

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 ( <Zu wenig Stellen zum aufteilen> ) {
7         return <Fehler>;
8     }
9
10    for (int i in 2..4) {
11        subAufteilung = TeileNummerAuf(nullstellen.Skip(i));
12
13        if ( <subAufteilung Fehler produziert hat> ) continue;
14
15        Möglichkeiten.Add([i].Concat(subAufteilung));
16    }
17
18    return Möglichkeiten.Max(aufteilung =>
19        BewerteAufteilung(nullstellen, aufteilung));
20 }
21 BewerteAufteilung(nullstellen, aufteilung) {
22     return <Anzahl an führenden Nullstellen in der Aufteilung>;
23 }
```

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

2 Umsetzung

Für die Umsetzung haben uns für eine Implementierung in C# 8.0 mit .NET Core 3.1 entschieden. Der Sourcecode ähnelt stark dem Pseudocode (siehe Lösungsidee); 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.

Für das speichern alter Ergebnisse wird ein struct *MerkedNummer* 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 *MerkedNummer* eine *NummerMerkingSolution* zugeordnet ist.

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

```
1 Starting splitting of number 005480000005179734 with segments of
  length 2..4
2   Digits:           18
3 Results:
4   Leading zeros hit: 2
5   Final distribution: 0054 8000 0005 1797
6
7 Starting splitting of number 03495929533790154412660 with segments of
  length 2..4
8   Digits:           23
9 Results:
10  Leading zeros hit: 1
11  Final distribution: 0349 5929 5337 9015 441 26
12
13 Starting splitting of number 5319974879022725607620179 with segments
  of length 2..4
14  Digits:           25
15 Results:
16  Leading zeros hit: 0
17  Final distribution: 5319 9748 7902 2725 6076 201
18
19 Starting splitting of number 9088761051699482789038331267 with
  segments of length 2..4
20  Digits:           28
21 Results:
22  Leading zeros hit: 0
23  Final distribution: 9088 7610 5169 9482 7890 3833 12
24
25 Starting splitting of number 011000000011000100111111101011 with
  segments of length 2..4
26  Digits:           30
27 Results:
28  Leading zeros hit: 3
29  Final distribution: 0110 0000 001 1000 1001 1111 110 10
```

4 Quellcode

```

1 private static readonly Dictionary<MarkedNumber,
  NummerMerkingSolution> MarkedNumbers =
2     new Dictionary<MarkedNumber, NummerMerkingSolution>();
3
4 public static NummerMerkingSolution MerkNummern(ArraySegment<bool>
  zeros, int minSequenceLength, int maxSequenceLength) =>
5     MerkNummern(new MarkedNumber(zeros, minSequenceLength,
  maxSequenceLength));
6
7 private static NummerMerkingSolution MerkNummern(MarkedNumber
  merkedNumber)
8 {
9     if (MarkedNumbers.TryGetValue(merkedNumber, out var
  optimalDistribution)) return optimalDistribution;
10
11     if (merkedNumber.Zeros.Count < merkedNumber.MinSequenceLength)
12     {
13         return MarkedNumbers[merkedNumber] =
  NummerMerkingSolution.Failure();
14     }
15
16     int nextGenerationSize =
17         Math.Min(
18             merkedNumber.Zeros.Count,
19             merkedNumber.MaxSequenceLength + 1)
20         - merkedNumber.MinSequenceLength;
21
22     if (nextGenerationSize <= 0)
23     {
24         return MarkedNumbers[merkedNumber] =
  NummerMerkingSolution.Empty();
25     }
26
27     var nextGeneration = new NummerMerkingSolution[nextGenerationSize];
28
29     for (int i = 0; i < nextGenerationSize; i++)
30     {
31         int length = i + merkedNumber.MinSequenceLength;
32
33         var subSolution =
34             MerkNummern(
35                 new ArraySegment<bool>(
36                     merkedNumber.Zeros.Array,
37                     merkedNumber.Zeros.Offset + length,
38                     merkedNumber.Zeros.Count - length),
39                     merkedNumber.MinSequenceLength,
40                     merkedNumber.MaxSequenceLength);

```

```

41
42     nextGeneration[i] = !subSolution.IsSuccessful
43         ? NummerMerkingSolution.Failure()
44         : NummerMerkingSolution.Success(
45             subSolution.Distribution.PrecedeOne(length),
46             subSolution.LeadingZerosHit
47             + (((IList<bool>)merkedNummer.Zeros)[0] ? 1 : 0));
48     }
49
50     var elements = nextGeneration.WhereF(x => x.IsSuccessful);
51
52     if (elements.Length == 0)
53     {
54         return MerkedNummers[merkedNummer] =
55             NummerMerkingSolution.Failure();
56     }
57
58     if (elements.Length == 1)
59     {
60         return MerkedNummers[merkedNummer] = elements[0];
61     }
62
63     NummerMerkingSolution bestSolution = elements.AggregateF((x, y) =>
64         x.LeadingZerosHit < y.LeadingZerosHit ? x : y);
65     return MerkedNummers[merkedNummer] = bestSolution;
66 }

```