Aufgabe 3: Tobis Turnier

Team-ID: ?????

Team: Teamname (oder dein Name)

Bearbeiter/-innen dieser Aufgabe:
Christopher Besch, Katharina Libner

10. November 2020

Lösungsidee

Es soll die Turniervariante gefunden werden, bei dem der Spieler mit der höchsten Spielstärke am häufigsten gewonnen hat.

Die verschiedenen Turniervarianten werden alle jeweils n-mal simuliert und anschließend miteinander verglichen. Da das Spiel RNG zufällig abläuft, ist die Wahl von n entscheidend. Je größer n gewählt wird, desto öfter werden die Turniervarianten simuliert und das bessere Spiel hebt sich von den anderen auffälliger ab.

Beim Vergleichen kommt es auf die gewonnenen Spiele des Spielers mit der höchsten Spielstärke an. Jeder Sieg dieses Spielers muss also festgehalten werden. Das Spiel mit den meisten Siegen des Spielers mit der höchsten Spielstärke soll ausgegeben werden.

Umsetzung

Die Lösungsidee wird in Python implementiert. Das gesamte Programm ist in verschiedene Pythondatein unterteilt, damit die Zusammenhänge der Methoden deutlicher erscheinen.

Die **Main-Methode** führt das Programm aus und läuft wie folgt ab:

Zunächst wird beim Starten des Programms vom Ausführer verlangt die Anzahl der Wiederholungen und die Testdatei der Spielstärken in der Konsole als Argumente einzugeben. Zu empfehlen ist ein Wert von >100, damit die beste Spielvariante mit höherer Wahrscheinlichkeit gefunden werden kann.

Das Programm liest mit Hilfe der Methode **read_skill_levels** die Spielstärken chronologisch ein und gibt sie in Form einer Liste zurück. Die Spielstärken stehen in den Testdateien und belegen jeweils eine Zeile.

Darauffolgend werden die verschiedenen Turniervarianten je nach Eingabe der Wiederholungen in einer while-Schleife mehrmals simuliert und die Siege des besten Spielers pro Variante gezählt. Damit nicht jedes Turnier in der gleichen Reinfolge der Spieler abläuft, die gegeneinander antreten, werden pro Wiederholungen, die Reinfolgen der Spieler mit ihren Spielstärken per Zufall neu gemischt.
Es existieren drei Spielvarianten:

1. RNG Liga
2. RNG KO
3. RNG KO5

Die Ausführung der Spielvarianten konnte wie folgt strukturiert werden:

I. Es wird nach dem Gewinner gesucht, dabei müssen folgende Unterschiede beachten werden:

Beim Turnierablauf unterscheidet sich RNG Liga zu RNG KO und RNG KO5. Es werden zwei verschiedene Methoden für die Turnierabläufe, die den Gewinner zurückgeben implementiert:

1. **find_winning_player_liga**

   Diese Methode wird für die 1. Spielvariante benutzt.

   Jeder spielt gegen jeden einmal **play_RNG** und die Siege jedes Spielers werden gezählt. Das ganze Turnier muss simuliert werden, bis der endgültige Gewinner feststeht.

2. **find_winning_player_KO**

   Diese Methode wird für die 2. und 3. Spielvariante benutzt.

   Alle Spieler werden in Paare aufgeteilt und spielen **play_RNG** oder **play_RNG_5** gegeneinander, die Gewinner werden wiederum in Paare aufgeteilt usw. Das Ganze wurde rekursiv implementiert. Sobald der beste Spieler während des Turniers einmal verliert, ist er aus dem Rennen und die Simulation kann abgebrochen werden. Dann wird ein Wert zurückgegeben, der ungleich der Nummer des besten Spielers ist.

II. Um den Gewinner zu finden, muss das Spiel RNG simuliert werden:

Die erste und zweite Variante unterscheiden sich zur 3. in den Wiederholungen des Spiels RNG. Es werden zwei verschiedene Methoden der Spielmodi implementiert.

1. **play_RNG**

   RNG wird bei der 1. und 2. Variante einmal gespielt.

2. **play_RNG_5**

RNG wird bei der 3. Variante fünf mal gespielt.

III. Es wird überprüft, ob der beste Spieler gewonnen hat.

Nachdem der Gewinner ermittelt wurde, wird mit Hilfe der Methode **does_player_win** bei jeder Simulation festgestellt, ob es sich um den besten Spieler handelt. Ist dies der Fall wird die Anzahl der Siege des Spiels hochgezählt.

In der **Main-Methode** werden die Siege der Turniervariante des besten Spielers durch Fallunterscheidungen verglichen. Die Spielvariante mit den meisten Siegen und der Durchschnitt der Siege des besten Spielers pro Variante werden vom Programm zurückgegeben.

Beispiele

Das Programm wurde mit den folgenden Beispielen und der CMD Eingabe für n=1000 getestet.

1.Beispiel: spielstaerken1.txt

Ausgabe:

Wie oft hat der spielstärkste Spieler im Durschnitt gewonnen:

LIGA: 0.116

KO: 0.382

KO5: 0.615

Beste Spielvariante: KO5

2.Beispiel: spielstaerken2.txt

Ausgabe:

Wie oft hat der spielstärkste Spieler im Durschnitt gewonnen:

LIGA: 0.117

KO: 0.277

KO5: 0.363

Beste Spielvariante: KO5

3.Beispiel: spielstaerken3.txt

Ausgabe:

Wie oft hat der spielstärkste Spieler im Durschnitt gewonnen:

LIGA: 0.062

KO: 0.164

KO5: 0.314

Best game mode: KO5

4.Beispiel: spielstaerken4.txt

Ausgabe:

Wie oft hat der spielstärkste Spieler im Durschnitt gewonnen:

LIGA: 0.066

KO: 0.082

KO5: 0.086

Best game mode: KO5

Genügend Beispiele einbinden! Die Beispiele von den BWINF-Webseiten sollten hier diskutiert werden, aber auch eigene Beispiele sind sehr gut – besonders wenn sie Spezialfälle abdecken. Bitte jedoch nicht 30 Seiten Programmausgabe hier einfügen!

Quellcode

```
def find_winning_player_liga(skill_levels: List[int], play_game = play_RNG) ->
int:
    """
    simulate the game mode "LIGA" and return the winner
    """

    # the number of players equals to the length of existing skill levels
    players = list(range(len(skill_levels)))
    # declare a list of integers that represents the ranks, every player starts
with a score of zero
    ranking = [0] * len(skill_levels)
    # in the following for-loop, the list of players will shape to an two dimen-
sional list
    # every element in players presents a pair of two
    for first_player, second_player in itertools.combinations(players, 2):
        # first_player and second_player play against each other
        # save the winner
        winning_player = play_game(first_player, second_player, skill_levels)
        # counts the wins of the winner
        ranking[winning_player] = ranking[winning_player] + 1
    # find the player with most wins and return his number
    best_player = max(ranking)
    return best_player
```

```
def find_winning_player_KO(start_player: int, end_player: int, skill_levels, ex-
pected_best_player, play_game) -> int:
    """
    recursive function
    simulate the game mode "KO" or "KO5" and return the winner
    start_player and end_player define an interval
    start_player is located in the interval and marks the first player on the
left
    end_player isn't located in the interval and marks the limit on the right

    """

    if start_player == end_player - 1:
        # base case
        return start_player

    middle_player = int((start_player + end_player) / 2)

    # search the interval in which the expected best player takes place
    if middle_player <= expected_best_player < end_player:
        # interval on the right
        right_player = find_winning_player_KO(middle_player, end_player,
skill_levels, expected_best_player, play_game)
        if right_player != expected_best_player:
            return -1
        left_player = find_winning_player_KO(start_player, middle_player,
skill_levels, expected_best_player, play_game)
    elif start_player <= expected_best_player < middle_player:
        # interval on the left
        left_player = find_winning_player_KO(start_player, middle_player,
skill_levels, expected_best_player, play_game)
        if left_player != expected_best_player:
            return -1
        right_player = find_winning_player_KO(middle_player, end_player,
skill_levels, expected_best_player, play_game)
    else:
        # expected_best_player is not located in any interval
        left_player = find_winning_player_KO(start_player, middle_player,
skill_levels, expected_best_player, play_game)
        right_player = find_winning_player_KO(middle_player, end_player,
skill_levels, expected_best_player, play_game)
    return play_game(left_player, right_player, skill_levels)
```

```
def play_RNG(player1: int, player2: int, skill_levels: List[int]) -> int:
    """
    player1 and player2 plays RNG against each other
    the skill level of a player determines the number of murmurs in the urn
    return the winner

    """
    skill_level1 = skill_levels[player1]
    skill_level2 = skill_levels[player2]
    # determine a random murmur
    random_murmur = random.randint(1, skill_level1 + skill_level2)
    # the owner of the drawn murmur wins
    if random_murmur <= skill_level1:
        return player1
    else:
        return player2
```