Suche in Lempel-Ziv-codierten Texten

Werner Messner[1]

Ausgearbeitet nach einem Artikel von Martin Farach and Mikkel Thorup

17. März 1997

[1]messnerw@informatik.tu-muenchen.de

Einleitung

Da in unserer Zeit sehr viele Datenbestände anfallen, findet die Kompression von Daten eine weite Verbreitung, so daß es häufig vorkommt, daß man in komprimierten Datenbeständen nach bestimmten Schlüsselworten suchen muß. Die Standardlösung wäre hier sicher, die Datenbestände zu dekomprimieren, und anschließend den Text nach dem Schlüsselwort zu durchsuchen. Hierbei stellt sich die Frage, ob man nicht durch eine intelligentere Methode auch zum selben Ergebnis kommt, ohne alle Daten zu dekomprimieren. Man kann sich dabei die Verwandtschaft der Suchalgorithmen mit den Kompressionsalgorithmen zunutze machen. Es interessieren sich nämlich beide Verfahren für Wiederholungen. Die einen versuchen gewöhnlich Wiederholungen zu entdecken, während die anderen eben diese Wiederholungen eliminieren wollen - dies ist ja gerade die Kompression. Man erhöht die Entropie, indem man alle Wiederholungen eliminiert. Ein kleines, sehr einfaches Beispiel verdeutlicht diese Verwandtschaft:

$$(1.000.000''A'')(1.000.000''B'')$$

Es soll einen Text beschreiben, der aus einer Million Buchstaben "A", gefolgt von einer Million Buchstaben "B" besteht. Sucht man jetzt nach der Zeichenfolge "AB", so sieht man sofort, daß in den "A"s das "AB" bestimmt nicht zu finden ist, und genausowenig in den "B"s. Beim Schnittpunkt der beiden findet man allerdings die Zeichenfolge. Das Interessante an diesem Beispiel ist aber, daß es vom Zeitaufwand her völlig gleichgültig ist, ob wir Tausend, eine Million oder eine Milliarde Zeichen haben. Der Zeitaufwand für das Suchen im unkomprimierten Text wächst allerdings mit der Anzahl der "A"s und "B"s! Es gibt also eine schnellere Methode, um Muster in einem komprimierten Text zu finden. Da hier nur ein Algorithmus zur Suche in Lempel-Ziv kodierten Texten beschrieben wird, folgt zuerst eine kurze Beschreibung des LZ-1 [1].
Vorher noch ein paar Definitionen:

Klartext unkomprimierter Text

M zu suchendes Muster

u Länge des unkomprimierten Textes

n Anzahl der Datensätze [2] des LZ-1 kodierten Textes

Der LZ-1 Algorithmus

Der LZ1-Algorithmus ist relativ einfach zu verstehen. Die Grundidee, die hinter ihm steckt, ist die Annahme, daß Passagen im Text irgendwann später wieder an einer Stelle auftauchen werden. In diesem Fall wird dann an der zweiten und an späteren Stellen statt des eigentlichen Textes ein Verweis auf die erste Stelle gespeichert. Wenn diese Passagen recht lang sind und häufig auftreten, dann komprimiert dieses Verfahren recht gut. Der eben genannte Verweis sieht folgendermaßen aus: (P_i, L_i, C_i)

i ist ein Index

P_i gibt die Position des Textes an, auf den der Verweis zeigt

[1] auch LZ-77 genannt
[2] genaueres dazu folgt in der Beschreibung des LZ-1 — merken sollte man sich hier nur, daß n eine Maßzahl für die Länge des komprimierten Textes ist

L_i ist die Länge der Textpassage

C_i ist das direkt darauf folgende Zeichen. Dies ist notwendig, da der gesamte Text in das LZ1 Format umgewandelt werden muß und ganz zu Anfang noch keine kopierbaren Stellen zur Verfügung stehen.

Um den Aufbau der Verweise genauer zu erklären, betrachten wir folgendes Beispiel: "ABBABBA"
Im LZ1-Format sieht das so aus:
$\underbrace{(0,0,''A'')}_{A},\underbrace{(0,0,''B'')}_{B},\underbrace{(1,1,''A'')}_{BA},\underbrace{(2,2,''A'')}_{BBA}$
Also wenn ein Datensatz mit "(0,0," beginnt, so heißt das, daß das zu beschreibende Zeichen noch nicht im Text vorkam. Es muß also neu eingeführt werden, was mit C_i geschieht. Wenn der Datensatz mit z.B. "(3,2," beginnt, dann heißt dies, kopiere die 2 Zeichen, die sich 3 Positionen links von der jetzigen befinden. Dann folgt C_i, was wiederum angibt, welches einzelne Zeichen danach kommt.
Wir nehmen jetzt ein paar Modifikationen am Aussehen der Datensätze vor, damit der Suchalgorithmus leichter verständlich wird. Alle Datensätze sollen ein gleiches Erscheinungsbild haben, und es sollen keine Längen mit dem Wert 0 auftreten:
1. $(P_i, L_i, C_i) \rightarrow (P_i, L_i), (-C_i, 1)$
2. $(0, 0, C_i) \rightarrow (-C_i, 1)$
Es wurde noch etwas weiteres unternommen, das man hier aber noch nicht sieht. Alle Positionsangaben vor der Transformation waren relativ, also bezogen sich immer von der aktuellen Stelle im Text eine gewisse Anzahl von Zeichen nach links. Jetzt speichern wir die absoluten Werte in P_i.
Ein Beispiel soll dies näher verdeutlichen:

Klartext:	$\overbrace{cba}^{spezial}$	"$\overbrace{a}^{D_0}$	$\overbrace{b}^{D_1}$	$\overbrace{c}^{D_2}$	$\overbrace{ab}^{D_3}$	$\overbrace{abcab}^{D_4}$	$\overbrace{bab}^{D_5}$	$\overbrace{abcabb}^{D_6}$"
Position:	-3 -2 -1	0	1	2	34	56789	10.12	13.15..18
Komprimiert:		$\underbrace{(-1,1)}_{D_0}$	$\underbrace{(-2,1)}_{D_1}$	$\underbrace{(-3,1)}_{D_2}$	$\underbrace{(0,2)}_{D_3}$	$\underbrace{(0,5)}_{D_4}$	$\underbrace{(4,3)}_{D_5}$	$\underbrace{(5,6)}_{D_6}$

Wir haben jetzt wegen der Transformation vor dem Text zusätzlich die drei Zeichen "cba" mit den Positionen -3 bis -1 plaziert. Der eigentliche Text ist "abcab....abb". Die Zeile über dem unkomprimierten und dem komprimierten Text soll verdeutlichen, welcher Datensatz sich auf eine bestimmte Stelle im Text bezieht. Man sieht, daß das "a" an Pos. 0 durch (-1, 1), das "b" an Pos. 1 durch (-2, 1) dargestellt wird, usw. Betrachten wir den Datensatz D_5, so sehen wir, daß ab Pos. 4 die 3 Zeichen "bab" kopiert wurden. Jetzt sollte klar werden, warum man vor den eigentlichen Text zusätzlich die Zeichen "cba" angebracht hat. Es reichen die Positions- und Längenangaben vollständig aus, um den Text in den Datensätzen darzustellen. Den vorher zusätzlich benötigten dritten Wert C_i kann man sich durch die vorangestellte Buchstabenfolge "cba" sparen.
Wie lange die Zeichenfolge vor einem bestimmten Datensatz ist, wäre auch interessant zu wissen. Nehmen wir uns als Beispiel den Datensatz D_5 von vorher. Im Text sehen wir, daß D_5 bei Pos. 10 beginnt, aus (4, 3) geht dies jedoch nicht hervor. Aber man kann die Länge leicht ausrechnen als Summe der Längen aller Datensätze, die vorher aufgetreten sind. Hier sind dies 1+1+1+2+5 = 10 Zeichen, was man durch abzählen im unkomprimierten Text leicht nachprüfen kann. Als Formel: $U_i = \sum_{j=0}^{n} L_j$, wobei U_0=0

Die U_i geben an, wo ein bestimmter Datensatz im Klartext beginnt. In unserem Fall sehen sie so aus:
$U_0 = 0$
$U_1 = 1$
$U_2 = 2$
$U_3 = 3$
$U_4 = 5$
$U_5 = 10$
$U_6 = 13$
$U_7 = 19$
Als weitere vereinfachende Einschränkung verzichten wir auf Selbstreferenzen in den Datensätzen. **Selbstreferenzen** sind Verweise, die mehr kopieren, als eigentlich vorhanden ist. Das folgende Beispiel zeigt, wie dies zu verstehen ist:
"bba bba bba" läßt sich schreiben als: (Kodierung von **"bba"**), gefolgt von (0,6)
Dies funktioniert, obwohl nur 3 der geforderten 6 Zeichen bisher im Text vorhanden sind. Man kopiert die bekannten ersten drei Zeichen von Pos. 0-2 auf Pos. 3-5. An Pos. 3 angelangt steht man jetzt nicht mehr im Freien, sondern kann gleich weiterkopieren, da an Pos. 3-5 schon die neuen Zeichen stehen. Diese letzten drei Zeichen sind verantwortlich für den Begriff Selbstreferenzen.

Behauptung der P_i

Folgende Beobachtung erleichtert das Suchproblem ungemein:
Beobachtung: Wenn u die niedrigste Position im Klartext ist, an der das gesuchte Muster M auftritt, dann liegt M immer zwischen zwei Datensatzgrenzen.
Skizze:
Klartext: $\underbrace{T}_{D_0}\underbrace{TT}_{D_1}\underbrace{TTT}_{D_2}\underbrace{TTTT}_{D_3}\underbrace{TTMM}_{D_4}\underbrace{MMTT}_{D_5}\underbrace{TTTT}_{D_6}\underbrace{TTTT}_{D_7}$
Beweis:
Nehmen wir an, die Behauptung ist falsch und das erste Erscheinen des Musters M im Klartext (D_4) liegt **nicht** zwischen zwei Datensatzgrenzen, d.h. M liegt innerhalb eines Datensatzes. Dann merkt man sofort, daß diese Annahme falsch ist, denn ein Datensatz ist im LZ-1 immer nur eine Kopie von einer Textpassage, die vorher schon mal vorkam. $\rightarrow$ Das erste Muster im Klartext tritt immer nur zwischen zwei Datensätzen auf.
Diese Beobachtung kann man jetzt sehr gut nutzen. Man sucht nun an jeder Stelle U_i den längsten nach rechts wachsenden String $P^+(i)$, der im Muster enthalten ist, und den längsten nach links wachsenden String $P^-(i)$, der im Muster enthalten ist. Mit den so erhaltenen $P^+(i)$ und $P^-(i)$ ist man in der Lage festzustellen, ob das Muster im Text vorkommt oder nicht. Hierfür genügt es, die Verknüpfungen $P^-(i)P^+(i+1)$ für i=0,...,n-1 nach dem Muster zu durchsuchen. Kommt es vor, so hat man das erste Auftauchen des Musters im Klartext entdeckt —es kann allerdings zu späterer Stelle nochmal auftauchen. Kommt das Muster nicht vor, so ist es im gesamten Text nicht enthalten.

Informanten

Um die vorige Behauptung besser nutzen zu können, führen wir zu jedem Datensatz zwei neue Werte ein. Den linken Informanten l_i und den rechten r_i, die jeweils am äußersten rechten bzw. linken Rand des Datensatzes D_i liegen. Sie haben als Wert die Position, auf der sie sich befinden und berechnen sich dementsprechend folgendermaßen: $r_i = U_i + L_i - 1$; oder kürzer: $r_i = U_{i+1} - 1$; $l_i = U_i$;

Am Beispiel unseres Datensatzes:

Klartext:	$\overbrace{cba}^{spezial}$	"$\overbrace{a}^{D_0}$	$\overbrace{b}^{D_1}$	$\overbrace{c}^{D_2}$	$\overbrace{ab}^{D_3}$	$\overbrace{abcab}^{D_4}$	$\overbrace{bab}^{D_5}$	$\overbrace{abcabb}^{D_6}$"
Position:	-3 -2 -1	0	1	2	34	56789	10.12	13.15..18

$$\begin{aligned} l_0 &= 0; & r_0 &= 0; \\ l_1 &= 1; & r_1 &= 1; \\ l_2 &= 2; & r_2 &= 2; \\ l_3 &= 3; & r_3 &= 4; \\ l_4 &= 5; & r_4 &= 9; \\ l_5 &= 10; & r_5 &= 12; \\ l_6 &= 13; & r_6 &= 18; \end{aligned}$$

Nehmen wir zur Verbildlichung Datensatz D_5. Der linke Informant l_5 liegt auf dem "b" von "bab" und hat demzufolge als Wert die Position 10, während r_5 auf dem zweiten "b" von "bab" liegt, also auf Position 12. Wir wissen, daß dies richtig ist, da U_5=10 und $U_5 + L_5 - 1 = U_6 - 1 = 12$.

Linksschieben

Jetzt schieben wir alle Informanten nach links. Dies geschieht nach folgendem Schema:

L For s:= n-1 downto 0

L.1 Verschiebe alle Informanten, die sich im Bereich $[U_s, U_s + L_s - 1]$ befinden, um $U_s - P_s$ nach links.

L.2 Speichere dabei jeden bewegten Informanten in X_s.

Aus Zeile L ist ersichtlich, daß wir mit dem rechts außen liegenden Datensatz beginnen (n ist die Anzahl der Datensätze, und wir haben 0,...,6 = 7 Datensätze). Hier erkennt man schon den ersten Unterschied zur Standardmethode [3]. In Zeile L.1 verschieben wir die Informanten eines Bereiches. Der Bereich $[U_s, U_s + L_s - 1]$ kann auch anders als $[U_s, U_{s+1} - 1]$ geschrieben werden, und bezeichnet den Sektor im unkomprimierten Text, der von Datensatz Nummer D_s dargestellt wird.
In unserem Beispiel verschieben wir (beginnend bei s=6) den Informanten r_6 und l_6, da nur diese beiden in [13 ; 18] liegen
Nach Ablauf des Algorithmus stehen alle Informanten links von U_0, also auf negativer Position. Zusätzlich ist in X_s für jeden Schritt s gespeichert, welcher Informant bewegt wurde.
Dieses Verfahren reicht aus (Beweis siehe bei "Behauptung der P_i"), um nach Mustern der Länge $\leq$2 Zeichen zu suchen. Mehr ist für so kurze Muster nicht nötig und obwohl der Algorithmus noch sehr einfach ist, ist er bei gut komprimierten Texten um Klassen schneller als die Standardmethode. Er beschreibt auch genau den Fall, der zu Anfang in der Einleitung besprochen wurde (1 million mal "a" gefolgt von 1 million mal "b").

[3] Die Standardmethode Dekompression mit anschließender Suche im unkomprimierten Text beginnt von links und rollt den gesamten Text auf

Rechtsschieben

Für längere Muster braucht man natürlich eine etwas aufwendigere Methode, die auf dem Linksschieben aufbauend, die Informanten wieder zurck nach rechts schiebt. Dabei werden die vorher gewonnenen Informationen benutzt, um zu entscheiden, ob die in der Umgebung des Informanten befindlichen Zeichen interessant [4] sind und deshalb dieser Bereich genauer betrachtet werden muß, oder ob der Bereich uninteressant [5] ist und deshalb übergangen werden kann.
Man geht im Detail folgendermaßen vor:
Zuerst wird zu jedem Informanten r_i und l_i ein Textstring $P^+(i)$ und $P^-(i)$ eingeführt. Dieser Textstring $P^-(i)$ speichert für jeden Informanten l_i die längste nach links wachsende Passage des Klartextes, die im Muster vorkommt und analog dazu speichert der Textstring $P^+(i)$ für jeden Informanten r_i die längste nach rechts wachsende Passage des Klartextes, die im Muster vorkommt. Zum Aufbau der Textstrings $P^{+/-}$ gibt es zwei voneinander unabhängig bearbeitbare Algorithmen. Das Rechtsschieben der l_i und r_i.

Rechtsschieben der l_i

Algorithmus:

R.Li.1 Schreibe in jedes $P^-(i)$ den Buchstaben, auf dem der Informant l_i steht.

R.Li.2 s:=0 bis n-1

R.Li.2.1 Schiebe alle l_i, die sich in X_s befinden, um $U_s - P_s$ nach rechts. Schneide $P^-(i)$ links bei [6]U_s ab

R.Li.2.2 Wenn $P^-(i)$ zurück bis auf U_s geht, dann verlängere $P^-(i)$ folgendermaßen:

$$P^-(s-1)P^-(i)$$

R.Li.2.3 Vergleiche $P^-(i)$ mit dem Muster und kürze entsprechend von links

Erklärung:

R.Li.1 Sollte klar sein.

R.Li.2 Sollte klar sein.

R.Li.2.1 In X_s wurden alle Informanten gespeichert, die beim Linksschieben im Schritt s bewegt wurden. Es wird deshalb beim Rechtsschieben der gleiche Weg eingeschlagen, nur eben rückwärts. Die l_i werden um den selben Betrag nach rechts geschoben, wie sie vorher nach links geschoben wurden. Dabei wird jedoch alles, was nach links über die Grenze von U_s hinausgeht, abgeschnitten (weshalb dies erforderlich ist, sieht man im nächsten Punkt).

R.Li.2.2 Wenn $P^-(i)$ zurück auf Us geht, also der String welcher komplett im Muster enthalten ist bis zum Startpunkt des vorigen $P^-(i-1)$ geht, so sollte sichergestellt werden, ob der String über mehrere $P^-(i)$s hinweggeht. Hier sieht man auch, warum im vorigen Punkt (R.Li.2.1) alles was über die Grenze U_s hinausging abgeschnitten wurde. Der über U_s hinausgehende und im Muster enthaltene String steht in $P^-(i-1)$ und kann jetzt an $P^-(i)$ angehängt werden. (z.B. Suche nach "AliBaba" wobei $P^-(i)$="ab" bis zur Stelle $P^-(i-1)$="kaAliB" zurückgeht. In diesem Fall geht der gesuchte String

[4]also im Muster enthalten sind
[5]die Zeichen also nicht im Muster enthalten sind
[6]letzter Buchstabe, der noch zählt ist auf Position U_s

über zwei $P^-(i)$ hinweg, folglich muß $P^-(i)$ um $P^-(i-1)$ verlängert werden $\rightarrow$ $P^-(i)$ wird also um "kaAliB" zu "kaAliBab" verlängert. Ließe man diese Möglichkeit außer Acht, so müßte später das Muster nicht nur in allen Paaren $P^-(i-1)P^+(i)$ gesucht werden —also 2n Paare— sondern über die Paargrenzen aller $P^-(i)$ und $P^+(i)$ hinweg gesucht werden)

R.Li.2.3 Der Vergleich dient jetzt dazu, Ballast abzuwerfen. Es bringt ja nichts, läßt man alle Paare ewig nach links wachsen, obwohl nur ein kleiner Teil davon im Muster enthalten ist. (nach unserem vorigen Beispiel würde jetzt $P^-(i)$ so gekürzt werden, daß es gerade noch im Muster enthalten ist. Also von "kaAliBab" würden die Buchstaben "ka" abgeschnitten werden, da "kaAliBab" nicht in "AliBaba" vorkommt, so daß ein String übrigbleibt, der vollstndig im Muster enthalten ist. Als Ergebnis erhalten wir folglich "AliBab")

Am Ende des Algorithmus sind alle $P^-(i)$ bekannt.

Rechtsschieben der r_i

Algorithmus:

R.Ri.1 Schreibe in jedes $P^+(i)$ den Buchstaben, auf dem der Informant r_i steht.

R.Ri.2 s:=0 bis n-1

R.Ri.2.1 Schiebe alle r_i, die sich in X_s befinden, um $U_s - P_s$ nach rechts. Schneide $P^-(i)$ bei [7] $U_s + L_s - 1$ rechts ab

R.Ri.2.2 Verlängere alle $P^+(i)$, deren Informanten laut X_s nicht bewegt wurden und für die gilt, daß $P^+(i)$ genau bis $U_s - 1$ geht um $P^+(s)$

R.Ri.2.3 Vergleiche mit dem Muster und kürze entsprechend

Erklärung:

R.Ri.1 Sollte klar sein.

R.Ri.2 Sollte klar sein.

R.Ri.2.1 In X_s wurden alle Informanten gespeichert, die beim Linksschieben im Schritt s bewegt wurden. Es wird deshalb beim Rechtsschieben der gleiche Weg eingeschlagen, nur eben rückwärts. Die r_i werden um den selben Betrag nach rechts geschoben, wie sie vorher nach links geschoben wurden. Dabei wird jedoch alles, was nach rechts über die Grenze von Us+Ls-1 hinausgeht, abgeschnitten (weshalb dies erforderlich ist, sieht man im nächsten Punkt).

R.Ri.2.2 Hier muß ein anderer Weg eingeschlagen werden, als bei den l_i. Die l_i wachsen nach links, also in einen Bereich, der schon bekannt ist (da von links nach rechts geschoben wird ist der linke Bereich bekannt), während die r_i nach rechts in einen Bereich wachsen, der noch unbekannt ist. Es interessieren also für eine mögliche Verlängerung nicht die r_i, die im Schritt s bewegt wurden, denn diese wurden gerade nach rechts geschoben und wachsen folglich nicht in diesem Schritt, sondern es interessieren uns die r_i, die nicht verschoben wurden, denn sie können bis zu der Stelle reichen, die jetzt durch das Verschieben der r_i neu erforscht wurde. Wenn sie also bis zur neu erforschten Stelle reichen (bis zu $U_s + L_s - 1$), dann werden sie um $P^+(s)$ verlängert, also um den neu erforschten Bereich.

[7] letzter Buchstabe, der noch zählt ist auf Position $U_s + L_s - 1$

R.Ri.2.3 Jetzt muß natürlich wieder ein Vergleich mit dem Muster stattfinden und von rechts alle überflüssigen Zeichen abgeschnitten werden.

Am Ende steht in jedem $P^{+}(i)$ für jedes i der jeweils längste nach rechts wachsende String, der im Muster enthalten ist. Wie vorher schon erwähnt, genügt es nun, die Paare $P^{-}(i-1)P^{+}(i)$ nach dem Muster zu durchsuchen. Ist das Muster in $P^{-}(i-1)P^{+}(i)$ für 0<i<n nicht enthalten, so kommt das Muster im gesamten Text nicht vor.

Beschleunigung

Schnelles Linksschieben

Wie Sie vielleicht schon bemerkt haben, werden im vorher beschriebenen Algorithmus teilweise unnötige Arbeiten erledigt. Es werden z.B. alle r_i und alle l_i verschoben, auch wenn sie auf der gleichen Stelle stehen. (z.B. wenn r_3, r_4 und l_5 nach etlichen Verschiebungen auf der gleichen Stelle 10 im Bereich von U_5 (von z.B. 9 bis 11)stehen, werden sie im Schritt 5 alle einzeln wieder um P_5 nach links verschoben. Also landen sie wieder auf der selben Stelle, nur um P_5 weiter links) Es kann deshalb viel Aufwand gespart werden, verschiebt man nur noch einen Informanten von denen, die auf der gleichen Stelle stehen, wobei die anderen Informanten natürlich als Duplikate gespeichert werden müssen. Der Algorithmus hierfür ist einfach.

A.1 Alle Informanten r_i und l_i werden getrennt in einer Datenstruktur, die im folgenden LIST genannt wird, gespeichert. Die Informanten sind in LIST nach ihrer Position aufsteigend geordnet. Als Datenstruktur für LIST bietet sich hier der "Balanced Tree" besonders an, da man mit ihm sehr schnell eine bestimmte Position finden kann.

A.1.1 Schneide LIST bei Position U_s in LIST0 und LIST1, wobei in LIST0 nur noch Informanten stehen, deren Position $< U_s$ ist und in LIST1 nur Informanten stehen, deren Position $\geq U_s$ ist.

A.1.2 Schiebe alle Informanten von LIST1 um U_s-P_s nach links.

A.1.3 Verschmelze LIST0 mit LIST1 zu LIST, und eliminiere dabei alle Duplikate, d.h. speichere für jedes Duplikat einen Verweis auf den Informanten, der nicht eliminiert wurde und lösche die Duplikate aus LIST. (Als Datenstruktur hierfür bietet sich z.B. eine Liste an. Für jeden Informanten, der bei der Duplikatelimination übrigbleibt, wird eine erstellt, in der alle eliminierten Informanten und der Schritt s, in dem sie eliminiert wurden, gespeichert wird.)

Der Verschmelzungsalgorithmus ist bei diesem Verfahren der zeitaufwendigste Teil. Ansonsten spart man sich besonders bei großen gut komprimierten Dateien sehr viel Aufwand, da dort besonders viele Informanten mehrfach verschoben werden.

Segment-Merge

Als Verschmelzungsalgorithmus bietet sich (nach Martin Farach und Mikkel Thorup) besonders der Segment-Merge an.
Seg-Merge(LIST0,LIST1)

B.1. Ist eine Liste leer, so gib die andere zurück

B.2. x0=head(LIST0)
x1=head(LIST1)

B.3. Wenn die Position von x0 = Position von x1 dann gib
x0::Seg-Merge(tail(LIST0),tail(LIST1)) [8] zurück

B.4. Wenn Pos. von x0 < Pos. von x1 dann schneide LIST0 bei Pos.(x1) in LIST01 und LIST02 und gib LIST01::Seg-Merge(LIST02,LIST1) zurück

B.5. Wenn Pos. von x1 < Pos. von x0 dann schneide LIST1 bei Pos.(x0) in LIST11 und LIST12 und gib LIST11::Seg-Merge(LIST12,LIST0) zurück

Weitere Beschleunigungsverfahren

Als weitere Maßname führen Martin Farach und Mikkel Thorup ein Bucketing Argument ein, das durch Transformation der Datensätze des LZ-1 in mehrere Datensätze beschränkter Länge eine weitere Beschleunigung des Segment-Merge bewirkt und es wird gezeigt, wie man mit Selbstreferenzen umgehen kann. Außerdem werden für besonders lange Muster zwei randomisierte Algorithmen angegeben (Monte Carlo und Las Vegas) auf die ich jedoch nicht weiter eingehen möchte.

[8] x0::Seg-Merge(...) heißt: x0 konkateniert mit Seg-Merge(...)