1/7

Aufgabe 3: Tobis Turnier

Team-ID: 00301

Team: Volatile Violets

Bearbeiter/-innen dieser Aufgabe:  
Katharina Libner

10. November 2020

[Lösungsidee 1](#_Toc56178028)

[Umsetzung 1](#_Toc56178029)

[Beispiele 3](#_Toc56178030)

[Quellcode 5](#_Toc56178031)

# 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 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.py-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 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 kann wie folgt strukturiert werden:

1. 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.

1. **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.

1. 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.

1. **play\_RNG\_5**

RNG wird bei der 3. Variante fünf Mal gespielt.

1. 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**

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.

1. **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.

1. **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.

1. **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.

1. **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 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.

1. **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 an.

1. **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 dimensional 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, expected\_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 unimportant  
 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)

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

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\_levels, expected\_best\_player, play\_rng)  
 elif game\_name == "KO5":  
 winning\_player = find\_winning\_player\_ko(0, len(skill\_levels), skill\_levels, 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