- Für einen Blocksatz mit einer einheitlichen Farbe des Texts ist die Möglichkeit, Wörter bei Bedarf trennen zu können, essentiell.
- Einfache Trennungsregeln stehen nicht zur Verfügung oder liefern zuviele Falschtrennungen oder finden zuwenig Trennungsstellen.
- Die erste TEX-Implementierung von 1977 entfernte zunächst die Nachsilbe, die Vorsilbe und suchte dann nach Folgen von Vokal-Konsonant-Konsonant-Vokal und trennte dann zwischen den beiden Konsonanten. Damit werden jedoch nur ca. 40% der zulässigen Trennstellen erfasst und es bleiben Fehler wie etwa in „prog-ram“, die von längeren Ausnahmenlisten erfasst werden müssen.
- Einzelne Wörterbücher nennen umfassend Trennstellen (wie etwa Webster's oder der Duden). Diese Listen sind jedoch recht umfangreich und trotz ihres Umfangs nicht umfassend. Dies gilt insbesondere für Sprachen, die Zusammensetzungen erlauben und zahlreiche Beugungen (Flexionen) vorsehen wie etwa die deutsche Sprache.

- Gegeben sei eine Liste mit Trennstellen für einen umfangreichen Wortschatz. (Bei TEX wurde hier das *Merriam-Webster Pocket Dictionary* von 1974 mit ca. 50.000 Einträgen verwendet. Für die deutsche Sprache gibt es eine von Werner Lemberg gepflegte Liste mit 472.479 Einträgen, siehe `http://repo.or.cz/w/wortliste.git`.)
- Gesucht werden
  - ein sprachunabhängiges Regelsystem,
  - ein Verfahren, dass das Regelsystem für eine Sprache mit Hilfe von einer umfangreichen Trennliste automatisiert konfiguriert, so dass die Zahl der erzeugten Regeln möglichst minimal ist und
  - Datenstrukturen und Algorithmen, die mit geringem Laufzeit- und Speicheraufwand die möglichen Trennstellen eines Wortes finden, selbst wenn es nicht auf der Liste enthalten war.
- Ziel sollte es sein, dass die Mehrheit der korrekten Trennstellen gefunden wird und nur marginal wenige falsche Trennstellen geliefert werden, die notfalls mit Hilfe von Ausnahmelisten behandelt werden können.

- Das Regelsystem betrachtet nur vier aufeinanderfolgende Buchstaben $b_1 \ldots b_4$, um zu beurteilen, ob zwischen b_2 und b_3 getrennt werden kann.
- Da eine Tabelle mit $26^4 = 456976$ Einträgen zu umfangreich ist, gibt es stattdessen drei Tabellen für die Buchstabenpaare (b_1, b_2), (b_2, b_3) und (b_3, b_4), die jeweils die Wahrscheinlichkeiten angeben, dass hinter, zwischen und vor dem jeweiligen Buchstabenpaar in der gegebenen Wortliste getrennt wird.
- Die drei Werte aus den Tabellen werden miteinander multipliziert und es werden alle Trennstellen berücksichtigt, bei denen das Produkt einen vorgegebenen Mindestwert erreicht. (Hierbei wird ignoriert, dass die drei Wahrscheinlichkeiten nicht unabhängig voneinander sind.)
- Franklin Liang stellte in seiner Untersuchung jedoch fest, dass mit diesem Verfahren nur ca. 40% der Trennstellen gefunden werden bei einer Fehlerquote von 8%.

- Flexibler als die Wahrscheinlichkeitstabellen für die Buchstabenpaare sind Trennungsmuster, da sie beliebig lang sein können und auch gängige Vor- und Nachsilben berücksichtigen können.
- Beispiele für Trennungsmuster der englischen Sprache aus der Arbeit von Franklin Liang:

  .in-d .in-s .in-t .un-d b-s -cia con-s con-t e-ly erl-l er-m
  ex- -ful it-t i-ty -less l-ly -ment n-co -ness n-f n-l n-si n-v om-m
  -sion s-ly s-nes ti-ca x-p

  (Der Punkt repräsentiert jeweils den Anfang oder das Ende eines Worts.)
- Beispiele für die deutsche Sprache:

  .es-p .ob-l a-bl an-kl eu-e e-z ge-s g-q h-d h-h i-che i-d ll-b o-ra.

- Auch wenn Trennungsmuster viele Fälle korrekt erfassen, so bleibt doch eine signifikante Zahl von Fällen, bei denen falsch getrennt wird.
- Beispiel: Obwohl die Regel *-tion* in vielen Fällen zutrifft, darf „cation“ nicht getrennt werden.
- Wenn Ausnahmen nur über eine explizite Ausnahmeliste geregelt werden können, wird sie zu umfangreich.
- Deswegen ist es sinnvoll, auch Ausnahmeregeln in Form von Regeln zu formulieren. Als Ausnahmeregel würde sich hier *.cat* empfehlen.
- Das System lässt sich fortsetzen mit Ausnahmen der Ausnahmeregeln etc.

- Das Regelsystem besteht aus einer beliebigen Zahl von Trennungsmustern und einer Festlegung der Variablen *leftmin* und *rightmin*. (Die beiden Variablen geben den Mindestumfang eines Wortteils an, der vorne oder am Ende des Worts abgetrennt werden kann. In der englischen Sprache sind beide Werte 2, in der deutschen Sprache kann *leftmin* auf 1 gesetzt werden.)
- Jedes Trennungsmuster besteht aus Buchstaben, dem Punkt "." als Zeichen für den Wortanfang oder das Wortende und Ziffern, die Trennstellen bewerten.
- Ziffern repräsentieren den Rang einer Trennstelle in einer Regel. Ungerade Ränge (beginnend ab 1) befürworten eine Trennung, gerade Ränge (beginnend ab 2) stehen für eine Unterdrückung einer Trennstelle.
- Für ein zu trennendes Wort werden sämtliche zutreffenden Regeln berücksichtigt. Wenn sich mehrere Regeln für eine potentielle Trennregel einander widersprechen, dann setzt sich die Regel mit dem höheren Rang für die Trennstelle durch.

- Zu trennen ist das Wort „typography“. Folgende Trennungsmuster bei TEX treffen darauf zu (siehe Datei *hyphen.tex*):
  - *1ty*
  - *y3po*
  - *5po4g*
  - *1gr*
  - *4graphy*
  - *3raphy*
  - *1phy*
- Das Resultat ist „ty-pog-ra-phy“.
- Zu sehen ist hier, dass normalerweise durchaus vor „gr“ getrennt wird (4. Regel), dies in diesem konkreten Fall jedoch durch zwei Ausnahmeregeln unterbunden wird: *5po4g* und *4graphy* – in beiden Fällen schlägt der Rang 4 den Rang 1.
- Die vorgeschlagene Trennung entspricht genau der, die von Merriam-Webster vorgegeben wird.

- Jede Trennungsregel enthält mindestens einen Buchstaben.
- Entsprechend müssen bei einem Wort der Länge n insgesamt $\frac{n(n+1)}{2}$ Teilzeichenfolgen untersucht werden, was einem Aufwand von $O(n^2)$ entspricht.
- Dieser Aufwand multipliziert sich mit dem Aufwand, nach einer Regel für eine Zeichenfolge zu suchen.
- Das entspricht zumindest der einfachsten Vorgehensweise, die alle Regeln beispielsweise über eine *HashMap* zugänglich macht.
- Alternativ wäre es denkbar, einen endlichen Automaten für ein Regelsystem zu erzeugen. Dann reduziert sich der Aufwand auf $O(n)$. Allerdings wäre der Automat ziemlich umfangreich. (LuaTEX arbeitet inzwischen mit einem endlichen Automaten.)
- Eine weitere Alternative sind Tries.

- Der Name Trie leitet sich ab von *re-trie-val*. Die Datenstruktur wurde zuerst von Briandais (1959) und Fredkin (1960) beschrieben. Der Begriff geht auf Fredkin zurück.
- Anders als bei assoziativen Arrays (Maps) wird davon ausgegangen, dass Schlüssel sich definieren als eine Sequenz von Zeichen aus einem endlichen Alphabet.
- Entsprechend erfolgt der Zugriff zeichenweise. An jedem erreichten Punkt können wir dabei auf eine Regel stoßen, die Suche aber noch fortsetzen. Die Suche wird abgebrochen, sobald eine Zeichenfolge als Anfang eines Schlüssels nicht vorkommen kann.
- Tries können durch Baumstrukturen repräsentiert werden. Baumknoten können auf Regeln verweisen und die Zahl der möglichen Unterbäume zu einem Baumknoten ist nur durch den Umfang des Alphabets begrenzt.

Lesende Schnittstelle eines Tries

TrieReader.java

```
package de.uniulm.mathematik.tries;

public interface TrieReader<TrieInfo> {

   public interface TriePointer<TrieInfo> {
      public TriePointer<TrieInfo> descend(int code);
      public TrieInfo getInfo();
   }

   public TriePointer<TrieInfo> getRoot();
   public int getNumberOfEntries();
   public int getNumberOfNodes();
}
```

- Der Typparameter *TrieInfo* repräsentiert hier den Typ der durch die Datenstruktur auffindbaren Objekte (hier: Trennungsregeln).
- Eine Suche beginnt mit dem Aufruf von *getRoot*, das einen Zeiger (*TriePointer*) in die Datenstruktur liefert. An jedem Punkt lässt sich mit *getInfo* abfragen, ob ein Objekt zu finden ist und ein weiterer Abstieg ist mit *descend* möglich, das *null* zurückliefert, wenn die Zeichenfolge als Anfang eines Schlüssels nicht vorkommen kann.

Implementierung eines Tries

HashedTrie.java

```
public class TrieNode<TrieInfo> implements TrieIterator<TrieInfo> {
   protected TrieInfo info;
   protected HashMap<Integer, TrieNode<TrieInfo> > subnodes;
   protected TrieNode() {
      info = null;
      subnodes = new HashMap<Integer, TrieNode<TrieInfo> >();
   }
   public TrieNode<TrieInfo> descend(int code) {
      return subnodes.get(new Integer(code));
   }
   public TrieInfo getInfo() { return info; }
   public Iterator<Integer> iterator() {
      return subnodes.keySet().iterator();
   }
}
```

- Die einem Baumknoten untergeordneten Unterbäume werden hier mit einer *HashMap* verwaltet.
- Franklin Liang hat für die TEX-Implementierung eine komprimierte Datenstruktur (*packed tries*) entwickelt, die noch effizienter ist und insbesondere nur sehr wenig Speicherplatz benötigt. (Allerdings lässt sich das in Java nicht in gleicher Weise umsetzen.)

HashedTrie.java

```
public class HashedTrie<TrieInfo>
   implements TrieReader<TrieInfo>, TrieConstructor<TrieInfo> {

   // public class TrieNode<TrieInfo> ...

   private TrieNode<TrieInfo> root;
   private int numberOfNodes; private int numberOfEntries;

   public HashedTrie() {
      root = new TrieNode<TrieInfo>();
      numberOfNodes = 0; numberOfEntries = 0;
   }

   public TrieNode<TrieInfo> getRoot() {
      return root;
   }

   // public void insert(String word, TrieInfo info) ...

   public int getNumberOfEntries() { return numberOfEntries; }
   public int getNumberOfNodes() { return numberOfNodes; }
}
```

HashedTrie.java

```
public void insert(String word, TrieInfo info) {
   assert info != null;
   TrieNode<TrieInfo> node = root;
   for (int index = 0; index < word.length();
         index = word.offsetByCodePoints(index, 1)) {
      int ch = word.codePointAt(index);
      Integer key = new Integer(ch);
      TrieNode<TrieInfo> subnode = node.subnodes.get(key);
      if (subnode == null) {
         subnode = new TrieNode<TrieInfo>();
         node.subnodes.put(key, subnode);
         ++numberOfNodes;
      }
      node = subnode;
   }
   node.info = info;
   ++numberOfEntries;
}
```

HyphenationPoint.java

```
public interface HyphenationPoint {
   public int getPosition();
   public int getValue();
}
```

- Die Trennungsposition ist relativ zum Beginn der Trennungsregel und *getValue* liefert den zugehörigen Rang.

HyphenationRule.java

```
public interface HyphenationRule
   extends Iterable<HyphenationPoint> {
}
```

- Eine Trennungsregel besteht dann nur noch aus einer Folge von Trennungspositionen und zugehörigen Rängen.
- Die Buchstabenfolge der Regel gehört zum Schlüssel und wird in dieser Datenstruktur nicht wiederholt.

Hyphenator.java

```
public int[] hyphenate(String s) {
   // extract array of codepoints
   // where upper case letters are mapped to lower case
   String word = "." + s + ".";
   int len = word.codePointCount(0, word.length());
   int[] codepoints = new int[len];
   for (int i = 0; i < word.length(); i = word.offsetByCodePoints(i, 1)) {
      codepoints[i] = word.codePointAt(i);
      if (Character.isUpperCase(codepoints[i])) {
         codepoints[i] = Character.toLowerCase(codepoints[i]);
      }
   }
   // find matching hyphenation patterns and apply them to value[] ...
   // create array with hyphenation positions ...
}
```

- Ein zu trennendes Wort wird mit zwei Punkten ergänzt, damit auch die Trennregeln berücksichtigt werden, die sich nur auf den Anfang oder das Ende eines Worts beziehen.
- Das Eingabealphabet wird standardisiert, indem hier Groß- auf Kleinbuchstaben abgebildet werden. Es ist hier auch mehr denkbar wie etwa die Wegnahme von Akzenten.

Hyphenator.java

```
// find matching hyphenation patterns and apply them to value[]
int[] value = new int[len+1];
for (int start = 0; start+leftmin < len; ++start) {
   int pos = start;
   for (TrieReader.TriePointer<HyphenationEntry>
         ptr = patterns.getRoot();
         ptr != null;
         ptr = pos < len? ptr.descend(codepoints[pos++]): null) {
      HyphenationEntry rule = ptr.getInfo();
      if (rule != null) {
         for (HyphenationPoint point: rule) {
            int index = point.getPosition() + start - 1;
            if (index >= leftmin && index + rightmin < len-1) {
               int val = point.getValue();
               if (val > value[index]) {
                  value[index] = val;
               }
            }
         }
      }
   }
}
```

Hyphenator.java

```
// create array with hyphenation positions
int count = 0;
for (int i = 0; i <= len; ++i) {
   if (value[i] % 2 == 1) {
      ++count;
   }
}
if (count == 0) {
   return null;
}
int[] result = new int[count]; int index = 0;
for (int i = 0; i <= len; ++i) {
   if (value[i] % 2 == 1) {
      result[index++] = i;
   }
}
return result;
```

- Der Trennungsalgorithmus von Franklin Liang ist bis heute das wichtigste Trennungsverfahren, das auch außerhalb von TEX bei anderen Programmen wie etwa *troff* oder *QuarkXPress* eingesetzt wird.
- Verschiedene Verbesserungen wurden jedoch bzw. bereits umgesetzt (in Ω_2), aber von LuaTEX bislang nicht übernommen:
  - Trennungsregeln sollten Gewichtungen vorsehen, damit gute Trennstellen von weniger guten unterschieden werden können. In der deutschen Sprache sollte eher die Trennung „Trennungs-regel“ vor „Trennungsre-gel“ bevorzugt werden. (Dies könnte in den Strafwert der entsprechenden Sollbruchstelle einfließen.)
  - Bei gleich guten Trennungsstellen sollten die Trennungsstellen in der Mitte eines Worts bevorzugt werden.
  - Mehrdeutigkeiten sollten nicht getrennt werden: „Wach-stube“ vs. „Wachs-tube“.