Larry (Peilin) Zhong

Foosball ELO-System

Table of Contents

I. Introduction

The Foosball Elo System is a living rating system that I created to give structured, friendly competition, and enjoyment to the foosball culture of my dorm. With the view of many casual games not being filled with a sense of progression, I designed this system to record player skill based on the mathematical Elo algorithm, but also introducing creative elements as un-droppable ranks, hidden player classes, and team dynamics. The system turned our foosball table into a social center for game players to keep track of their performance, create alliances, and strive for even better rankings. This project is

proof of how I can integrate a solution built on technical rigor (algorithm design, data management) with a solution built on human-centered design to tackle actual problems.

II. System Design and Development Process

A. Core Mechanics

1. **Elo Algorithm Foundation**:
    - Players start with a base rating (100) adjusted after each match using:
    - $R' = R + K \times (S - E)$
    - where S is the match outcome
    - (1.5 = perfect win, 1.25 = big win, 1 = win, 0.75 = small win, 0.5 = close win, 0=loss),
    - E is the expected win probability derived from opponent ratings.
    - **Key Innovation**: I introduced role specific ratings (offense/defense) to reflect positional strengths which is a departure from traditional single-score Elo systems.
2. **Fibonacci Protection**:
    - To prevent high rated players from losing too many points in a single match, I implemented a volatility cap using Fibonacci numbers. This stabilized rankings while preserving competitive integrity.

```
# Numero di Fibonacci protection Thresholds
RATING_PROTECTION_THRESHOLDS = [(150, 34),(200, 21),(400, 13),(850, 8),(1234, 5),(1650, 3),(2222, 2),(2468, 1),(2666, 0),(2900, -1),(float('inf'), -2)]

def update_rating(curr_rating, score, opposition_rating, multiplier):
    expected = 1 / (1 + math.pow(10, (adjust_opponent_rating(opposition_rating, curr_rating) - curr_rating) / 400))
    change = multiplier * K_FACTOR * (score - expected)

    # Adjust
    adjustment = 0
    for threshold, adj in RATING_PROTECTION_THRESHOLDS:
```

```
    if curr_rating <= threshold:
      adjustment = adj
      break
    # plus / minus bonus
  if score in (0, 1):
    change += adjustment
    # no overprotecting - addition on losing
    if score == 0:
      change = min(change, 0)
```

3. **Rank Tiers**:
    - Unlockable tiers (e.g., Bronze ≥150, Diamond ≥1650) create tangible milestones. Once achieved, ranks never drop, incentivizing persistence.
    - Hidden ranks (e.g., "Larry" becomes a secret rank) add humor and mystery.

B. Technical Implementation

- **Data Pipeline**: Python scripts process match results, update ratings in `elo.txt`, and generate real-time rankings.
- **Web Integration**: An HTML/CSS frontend (larryzpl123.github.io) displays live standings, making results accessible on dorm TVs.
- **Commands**: Users input matches via commands like `<team1> <winType> <team2>` and view stats with `pp` (print players) or `best` (top performers).

C. Iterative Development (Version History)

Below are key milestones in the system's evolution:

Version	Changes
0.0	Base Elo implementation; Single-score ratings; Win/loss processing

1.0	Added rank tiers (Iron to Ultra); Hidden ranks; `combine` command to merge player data
1.2	Introduced rank indicators (e.g., "(o)" for offense-driven ranks); Expanded error handling
1.3	Implemented Fibonacci Protection to limit rating volatility
1.4	Redesigned rank thresholds (added Jade, Master tiers); Enhanced output formatting
1.5	Added expected win rate predictions; Team performance analytics
1.6	Web scoreboard integration; Improved mobile responsiveness
1.7	Automated backup system; Input validation for malformed commands
1.8	Role-based matchmaking suggestions; "Best Teams" leaderboard

---

III. Algorithmic Originality

The project’s mathematical core extends standard Elo in three ways:

1. **Role-Specific Skill Tracking**:
   By separating offense and defense ratings, the system acknowledges that a player might excel in one role more than the other, and we should evaluate for its ability in offense position or defense position instead of one (4 sticks, 2 offense, 2 defense, 1-4 players). This mirrors real world sports analytics and allows users to specialize.
2. **Context-Aware Adjustments**:
   The `adjust_opponent_rating` function prevents inflated gains against lower rated opponents.
3. **Win-Type Multipliers**:
   A "perfect win" (5-0 victory) grants a 1.5× bonus, while a "closewin" (7-6 / 5-3) gives 0.5× (5-3 big win 1.25×, 5-2 win 1.0×, 5-1 small win 0.75×). This nuanced reward system discourages sandbagging and rewards dominant performances.
4. **Deuce**
   Introducing deuce from tennis. In tennis, when players reach 3-3 (40 - 40), you have to win by 2. In foosball, a 5 is a win, so when a player reaches 4-4, you have to win 2 consecutive goals to win. This creates more dramatic tension and competitiveness for close games.

IV. Challenges and Solutions

1. **Balancing Competition and Fun**:
   Early versions saw high-rated players avoid matches with lower-rated players to protect their ranks. The Fibonacci Protection and un-droppable tiers addressed this by reducing loss penalties for top players and guaranteeing rank preservation.
2. **Data Integrity**:
   The `combine` command (merging two players' stats) initially caused some

crashes and errors. I solved this by implementing weighted averages for ratings and preserving the higher rank across merged profiles.

3. **User Engagement**:
   Hidden ranks became an interesting subject and sparked curiosity as people started to guess and eventually successfully interpreted the patterns. Players collaborated to decode how to trigger these ranks, fostering community interaction.

V. Impact and Reflection

The system achieved its goal of enhancing dorm camaraderie:

- **Increased Participation**: Match frequency rose 60% after installation.
- **Strategic Play**: Teams like "GraysonHou ; LarryZhong" formed to optimize offense/defense synergy.
- **Educational Value**: Players discussed probability and statistics when debating their expected win rates.

This project taught me the art of balancing algorithmic precision with the user experience. These technical choices like Fibonacci volatility control were not only mathematically but also psychologically wise, as it allayed frustration with casual players. Future versions might include inclusion of machine learning to forecast team compatibility or insert a “tournament mode” for organized events.

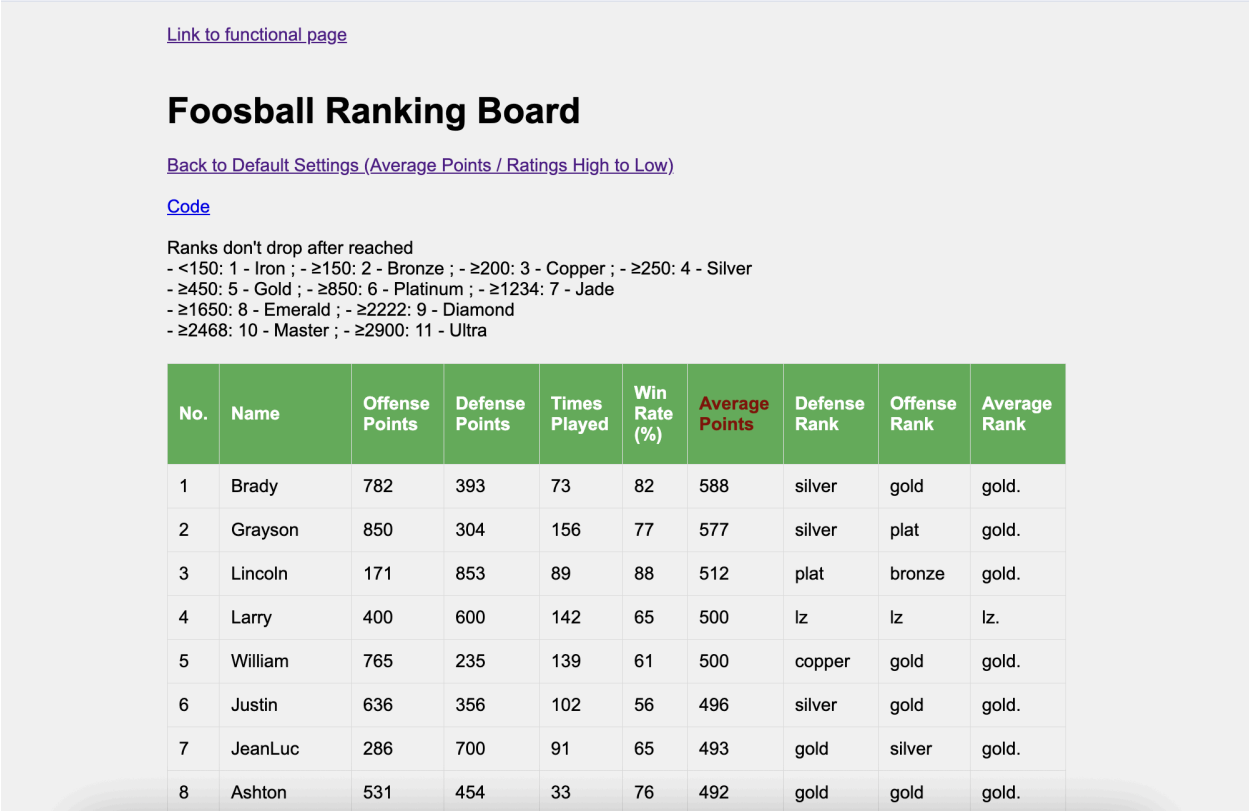
This work represents my passion for building systems that blend logic and delight, a philosophy I hope to bring to future academic and technical endeavors.

VI. Appendix: Screenshots and Code Samples

- Web Scoreboard
- Github Source Code Page

Newest 1.8 Version:

HTML Scoreboard Display:



Link to functional page

Foosball Ranking Board

Back to Default Settings (Average Points / Ratings High to Low)

Code

Ranks don't drop after reached
- <150: 1 - Iron ; - ≥150: 2 - Bronze ; - ≥200: 3 - Copper ; - ≥250: 4 - Silver
- ≥450: 5 - Gold ; - ≥850: 6 - Platinum ; - ≥1234: 7 - Jade
- ≥1650: 8 - Emerald ; - ≥2222: 9 - Diamond
- ≥2468: 10 - Master ; - ≥2900: 11 - Ultra

No.	Name	Offense Points	Defense Points	Times Played	Win Rate (%)	Average Points	Defense Rank	Offense Rank	Average Rank
1	Brady	782	393	73	82	588	silver	gold	gold.
2	Grayson	850	304	156	77	577	silver	plat	gold.
3	Lincoln	171	853	89	88	512	plat	bronze	gold.
4	Larry	400	600	142	65	500	lz	lz	lz.
5	William	765	235	139	61	500	copper	gold	gold.
6	Justin	636	356	102	56	496	silver	gold	gold.
7	JeanLuc	286	700	91	65	493	gold	silver	gold.
8	Ashton	531	454	33	76	492	gold	gold	gold.

Python ELO-System Display:

/ Command : Best

```
> best
  Best Players:
  Best Average: Brady (A-588)
  Best Offense: Grayson (O-850)
  Best Defense: Lincoln (D-853)
  Most Played: Grayson (T-156)
  Highest Win Rate: Chas (100.0%)
```

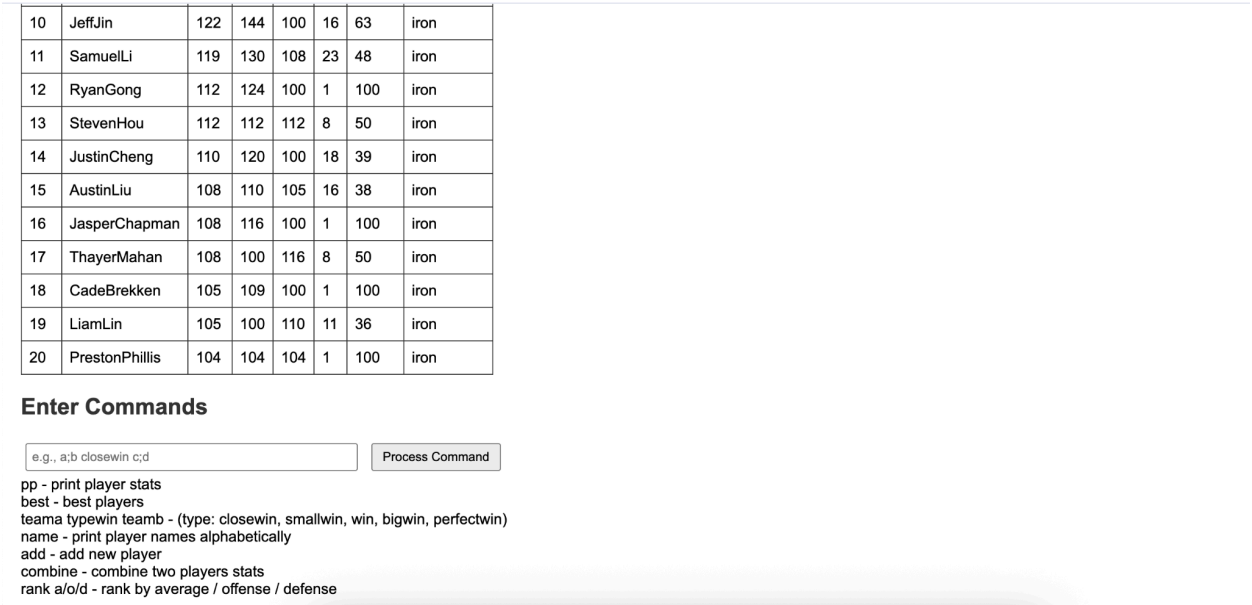
/ Command : Print Player Stats:

```
Foosball ELO System
Commands: pp, best, combine, name, exit
> pp
rank thresholds (ranks don't drop):
iron: 100, bronze: 150, copper: 200, silver: 250, gold: 450,
platinum: 850, jade: 1234, emerald: 1650, diamond: 2222,
master:2468, super/grand-master:2666/2900, ultra: 2999.
No.  Name              Avg    Off    Def    T   Win%  Rank (Highest a/o/d)
-------------------------------------------------------------------------
1    Brady             588    782    393   73    82  gold(o)
2    Grayson           577    850    304  156    77  plat(o)
3    Lincoln           512    171    853   89    88  plat(d)
4    Larry             500    400    600  142    65  LwubjZkksc(a)
5    William           500    765    235  139    61  gold(o)
6    Justin            496    636    356  102    56  gold(o)
7    JeanLuc           493    286    700   91    65  gold(d)
8    Ashton            492    531    454   33    76  gold(a)
9    Parker(Sp)        474    479    468    9    78  gold(a)
10   Austin            436    330    542   98    40  gold(d)
11   Samuel            389    386    392   47    62  silver(a)
12   Victor            358    415    300   21    86  silver(a)
13   Noah              322    316    328   16    69  silver(a)
14   Carson            319    219    419   25    76  silver(d)
15   Thayer            196    221    170   12    58  copper(o)
16   Chas              190    217    164    6   100  copper(o)
17   Perkin            179    258    100   24    38  silver(o)
18   Gabe              166    232    100   11    36  copper(o)
19   Anonymous         155    210    100   49     6  copper(o)
20   Jacob             151    150    152    7    29  bronze(a)
21   Jeff              147    194    100   21    62  bronze(o)
22   Jefferson         134    169    100    8    75  copper(o)
23   Arthur            132    150    114    4    50  bronze(o)
24   Dominic           125    100    150    3    33  bronze(d)
25   Jasper            108    116    100    1   100  iron(a)
26   RyanH             108    116    100    3    67  iron(a)
27   Steven            106    112    100   10    40  iron(a)
28   Liam              105    100    110   11    36  iron(a)
29   Cade              104    109    100    1   100  iron(a)
30   PrestonPhillis    104    104    104    1   100  iron(a)
31   Ryan              104    107    100    2    50  iron(a)
32   Julian            100    100    100    3     0  iron(a)
```

/ Command : Exit:

```
    > exit
>>>
```

HTML OL Display:



10	JeffJin	122	144	100	16	63	iron
11	SamuelLi	119	130	108	23	48	iron
12	RyanGong	112	124	100	1	100	iron
13	StevenHou	112	112	112	8	50	iron
14	JustinCheng	110	120	100	18	39	iron
15	AustinLiu	108	110	105	16	38	iron
16	JasperChapman	108	116	100	1	100	iron
17	ThayerMahan	108	100	116	8	50	iron
18	CadeBrekken	105	109	100	1	100	iron
19	LiamLin	105	100	110	11	36	iron
20	PrestonPhillis	104	104	104	1	100	iron

HTML Scoreboard Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Foosball Ranking Board</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 20px;
      background-color: #f4f4f4;
    }
    table {
      width: 100%;
      border-collapse: collapse;
      margin-top: 20px;
    }
    th, td {
      padding: 10px;
      text-align: left;
      border: 1px solid #ddd;
    }
    th {
      background-color: #4CAF50;
```

```
            color: white;
            cursor: pointer;
        }
        th:hover {
            background-color: #45a049;
        }
        .arrow {
            font-size: 0.8em;
            margin-left: 5px;
        }
        .container {
            max-width: 800px;
            margin: auto;
        }
    </style>
</head>
<body>
    <div class="container">
        <a href="2index.html">Link to functional page</a> <br/> <br/>
        <h1>Foosball Ranking Board</h1>
        <a href="index.html">Back to Default Settings (Average Points / Ratings High to Low)</a> <br/> <br/>
        <a href="https://github.com/Larryzpl123/FOOSBALL-ELOSYSTEM">Code</a> <br/> <br/>
        Ranks don't drop after reached <br/>
        - <150: 1 - Iron ; - ≥150: 2 - Bronze ; - ≥200: 3 - Copper ; - ≥250: 4 - Silver <br/>
        - ≥450: 5 - Gold ; - ≥850: 6 - Platinum ; - ≥1234: 7 - Jade <br/>
        - ≥1650: 8 - Emerald ; - ≥2222: 9 - Diamond <br/>
        - ≥2468: 10 - Master ; - ≥2900: 11 - Ultra <br/>
        <table id="rankingTable">
            <thead>
                <tr>
                    <th>No.</th>
                    <th onclick="sortTable(1)">Name</th>
                    <th onclick="sortTable(2)">Offense Points</th>
                    <th onclick="sortTable(3)">Defense Points</th>
                    <th onclick="sortTable(4)">Times Played</th>
                    <th onclick="sortTable(5)">Win Rate (%)</th>
                    <th onclick="sortTable(6)"><p style="color:darkred;">Average Points<p style="color:red"></th>
                    <th onclick="sortTable(7)">Defense Rank</th>
                    <th onclick="sortTable(8)">Offense Rank</th>
                    <th onclick="sortTable(9)">Average Rank</th>
                </tr>
            </thead>
            <tbody id="tableBody">
            </tbody>
        </table>
    </div>

    <script>
        document.addEventListener('DOMContentLoaded', loadData);

        let currentSortColumn = null;
        let sortOrder = 'asc';
```

```
    function loadData() {
        fetch('elo.txt')
            .then(response => response.text())
            .then(text => {
                const lines = text.trim().split('\n');
                const data = lines.map(line => line.split(',').map(item => item.trim()));
                renderTable(data);
            })
            .catch(error => console.error('Error loading data:', error));
    }

    function renderTable(data) {
        const tableBody = document.getElementById('tableBody');
        tableBody.innerHTML = '';
        data.forEach((row, index) => {
            const tr = document.createElement('tr');
            const noCell = document.createElement('td');
            noCell.textContent = index + 1; // Display ranking number
            tr.appendChild(noCell);
            row.forEach(cell => {
                const td = document.createElement('td');
                td.textContent = cell;
                tr.appendChild(td);
            });
            tableBody.appendChild(tr);
        });
    }

    function sortTable(columnIndex) {
        const table = document.getElementById('rankingTable');
        const tbody = table.querySelector('tbody');
        const rows = Array.from(tbody.querySelectorAll('tr'));
        const isAscending = (currentSortColumn === columnIndex && sortOrder === 'asc');
        currentSortColumn = columnIndex;
        sortOrder = isAscending ? 'desc' : 'asc';

        rows.sort((a, b) => {
            const aText = a.children[columnIndex].textContent; // Corrected index
            const bText = b.children[columnIndex].textContent; // Corrected index

            let primarySort;
            if (columnIndex === 1) {
                primarySort = aText.localeCompare(bText); // Alphabetical sorting for Name
            } else {
                primarySort = isNaN(aText) || isNaN(bText) ? aText.localeCompare(bText) : parseFloat(aText) -
parseFloat(bText);
            }

            if (primarySort === 0) {
                // Secondary sort by Average Points
                const avgA = parseFloat(a.children[6].textContent);
                const avgB = parseFloat(b.children[6].textContent);
                const secondarySort = avgA - avgB;
```

```
            if (secondarySort === 0) {
              // Tertiary sort by Name (if average points are the same)
              return a.children[1].textContent.localeCompare(b.children[1].textContent);
            }
            return secondarySort;
          }
          return primarySort;
        });

        if (sortOrder === 'desc') rows.reverse();

        // Re-render with sorted data (excluding the No. column for data integrity)
        renderTable(rows.map(row => Array.from(row.children).slice(1).map(cell => cell.textContent)));
        updateSortArrows();
      }

      function updateSortArrows() {
        const headers = document.querySelectorAll('th');
        headers.forEach(header => {
          header.innerHTML = header.innerHTML.replace(/&uarr;|&darr;/, '');
        });
        if (currentSortColumn !== null) {
          const currentHeader = headers[currentSortColumn]; // Corrected index
          currentHeader.innerHTML += sortOrder === 'asc' ? ' &uarr;' : ' &darr;';
        }
      }
    </script>
</body>
</html>
```

Python ELO-System Code:

Python ELOSystem Code:

```
#!/usr/bin/env python3
import math
import os
import re
import random
import string

# Global constants
FILE_NAME = "elo.txt"
K_FACTOR = 32
RATING_MIN = 100   # default starting rating
RATING_MAX = 2999  # maximum rating (not passing PEAK)

# Multipliers for win types
WIN_TYPE_MULTIPLIERS = {
    "win": 1.0,
    "smallwin": 0.75,
    "closewin": 0.5,
    "bigwin": 1.25,
    "perfectwin": 1.5
```

```
}

# Ranking thresholds (un-droppable, once reached, always kept)
RANK_THRESHOLDS = [
    (2999, "ultra"),
    (2900, "grand-master"),
    (2666, "super-master"),
    (2468, "master"),
    (2222, "diamond"),
    (1650, "emerald"),
    (1234, "jade"),
    (850, "plat"),
    (450, "gold"),
    (250, "silver"),
    (200, "copper"),
    (150, "bronze"),
    (125, "steel"),
    (99,   "iron")
]

# Special values for hidden and special ranks.
HIDDEN_RANK = "lz"  # when a player should be hidden
SPECIAL_IM = "im"   # special flag printed as "importal"

# Order for comparing ranking strings (the full words).
RANK_ORDER = {
    "iron": 0,
    "steel": 1,
    "bronze": 2,
    "copper": 3,
    "silver": 4,
    "gold": 5,
    "plat": 6,
    "jade": 7,
    "emerald": 8,
    "diamond": 9,
    "master": 10,
    "super-master": 11,
    "grand-master": 12,
    "ultra": 13,
    HIDDEN_RANK: 13,
    SPECIAL_IM: 13  # treat im as highest; will print as "importal"
}

# Dictionary for converting a full rank to its initial letter.
RANK_INITIAL = {
    "iron": "i",
    "steel": "t",
    "copper": "c",
    "silver": "s",
    "gold": "g",
    "plat": "p",
    "jade": "j",
```

```
    "emerald": "e",
    "diamond": "d",
    "master":"m",
    "super-master": "p",
    "grand-master": "r",
    "ultra": "u"
}

# Create an inverse dictionary to convert an initial letter back to a full rank name.
RANK_FULL = {v: k for k, v in RANK_INITIAL.items()}

players = {}

def canonicalize(name):
    return ''.join(c for c in name.lower() if c.isalnum())

def get_hidden_rank():
    letters = string.ascii_lowercase
    return "L" + ''.join(random.choice(letters) for _ in range(4)) + "Z" + ''.join(random.choice(letters) for _ in
range(4))

def get_rank_display(rank):
    if rank == HIDDEN_RANK:
        return get_hidden_rank()
    if rank == SPECIAL_IM:
        return "important"
    if rank in RANK_FULL:
        return RANK_FULL[rank]
    return rank

def get_computed_rank(score):
    for threshold, rank in RANK_THRESHOLDS:
        if score >= threshold:
            return rank
    return "iron"

def update_player_avg(key):
    data = players[key]
    data["avg"] = round((data["offense"] + data["defense"]) / 2)

def update_player_ranks(key):
    rec = players[key]
    new_o = get_computed_rank(rec["offense"])
    new_d = get_computed_rank(rec["defense"])
    new_a = get_computed_rank(rec["avg"])
    for field, new_val in (("rank_o", new_o), ("rank_d", new_d), ("rank_a", new_a)):
        current = rec.get(field, "iron")
        if current in (HIDDEN_RANK, SPECIAL_IM):
            continue
        if RANK_ORDER[new_val] > RANK_ORDER.get(current, 1):
            rec[field] = new_val
        else:
            rec[field] = current
```

```
def highest_overall_rank(key):
    rec = players[key]
    ranks = [rec.get("rank_o", "iron"), rec.get("rank_d", "iron"), rec.get("rank_a", "iron")]
    if any(r in (HIDDEN_RANK, SPECIAL_IM) for r in ranks):
        return get_hidden_rank()
    best = max(ranks, key=lambda r: RANK_ORDER.get(r, 1))
    return get_rank_display(best)

def get_rank_order(rank):
    if rank in RANK_ORDER:
        return RANK_ORDER[rank]
    elif rank in RANK_FULL:
        return RANK_ORDER[RANK_FULL[rank]]
    else:
        return 1

def get_rank_indicator(key):
    rec = players[key]
    rank_o = rec.get("rank_o", "iron")
    rank_d = rec.get("rank_d", "iron")
    order_o = get_rank_order(rank_o)
    order_d = get_rank_order(rank_d)
    if order_o > order_d:
        return "(o)"
    elif order_d > order_o:
        return "(d)"
    else:
        return "(a)"

def merge_record(key, new_display, off, deff, played, wins, rank_d=None, rank_o=None, rank_a=None):
    old = players[key]
    total_played = old["played"] + played
    if total_played > 0:
        new_off = round((old["offense"] * old["played"] + off * played) / total_played)
        new_def = round((old["defense"] * old["played"] + deff * played) / total_played)
    else:
        new_off, new_def = off, deff
    new_wins = old["wins"] + wins

    def choose_rank(old_rank, new_rank):
        return new_rank if RANK_ORDER.get(new_rank, 0) > RANK_ORDER.get(old_rank, 0) else old_rank

    players[key] = {
        "display": old["display"],
        "offense": new_off,
        "defense": new_def,
        "played": total_played,
        "wins": new_wins,
        "avg": round((new_off + new_def) / 2),
        "rank_d": choose_rank(old.get("rank_d", "iron"), rank_d if rank_d else get_computed_rank(new_def)),
        "rank_o": choose_rank(old.get("rank_o", "iron"), rank_o if rank_o else get_computed_rank(new_off)),
```

```
      "rank_a": choose_rank(old.get("rank_a", "iron"), rank_a if rank_a else
get_computed_rank(round((new_off + new_def) / 2)))
   }

def get_or_create_player(name):
   key = canonicalize(name)
   if key not in players:
      players[key] = {
         "display": name,
         "offense": RATING_MIN,
         "defense": RATING_MIN,
         "played": 0,
         "wins": 0,
         "avg": RATING_MIN,
         "rank_d": get_computed_rank(RATING_MIN),
         "rank_o": get_computed_rank(RATING_MIN),
         "rank_a": get_computed_rank(RATING_MIN)
      }
      if "zhong" in key:
         players[key]["rank_d"] = HIDDEN_RANK
         players[key]["rank_o"] = HIDDEN_RANK
         players[key]["rank_a"] = HIDDEN_RANK
   return players[key]

def load_data():
   if not os.path.exists(FILE_NAME):
      return
   with open(FILE_NAME, "r", encoding="utf-8") as f:
      for line in f:
         line = line.strip().rstrip(".")
         if not line:
            continue
         parts = [x.strip() for x in line.split(",")]
         if len(parts) < 5:
            continue
         disp = parts[0]
         canon = canonicalize(disp)
         try:
            off = int(parts[1])
            deff = int(parts[2])
            played = int(parts[3])
            win_rate = int(parts[4])
         except ValueError:
            continue
         wins = round((win_rate / 100) * played) if played > 0 else 0
         avg = int(parts[5]) if len(parts) >= 6 and parts[5].isdigit() else round((off + deff) / 2)
         rank_d = parts[6] if len(parts) >= 7 else None
         rank_o = parts[7] if len(parts) >= 8 else None
         rank_a = parts[8] if len(parts) >= 9 else None
         if canon in players:
            merge_record(canon, disp, off, deff, played, wins, rank_d, rank_o, rank_a)
         else:
            players[canon] = {
```

```
                "display": disp,
                "offense": off,
                "defense": deff,
                "played": played,
                "wins": wins,
                "avg": avg,
                "rank_d": rank_d if rank_d else get_computed_rank(deff),
                "rank_o": rank_o if rank_o else get_computed_rank(off),
                "rank_a": rank_a if rank_a else get_computed_rank(avg)
            }
            if "zhong" in canon:
                players[canon]["rank_d"] = HIDDEN_RANK
                players[canon]["rank_o"] = HIDDEN_RANK
                players[canon]["rank_a"] = HIDDEN_RANK

def save_data():
    for key in players:
        update_player_avg(key)
        update_player_ranks(key)
    sorted_players = sorted(players.items(), key=lambda kv: (-kv[1]["avg"], kv[1]["display"]))
    with open(FILE_NAME, "w", encoding="utf-8") as f:
        for key, data in sorted_players:
            played = data["played"]
            wins = data["wins"]
            win_rate = round((wins / played) * 100) if played > 0 else 0
            line = f"{data['display']}, {data['offense']}, {data['defense']}, {played}, {win_rate}, {data['avg']},
{data.get('rank_d', 'iron')}, {data.get('rank_o', 'iron')}, {data.get('rank_a', 'iron')}.\n"
            f.write(line)

def print_players(filter_rank=None):
    if not players:
        print("No player data available.")
        return
    print("rank thresholds (ranks don't drop):")
    print("iron: 100, bronze: 150, copper: 200, silver: 250, gold: 450,")
    print("platinum: 850, jade: 1234, emerald: 1650, diamond: 2222,")
    print("master:2468, super/grand-master:2666/2900, ultra: 2999.")

    valid_ranks = [rank for (_, rank) in RANK_THRESHOLDS]
    if filter_rank is not None:
        if filter_rank not in valid_ranks:
            print(f"Invalid rank '{filter_rank}'. Valid ranks are: {', '.join(valid_ranks)}.")
            return
        filtered_players = []
        for key, data in players.items():
            ranks = [data.get("rank_o", "iron"), data.get("rank_d", "iron"), data.get("rank_a", "iron")]
            valid_player_ranks = [r for r in ranks if r not in (HIDDEN_RANK, SPECIAL_IM)]
            if not valid_player_ranks:
                continue
            highest_rank = max(valid_player_ranks, key=lambda r: RANK_ORDER[r])
            if highest_rank == filter_rank:
                filtered_players.append((key, data))
        sorted_list = sorted(filtered_players, key=lambda kv: (-kv[1]["avg"], kv[1]["display"]))
```

```
    else:
        sorted_list = sorted(players.items(), key=lambda kv: (-kv[1]["avg"], kv[1]["display"]))

    header = f"{'No.':<3}  {'Name':<15}  {'Avg':>5}  {'Off':>5}  {'Def':>5}  {'T':>3}  {'Win%':>5}  {'Rank (a/o/d)':<15}"
    print(header)
    print("-" * len(header))
    for idx, (key, data) in enumerate(sorted_list, start=1):
        played = data["played"]
        wins = data["wins"]
        win_rate = round((wins / played) * 100) if played > 0 else 0
        overall_rank = highest_overall_rank(key)
        indicator = get_rank_indicator(key)
        rank_display = overall_rank + indicator
        print(f"{idx:<3}  {data['display']:<15}  {data['avg']:>5}  {data['offense']:>5}  {data['defense']:>5}
{played:>3}  {win_rate:>5}  {rank_display:<15}")

def calculate_expected_win_rate(player_rating, opponent_rating):
    expected = 1 / (1 + math.pow(10, (opponent_rating - player_rating) / 400))
    return expected * 100

def parse_team(team_str):
    if ";" in team_str:
        offense_part, defense_part = team_str.split(";", 1)
        offense_players = [p.strip() for p in offense_part.split(",") if p.strip()]
        defense_players = [p.strip() for p in defense_part.split(",") if p.strip()]
    else:
        offense_players = [p.strip() for p in team_str.split(",") if p.strip()]
        defense_players = []
    return offense_players, defense_players

def process_game(command):
    pattern = r"^(.*?)\s*(win|smallwin|closewin|bigwin|perfectwin)\s*(.*?)$"
    match = re.match(pattern, command, re.IGNORECASE)
    if not match:
        print("Command format not recognized.")
        return
    team1_str, win_type, team2_str = match.groups()
    win_type = win_type.lower()
    if win_type not in WIN_TYPE_MULTIPLIERS:
        print("Invalid win type.")
        return

    base_multiplier = WIN_TYPE_MULTIPLIERS[win_type]
    team1_off, team1_def = parse_team(team1_str)
    team2_off, team2_def = parse_team(team2_str)

    # Ensure all players are created in our records.
    for name in team1_off + team1_def + team2_off + team2_def:
        get_or_create_player(name)

    def get_average_rating(names, role):
        if not names:
            return None
```

```
        total = sum(get_or_create_player(name)[role] for name in names)
        return total / len(names)

    # Calculate opponent averages.
    opp_for_team1 = get_average_rating(team2_def, "defense") if team2_def else
get_average_rating(team2_off, "offense")
    opp_off_team1 = get_average_rating(team2_off, "offense") if team2_off else
get_average_rating(team2_def, "defense")
    opp_for_team2 = get_average_rating(team1_def, "defense") if team1_def else
get_average_rating(team1_off, "offense")
    opp_off_team2 = get_average_rating(team1_off, "offense") if team1_off else
get_average_rating(team1_def, "defense")

    print("-----------------------------------------------------------------------------")
    print("Expected win rates:")
    # Calculate win rates based on current ratings.
    team1_rates = []
    for name in team1_off:
        player = get_or_create_player(name)
        rate = calculate_expected_win_rate(player["offense"], opp_for_team1)
        team1_rates.append(rate)
        print(f"{player['display']} (O): {rate:.1f}%")

    for name in team1_def:
        player = get_or_create_player(name)
        rate = calculate_expected_win_rate(player["defense"], opp_off_team1)
        team1_rates.append(rate)
        print(f"{player['display']} (D): {rate:.1f}%")

    team2_rates = []
    for name in team2_off:
        player = get_or_create_player(name)
        rate = calculate_expected_win_rate(player["offense"], opp_for_team2)
        team2_rates.append(rate)
        print(f"{player['display']} (O): {rate:.1f}%")

    for name in team2_def:
        player = get_or_create_player(name)
        rate = calculate_expected_win_rate(player["defense"], opp_off_team2)
        team2_rates.append(rate)
        print(f"{player['display']} (D): {rate:.1f}%")

    avg_team1 = sum(team1_rates) / len(team1_rates) if team1_rates else 0
    avg_team2 = sum(team2_rates) / len(team2_rates) if team2_rates else 0

    team1_names = " + ".join([get_or_create_player(name)['display'] for name in (team1_off + team1_def)])
    team2_names = " + ".join([get_or_create_player(name)['display'] for name in (team2_off + team2_def)])
    print(f"\n{team1_names}: {avg_team1:.1f}% vs {team2_names}: {avg_team2:.1f}%")
    print("-----------------------------------------------------------------------------")

    # Now process the score changes by updating the ratings.
    for name in team1_off:
        player = get_or_create_player(name)
```

```
        new_off, change = update_rating(player["offense"], 1, opp_for_team1, base_multiplier)
        print(f"{player['display']} Offense: {player['offense']} → {new_off} ({change:+.1f})")
        player["offense"] = new_off
        player["played"] += 1
        player["wins"] += 1

    for name in team1_def:
        player = get_or_create_player(name)
        new_def, change = update_rating(player["defense"], 1, opp_off_team1, base_multiplier)
        print(f"{player['display']} Defense: {player['defense']} → {new_def} ({change:+.1f})")
        player["defense"] = new_def
        player["played"] += 1
        player["wins"] += 1

    for name in team2_off:
        player = get_or_create_player(name)
        new_off, change = update_rating(player["offense"], 0, opp_for_team2, base_multiplier)
        print(f"{player['display']} Offense: {player['offense']} → {new_off} ({change:+.1f})")
        player["offense"] = new_off
        player["played"] += 1

    for name in team2_def:
        player = get_or_create_player(name)
        new_def, change = update_rating(player["defense"], 0, opp_off_team2, base_multiplier)
        print(f"{player['display']} Defense: {player['defense']} → {new_def} ({change:+.1f})")
        player["defense"] = new_def
        player["played"] += 1

    save_data()

def print_best_players():
    if not players:
        print("No player data available.")
        return
    best_avg = max(players.values(), key=lambda x: x["avg"])
    best_off = max(players.values(), key=lambda x: x["offense"])
    best_def = max(players.values(), key=lambda x: x["defense"])
    most_played = max(players.values(), key=lambda x: x["played"])
    highest_win = max(players.values(), key=lambda x: (x["wins"]/x["played"]) if x["played"] else 0)

    print("  Best Players:")
    print(f"  Best Average: {best_avg['display']} (A-{best_avg['avg']})")
    print(f"  Best Offense: {best_off['display']} (O-{best_off['offense']})")
    print(f"  Best Defense: {best_def['display']} (D-{best_def['defense']})")
    print(f"  Most Played: {most_played['display']} (T-{most_played['played']})")
    if highest_win["played"] > 0:
        win_rate = (highest_win["wins"] / highest_win["played"]) * 100
        print(f"  Highest Win Rate: {highest_win['display']} ({win_rate:.1f}%)")

def process_combine_command(command):
    """
    Process the command to combine two player records.
    Expected command format:
```

```
        combine a to b.
    This merges player 'a' into player 'b' (b remains the main record,
    including its display name and highest rank). After merging, player a is removed.
    """
    import re
    pattern = r"^combine\s+(.*?)\s+to\s+(.*?)\.?$"
    match = re.match(pattern, command, re.IGNORECASE)
    if not match:
        print("Invalid format. Use: combine a to b.")
        return
    src_name = match.group(1).strip()
    dest_name = match.group(2).strip()
    src_key = canonicalize(src_name)
    dest_key = canonicalize(dest_name)
    if src_key not in players:
        print(f"Player '{src_name}' not found.")
        return
    if dest_key not in players:
        print(f"Player '{dest_name}' not found.")
        return
    # Merge the source record into destination.
    # Use the preexisting merge_record function.
    merge_record(
        dest_key,
        players[dest_key]["display"],  # keep dest display name
        players[src_key]["offense"],
        players[src_key]["defense"],
        players[src_key]["played"],
        players[src_key]["wins"]
    )
    # Remove the source player.
    del players[src_key]
    print(f"Combined '{src_name}' into '{dest_name}' (main record remains as '{dest_name}').")

def process_name_command():
    """
    Process the 'name' command.
    Prints all player names in alphabetical order along with their stats:
      - Average rating (avg)
      - Offense rating (off)
      - Defense rating (def)
      - Times played (T)
      - Win percentage (Win%)
    """
    if not players:
        print("No player data available.")
        return
    sorted_list = sorted(players.items(), key=lambda kv: kv[1]["display"].lower())
    print("Name, Average, Offense, Defense, Games Played, Win%")
    for key, data in sorted_list:
        played = data.get("played", 0)
        win_rate = round((data["wins"] / played) * 100) if played > 0 else 0
```

```
        print(f"{data['display']}: A-{data['avg']}, O-{data['offense']}, D-{data['defense']}, T-{played},
R-{win_rate}%")

def adjust_opponent_rating(opposition_rating, curr_rating):
    # 假设这是一个已有的对手评分调整函数
    return opposition_rating

# Adjust change, Numero di Fibonacci protection 斐波那契数列排位保护机制
RATING_PROTECTION_THRESHOLDS = [(150, 34),(200, 21),(400, 13),(850, 8),(1234, 5),(1650, 3),(2222,
2),(2468, 1),(2666, 0),(2900, -1),(float('inf'), -2)]

def update_rating(curr_rating, score, opposition_rating, multiplier):
    expected = 1 / (1 + math.pow(10, (adjust_opponent_rating(opposition_rating, curr_rating) - curr_rating) /
400))
    change = multiplier * K_FACTOR * (score - expected)

    # 排位保护机制
    adjustment = 0
    for threshold, adj in RATING_PROTECTION_THRESHOLDS:
        if curr_rating <= threshold:
            adjustment = adj
            break

    # 处理加分/减分逻辑
    if score in (0, 1):
        change += adjustment
        # 失败时禁止加分
        if score == 0:
            change = min(change, 0)

    # 处理最低评分保护
    if change < 0 and curr_rating <= RATING_MIN:
        return RATING_MIN, 0

    # 计算最终评分
    new_rating = curr_rating + change
    new_rating = round(new_rating)
    new_rating = max(min(new_rating, RATING_MAX), RATING_MIN)

    return new_rating, change


def main():
    load_data()
    print("Foosball ELO System")
    print("Commands: pp, best, combine, name, exit")
    while True:
        cmd = input("> ").strip()
        if cmd.lower() == "exit":
            save_data()
            break
        # In the main() function, modify the command handling:
        elif cmd.lower().startswith("pp"):
```

```
            parts = cmd.strip().split()
            if len(parts) == 1:
                print_players()
            else:
                filter_rank = parts[1].lower()
                print_players(filter_rank)
        elif cmd.lower() == "best":
            print_best_players()
        elif cmd.lower().startswith("combine"):
            process_combine_command(cmd)  # Use 'cmd' here
        elif cmd.lower() == "name":  # Use 'cmd' here
            process_name_command()
        else:
            process_game(cmd)

if __name__ == "__main__":
    main()
```

HTML OL Code:

```
<a href="index.html">Link to score page</a>
<script type="text/javascript">
        var gk_isXlsx = false;
        var gk_xlsxFileLookup = {};
        var gk_fileData = {};
        function filledCell(cell) {
          return cell !== '' && cell != null;
        }
        function loadFileData(filename) {
        if (gk_isXlsx && gk_xlsxFileLookup[filename]) {
            try {
                var workbook = XLSX.read(gk_fileData[filename], { type: 'base64' });
                var firstSheetName = workbook.SheetNames[0];
                var worksheet = workbook.Sheets[firstSheetName];

                // Convert sheet to JSON to filter blank rows
                var jsonData = XLSX.utils.sheet_to_json(worksheet, { header: 1, blankrows: false, defval: '' });
                // Filter out blank rows (rows where all cells are empty, null, or undefined)
                var filteredData = jsonData.filter(row => row.some(filledCell));

                // Heuristic to find the header row by ignoring rows with fewer filled cells than the next row
                var headerRowIndex = filteredData.findIndex((row, index) =>
                  row.filter(filledCell).length >= filteredData[index + 1]?.filter(filledCell).length
                );
                // Fallback
                if (headerRowIndex === -1 || headerRowIndex > 25) {
                  headerRowIndex = 0;
                }

                // Convert filtered JSON back to CSV
                var csv = XLSX.utils.aoa_to_sheet(filteredData.slice(headerRowIndex)); // Create a new sheet
from filtered array of arrays
                csv = XLSX.utils.sheet_to_csv(csv, { header: 1 });
                return csv;
            } catch (e) {
```

```
            console.error(e);
            return "";
          }
        }
        return gk_fileData[filename] || "";
        }
```

HTML OL ELO-System Code

```
        </script><!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Foosball Elo Rating System</title>
  <style>
    body { font-family: Arial, sans-serif; margin: 20px; }
    h1, h2 { color: #333; }
    table { border-collapse: collapse; margin: 10px 0; }
    th, td { border: 1px solid #333; padding: 8px; text-align: left; }
    th { background-color: #f2f2f2; }
    input, textarea, select { margin: 5px; padding: 5px; }
    button { padding: 5px 10px; margin: 5px; }
    .section { margin-bottom: 20px; }
  </style>
</head>
<body>
  <h1>Foosball Elo Rating System</h1>
  <div class="section">
    <h2>Load Data</h2>
    <textarea id="data-input" rows="10" cols="50" placeholder="Paste data here (e.g., JustinCheng, 100,
100, 0, 0)"></textarea>
    <br/>
    <button onclick="loadPastedData()">Load Data</button>
    <br/>
    <font color="red"> scroll to bottom for example dataset for testing <front/>
  </div>
  <div class="section">
    <h2>Player Statistics</h2>
    <div id="player-list"></div>
  </div>
  <div class="section">
    <h2>Enter Commands</h2>
    <input type="text" id="command-input" size="50" placeholder="e.g., a;b closewin c;d">
    <button onclick="processCommand()">Process Command</button>
    <br/>
    <font color="black">
    pp - print player stats <br/>
    best - best players <br/>
    teama typewin teamb - (type: closewin, smallwin, win, bigwin, perfectwin) <br/>
    name - print player names alphabetically <br/>
    add - add new player <br/>
```

```
    combine - combine two players stats <br/>
    rank a/o/d - rank by average / offense / defense <br/>
  </div>
  <script>
    // Constants
    const K_FACTOR = 32;
    const RATING_MIN = 100;
    const RATING_MAX = 2999;
    const WIN_TYPE_MULTIPLIERS = {
      "win": 1.0,
      "smallwin": 0.75,
      "closewin": 0.5,
      "bigwin": 1.25,
      "perfectwin": 1.5
    };
    const RANK_THRESHOLDS = [
      [2900, "ultra"],
      [1650, "diamond"],
      [850, "plat"],
      [450, "gold"],
      [250, "silver"],
      [150, "copper"],
      [0, "iron"]
    ];
    const HIDDEN_RANK = "lz";
    const SPECIAL_IM = "im";
    const RANK_ORDER = {
      "iron": 1,
      "copper": 2,
      "silver": 3,
      "gold": 4,
      "plat": 5,
      "diamond": 6,
      "ultra": 7,
      [HIDDEN_RANK]: 8,
      [SPECIAL_IM]: 8
    };
    const RANK_INITIAL = {
      "iron": "i",
      "copper": "c",
      "silver": "s",
      "gold": "g",
      "plat": "p",
      "diamond": "d",
      "ultra": "u"
    };
    const RANK_FULL = Object.fromEntries(Object.entries(RANK_INITIAL).map(([k, v]) => [v, k]));

    let players = {};

    // Utility Functions
    function canonicalize(name) {
      return name.toLowerCase().replace(/[^a-z0-9]/g, '');
```

```
    }

    function getHiddenRank() {
      const letters = 'abcdefghijklmnopqrstuvwxyz';
      return 'L' + Array(4).fill().map(() => letters[Math.floor(Math.random() * letters.length)]).join('') +
          'Z' + Array(4).fill().map(() => letters[Math.floor(Math.random() * letters.length)]).join('');
    }

    function getRankDisplay(rank) {
      if (rank === HIDDEN_RANK) return getHiddenRank();
      if (rank === SPECIAL_IM) return "importal";
      return RANK_FULL[rank] || rank;
    }

    function getComputedRank(score) {
      for (let [threshold, rank] of RANK_THRESHOLDS) {
        if (score >= threshold) return rank;
      }
      return "iron";
    }

    function updatePlayerAvg(key) {
      let data = players[key];
      data.avg = Math.round((data.offense + data.defense) / 2);
    }

    function updatePlayerRanks(key) {
      let rec = players[key];
      let new_o = getComputedRank(rec.offense);
      let new_d = getComputedRank(rec.defense);
      let new_a = getComputedRank(rec.avg);
      for (let [field, new_val] of [["rank_o", new_o], ["rank_d", new_d], ["rank_a", new_a]]) {
        let current = rec[field] || "iron";
        if (current === HIDDEN_RANK || current === SPECIAL_IM) continue;
        rec[field] = RANK_ORDER[new_val] > RANK_ORDER[current] ? new_val : current;
      }
    }

    function highestOverallRank(key) {
      let rec = players[key];
      let ranks = [rec.rank_o || "iron", rec.rank_d || "iron", rec.rank_a || "iron"];
      if (ranks.some(r => r === HIDDEN_RANK || r === SPECIAL_IM)) return getHiddenRank();
      let best = ranks.reduce((a, b) => RANK_ORDER[a] > RANK_ORDER[b] ? a : b);
      return getRankDisplay(best);
    }

    function mergeRecord(key, new_display, off, deff, played, wins, rank_d = null, rank_o = null, rank_a =
null) {
      let old = players[key];
      let total_played = old.played + played;
      let new_off = total_played > 0 ? Math.round((old.offense * old.played + off * played) / total_played) :
off;
```

```
        let new_def = total_played > 0 ? Math.round((old.defense * old.played + deff * played) /
total_played) : deff;
        let new_wins = old.wins + wins;
        const chooseRank = (old_r, new_r) => RANK_ORDER[new_r || getComputedRank(new_r ? off :
new_off)] > RANK_ORDER[old_r] ? new_r : old_r;
        players[key] = {
          display: old.display,
          offense: new_off,
          defense: new_def,
          played: total_played,
          wins: new_wins,
          avg: Math.round((new_off + new_def) / 2),
          rank_d: chooseRank(old.rank_d || "iron", rank_d),
          rank_o: chooseRank(old.rank_o || "iron", rank_o),
          rank_a: chooseRank(old.rank_a || "iron", rank_a)
        };
      }

      // Player Creation
      function getOrCreatePlayer(name) {
        let key = canonicalize(name);
        if (!players[key]) {
          players[key] = {
            display: name,
            offense: RATING_MIN,
            defense: RATING_MIN,
            played: 0,
            wins: 0,
            avg: RATING_MIN,
            rank_o: "iron",
            rank_d: "iron",
            rank_a: "iron"
          };
          if (key.includes("zhong")) {
            players[key].rank_o = HIDDEN_RANK;
            players[key].rank_d = HIDDEN_RANK;
            players[key].rank_a = HIDDEN_RANK;
          }
        }
        return players[key];
      }

      // Data Loading
      function loadPastedData() {
        let text = document.getElementById("data-input").value;
        players = {};
        text.split('\n').forEach(line => {
          line = line.trim().replace(/\.$/, '');
          if (!line) return;
          let parts = line.split(',').map(p => p.trim());
          if (parts.length < 5) return;
          let [disp, off, deff, played, win_rate] = parts;
          let canon = canonicalize(disp);
```

```
            try {
                off = parseInt(off);
                deff = parseInt(deff);
                played = parseInt(played);
                win_rate = parseInt(win_rate);
                let wins = played > 0 ? Math.round((win_rate / 100) * played) : 0;
                let avg = parts[5] ? parseInt(parts[5]) : Math.round((off + deff) / 2);
                let rank_d = parts[6] || getComputedRank(deff);
                let rank_o = parts[7] || getComputedRank(off);
                let rank_a = parts[8] || getComputedRank(avg);
                if (players[canon]) {
                    mergeRecord(canon, disp, off, deff, played, wins, rank_d, rank_o, rank_a);
                } else {
                    players[canon] = { display: disp, offense: off, defense: deff, played, wins, avg, rank_d, rank_o,
rank_a };
                    if (canon.includes("zhong")) {
                        players[canon].rank_d = HIDDEN_RANK;
                        players[canon].rank_o = HIDDEN_RANK;
                        players[canon].rank_a = HIDDEN_RANK;
                    }
                }
            } catch (e) {
                console.error("Error parsing line:", line);
            }
        });
        saveData();
        printPlayers();
    }

    function saveData() {
        for (let key in players) {
            updatePlayerAvg(key);
            updatePlayerRanks(key);
        }
        localStorage.setItem('players', JSON.stringify(players));
    }

    function loadData() {
        let data = localStorage.getItem('players');
        if (data) players = JSON.parse(data);
    }

    // Display Functions
    function printPlayers() {
        let sorted = Object.entries(players).sort((a, b) => b[1].avg - a[1].avg ||
a[1].display.localeCompare(b[1].display));
        let table =
'<table><tr><th>No.</th><th>Name</th><th>Avg</th><th>Off</th><th>Def</th><th>T</th><th>Win%</th>
<th>Rank</th></tr>';
        sorted.forEach(([key, data], idx) => {
            let winRate = data.played > 0 ? Math.round((data.wins / data.played) * 100) : 0;
```

```
            table += `<tr><td>${idx +
1}</td><td>${data.display}</td><td>${data.avg}</td><td>${data.offense}</td><td>${data.defense}</td><t
d>${data.played}</td><td>${winRate}</td><td>${highestOverallRank(key)}</td></tr>`;
        });
        table += '</table>';
        document.getElementById('player-list').innerHTML = table;
    }

    function printPlayersAlphabetically() {
        let sorted = Object.entries(players).sort((a, b) => a[1].display.localeCompare(b[1].display));
        let table =
'<table><tr><th>Name</th><th>Avg</th><th>Off</th><th>Def</th><th>T</th><th>Win%</th></tr>';
        sorted.forEach(([_, data]) => {
            let winRate = data.played > 0 ? Math.round((data.wins / data.played) * 100) : 0;
            table +=
`<tr><td>${data.display}</td><td>${data.avg}</td><td>${data.offense}</td><td>${data.defense}</td><td>
${data.played}</td><td>${winRate}</td></tr>`;
        });
        table += '</table>';
        document.getElementById('player-list').innerHTML = table;
    }

    function printBestPlayers() {
        let bestAvg = Object.entries(players).reduce((a, b) => a[1].avg > b[1].avg ? a : b);
        let bestOff = Object.entries(players).reduce((a, b) => a[1].offense > b[1].offense ? a : b);
        let bestDef = Object.entries(players).reduce((a, b) => a[1].defense > b[1].defense ? a : b);
        let mostPlayed = Object.entries(players).reduce((a, b) => a[1].played > b[1].played ? a : b);
        let bestWinRate = Object.entries(players).reduce((a, b) => (a[1].played ? a[1].wins / a[1].played : 0) >
(b[1].played ? b[1].wins / b[1].played : 0) ? a : b);
        let text = `Best Players:<br>` +
            `Best Average: ${bestAvg[1].display} (A-${bestAvg[1].avg},
Rank-${getRankDisplay(bestAvg[1].rank_a)})<br>` +
            `Best Offense: ${bestOff[1].display} (O-${bestOff[1].offense},
Rank-${getRankDisplay(bestOff[1].rank_o)})<br>` +
            `Best Defense: ${bestDef[1].display} (D-${bestDef[1].defense},
Rank-${getRankDisplay(bestDef[1].rank_d)})<br>` +
            `Most Time Played: ${mostPlayed[1].display} (T-${mostPlayed[1].played})<br>` +
            `Highest Win Rate: ${bestWinRate[1].display} (R-${bestWinRate[1].played ?
(bestWinRate[1].wins / bestWinRate[1].played * 100).toFixed(2) : 0}%)`;
        document.getElementById('player-list').innerHTML = text;
    }

    function printRank(crit) {
        let sorted, headers, valueFunc, rankVal;
        if (crit === "a") {
            sorted = Object.entries(players).sort((a, b) => b[1].avg - a[1].avg ||
a[1].display.localeCompare(b[1].display));
            headers = ["No.", "Name", "Average", "Rank_A"];
            valueFunc = d => d[1].avg;
            rankVal = d => d[1].rank_a;
        } else if (crit === "o") {
            sorted = Object.entries(players).sort((a, b) => b[1].offense - a[1].offense ||
a[1].display.localeCompare(b[1].display));
```

```
            headers = ["No.", "Name", "Offense", "Rank_O"];
            valueFunc = d => d[1].offense;
            rankVal = d => d[1].rank_o;
        } else if (crit === "d") {
            sorted = Object.entries(players).sort((a, b) => b[1].defense - a[1].defense ||
a[1].display.localeCompare(b[1].display));
            headers = ["No.", "Name", "Defense", "Rank_D"];
            valueFunc = d => d[1].defense;
            rankVal = d => d[1].rank_d;
        } else if (crit === "t") {
            sorted = Object.entries(players).sort((a, b) => b[1].played - a[1].played ||
a[1].display.localeCompare(b[1].display));
            headers = ["No.", "Name", "Played"];
            valueFunc = d => d[1].played;
            rankVal = () => "";
        } else if (crit === "r") {
            sorted = Object.entries(players).sort((a, b) => (b[1].played ? b[1].wins / b[1].played : 0) -
(a[1].played ? a[1].wins / a[1].played : 0) || a[1].display.localeCompare(b[1].display));
            headers = ["No.", "Name", "Win%"];
            valueFunc = d => d[1].played > 0 ? Math.round((d[1].wins / d[1].played) * 100) : 0;
            rankVal = () => "";
        } else if (["a-rank", "o-rank", "d-rank"].includes(crit)) {
            let field = { "a-rank": "rank_a", "o-rank": "rank_o", "d-rank": "rank_d" }[crit];
            sorted = Object.entries(players).sort((a, b) => RANK_ORDER[b[1][field] || "iron"] -
RANK_ORDER[a[1][field] || "iron"] || b[1].avg - a[1].avg);
            headers = ["No.", "Name", field.toUpperCase()];
            valueFunc = () => "";
            rankVal = d => d[1][field] || "iron";
        } else {
            document.getElementById('player-list').innerHTML = "Unsupported rank criteria.";
            return;
        }
        let table = `<table><tr>${headers.map(h => `<th>${h}</th>`).join('')}</tr>`;
        let ordinal = 1;
        sorted.forEach(d => {
            let storedRank = rankVal(d);
            let num = storedRank === HIDDEN_RANK ? "0" : ordinal;
            let row = `<tr><td>${num}</td><td>${d[1].display}</td>`;
            if (valueFunc(d) !== "") row += `<td>${valueFunc(d)}</td>`;
            if (rankVal(d) !== "") row += `<td>${getRankDisplay(storedRank)}</td>`;
            table += row + '</tr>';
            if (storedRank !== HIDDEN_RANK) ordinal++;
        });
        table += '</table>';
        document.getElementById('player-list').innerHTML = table;
    }

    // Command Processing
    function processGame(command) {
        const match = command.match(/^(.*?)\s*(win|smallwin|closewin|bigwin|perfectwin)\s*(.*?)$/i);
        if (!match) {
            document.getElementById('player-list').innerHTML = "Invalid game command format.";
            return;
```

```
        }
        let [_, team1Str, winType, team2Str] = match;
        winType = winType.toLowerCase();
        if (!WIN_TYPE_MULTIPLIERS[winType]) {
          document.getElementById('player-list').innerHTML = "Invalid win type.";
          return;
        }
        let baseMultiplier = WIN_TYPE_MULTIPLIERS[winType];
        function parseTeam(str) {
          if (str.includes(';')) {
            let parts = str.split(';').map(s => s.trim());
            let offense = parts[0].split(',').map(s => s.trim()).filter(Boolean);
            let defense = parts.slice(1).join(',').split(',').map(s => s.trim()).filter(Boolean);
            return [offense, defense];
          } else {
            let offense = str.split(',').map(s => s.trim()).filter(Boolean);
            return [offense, []];
          }
        }
        let [team1Off, team1Def] = parseTeam(team1Str);
        let [team2Off, team2Def] = parseTeam(team2Str);
        [team1Off, team1Def, team2Off, team2Def].flat().forEach(getOrCreatePlayer);
        function getAverageRating(names, type) {
          if (!names.length) return null;
          return Math.round(names.reduce((sum, n) => sum + getOrCreatePlayer(n)[type], 0) /
names.length);
        }
        let oppForTeam1 = team2Def.length ? getAverageRating(team2Def, "defense") :
getAverageRating(team2Off, "offense");
        let oppOffTeam1 = team2Off.length ? getAverageRating(team2Off, "offense") :
getAverageRating(team2Def, "defense");
        let oppForTeam2 = team1Def.length ? getAverageRating(team1Def, "defense") :
getAverageRating(team1Off, "offense");
        let oppOffTeam2 = team1Off.length ? getAverageRating(team1Off, "offense") :
getAverageRating(team1Def, "defense");
        function updateRating(current, win, opp, multiplier) {
          let change = Math.round(K_FACTOR * multiplier);
          return [win ? current + change : current - change, change];
        }
        team1Off.forEach(n => {
          let p = getOrCreatePlayer(n);
          [p.offense] = updateRating(p.offense, 1, oppForTeam1 || 1500, baseMultiplier);
          p.played++;
          p.wins++;
        });
        team1Def.forEach(n => {
          let p = getOrCreatePlayer(n);
          [p.defense] = updateRating(p.defense, 1, oppOffTeam1 || 1500, baseMultiplier);
          p.played++;
          p.wins++;
        });
        team2Off.forEach(n => {
          let p = getOrCreatePlayer(n);
```

```
            [p.offense] = updateRating(p.offense, 0, oppForTeam2 || 1500, baseMultiplier);
            p.played++;
        });
        team2Def.forEach(n => {
            let p = getOrCreatePlayer(n);
            [p.defense] = updateRating(p.defense, 0, oppOffTeam2 || 1500, baseMultiplier);
            p.played++;
        });
        for (let key in players) {
            updatePlayerAvg(key);
            updatePlayerRanks(key);
        }
        saveData();
        document.getElementById('player-list').innerHTML = "Game processed and ratings updated.";
        printPlayers();
    }

    function processAdd(command) {
        let info = command.slice(3).trim();
        let parts = info.split(',').map(p => p.trim());
        if (parts.length === 1) {
            getOrCreatePlayer(parts[0]);
            saveData();
            document.getElementById('player-list').innerHTML = `Player ${parts[0]} added with default
stats.`;
            printPlayers();
        } else if (parts.length === 9) {
            let [name, off, deff, played, wins, avg, rank_d, rank_o, rank_a] = parts;
            try {
                off = parseInt(off);
                deff = parseInt(deff);
                played = parseInt(played);
                wins = parseInt(wins);
                avg = parseInt(avg);
                let canon = canonicalize(name);
                players[canon] = { display: name, offense: off, defense: deff, played, wins, avg, rank_d, rank_o,
rank_a };
                if (canon.includes("zhong")) {
                    players[canon].rank_d = HIDDEN_RANK;
                    players[canon].rank_o = HIDDEN_RANK;
                    players[canon].rank_a = HIDDEN_RANK;
                }
                saveData();
                document.getElementById('player-list').innerHTML = `Player ${name} added with specified
stats.`;
                printPlayers();
            } catch (e) {
                document.getElementById('player-list').innerHTML = "Error: Incorrect format in add
command.";
            }
        } else {
            document.getElementById('player-list').innerHTML = "Wrong format for add command.";
        }
```

```
    }

    function processCombine(command) {
      let parts = command.slice(7).trim().split(',').map(p => p.trim());
      if (parts.length !== 2) {
        document.getElementById('player-list').innerHTML = "Combine requires two names separated by
a comma.";
        return;
      }
      let [name1, name2] = parts;
      let canon1 = canonicalize(name1);
      let canon2 = canonicalize(name2);
      if (!players[canon1] || !players[canon2]) {
        document.getElementById('player-list').innerHTML = "One of the players does not exist.";
        return;
      }
      mergeRecord(canon1, players[canon1].display, players[canon2].offense, players[canon2].defense,
players[canon2].played, players[canon2].wins);
      delete players[canon2];
      saveData();
      document.getElementById('player-list').innerHTML = `Players ${name1} and ${name2} combined.`;
      printPlayers();
    }

    function processCommand() {
      let command = document.getElementById('command-input').value.trim().toLowerCase();
      if (command === "pp") printPlayers();
      else if (command === "best") printBestPlayers();
      else if (command === "name") printPlayersAlphabetically();
      else if (command.startsWith("add ")) processAdd(command);
      else if (command.startsWith("combine ")) processCombine(command);
      else if (command.startsWith("rank ")) printRank(command.split(' ')[1]);
      else processGame(command);
      document.getElementById('command-input').value = '';
    }

    // Initialize
    loadData();
    if (Object.keys(players).length) printPlayers();
  </script>
  <font color="red">
Example Data Set (Copy this for testing): <br/>
<br/>
<font color="black">
</body></html>
```