Die Lösungsidee:

Um die langen Zahlen in kleinere Blöcke zu zerlegen gibt es verschiedene Ansätze: der erste ist, einfach alle möglichen Blockkombinationen auszuprobieren und anschließend solche Kombinationen auszugeben, welche möglichst selten eine Null am Anfang eines Blocks aufweisen. Dies ist verhältnismäßig recht ineffektiv, funktioniert jedoch für die gegebene Maximallänge von 30 Ziffern auf einem neueren Computer in angemessener Laufzeit.

Es gibt jedoch auch einen Lösungsansatz, welcher letztlich beliebig lange Zahlen in linear ansteigender Laufzeit ziemlich gut aufteilen kann. Hierbei wird gewissermaßen abgeschätzt, wie sinnvoll es ist die noch übrigen Ziffern in Blöcke gewisser Längen aufzuteilen. Bleibt so zum Beispiel noch die Zahl 1110111, welche noch nicht zu Blöcken aufgeteilt wurde, so lässt sich sagen, dass als nächstes ein Block der Länge 2 oder 4 entstehen sollte, nicht aber der Länge 3, da dadurch im nächsten Durchlauf die Nummer 0111 übrig bleibt, welche mit einer Null beginnt. Dieses Verfahren hat jedoch einen Haken (zumindest in meiner Implementation): Wird auf diesem Weg einmal vermieden, dass der nächste Block mit einer Null beginnt, es dadurch aber anschließend zwei oder mehr Mal unmöglich wird keine Null am Anfang eines Blocks zu haben, so wäre es besser gewesen bei dem genannten Block eine Null am Anfang zu haben, als später zwei oder mehr Mal. Ein Beispiel hierzu:

10100001000100	wird zu einer Blockfolge mit zwei Blöcken mit Null am Anfang:
10 1000 0100 0100	obwohl es eine Lösung mit nur einem solchen Block gibt:
101 0000 1000 100	

Ein solcher Sonderfall kommt jedoch nicht in den Beispielen vor. Wie sich sehen lässt, benötigt eine Ziffer zudem auch einige Nullen, um einen solchen Effekt hervorzurufen. Für derartige Zahlen wäre folglich ein erweitertes Verfahren besser geeignet, bei welchem zum Beispiel größere Folgen von Nullen als Zahl mit einer Ziffer beschrieben werden. Die obige Zahl sähe dann aufgeteilt beispielsweise wie folgt aus:

101 4 1000 100

Daran, dass es eine einzelne Ziffer ist, ließe sich dann immer noch erkennen, dass es an der Stelle zum Beispiel 4 auf einander folgende Nullen sind. Sollte es mehr als 9 auf einander folgende Nullen geben, so wäre ein solches Verfahren verständlicherweise nicht denkbar, aber in dem Fall ist es auch schwer vorstellbar, dass es sich um eine Zahl handelt, die ein Mensch sich merken müsste.

Letzteres Verfahren habe ich nicht implementiert, da es nur dazu dienen soll zu zeigen, dass sich durch Abschätzung für „gewöhnliche“ Nummern ein gleichermaßen gutes Ergebnis erzielen lässt wie durch das stumpfe Ausprobieren und „ungewöhnlichere“ Nummern nicht vorkommen, da dies nicht dem Kontext der Aufgabe entsprechen würden.

Die Umsetzung erfolgt bei meinen Implementierungen der zuvor genannten Verfahren in beiden Fällen durch rekursive Methoden, welche je die bereits zu Blöcken aufgeteilten Ziffern sowie den noch übrigen Teil der Nummer übergeben bekommen. Das erste Verfahren hat jedoch hierdurch eine exponentiell ansteigende Laufzeit, da es alle möglichen Blockkombinationen ausprobiert, während das andere nur eine linear zunehmende Laufzeit hat, da jede Ziffer nur einmal in einen möglichst guten Block eingeteilt wird.

Die Umsetzung mit Quelltext:

Die Umsetzung der Lösungsidee erfolgt in meiner Bearbeitung in der Programmiersprache Ruby. Das Programm liegt im Ordner „Nummernmerker“ und stellt zeitgleich Quelltext und ausführbare Datei dar. Ich habe das Programm unter macOS 10.14.6 und der Ruby Version 2.3.7p456 (2018-03-28 revision 63024) [universal.x86_64-darwin18] getestet.

In beiden oben beschriebenen Versionen des Programms beginnt der Algorithmus erst in Zeile 6 und endet in Zeile 27 beziehungsweise 31. Die vorangehenden Zeilen dienen ausschließlich dem einlesen der Datei, während die darauf folgenden Zeilen ausschließlich den Aufruf des Ganzen sowie die Ausgabe darstellen. Nur die Methode „aufteilen“ wird deshalb im folgenden beschrieben:

Rekursiv ohne Annäherung/Abschätzung:

```
$bloecke = Array.new
def aufteilen(anfang, uebrig)
   if(uebrig == nil)
      $bloecke.push(anfang)
```

Die Methode erhält die bereits in Blöcke aufgeteilten Ziffern („anfang“ -> Array) sowie die noch übrigen Ziffern („uebrig“ -> String/char Array). Die Methode hat keinen Rückgabewert und speichert stattdessen, wenn alle Ziffern in Blöcke aufgeteilt wurden, diese Blöcke in dem Array „$bloecke“. Dabei ist zu beachten, dass durch die folgende Rekursion alle möglichen Blockkombinationen ausprobiert werden und diese allesamt in „$bloecke“ gespeichert werden.

```
   else
      if(uebrig.length < 3)
         aufteilen(anfang + [uebrig[0..1].join], nil)
      elsif(uebrig.length < 4)
         aufteilen(anfang + [uebrig[0..2].join], nil)
      elsif(uebrig.length < 5)
         aufteilen(anfang + [uebrig[0..1].join], uebrig[2..3])
         aufteilen(anfang + [uebrig[0..3].join], nil)
      elsif(uebrig.length < 6)
         aufteilen(anfang + [uebrig[0..1].join], uebrig[2..4])
         aufteilen(anfang + [uebrig[0..2].join], uebrig[3..4])
      else
         aufteilen(anfang + [uebrig[0..1].join], uebrig[2..-1])
         aufteilen(anfang + [uebrig[0..2].join], uebrig[3..-1])
         aufteilen(anfang + [uebrig[0..3].join], uebrig[4..-1])
      end
   end
end
```

Wenn nun bei einem Aufruf der Methode „uebrig“ nicht leer ist, wird kontrolliert, wie viele Ziffern noch verbleiben. Ausgehend davon wird dann entschieden, welche Blöcke gebildet werden können. Bleiben zum Beispiel noch zwei oder drei Ziffern, so kann nur ein Block aus zwei beziehungsweise drei Blöcken gebaut werden. Bleiben noch vier Ziffern könnte jedoch ein Block aus zwei oder vier, nicht aber aus drei Ziffern gebaut werden, da sonst genau eine Ziffer verbleiben würde, die im nächsten Durchlauf nicht zu einem regelkonformen Block gemacht werden könnte. Für den Fall, dass noch fünf Ziffern verbleiben, so lässt sich ein Block aus zwei oder drei, nicht aber aus vier Ziffern erschaffen, wobei die Begründung der vorherigen entspricht. Für alle anderen Fälle wird stets jede Möglichkeit ausprobiert - also sowohl einen zweier, dreier oder vierer Block als nächstes zu bilden. Das Bilden eines neuen Blockes und das Entfernen der dafür verwendeten Ziffern findet innerhalb der rekursiven Aufrufe statt.
Bei der Ausgabe werden aufgrund dessen, dass mehrere Lösungen in dieser Version des Programms berechnet werden, zunächst alle Ergebnisse mit nicht minimaler Anzahl an Blöcken mit Null am Anfang aussortiert. Anschließend lassen sich alle optimalen Lösungen ausgeben.

Rekursiv mit Annäherung/Abschätzung:

Die zweite Version, wie sie zuvor beschrieben steht, beginnt ähnlich wie die erste Variante:

```
$bloecke = Array.new
def aufteilen(anfang, uebrig)
   if(uebrig == nil)
      $bloecke = anfang
```

Dennoch gibt es hier bereits einen ersten Unterschied: im Array „$bloecke“ werden nicht mehr verschiedene Blockfolgen gespeichert, sondern nur die eine, welche durch Annäherung/Abschätzung erreicht wird.

```
   else
      if(uebrig.length < 5)
         aufteilen(anfang + [uebrig[0..-1].join], nil)
      elsif(uebrig.length < 6)
         if not uebrig[3] == "0"
            aufteilen(anfang + [uebrig[0..2].join], uebrig[3..4])
         else
            aufteilen(anfang + [uebrig[0..1].join], uebrig[2..4])
         end
      else
         if not uebrig[4] == "0"
            aufteilen(anfang + [uebrig[0..3].join], uebrig[4..-1])
         elsif not uebrig[3] == "0"
            aufteilen(anfang + [uebrig[0..2].join], uebrig[3..-1])
         elsif not uebrig[2] == "0"
            aufteilen(anfang + [uebrig[0..1].join], uebrig[2..-1])
         else
            aufteilen(anfang + [uebrig[0..3].join], uebrig[4..-1])
         end
      end
   end
end
```

Wie zuvor wird entsprechend dessen, wie viele Ziffern in „uebrig“ sind zunächst entschieden, wie vorgegangen werden muss. Wenn weniger als fünf Ziffern verbleiben, so werden diese direkt zu einem Block und die Methode wird rekursiv aufgerufen, wobei die letzten Ziffern zum letzten Block im Array „anfang“ werden und für „uebrig“ *nil* übergeben wird.
Verbleiben stattdessen fünf Ziffern, so könnte ein Block aus zwei oder drei Ziffern gebaut werden, wobei einer aus drei gebaut wird, falls die vierte übrige Ziffer keine Null ist und ansonsten wird eine Block aus zwei Ziffern gebildet. Sollte die dritte Ziffer eine Null sein entsteht hierbei ein Block mit einer Null, doch lässt es sich in dem Fall auch nicht mehr verhindern, weshalb nicht zusätzlich kontrolliert wird, ob die dritte Ziffer eine Null ist.
Sollten nun aber mindestens sechs Ziffern noch verbleiben, so könnte sowohl ein Block mit zwei, drei wie auch vier Ziffern gebildet werden. In diesem Fall wird zunächst kontrolliert, ob die fünfte Ziffer eine Null ist, ist dies der Fall, wird die vierte Ziffer kontrolliert und dementsprechend folgend die dritte Ziffer. Sollte die fünfte Ziffer keine Null sein wird im rekursiven Aufruf der Methode ein vierer Block als nächstes gebildet, ansonsten ein dreier, wenn die vierte Ziffer keine Null ist oder ein zweier Block, falls die dritte Ziffer keine Null ist. Sind sowohl die fünfte, vierte und dritte noch übrige Ziffer je eine Null, so wird ein vierer Block gebaut, da sich zwar ein Block mit Null am Anfang nicht vermeiden lässt, aber so zumindest eine möglichst große Anzahl an Ziffern bereits in einem Block landet, was die anschließende Anzahl an Durchläufen minimal verringert.

Durch diesen Algorithmus entsteht nur eine Lösung, welche anschließend in den darauf folgenden Zeilen ausgegeben wird.

Beispiel/Demonstration:

Im Ordner des Programms befindet sich die Datei „nummern.txt“ von der Materialseite, welche um die obige Nummer aus dem Beispiel (10100001000100) ergänzt ist. Die Ausgabe der Variante durch Abschätzen sieht wie folgt aus:

```
NIN0:2 nin0$ ruby abschaetzung.rb
0054-8000-0005-1797-34
Anzahl der Blöcke mit 0 am Anfang: 2

0349-5929-5337-9015-4412-660
Anzahl der Blöcke mit 0 am Anfang: 1

5319-9748-7902-2725-6076-201-79
Anzahl der Blöcke mit 0 am Anfang: 0

9088-7610-5169-9482-7890-3833-1267
Anzahl der Blöcke mit 0 am Anfang: 0

01-1000-0000-1100-0100-1111-1110-1011
Anzahl der Blöcke mit 0 am Anfang: 3

10-1000-0100-0100
Anzahl der Blöcke mit 0 am Anfang: 2
```

Die Ausgabe der ausschließlich rekursiven Variante ohne Abschätzung ist beachtlich länger, wenn man alle Blockkombinationen ausgeben lässt. Aus diesem Grund liegt sie neben der obigen Ausgabe im Ordner des Programms als .txt-Datei. Wie sich in dieser .txt-Datei sehen lässt, ist die letzte, nachträglich eingefügte Zahl, bei der ausschließlich rekursiven Variante optimal gelöst. Ich bitte zuletzt darum die .txt-Dateien im Ordner des Programmes wahrzunehmen.