1 Lösungsidee

Um die optimale Aufteilung zu ermitteln, verwende ich 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 =>
           BewerteAufteilung(nullstellen, aufteilung));
19 }
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, *NummerMerkingSolution MerkNummern(ArraySegment<bool> zeros, int minSequenceLength = 2, int maxSequenceLength = 4)*. Hierbei sind min-/maxSequenceLength die Minimal-/Maximallängen der einzelnen aufgeteilten Segmente.

Für das speicheren 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.Generics.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 Quellcode

```
1  private static readonly Dictionary<MerkedNummer,
       NummerMerkingSolution> MerkedNummers =
2      new Dictionary<MerkedNummer, NummerMerkingSolution>();
3
4  public static NummerMerkingSolution
       MerkNummern(ArraySegment<bool> zeros, int
       minSequenceLength, int maxSequenceLength) =>
5      MerkNummern(new MerkedNummer(zeros, minSequenceLength,
           maxSequenceLength));
6
7  private static NummerMerkingSolution MerkNummern(MerkedNummer
       merkedNummer)
8  {
9      if (MerkedNummers.TryGetValue(merkedNummer, out var
           optimalDistribution)) return optimalDistribution;
10
11     if (merkedNummer.Zeros.Count <
           merkedNummer.MinSequenceLength)
12     {
13         return MerkedNummers[merkedNummer] =
               NummerMerkingSolution.Failure();
14     }
15
16     int nextGenerationSize =
17         Math.Min(
18             merkedNummer.Zeros.Count,
19             merkedNummer.MaxSequenceLength + 1)
20         - merkedNummer.MinSequenceLength;
21
```

```
22      if (nextGenerationSize <= 0)
23      {
24          return MerkedNummers[merkedNummer] =
               NummerMerkingSolution.Empty();
25      }
26
27      var nextGeneration = new
           NummerMerkingSolution[nextGenerationSize];
28
29      for (int i = 0; i < nextGenerationSize; i++)
30      {
31          int length = i + merkedNummer.MinSequenceLength;
32
33          var subSolution =
34              MerkNummern(
35                  new
                       ArraySegment<bool>(merkedNummer.Zeros.Array,
                       merkedNummer.Zeros.Offset + length,
                       merkedNummer.Zeros.Count - length),
36                  merkedNummer.MinSequenceLength,
37                  merkedNummer.MaxSequenceLength);
38
39          nextGeneration[i] = !subSolution.IsSuccessful
40              ? NummerMerkingSolution.Failure()
41              : NummerMerkingSolution.Success(
42                  subSolution.Distribution.PrecedeOne(length),
43                  subSolution.LeadingZerosHit +
                       (((IList<bool>)merkedNummer.Zeros)[0] ? 1 :
                       0));
44      }
45
46      var elements = nextGeneration.WhereF(x => x.IsSuccessful);
47
48      if (elements.Length == 0)
49      {
50          return MerkedNummers[merkedNummer] =
               NummerMerkingSolution.Failure();
51      }
52
53      if (elements.Length == 1)
54      {
55          return MerkedNummers[merkedNummer] = elements[0];
56      }
```

```
57
58      NummerMerkingSolution bestSolution =
           elements.AggregateF((x, y) => x.LeadingZerosHit <
           y.LeadingZerosHit ? x : y);
59      return MerkedNummers[merkedNummer] = bestSolution;
60  }
```