Text Processing

Text Processing

- Document pre-processing
  1. Lexical analysis
  2. Stopword elimination
  3. Stemming
  4. Index-term selection
  5. Thesauri
- Text Compression
  1. Statistical methods
  2. Huffman coding
  3. Dictionary methods
  4. Ziv-Lempel compression

Document Pre-processing

- Document pre-processing is the process of incorporating a new document into an information retrieval system.
- The goal is to
  - Represent the document *efficiently* in terms of both *space* (for storing the document) and *time* (for processing retrieval requests) requirements.
  - Maintain good *retrieval performance* (precision and recall).
- Document pre-processing is a complex process that leads to the representation of each document by a selected set of *index* terms.
- However, some Web search engines are giving up on much of this process and index *all* (or virtually all) the words in a document).
- Document pre-processing includes 5 stages:
  - Lexical analysis
  - Stopword elimination
  - Stemming
  - Index-term selection
  - Construction of thesauri

Lexical Analysis

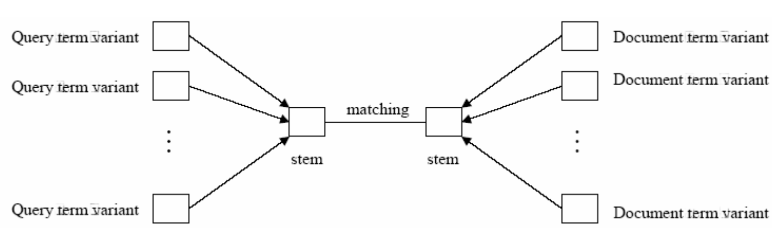
- *Objective*: Determine the words of the document.
- Lexical analysis separates the input alphabet into
  - Word characters (e.g., the letters a-z)
  - Word separators (e.g., space, newline, tab)
- The following decisions may have impact on retrieval
  - *Digits*: Used to be ignored, but the trend now is to identify numbers (e.g., telephone numbers, credit card numbers) and mixed strings (e.g., model numbers) as words.
  - *Punctuation marks*: Usually treated as word separators (but in documents about programming languages “x.id” may be different from “xid”)
  - *Hyphens*: Should we interpret “pre-processing” as “pre processing” or as “preprocessing”?
  - Letter case: Often ignored, but then a search for “First Bank” (a specific bank) would retrieve a document with the phrase “Bank of America was the first bank to offer its customers…”

Stopword Elimination

- *Objective*: Filter out words that occur in most of the documents.
- Such words have no value for retrieval purposes (they are said to have *low discrimination value*).
- These words are referred to as stopwords. They include
  - Articles (a, an, the, …)
  - Prepositions (in, on, of, …)
  - Conjunctions (and, or, but, if, …)
  - Pronouns (I, you, them, it…)
  - Possibly some verbs, nouns, adverbs, adjectives (make, thing, similar, …)
- A typical stopword list may include several hundred words.
- The 100 most frequent words add-up to about 50% of the words in a document.
- Hence, stopword elimination improves the size of the indexing structures.
- Elimination of stop words might reduce recall. After elimination of “or”, “to”, “not”, it might be difficult to find relevant documents for the query “to be or not to be”.

Stemming

- *Objective*: Replace all the *variants* of a word with the single *stem* of the word.
- Variants include plurals, gerund forms (ing-form), third person suffixes, past tense suffixes, etc.
- *Example*: **connect**: connects, connected, connecting, connection,…
- All have similar semantics and relate to a single concept.
- In parallel, stemming must be performed on the user query.



Stemming (cont.)

- Stemming improves
  - *Storage and search efficiency*: less terms are stored.
  - *Recall*:
    - without stemming a query about “connection”, matches only documents that have “connection”.
    - With stemming, the query is about “connect” and matches *in addition* documents that originally had “connects”, “connected”, “connecting”, etc.
- However, stemming may hurt *precision*, because users can no longer target just a particular form.
- Stemming may be performed using
  - *Algorithms* that remove suffixes according to substitution rules, e.g.:
    - IF a word ends in “ies”, but not “eies” or “aies” THEN “ies” → “y” (flies → fly)
    - IF a word ends in “es”, but not “aes”, “ees”, or “oes” then “es”→ “e” (engines → engine)
    - IF a word ends in “s”, but not “us” or “ss” THEN “s” → NULL (documents → document)
  - Large *dictionaries*, that provide the stem of each word.

Index Term Selection (Indexing)

- *Objective*: Increase efficiency by extracting from the resulting document a *selected set of terms* to be used for indexing the document.
  - If full text representation is adopted then *all* words are used for indexing.
- Indexing is a critical process: User's ability to find documents on a particular subject is limited by the indexing process having created index terms for this subject.
- Index can be done *manually* or *automatically*.
- Historically, manual indexing was performed by professional indexers associated with library organizations.
- However, automatic indexing is more common now (or, with full text representations, indexing is altogether avoided).

Index Term Selection (cont.)

Reducing the *size* of the index:

- Recall that articles, prepositions, conjunctions, pronouns have already been removed through a stopword list.
  - Recall that the 100 most frequent words account for 50% of all word occurrences.
- Words that are *very infrequent* (occur only a few times in a collection) are often removed, under the assumption that they would probably not be in the user's vocabulary.
  - Recall that words that occur 1-3 times account for 75% of the vocabulary.
- *Nouns* are often preferred over verbs, adjectives, or adverbs.

Thesauri

Objective: Standardize the index terms that were selected.

- In its simplest form a thesaurus is
  - A list of "important" words (concepts).
  - For each word, an associated list of synonyms.
- A thesaurus may be generic (cover all of English) or concentrate on a particular domain of knowledge.
- The role of a thesaurus in information retrieval
  1. Provide a standard vocabulary for indexing
  2. Help users locate proper query terms.
  3. Provide hierarchies for automatic broadening or narrowing of queries.
- Here, our interest is in providing a standard vocabulary (a controlled vocabulary).
- Essentially, in this final stage, each indexing term is *replaced* by the concept that defines its thesaurus class.
- A sample thesaurus entry:
  - **EYEGLASSES**
UF ***SPECTACLES***
BT **MEDICAL EQUIPMENT AND SUPPLIES**
BT **OPTICAL DEVICES**
NT **MONOCLES**
NT **SUNGLASSES**
RT **CONTACT LENSES**
RT **EYE PATCHES**
RT **GOGGLES**

(UF=used for, BT=broader term, NT=narrower term, RT=related term)

Text Compression

- *Data Encoding*: Transform encoding units (characters, words, etc.) into code values.
  - *Objective* is to reduce size (compression)
- *Lossless encoding*: The transformation is *reversible* – original file can be recovered from encoded (compressed) file.
- *Compression ratio*:
  - *S*: size of the uncompressed file.
  - *C*: size of the compressed size file.
  - *Compression-rate* = S/C.
  - *Example*:
    - S= 300,000 bytes, C=100,000 bytes.
    - Compression rate: 300,000 /100,000=3:1.

Text Compression *(cont.)*

- *Advantages* of compression:
  - Reduction in storage size.
  - Reduction in transmission time.
  - Reduction in processing times (e.g., searching).
- Disadvantages:
  - Requires time for compression/decompression.
  - Processing of compressed text is more complex.
- Specific for information retrieval:
  - Decompression time is often more critical than compression time.
    - Unlike transmission-motivated compression (modems), documents in an information retrieval system are encoded once and decoded many times.
  - Prefer compression techniques that allow searching in the compressed file (without decompressing it).

Text Compression *(cont.)*

Basic methods:

- Statistical methods:
  - Estimate the probability of occurrence of each encoding unit (character or word) in the alphabet.
  - Assign codes to units: more frequent units are assigned shorter codes.
  - In information retrieval, word-encoding is preferred over character encoding.
- Dictionary methods:
  - Substitute a *phrase* (string of units) by a pointer to a dictionary or a previous occurrence of the phrase.
- Compression is achieved because the pointer is shorter than the phrase.

Statistical Methods

- Recall from the discussion of information theory:
  - Assume a message from an alphabet of *n* symbols.
  - Assume that the probability of the *i*'th symbol is *pi*.
  - The average information content (*entropy*) of a symbol is

$$E = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i)$$

- *Optimal encoding* is achieved when a symbol with probability *pi* is assigned a code whose length is log2(1/*pi*) = –log2(*pi*) .
- Hence, *E* also represents *optimal average code length* (measured in bits per character).
- Therefore, *E* is the *lower bound* on compression.

Statistical Methods *(cont.)*

- Statistical methods must first estimate the frequencies of the encoding units, and then assign codes based on these frequencies.
- Approaches:
  - *Static*: Use a single distribution for all texts.
    - Fast, but not optimal because different texts exhibit different distributions.
    - The encoding table is stored in the application (not in the text).
    - Decompression can start at any point in the file.
  - *Dynamic (semi-static)*: Determine the frequencies in a preliminary pass.
    - Excellent compression, but a total of two passes is required.
    - The encoding table is stored at the beginning of the text.
    - Decompression can start at any point in the file.
  - *Adaptive*: Progressively learn the distribution of the text while compressing; each character is encoded on the basis of the preceding characters in a text.
    - Fast, and close to optimal compression.
    - Decompression must start from the beginning

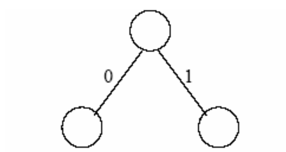
Huffman Coding

- *General*:
  - Huffman coding is one of the best known compression techniques (1952).
  - It is used in the Unix programs pack/unpack.
  - It is a statistical method based on variable length codes.
  - Compression is achieved by assigning shorter codes to more frequent units.
  - Decompression is unique because no code is the prefix of another.
  - Encoding units may be either bytes or words.
  - Does not exploit the *dependencies* between the encoding units.
  - Yields *optimum* average code length when these units are independent.
  - Can be used with the static, dynamic and adaptive approaches.

Huffman Coding *(cont.)*

- *Method*:
  1. Build a table of the encoding units and their frequencies (probabilities).
  2. Combine the two *least* frequent units into a unit with the *sum* of the probabilities and encode it in a new "unit".

New unit: *p*1 + *p*2

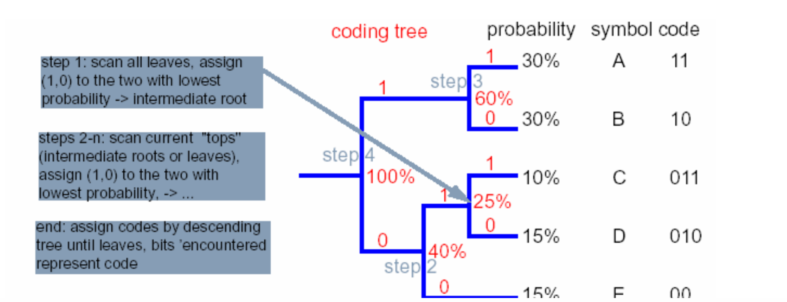


Unit 1: *p*1 Unit 2: *p*2

  3. Repeat this process until the entire dictionary is represented by a root whose probability is 1.0.
  4. When there is a *tie* for the two least frequent units, any tie-breaking procedure is acceptable.

Huffman coding/decoding

- Example:
  – p(A)=0,3, p(B)=0,3, p(C)=0,1, p(D)=0,15, p(E)=0,15



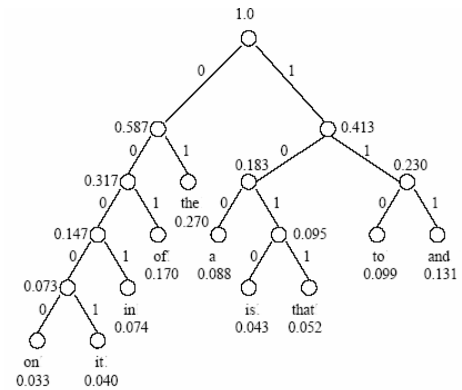
symbol	code
A	11
B	10
C	011
D	010
E	00

| B | A | C | D | A | B | E | B | A | E |

10 11 011 010 11 10 00 10 11 00

Huffman Coding *(cont.)*

- *Example (word encoding)*:
  - Assume the dictionary is 10 words long:
  - The Huffman tree is:



Encoding unit	Occurrence Probability (p_i)
the	.270
of	.170
and	.131
to	.099
a	.088
in	.074
that	.052
is	.043
it	.040
on	.033

Huffman Coding *(cont.)*

- *Example (cont.)*:
  - The resulting code:
  - Average code length:

$$\sum_{i=1}^{10} p_i \cdot l_i = 3.05 \text{ bits}$$

  - The entropy (compression lower bound) is

$$-\sum_{i=1}^{10} p_i \cdot \log_2 p_i = 3.01 \text{ bits}$$

  - Fixed code length would have required $\log_2 10 = 3.32$ bits (which, in practice, would require 4 bits).
  - Compression ratio:
    S/C = 3.32/3.05 = 1,088:1

Encoding unit	Code value	Code length (l_i)
the	01	2
of	001	3
and	111	3
to	110	3
a	100	3
in	0001	4
that	1011	4
is	1010	4
it	00001	5
on	00000	5

Huffman Coding *(cont.)*

- *Example:* When the letters A-Z are thus encoded:
  - Code lengths are between 3 and 10 bits.
  - Average code length is 4.12 bits.
  - A fixed code would have required log226 = 4.70 bits (i.e., 5 bits).
- More compression is obtained by encoding *words*:
  - With the 800 most frequent English words (small table!) are encoded in this method (all other words are in plain ASCII), 40-50% compression has been reported.
- Huffman codes are prefix-specific:
  - No code is the beginning of another code.
  - Hence, a left-to-right decoding operation is *unique*.
  - It is possible to search the compressed text.

Huffman code for three-letter alphabet

Letter	Codeword
a_1	0
a_2	11
a_3	10

- A={α_1,α_2,α_3} με P(α_1)=0,95, P(α_2)=0,02 ,P(α_3)=0,03
- Entropy for A: 0,335 bits/symbol
- Average Huffman code length: 1,05 bits/σύμβολο: 213% higher than entropy
  - The reason: symbol α_1 is encoded with 1 bit (>> 0,1520 = $\log_2(1/P(\alpha_1))$)

Αριθμητική κωδικοποίηση

- Πιο σύγχρονη μέθοδος κωδικοποίησης
- Συνήθως υψηλότερα ποσοστά συμπίεσης σε σχέση με την κωδικοποίηση Huffman
- Η κωδικοποίηση Huffman αντιστοιχεί σε κάθε σύμβολο ένα κωδικό με ακέραιο αριθμό δυαδικών ψηφίων.
- Η αριθμητική κωδικοποίηση μπορεί να χειριστεί όλο το μήνυμα σαν μία μονάδα
- Ένα μήνυμα αναπαρίσταται με ένα διάστημα [α,β) όπου α και β πραγματικοί αριθμοί στο διάστημα [0,1].
- Τα σύμβολα αντιστοιχούν σε υποδιαστήματα του [0,1] τα εύρη των οποίων είναι ανάλογα με την πιθανότητα εμφάνισής των συμβόλων
- Το μήνυμα πρέπει να τελειώνει με ένα ειδικό τερματικό σύμβολο π.χ "$" προκειμένου να μην υπάρχει ασάφεια στην αποκωδικοποίηση του μηνύματος
- Καθώς το μήνυμα μεγαλώνει, το μήκος του διαστήματος μειώνεται και ο αριθμός των δυαδικών ψηφίων που χρειάζονται για την αναπαράσταση του διαστήματος αυξάνει.

Ο κωδικοποιητής Αριθμητικής Κωδικοποίησης

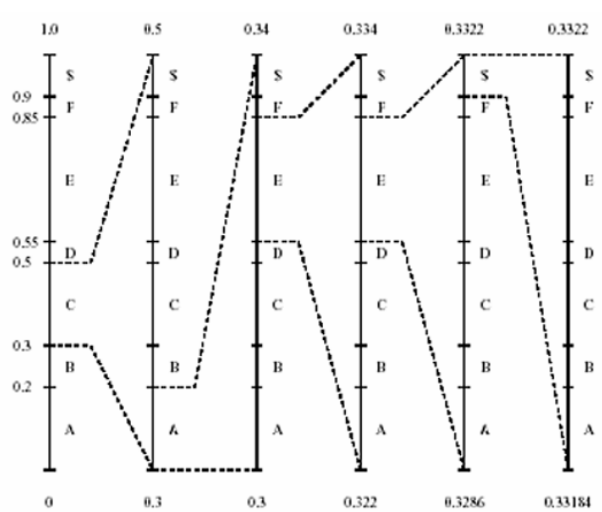
```
BEGIN
   low = 0.0; high = 1.0; range = 1.0;
   while (symbol != terminator)
    {
        get (symbol);
        high = low + range * Range_high(symbol);
        low = low + range * Range_low(symbol);
        range = high - low;
    }
   output a code so that low <= code < high;
END
```

Παράδειγμα

Κωδικοποίηση του μηνύματος “CAEE$”

Symbol	**Probability**	**Range**
A	0.2	[0, 0.2)
B	0.1	[0.2, 0.3)
C	0.2	[0.3, 0.5)
D	0.05	[0.5, 0.55)
E	0.3	[0.55, 0.85)
F	0.05	[0.85, 0.9)
$	0.1	[0.9, 1.0)

Η κατανομή πιθανότητας των συμβόλων



- Σταδιακή μείωση του εύρους του διαστήματος [α,β)
- Αρχικά α=0, β=1
- Απαραίτητη η ύπαρξη τερματικού συμβόλου. Διαφορετικά π.χ. τα μηνύματα A, AA, AAA, AAAA, AAAAA,…. κωδικοποιούνται όλα με το 0:
  - Αδύνατη η αποκωδικοποίηση: ο ίδιος κωδικός αντιστοιχεί σε πολλά μηνύματα

Οι τιμές των παραμέτρων κατά την εκτέλεση του αλγόριθμου

Symbol	low	high	range
	0	1.0	1.0
C	0.3	0.5	0.2
A	0.30	0.34	0.04
E	0.322	0.334	0.012
E	0.3286	0.3322	0.0036
$	0.33184	0.33220	0.00036

Αλγόριθμος προσδιορισμού του αριθμού ($0.33203125_{10}=0.01010101_2$) με τη συντομότερη δυαδική αναπαράσταση ο οποίος ανήκει στο τελικό διάστημα [low, high)=[0.33184 , 0.33220).

```
BEGIN
 code = 0;
 k = 1;
 while (value(code) < low)
 { assign 1 to the kth binary fraction bit
    if (value(code) > high)
       replace the kth bit by 0
    k = k + 1;
 }
END
```

Αποκωδικοποιητής Αριθμητικής Κωδικοποίησης

```
BEGIN
    get binary code and convert to
    decimal value = value(code);
    range=1; low=0; high=1;
    Do
     { find a symbol s so that
       Range_low(s) <= (value-low)/(high-low) < Range_high(s);
       output s;
       high = low + range * Range_high(symbol);
        low = low + range * Range_low(symbol);
       range = high - low;
       }
    Until symbol s is a terminator
END
```

Αποκωδικοποίηση του 0.33203125 σε "CAEE$"

Output Symbol	low	high	range
C	0.3	0.5	0.2
A	0.30	0.34	0.04
E	0.322	0.334	0.012
E	0.3286	0.3322	0.0036
$	0.33184	0.33220	0.00036

Dictionary Methods

- Dictionary methods construct a dictionary of phrases, and replace their occurrences with dictionary pointers.
- The choice of phrases may be static, dynamic or adaptive.
- A simple method (digrams):
  - Construct a dictionary of *pairs* of letters that occur together frequently (e.g., ou, ea, ch, …).
  - If *n* such pairs are used, a pointer (location in the dictionary) requires $\log_2 n$ bits.
  - At each step in the encoding, the next pair is examined.
    - If it corresponds to a dictionary pair, it is replaced by its encoding, and the encoding position moves by 2 characters.
    - Otherwise, the single character encoding is kept, and the position moves by one character.
  - To assure that decoding is unambiguous, an extra bit is needed to indicate whether the next unit is a single character code or a digram code.

Input Parsing

Input: compression ratios measure compression

Dictionary D = {press,
comp, pres, sion,
asu, com, eas, ure,
io, me, on, ra,
a, c, e, i, m, n, o, p, r, s, t, u, <blank>}

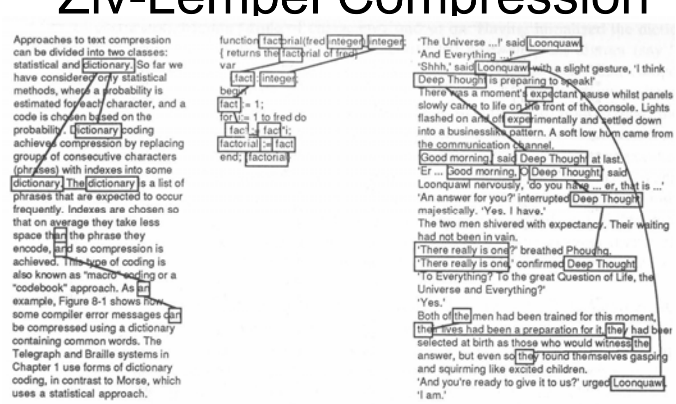
Greedy: comp/r/e/s/sion/ /ra/t/io/s/ /me/a/s/ure/ /comp/r/e/s/sion
LFF: com/press/i/on/ /ra/t/io/s/ /m/eas/ure/ /com/press/i/on
Optimal: com/pres/sion/ /ra/t/io/s/ /m/eas/ure/ /com/pres/sion

- 25 entries in dictionary: five bits per dictionary index
- Greedy parsing: the encoder finds the longest dictionary phrase that matches a prefix of the un-coded portion of the input stream and the index of that dictionary entry is used to encode the input prefix.
- Longest fragment first (LFF) parsing: the encoder parses the input by repeatedly locating the longest substring of the uncoded portion of the input which matches a dictionary entry and replacing it with the corresponding dictionary reference. This process continues until the input is completely replaced by references.
- In general, the compression performance of LFF lies between greedy and optimal parsing.
- However, to determine an optimal or LFF parsing, a sequential encoder must be capable of looking at arbitrarily large prefixes of the input. Consequently, greedy parsing is widely used in sequential compression systems since it requires only limited look-ahead and is computed on-line.

Ziv-Lempel Compression

- *General*:
  - The Ziv-Lempel method (1977) uses a single-pass adaptive scheme.
  - While compressing, it constructs a dictionary from phrases encountered so far.
  - Many popular programs (Unix compress/uncompress, GNU gzip/gunzip, and Windows WinZip) are based on the Ziv-Lempel algorithm.
  - Compression is slightly better than Huffman codes (S/C of 2,2:1 vs. 1,81:1).
  - Disadvantage for information retrieval: decompressed file cannot be searched and decoding cannot start at a random place in the file.

Ziv-Lempel Compression



- Phrases are replaced with a pointer to where they have occurred earlier in the text.
- A phrase might be a word, part of a word, or several words.
- It can be replaced with a pointer as long as it has occurred once before in the text, so coding adapts quickly to a new topic.
- For example, the phrases "dictionary," "fact," and "Loonquawl" occur frequently in the particular examples, yet they are not particularly common in general.
- More common words are also susceptible to this type of coding because repetitions are never far apart (e.g. the phrases "an," "integer," and "the.“)

LZ78 Compression/Decompression

- LZ78 *Compression:*
  1. Initialize the dictionary to contain all “phrases” of length one.
  2. Examine the input stream and search for the longest prefix which has appeared in the dictionary.
  3. Encode this prefix by its index in the dictionary.
  4. Add the prefix followed by the next symbol in the input stream to the dictionary.
  5. Go to Step 2.

- LZ78 *Decompression:*
  1. Initialize the dictionary to contain all “phrases” of length one.
  2. Decode the first value in the input stream using the dictionary.
  3. Decode the next value in the input stream using the dictionary.
  4. Add to the dictionary a phrase made of the previous decoded phrase and the first symbol of the current decoded phrase.
  5. Go to step 3.

LZ78 Compression *(cont.)*

- *Example*:
  – Assume a dictionary of 16 phrases (4 bit index).

Data	a	b	b	a	a	b	b	a	a	b	a	b	b	a	a	a	a	b	a	a	b	b	a	b	a
Encryption	0	1	1	0	2		4		2		6			5		5		7			3		8		

- *Compression output*:
  13 pointers of 4 bits require a total of 52 bits.
- In practice, the Lempel-Ziv algorithm works well only when the input data is sufficiently large and there is sufficient redundancy in the data.

Dictionary			
Index	**Entry**	**Index**	**Entry**
0	a	8	aba
1	b	9	abba
2	ab	10	aaa
3	bb	11	aab
4	ba	12	baab
5	aa	13	bba
6	abb	14	
7	baa	15	

LZ78 Compression *(cont.)*

- *Dictionary size*:
  - In theory, it can grow without bound.
  - In practice, it is limited:
    - Once the limit is reached, no more entries are added.
    - Typical size is 4096 entries (12 bit index).
  - The dictionary can also be a “sliding window” over the most recent input:
    - New phrases are shifted-in and old phrases are shifted-out.
    - The decoder and encoder follow the same dictionary update rules to assure that their dictionaries are synchronized.

LZ77: Sliding Window Lempel-Ziv

Cursor

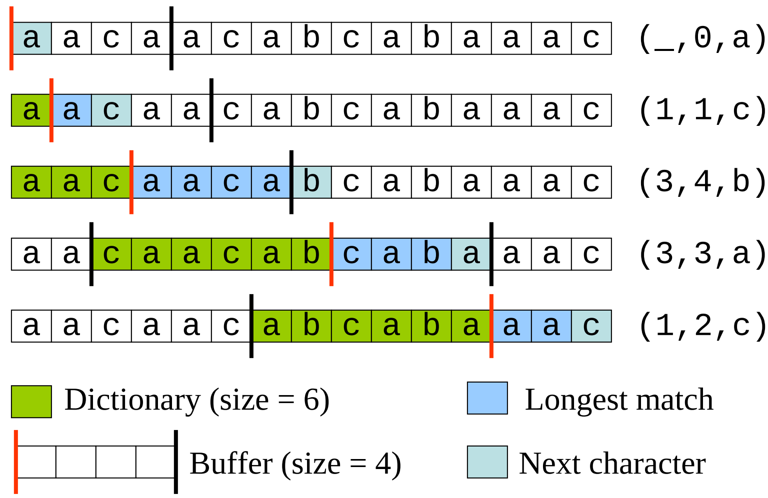
a	a	c	a	a	c	a	b	c	a	b	a	b	a	c

Dictionary (previously coded)

Lookahead Buffer

- **Dictionary** and **buffer** “windows” are fixed length and slide with the **cursor**
- On each step:
- Output (p, *l*, c) where
  p = relative position of the longest match in the dictionary
  l = length of longest match
  c = next char in buffer beyond longest match
- Advance window by *l* + 1

LZ77: Example



LZ77 (cont.)

- A window of moderate size, typically *N*<= 8192, can work well for a variety of texts for the following reasons.
- Common words and fragments of words occur regularly enough in a text to appear more than once in a window. Some English examples are "the," "of," "pre-," "-ing"; while a source program may use keywords such as "while," "if," "then."
- Specialist words tend to occur in clusters: for example, words in a paragraph on a technical topic, or local identifiers in a procedure of a source program.
- · Less common words may be made up of fragments of other words. For example, “experimentally" could be constructed from "expectant" and other suitable words.
- Runs of characters are coded compactly. For example, *k* spaces followed by the letter b may be coded recursively as <0, 0, ‘ '> <1, *k-1, b*>, the first triple establishing the space character, and the second repeating it *k* -1 times.

LZ77 Decoding

- Decoder keeps same dictionary window as encoder.
- For each message it looks it up in the dictionary and inserts a copy
- What if *l* > p? (only part of the message is in the dictionary.)
- E.g. dict = abcd, codeword = (2,9,e)
- Simply copy from left to right

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

- Out = `abcdcdcdcdcdce`