PRODUCE OF QIAN YI

DIFFICULTY: SEC 4 END-OF-YEAR

# MOCK COMPUTING

## DURATION: 2 HOURS

## TOTAL MARKS: 55 MARKS

MINIMUM A1: 41 MARKS

THIS PAPER CONTAINS OF **2 TASKS.**

YOU WILL BE TESTED, IN SOME ORDER, ON FILE OPERATIONS, SEARCH AND SORT ALGORITHMS, OBJECT-ORIENTED PROGRAMMING, DATABASES WITH SQLITE3, CSV POPULATION AND FLASK WEB APP DEVELOPMENT.

| Please **download** the following items in the Github before starting: | |
|---|---|
| exam.ipynb | Fill this up for all tasks<br>(You need to create your own files for Task 2.4) |
| data_files/messy.csv | Task 1.3 |
| data_files/expected_output_1_3.txt | Expected Output for Task 1.3 (for correctness) |
| data_files/players.csv | Task 2.2 |
| data_files/games.csv | Task 2.2 |
| data_files/entries.csv | Task 2.2 |

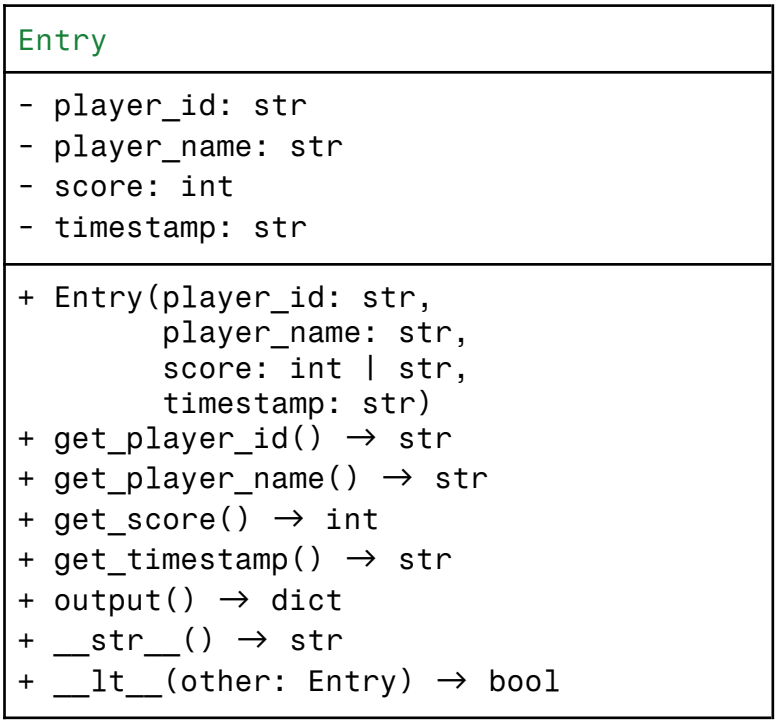
After completion of this paper, the **recommended answer key** is attached in the `/QUIZ_ANSWERS` folder for your reference. Good luck.

# Task 1

Your City runs the greatest arcade in town, Tombzone, which has implemented a system of tracking entries from its many happy users. However, the City spent too much money buying arcade machines, and they are now bankrupt from hiring HR/tech specialists. You have been hired (for free?) to implement a Python programme to sort out their mess.

## Task 1.1

To store each game entry in Python, implement the `Entry` class according to the UML diagram. **[8]**



| Attributes/Methods | Description |
|---|---|
| - player_id: str<br>- player_name: str<br>- score: int<br>- timestamp: str | Private attributes to store the player's ID, name, score, and the time in which the entry is logged (which is formatted YYYY-MM-DD HH:MM:SS) |
| + Entry(player_id: str, player_name: str, score: int \| str, timestamp: str) | An initialiser method that sets the private attributes. score can be an integer or a string, so you must convert it into an integer. If that fails, **default** it to **0**. |
| + get_player_id() → str<br>+ get_player_name() → str<br>+ get_score() → int<br>+ get_timestamp() → str | Getter methods for the private attributes |
| + output() → dict | Returns a dictionary of attributes such that {"player_id":..., "player_name":..., "score":..., "timestamp":...} |
| + __str__() → str | Returns the string containing all private attributes by the format: "NAME (ID) - SCORE @ TIMESTAMP" |
| + __lt__(other: Entry) → bool | A special method to check if the current Entry object (self) is less than another Entry object (other).<br><br>Compare by seeing if the self's score is lower. If they are equivalent, compare by timestamp to see if it is earlier |

---

## Task 1.2** (updated)

Implement merge_sort(entries: list[Entry]) to sort a list of Entries by **Merge Sort**, such that it is in **descending** order. **[3]**

---

## Task 1.3

Tombzone tried to digitise their own archived entries list. However, the scanner is dysfunctional and some of the data appears to be formatted incorrectly (likely due to the cashier's poor handwriting?). Thankfully, most of the critical data is preserved.

You are given `messy.csv`, of which a **snippet** is as follows:

```
player_id,player_name,score,timestamp
S-2024-001,lee wei,1,200,2025/09/01 14:03
S-2024-002,ALICE TAN,980,2025-09-01 13:59:30
,Unknown,500,2025-09-01 12:00
S-2024-003,bob lim,"1,050","01-09-2025 11:05"
```

You are to **clean** the entries as follows: **[6]**

1. Remove any entries (rows) that has an **empty** `player_id` or `player_name`
2. All players' names are to be **titlecased** (Hint: use `s1 = s2.title()` where s1 and s2 are strings)
3. Convert all `scores` to **integers**. If they cannot be converted into integers, remove that entry.
4. Normalise `timestamp` to the exact formatting of **`YYYY-MM-DD HH:MM:SS`**. Follow the conversion constraints below:
    - If `timestamp` is missing seconds, you can safely add `:00`.
    - If there is a `YYYY/MM/DD` formatting with slashes, convert it into `YYYY-MM-DD`
5. Convert all entries into `Entry` objects, then conduct a **Selection Sort** of the Entries in descending order.

Then, produce `clean.txt` such that it includes all string implementations of the **sorted** entries (Hint: use `__str__`)

At the top of the text file, you must include this, where `X` and `Y` are appropriate **count** numbers, and `Z` represents the average score from all entries, **rounded** to the nearest integer: **[2]**

```
Entries processed: X
Entries removed: Y
Average rounded Score: Z
```

You can check your output against `expected_output_1_3.txt`.

# Task 2

It's been a month, and Tombzone now has terabytes of text files to parse through for past audits. Its new employees are no longer as tech-savvy. The City, having once again spent too much money on arcade machines, asks (by ask, I mean demand) you to create a User Interface, integrated with a SQL database, for its employees to readily track and add new entries.

## Task 2.1

In **Python**, generate `tombzone.db` and the tables: `Player`, `Game`, `Entry`. The CREATE statements are to include checks, non-nulls, defaults, types and key references where appropriate. **[6]**

1. The **Player** table is as follows:
    - `player_id` – unique primary key text that is formatted `S-YYYY-XXX`, where XXX is a unique 3-digit number (you may assume all inputs are valid)
    - `name` – the player's name, which may not be unique
    - `gender` – a text, to be stored as a single character, between "M" or "F"
    - `hp` – the player's handphone number

2. The **Game** table is as follows:
    - `game_id` – an autoincremental primary key representing a unique game
    - `game_name` – the Tombzone arcade game's name. No two games share the same name.
    - `max_score` - the maximum score attainable in the game, which defaults to 1000

3. The **Entry** table is as follows:
    - `entry_id` - an autoincremental primary key representing a unique entry
    - `player_id` - references a Player's ID
    - `game_id` - references a Game's ID
    - `score` - an integer of the score attained, which must either be 0 or positive.
    - `timestamp` - a timestamp of the entry formatted as `YYYY-MM-DD HH:MM:SS`.

## Task 2.2

Your HR team has converted the text files into comma-separated values (CSV) files. They want you to populate the tables in `tombzone.db` with files `players.csv`, `games.csv` and `entries.csv`.

Each file starts with a header of the respective columns, then the rows of values. However, due to HR technology defects (what did you expect anyway), there are **duplicate** entries in each file, which you will need to ignore. **[7]**

(Hint: To ignore duplicates, use `INSERT OR IGNORE INTO`)

---

## Task 2.3

A Tombzone manager wants to investigate (for whatever reason) whether gender affects a player's performance in a particular game. Implement `investigate(game_name: str, gender: str)`, where given `game_name` and `gender`, the function queries the database and returns all corresponding players' `names`, `handphone numbers`, `timestamp` and *Score Percentage*.

*Score Percentage* is defined as the score as a fraction of the game's maximum score in decimal form to 2 decimal places.

Order the table by their Score Percentage in **descending** order, then by the `timestamp` starting from the **earliest**. **[5]**

*A snippet output for the first 2 entries of female players in "Clash Royole" is as follows:*

| name | hp | timestamp | score_percentage |
|---|---|---|---|
| Zoey | 88385346 | 2025-09-03 16:25:00 | 0.3 |
| Ella O'Brien | 58488556 | 2025-09-01 16:05:00 | 0.15 |
| ... | | | |

(Hint: To **divide X by Y** in SQL, you can do `(X*1.0)/Y` to cast X as a floating point to be divisible by Y. To **round** numbers in SQL, use `ROUND(number, decimal_places)`)

---

## Task 2.4

You are now tasked with developing a Flask Application for the new employees. This Task shall be subdivided into Task 2.4A and Task 2.4B to implement **both** /search and /add routes respectively, but your submission can be parked under the same set of files.

## Task 2.4A

On /search route, implement a **search form** with a dropdown menu of all game names, queried dynamically, in **ascending** order.

Upon searching, the App should query the database and display all players' names, handphone numbers, gender, entry timestamps and *Score Percentage* under that game name, ordered by Score Percentage in descending order.

**[10]**

# Results for Basketball

| Name | Handphone Number | Gender | Entry Timestamp | Score Percentage |
|---|---|---|---|---|
| Nuna | 24562753 | F | 2025-09-02 14:10:00 | 0.19 |
| Sonia Lim | 68987796 | M | 2025-09-02 09:15:00 | 0.17 |
| Zoey | 88385346 | F | 2025-09-03 14:20:00 | 0.13 |
| Zoey | 88385346 | F | 2025-09-03 14:17:00 | 0.08 |
| Felton | 97826788 | M | 2025-09-04 09:15:00 | 0.07 |
| Nuna | 24562753 | F | 2025-09-03 14:50:00 | 0.07 |
| Ah Seng | 79863444 | M | 2025-09-03 12:35:00 | 0.06 |
| Sam Tan | 81266686 | M | 2025-09-03 16:30:00 | 0.05 |
| Sonia Lim | 68987796 | M | 2025-09-02 15:10:00 | 0.02 |

### Task 2.4B [8]

On /add route, implement a **form** with player_id and game_name as two separate **dropdown** menus queried from the database, both in **ascending order**. Add score as an **integer** field.

Upon submission, you must **validate** that the score is not a negative integer. Otherwise, output an **error message** for the User.

Then **add** the data to the database based on the player's inputs. Use the SQL function CURRENT_TIMESTAMP for the timestamp field. Finally, return a **success** message that shows the entry's fields of player's name, gender, game name, score and entry timestamp.

**[8]**