1 Lösungsidee

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

```
TeileNummerAuf(nullstellen) {
    if ( <Bereits für gleiche Parameter aufgerufen> ) {
        return <Bereits errechnetes Ergebnis>;
    }

    if ( <Zu wenig Stellen zum aufteilen> ) {
        return <Fehler>;
    }

    for (int i in 2..4) {
        subAufteilung = TeileNummerAuf(nullstellen.Skip(i));

                if ( <subAufteilung Fehler produziert hat> ) continue;

        Möglichkeiten.Add([i].Concat(subAufteilung));
    }

    return Möglichkeiten.Max(aufteilung =>
       BewerteAufteilung(nullstellen, aufteilung));
}

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

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 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 Beispiele

```
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 01100000001100010011111101011 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

```
private static readonly Dictionary<MerkedNummer, NummerMerkingSolution> MerkedNummers =
    new Dictionary<MerkedNummer, NummerMerkingSolution>();

public static NummerMerkingSolution MerkNummern(ArraySegment<bool> zeros, int minSequenceLength, int maxSequenceLength) =>
    MerkNummern(new MerkedNummer(zeros, minSequenceLength, maxSequenceLength));

private static NummerMerkingSolution MerkNummern(MerkedNummer merkedNummer)
{
    if (MerkedNummers.TryGetValue(merkedNummer, out var optimalDistribution)) return optimalDistribution;

    if (merkedNummer.Zeros.Count < merkedNummer.MinSequenceLength)
    {
        return MerkedNummers[merkedNummer] = NummerMerkingSolution.Failure();
    }

    int nextGenerationSize =
        Math.Min(
            merkedNummer.Zeros.Count,
            merkedNummer.MaxSequenceLength + 1)
        - merkedNummer.MinSequenceLength;

    if (nextGenerationSize <= 0)
    {
        return MerkedNummers[merkedNummer] = NummerMerkingSolution.Empty();
    }

    var nextGeneration = new NummerMerkingSolution[nextGenerationSize];

    for (int i = 0; i < nextGenerationSize; i++)
    {
        int length = i + merkedNummer.MinSequenceLength;

        var subSolution =
            MerkNummern(
                new ArraySegment<bool>(
                  merkedNummer.Zeros.Array,
                  merkedNummer.Zeros.Offset + length,
                  merkedNummer.Zeros.Count - length),
                merkedNummer.MinSequenceLength,
                merkedNummer.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] =
                NummerMerkingSolution.Failure();
55      }
56
57      if (elements.Length == 1)
58      {
59          return MerkedNummers[merkedNummer] = elements[0];
60      }
61
62      NummerMerkingSolution bestSolution = elements.AggregateF((x, y) =>
            x.LeadingZerosHit < y.LeadingZerosHit ? x : y);
63      return MerkedNummers[merkedNummer] = bestSolution;
64  }
```