Aufgabe 3: Tobis Turnier

Team-ID: 00301

Team: Volatile Violets

Bearbeiter/-innen dieser Aufgabe:
Katharina Libner

10. November 2020

Lösungsidee 1
Umsetzung 1
Beispiele..... 3
Quellcode 5

Lösungsidee

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

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 und der Siegesdurchschnitt jeder Variante soll ausgegeben werden.

Umsetzung

Die Lösungsidee wird in Python implementiert. Das gesamte Programm ist in verschiedene Pythondateien 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 genügend hoher 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 pro Spieler.

Darauffolgend werden die verschiedenen Turniervarianten je nach Eingabe der Wiederholungen in einer 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 kann 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

   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 zwei Hälften geteilt. Jede Hälfte wird erneut geteilt usw. Hier wird die Rekursion deutlich. Die überbleibenden Spieler spielen **play_rng** oder **play_rng_5** gegeneinander, die Gewinner wird zurückgegeben und spielt gegen einen Partner auf einer höheren Ebene weiter. Die Hälfte, in der sich nicht der erwartete Gewinner aufhält, wird aus Optimierungsgründen ignoriert. Sobald der beste Spieler während des Turniers einmal verliert, ist er aus dem Rennen und die Simulation kann abgebrochen werden. Dann wird die Nummer -1 zurückgegeben, die nie der Spielnummer des besten Spielers entspricht.

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

Die erste und zweite Variante unterscheiden sich zur 3. in den Wiederholungen der Spielmodi 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=10000 getestet.

1. **Beispiel: spielstaerken1.txt**

   LIGA: 0.4701

   KO: 0.4022

   KO5: 0.6002

   Beste Spielvariante: K05

Die Anzahl der Teilnehmer ist gering und der beste Spieler hat eine hohe Spielstärkendifferenz, somit weist er einen hohen Siegesdurchschnitt auf.

3. **Beispiel: spielstaerken2.txt**

   LIGA: 0.3248

   KO: 0.305

   KO5: 0.3639

   Beste Spielvariante: K05

Hier sind ähnliche Bedingungen wie beim ersten Beispiel gegeben, nur dass die Spielstärkendifferenz geringer ist. Deswegen ist der Siegesdurschnitt ebenfalls geringer.

4. **Beispiel: spielstaerken3.txt**

   LIGA: 0.2575

KO: 0.1675

KO5: 0.2743

Beste Spielvariante: K05

Die Anzahl der Teilnehmer ist deutlich größer, somit wird die Wahrscheinlichkeit, dass der beste Spieler gewinnt geringer. Dies simuliert der verminderte Siegesdurchschnitt.

5. **Beispiel: spielstaerken4.txt**

   LIGA: 0.0692

   KO: 0.0694

   KO5: 0.0736

   Beste Spielvariante: K05

Aufgrund der enorm starken Konkurrenz ist die Wahrscheinlichkeit für den besten Spieler im Vergleich zu den vorherigen Beispielen viel geringer, am häufigsten zu gewinnen.

6. **Beispiel: my_speilstaerken1.txt**

   LIGA: 0.1251

   KO: 0.1268

   KO5: 0.1234

   Beste Spielvariante: KO

   Da alle die gleiche Spielstärke haben, erwarten man, dass alle die gleiche Wahrscheinlichkeit $\frac{1}{13} \approx 0{,}0769$ aufweisen. Diese Wahrscheinlichkeit ist als Siegesdurchschnitt für den besten Spieler zu erwarten, der bei einer Teilnahme von mehreren besten Spielern der Spieler mit der kleinsten Nummer ist. Wenn beim Spielmodus Liga mehrere Spieler gleich oft gewinnen, soll der Spieler mit der kleinsten Nummer zurückgegeben werden. Hier lässt sich erkennen, dass der beste Spieler beim Spielmodus Liga einen Vorteil besitzt, deswegen weist er eine höhere Wahrscheinlichkeit auf als die anderen Spieler. Da die Anzahl der Teilnehmer keine Potenz von 2 ist, haben beim Ablauf in mindestens einer Runde von KO oder KO5 keinen Partner, weshalb sie diese Runde automatisch gewinnen und daher einen Vorteil gegenüber den anderen Spielern mit Partner besitzen. Wenn Spieler ohne Partner vorliegen gehört der erste zu diesen. Wie bereits beschrieben, wird, wenn alle Spieler die gleiche Spielstärke aufweisen, der Erste als erwartete beste Spieler ausgewählt. Somit gewinnt der gewählte beste Spieler deutlich häufiger als erwartet.

7. **Beispiel: my_speilstaerken2.txt**

   LIGA: 0.1002

   KO: 0.0643

   KO5: 0.0591

   Beste Spielvariante: LIGA

   Dieses Beispiel ähnelt dem vorherigen, der einzige Unterschied liegt in der Anzahl der Teilnehmer, die von 13 zu 16 erhöht wurde. Liga weist ein leicht geringeren Siegesdurchschnitt auf als beim vorherigen Beispiel, was mit der vergrößerten Konkurrenz zu erklären ist. Da die Anzahl nun eine Potenz von 2 ist, existieren nie Partnerlose, somit nähert sich der Siegesdurchschnitt von KO und KO5 der erwarteten Wahrscheinlichkeit von $\frac{1}{16} \approx 0{,}0625$ an.

8. **Beispiel: my_speilstaerken3.txt**
   LIGA: 0.058

   KO: 0.018

   KO5: 0.017

   Beste Spielvariante: LIGA

   Trotz einer langen Eingabe produziert das Programm ein korrektes Ergebnis.

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] += 1
    # return the first player with the most wins
    return ranking.index(max(ranking))
```

```
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

    return -1 when the best player loses
    """

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

    middle_player = (start_player + end_player) // 2

    # search the interval in which the expected best player takes place
    # <- it's only important if the expected player wins, everyone else is unim-
portant
    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 in neither 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)
```

```python
def play_rng(player1: int, player2: int, skill_levels: List[int]) -> int:
    """
    player1 and player2 play 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 marble
    random_marble = random.randint(1, skill_level1 + skill_level2)
    # the owner of the drawn marble wins
    if random_marble <= skill_level1:
        return player1
    else:
        return player2
```

```python
def does_player_win(game_name: str, skill_levels: List[int]) -> bool:
    """
    find out if the expected best player wins and return a boolean
    """
    # determine the player with the best skill level
    expected_best_player = skill_levels.index(max(skill_levels))
    # determine the winner of the given play mode
    if game_name == "LIGA":
        winning_player = find_winning_player_liga(skill_levels)
    elif game_name == "KO":
        winning_player = find_winning_player_ko(0, len(skill_levels), skill_lev-
els, expected_best_player, play_rng)
    elif game_name == "KO5":
        winning_player = find_winning_player_ko(0, len(skill_levels), skill_lev-
els, expected_best_player, play_rng_5)
    else:
        raise ValueError("unsupported game_name used.")
    # compare the winner with expected best player
    return winning_player == expected_best_player
```