University of Helsinki
Department of Computer Science

582487
Data Compression Techniques

Lecture 7: Dictionary Compression

Simon J. Puglisi
(puglisi@cs.helsinki.fi)

Outline

- Brief introduction about *text compression*
- Why would we do it? (i.e. motivation)
- Three classic dictionary compression algorithms
  - Lempel-Ziv '78 (LZ78, LZW)
  - Lempel-Ziv '77 (LZ77, RLZ)
  - Grammar Compression (RePair)
- Thursday: no lecture
  - (Travis in Warsaw, Simon in London)
- Next Tuesday: Travis with more text compression

Remember how Google works?

- Crawl the web, gather a collection of documents
- For each word t in the collection, store a list of all documents containing t:
- Query: blue mittens

Inverted Index

...

blue → 1 | 2 | 4 | 11 | 31 | 45 | 173 | 174

blunt → 1 | 2 | 4 | 5 | 6 | 16 | 57 | 132 | 173 | ...

...

mint → 2 | 31 | 54 | 101

mittens → 1 | 4 | 5 | 11 | 31 | 45 | 174 | 288

... ...

lexicon — lists

Remember how Google works?

- **Query:** blue mittens

blue →

1	2	4	11	31	45	173	174

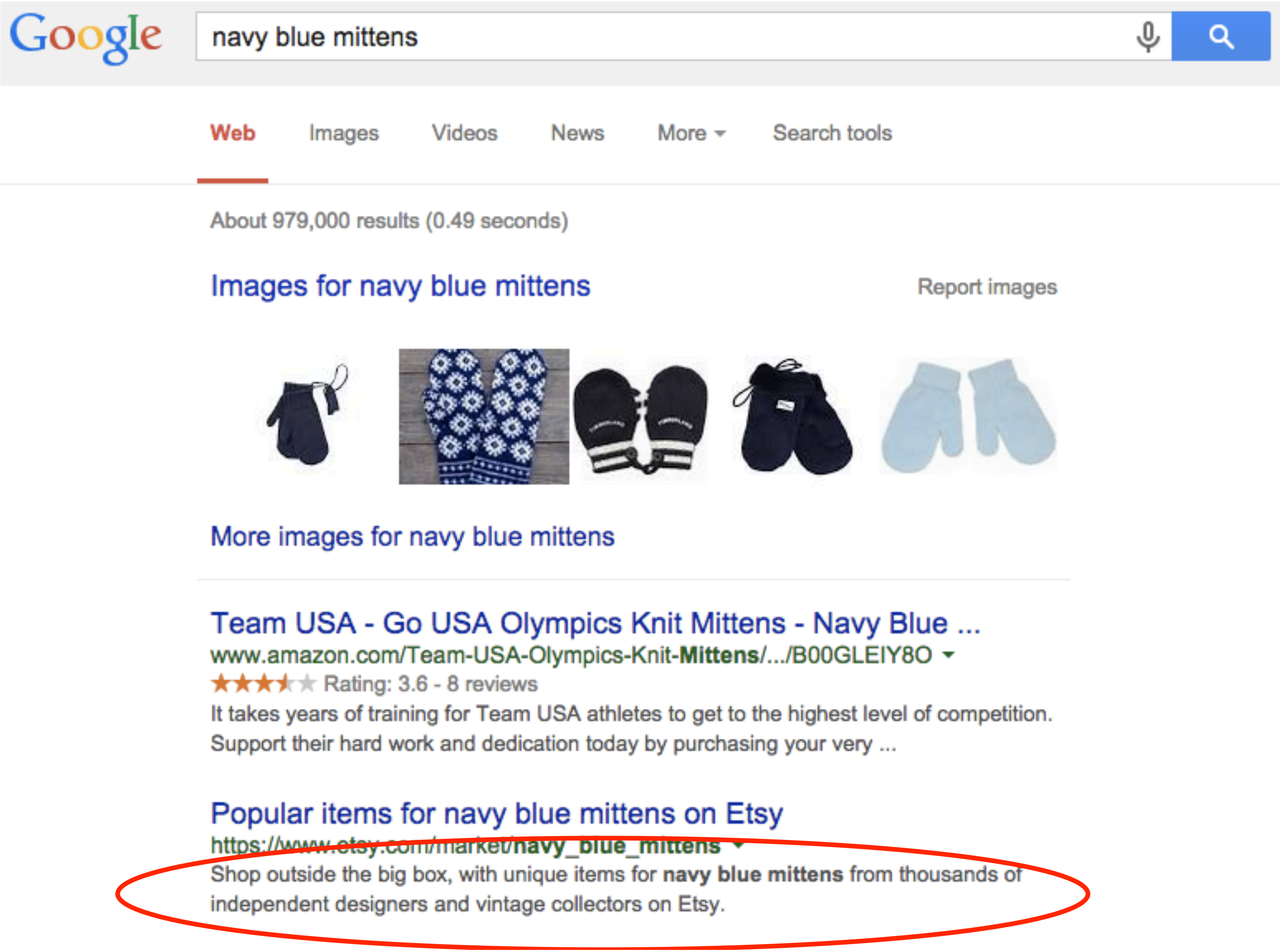
1	4	11	31	45	174

Intersect lists

mittens →

1	4	5	11	31	45	174	288

What happens after the result list is determined?


Google
navy blue mittens
Web
Images
Videos
News
More
Search tools
About 979,000 results (0.49 seconds)
Images for navy blue mittens
Report images
More images for navy blue mittens
Team USA - Go USA Olympics Knit Mittens - Navy Blue ...
www.amazon.com/Team-USA-Olympics-Knit-Mittens/.../B00GLEIY8O
Rating: 3.6 - 8 reviews
It takes years of training for Team USA athletes to get to the highest level of competition. Support their hard work and dedication today by purchasing your very ...
Popular items for navy blue mittens on Etsy
https://www.etsy.com/market/navy_blue_mittens
Shop outside the big box, with unique items for navy blue mittens from thousands of independent designers and vintage collectors on Etsy.

How does compression help?

- Search engines have to store the documents they index
- We want to compress web collections in order to...
- Reduce space
  - Web crawls are large and contain lots of redundancy....
  - ...duplicate documents, reused text, HTML boilerplate...
- <u>Compression aids throughput</u>
  - Faster to transfer compressed data from disk to memory

Elsewhere…

- Text compression is, of course, not just happening at Google…
- Personal file compression: zip, gzip, bzip, p7zip, et c.
- Online file repositories: e.g., dropbox.
- Storage of genomic data: DNA read sets are shipped compressed.
- Et c., et c., et c. – it's everywhere.

“Text compression”

- Keep in mind that the term “text compression” encompasses more than just compression of natural language documents…
- Suitable for other data with similar sequential structure, such as program source code
- Text compressors achieve some compression on almost any kind of (uncompressed) data

Dictionary compression…

Dictionary compression

- In dictionary compression variable length substrings are replaced by short, possibly even fixed length, codewords

- The dictionary D is a collection of strings, often called *phrases*. For completeness, it includes all single symbols.

- The text T (over an alphabet of size σ) is parsed into phrases

$$T = T_1 T_2 \ldots T_z,\ T_i \text{ in } D.$$

- The sequence is called a *parsing* or *factorization* of T with respect to D

- The text is encoded by replacing each phrase T_i with a code than acts as a pointer to the dictionary

Dictionary compression

- Here is a simple static dictionary compression scheme for English text:
  - Dictionary consists of some set of English words + individual symbols
  - Compute frequencies of the words in some corpus of English texts.
  - Compute frequencies of symbols in the corpus from which the dictionary words have been removed
  - Number the words and symbols in descending order of frequencies
- To encode a text, replace each dictionary word and each symbol that does not belong to a word with its corresponding number. Encode the sequences of numbers using γ coding.

Lempel-Ziv compression…

Lempel-Ziv compression

- In 1977 and 1978, Abraham Lempel and Jacob Ziv published two *adaptive* dictionary compression algorithms that soon came to dominate practical text compression
- Many variants have been published and implemented (`zip`, `gzip`)
  - the most widely used algorithms in general purpose compression tools
- The common feature of the two algorithms is that the dictionary consists of substrings of the already processed part of the text
  - In this way the dictionary is able to adapt to the text
- The two algorithms – called LZ77 and LZ78 – differ primarily in the way they represent phrases
  - LZ77 uses direct pointers to the preceding text
  - LZ78 uses pointers to a separate dictionary

LZ78

- The dictionary consists of phrases numbered from 0 upwards:

$$D = \{Z_0, Z_1, Z_2, \ldots\}$$

- Initially, the only phrase is the empty string $Z_0 = \varepsilon$. Each new phrase is inserted to the dictionary and gets the next free number.

- Suppose we have computed the parsing $T_1 \ldots T_{j-1}$ for T[0..i) and the next phrase T_j starts at position i. Let Z_k in D be the longest phrase in the dictionary that is a prefix of T[i..n-1).

- Then the next phrase is $T_j = Z_j = T[i..i+|Z_k|] = Z_k t_{i+|Zk|}$, and it is inserted into the dictionary.

- The phrase T_j is encoded as the pair $<k, t_{i+|Zk|}>$. Using fixed length codes, the pair needs ceil(log(j+1)) + ceil($\log \sigma$) bits.

LZ78 (example)

Let T = b a d a d a d a b a a b

Phrase #	0	1	2	3	4	5	6	7
Phrase	ε	b	a	d	ad	ada	ba	ab
Encoding		<0,b>	<0,a>	<0,d>	<2,d>	<4,a>	<1,a>	<2,b>
Code len		0+2	1+2	2+2	2+2	3+2	3+2	3+2

LZ78 as a trie

- One way to think about LZ78 is by the trie it produces
- Let T = b a d a d a d a b a a b

(0,b)

(0,a)

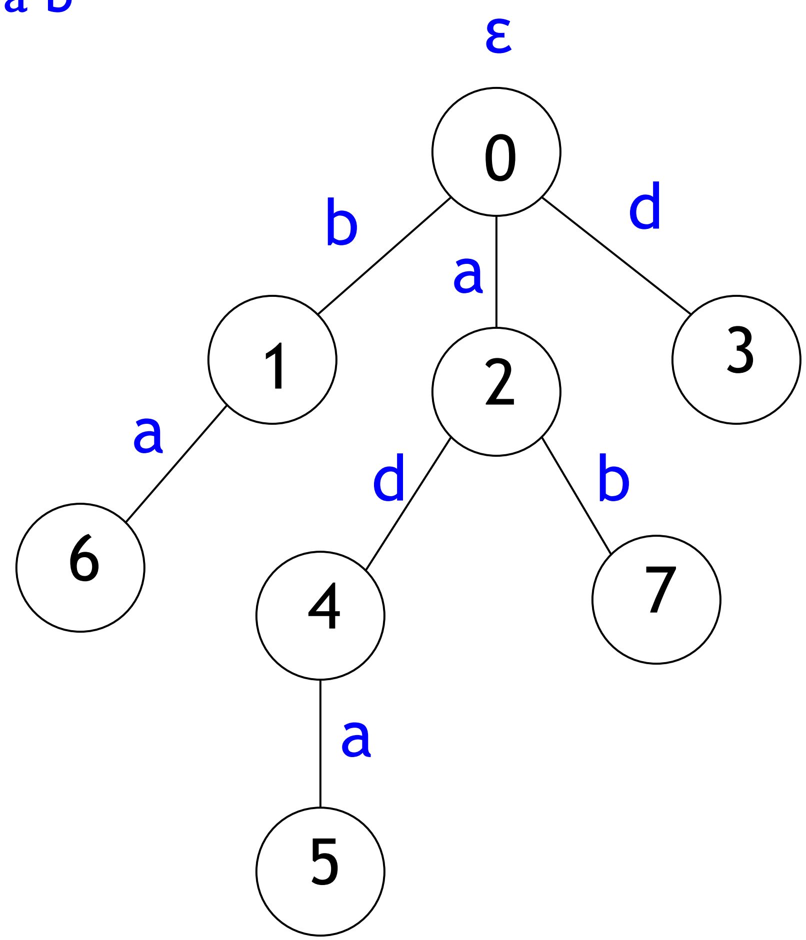
(0,d)

(2,d)

(4,d)

(1,a)

(2,d)



Lempel-Ziv-Welch (LZW)

- Terry Welch came up with a simple but important spin on LZ78 in 1984…

Lempel-Ziv-Welch (LZW)

- Welch's algorithm is used in the unix tool compress.

- Initially the dictionary D contains all individual symbols:

$$D = \{Z_1,\ldots,Z_\sigma\}$$

- Suppose the next phrase T_j starts at position i. Let Z_k in D be the longest phrase in the dictionary that is a prefix of T[i..n). Now the next text phrase is $T_j = Z_k$ and the phrase added to the dictionary is $Z = T_j T[i+|T_j|]$.

- The phrase T_j is encoded with the index k, requiring

$$\log(\sigma+j-1) \text{ bits.}$$

- Idea: Omitting symbol codes (of LZ78) saves space in practice, even though index codes can be longer and phrases shorter

LZW: encoding

Let T = b a d a d a d a b a a b

Phrase					b	a	d	ad	ada	ba	a	b
Enc.					2	1	4	6	8	5	1	2
CodeLen.					2	3	3	3	3	4	4	4
Dict.	a	b	c	d	ba	ad	da	ada	adab	baa	ab	
Index	1	2	3	4	5	6	7	8	9	10	11	

LZW: encoding

- The decoder starts with just a dictionary of single symbols (like the encoder did).

Dict.	a	b	c	d	ba	ad	da
Index	1	2	3	4	5	6	7

- Then is receives the stream of phrase identifiers:

2 1 4 6 8 5 1 2

b a d ad ad?

ad a

- We call "ada" a *self-referential* phrase.

LZ77…

Relative Lempel-Ziv

- The Lempel-Ziv factorization (or parsing) breaks a string X of *n* symbols into *z* factors (or phrases).

- If the parsing is up to position i, then next phrase is either
  - X[i] – if symbol X[i] has not appeared before, or
  - X[i..j] – the longest substring starting at i and some $p_i < i$ in X

Relative Lempel-Ziv

- The Lempel-Ziv factorization (or parsing) breaks a string X of *n* symbols into *z* factors (or phrases).

- If the parsing is up to position i, then next phrase is either
  - X[i] – if symbol X[i] has not appeared before, or
  - X[i..j] – the longest substring starting at i and some p_i < i in X

1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	a	b	a

a | b | a | a b a | b a a b a |

(a,0) (b,0) (1,1) (1,3) (2,5)

Relative Lempel-Ziv

- LZ77's phrase length grows much faster than LZ78 and LZW

- One problem is efficiently finding the previous occurrences:
  - We have to search through the entire previously processed string at each point, not a summary dictionary as in the other schemes

- One solution: limit the dictionary to be some fixed size window immediately prior to the start of the current phrase (`gzip`)
  - generally degrades compression

- Another solution: maintain a suitable index data structure over the already processed string… or even the whole string (`7zip`)

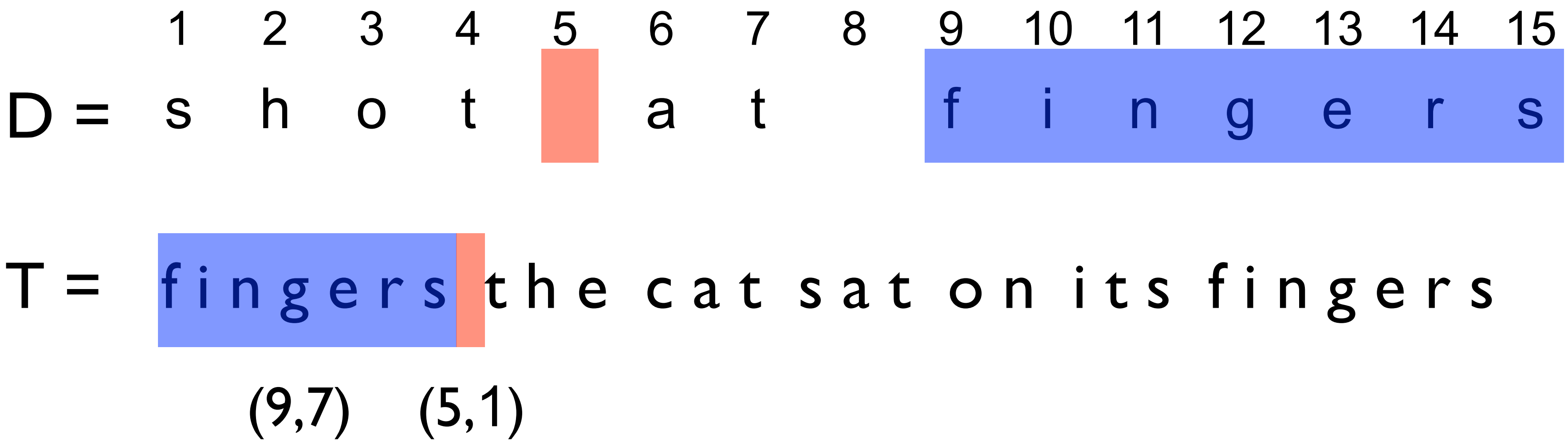
Relative Lempel-Ziv (RLZ)…

Relative Lempel-Ziv

- Very simple spin on LZ77 (kind of a *static LZ77*)
- Let D be a string of d symbols, called the *dictionary* or *reference*, and let T be the collection we want to compress
- Process T left-to-right and parse it into phrases…
- If the parsing is up to position i, to generate the next phrase we find the largest j such that T[i..j] occurs in D.
  - Output position of T[i..j] in D and length (j-i+1)
  - Continue parsing from position j+1.

Relative Lempel-Ziv

- If the parsing is up to position i, to generate the next phrase we find the largest j such that T[i..j] occurs in D.
  - Output position of T[i..j] in D and length (j-i+1)
  - Continue parsing from position j+1.



Relative Lempel-Ziv

- RLZ results in great compression when the dictionary matches the string (or strings) you're trying to compress, but is tragic otherwise
- Very fast for decoding large files (like LZ77)
- Advantage over LZ77 is that pointers to previous phrase occurrences are limited in the amount they can vary: they must point into the dictionary.
- When compressing massive collections, choose the dictionary so that it completely fills the available RAM

Grammar Compression…

Grammar compression

- Grammar compression represents the text as a context-free grammar.

  T = a_rose_is_a_rose_is_a_rose

  S ➔ ABB
  A ➔ a_rose
  B ➔ _is_A

- The grammar should generate exactly one string. Such a grammar is called a straight-line grammar because:
  - The are no branches; i.e., each non-terminal is the left-hand side of only one rule. Otherwise multiple strings could be generated.
  - The are no loops; i.e., no cyclic dependencies between non-terminals. Otherwise infinite strings could be generated.

Smallest grammar?

- The size of a grammar is the total length of the right-hand sides.
- The smallest grammar problem of computing the smallest straight-line grammar that generates a given string is NP hard.
- But there are algorithms for constructing small grammars, e.g.:
  - LZ78 parsing is easily transformed into a grammar with one rule for each phrase
  - The best approximation ratio O(log(n/g)), where g is the size of the smallest grammar, has been achieved by algorithms that transform the LZ77 parsing into a grammar
  - Greedy algorithms add one rule at a time as log as they find a new rule that reduces the size of the grammar.

Re-Pair

- Invented by Larrson and Moffat (2001)

Works as follows…

1. Find the pair of symbols XY that is the most frequent in the text T. If no pair occurs twice in T, stop.

2. Create a new non-terminal Z and add the rule Z ➔ XY to the grammar.

3. Replace all occurrences of XY in T by Z. Go to step 1.

Re-Pair (example)

1. Find most frequent pair XY. If no pair occurs twice in T, stop.
2. Create new non-terminal Z and add rule Z → XY to grammar.
3. Replace all occurrences of XY in T by Z. Go to step 1.

T = chchchanges_time_to_make_the_change_chchchanges

Rule added	Text after replacement
A → ch	AAAanges_time_to_make_the_Aange_AAAanges
B → e_	AAAanges_timBto_makBthBAangBAAAanges
C → Aa	AACnges_timBto_makBthBCngBAACnges
D → ng	AACDes_timBto_makBthBCDBAACDes
E → CD	AAEes_timBto_makBthBEBAAEes
F → es	AAEF_timBto_makBthBEBAAEF
…	…

Re-Pair: complexity

- The whole process can be performed in linear time using suitable data structures.

- We won't go into the detail here, but…

- The key observation is that, if n_{XY} is the number of occurrences of the most frequent pair XY in a given step, then the replacement reduces the size of the grammar by n_{XY}-2.

- Thus we can spend $O(n_{XY})$ time to perform the step (to achieve overall linear time).

Re-Pair: example encoding

- We could encode the output of Re-Pair as follows:
  - The number of rules, r, and the length, z, of the compressed text as γ codes.
  - The right-hand sides of rules using ceil(log(σ+i-1))-bit fixed length codes to encode the ith rule.
  - The compressed text using ceil(log(σ+r))-bit fixed length codes.
- Better compression can (probably) be achieved with a more sophisticated encoding.

Compression vs. decompression

- A common feature of most dictionary compression algorithms is asymmetry of compression and decompression:
  - The compressor needs to do lots of work choosing phrases or rules
  - The decompressor needs only to replace each phrase

- Thus the decompressor is often simple and fast*
  - LZ77-type methods are particularly simple and fast as they have no dictionary other than the already decoded part of the text
  - LZ78-type and grammar-based methods need some extra effort in constructing and accessing the dictionary

- Many implementations use simple encodings of the phrases and are optimized more for speed than maximum compression: being 2x faster is usually much more important than being 1% smaller.

Summary

- Dictionary-based compression comes in lots of varieties

- We will encounter LZ and grammar compression again when we look at compressed data structures, where we will augment these schemes so that they do more than just compression: they will allow us fast access and search in the compressed file.

Next lecture…

13/01	Shannon's Theorem
15/01	Huffman Coding
20/01	Integer Codes I
22/01	Integer Codes II
27/01	Dynamic Prefix Coding
29/01	Arithmetic Coding
03/02	Dictionary Compression
05/02	*No Lecture*
10/02	Burrows-Wheeler Transform
12/02	Compressed Data Structures
17/02	Compressed Data Structures
19/02	Compressed Data Structures
24/02	Compressed Data Structures

End