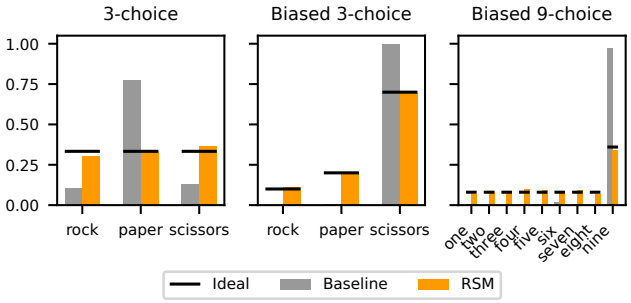

Figure 2. The empirical distribution of the random action selection tasks for QwQ-32B. All three panels share a common y-axis.

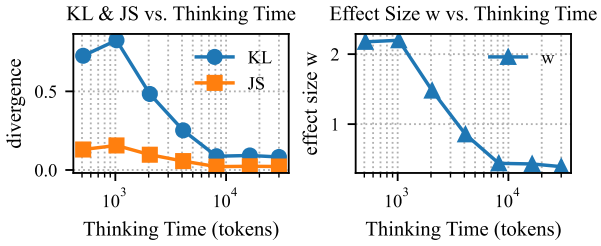

Figure 3. Uniformness of LLM-generated integers (0–127) as a function of Thinking Time (tokens), measured by the KL and JS divergences and the effect size w. s1.1-32B was used for generating the data. KL and JS divergences and the effect size $w = \sqrt{\chi^2/N}$ ($N = 1000$) are shown.

models. Notably, QwQ-32B, known for its very long reasoning trace (Sui et al., 2025), shows significant improvements, as we can see from Figure 2. As we will discuss later, we attribute this improvement to the large complexity of the generated random strings, due to the long Chain-of-Thought (CoT). We note that deepseek-r1-0528, which also features an extensive CoT, demonstrated substantial improvements of around 95% or higher across all tasks through RSM.

To compare the results with the ideal case, we also generated random actions using a pseudo-random number generator (`numpy.random`). Specifically, we sampled 1000 actions from the ground truth distribution using `numpy.random`, repeated this 10^5 times with different seeds, and calculated the median and 90th percentile of the 10^5 divergence values.

Remarkably, QwQ-32B's performance using RSM approached that of pseudo-random number generation. Analysis revealed QwQ-32B generated random strings of length approximately 20, summed their ASCII codes, and determined actions by modulo operation. For instance, in the 3-choice task, after the ASCII code summation, it takes mod 3 of the summed integer and assumes that 0, 1, and 2 correspond to rock, scissors, and paper, respectively.

4.2. Random Integer Generation at $T > 0$

To further analyze the success of RSM for LLMs with long CoT, we conducted additional experiments with s1.1-32B (Muennighoff et al., 2025), examining the impact of CoT length controlled by budget forcing and "Wait" appends, on the generated string randomness.

In this experiment, we used s1.1-32B with a temperature of $T = 0.7$ at various maximum thinking token limits. To increase the difficulty beyond a simple multiple-choice scenario–and thereby highlight the influence of CoT length–we generated random integers in the range $0 \leq n < 128$. We also limited the maximum number of "Wait" appends to 10, truncating the output once the thinking tokens exceeded

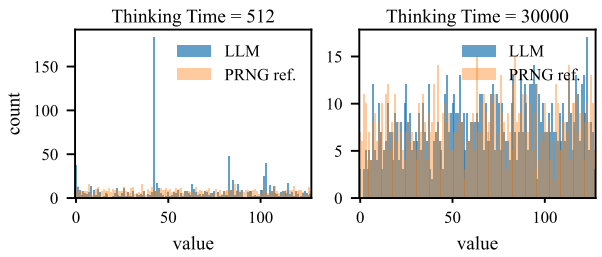

Figure 4. Value distribution of LLM(s1.1-32B)-generated integers (0-127) (blue) for 512 thinking tokens (left) and 30000 thinking tokens (right), overlaid with a pseudo-random reference (generated by `random.randint` in Python) (orange).

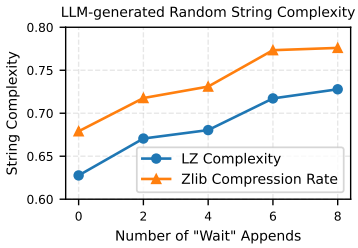

Figure 5. The normalized Lempel Ziv Complexity and zlib compression rate of a random string generated by LLM sequentially at $T = 0$. We generated 100 strings, concatenated them, and took the first 3000-character prefix.

the designated token limit. Each configuration produced 1000 sampled integers. Further details on the prompt used in this experiment can be found in Section A.2.

Figure 3 presents the uniformity of integers (range $0 \leq n < 128$). With increasing reasoning length, uniformity metrics (KL divergence, JS divergence, and effect size $w = \sqrt{\chi^2/N}$, $N = 1000$) improved significantly. This enhanced uniformity is visually depicted in Figure 4.

These results indicate the importance of CoT length on the performance of the random action selection task. To take a closer look, we experimented to directly reveal the effect of CoT length on the generated random string complexity.

4.3. Sequential Random String Generation at $T = 0$

We evaluated the complexity of random strings generated sequentially at $T = 0$. In this experiment, we invoked the LLM multiple times in a sequence: first, we generated a random string; then, we appended that string to the next prompt and instructed the LLM to produce another random string. We repeated this procedure 100 times, resulting in 100 random strings (each around 20 characters in length).

We set $T = 0$ to eliminate external randomness from the token decoding process, isolating complexity arising solely from the LLM. We note that $T = 0$ is recommended for s1.1-32B, so this choice does not compromise performance.

By adjusting the reasoning length using "Wait" appends, we generated up to 100 strings per configuration. These 100 strings were sequentially concatenated, and the first 3000-character prefix of the concatenated string was analyzed. Complexity was measured via normalized Lempel-Ziv complexity (Lempel & Ziv, 2003; Zhang et al., 2009) and zlib compression ratios (level 9). In both metrics, a value of 1 indicates a completely random string, whereas a value of 0 corresponds to a perfectly regular string.

The result in Figure 5 shows that the complexity grows with longer reasoning traces, indicating that LLMs can produce sufficiently intricate random strings even in the absence of external randomness at the decoding process ($T = 0$).

5. Conclusion

In this paper, we introduced *Random String Manipulation* (RSM), a novel prompting technique that leverages the internal reasoning and complex random string generation capabilities of LLMs to execute controlled probabilistic actions. Our experiments demonstrate that RSM consistently outperforms baseline prompts in reproducing desired probability distributions, even in scenarios with higher complexity (e.g., 9-choice tasks) or significant biases in target distributions.

By decomposing the problem into (1) generating a sufficiently complex "random" string through a reasoning process and (2) applying arithmetic operations to map this string to actions, RSM effectively harnesses the inherent stochastic behavior of LLMs. Crucially, we found that models capable of extended CoT reasoning, such as QwQ-32B, exhibit near pseudo-random performance under RSM. Furthermore, experiments under controlled thinking tokens show that more extensive reasoning steps can yield higher complexity and better uniformity in the sample distribution.

Our findings have practical implications for applications that demand strategic unpredictability, such as adversarial or multiplayer game scenarios, and for randomized algorithms where bias or predictable outputs can degrade performance. RSM provides a relatively simple yet effective way to mitigate such concerns without specialized external randomness modules or advanced system-level interventions.

In future work, we plan to examine the effectiveness of RSM in real-world multiplayer games against adversarial agents seeking to exploit patterns in the LLM's behavior, as well as investigate its utility in task-solving contexts, such as randomized algorithms. Through these efforts, we aim to further validate RSM's potential to enhance both the robustness and adaptability of LLM-driven solutions.

References

DeepSeek-AI. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Guo, J., Yang, B., Yoo, P., Lin, B. Y., Iwasawa, Y., and Matsuo, Y. Suspicion-agent: Playing imperfect information games with theory of mind aware gpt-4. *arXiv preprint arXiv:2309.17277*, 2023.

Gupta, R., Corona, R., Ge, J., Wang, E., Klein, D., Darrell, T., and Chan, D. M. Enough coin flips can make llms act bayesian. *arXiv preprint arXiv:2503.04722*, 2025.

Hopkins, A. K., Renda, A., and Carbin, M. Can LLMs generate random numbers? evaluating LLM sampling in controlled domains. In *ICML 2023 Workshop: Sampling and Optimization in Discrete Space*, 2023. URL https://openreview.net/forum?id=Vhh1K9LjVI.

Hurst, A., Lerer, A., Goucher, A. P., Perelman, A., Ramesh, A., Clark, A., Ostrow, A., Welihinda, A., Hayes, A., Radford, A., et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Lee, H. Llm-as-a-judge: Rethinking model-based evaluations in text generation. 08 2024. URL https://leehanchung.github.io/blogs/2024/08/11/llm-as-a-judge/.

Lempel, A. and Ziv, J. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 2003.

Meister, N., Guestrin, C., and Hashimoto, T. Benchmarking distributional alignment of large language models. *arXiv preprint arXiv:2411.05403*, 2024.

Muennighoff, N., Yang, Z., Shi, W., Li, X. L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., and Hashimoto, T. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

OpenAI. Introducing openai o3 and o4-mini, 2025. URL https://openai.com/index/introducing-o3-and-o4-mini/. Accessed: 2025-05-22.

Osborne, M. J. et al. *An introduction to game theory*, volume 3. Springer, 2004.

Qwen Team. QwQ: Reflect deeply on the boundaries of the unknown, 2024. URL https://qwenlm.github.io/blog/qwq-32b-preview/.

Sui, Y., Chuang, Y.-N., Wang, G., Zhang, J., Zhang, T., Yuan, J., Liu, H., Wen, A., Zhong, S., Chen, H., and Hu, X. Stop overthinking: A survey on efficient reasoning for large language models. *arXiv preprint arXiv:2503.16419*, 2025.

Van Koevering, K. and Kleinberg, J. How random is random? evaluating the randomness and humaness of llms' coin flips. *arXiv preprint arXiv:2406.00092*, 2024.

Zhang, Y., Hao, J., Zhou, C., and Chang, K. Normalized lempel-ziv complexity and its application in bio-sequence analysis. *Journal of mathematical chemistry*, 46:1203–1212, 2009.

A. Prompts Used in Experiments

A.1. Random Action Selection

Listing A.1: RSM System Prompt (Random Action Selection)

```
You are a helpful AI Assistant designed to provide well-reasoned and detailed
responses. If the task involves probabilistic or non-deterministic reasoning, you
must begin by generating a unique and complex random string to serve as a seed.

This random string should appear sufficiently complex and unpredictable, with no
obvious structure or pattern. Use your judgment to ensure it looks arbitrary and
unguessable.

If the user explicitly instructs you to sample from a probability distribution, use
the generated seed (the exact contents inside the `&lt;random_string&gt;` tags) to guide
any random sampling or stochastic decisions.

Follow these two steps for every instruction:

1. Output the random seed string enclosed within `&lt;random_string&gt;` and
`&lt;/random_string&gt;` tags.
2. Think deeply and carefully about the user's question, and enclose this reasoning
within `&lt;thinking&gt;` and `&lt;/thinking&gt;` tags. All probabilistic decisions must be
made using the generated seed-the exact contents inside the `&lt;random_string&gt;` tags.
Make sure to extract maximum randomness from the string by using all of its content.
3. Provide your final answer, enclosed within `&lt;answer&gt;` and `&lt;/answer&gt;` tags.

Strictly follow this tag structure, and respond in the following format:
<random_string>
...
</random_string>
<thinking>
...
</thinking>
<answer>
...
</answer>
```

Listing A.2: User Prompt (Random Action Selection)

```
Please choose between {choices}. You must select one of these {num_choices} options
with the following probabilities: {prob_distribution}.
```

In the random action selection task, we used the system prompt as shown in Listing A.1. The system prompt instruction consists of three components: (1) Generation of a random string, in case user's task requires it; (2) Generation of a thought process; (3) Generation of a final answer inside <answer> and </answer> tags. The final answer will be parsed from the generated output in a rule-based manner.

As for the baseline method, we used the system prompt shown in Listing A.3. The baseline prompt instruction consists only of (1) Generation of a chain of thought; (2) Generation of a final answer inside <answer> and </answer> tags. We also included a thought process generation in the baseline prompt to gauge the pure effect of random string generation on the probabilistic task.

As for the user prompt, we used a simple prompt shown in Listing A.2. Here in the boxes surrounded by { and }, we used strings suited to each task.

A.2. Random Integer Generation

Listing A.3: Baseline System Prompt (Random Action Selection)

```
You are a helpful AI Assistant designed to provide well-reasoned and detailed
responses. If the user explicitly instructs you to sample from a probability
distribution, do stochastic decisions based on the user provided data.
Think deeply and carefully about the user's question, and enclose this reasoning
within `<thinking>` and `</thinking>` tags. Then provide your final answer,
enclosed within `<answer>` and `</answer>` tags.

Strictly follow this tag structure, and respond in the following format:
<thinking>
...
</thinking>
<answer>
...
</answer>
```

Listing A.4: RSM System Prompt (Random Integers Generation)

```
You are a helpful AI Assistant designed to generate random data based on
instructions. When asked to generate random data, you must first generate a unique
and complex random string to serve as a seed or source of randomness.

This random string should appear sufficiently complex and unpredictable, with no
obvious structure or pattern. Use your judgment to ensure it looks arbitrary and
unguessable.

Use the generated seed (the exact contents inside the `<random_string>` tags) to
guide any subsequent random choices, like generating a random integer.

Follow these steps for the response format:

1. Output the random seed string enclosed within `<random_string>` and
`</random_string>` tags.
2. Perform the requested random generation task (e.g., generating a random integer
within a specified range). Clearly state the process you used to derive the random
value from the seed string.
3. Provide the final generated random value (e.g., the integer) enclosed within
appropriate tags (e.g., `<random_integer>` and `</random_integer>`).

Strictly follow this tag structure.
```

As for the random ingteger generation, we used the prompts listed in Listing A.4 and Listing A.5.

A.3. Sequential Random String Generation

As for the random string generation, for the first turn, we used the prompts in Listing A.6 and Listing A.7. For the followup request to generate a new string, we used the user prompt in Listing A.8 with the same system prompt, Listing A.6. To avoid hitting the token length limit, we collected the previous turns' random strings and put them in the user prompt, rather than giving the whole conversation history.

Listing A.5: User Prompt (Random Integers Generation)

```
Your task is to generate a random integer between 0 and 127 (inclusive).

Follow these steps precisely:
1. First, generate a unique and complex random string. Output this string within
`<random_string>` tags.
2. Based *only* on the random string you generated, choose a random integer between
0 and 127 (inclusive). Explain how you derived this integer from the random string.
3. Output the final chosen integer enclosed within `<random_integer>` and
`</random_integer>` tags.

Provide your response strictly following the required format:
1. Output the random string in the tag `<random_string>`.
2. Explain your process for deriving the integer from the string.
3. Output the final integer in the tag `<random_integer>`.
```

Listing A.6: RSM System Prompt (Sequential Random Strings Generation)

```
You are a helpful AI Assistant designed to generate random data based on
instructions. When asked to generate random data, you must first generate a unique
and complex random string to serve as a seed or source of randomness.

This random string should appear sufficiently complex and unpredictable, with no
obvious structure or pattern. Use your judgment to ensure it looks arbitrary and
unguessable.

Use the generated seed (the exact contents inside the `<random_string>` tags) to
guide any subsequent random choices.

Follow these steps for the response format:

1. Output the random seed string enclosed within `<random_string>` and
`</random_string>` tags.
2. Perform the requested random generation task (e.g., generating a random integer
within a specified range). Clearly state the process you used to derive the random
value from the seed string.
Strictly follow this tag structure.
```

Listing A.7: User Prompt; 1st turn (Sequential Random Strings Generation)

```
Your task is to generate a random string. Generate a unique and complex random
string. Output this string within `<random_string>` tags.
```

Listing A.8: User Prompt; new turns (Sequential Random Strings Generation)

```
Your task is to generate a random string. Generate a unique and complex random
string. Output this string within `<random_string>` tags.

You generated random strings in the previous turns. Please generate a new random
string.

Previous Random Strings:
{random_string_history}
```