The Mathematics of Luck: Number Theory in the Gacha System of Love and Deepspace

Bertha Soliany Frandi - 13523026[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]bertha,soliany@gmail.com, 13523026@std.stei.itb.ac.id

***Abstract*—This paper explores the implementation and analysis of gacha systems in the context of "Love and Deepspace," focusing on integrating number theory principles such as modular arithmetic and probabilities. This study highlights the design and functionality of gacha systems, demonstrating how they replicate the dynamics of real-world gacha games within the limitations of the Ren'Py engine. Additionally, the paper evaluates the effectiveness of the implemented system through simulated results, providing insights into the statistical behavior and user experience of the gacha system. By examining the intersection of mathematics and game design, this research underscores the importance of mathematical principles in shaping interactive entertainment while addressing the ethical and economic considerations surrounding gacha mechanics.**

***Keywords*—Gacha Systems, Number Theory, Otome Games, Pseudorandom Number Generation.**

I. Introduction

The charm of games has enchanted players all over the world. This is particularly true for games that involve elements of unpredictability and strategy. Among these, games that combine gacha system and otome games hold a unique place in gamer's heartper. These types of games blend mathematics, storytelling, and chance to create engaging experiences. This paper emerges from the author's fascination with these games. As a dedicated player of both gacha and otome games, the intricate mechanics and mathematical frameworks underpinning their design have often sparked curiosity and inspired deeper investigation.

The word gacha is derived from the Japanese term for capsule toy vending machine, gachapon. Gachapon itself is an onomatopoeia where gacha is the sound of the turning in the vending machine and pon is the sound that the capsule makes when falling. Gacha systems employ probability-driven mechanics that dictate outcomes ranging from acquiring rare items to shaping narrative paths. These systems are characterized by their reliance on randomized rewards, where players expend in-game resources or real-world currency for a chance to obtain desired items. The mechanics often involve layered probability structures, such as pity systems and drop-rate adjustments, which add complexity to the player's decision-making process. On the other hand, otome games are narrative-driven experiences that center on romantic storytelling. These games provide players with choices that influence the progression and outcomes of the plot. When gacha mechanics are integrated into otome games, they create a dynamic interplay between statistical odds and emotional engagement, encouraging players to strategize while immersing themselves in the storyline. This blend of mathematical unpredictability and narrative depth exemplifies how entertainment and intellectual intrigue can coexist within modern gaming.

The urgency of this study stems from the increasing prevalence of gacha systems across global gaming platforms and their impact on player behavior, economics, and design ethics. By examining the mathematical structures within these systems, this paper aims to demystify their mechanics, fostering a greater understanding of how probabilities and algorithms influence player experiences. Furthermore, this exploration seeks to shed light on the implications of these systems, particularly in promoting responsible gaming and informed decision-making among players.

The primary objective of this research is to bridge the gap between mathematical theory and practical application within the context of gacha systems and otome games. Through an analysis of number theory, this study aspires to provide insights into the design strategies that govern these games, emphasizing their role in shaping player engagement. By doing so, this paper contributes to the broader discourse on the integration of mathematics in gaming, underscoring its significance as both an art form and a discipline rooted in logic and chance.

II. Theoretical Foundation

A. Number Theory

Number theory is devoted to the study of integers and integer-valued functions. It is a branch of pure mathematics. Its historical significance lies in its applications to cryptography, coding theory, and the development of algorithms in computer science. Fundamental concepts in number theory include divisors, prime numbers, and modular arithmetic.

Prime numbers, defined as integers greater than one with no divisors other than one and themselves, play a pivotal role in modern cryptographic protocols and randomization algorithms. These properties are crucial for creating secure and fair gacha systems.

Modular arithmetic, which involves computations with remainders, serves as the backbone of numerous cryptographic algorithms, such as RSA encryption. The concept of greatest common divisors (GCD) and the Euclidean algorithm further enhance our ability to simplify and solve problems involving

integers. To put it simply modular arithmetic is a system of arithmetic for integers where numbers "wrap around" after reaching a certain value, called the modulus. In gacha systems, modular arithmetic underpins the Pseudorandom Number Generators (PRNGs) that ensure fairness and unpredictability.

A Pseudorandom Number Generator (PRNG) is an algorithm designed to produce sequences of numbers that appear random but are generated deterministically using mathematical formulas. One of the simplest and most widely used PRNGs is the Linear Congruential Generator (LCG). LCG generates numbers based on the recursive formula.

$$X_{n+1} = (aX_n + c) \bmod m$$

Parameter a is the multiplier, c is the increment, m is the modulus, and X_n is the current number in sequence (or seed). The modulus m ensures the output remains within a specific range, while the constants a and c affect the quality of the randomness and the length of the cycle before numbers start repeating. PRNGs like LCG are computationally efficient and ideal for simulations, gaming randomness, and other non-cryptographic applications. However, because they are deterministic and have limited periods, they are unsuitable for high-security tasks like cryptography, where truly random or cryptographically secure random numbers are required.

In number theory, probability and combinatorics is a relevant topic. Number theory often intersects with probability in calculating the likelihood of outcomes in gacha systems. Understanding these principles helps quantify the odds of obtaining rare items based on predefined drop rates.

Another critical aspect of number theory is its exploration of congruences. The study of congruences, initiated by Carl Friedrich Gauss, enables the development of efficient algorithms for solving systems of linear and non-linear equations over integers. Additionally, advanced topics such as Euler's totient function and Fermat's Little Theorem provide foundational tools for evaluating large modular exponentiations in cryptographic systems. This body of work demonstrates how number theory bridges abstract mathematical principles and practical computational applications.

B. Gacha Games

One of many subgenres of mobile and video games is gacha games. Gacha games employ a monetization mechanism inspired by Japanese capsule toy vending machines. In gacha games, players spend in-game currency to receive randomized virtual items. The gacha system has evolved into a global phenomenon, blending entertainment and monetization strategies.

Common gacha models include "Standard," "Limited-Time," and "Step-Up" banners, each offering varying probabilities of acquiring high-value items or characters. Standard Gacha is the most basic form where players roll for items with fixed probabilities. Limited-time gacha offer exclusive rewards that are only available during specific times. Step-Up gacha increases rewards or higher probabilities for rare items as players perform consecutive rolls.

To enhance player satisfaction, developers often incorporate "pity systems" to guarantee rare items after a specific number of attempts, addressing player frustration and promoting continued engagement. There are "soft pity" and "hard pity". A "soft pity" system gradually increases the probability of obtaining rare items, while a "hard pity" system guarantees a rare item after a fixed number of pulls.

C. Love and Deepspace

"Love and Deepspace," a next-generation otome game developed by Infold Games, introduces groundbreaking innovations in the genre of romantic visual novels. Set in a futuristic universe, the game combines elements of space exploration, political intrigue, battle systems, and complex character relationships. Players assume the role of a customizable protagonist navigating interstellar conflicts while forming romantic bonds with a diverse cast of characters.

The game's appeal is enhanced by its advanced graphics engine, immersive storytelling, and integration of gacha mechanics. Players collect in-game assets, such as costumes and memories used for battle, by participating in gacha draws. The gacha mechanics in "Love and Deepspace" include a system where players can perform single summons or multi-summons (typically ten at a time) to acquire memories. Each memory is assigned a star rating, indicating its rarity and power, with higher-starred memories being significantly rarer. Talking about memory, memory is a character card that players can acquire. Memory is used for the battle system that the players need to accomplish to advance the story. To put it simply, a higher-starred memory is required for the gameplay.

In Love and Deepspace, the gacha itself has the name of Xspace Echo. This is a "Normal" banner with a hard pity of 70 pulls. In all gacha banners, 150 Diamonds (the in-game currency) are needed for one single pull. So, the player needs to have 1500 to do 10 pulls. Usually, for every banner, the rate for memory is 92% for three-starred memory, 7% for four-starred memory, and 1% for five-starred memory. For every 10 pulls, it is guaranteed to obtain a four-star memory while for every 70 pulls, it is guaranteed to obtain a five-star memory. There is also an indication that Love and Deepspace gacha systems have a soft pity system, but this information is not yet to be official.

Through this theoretical framework, this paper builds a foundation for analyzing how mathematical principles and design strategies converge in the gacha systems of "Love and Deepspace." The findings aim to contribute to the broader understanding of gaming mechanics and player engagement.

III. Implementation

For the implementation of the gacha system, the author use Ren'Py, a visual novel engine known for its flexibility and simplicity. Ren'Py was chosen for this project because it provides a suitable platform for simulating a gacha-like experience while maintaining a balance between functionality and accessibility. Although the resulting system cannot match the visual and technical quality of real gacha games, it effectively replicates the core mechanics and atmosphere. The gacha system is built using a Pseudorandom Number Generator (PRNG) alongside custom functions. These functions implement both hard pity and soft pity systems, ensuring fairness and replicating the reward dynamics typical of actual gacha games.

A. Pseudorandom Number Generators (PRNGs)

```python
# PRNG parameters
self.a = 1664525
self.c = 1013904223
self.m = 2**32  # large prime modulus
self.seed = int(time.time() * 1000) % self.m  # initial seed

def prng(self):
    self.seed = (self.a * self.seed + self.c) % self.m
    return self.seed / self.m
```

Fig. 3.1 PRNG Function (Source: Author).

The provided code implements a simple Pseudorandom Number Generator (PRNG) using a Linear Congruential Generator (LCG). It generates reproducible random sequences based on the initial seed. a is a constant multiplier for the LCG formula which determines the randomness properties. c is a constant increment that ensures the generated sequence does not become trivial (i.e., all zeros). m is the modulus which defines the range of values generated by the LCG. In this case, m is set to 2^{32} which allows a large range of random values. *seed* is the starting value of the random sequence. It is set based on the current time (milliseconds). This ensures the seed remains within the range defined by the modulus.

$$seed = (a \times seed + c) \bmod m$$

This formula updates the seed to a new value based on the multiplier, increment, and modulus. This formula is the core of the LCG. The return value for the *prng* function is normalized so the output is suitable for applications requiring a fractional random value. The result of $seed \div m$ scales the generated number to range between 0 to 1 (inclusive).

B. Drawing

The implementation of handling pulling results is the core of a gacha system. For drawing, there are two Python functions and one Ren'Py label. There is a function for an overall drawing process and there is function for handling pity system alongside drawing the memory one by one. The Ren'Py label is used for showing the result that has been through the two Python functions.

Draw function determines the results of pulls. After checking if the player has enough resources to pull it then deduct the balance that the player has. The function then loops through the number of pulls and calls the other function for drawing, *draw_single*, to determine the results of each pull. It also simultaneously updates the flags (*four_star_obtained*, *five_star_obtained*, and *event_memory_obtained*) based on the tier and type of the obtained memory.

In the *draw* function, there are codes to ensure a few gacha requirements. The first one is for guarantee four-star in 10 pulls. If the player performs 10 pulls and no four-star memory was obtained, the function will iterates through the results to replace the first three-star memory with a randomly selected four-star memory. The second requirement is for a guarantee five-star memory in event banner. If the event pity count exceeds the threshold (in this case 70) and no five-star memory was obtained, it will reset the *event_pity_count* and replaces the first three-star memory in the results with a five-star memory. Of course there is also a checking for the previous five-star pull. The last requirement that is check is for a guarantee five-star in normal banner. The logic for this is similar to the event banner. It check whether the player obtain a five-star when the pity count equal to the pity threshold and then proceeding to switch a three-star with a five-star if the player hasn't obtain a five-star memory.

So, the key features of *draw* function is a four-star guarantee that ensure every 10 pulls include at least one four-star memory, a event-specific guarantee that check if the previous five-star was non-event, the next five-star is guaranteed to be an event-specific memory, and lastly a five-star pity that guarantees a five-star memory after reaching the pity threshold, with separate logic for event and normal banners.

```python
def draw(self, banner="normal", pulls=1):
    if not self.can_afford(pulls):
        return ["You do not have enough diamonds to pull."]

    self.deduct_coins(pulls)
    results = []
    four_star_obtained = False
    five_star_obtained = False
    event_memory_obtained = False

    for _ in range(pulls):
        result = self.draw_single(banner=banner)
        results.append(result)
        if self.get_memory_tier(result) == "Four-star":
            four_star_obtained = True
        if self.get_memory_tier(result) == "Five-star":
            five_star_obtained = True
            if banner == "event" and result in self.event_memory and self.previous_non_event:
                event_memory_obtained = True

    # untuk guaranteed four-star every 10 pulls
    if (pulls == 10 and not four_star_obtained):
        for i in range(len(results)):
            if self.get_memory_tier(results[i]) == "Three-star":
                results[i] = self.memories_by_tier["Four-star"][int(self.prng() * len(self.memories_by_tier["Four-star"]))]
                break

    # untuk guaranteed five-star ketika pas di 70/70 event banner
    if (self.event_pity_count == self.pity_threshold and not five_star_obtained):
        self.event_pity_count = 0


        if self.previous_non_event:
            five_star_memories = self.event_memory
            self.previous_non_event = False
        else:
            five_star_memories = self.memories_by_tier["Five-star"] + self.event_memory
            self.guarantee_pity_count += 1

        for i in range(len(results)):
            if self.get_memory_tier(results[i]) == "Three-star":
                results[i] = five_star_memories[int(self.prng() * len(five_star_memories))]
                break

    # untuk guaranteed five-star ketika pas di 70/70 normal banner
    if (self.normal_pity_count == self.pity_threshold and not five_star_obtained):
        self.normal_pity_count = 0

        for i in range(len(results)):
            if self.get_memory_tier(results[i]) == "Three-star":
                results[i] = self.memories_by_tier["Five-star"][int(self.prng() * len(self.memories_by_tier["Five-star"]))]
                break

    return results
```

Fig. 3.2 draw Function (Source: Author).

The second function is *draw_single*. *draw_single* integrates advanced gacha mechanics for generating random results with varying probabilities. It handles the selection of a memory based on tier probabilities, soft pity, and hard pity mechanics. It also incorporates rules for event-exclusive banners, guarantees a high-tier memory at specific thresholds, and dynamically adjusts probabilities to simulate a realistic gacha system.

```python
# soft pity logic
if banner == "event" and pity_count >= self.soft_pity_start:
    additional_probability = (pity_count - self.soft_pity_start) * self.soft_pity_increment
    effective_five_star_probability = min(
        self.tier_probabilities["Five-star"]["event"] + additional_probability,
        100
    )
else:
    effective_five_star_probability = self.tier_probabilities["Five-star"]["event"]
```

Fig. 3.3 Soft Pity Logic (Source: Author).

First, there are pity calculations. It determines the pity counters based on the banner type. Second, there is soft pity logic. It applies soft pity if the event banner's pity count surpasses the soft pity threshold. It then calculates an incremental probability based on pity count.

```
# random selection logic
rand_num = self.prng() * 100  # scale to [0, 100)
cumulative = 0
selected_tier = None

for tier, probability in self.tier_probabilities.items():
    if isinstance(probability, dict):
        probability = (
            effective_five_star_probability
            if tier == "Five-star" and banner == "event"
            else probability[banner]
        )
    cumulative += probability
    if rand_num < cumulative:
        selected_tier = tier
        break
```

Fig. 3.4 Random Selection Logic (Source: Author).

After applying soft pity for the event banner, the function then goes through a random selection logic. *Rand_num* generates a random number between 0 and 100. It then iterates through *tier_probabilities* to determine the probability of selecting each tier. The probability for "Five-star" memories during the event banner is adjusted using *effective_five_star_probability*. At the end, it will select the first tier where *rand_num* is less than the cumulative probability.

```
# select memory
memories = self.memories_by_tier[selected_tier]
selected_memory = memories[int(self.prng() * len(memories))]
```

Fig. 3.5 Selecting Memory (Source: Author).

The code then retrieves all memories within the chosen tier and uses PRNG to randomly select one memory.

The pity counters and hard pity logic comes at the end of the function. The pity counters will increase the pity count and *guarantee_pity_count*. *guarantee_pity_count* is used to track every five-star that the player pulls in event banner. It also updates *previous_non_event* flag.

For hard pity logic, it is used to guarantee a five-star memory if the pity threshold is reached. For the event banner, it ensures an event memory on every second pity and then reset it. Otherwise, it includes both regular five-star and event memories in the random selection. It then returns the selected memory as the output of the *draw_single* function.

Lastly, there is Ren'Py label. The label pull in Ren'Py implements the logic for handling a gacha pull in the game where the player spends diamonds to obtain in-game "memories." The pull can return various types of results based on probabilities with special handling for five-star and event memories. This label interacts with the *gacha* system to determine the results of the pull. The label pull orchestrates the entire process of executing a gacha pull, from checking affordability and determining the results to displaying visuals and updating the player's inventory. It provides special handling for five-star and event memories, ensuring a satisfying visual and auditory experience for the player.

```
label pull:
    if gacha.can_afford(pulls):
        python:
            results = gacha.draw(banner=banner, pulls=pulls)
            for result in results:
                if result == "You do not have enough diamonds to pull.":
                    renpy.say(None, result)
                    break
                obtained_memories[result] += 1
                memory_tier = gacha.get_memory_tier(result)

                if result in memory_images:
                    if memory_tier == "Five-star":
                        renpy.show(memory_images[result], at_list=(custom2, memory_zoom))
                        renpy.show(blinkr, at_list=(custom3r, memory_zoom2))
                        renpy.show(blinkl, at_list=(custom3l, memory_zoom2))

                        if result in gacha.event_memory:
                            renpy.say(None, f"You got Event {memory_tier} memory: {result}!")
                        else:
                            renpy.say(None, f"You got a {memory_tier} memory: {result}!")
                        renpy.hide(blinkr)
                        renpy.hide(blinkl)
                        renpy.hide(memory_images[result])
                    else:
                        renpy.show(memory_images[result], at_list=(custom2, memory_zoom))
                        renpy.say(None, f"You got a {memory_tier} memory: {result}!")
                        renpy.hide(memory_images[result])
            renpy.say(None, f"You now have {gacha.coins} diamonds remaining.")
    else:
        "You do not have enough diamonds to pull."
        call coin from _call_coin

    call menu from _call_menu_2
    return
```

Fig. 3.6 Ren'Py Code (Source: Author).

C. Other functions

```
def can_afford(self, pulls):
    return self.coins >= pulls * self.cost_per_pull

def deduct_coins(self, pulls):
    self.coins -= pulls * self.cost_per_pull

def get_memory_tier(self, memory_name):
    for tier, memories in self.memories_by_tier.items():
        if memory_name in memories:
            return tier
    if memory_name in self.event_memory:
        return "Five-star"
    return "Unknown"
```

Fig. 3.7 Complementary Function (Source: Author).

The provided image shows three functions that help the implementation of gacha system. The first function *can_afford* purposes if to check whether the player has enough coins (in this case Diamonds) to perform a specified number of pulls. The *pulls* parameter is the number of pulls the player wants to perform. It multiplies the number of pulls by the cost per pull and compares the result with the player's available coins. It is set that every one pull cost 150 coins (the value of *cost_per_pull*). The function then returns a boolean value (*True* or *False*). Return *True* if the player can afford the pulls.

The second function, *deduct_coins* is for deducting the appropriate amount of coins from the player's balance after performing a certain number of pulls. The parameter for this function is the same as the previous function. The logic for this function is multiplying the number of pulls with the cost per pull and subtracts the result from the player's current coin balance. This function does not return anything as it simply updates the *coins* variable.

The third function is *get_memory_tier*. This function is for determining the tier of a given memory. The parameter, *memory_name*, is the name of the memory for which the tier is being determined. This function iterates through the dictionary *memory_by_tier*, where keys are tiers and values are list of

memory names belonging to those tiers. If the *memory_name* is found in the list of memories for a tier, the function returns the corresponding tier. If the *memory_name* is in the *event_memory* list, it is considered a "Five-star" event memory. If the memory is not found in any category, the function returns "Unknown".

D. Memories and Tiers

As mentioned before, there is a dictionary of memories where the keys are tiers, and the values are a list of memories. This dictionary is an initialize data for normal banner. For the event banner, the memories are initialized with a list of memories because for this paper, the only tier for the event memory is a five-star memory. There is also a dictionary for probability. Here are the images for a clear picture.

```
self.tier_probabilities = {
    "Five-star": {"event": 1, "normal": 1},
    "Four-star": 6,
    "Three-star": 92,
}
```

Fig. 3.8 Initialized Probability (Source: Author).

```
self.memories_by_tier = {
    "Five-star": [
        "Blossoms",
        "Business Trip",
        "Captivating Flavor",
        "Cozy Afternoon",
        "Deep Sea Riches",
        "Fluffy Trap",
        "Fragment Of Time",
        "Gentle Twilight",
        "Immobilized",
        "Nightplumes",
        "Precious Bonfire",
        "Promise Everlasting",
        "Sparkling Traces"
    ],
```

```
    "Four-star": [
        "A Day Of Snow",
        "A Long Night",
        "Beginning",
        "Hidden Shadow",
        "Lost Signal",
        "Razor's Dance",
        "Sapphire Scar",
        "Scorching Rain",
        "Spring Remnants",
        "Starry Sound",
        "Tender Curve",
        "Wild Gaze"
    ],
```

Fig. 3.9 Initialized Memories 1 (Source: Author).

```
    "Three-star": [
        "Backlight",
        "Clean Up",
        "Guardian",
        "Just Encounter",
        "Lame Party",
        "Listen",
        "My Decision",
        "Nestle",
        "Shoo In",
        "Social Occasion",
        "Sweet Burden",
        "Thoughts"
    ],
}
```

```
self.event_memory = [
    "Abyssal Mark",
    "Floof Attack",
    "Fluffy Treatment",
    "Goodcat Code",
    "Seething Flames",
    "Snowfall Encounter",
    "Temple Promise",
    "Tender Night",
    "Unforgettable Adventure"
]
```

Fig. 3.10 Initialized Memories 2 (Source: Author).

IV. Results

These are the results of the gacha system implementation developed using Ren'Py. The outcomes demonstrate how the implemented system simulates the mechanics of real-world gacha games, including the functionality of hard pity and soft pity systems. Screenshots and test outputs are provided to illustrate the effectiveness of the Pseudorandom Number Generator (PRNG) in producing randomized pulls and ensuring fairness. Additionally, it highlights the user interface and interactions withing the system, showcasing how the core gacha experience was recreated, albeit with limitations compared to commercial gacha games. These results validate the functionality of the implementation and its ability to simulate a compelling gacha-like environment.

A. Screenshots and Demonstration

These are the results for the interface using Ren'Py.

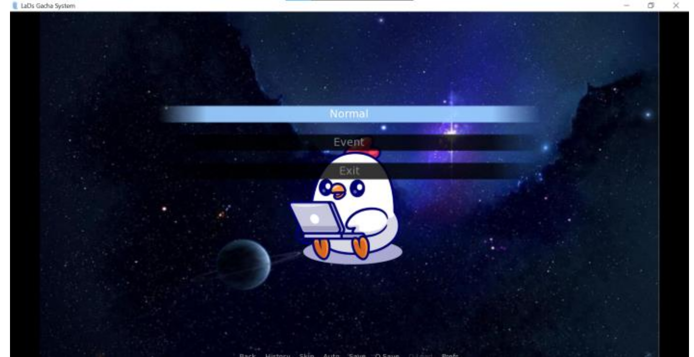

Fig. 4.1 Choices (Source: Author).

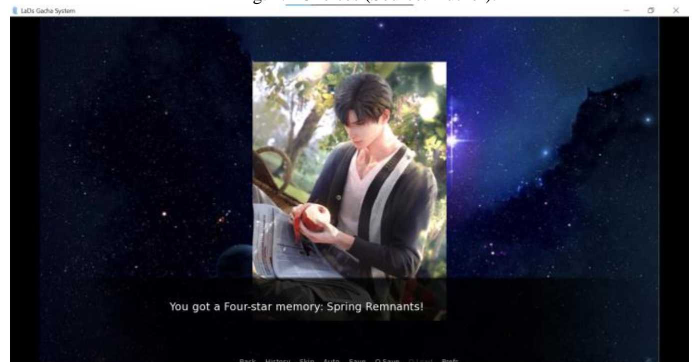

Fig. 4.2 Pull Result (Source: Author).

Fig. 4.3 Pity Count for Normal Banner (Source: Author).

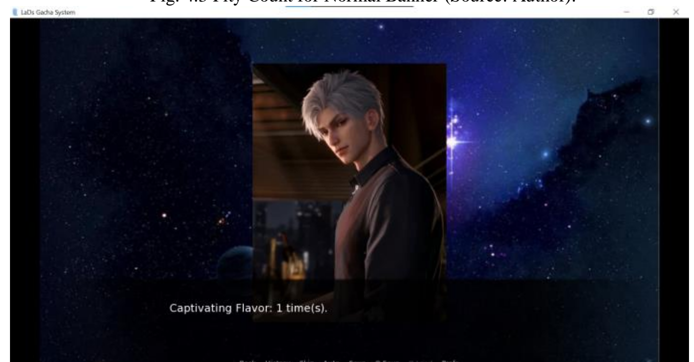

Fig. 4.4 Gacha Results Recap (Source: Author).

B. Distribution Table

100 pulls were simulated to assess the efficacy of the gacha mechanism that was put into place. The findings were examined to ascertain how results were allocated among various tiers and

to track how the pity system behaved. The method of distribution of results is summed up in the table below:

Tab. 4.1 Distribution Table

Tier	Percentage (%)	Number of Pulls	Expected Probability (%)
Five-star	2	2	1 (normal) or 1 (event)
Four-star	18	18	6
Three-star	80	80	92

The following is a summary of the gacha simulation's outcomes for 100 pulls on the event banner: Five-star memories accounted for 2% (2 pulls), Four-star memories made up 18% (18 pulls), and Three-star memories dominated with 80% (80 pulls). The results presented show minor differences from the event banner's base probabilities because of the PRNG's randomness. The slightly higher Five-star percentage (2% compared to the combined base of 2%) and the higher percentage of Four-star memories (18% compared to the base 6%) point to the impact of the soft pity system, which gradually increases the odds of receiving higher-tier memories following a string of failed attempts. There is also an influence for the four-star in *draw* function where every 10 pulls guarantee a four-star memory. Despite these variations, the overall distribution reflects the expected behavior of the implemented gacha system, confirming its alignment with typical gacha game mechanics.

Furthermore, the simulation data was represented through a bar graph that illustrated the frequency of each tier. The graph highlights the dominance of Three-star memories, consistent with the defined probabilities, and the increased occurrence of Four-star memories compared to the base probability, likely due to natural random variation. While the frequency of Five-star memories slightly exceeds the base probability, it aligns with expectations given the limited sample size and the influence of the soft pity system. These results validate the functionality of the implemented gacha system, demonstrating its ability to replicate key mechanics of a real-world gacha experience with fairness and accuracy.

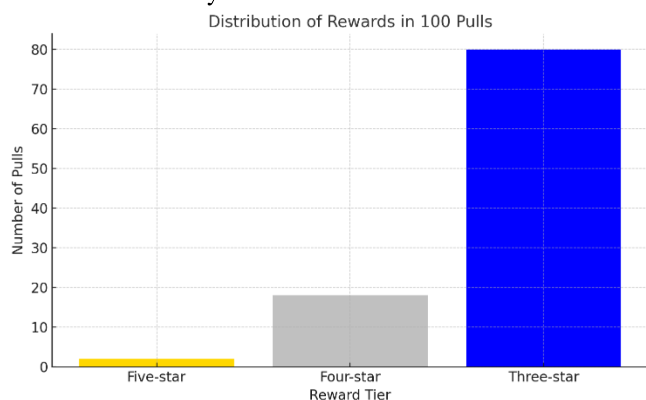

Fig. 4.5 Bar Graph (Source: Author).

V. Conclusion

This study has explored the integration of gacha systems in "Love and Deepspace," focusing on the significance of number theory in determining their functionality. By utilizing mathematical concepts such as modular arithmetic and probabilities, the implementation illustrates how these ideas support fair and engaging game mechanics. The integration of gacha mechanics in otome games like "Love and Deepspace" reveals how statistical randomness can enhance narrative-driven gameplay, creating immersive player experiences. Features such as soft and hard pity systems emphasize the importance of balancing randomness with predictability to sustain player trust and enjoyment. This paper ultimately highlights the critical role of mathematics in the design and refinement of gacha systems, encouraging further exploration into their ethical, social, and economic implications as these mechanics continue to evolve within the global gaming industry.

VI. Appendix

The GitHub repository for source code can be accessed at https://github.com/BerthaSoliany/makalah-matdis. There is also a video explanation and demonstration that can be accessed at https://youtu.be/OJpbWR1kIGs.

VII. Acknowledgment

The author would like to express gratitude to God for providing strength and clarity, to the game development community for their shared knowledge, and to Mr. Rila Mandala for his guidance in Linear Algebra and Geometry. The author is also grateful to InFold Pte. Ltd., the developers of Love and DeepSpace, for creating a groundbreaking game that inspired this study. Lastly, the author extends gratitude to the family for the support and encouragement throughout this study.

References

[1] R. Munir, *Teori Bilangan*. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/Teori%20Bilangan.pdf. [Accessed: Jan. 7, 2025]

[2] N. S. R. Wahyuni, *Teori Bilangan*. [Online]. Available: https://eprints.hamzanwadi.ac.id/5712/1/18.%20Buku_Teori%20Bilangan.pdf. [Accessed: Jan. 7, 2025].

[3] GameRefinery, “The complete guide to mobile game gachas in 2022,” [Online]. Available: https://www.gamerefinery.com/the-complete-guide-to-mobile-game-gachas-in-2022/. [Accessed: Jan. 7, 2025].

[4] M. Grguric, “Gacha system in mobile games,” [Online]. Available: https://www.blog.udonis.co/mobile-marketing/mobile-games/gacha-system. [Accessed: Jan. 7, 2025].

[5] Otome Kitten, “Infold Games x Otomate: Interview on the next-gen otome game Love and Deepspace,” [Online]. Available: https://otomekitten.com/2024/01/21/infold-games-x-otomate-interview-on-the-next-gen-otome-game-love-and-deepspace/. [Accessed: Jan. 7, 2025].

[6] LDPlayer, “Love and Deepspace game review: Gacha rates, battle guide, otome dating sim,” [Online]. Available: https://www.ldplayer.net/blog/love-and-deepspace-game-review-gacha-rates-battle-guide-otome-datingsim.html. [Accessed: Jan. 7, 2025].

[7] Love and Deepspace Wiki, “Love and Deepspace Wiki,” [Online]. Available: https://loveanddeepspace.fandom.com/wiki/Love_and_Deepspace_Wiki. [Accessed: Jan. 7, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 7 Januari 2025

Bertha Soliany Frandi