ADS: Algorithmen und Datenstrukturen 2

Teil V

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University of Leipzig**

04. Mai 2011

Datenkompression

Bisheriges Hauptziel bei der Formulierung von Algorithmen

- schnelle Verarbeitung von Daten
- Effizienz bei der *Zeit*-Komplexität

Jetzt:

- *Speicher*-effiziente *Kodierung* (Darstellung)
- Datenkompression

Datenkompression

Unterscheidung von zwei Typen der Kompression

- verlustbehaftet (“lossy”): Originaldaten können i.A. nicht eindeutig aus der komprimierten Codierung zurückgewonnen worden. Beispiele: JPEG, MPEG (mp3)
- verlustlos (“lossless”): Codierung eindeutig umkehrbar.

Aussprache von www

Sprache	direkt	codiert
Deutsch	we-we-we	drei mal w
Englisch	double-u double-u double-u	three times double-u
Dänisch	dobbelt-ve dobbelt-ve dobbelt-ve	tre gange dobbelt-ve
Spanisch	uve doble uve doble uve doble	tres uve dobles

Kompressionsverfahren: Übersicht

- Lauflängenkodierung: Aufeinanderfolgende identische Zeichen werden zusammengefasst.
- Huffman-Kodierung: Häufiger auftretende Zeichen werden mit weniger Bits codiert.
- LZW, LZ77, gzip: Mehrfach auftretende Zeichenfolgen werden indiziert (“Wörterbuch”) bzw. zu einzelnen Zeichen zusammengefasst.

Allgemeines Prinzip: Ausnutzen von Redundanz = Abweichungen von zufälligen Datensequenzen.

Lauflängencodierung I

- Einfachster Typ von Redundanz: *Läufe* (runs) = lange Folgen sich wiederholender Zeichen.
- Beispiel: betrachte die folgende Zeichenkette AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD.
- Ersetze Lauf durch Angabe der Länge und des sich wiederholdenden Zeichens: 4A3B2A5B8CDABCB3A4B3CD
- Lohnt sich nur für Läufe mit Länge > 2

Lauflängencodierung II

Für binäre Dateien, also Sequenzen von Bitwerten $\in \{0, 1\}$:

- Ausnutzen, dass sich Läufe von 0 und 1 abwechseln
- Original: 000111011111011000111111
- Kodierung: 3 3 1 4 1 2 3 5

Lauflängencodierung III

- Wie kann man erreichen, dass der gesamte Zeichenvorrat (inclusive Ziffern) in der Eingabe zulässig ist?
- Deklariere als sog. *Escape-Zeichen* ein Zeichen Q, das in der Eingabe wahrscheinlich nur selten auftritt.
- Jedes Auftreten dieses Zeichens besagt, dass die folgenden beiden Buchstaben ein Paar (Zähler, Zeichen) bilden.
- Ein solches Tripel wird als *Escape-Sequenz* bezeichnet.
- Zählerwert *i* wird durch i-tes Zeichen des Alphabets dargestellt.
- Beispiel: betrachte die folgende Zeichenkette AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD.
- Codierung: QDABBBAAQEBQHCDABCBAAAQDBCCCD
- Codierung erst für Läufe ab Länge 4 sinnvoll.
- Auftreten des Escape-Zeichens Q selbst in der Eingabe wird durch Q<Leerzeichen> codiert.

Codierung mit variabler Länge I

- In einem Text treten Buchstaben mit unterschiedlicher Häufgkeit auf, z.B. E häufiger als Y (im Deutschen). Versuche, dies zur Kompression auszunutzen.
- Beispieltext: ABRACADABRA
- Standardcodierung (unkomprimiert) mit 5 Bits pro Zeichen: 00001000101001000001000110000100100000010001010010000001
- Häufgkeit der Buchstaben:

A	B	C	D	R
5	2	1	1	2

- Versuche, für häufige Zeichen möglichst kurze Bitsequenz zu finden, für seltene dafür längere, so dass die Gesamtzahl verwendeter Bits minimiert wird.
- Trotz unterschiedlich langer Bitfolgen muss erkennbar bleiben, wann die Bits eines Zeichens enden, und wann das nächste anfängt.

Codierung mit variabler Länge II

Präfixcode (Präfix-freier Code)

- Kein Codewort ist Präfix eines anderen Codeworts.
- Codierung eindeutig umkehrbar.

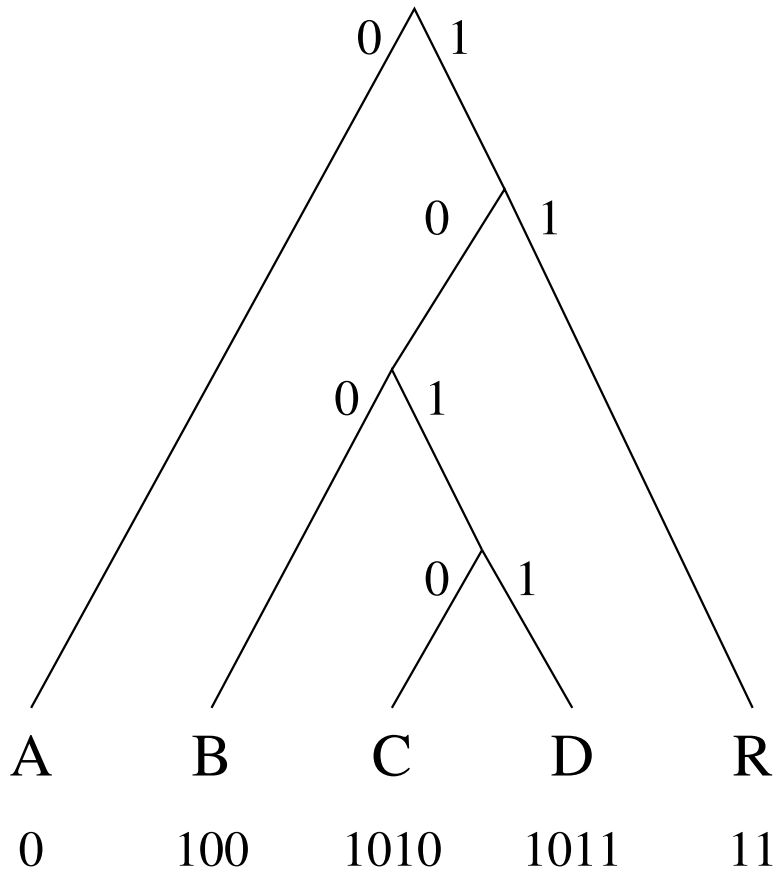
Buchstabe	A	B	C	D	R
Häufigkeit	5	2	1	1	2
Code	0	100	1010	1011	11

- `01001101010010110100110`
  `AB  R AC   AD   AB  R A`

Codierung mit variabler Länge IV

Binärer Präfixcode lässt sich durch Binärbaum darstellen:

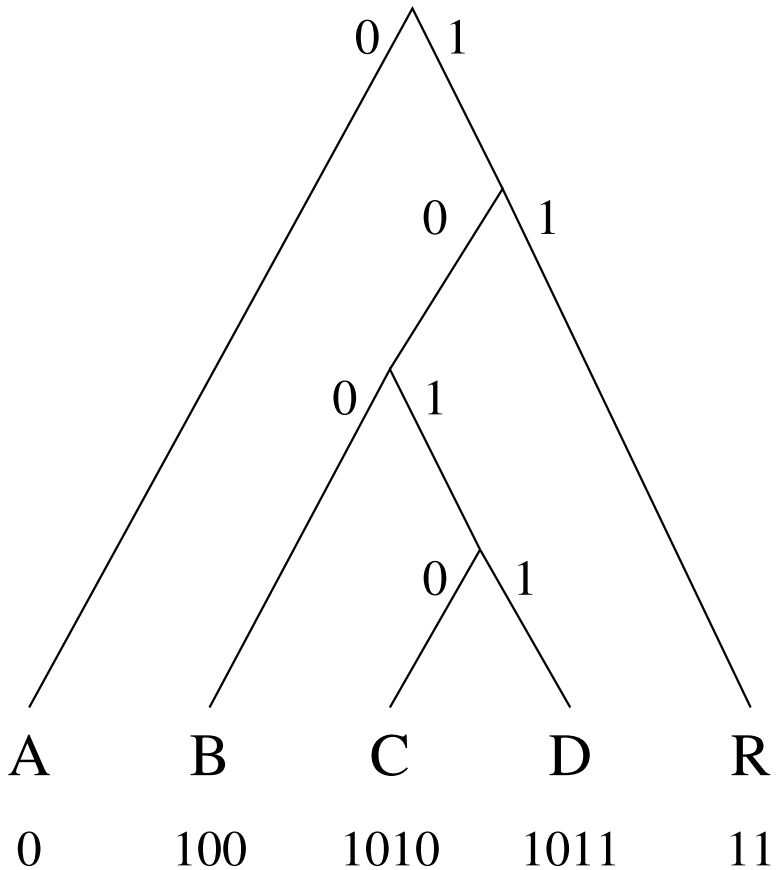
- Blätter sind zu codierende Zeichen
- Zur Codierung eines Zeichens wird Pfad von Wurzel zum Blatt des Zeichens abgelaufen.
- Ausgabe 0 bei Verzweigung nach links, 1 bei Verzweigung nach rechts.
- Binärbaum wird sinnvollerweise strikt gewählt.
- Andere Bezeichnung: Trie (dazu später mehr)



Decodierung

Binärer Präfixcode lässt sich durch Binärbaum darstellen:

1. Beginne bei Wurzel im Baum
2. Lies nächstes Bit, bei 0 verzweige nach links im Baum, sonst nach rechts.
3. Falls erreichter Knoten ein Blatt ist, gib das Zeichen des Blatts aus und springe zurück zur Wurzel.
4. Falls Eingabeende noch nicht erreicht, gehe zu 2.



Erzeugung des Huffman-Codes

Das allgemeine Verfahren zur Bestimmung dieses Codes wird Huffman-Codierung genannt. Der erste Schritt bei der Erzeugung des Huffman-Codes besteht darin, durch Zählen die Häufigkeit jedes Zeichens innerhalb der zu codierenden Zeichenfolge zu ermitteln. Das folgende Programm ermittelt die Buchstaben-Häufigkeiten einer Zeichenfolge a und trägt diese in ein Feld count[26] ein. Die Funktion index dient hier dazu, dass der Häufigkeitswert für den i-ten Buchstaben des Alphabets in dem Eintrag count[i] eingetragen wird, wobei wie üblich der Index 0 für das Leerzeichen verwendet wird.

```
for ( i = 0 ; i <= 26 ; i++ ) count[i] = 0;
  for ( i = 0 ; i < M ; i++ ) count[index(a[i])]++;
```

Beispiel und Algorithmus

A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

Die dazugehörige Häufigkeits-Tabelle

	_	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
count[k]	11	3	3	1	2	5	1	2	0	6	0	0	2	4	5	3	1	0	2	4	3	2	0	0	0	0	0

Es sind 11 Leerzeichen, drei A, drei B usw. Der nächste Schritt ist der Aufbau des Codierungs-Tries entsprechend den Häufigkeiten. Während der Erzeugung des Trie betrachtet man ihn als binären Baum mit Häufigkeiten, die in den Knoten gespeichert sind; nach seiner Erzeugung betrachtet man ihn dann als einen Trie für die Codierung in der oben beschriebenen Weise. Für jede von Null verschiedene Häufigkeit wird ein Knoten des Baumes erzeugt.

Algorithmus II

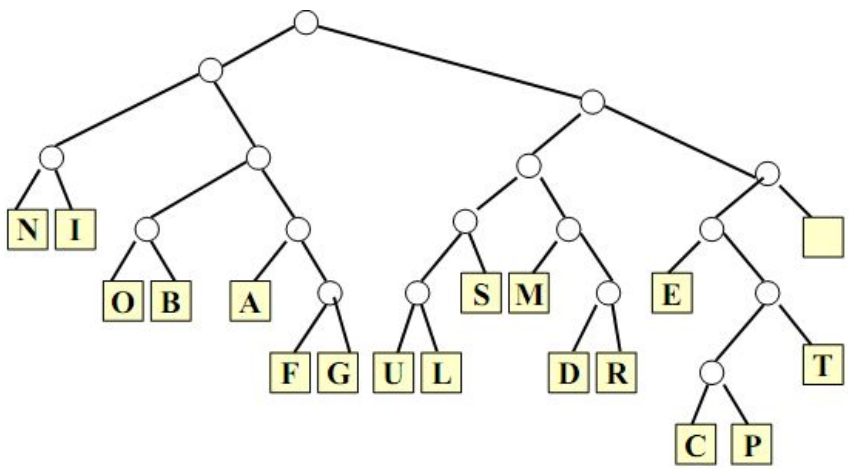
Dann werden die beiden Knoten mit den kleinsten Häufigkeiten ausgewählt und es wird ein neuer Knoten erzeugt, der diese beiden Knoten als Nachfolger hat und dessen Häufigkeit einen Wert hat, der gleich der Summe der Werte für seine Nachfolger ist. Danach werden die beiden Knoten mit der kleinsten Häufigkeit in diesem Wald ermittelt, und ein neuer Knoten wird auf die gleiche Weise erzeugt. Am Schluss sind alle Knoten miteinander zu einem einzigen Baum verbunden. Man beachte, dass sich am Ende Knoten mit geringen Häufigkeiten weit unten im Baum befinden, Knoten mit großen Häufigkeiten in der Nähe der Wurzel des Baumes.

Algorithmus III

Nunmehr kann der Huffman-Code abgeleitet werden, indem die Häufigkeiten an den unteren Knoten einfach durch die zugehörigen Buchstaben ersetzt werden und der Baum dann als ein Trie für die Codierung angesehen wird, wobei genau wie oben “links” einem Bit 0 und “rechts” einem Bit 1 im Code entspricht. Der Code für N ist 000, der Code für I ist 001, der Code für C ist 110100 usw. Die kleine Zahl oberhalb jedes Knotens in diesem Baum ist der Index für das Feld count, der angibt, wo die Häufigkeit gespeichert ist. Diese Angabe benötigt man, um sich bei der Untersuchung des Programms, das den untenstehenden Baum erzeugt, darauf beziehen zu können. Folglich ist für dieses Beispiel count[33] gleich 11, der Summe der Häufigkeitszähler für N und I.

Trie für die Huffman-Codierung von “A SIMPLE STRING ...”

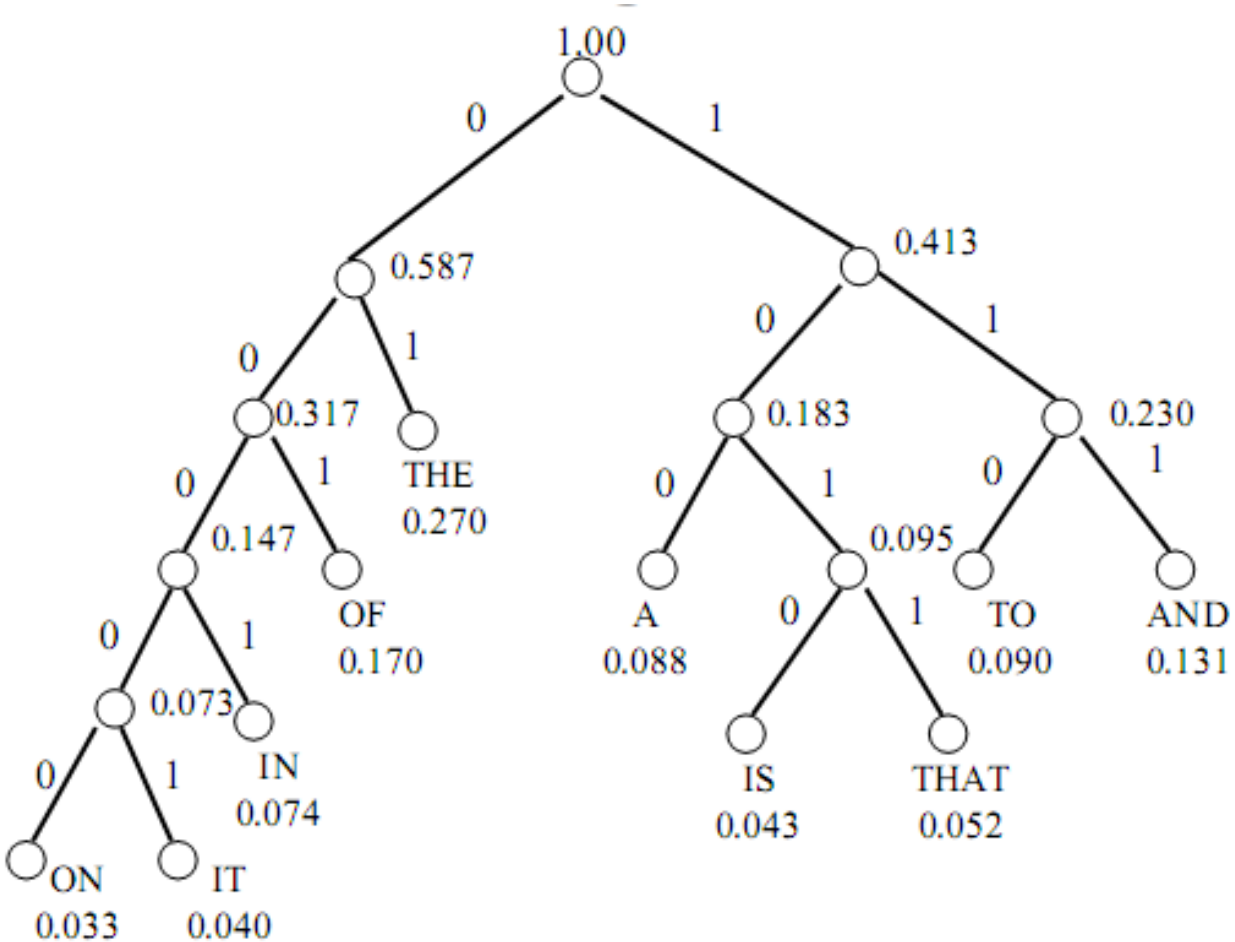
	_	A	B	C	D	E	F	G	I	L	M	N	O	P	R	S	T	U
k	0	1	2	3	4	5	6	7	9	12	13	14	15	16	18	19	20	21
count[k]	11	3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2



Huffman-Codierungs-Baum für Wörter aus der Englischen Sprache

Codierungs-Einheit	Wahrscheinlichkeit des Auftretens	Code-Wert	Code-Länge
the	0.270	01	2
of	0.170	001	3
and	0.137	111	3
to	0.099	110	3
a	0.088	100	3
in	0.074	0001	4
that	0.052	1011	4
is	0.043	1010	4
it	0.040	00001	5
on	0.033	00000	5

Huffman-Codierungs-Baum II



Fragen zur Kompression

- Verschiedene Kompressionsverfahren arbeiten unterschiedlich gut bei Daten unterschiedlichen Typs (Bilder, numerische Daten, Text).
- Welches Verfahren sollte wann benutzt werden?
- Welches Verfahren ist gut für Text?
- Sind Kombinationen, d.h. die Hintereinanderausführung mehrerer solcher Verfahren sinnvoll?
- Schlussfolgerung: Wir interessieren uns für mehr Verfahren.

Geschichte

1977	Der Kompressionsalgorithmus LZ77 wird von Abraham Lempel und Jacob Ziv erfunden
1983	(20. Juni) Terry A. Welch (Sperry Corporation - später Unisys) patentiert das LZW-Kompressionsverfahren (Variante von LZ78)
1987	(15. Juni) CompuServe veröffentlicht GIF als freie und offene Spezifikation (Version 87a)
1989	GIF 89a wird vorgestellt

Geschichte

1993	Unisys informiert CompuServe über die Verwendung ihres patentierten LZW-Algorithmus in GIF
1994	(29. Dez.) Unisys gibt öffentlich bekannt, Gebühren für die Verwendung des LZW-Algorithmus einzufordern
1995	(4. Jan.) die PNG Gruppe wird gegründet (7. März) erste PNG Bilder werden ins Netz gestellt (8.Dez.) die PNG-Spezifikation 0.92 steht im W3C
1997	Netscape 4.04 und Internet Explorer 4.0 erscheinen mit PNG Unterstützung

Lempel-Ziv-Welch(LZW)

- Erfunden 1978 (nach dem heute üblichen LZ77), siehe später.
- Erlaubt Kompression, ohne Initialisierungsdaten zu übertragen.
- Arbeitet mit Zeichensatz variabler Länge l. Die Zeichen selbst bestehen aus einem oder mehreren Buchstaben.

Algorithmus:

- Zunächst wird der Zeichensatz mit den 256 Byte-Zeichen und einem Ende-Zeichen initialisiert.
- In der Kodierungsschleife wird das längste (ein-oder mehrbuchstabige) Zeichen aus dem Zeichensatz ermittelt, das mit der Buchstabenfolge des Eingabestroms übereinstimmt.
  - Die Nummer dieses Zeichens wird in die Ausgabe geschrieben.
  - Zusätzlich wird ein neues Zeichen definiert: Die Verlängerung der eben gefundenen Buchstabenfolge um den nächsten Buchstaben.
- Der Zeichensatz wird so immer größer. Bei einer maximalen Größe wird der Zeichensatz wieder auf 257 Zeichen reduziert und der Vorgang wiederholt sich so lange, bis der Eingabestrom vollständig codiert ist.

LZW Algorithmus

Encode:
Codetabelle initialisieren (jedes Zeichen erhält einen Code);
präfix= “ “;
while *Ende des Eingabedatenstroms noch nicht erreicht* **do**
    suffix:= nächstes Zeichen aus dem Eingabedatenstrom;
    muster:= präfix+ suffix;
    if *muster* $\in$ *Codetabelle* **then** präfix:= muster;
    else
        muster in Codetabelle eintragen;
        LZW-Code von präfix ausgeben;
        präfix:= suffix;
    end
end
if *präfix* $\neq \emptyset$ **then**
    LZW-Code von präfix ausgeben;
end

LZW Algorithmus

Beispiel: Codetabelle: 0:**A** 1:**B** 2:**C** 3:**D**

	präfix	muster	suffix	Eingabedatenstrom	LZW-Code	Ausgabe
0		**A**	**A**	**ABCABCABCD**		
1	**A**	**AB**	**B**	**BCABCABCD**	4:**AB**	0 (**A**)
2	**B**	**BC**	**C**	**CABCABCD**	5:**BC**	1 (**B**)
3	**C**	**CA**	**A**	**ABCABCD**	6:**CA**	2 (**C**)
4	**A**	**AB**	**B**	**BCABCD**		
5	**AB**	**ABC**	**C**	**CABCD**	7:**ABC**	4 (**AB**)
6	**C**	**CA**	**A**	**ABCD**		
7	**CA**	**CAB**	**B**	**BCD**	8:**CAB**	6 (**CA**)
8	**B**	**BC**	**C**	**CD**		
9	**BC**	**BCD**	**D**	**D**	9:**BCD**	5 (**BC**)
10	**D**					3 (**D**)

LZW Algorithmus

Decode:
Codetabelle initialisieren (jedes Zeichen erhält einen Code);
präfix= “ “;
while *Ende der Daten noch nicht erreicht* **do**
    lies LZW-Code;
    muster:= dekodiere (LZW-Code);
    gib muster aus;
    neuer LZW-Code := präfix + erstes Zeichen von muster;
    präfix:= muster;
end

LZW Algorithmus

Beispiel: (Dekodierung) Codetabelle: 0:A1:B2:C3:D

Code	präfix	muster	neuer Code	Ausgabe
0		A		A
1	A	B	4 = AB	B
2	B	C	5 = BC	C
4	C	AB	6 = CA	AB
6	AB	CA	7 = ABC	CA
5	CA	BC	8 = CAB	BC
3	ABC	D	9 = BCD	D

Vorteile von LZW:

- Zeichensatz und Codierung werden häufig gewechselt: LZW kann sich einem Kontextwechsel im Eingabestrom gut anpassen.
- Um den damit verbundenen “Gedächtnisverlust” zu beschränken, kann man die Zeichen aus dem Zeichensatz (nach Anzahl der Benutzung und Länge) bewerten und die besten n Zeichen behalten.

Lempel-Ziv Algorithmen

LZ77 (Sliding Window)

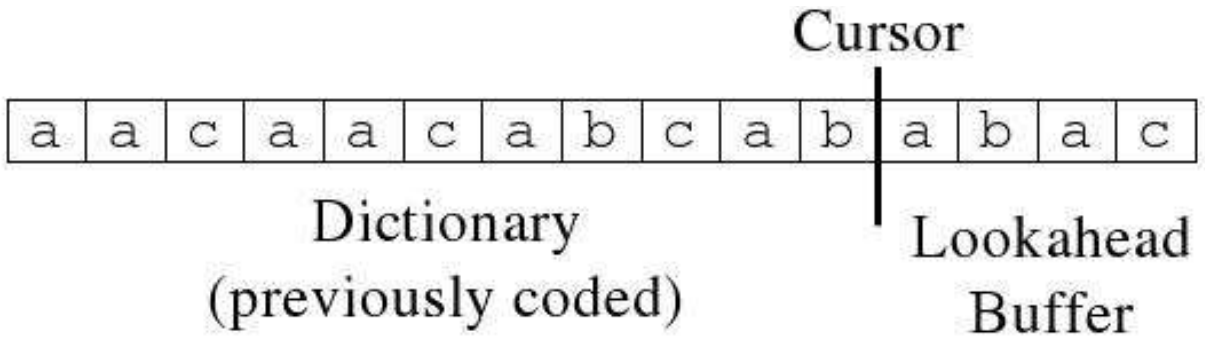
- Varianten: LZSS (Lempel-Ziv-Storer-Szymanski)
- Applications: `gzip`, Squeeze, LHA, PKZIP, ZOO

LZ78 (Dictionary Based)

- Variants: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Applications: `compress`, GIF, CCITT (modems), ARC, PAK

Normalerweise wurde LZ77 als besser und langsamer als LZ78 betrachtet, aber auf leistungsfähigeren Rechnern ist LZ77 auch schnell.

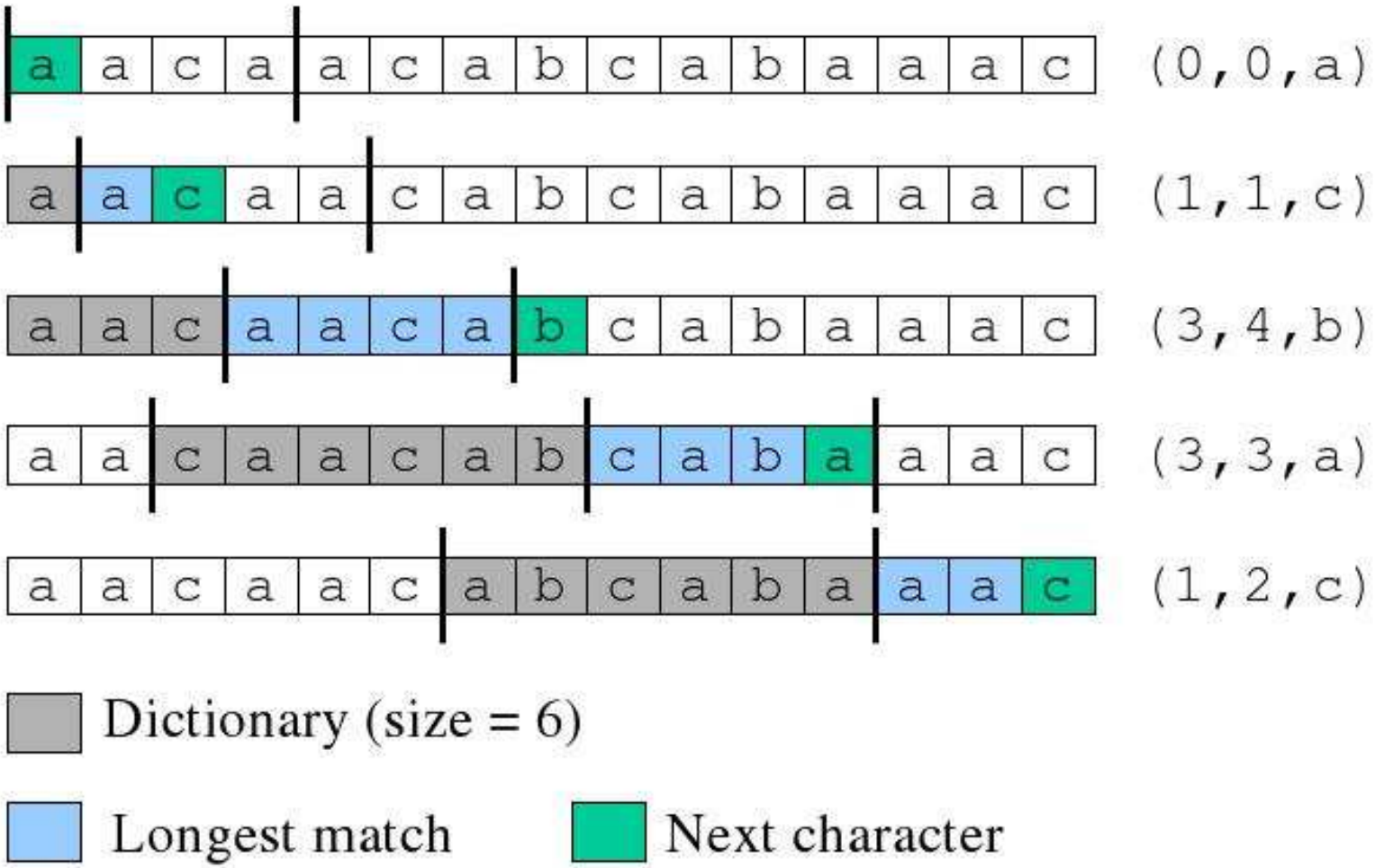
LZ77: Sliding Window Lempel-Ziv



Dictionary- und Buffer-Windows haben feste Länge und verschieben sich zusammen mit dem Cursor.
An jeder Cursor-Position passiert folgendes:

- Ausgabe des Tripels `(p,l,c)`
  - `p` = relative Position des longest match im Dictionary
  - `l` = Länge des longest match
  - `c` = nächstes Zeichen rechts vom longest match
- Verschiebe das Window um `l+1`

LZ77: Example



LZ77 Decoding

Der Decodierer arbeitet mit dem selben Dictionary-Window wie der Codierer

- Im Falle des Tripels `(p,l,c)` geht er `p` Schritte zurück, liest die nächsten `l` Zeichen und kopiert diese nach hinten. Dann wird noch `c` angefügt.

Was ist im Falle `l>p`? (d.h. nur ein Teil der zu copierenden Nachricht ist im Dictionary)

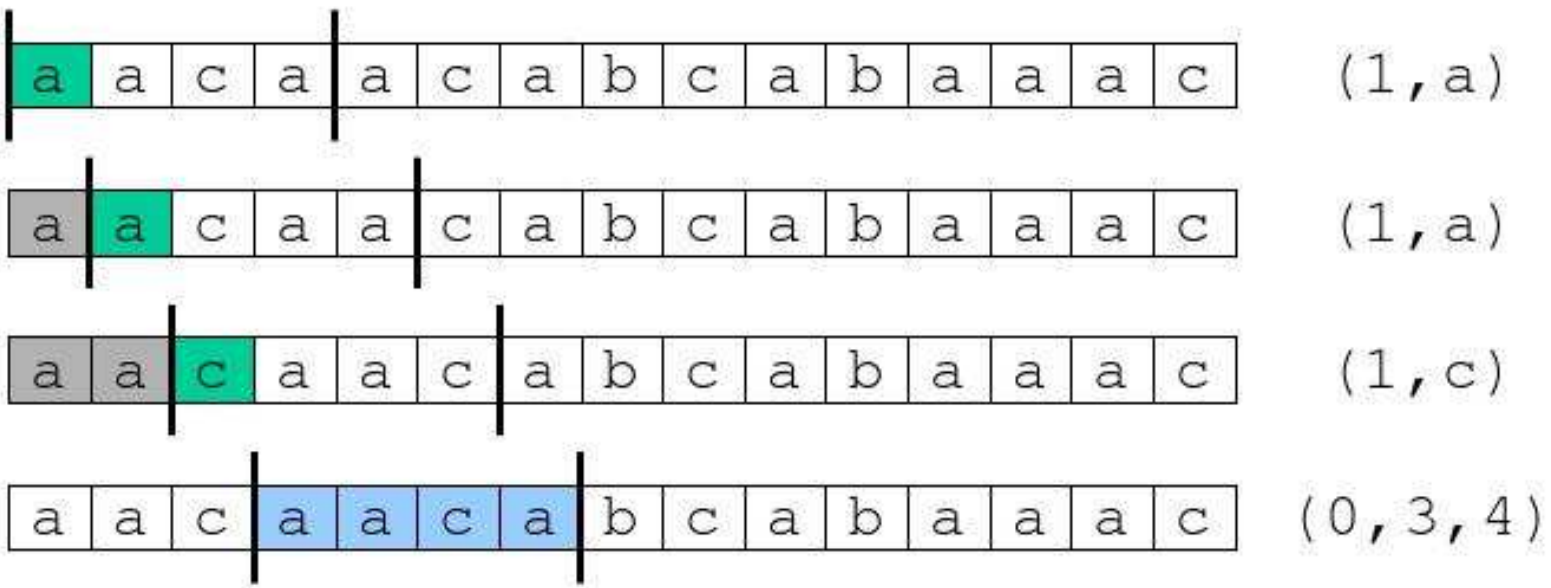
- Beispiel `dict = abcd, codeword = (2,9,e)`
- Lösung: Kopiere einfach zeichenweise:

```
for (i = 0; i < length; i++)
out[cursor+i] = out[cursor-offset+i]
```

- `Out = abcdcdcdcdcdce`

LZ77 Optimierungen bei `gzip` I

LZSS: Der Output hat eins der zwei Formate
`(0, position, length)` oder `(1,char)`
Benutze das zweite Format, falls `length < 3`.



Optimierungen bei `gzip` II

- Nachträgliche Huffman-Codierung der Ausgabe
- Clevere Strategie bei der Codierung: Möglicherweise erlaubt ein kürzerer Match in diesem Schritt einen viel längeren Match im nächsten Schritt
- Benutze eine Hash-Tabelle für das Wörterbuch.
  - Hash-Funktion für Strings der Länge drei.
  - Suche für längere Strings im entsprechenden Überlaufbereich die längste Übereinstimmung.

Theorie zu LZ77

LZ77 ist asymptotisch optimal [Wyner-Ziv,94]
LZ77 komprimiert hinreichend lange Strings entsprechend seiner Entropie, falls die Fenstergröße gegen unendlich geht.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

Achtung, hier ist wirklich eine sehr große Fenstergröße nötig.
In der Praxis wird meist ein Puffer von 216 Zeichen verwendet.