39. Bundeswettbewerb Informatik

Runde 1

Aufgabe 3: Tobis Turnier

01. November 2020

Inhaltsverzeichnis

Lösungsidee

Man muss also die drei gegebenen Turnierformen implementieren, nämlich Liga, K.O. und K.O. x5.

Liga

Die Turnierform *Liga*, wie in der Aufgabe beschrieben, heißt, dass jeder Spieler **einmal** gegeneinander spielt.

Um das zu ermöglichen, fängt man mit dem ersten Spieler an. Angenommen, es gibt n Spieler. Der Spieler 1 spielt gegen alle anderen Spieler. Danach spielt der Spieler 2 gegen alle anderen Spieler außer dem Spieler 1. Der Spieler 3 spielt dann gegen alle anderen Spieler, die eine größere Spielernummer als 3 besitzen, also gegen alle Spieler außer 1 und 2. So macht man weiter bis der Spieler *n-1* spielt gegen den Spieler *n*.

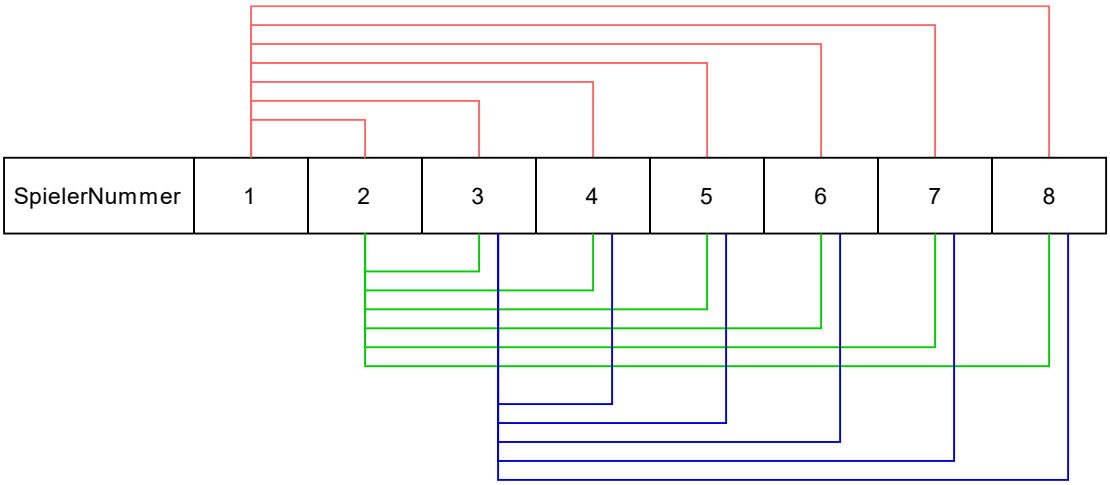


Abb. 1 Hier wird nur bis den Spieler 3 demonstriert. Man führt das Turnier mit demselben Verfahren bis Spieler 7 gegen 8.

Man speichert der Sieger jeder Runde. Nachdem alle Spieler einmal gegeneinander gespielt haben, rechnet man, wer die höchste Anzahl von Siegen hat. Beim Gleichstand gewinnt der Spieler mit der niedrigsten Spielernummer. Der ist dann der Gewinner des Turniers. Führt man das Turnier mehrmals durch und speichert dabei, wer wie oft jeder Spieler gewinnt, so erhält man am Ende das von der Aufgabe geforderte Ergebnis.

K.O. System

Hier wird die Spieler also mithilfe eines Spielplans verteilt und spielt **einmal** gegeneinander. Der Spieler, der gewinnt, kommt in die nächste Runde.

Das Verfahren kann man mit einem (umgedrehten) *vollständigen Binärbaum* deutlich machen. Die nach dem Spielplan verteilten Spieler stehen am Anfang in den äußeren Knoten (vgl. Abb.). Zur Demonstration nutzen wir hier den Spielplan *1,8,2,7,3,6,4,5*.

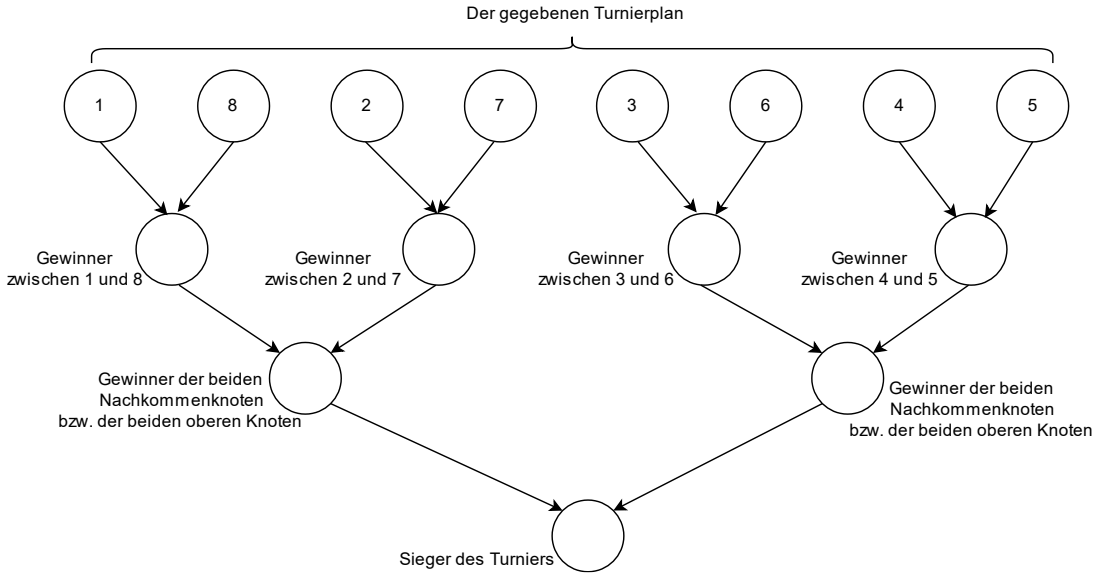


Abb. 2

Es wird also zunächst Spieler 1 gegen 8 gespielt und dann 2 gegen 7. Der Gewinner zwischen 1 und 8 spielt dann gegen den Gewinner zwischen 2 und 7. Man speichert also den Gewinner zwischen den beiden „Geschwister-Knoten“ in deren „Eltern-Knoten“. Führt man das für jeden zwei „Geschwister-Knoten“ durch, so erhält man am Ende an der Wurzel ein Gewinner, der gleichzeitig auch der Sieger des Turniers ist.

Führt man das Turnier mehrmals durch und notiert dabei die Anzahl der Siege für jeden Spieler (also wir vielmal ist der Spieler als Wurzel des Baums erschienen), so erhält man am Ende das von der Aufgabe geforderte Ergebnis, nämlich die relative Häufigkeit, wie oft der spielstärkste Spieler tatsächlich das Turnier auch gewinnt.

K.O. x5

Das Verfahren ist gleich wie das K.O. System, nur spielt jeder Spieler nun 5-mal gegeneinander.

Die relative Häufigkeit, dass der spielstärksten Spieler tatsächlich gewinnt, wird mit folgender Formel berechnet:

$$h = \frac{Anzahl\ der\ Siege\ von\ dem\ spielstärksten\ Spieler}{Anzahl\ der\ durchgeführten\ Turnier}$$

Die Empfehlung der Turniervariante

Unter Berücksichtigung der Ergebnisse, die in dem Abschnitt *Beispiel* vorgestellt werden, würde ich die Turnierform ***K.O. x5*** als die bestgeeignete Turniervariante nennen. Ich begründe diese Empfehlung wie folgt:

1. Durch diese Turnierform kann der beste Spieler mit der niedrigsten Anzahl von Wiederholungen gewinnen, wenn es zwischen den Spielstärken der Spieler großen Unterschied gibt. Das sieht man in Beispiel 1 und Beispiel 2. In diesem Fall wird der spielstärkste Spieler zwischen zwei „Geschwister-Knoten“ (vgl. Abb. 2) durch fünfmalige Durchführung von Vergleichen sehr wahrscheinlich gewinnen.

2. In Beispiel 3 und Beispiel 4 sieht die Situation etwas anders aus. Auf dem ersten Blick wäre *Liga* eine bessere Turniervariante, sodass der beste Spieler auch am häufigsten gewinnt. Jedoch muss man die Nummerierung der Spieler mitberücksichtigen. In dem beiden genannten Beispielen hat der spielstärkste Spieler nämlich auch eine sehr kleine Spielernummer (4/16 und 1/16). In der Regel von Liga gewinnt beim Gleichstand der Spieler mit der kleinsten Spielernummer. Somit erhalten die Spieler, die eine kleinere Spielernummer besitzen, einen Vorteil, der in einer Situation wie Beispiel 4, wo die Spielstärke der Spieler sehr ähnlich sind, das Endergebnis sehr stark beeinflussen wird.

   Im Beispiel 4 > Spielplanvariante 1 sieht das Ergebnis so aus:

```
KOx1: 1 - 0,069268 - 14,436680718369233
KOx5: 1 - 0,075014 - 13,33084490895033
Liga: 1 - 0,115126 - 8,686135191008113
```

   Tauschen wir nur die Reihfolge zweier Spieler, also der Spieler 1 und 16, wird das Ergebnis bereits sehr anders sein. Die **geänderte** Spielstärke-Datei sieht nun so aus (blau markierte sind die getauschten beiden Spieler):

```
16
95
95
95
95
95
95
95
95
95
95
95
95
95
95
95
100
```

   Führt man das Programm mit dieser geänderten Datei nochmal durch (der Turnierplan bleibt unberührt), sieht das Ergebnis nun aber so aus:

```
KOx1: 16 - 0,068972 - 14,498637128109957
KOx5: 16 - 0,075238 - 13,291156064754512
Liga: 16 - 0,048436 - 20,64580064414898
```

   Das Ergebnis für KOx5 bleibt unverändert, jedoch beträgt die benötigten Wiederholungen des spielstärksten Spielers bei der Turnierform *Liga* schon 20,6458,

welche stark von dem vorherigen Ergebnis abweichen. Das zeigt, dass das Ergebnis für Liga nicht nur von den Spielstärken der Spieler, sondern auch stark von der **Reihfolge** der Spieler abhängig. Würde man die Spieler einfach anders nummerieren, ändert sich das Ergebnis für Liga enorm.

3. Stattdessen beeinflusst die Reihfolge das Ergebnis für die beiden KO-Variante nicht. Die sind nämlich von dem eingegebenen Spielplan abhängig. Mit bestimmtem Spielplan lässt sich das Ergebnis auch voneinander abweichen (Vgl. Beispiel 3 > Spielplanvariante 4 mit allen drei anderen Ergebnissen). Wenn man das Aspekt vom Punkt 2 berücksichtigt, dann sieht man, dass das KOx5-System trotz solcher von dem Spielplan abhängigen Abweichungen im Allgemein das nach der Definition der Aufgabe beste Ergebnis liefert.

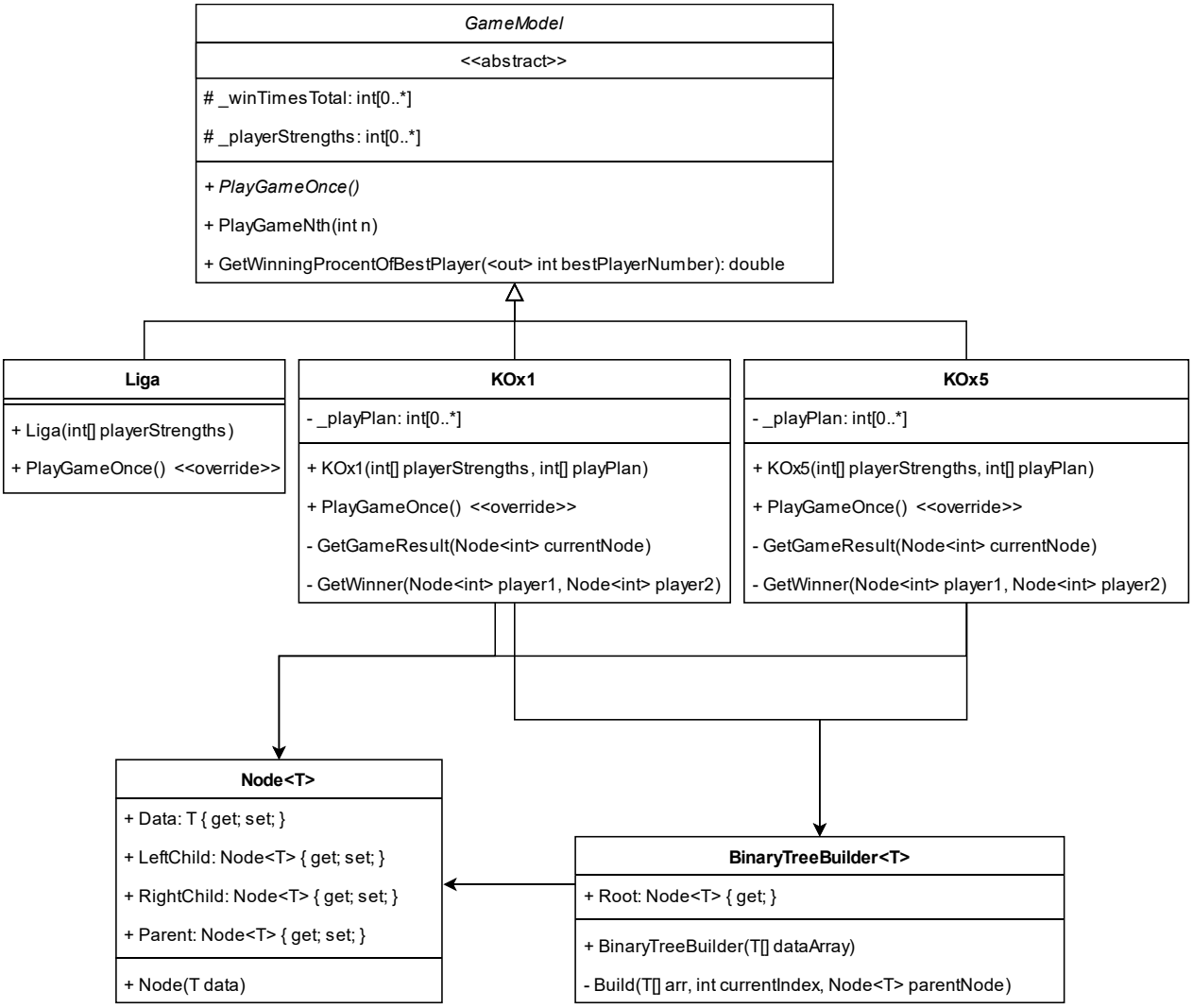
Umsetzung

Der Name der Test-Datei muss beim Öffnen des Programms eingegeben werden. Diese Datei wird dann mithilfe von *StreamReader* eingelesen. Danach muss man den Spielplan für die Turnierformen *KO* eingeben, deren Gültigkeit geprüft wird.

Das Einlesen von Daten wird hier nicht konkret beschrieben. Der Quellcode dazu befindet sich in *Quellcode>Aufgabe3.cs*.

Hier werden drei Klassen implementiert, die den drei in der Aufgabe beschriebenen Turnierformen, Liga, K.O. und K.O. x5 entsprechen.

Alle drei Klassen der Turnierformen erben eine abstrakte Vaterklasse *GameModel*, in der definiert wird, welche Attribute und Methode der jeweiligen konkreten Turnierformen haben sollten (vgl. Klassendiagramm).



Fangen wir mit der Klasse *GameModel* an.

```
    private abstract class GameModel
```

Das Attribute *_winTimesTotal: int[]* speichert die Anzahl der Siege von allen Spielern. D. h., es gibt an, wie viel Mal hat der Spieler das Turnier gewonnen. Bspw. ist *_winTimesTotal[3]* die Anzahl der Siege von dem Spieler 4.

Das Attribute *_playerStrengths: int[]* speichert die von der Test-Datei eingelesene Spielstärken.

```
      protected int[] _winTimesTotal;
      protected int[] _playerStrengths;
```

Die abstrakte Methode *PlayGameOnce()* wird in den konkreten Klassen, Liga, KOx1 und KOx5 implementiert. Diese Methode soll grundsätzlich das Turnier nach einer bestimmten Turnierform einmal durchführen und herausfinden, welcher Spieler gewinnt. Dann soll der Wert *_winTimesTotal[Nummer des Spielers, der dieses Turnier diesmal gewinnt - 1]* um eins erhöht werden. Sei der Gewinner mit der Nummer *n*, so wird *_winTimesTotal[n-1]++*. (-1 liegt daran, dass der Index mit 0 anfängt).

```
      public abstract void PlayGameOnce();
```

Die Methode *PlayGameNth(int n)* führt die Methode *PlayGameOnce()* n mal durch.

```
      public void PlayGameNth(int n)
      {
        for (int i = 0; i < n; i++)
        {
          PlayGameOnce();
        }
      }
```

Die Methode *GetWinningProcentOfBestPlayer(out int bestPlayerNummer): double* gibt nach genügender Durchführung ein Double-Wert zurück, nämlich die relative Häufigkeit in Dezimalzahl dafür, dass der spielstärkste Spieler das Turnier gewinnt. Außerdem wird die Nummer des spielstärksten Spielers auch durch das Parameter *bestPlayerNummer* angegeben.

```
public double GetWinningProcentOfBestPlayer(out int bestPlayerNumber){
    bestPlayerNumber = 0;
    for (int i = 0; i < _playerStrengths.Length; i++){
      if (_playerStrengths[i] > _playerStrengths[bestPlayerNumber])
           bestPlayerNumber = i; }
    int totalTimes = 0;    // Anzahl des durchgefuehrten Turniers
    foreach (var i in _winTimesTotal){
      totalTimes += i; }
    return (double)_winTimesTotal[bestPlayerNumber]/(double)totalTimes;
  }}
```

In der Klasse *Liga* wird dann die Turnierform Liga implementiert.

```
    private class Liga : GameModel
```

Hier wird die Methode *PlayGameOnce()* aus der Basisklasse *GameModel* überschrieben.

```
      public override void PlayGameOnce()
```

In der Methode wird zunächst ein Array angelegt, das speichert, wer in diesem Turnier wie vielmals gegenüber anderen Spielern gewinnt.

```
// Speichert, wer wie viel mal gewonnen hat
int[] winTimes = new int[_playerStrengths.Length];
// setzt die Anfangswerte zu 0
Array.Clear(winTimes, 0, winTimes.Length);
```

Durch zwei verschachtelte Schleife wird das in der Lösungsidee beschriebene Verfahren realisiert. Die äußere Schleife notiert den Index des aktuellen Spielers und die innere Schleife den Index des Gegenspielers.

```

for (int playerNumber = 0; playerNumber < _playerStrengths.Length - 1;
playerNumber++)
  for (int againstPlayerNum = playerNumber + 1; againstPlayerNum < _pla
yerStrengths.Length; againstPlayerNum++)
```

Dann wird mithilfe eines Zufallsgenerators den Gewinner zwischen diesen beiden Spielern festgestellt.

```
if (GetIntBetween(1, _playerStrengths[againstPlayerNum] +
    _playerStrengths[playerNumber]) <= _playerStrengths[playerNumber])
{
  // Falls der 1. Spieler gewinnt
  winTimes[playerNumber] += 1;
}
else
{
  // Falls der 2. Spieler gewinnt
  winTimes[againstPlayerNum] += 1;
}
```

Am Ende des Turniers wird der Sieger (mithilfe des Arrays *winTimes[]*) berechnet und das Attribute *winTimesTotal[]* an dem Index des Siegers um eins erhöht.

```
winTimesTotal[highstScorePlayer] += 1;
```

Die Klasse *KOx1* implementiert das K.O. System.

```
    private class KOx1 : GameModel
```

Das private Attribut *_playPlan* speichert den von dem Nutzer eingegebenen Spielplan.

```
      private int[] _playPlan;
```

In der überschriebene Methode *PlayGameOnce()* wird ein komplette Binärbaum aufgebaut und die äußeren Knoten(Blätter) mit dem Spielplan gefüllt (vgl. Lösungsidee-K.O. System, Abb.). Dann wird der Binärbaum in der Methode *GetGameResult()* deep-first traversiert. Am Ende hat man an der Wurzel des Baums den Sieger des Turniers. Wie bei dem Liga erhöht man das Attribute *winTimesTotal[]* an dem Index des Siegers um eins.

```
public override void PlayGameOnce()
{
  // Speichert den Spielplan in ein anderes Array initData
  // Welche zu einem Binaerbaum umgewandelt wird
  // der Spielplan befindet sich quasi in den aeusserste Knoten
  // vgl. Dokumentation
  int[] initData = new int[_playPlan.Length * 2 - 1];
  for (int i = 0; i < initData.Length; i++) initData[i] = -1;
  _playPlan.CopyTo(initData, _playPlan.Length - 1);
  Node<int> treeRoot = new BinaryTreeBuilder<int>(initData).Root;
  GetGameResult(treeRoot);
  _winTimesTotal[treeRoot.Data] += 1;
}
```

Danach wird das Turnier mit der Methode *GetGameResult()* durchgeführt. Diese Methode ist rekursiv, also ruft es sich innerhalb der Methode auf. Solange das aktuelle Knoten noch Nachkommen hat, wird den Wert der Nachkommen Knoten zunächst festgestellt, bevor man den Wert dieses Knotens feststellen kann. Falls das aktuelle Knoten kein Nachkommen mehr hat, wird dann den Wert des „Eltern-Knotens“ festgestellt, sodass die Knoten mit niedrigerer Höhe berechnet werden kann.

```
// Stellt den Sieger des Turniers fest
// Der Binaerbaum wird deep-first traversiert
private void GetGameResult(Node<int> currentNode){
  // Solange die jetzige Knote noch Nachkommen hat
  if (currentNode.RightChild != null){
    // Ruf sich recursiv auf fuer die Nachkommen Knoten
    GetGameResult(currentNode.LeftChild);
    GetGameResult(currentNode.RightChild);
    // Nachdem das Ergebnis der Nachkommen festgestellt wird,
    // kann das Ergebnis der jetzigen Knote bestimmt
    // indem man Zufallszahlen waehlt und vergleicht
    currentNode.Data = GetWinner(currentNode.LeftChild, current-
Node.RightChild);
  }
  // Falls die jetzige Knote kein Nachkommen mehr hat
  Else{
    // bestimmt man das Ergebnis zwischen dieser Knote und ih-
rer "Schwester-Knote"
    currentNode.Parent.Data = GetWinner(currentNode.Parent.Left-
Child, currentNode.Parent.RightChild);
  }
}
```

Die Methode *GetWinner()* stellt durch einen Zufallszahlengenerator fest, wer der beiden gegebenen Spieler gewinnt.

```
// Stellt fest, wer zwischen player1 und player2 diese Runde gewinnt
// und gibt die Nummer des gewonnenen Spielers zurueck
private int GetWinner(Node<int> player1, Node<int> player2)
{
  if (GetIntBetween(1, _playerStrengths[player1.Data] + _playerStrength
s[player2.Data]) <= _playerStrengths[player1.Data])
  {
    return player1.Data;
  }
  else return player2.Data;
}
```

Die Klasse *KOx5* ist großenteils mit der Klasse *KOx1* identisch. Der Unterschied liegt lediglich an der Methode *GetWinner()*, wo jetzt fünfmal Zufallszahlen generiert wird. Das Array *winningTimeCount[]* speichert, wer wievielmal in dieser Runde gewonnen hat. Der Spieler mit einer höheren Anzahl von Siegen wird als Gewinner zurückgegeben.

```
private int GetWinner(Node<int> player1, Node<int> player2)
{
  int[] winningTimeCount = new int[2];
  Array.Clear(winningTimeCount, 0, 2);
  for (int i = 0; i < 5; i++)
  {
    if (GetIntBetween(1, _playerStrengths[player1.Data] + _playerStreng
ths[player2.Data]) <= _playerStrengths[player1.Data]) winningTimeCount[
0]++;
    else winningTimeCount[1]++;
  }
  return winningTimeCount[0] > winningTimeCount[1] ? player1.Data : pla
yer2.Data;
}
```

In der *Main* Methode werden alle drei Klassen der Turnierformen instanziiert und das Turnier wird genügend viel (500.000 Mal) durchgeführt bei jeder Turnierform. Das Ergebnis werden dann in Console angezeigt.

```
// Das Turnier in den 3 Spielformen durchfuehren
int bestPlayerNum;
double procent;

KOx1 koX1 = new KOx1(playerStrength, spielPlan);
koX1.PlayGameNth(REPEAT);
procent = koX1.GetWinningProcentOfBestPlayer(out bestPlayerNum);
// Index ist 1 kleiner als die richtige Spielnummer
// also bestPlayerNum + 1 ,
// sodass anstatt des Indexes die Spielernummer gezeigt werden kann
Console.WriteLine($"KOx1: {bestPlayerNum + 1} - {procent} - {1d / pro-
cent}");

KOx5 koX5 = new KOx5(playerStrength, spielPlan);
koX5.PlayGameNth(REPEAT);
procent = koX5.GetWinningProcentOfBestPlayer(out bestPlayerNum);
Console.WriteLine($"KOx5: {bestPlayerNum + 1} - {procent} - {1d / pro-
cent}");

Liga liga = new Liga(playerStrength);
liga.PlayGameNth(REPEAT);
procent = liga.GetWinningProcentOfBestPlayer(out bestPlayerNum);
Console.WriteLine($"Liga: {bestPlayerNum + 1} - {procent} - {1d / pro-
cent}");
```

Hierbei ist *REPEAT* eine definierte Konstante, der in diesem Fall 500.000 beträgt.

```
    private const int REPEAT = 500000;
```

Beispiele

Erwartete Eingabe

Nach Anforderung des Programms sollte man den Namen der Test-Datei eingeben. Danach muss man noch den vorgesehenen Spielplan für die Turnierformen K.O. und K.O. x5 eingeben, indem man jede Nummer eines Spielers durch ein Komma trennt.

Bsp.

```
Bitte Name der Test-Datei eingeben...
spielstaerken1.txt      <- Erwartete Nutzereingabe
Geben Sie einen Spielplan ein...
Es kann wie Folgendes aussehen: 1,2,3,4,5,6,7,8
1,2,3,4,5,6,7,8         <- Erwartete Nutzereingabe
KOx1: 8 - 0,354532 - 2,8206198594203062
KOx5: 8 - 0,55467 - 1,8028737808066058
Liga: 8 - 0,347376 - 2,8787250702408915

Drueck eine beliebige Taste zu schliessen...
```

Ausgabeform

```
Spielform: der spielstärkste Spieler - Relative Häufigkeit, dass er gewinnt
- durchschnittliche Wiederholungen, die benötigt wird, sodass der
Spielstärkste Spieler das Turnier gewinnt
```

Bspw. heißt in der obigen Ausgabe, dass der Spieler 8 der spielstärkste Spieler ist und er bei der Turnierform KOx1 mit einer Häufigkeit von durchschnittlich 0,35516 gewinnt. Bei KOx5 beträgt diese Häufigkeit, dass er gewinnt, 0,555102 und bei Liga 0,347086.

Umgekehrt heißt der Kehrwert dieser relativen Häufigkeit die Anzahl der Wiederholungen, die durchschnittlich benötigt wird, bis der spielstärksten Spieler das Turnier gewinnt. Hierbei benötigt der spielstärksten Spieler bei der Turnierform KOx1 durchschnittlich 2,820620 Wiederholungen, bis er gewinnt, bei KOx5 dementsprechend 1,802874 und bei Liga 2,878725.

Spielstaerke1.txt

Spielplanvariante 1:

```
1,2,3,4,5,6,7,8
```

Ergebnis:

```
KOx1: 8 - 0,354532 - 2,8206198594203062
KOx5: 8 - 0,55467 - 1,8028737808066058
Liga: 8 - 0,347376 - 2,8787250702408915
```

Spielplanvariante 2:

```
1,8,2,7,3,6,4,5
```

Ergebnis:

```
KOx1: 8 - 0,477204 - 2,095539852976924
KOx5: 8 - 0,608256 - 1,6440446127946127
Liga: 8 - 0,347138 - 2,880698742286929
```

Spielplanvariante 3:

```
6,8,2,5,1,4,3,7
```

Ergebnis:

```
KOx1: 8 - 0,43268 - 2,311176851252658
KOx5: 8 - 0,693842 - 1,4412503134719432
Liga: 8 - 0,346362 - 2,88715274770327
```

Spielplanvariante 4:

```
1,7,2,3,4,5,6,8
```

Ergebnis:

```
KOx1: 8 - 0,330808 - 3,0229015017774663
KOx5: 8 - 0,517548 - 1,9321879323270499
Liga: 8 - 0,344972 - 2,8987859884280462
```

Spielstaerke2.txt

Spielplanvariante 1:

```
1,2,3,4,5,6,7,8
```

Ergebnis:

```
KOx1: 8 - 0,28133 - 3,554544485124231
KOx5: 8 - 0,362346 - 2,759793125907282
Liga: 8 - 0,2098 - 4,7664442326024785
```

Spielplanvariante 2:

```
1,8,2,7,3,6,4,5
```

Ergebnis:

```
KOx1: 8 - 0,307234 - 3,2548480962393485
KOx5: 8 - 0,36492 - 2,740326646936315
Liga: 8 - 0,210004 - 4,761814060684558
```

Spielplanvariante 3:

```
6,8,2,5,1,4,3,7
```

Ergebnis:

```
KOx1: 8 - 0,208252 - 4,801874651864088
KOx5: 8 - 0,223724 - 4,469793137973574
Liga: 8 - 0,208904 - 4,786887757055872
```

Spielstaerke3.txt

Spielplanvariante 1:

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
```

Ergebnis:

```
KOx1: 4 - 0,166428 - 6,008604321388228
KOx5: 4 - 0,27817 - 3,5949239673580906
Liga: 4 - 0,31591 - 3,165458516666139
```

Spielplanvariante 2:

```
1,16,2,15,3,14,4,13,5,12,6,11,7,10,8,9
```

Ergebnis:

```
KOx1: 4 - 0,15736 - 6,354855109303508
KOx5: 4 - 0,242986 - 4,115463442338242
Liga: 4 - 0,318064 - 3,144021329040696
```

Spielplanvariante 3:

```
1,7,2,12,3,14,16,15,13,6,4,5,10,8,9,11
```

Ergebnis:

```
KOx1: 4 - 0,149642 - 6,682615843145641
KOx5: 4 - 0,247482 - 4,040697909342901
Liga: 4 - 0,316316 - 3,1613955664588578
```

Spielplanvariante 4:

```
1,2,4,14,5,6,15,16,3,7,8,9,10,11,12,13
```

Ergebnis:

```
KOx1: 4 - 0,261146 - 3,829275577646221
KOx5: 4 - 0,456414 - 2,190993264886704
Liga: 4 - 0,316726 - 3,1573031579346185
```

Spielstaerke4.txt

Spielplanvariante 1:

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
```

Ergebnis:

```
KOx1: 1 - 0,069268 - 14,436680718369233
KOx5: 1 - 0,075014 - 13,33084490895033
Liga: 1 - 0,115126 - 8,686135191008113
```

Spielplanvariante 2:

```
1,16,2,15,3,14,4,13,5,12,6,11,7,10,8,9
```

Ergebnis:

```
KOx1: 1 - 0,06891 - 14,511681903932667
KOx5: 1 - 0,075444 - 13,25486453528445
Liga: 1 - 0,114846 - 8,70731240095432
```

Spielplanvariante 3:

```
1,7,2,12,3,14,16,15,13,6,4,5,10,8,9,11
```

Ergebnis:

```
KOx1: 1 - 0,069678 - 14,351732254083068
KOx5: 1 - 0,075832 - 13,187045046945881
Liga: 1 - 0,114316 - 8,747681864305958
```

Spielplanvariante 4:

```
1,2,4,14,5,6,15,16,3,7,8,9,10,11,12,13
```

Ergebnis:

```
KOx1: 1 - 0,069596 - 14,368641875969882
KOx5: 1 - 0,07511 - 13,313806417254694
Liga: 1 - 0,114518 - 8,732251698422957
```

Die Interpretierung der Beispiele befindet sich bereits in der Lösungsidee>Die Empfehlung der Turniervariante.

Quellcode

```
1.  using System;
2.  using System.IO;
3.  using System.Linq;
4.
5.  namespace Aufgabe3
6.  {
7.    public static class Turnier
8.    {
9.      private const int REPEAT = 500000;
10.     private static System.Random rng = new System.Random();
11.
12.     // Zufallszahlengenerator
13.     // i1 and i2 sind inclusive
14.     private static int GetIntBetween(int i1, int i2)
15.     {
16.       if (i1 < 0 || i1 > i2) throw new ArgumentException(nameof(i1));
17.       return rng.Next(i1, i2 + 1);
18.     }
19.
20.  Das Einlesen und die Ausgabe von Dateien werden hier nicht gezeigt
21.  Der vollständige Quellcode befindet sich in Aufgabe3->Quellcode->
    Aufgabe3.cs
22.
23.     private class Liga : GameModel
24.     {
25.       public Liga(int[] playerStrengths)
26.       {
27.         _playerStrengths = playerStrengths;
28.         _winTimesTotal = new int[playerStrengths.Length];
29.         // Anfangswert von 0
30.         Array.Clear(_winTimesTotal, 0, _winTimesTotal.Length);
31.       }
32.
33.       public override void PlayGameOnce()
34.       {
35.         // Speichert, wer wie viel mal gewonnen hat
36.         int[] winTimes = new int[_playerStrengths.Length];
37.         // setzt die Anfangswerte zu 0
38.         Array.Clear(winTimes, 0, winTimes.Length);
39.         // Array index = Spielernummer - 1
40.         for (int playerNumber = 0; playerNumber < _playerStrengths.Length -
    1; playerNumber++)
41.         {
42.           for (int againstPlayerNum = playerNumber + 1; againstPlayerNum < _
    playerStrengths.Length; againstPlayerNum++)
43.           {
44.             // Zufallszahl
45.             if (GetIntBetween(1, _playerStrengths[againstPlayerNum] + _playe
    rStrengths[playerNumber]) <= _playerStrengths[playerNumber])
46.             {
47.               // Falls der 1. Spieler gewinnt
48.               winTimes[playerNumber] += 1;
49.             }
50.             else
51.             {
52.               // Falls der 2. Spieler gewinnt
53.               winTimes[againstPlayerNum] += 1;
54.             }
55.           }
56.         }
57.         // Findet heraus, welcher Spieler am meisten gewonnen hat
58.         int highstScorePlayer = -1;
```

```
59.         int highstScore = -1;
60.         for (int i = 0; i < winTimes.Length; i++)
61.         {
62.           if (winTimes[i] > highstScore)
63.           {
64.             highstScorePlayer = i;
65.             highstScore = winTimes[i];
66.           }
67.           else if (winTimes[i] == highstScore)
68.           {
69.             highstScorePlayer = highstScorePlayer < i ? highstScorePlayer :
    i;
70.           }
71.         }
72.         // Erhoeht die Anzahl der Siege des Spielers um 1
73.         _winTimesTotal[highstScorePlayer] += 1;
74.       }
75.     }
76.
77.
78.     private class KOx1 : GameModel
79.     {
80.       private int[] _playPlan;
81.       public KOx1(int[] playerStrengths, int[] playPlan)
82.       {
83.         _playerStrengths = playerStrengths;
84.         _winTimesTotal = new int[playerStrengths.Length];
85.         // Set default val to 0
86.         Array.Clear(_winTimesTotal, 0, _winTimesTotal.Length);
87.         _playPlan = playPlan;
88.       }
89.
90.       public override void PlayGameOnce()
91.       {
92.         // Speichert den Spielplan in ein anderes Array initData
93.         // Welche zu einem Binaerbaum umgewandelt wird
94.         // der Spielplan befindet sich quasi in den aeusserste Knoten
95.         // alle andere Knoten haben einen Anfangswert von -1
96.         // vgl. Dokumentation
97.         int[] initData = new int[_playPlan.Length * 2 - 1];
98.         for (int i = 0; i < initData.Length; i++) initData[i] = -1;
99.         _playPlan.CopyTo(initData, _playPlan.Length - 1);
100.              Node<int> treeRoot = new BinaryTreeBuilder<int>(initData).Roo
    t;
101.              // Der Sieger dieser Runde wird dann in der Wurzel(Root) gesp
    eichert
102.              GetGameResult(treeRoot);
103.              // Erhoeht die Anzahl der Siege des gewonnenen Spielers um 1

104.              _winTimesTotal[treeRoot.Data] += 1;
105.            }
106.
107.            // Stellt den Sieger des Turniers fest
108.            // Der Binaerbaum wird deep-first traversiert
109.            private void GetGameResult(Node<int> currentNode)
110.            {
111.              // Solange die jetzige Knoten noch Nachkommen hat
112.              if (currentNode.RightChild != null)
113.              {
114.                // Ruf diese Methode recursiv auf fuer die Nachkommen
115.                GetGameResult(currentNode.LeftChild);
116.                GetGameResult(currentNode.RightChild);
117.                // Nachdem das Ergebnis der Nachkommen festgestellt wird,

118.                // kann das Ergebnis der jetzigen Knoten bestimmt
119.                // indem man Zufallszahlen waehlt und vergleicht
```

```
120.                 // vgl. GetWinner(...)
121.                 currentNode.Data = GetWinner(currentNode.LeftChild, current
    Node.RightChild);
122.               }
123.               // Falls die jetzige Knoten kein Nachkommen mehr hat
124.               else
125.               {
126.                 // bestimmt man das Ergebnis zwischen dieser Knoten und ihr
    er "Schwester-Knoten"
127.                 currentNode.Parent.Data = GetWinner(currentNode.Parent.Left
    Child, currentNode.Parent.RightChild);
128.               }
129.             }
130.
131.             // Stellt fest, wer zwischen player1 und player2 diese Runde ge
    winnt
132.             // und gibt die Nummer des gewonnenen Spielers zurueck
133.             private int GetWinner(Node<int> player1, Node<int> player2)
134.             {
135.               if (GetIntBetween(1, _playerStrengths[player1.Data] + _player
    Strengths[player2.Data]) <= _playerStrengths[player1.Data])
136.               {
137.                 return player1.Data;
138.               }
139.               else return player2.Data;
140.             }
141.           }
142.
143.           private class KOx5 : GameModel
144.           {
145.             private int[] _playPlan;
146.             public KOx5(int[] playerStrengths, int[] playPlan)
147.             {
148.               _playerStrengths = playerStrengths;
149.               _winTimesTotal = new int[playerStrengths.Length];
150.               // Set default val to 0
151.               Array.Clear(_winTimesTotal, 0, _winTimesTotal.Length);
152.               _playPlan = playPlan;
153.             }
154.             public override void PlayGameOnce()
155.             {
156.               // Speichert den Spielplan in ein anderes Array initData
157.               // Welche zu einem Binaerbaum umgewandelt wird
158.               // der Spielplan befindet sich quasi in den aeusserste Knoten

159.               // vgl. Dokumentation
160.               int[] initData = new int[_playPlan.Length * 2 - 1];
161.               for (int i = 0; i < initData.Length; i++) initData[i] = -1;
162.               _playPlan.CopyTo(initData, _playPlan.Length - 1);
163.               Node<int> treeRoot = new BinaryTreeBuilder<int>(initData).Roo
    t;
164.               GetGameResult(treeRoot);
165.               _winTimesTotal[treeRoot.Data] += 1;
166.             }
167.
168.             // Stellt den Sieger des Turniers fest
169.             // Der Binaerbaum wird deep-first traversiert
170.             private void GetGameResult(Node<int> currentNode)
171.             {
172.               // Solange die jetzige Knoten noch Nachkommen hat
173.               if (currentNode.RightChild != null)
174.               {
175.                 // Ruf sich recursiv auf fuer die Nachkommen Knoten
176.                 GetGameResult(currentNode.LeftChild);
177.                 GetGameResult(currentNode.RightChild);
```

```
178.                 // Nachdem das Ergebnis der Nachkommen festgestellt wird,
179.                 // kann das Ergebnis der jetzigen Knoten bestimmt
180.                 // indem man Zufallszahlen waehlt und vergleicht
181.                 // vgl. GetWinner(...)
182.                 currentNode.Data = GetWinner(currentNode.LeftChild, current
    Node.RightChild);
183.               }
184.               // Falls die jetzige Knoten kein Nachkommen mehr hat
185.               else
186.               {
187.                 // bestimmt man das Ergebnis zwischen dieser Knoten und ihr
    es "Schwester-Knoten"
188.                 currentNode.Parent.Data = GetWinner(currentNode.Parent.Left
    Child, currentNode.Parent.RightChild);
189.               }
190.             }
191.
192.             // Stellt fest, wer zwischen player1 und player2 diese Runde ge
    winnt
193.             // und gibt die Nummer des gewonnenen Spielers zurueck
194.             private int GetWinner(Node<int> player1, Node<int> player2)
195.             {
196.               int[] winningTimeCount = new int[2];
197.               Array.Clear(winningTimeCount, 0, 2);
198.               for (int i = 0; i < 5; i++)
199.               {
200.                 if (GetIntBetween(1, _playerStrengths[player1.Data] + _play
    erStrengths[player2.Data]) <= _playerStrengths[player1.Data]) winningTimeCou
    nt[0]++;
201.                 else winningTimeCount[1]++;
202.               }
203.               return winningTimeCount[0] > winningTimeCount[1] ? player1.Da
    ta : player2.Data;
204.             }
205.           }
206.
207.           private abstract class GameModel
208.           {
209.             protected int[] _winTimesTotal;
210.             protected int[] _playerStrengths;
211.             public abstract void PlayGameOnce();
212.             public void PlayGameNth(int n)
213.             {
214.               for (int i = 0; i < n; i++)
215.               {
216.                 PlayGameOnce();
217.               }
218.             }
219.             public double GetWinningProcentOfBestPlayer(out int bestPlayerN
    umber)
220.             {
221.               // Nummer des Spielers mit der hoechsten Spielstaerke
222.               bestPlayerNumber = 0;
223.               for (int i = 0; i < _playerStrengths.Length; i++)
224.               {
225.                 if (_playerStrengths[i] > _playerStrengths[bestPlayerNumber
    ]) bestPlayerNumber = i;
226.               }
227.
228.               // Anzahl des durchgefuehrten Turniers
229.               int totalTimes = 0;
230.               foreach (var i in _winTimesTotal)
231.               {
232.                 totalTimes += i;
233.               }
```

```
234.                 return (double)_winTimesTotal[bestPlayerNumber] / (double)tot
    alTimes;
235.               }
236.             }
237.
238.             private class Node<T>
239.             {
240.               public T Data { get; set; }
241.               public Node<T> LeftChild { get; set; }
242.               public Node<T> RightChild { get; set; }
243.               public Node<T> Parent { get; set; }
244.
245.               public Node(T data)
246.               {
247.                 this.Data = data;
248.               }
249.
250.             }
251.
252.             private class BinaryTreeBuilder<T>
253.             {
254.               public Node<T> Root { get; }
255.               public BinaryTreeBuilder(T[] dataArray)
256.               {
257.                 Root = Build(dataArray, 0, null);
258.               }
259.
260.               private Node<T> Build(T[] arr, int currentIndex, Node<T> parent
    Node)
261.               {
262.                 if (currentIndex < arr.Length)
263.                 {
264.                   Node<T> currentNode = new Node<T>(arr[currentIndex]);
265.                   currentNode.Parent = parentNode;
266.                   currentNode.LeftChild = Build(arr, currentIndex * 2 + 1, cu
    rrentNode);
267.                   currentNode.RightChild = Build(arr, currentIndex * 2 + 2, c
    urrentNode);
268.                   return currentNode;
269.                 }
270.                 return null;
271.               }
272.             }
273.           }
274.         }
```