

# Aufgabe 2: Nummernmerker

Team-ID: 00587

Team-Name: Doge.NET

Bearbeiter dieser Aufgabe:  
Nikolas Kilian

24. November 2019

## Inhaltsverzeichnis

<b>1</b>	<b>Lösungsidee</b>	<b>1</b>
1.1	Nochmmal in Worten . . . . .	2
<b>2</b>	<b>Umsetzung</b>	<b>2</b>
2.1	Datenstruktur . . . . .	2
2.2	Randinfo . . . . .	2
<b>3</b>	<b>Beispiele</b>	<b>2</b>
<b>4</b>	<b>Quellcode</b>	<b>3</b>

## 1 Lösungsidee

Um die optimale Aufteilung zu ermitteln, verwenden wir eine Variation des Knapsack-Algorithmus. Dieser funktioniert wie folgt:

```
1 TeileNummerAuf(nullstellen) {
2     if ( <Bereits für gleiche Parameter aufgerufen> ) {
3         return <Bereits errechnetes Ergebnis>;
4     }
5
6     if ( <Null Stellen bekommen> ) {
7         return [];
8     }
9
10    if ( <Zu wenig Stellen zum aufteilen> ) {
11        return <Fehler>;
12    }
13
14    for (int i from 2 to 4) {
15        subAufteilung = TeileNummerAuf(nullstellen.Skip(i));
16
17        if ( <subAufteilung Fehler produziert hat> ) continue;
18
19        Möglichkeiten.Add([i].Concat(subAufteilung));
20    }
21
22    return Möglichkeiten.Max(aufteilung => BewerteAufteilung(nullstellen, aufteilung));
```

```
23 }  
24  
25 BewerteAufteilung(nullstellen, aufteilung) {  
26     return <Anzahl an führenden Nullstellen in der Aufteilung>;  
27 }
```

Hierbei werden bereits errechnete Ergebnisse global gespeichert, sodass bei mehreren Rechnungen nacheinander die Ergebnisse der vorigen Durchläufe eventuell bei folgenden Rechnungen wiederverwendet werden können.

## 1.1 Nochmal in Worten

Der Algorithmus funktioniert also, indem einmal die ersten 2, 3 und dann 4 Ziffern der Zahl abgetrennt werden, und daraufhin die übrigen Ziffern noch einmal vom Algorithmus aufgeteilt werden. Jetzt, wo man 3 mögliche Aufteilungen hat, muss man nur noch die beste hiervon auswählen. Sind null Zahlen aufzuteilen, so wird eine leere Aufteilung zurückgegeben.

## 2 Umsetzung

Für die Umsetzung haben uns für eine Implementierung in C# 8.0 mit .NET Core 3.0 entschieden.

Der Sourcecode ähnelt stark dem Pseudocode (siehe Abschnitt 1); die zentrale Methode, die den Algorithmus ausführt, hat die Signatur

```
NummerMerkingSolution MerkNummern(ArraySegment<bool> zeros, int minSequenceLength  
= 2, int maxSequenceLength = 4).
```

Hierbei sind min-/maxSequenceLength die Minimal-/Maximallängen der einzelnen aufgeteilten Segmente; aus der Aufgabe heraus kommen hierbei für diese die Standardwerte 2 und 4.

### 2.1 Datenstruktur

Für das Speichern alter Ergebnisse wird ein struct `MerkedNumber` verwendet, welches die Eingaben für die Methode zwischenspeichert, und ein struct `NummerMerkingSolution`, welches die Ergebnisse zwischenspeichert. Diese werden in einem `System.Collections.Generic.Dictionary`2` aufeinander gemappt, sodass immer einer `MerkedNumber` eine `NummerMerkingSolution` zugeordnet ist.

### 2.2 Randinfo

Um Rechenzeit zu sparen, wird anders als im Pseudocode kein modifiziertes Array zurückgegeben, sondern eine Instanz des structs `System.ArraySegment`1`. Alle Instanzen dieses structs zeigen beim Ausführen auf das gleiche Array, womit unnötige Array-Allocations verhindert werden, was Kosten des Garbage Collectors spart.

## 3 Beispiele

Für mehr Info zu den Parametern des Programms führen sie `Nummernmerker.CLI --help` aus.

Ergebnisse für `Nummernmerker.CLI --file "examples/nummern.txt"`

Starting splitting of number 005480000005179734  
with segments of length 2..4

Digits: 18

Results:

Leading zeros hit: 2

Final distribution: 0054 8000 0005 1797

Starting splitting of number 03495929533790154412660  
with segments of length 2..4

Digits: 23

Results:

Leading zeros hit: 1

Final distribution: 0349 5929 5337 9015 441 26

Starting splitting of number 5319974879022725607620179  
with segments of length 2..4

Digits: 25

Results:

Leading zeros hit: 0

Final distribution: 5319 9748 7902 2725 6076 201

Starting splitting of number 9088761051699482789038331267  
with segments of length 2..4

Digits: 28

Results:

Leading zeros hit: 0

Final distribution: 9088 7610 5169 9482 7890 3833 12

Starting splitting of number 011000000011000100111111101011  
with segments of length 2..4

Digits: 30

Results:

Leading zeros hit: 3

Final distribution: 0110 0000 001 1000 1001 1111 110 10

## 4 Quellcode

Für ein bisschen Extraperformance verwenden wir `LinqFaster`; eine C# Library, welche einige LINQ Methoden für Arrays und Listen neu schreibt, um direkt passende Arrays/Listen zu generieren, anstatt nur `IEnumerableables` zu verwenden. `LinqFaster` Erweiterungsmethoden heißen genauso wie LINQ Methoden, mit einem F am Ende (bspw. `.WhereF` statt `.Where`).

```

1 public static NummerMerkingSolution MerkNummern(string text, int minSequenceLength, int
  maxSequenceLength) =>
2     MerkNummern(new ArraySegment<bool>(text.ToCharArray().SelectF(x => x == '0')),
  minSequenceLength, maxSequenceLength);
3
4 public static NummerMerkingSolution MerkNummern(in ArraySegment<bool> zeros, int
  minSequenceLength, int maxSequenceLength) =>
5     MerkNummern(new MarkedNumber(zeros, minSequenceLength, maxSequenceLength));
6
7 private static readonly Dictionary<MarkedNumber, NummerMerkingSolution> MarkedNumbers =
8     new Dictionary<MarkedNumber, NummerMerkingSolution>();
9
10 private static NummerMerkingSolution MerkNummern(in MarkedNumber markedNumber)
11 {
12     // If the input has already been processed once, return previous result.
13     if (MarkedNumbers.TryGetValue(markedNumber, out var optimalDistribution)) return
  optimalDistribution;
14
15     if (markedNumber.Zeros.Count == 0)
16     {
17         return MarkedNumbers[markedNumber] = NummerMerkingSolution.Empty();
18     }
19
20     // Not enough digits => Fail.
21     if (markedNumber.Zeros.Count < markedNumber.MinSequenceLength)
22     {
23         return MarkedNumbers[markedNumber] = NummerMerkingSolution.Failure();
24     }

```

```
25
26 int nextGenerationSize =
27     // Calculate the length of the longest segment possible.
28     Math.Min(
29         // Either the number of digits left,
30         merkedNummer.Zeros.Count,
31         // or the max length of distributions.
32         merkedNummer.MaxSequenceLength)
33     // Subtract min sequence length, as distributions smaller than it are not allowed.
34     - merkedNummer.MinSequenceLength
35     // Add one, as min is inclusive.
36     + 1;
37
38 if (nextGenerationSize <= 0)
39 {
40     return MerkedNummers[merkedNummer] = NummerMerkingSolution.Empty();
41 }
42
43 var nextGeneration = new NummerMerkingSolution[nextGenerationSize];
44
45 for (int i = 0; i < nextGenerationSize; i++)
46 {
47     int length = i + merkedNummer.MinSequenceLength;
48
49     var subSolution =
50         MerkNummern(
51             new ArraySegment<bool>(merkedNummer.Zeros.Array, merkedNummer.Zeros.Offset
52                                     + length, merkedNummer.Zeros.Count - length),
53             merkedNummer.MinSequenceLength,
54             merkedNummer.MaxSequenceLength);
55
56     nextGeneration[i] = !subSolution.IsSuccessful
57         ? NummerMerkingSolution.Failure()
58         : NummerMerkingSolution.Success(
59             subSolution.Distribution.PrependF(length),
60             subSolution.LeadngZerosHit + (merkedNummer.Zeros.ElementAtUnchecked(0) ? 1
61                                             : 0));
62 }
63
64 var elements = nextGeneration.WhereF(x => x.IsSuccessful);
65
66 if (elements.Length == 0)
67 {
68     return MerkedNummers[merkedNummer] = NummerMerkingSolution.Failure();
69 }
70
71 if (elements.Length == 1)
72 {
73     return MerkedNummers[merkedNummer] = elements[0];
74 }
75
76 NummerMerkingSolution bestSolution = elements[0];
77
78 for (int i = 1; i < elements.Length; i++)
79 {
80     // Find element with least leading zeros hit
81     if (elements[i].LeadingZerosHit < bestSolution.LeadngZerosHit)
82     {
```

```
81         bestSolution = elements[i];
82     }
83 }
84
85 return MerkedNummers[merkedNummer] = bestSolution;
86 }
```