Turn-Based Batch Scheduling in ‘Reincarnation Journey: Fantasy Fate’ using Priority Queues

Mahatma Brahmana - 13524015
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
mahatma.brahmana1@gmail.com , 13524015@std.stei.itb.ac.id

***Abstract*— This paper presents a discrete time turn-based scheduling model for the game Reincarnation Journey: Fantasy Fate, utilizing a priority queue implemented with a min heap. Each character's turn is dynamically determined based on their speed, which influences their action time. The simulation incorporates game mechanics such as buffs, debuffs, and speed modifiers that adjust scheduling in real time. By applying discrete event simulation principles and efficient heap operations, the system ensures fair, responsive, and scalable turn management. The proposed model demonstrates how mathematical structures like functions, recurrence relations, and trees, specifically heaps, can be applied to build practical and computationally efficient systems in interactive simulations.**

Keywords*— *Reincarnation Journey: Fantasy Fate; turn-based scheduling; priority queue; min heap; simulation modeling

I. Introduction

In the realm of interactive digital entertainment, turn-based game systems have long been valued for their strategic depth and clarity. Unlike real time systems that rely on continuous input and reaction, turn-based systems process player and enemy actions in discrete intervals, allowing thoughtful decision making and controlled pacing. However, implementing an efficient and fair turn order mechanism becomes increasingly complex when multiple entities with varying attributes, such as speed, delay, and temporary effects, are involved.

Reincarnation Journey: Fantasy Fate is a turn-based fantasy battle simulation where characters take turns to act according to their speed and various in game modifiers. To manage turn order dynamically and fairly, the simulation employs a priority queue, implemented using a min heap, to schedule each character’s next action based on a time value called `t_next`. This approach ensures that characters with higher speed (and thus lower `t_next` values) act sooner, while also allowing real time updates based on buffs, debuffs, and other temporary effects.

This paper explores the theoretical foundation, algorithmic implementation, and practical advantages of using a heap based priority queue in discrete time simulation. It applies fundamental concepts from discrete mathematics, such as functions, recurrence relations, and tree structures. to design a system that is both computationally efficient and faithful to gameplay mechanics. The simulation is intended not only as a game logic prototype, but also as a demonstration of how abstract mathematical structures can be effectively applied in real world programming contexts.

II. Theoretical Foundation

A. *Reincarnation Journey: Fantasy Fate*

Fig. 1. Screen Lobby *Reincarnation Journey: Fantasy Fate.*

The simulation developed in Reincarnation Journey: Fantasy Fate serves as a case study in how fundamental concepts from discrete mathematics can be applied to a turn-based scheduling system. This fantasy themed battle simulator models a combat scenario where multiple characters, each with unique attributes like speed and status effects, must take turns in a fair and efficient order. To achieve this, the system utilizes several core principles from discrete mathematics, including functions, recurrence relations, tree structures (especially binary heaps), and algorithmic complexity. These concepts form the backbone of the simulation’s turn scheduler, which ensures that gameplay remains responsive, balanced, and scalable.

Fig. 2. Battle in *Reincarnation Journey: Fantasy Fate.*

Look at Fig. 2 and you will see twelve fighters in the arena, six lined up on the left and six on the right, clearly marked as the players crew and the foes squad. Each fighter gets a turn set by her speed stat, and that stat assigns a `t_next` value the exact moment she can move again. The fighter with the lowest `t_next`, who acts next, sits at the bottom right, just left of the skill icons, while the one with the highest `t_next`, who will go last this cycle, hangs on the far left of the action list. This layout mirrors the internal timetable, because the engine uses a min heap to order moves by their `t_next` numbers and display them in the same tidy row you see on screen.

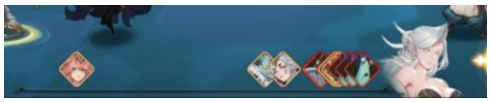

Fig. 3. Action Queue, the further to the right the smaller the `t_next` value.

B. *Function and Relation*

In discrete mathematics, a function is a relation that maps each element from a domain to one or zero element in the codomain.

For example:

A = {Hasan, Tanti, Rommi, Yusuf, Aditya}

This domain represent a person, and

B = {Toyota, Daihatsu, Mercedes, BMW}

This domain represent a car.

Let R be a relation that represents a person and the car he drives.

R = {(Hasan, Daihatsu), (Rommi, Toyota), (Yusuf, Mercedes), (Aditya, Toyota)}

This means that Hasan drives a Daihatsu, Rommi drives a Toyota, Yusuf drives a Mercedes, and Aditya drives a Toyota. Tanti does not drive any car. The BMW car is not driven by anyone in the relationship.

In the context of this project, the simulation uses a function to calculate how soon a character can act again based on their speed. For example:

$$f(pid) = \frac{BASE\_DELAY}{speed[pid]}$$

Fig. 4. `t_next` Function {f(pid)}.

This function assigns each player ID `(pid)` a value `t_next`, which determines the time until their next action.

A relation in discrete mathematics is a set of ordered pairs that shows how elements from one set are related to elements from another. For example, the turn order in the simulation can be seen as a relation between player ID and scheduled time.

C. *Recurrence Relation*

Called as a recursive, is a recurrence relation defines a sequence where each term is a function of its predecessors.

Recursive function divided by 2 part:

1) *Base*
   This part contains the explicitly defined function value. This part also stop the recursion (and gives the defined value to the recursive function)

2) *Requence*
   This section defines the function in its own terms. Also contains rules for finding the value of a function at one input from its values at smaller inputs.

For Example: f is divided as recursively as follow

$$f(n) = \begin{cases} 3 & ,n = 0 \quad basis \\ 2\,f(n-1) + 4 & ,n > 0 \quad rekurens \end{cases}$$

Fig. 5. Recursive Example. *source :* Rekursi dan Relasi Rekurens

determine f(4) value!

The solution is :

$$\begin{aligned} f(4) &= 2f(3) + 4 \\ &= 2(2f(2) + 4) + 4 \\ &= 2(2(2f(1) + 4) + 4) + 4 \\ &= 2(2(2(2f(0) + 4) + 4) + 4) + 4 \\ &= 2(2(2(2 \cdot 3 + 4) + 4) + 4) + 4 \\ &= 2(2(2(10) + 4) + 4) + 4 \\ &= 2(2(24) + 4) + 4 \\ &= 2(52) + 4 \\ &= 108 \end{aligned}$$

Fig. 6. Recursive Solution. *source :* Rekursi dan Relasi Rekurens

In this scheduling simulations, time evolution can be described using recurrence. For example, if t_0 is the initial time for a player, then their next action time t_1 might be determined by:

$$t_{n+1} = t_n + f(pid)$$

Fig. 7. Recursive Pattern in This Scheduling Simulation.

This recursive pattern is essential for modeling ongoing processes in a turn-based system.

D. *Trees in Discrete Structure*

A trees in Discrete Structure is a connected, acyclic graph in which any two nodes are connected by exactly one path. Trees are useful for representing data that has a hierarchical structure, such as file systems, organizational charts, and decision processes.

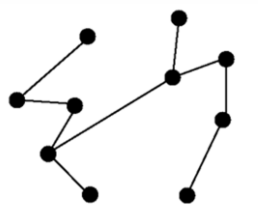

Fig. 8. Tree Picture Example. *source :* Rekursi dan Relasi Rekurens

One special form of a tree is the binary tree, a tree in which each node has at most two children. These children are typically referred to as the left child and right child. Binary trees are the foundational structure for various efficient algorithms and data structures such as binary search trees, heaps, and expression trees.

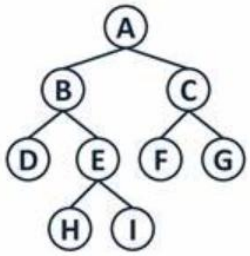

Fig. 9. Binary Tree Picture Example. *source :* Rekursi dan Relasi Rekurens

From the Fig. 9., we can see that the root of the tree is node A. Node A has two children: B (left child) and C (right child). Node B has children D (left child) and E (right child), while C has children F and G. And node E, in turn, has two children: H (left child) and I (right child). This tree is a complete binary tree up to level 2 (every node has two children where possible), and illustrates a clear hierarchy from top to bottom.

1) Heap as a Binary Tree

A heap is a special kind of binary tree that satisfies the heap property. In a min heap, each parent node has a value less than or equal to its children. The smallest element is always at the root. This structure supports efficient retrieval and deletion of the minimum (or maximum) element.

In our simulation, we use a min heap to maintain and sort player actions based on their next available time `(t_next)`. This ensures that the player who should act first is always at the top of the priority queue.

In this simulation, each heap entry has the form `(t_next, pid, speed)`. Here, `t_next` is the time when the player is scheduled to act, and it determines the priority. Python's `heapq` module uses an array to represent this binary heap, maintaining the heap property such that the smallest `t_next` is always accessible at index 0.

This is directly related to the tree structure: although implemented via array, the logical relationship between parent and children (at indices i, 2i+1, 2i+2) forms a conceptual binary tree.

A heap is chosen in this simulation primarily because of its computational efficiency and practicality in implementation. The most significant advantage of using a heap is that both insertion and deletion operations can be performed in O(log n) time. This is particularly valuable in a turn-based scheduling system, where players' actions are frequently updated and reordered based on timing.

In addition to its time efficiency, a heap is also memory efficient. It can be represented as a simple array or list rather than as a full fledged tree structure. In this flat representation, the relationships between parent and child nodes are easily determined using index calculations, making the structure not only efficient but also simple to implement.

By using a min heap to manage the action order based on `t_next` values (the time at which a player can act next), the simulation maintains the performance. The player with the lowest `t_next` is always at the top of the heap, allowing the system to determine the next action. This design choice ensures that the system remains fast, fair, and scalable, even as the number of characters or turns increases, making it ideal for real time turn-based simulations.

E. Priority Queue

A priority queue is an abstract data type where each element is associated with a priority, and elements are served based on their priority order. In a min priority queue, elements with the lowest priority value are accessed first.

While a heap provides the underlying structure, the priority queue ensuring that character actions are scheduled and processed in the correct order based on urgency, in this case, their `t_next` values. This behavior is crucial in simulations where time sensitive decisions (like turn order) must be resolved efficiently and fairly.

By implementing the priority queue via a min heap, the system benefits from a execution order based on parameters such as `t_next`, as well as dynamic adjustability where insertion, removal, and updating of elements can be performed efficiently. Furthermore, this structure is in line with the principles of discrete event simulation (DES), in which actions are driven by events rather than progressing in a continuous timeline. Thus, in this simulation, the priority queue functions not merely as a data structure, but as a scheduling policy that is effectively and efficiently realized through the use of a heap.

F. Algorithmic Complexity

Algorithmic complexity basically captures how much time and memory an algorithm needs to finish when you feed it data of a certain size. By looking at these numbers, we can tell if an approach will slow down or crash when the input grows, and we can line up several methods side by side to pick the best one. Time complexity, one popular measure, tracks how running time itself stretches as the input grows, and we usually write that using Big O symbols. So, a routine tagged O(n) gets one additional step for every extra item you add, whereas one marked O(log n) hardly budges, making it much quicker on huge lists. Knowing these labels helps programmers guess how an idea scales and choose the least greedy option before the data pile up kicks in.

G. *Discrete Scheduling and Simulation Modeling*

The nature of turn-based games is inherently discrete. The system does not rely on continuous timelines but instead processes one event at a time. This aligns with discrete event simulation (DES), a technique where time advances in steps based on the occurrence of events rather than in fixed intervals. In our simulation, time is represented using floating point milliseconds, but actual "progress" only occurs when a player's action is processed.

The simulation thus mimics discrete time evolution, with the scheduler determining the next actor, applying buffs or debuffs, adjusting their speed or delay, and rescheduling the queue accordingly.

III. IMPLEMENTATION

In this section, the writer presents a concrete implementation of the proposed turn-based batch scheduling model using Python. The goal is to simulate how multiple characters in *Reincarnation Journey: Fantasy Fate* take turns based on speed, action point buffs or debuffs, and timed speed modifiers. The core of the implementation is a priority queue (min heap), which ensures the next character to act is always the one with the smallest upcoming turn time (`t_next`). This mirrors established practices in game development, such as those described in software like *Rogue Basin's* turn scheduler.

A. *Priority Queue*

In this scheduling model, each character's upcoming action is determined by a value called `t_next`, which represents the virtual time at which the character is next allowed to act. This value is dynamically computed based on the character's current speed and a constant called `BASE_DELAY`.

`BASE_DELAY` is a predefined fixed delay constant used as the baseline reference for turn progression. It simulates the standard time interval required before a character can act again, assuming a neutral or average speed.

The character's speed represents how quickly they can perform actions relative to others. A higher speed means the character will be able to act more frequently. To calculate when a character should next take a turn, we divide `BASE_DELAY` by the character's speed:

```
t_next = BASE_DELAY / speed
```

This inverse relationship ensures that faster characters (with higher speed values) generate smaller `t_next` values, causing them to act sooner in the simulation cycle. By organizing all characters in a min heap (priority queue) based on their `t_next`, the system always selects the character with the lowest `t_next`, the one whose next action is due the earliest, to take the next turn.

```python
def init_queue():
    pq = []
    for pid, sp in speeds.items():
        heapq.heappush(pq, (BASE_DELAY / sp, pid, sp))
    return pq

pq = init_queue()
```

Fig. 10. Initialize Priority Queue Based on Character Speed

By invoking `heapq.heappush()`, these tuples are inserted into a binary min heap, a complete binary tree structure where the parent node's value is always less or equal to its children's values. This value represent `t_next`. This structure is crucial because it guarantees that `pq[0]`, the heap's root, always contains the character scheduled to act next based on the smallest `t_next`.

B. *Executing a Turn: Priority Extraction*

```python
t_min, active_pid, active_sp = heapq.heappop(pq)
```

Fig. 11. Extracting the Next Active Player from the Priority Queue

During each simulation turn, `heapq.heappop()` removes the tuple with the smallest `t_next` from the heap and returns it. This operation both identifies the next character to act and rebalances the heap in "O(log n)" time through a process knows as *bubble down*, where the tree structure is restored after removing the root. This ensures that extracting the next turn remains efficient even as multiple turns are processed.

C. *Resetting and Requeuing Turn Times*

```python
temp = []
for t, pid, sp in pq:
    new_t = t - t_min
    temp.append((new_t, pid, sp))
temp.append((BASE_DELAY / speeds[active_pid], active_pid, speeds[active_pid]))
pq = temp
heapq.heapify(pq)
```

Fig. 12. Recalculating and Rebuilding the Turn Queue After a Characterr's Action

In this block, we normalize the event schedule so that after the active character acts, their remaining time is reset to zero and other characters times are adjusted relative to this references. First, we iterate over the existing heap `pq`, which holds `(t_next, character_id, speed)` tuples, and subtract the smallest elapsed time `t_min` (from active character's turn) from every other character's `t_next`. This effectively shift the timeline, ensuring that the active turn reset the time baseline for the next cycle.

Next, we insert the active character back into the queue with a newly computed `t_next`, derived from their current speed: `BASE_DELAY/speed[active_pid]`. This models the character's subsequent turn interval after completing an action. The combination of relative time adjustment and reinsertion ensures that each turn's scheduling remains accurate and avoids cumulative inflation of time values.

Finally, we rebuild the min heap by calling `heapq.heapify(pq)`, which reorganized the updated list into a valid heap in 0(n) time. This rebuild is essential for maintaining correct priority ordering after multiple adjustment to `t_next`.

D. *Action Point Buffs and Debuffs*

Within *Reincarnation Journey: Fantasy Fate,* characters can receive temporary buffs (boosts) or debuffs (penalties) that influence how soon they can act again, these are represented by Action Point (AP). In our implementation (simulate), these effects are triggered at the start of a turn when the turn index matches a predefined entry in `AP_BUFF_schedule`. Each entry consists of `(character_id, delta_b)`, where `delta_b` is a positive value for a buff (accelerating the turn) or a negative value for a debuff (delaying the turn).

Rather than always rebuilding the entire heap, the code optimizes performance by first checking whether the affected character is at the root of the min heap `(pq[0])`. If they are, the code employs `heapq.heapreplace()`, which atomically removes the smallest element (root) and inserts the updated tuple in its place. This operation executes in "O(log n)" time since it adjusts only a single path in the tree

If the affected character is not at the root, a simple loop is used to locate their entry in the heap. Once found, their `t_next` is updated in place using `max(t - delta_b, 0)` to avoid negative values, and the heap is rebalanced with `heapq.heapify()`. While this takes "O(n)" time to rebuild, it remains more efficient than reconstructing the entire heap from scratch for every update .

```
if turn_idx in AP_BUFF_schedule:
    pid_b, delta_b = AP_BUFF_schedule[turn_idx]
    t0, pid0, sp0 = pq[0]
    if pid0 == pid_b:
        new_entry = (max(t0 - delta_b, 0), pid0, sp0)
        heapq.heapreplace(pq, new_entry)
    else:
        for i, (t, pid, sp) in enumerate(pq):
            if pid == pid_b:
                new_t = max(t - delta_b, 0)
                pq[i] = (new_t, pid, sp)
                heapq.heapify(pq)
                break
```

Fig. 13. Adjusting Action Point Through AP Buffs and Debuffs

Mechanically, this design faithfully mirrors how AP effects work in the actual game: a positive AP buff grants character an immediate "rush", an earlier turn within the same cycle, while a negative AP effect simulates conditions such as being stunned or slowed, pushing the character's turn into further into the future. This dynamic adjustment ensures that each AP buff or debuff has an instant and tangible impact on the turn order, just like in gameplay.

This design also balances correctness with performance efficiency. It ensures that an AP buff or debuff takes effect immediately, just as it does in actual gameplay, without causing unnecessary computation. Using `heapreplace()` when the affected character is next to act is both efficient and succinct, while the fallback minimizes disruption for nonroot changes. This approach ensures that the scheduler remains responsive and accurate, even when multiple AP changes occur during a simulation.

From a performance perspective, scanning the queue to find the affected character takes "O(n)" time (with n representing the number of characters), while re-`heapifying` requires "O(n)" time as well making each application of an AP effect an "O(n)" operation in the worst case. Although this is efficient for a small number of characters, this cost becomes more significant with larger queues. Regardless, it reliably enforces the instantaneous timing effects central to the gameplay experience. Fortunately, *Reincarnation Journey: Fantasy Fate* only have 6 characters most in each fight.

By integrating this inplace update and heap rebalancing, the system cleanly efficiently enforces both beneficial and detrimental AP modifiers. This approach preserves overall turn order fairness while accurately reflecting the intended game mechanics of *Reincarnation Journey: Fantasy Fate.*

E. *Speed Buffs and Debuffs with Timed Duration*

In Reincarnation Journey: Fantasy Fate, characters may receive temporary Speed Up or Speed Down effects that alter their movement or action rate over a fixed number of turns. Our implementation handles these effects using two main mechanisms: adjusting the character's speed value and automatically expiring the effect after its duration elapses.

1. Applying Speed Modifier

When the simulation reaches a turn index listed in `SPEED_BUFF_schedule`, it extracts `(character_id, delta_s, duration)` from the schedule. A positive `delta_s` value increases the character's speed (Speed Up), while a negative value slows them down (Speed Down). The algorithm then updates the character's base speed and appends the active effect to `active_speed_buffs`.

```
if turn_idx in SPEED_BUFF_schedule:
    pid_s, delta_s, dur = SPEED_BUFF_schedule[turn_idx]
    active_speed_buffs[pid_s].append([dur, delta_s])
    speeds[pid_s] += delta_s
    print(f"[Speed Buff] Turn {turn_idx}: Player {pid_s} speed changes by {delta_s:+.1f} for {dur} turns")
    pq = init_queue()
```

Fig. 14. Applying and Registering Timed Speed Buffs in the Turn Scheduler

Because every character's `t_next` time depends on their current speed `(BASE_DELAY / speed)`, it becomes necessary to rebuild the entire priority queue `(pq = init_queue())` whenever a speed change occurs. This ensures all entries correctly reflect the updated speeds. Rebuilding the heap via `heapify()` takes "O(n)" time.

2. Expiring Buffs and Restoring Speed

After each turn, the system checks `active_speed_buffs` to decrement the remaining duration of any active speed modifiers. If a character's duration reaches zero after their turn, the effect is removed and their speed is restored to its prior value.

```
for pid in list(active_speed_buffs):
    new_list = []
    for dur, delta in active_speed_buffs[pid]:
        if pid == active_pid:
            dur -= 1
        if dur > 0:
            new_list.append([dur, delta])
        else:
            speeds[pid] -= delta
            print(f"[Buff Expired] Player {pid}: speed delta {delta:+.1f} removed")
    active_speed_buffs[pid] = new_list
```

Fig. 15. Managing Expiration of Temporary Speed Buffs

This ensures the buff or debuff remains in effect exactly for the intended number of turns, gracefully reversing once expired. Each update is handled in “O(b)” time per turn, where b is the number of active buffs (typically small).

Overall, the application of speed effects and the subsequent rebuild of the priority queue demonstrate both algorithmic correctness and performance efficiency, as they balance the need to keep the turn order accurate with the constraints of a small scale simulation. This mechanism faithfully captures multiturn speed alterations while maintaining computational quality in alignment with discrete mathematics and game simulation principles.

IV. RESULT

A. Simulation Overview

```python
import heapq
import sys
from collections import Counter, defaultdict
import functools

print = functools.partial(print, flush=True)

# KONFIGURASI
BASE_DELAY = 1000.0
EPS = sys.float_info.epsilon * 100
N_TURNS = 100

AP_BUFF_schedule = {5: (3, +3), 15: (8, +2)}
SPEED_BUFF_schedule = {10: (2, +30, 3), 25: (7, -40, 2)}

# Speed domain sesuai data
speeds = {
    1: 123, 2: 167, 3: 157, 4: 138,
    5: 123, 6: 127, 7: 132, 8: 139,
    9: 147, 10: 146, 11: 129, 12: 130
}

active_speed_buffs = defaultdict(list)
turn_count = Counter()

def almost_equal(a, b):
    return abs(a - b) <= max(abs(a), abs(b)) * EPS

def init_queue():
    pq = []
    for pid, sp in speeds.items():
        heapq.heappush(pq, (BASE_DELAY / sp, pid, sp))
    return pq

pq = init_queue()

print("Interactive Scheduler (Enter to run each turn)\n")

for turn_idx in range(1, N_TURNS + 1):
    print(f"\n--- Upcoming Turn {turn_idx} (before pressing Enter) ---")
    upcoming = sorted(pq, key=lambda x: x[0])
    for t, pid, sp in upcoming:
        print(f"Character {pid:2}: t_next = {t:.2f} ms | speed = {sp}")
    input("Press Enter to execute this turn...")

    # Attack Point Buff/Debuff
    if turn_idx in AP_BUFF_schedule:
        pid_b, delta_b = AP_BUFF_schedule[turn_idx]
        t0, pid0, sp0 = pq[0]
        if pid0 == pid_b:
            new_entry = (max(t0 - delta_b, 0), pid0, sp0)
            heapq.heapreplace(pq, new_entry)
            print(f"[AP Buff] Turn {turn_idx}: Character {pid_b} gets AP delta {delta_b:+} ms (new t_next = {new_entry[0]:.2f} ms)")
        else:
            for i, (t, pid, sp) in enumerate(pq):
                if pid == pid_b:
                    new_t = max(t - delta_b, 0)
                    print(f"[AP Buff] Turn {turn_idx}: Character {pid_b} gets AP delta {delta_b:+} ms (old t_next = {t:.2f} ms → new t_next = {new_t:.2f} ms)")
                    pq[i] = (new_t, pid, sp)
                    heapq.heapify(pq)
                    break

    # Speed Buff/Debuff
    if turn_idx in SPEED_BUFF_schedule:
        pid_s, delta_s, dur = SPEED_BUFF_schedule[turn_idx]
        active_speed_buffs[pid_s].append([dur, delta_s])
        speeds[pid_s] += delta_s
        print(f"[Speed Buff] Turn {turn_idx}: Character {pid_s} speed changes by {delta_s:+.1f} for {dur} turns")
        for i, (t, pid, sp) in enumerate(pq):
            if pid == pid_s:
                pq[i] = (t, pid, speeds[pid_s])
                break

    # Execute turn
    t_min, active_pid, active_sp = heapq.heappop(pq)
    print(f"\n>>> TURN {turn_idx}: Character {active_pid} acts now at t = 0.00 ms | speed = {active_sp}")
    turn_count[active_pid] += 1

    # Expire speed buffs
    for pid in list(active_speed_buffs):
        new_list = []
        for dur, delta in active_speed_buffs[pid]:
            if pid == active_pid:
                dur -= 1
            if dur > 0:
                new_list.append([dur, delta])
            else:
                speeds[pid] -= delta
                print(f"[Buff Expired] Character {pid}: speed delta {delta:+.1f} removed")
        active_speed_buffs[pid] = new_list

    # Reset times and requeue
    temp = []
    for t, pid, sp in pq:
        new_t = t - t_min
        temp.append((new_t, pid, sp))
    temp.append((BASE_DELAY / speeds[active_pid], active_pid, speeds[active_pid]))
    pq = temp
    heapq.heapify(pq)

print("\n=== SUMMARY ===")
for pid in sorted(speeds):
    print(f"Character {pid:2}: total turns = {turn_count[pid]}")
```

Fig. 16. Simulation of Turn-Based Scheduling System in *Reincarnation Journey: Fantasy Fate*

In this simulation, a turn-based action scheduling system is implemented using a priority queue to determine the order in which characters take turns. The simulation models 12 characters, each with a predefined base speed. At the beginning of the simulation, the initial turn time for each character is calculated by dividing a fixed base delay (1000 milliseconds) by their speed, producing a `t_next` value that indicates how soon each character can act. These values are stored in a min heap priority queue, ensuring that the character with the smallest `t_next` is always selected to act next.

Over 50 simulation turns, the system dynamically adjusts each character’s turn timing based on scheduled action point (AP) buffs or debuffs and speed modifications. AP effects directly shift a character's `t_next` value forward or backward, simulating abilities like haste or stun. These changes are applied either with `heapreplace` for efficient root updates or `heapify` for internal heap restructuring. Meanwhile, speed buffs or debuffs temporarily modify a character’s speed value for a given number of turns. Unlike AP effects, these do not immediately change the character’s current position in the queue; instead, they influence the delay before their next turn, after their current turn is executed.

Each time a character acts, their elapsed time (`t_min`) is subtracted from every other character's `t_next`, effectively resetting the relative timeline. The active character is then reinserted into the queue with a new turn time based on their updated speed. Temporary speed effects are tracked in a dictionary and expire after a certain number of turns, at which point the character’s speed is reverted, and this event is logged.

The simulation prints the scheduled characters before each turn, shows which character acts, and provides detailed logs whenever a buff or debuff is applied or expires. At the end, a summary reports how many times each character acted, illustrating the impact of speed and timing on turn frequency. The behavior of the scheduler confirms the correctness of the min heap implementation and the responsiveness of the system to time based modifications.

B. Simulation Output

To validate the effectiveness of the proposed turn-based batch scheduling model, a simulation was conducted using a Python implementation. This simulation aims to demonstrate how character turns are dynamically managed based on their speed, applied buffs or debuffs, and the priority scheduling mechanism. In this chapter, we present the output of the simulation over multiple turns and analyze how the system behaves under different conditions, including the application of speed and action point (AP) modifiers.

1) Start Games

```
--- Upcoming Turn 1 (before pressing Enter) ---
Character  2: t_next = 5.99 ms | speed = 167
Character  3: t_next = 6.37 ms | speed = 157
Character  9: t_next = 6.80 ms | speed = 147
Character 10: t_next = 6.85 ms | speed = 146
Character  8: t_next = 7.19 ms | speed = 139
Character  4: t_next = 7.25 ms | speed = 138
Character  7: t_next = 7.58 ms | speed = 132
Character 12: t_next = 7.69 ms | speed = 130
Character 11: t_next = 7.75 ms | speed = 129
Character  6: t_next = 7.87 ms | speed = 127
Character  1: t_next = 8.13 ms | speed = 123
Character  5: t_next = 8.13 ms | speed = 123
Press Enter to execute this turn...
```

Fig. 17. Start Game Output

At the start of the program, all characters are displayed along with their corresponding turn time (`t_next`) and speed.

The list is sorted in ascending order of t_next, from the earliest turn at the top to the latest at the bottom. This list shows which character will act next and in what order. Each entry includes the character's ID, their current t_next value (indicating how soon they will act), and their current speed.

2) Next Turns

```
>>> TURN 1: Character 2 acts now at t = 0.00 ms | speed = 167

--- Upcoming Turn 2 (before pressing Enter) ---
Character  3: t_next = 0.38 ms | speed = 157
Character  9: t_next = 0.81 ms | speed = 147
Character 10: t_next = 0.86 ms | speed = 146
Character  8: t_next = 1.21 ms | speed = 139
Character  4: t_next = 1.26 ms | speed = 138
Character  7: t_next = 1.59 ms | speed = 132
Character 12: t_next = 1.70 ms | speed = 130
Character 11: t_next = 1.76 ms | speed = 129
Character  6: t_next = 1.89 ms | speed = 127
Character  1: t_next = 2.14 ms | speed = 123
Character  5: t_next = 2.14 ms | speed = 123
Character  2: t_next = 5.99 ms | speed = 167
Press Enter to execute this turn...
```

Fig. 18. Next Turn Output

The next action will occur when the user presses Enter, executing the turn of the character at the top of the list. After each character takes their turn, the program displays the upcoming turn schedule.For example, after character 2 finishes their action at t = 0.00 ms with a speed of 167, the program outputs a sorted list of the next characters based on their upcoming turn times

3) Attack Point (AP) buffs

```
[AP Buff] Turn 5: Character 3 gets AP delta +3 ms (old t_next = 5.89 ms → new t_next = 2.89 ms)

>>> TURN 5: Character 8 acts now at t = 0.00 ms | speed = 139

--- Upcoming Turn 6 (before pressing Enter) ---
Character  4: t_next = 0.05 ms | speed = 138
Character  7: t_next = 0.38 ms | speed = 132
Character 12: t_next = 0.50 ms | speed = 130
Character 11: t_next = 0.56 ms | speed = 129
Character  6: t_next = 0.68 ms | speed = 127
Character  1: t_next = 0.94 ms | speed = 123
Character  5: t_next = 0.94 ms | speed = 123
Character  3: t_next = 2.54 ms | speed = 157
Character  2: t_next = 4.78 ms | speed = 167
Character  9: t_next = 6.41 ms | speed = 147
Character 10: t_next = 6.50 ms | speed = 146
Character  8: t_next = 7.19 ms | speed = 139
Press Enter to execute this turn...
```

Fig. 19. Attack Point (AP) Buffs Output

When an Attack Point (AP) buff or debuff is triggered, the t_next value of a specific character is directly modified, simulating an immediate acceleration or delay in their turn.

For instance, on Turn 5, Character 3 receives an AP buff of +3 ms, which significantly reduces their t_next from 5.89 ms to 2.89 ms. Despite this buff, Character 8 remains the one with the smallest t_next value and is therefore selected to act during this turn.

Despite this buff, Character 8 remains the one with the smallest t_next value and is therefore selected to act during this turn:

4) Speed buffs

```
[Speed Buff] Turn 10: Character 2 speed changes by +30.0 for 3 turns

>>> TURN 10: Character 11 acts now at t = 0.00 ms | speed = 129

--- Upcoming Turn 11 (before pressing Enter) ---
Character  6: t_next = 0.12 ms | speed = 127
Character  1: t_next = 0.38 ms | speed = 123
Character  5: t_next = 0.38 ms | speed = 123
Character  2: t_next = 4.22 ms | speed = 197
Character  3: t_next = 5.47 ms | speed = 157
Character  9: t_next = 5.85 ms | speed = 147
Character 10: t_next = 5.95 ms | speed = 146
Character  8: t_next = 6.64 ms | speed = 139
Character  4: t_next = 6.74 ms | speed = 138
Character  7: t_next = 7.40 ms | speed = 132
Character 12: t_next = 7.63 ms | speed = 130
Character 11: t_next = 7.75 ms | speed = 129
Press Enter to execute this turn...
```

Fig. 20. Speed Buffs Output

When a speed buff or debuff is triggered, the speed of a specific character is directly modified, simulating an adding or subtracting the speed of that character.

For example, On Turn 10, a Speed Buff is applied to Character 2, increasing their speed by 30 for a duration of 3 turns.

This results in Character 2's speed rising from 167 to 197, which will influence how soon they act in future turns. However, the application of this speed buff does not alter their current t_next value immediately, preserving the fairness of the turn order.

After the turn is executed, the simulation updates the queue. Now, although Character 2's speed has been increased, their t_next remains at 4.10 ms, placing them closer to the front of the queue than before. We can expect that in subsequent turns, Character 2 will take actions more frequently due to their higher speed, and this effect will last for the next three turns, after which the buff will expire and their speed will return to normal.

5) Summary

```
>>> TURN 100: Character 6 acts now at t = 0.00 ms | speed = 127

=== SUMMARY ===
Character  1: total turns = 7
Character  2: total turns = 10
Character  3: total turns = 10
Character  4: total turns = 8
Character  5: total turns = 7
Character  6: total turns = 8
Character  7: total turns = 7
Character  8: total turns = 9
Character  9: total turns = 9
Character 10: total turns = 9
Character 11: total turns = 8
Character 12: total turns = 8
```

Fig. 21. Summary Output

After executing 100 turns in the simulation, the system prints a final summary displaying the total number of turns taken by each character throughout the battle timeline. This output reflects how often each character was able to act, which is primarily influenced by their base speed and any temporary buffs or debuffs applied during the simulation.

For example, Character 2 and Character 3, both with relatively high base speeds and occasional speed enhancements, ended up acting the most, each with 10 total

turns. In contrast, characters with lower speeds, such as Character 1 and Character 5, only managed to act 7 times.

C. Complexity Analysis

The simulation was executed for 100 turns involving 12 characters, similar to *Reincarnation Journey: Fantasy Fate,* each with varying speed values. The turn scheduler successfully maintained accurate ordering of character actions, including the application and expiration of action point (AP) buffs and speed buffs/debuffs with fixed durations.

At the end of the simulation, a turn count was computed for each character, showing how speed and AP modifiers influenced the number of actions per character. The results demonstrate the scheduler's ability to prioritize high speed characters while dynamically adjusting for temporary effects.

In terms of computational complexity, the program uses a min heap to manage turn order, where each insertion or extraction operation (`heapq.heappush`, `heapq.heappop`) takes O(log n) time, with n being the number of characters.

The main loop runs for T turns, so the base complexity is O(T log n). Occasional AP buffs may trigger `heapify`, which has a worst case time of O(n). Speed buffs also involve a bounded scan of the heap (O(n)) and minor list updates.

Therefore, the overall complexity of the simulation is O(T log n + Bn), where B is the number of buff/debuff events. Given that both n and B are small in this simulation, the implementation is efficient for real time or turn-based game scenarios.

D. Conclusion

This paper describes a turn-based scheduling engine added to the game Reincarnation Journey: Fantasy Fate, relying on a priority queue with a min heap design so that character orders shift according to live speed numbers and timing changes. By drawing on core ideas from discrete math functions, recurrences, binary trees, and big-O analysis the scheduler handles each action in a way that feels fair, runs fast, and scales up as more units enter battle.

The system tracks `t_next` values to pinpoint the exact virtual moment when a hero can act again, and it rewrites the timetable on the fly whenever buffs, debuffs, or action point costs move that clock. Because the underlying heap allows quick insert, extract, and update operations, those real time adjustments take little CPU time even in long fights.

Overall, the simulation imitates a discrete event system in that game time jumps forward only when a scheduled action occurs, not at fixed ticks. This design keeps the engine light yet nimble, faithfully reproducing tense combat situations where the order of moves and the fleeting impact of tactics can change the outcome.

In short, the priority queue technique shown here connects classroom discrete mathematics to living gameplay, proving that theory can bear concrete, player facing results. The finished scheduler is fast, responsive, and mathematically robust, giving players richer strategic choices and deeper satisfaction during every battle.

APPENDIX

The full source code of the navigation compass puzzle solver can be found on this github repository:

https://github.com/llennmaha/DiscreteMath

ACKNOWLEDGMENT

The author first gives all credit to God, whose endless grace, wisdom, and quiet guidance made both the simulations and this paper possible. The author is also genuinely thankful to the lecturers of the IF1220 course, especially Dr. Ir. Rinaldi Munir, M.T.; their enthusiastic teaching and careful reading material have sparked his interest and shaped his studies. Finally, the author offers warm gratitude to his family, whose steady encouragement, prayers, and everyday support have carried him through each step of the academic journey.

REFERENCES

[1] R. Munir, "Pohon (Bag. 1)," Homepage Rinaldi Munir, 2025. Availlable: 23-Pohon-Bag1-2024 [Accessed: June 19th 2025]

[2] R. Munir, "Relasi dan Fungsi Bagian 1," Homepage Rinaldi Munir, 2025. Availlable: 05-Relasi-dan-Fungsi-Bagian1-(2024) [Accessed: June 19th 2025]

[3] R. Munir, "Rekursi dan Relasi Rekuens (Bagian 1)," Homepage Rinaldi Munir, 2025. Availlable: Rekursi dan Relasi Rekurens [Accessed: June 19th 2025]

[4] Chizaruu, "A priority queue based turn scheduling system," Github. Available: A priority queue based turn scheduling system | RogueBasin [Accessed: June 18th 2025]

[5] "Priority Queue using Binary Heap," GeeksforGeeks, March 28th 2025. Available: Priority Queue using Binary Heap - GeeksforGeeks [Accessed: June 18th 2025]

[6] David Ang, "Turn Based Battle System using Phaser," programmingmind, January 23th 2018. Available: Turn Based Battle System using Phaser · by David Ang [Accessed: June 18th 2025]

[7] "Journey Renewed Fate fantasy," myth-inmedia.fandom. Available: Journey Renewed Fate fantasy | Myths in Media Wiki | Fandom[Accessed: June 18th 2025]

[8] R.Munir, "Kompleksitas Algoritma Bagian1," Homepage Rinaldi Munir, 2025. Available: 25-Kompleksitas Algoritma Bagian 1 - 2024[Accessed: June 19th 2025]

STATEMENT

I hereby declare that this paper that I wrote is my own work, not an adaptation or translation of someone else's work, and not plagiarized.

Bandung, 20 Juni 2025

Mahatma Brahmana (13524015)