LZ77 Parsing, et c.

Simon J. Puglisi
University of Helsinki

Seminar on Data Compression Techniques

LZ77 is a lossless compression method discovered by Abraham Lempel and Jacob Ziv in 1977

Lossless

(We can get all the data back)

vs.

Lossy

(We can't necessarily)

Today

- LZ77 parsing (compression)

- Getting the data back (decompression)

- Variations on the basic scheme
  - Real compressors that use LZ
  - Rightmost parsing
  - Explicit literals
  - Relative pointers
  - Encoding the output of the parsing efficiently…

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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

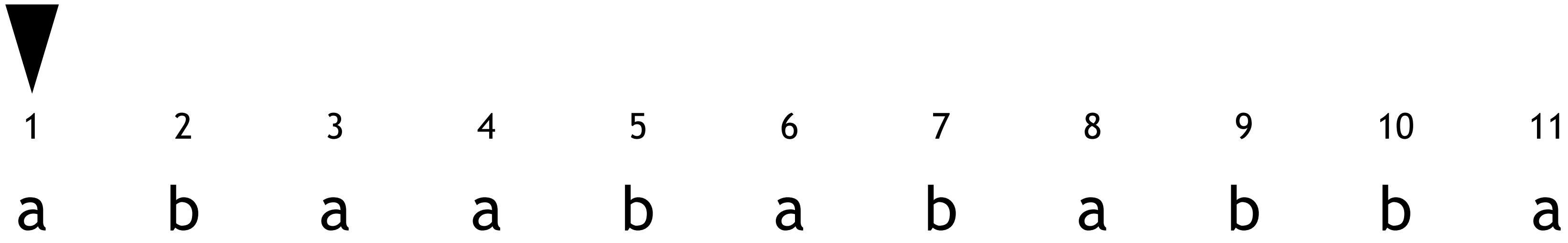
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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



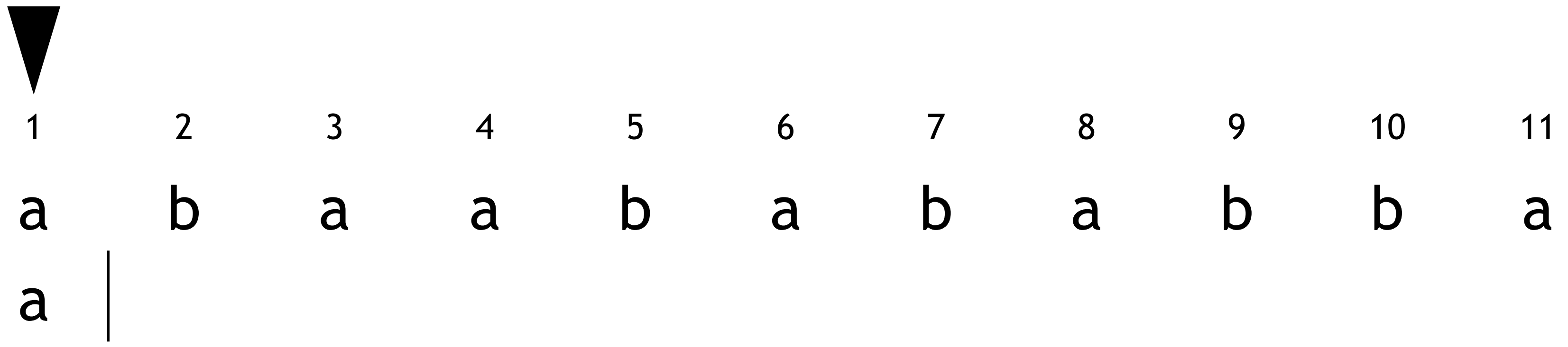
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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



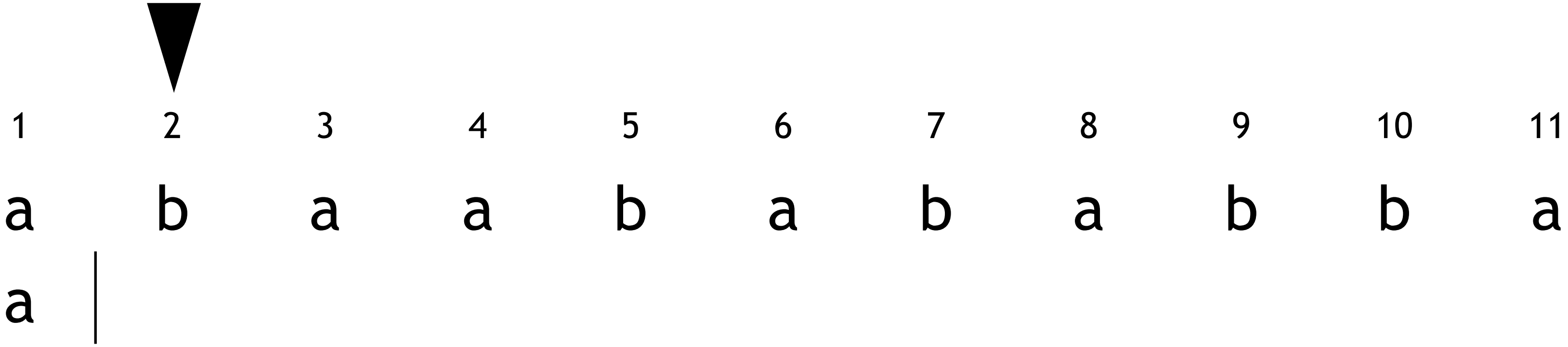
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a
a |										

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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



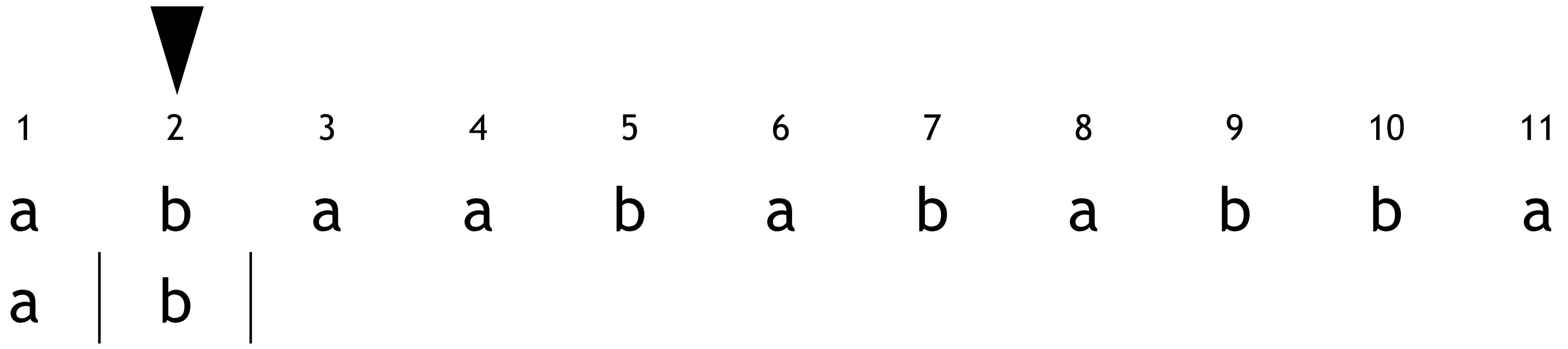
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a
a |										

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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



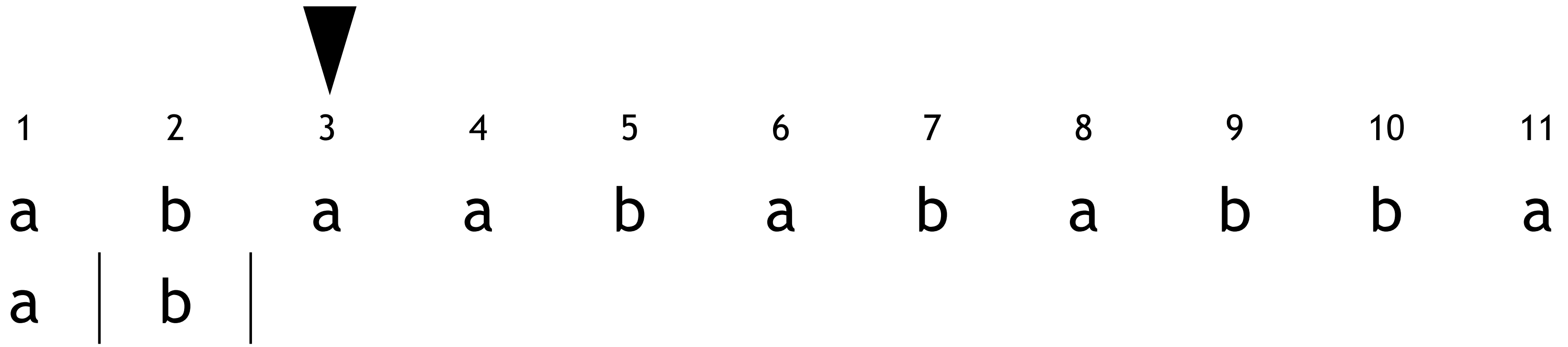
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a
a	b									

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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



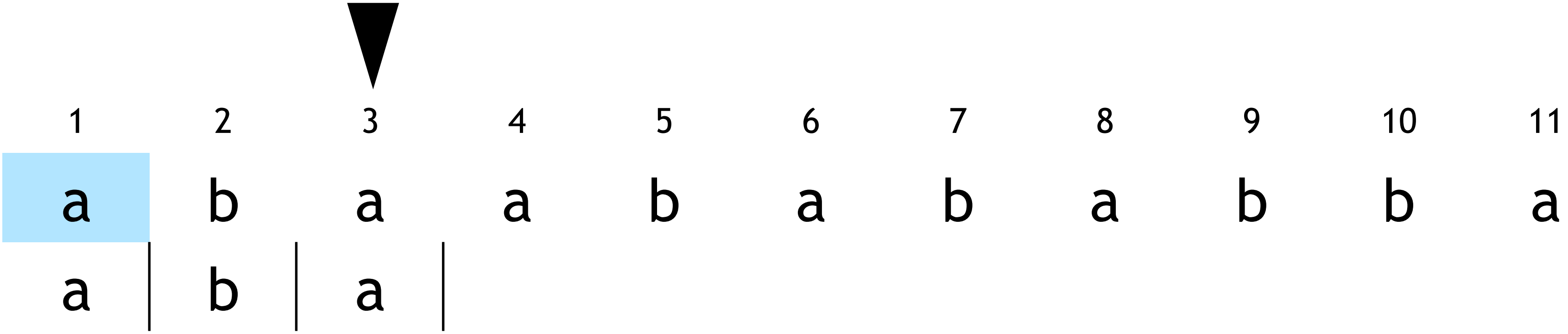
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a
a |	b |									

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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



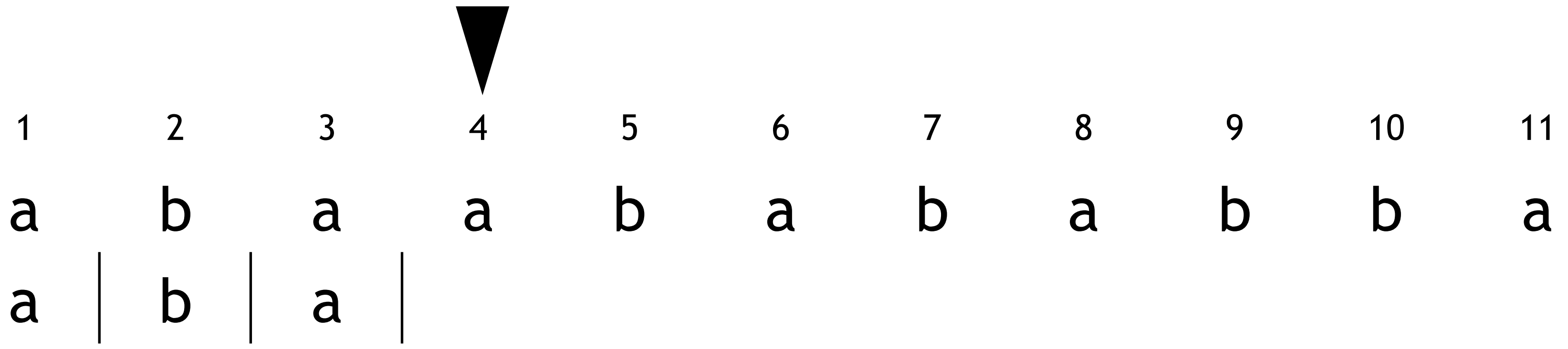
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a
a	b	a								

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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



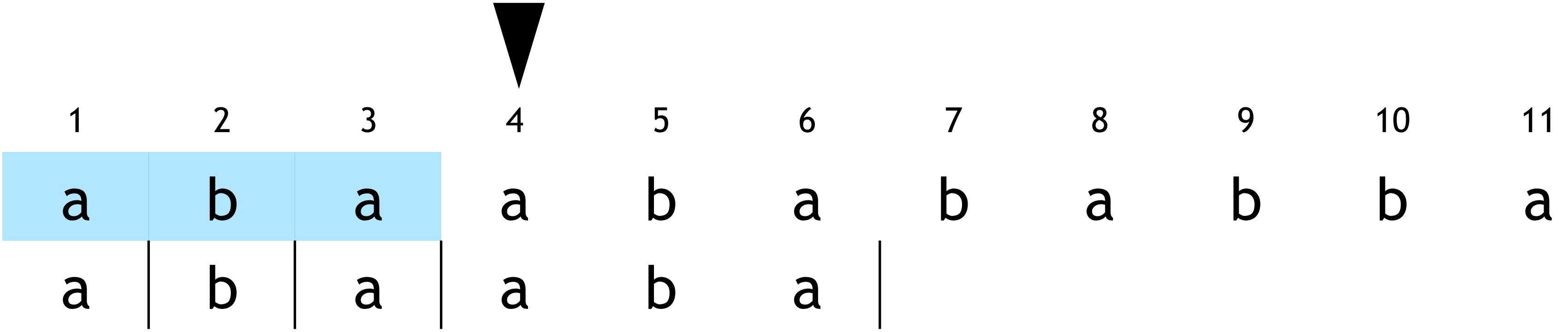
1	2	3	4	5	6	7	8	9	10	11
a	b	a	a	b	a	b	a	b	b	a
a	b	a								

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

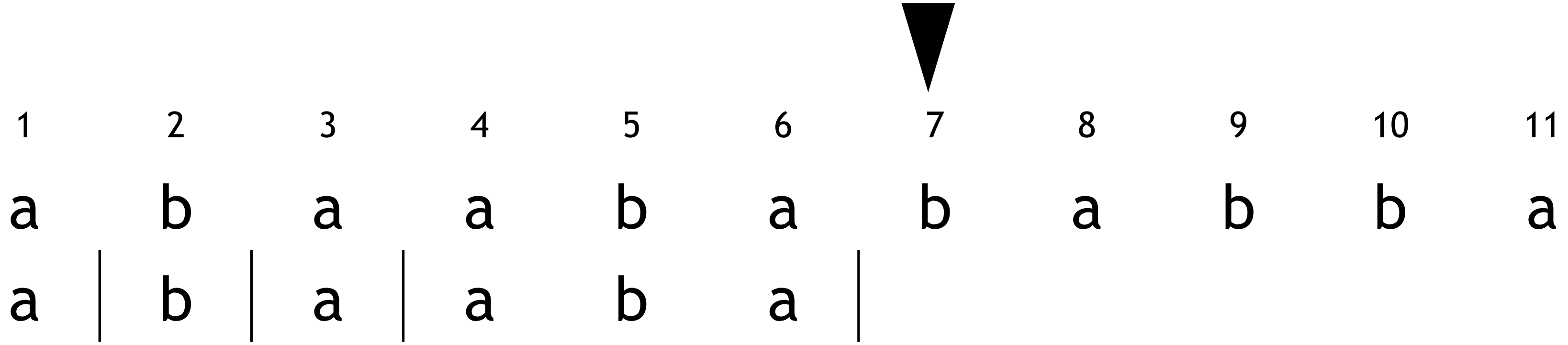
a | b | a | a b a |

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

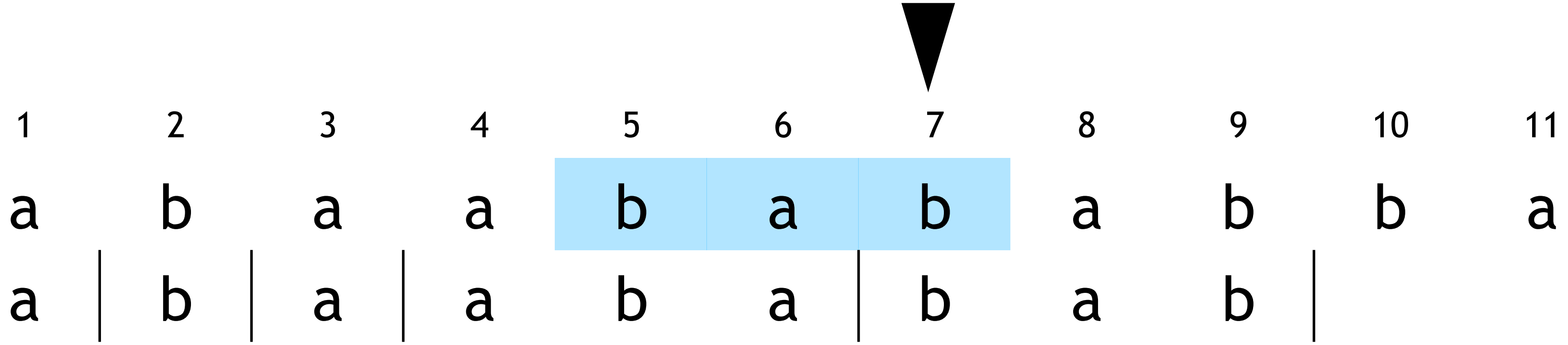
a | b | a | a b a |

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

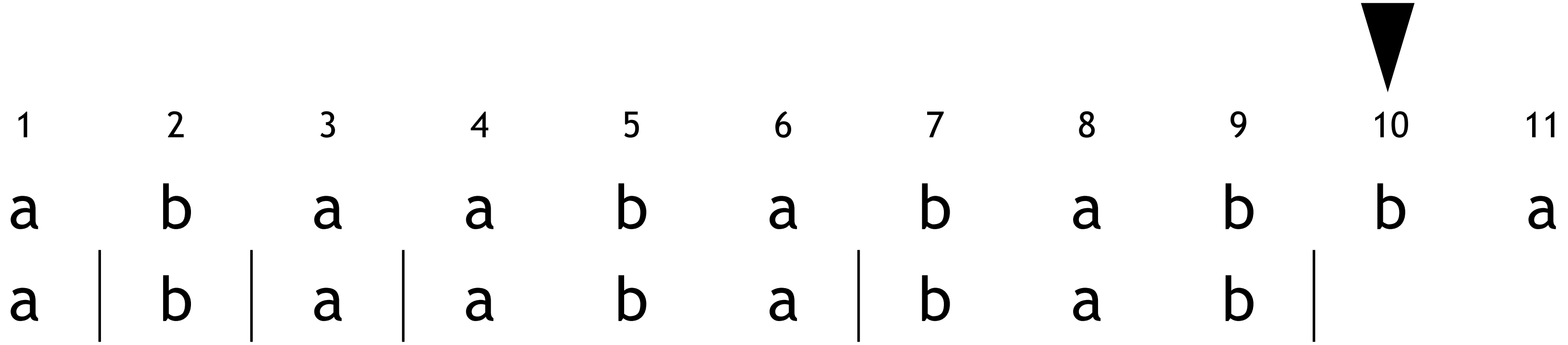
a | b | a | a b a | b a b |

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

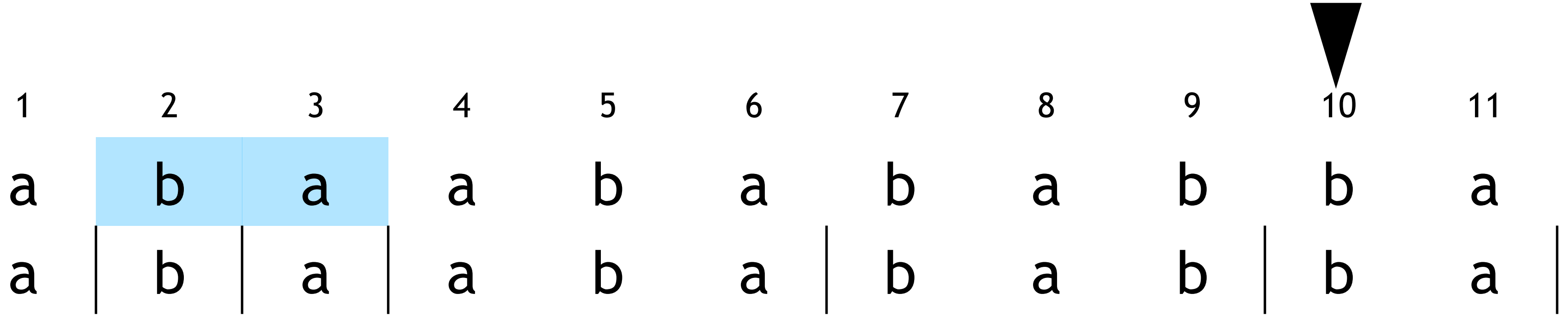
a | b | a | a b a | b a b |

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a
a	b	a	a	b	a	b	a	b	b	a

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

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

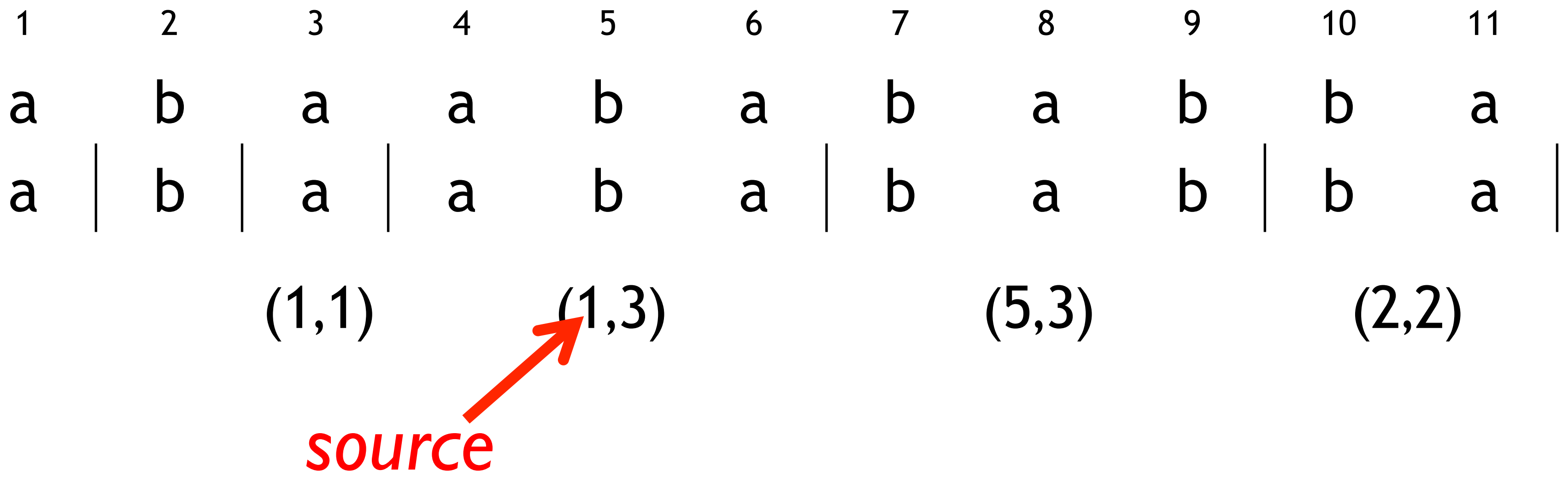
(1,1) (1,3) (5,3) (2,2)

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

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

(1,1) (1,3) (5,3) (2,2)

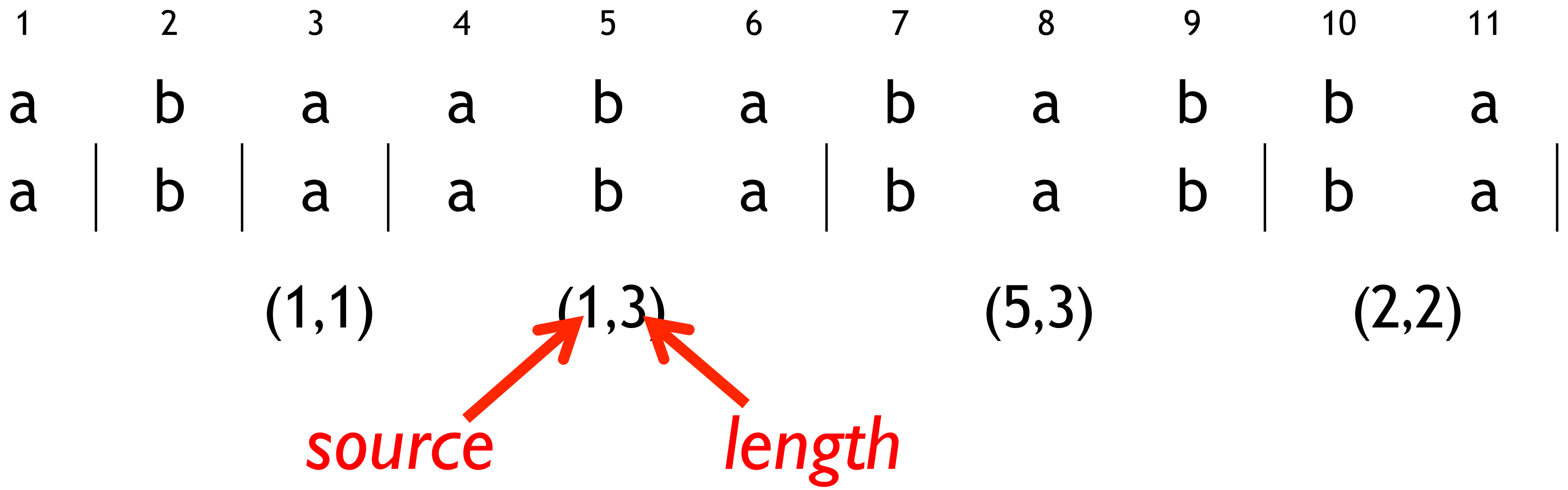
source

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

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

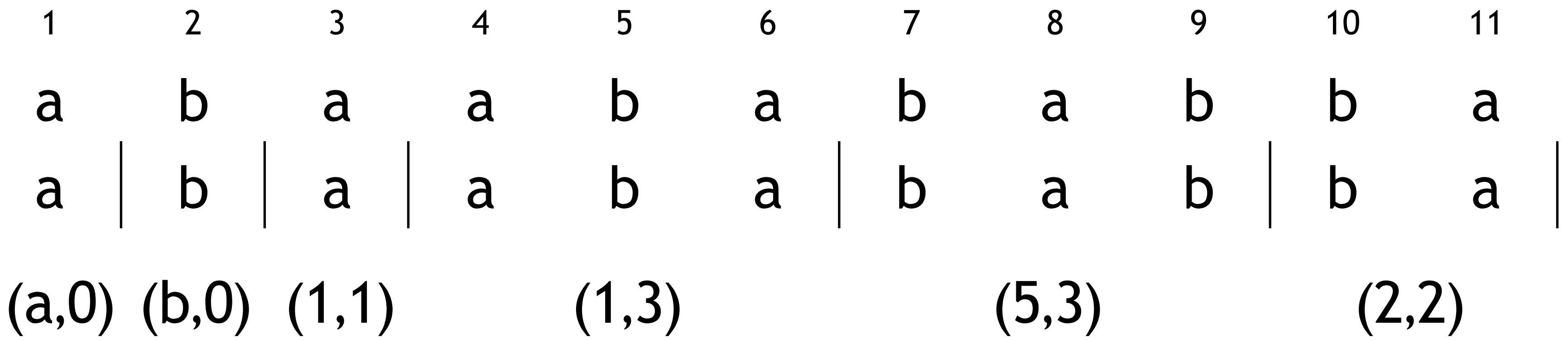
(1,1) (1,3) (5,3) (2,2)

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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 b b a

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

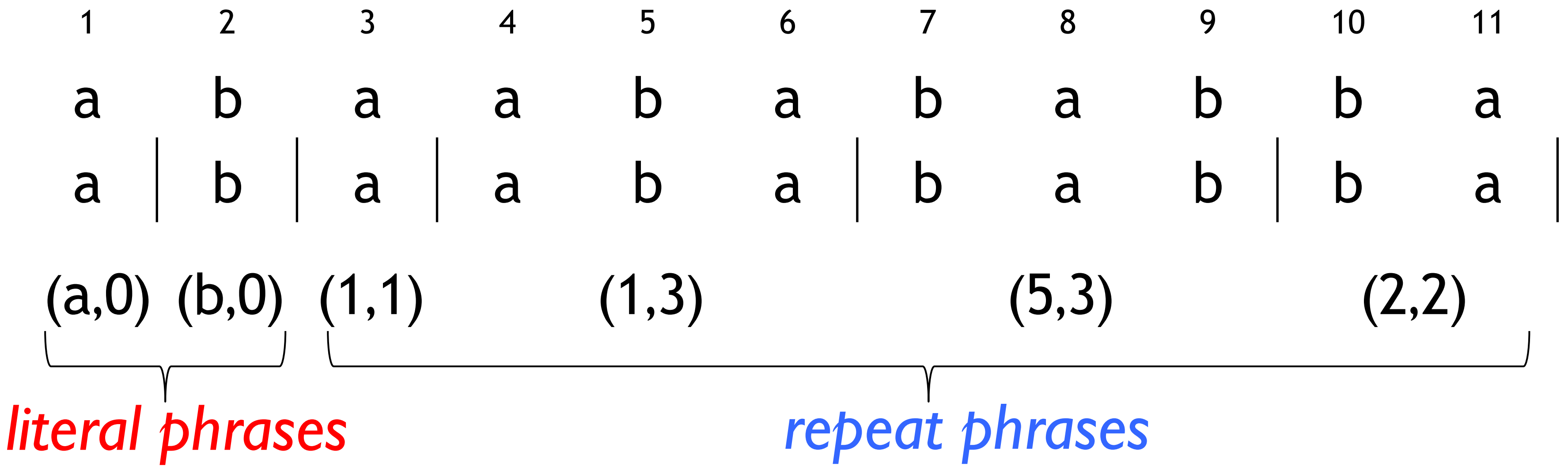
(a,0) (b,0) (1,1) (1,3) (5,3) (2,2)

LZ77 Parsing (Encoding)

The Lempel-Ziv parsing breaks a string X of n symbols into z phrases.

If the parsing is up to position i, then the 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	b	b	a

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

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

literal phrases

repeat phrases

Applications of LZ77

Major technique of lossless data compression for almost 40 years

- gzip, 7zip, LZ4 are some popular file compressors
- Powerful compression

Also a very handy algorithmic tool for string processing

- Key to algorithms and data structures for repetition detection, covering, approximate pattern matching, compressed text indexing, et c., et c.

Lots of research on how to compute the parsing efficiently…

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

▼

ababbaabababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

ababbaababababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

ababbaababababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

▼ ▼

ababbaabababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

ababbaabababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

ababbaabababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

▼ ▼

ababbaabababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

ababbaababababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

ababbaababababazababbzabbbz

Computing the parsing

Näive parsing algorithm just scans the already parsed portion of the string, looking for matches…

ababbaababababazababbzabbbz

…something like $O(n^2)$ time

Faster parsing

Most compressors maintain some data structure over the already parsed portion of the string to speed up matching

▼

ababbaabababazababbzabbbz

Faster parsing

Most compressors maintain some data structure over the already parsed portion of the string to speed up matching

- e.g. a hash table containing the locations of short matches

▼

ababbaabababazababbzabbbz

Faster parsing

Most compressors maintain some data structure over the already parsed portion of the string to speed up matching

aa: 6
ab: 1,3,7,9,11,15,17,21
ba: 2,5,8,10,12,16
bb: 4,18
…

▼
ababbaababababazababbzabbbz

Faster parsing

Most compressors maintain some data structure over the already parsed portion of the string to speed up matching

aa: 6
ab: 1,3,7,9,11,15,17,21
ba: 2,5,8,10,12,16
bb: 4,18
…

ababbaababababazababbzabbbz

Getting the data back…

(i.e. decompression)

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

▼

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

▼

ababbaabababa

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

▼

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

▼

ababbaababababaz

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

ababbaabababaz

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

ababbaabababaz

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

ababbaabababazababb

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- $O(n)$ time, $O(z)$ accesses to the already decoded part of the file

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

ababbaabababazababb

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

ababbaababababazababb

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

▼

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

▼

ababbaabababazababbzab

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

▼

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

▼

ababbaababababazababbzab

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

ababbaabababazababbzab

LZ Decoding

Decompressing the original file from the LZ phrases is simple & *fast*

- O(n) time, O(z) accesses to the already decoded part of the file

▼

(a,0),(b,0),(1,2),(2,2),(1,4),(1,3),(z,0),(1,5),(14,3),(4,2),(19,3),(14,2)
(26,4)(22,3)(1,3)(11,5)(13,4),(30,6)(1,1)

▼

ababbaabababazababbzabbb

LZ Decoding

```
c = 0
for each phrase (p,l)
     if l == 0 then /*literal phrase*/
          S[c++] = p
     else           /*repeat phrase*/
          for i = 0 to l-1 do
               S[c++] = S[p++]
```

Dealing with big files…

Practicalities with big files

- Big files cause difficulties for LZ77, for at least two reasons:
  - During compression: the index data structures used to perform parsing efficiently grow linearly with the input size – on big files they can start to exceed this size of memory
  - During decompression: copying characters from the already decompressed part of the file requires a random access to retrieve those characters. If the decompressed part of the file exceeds memory, this random access will become a disk access (which is very slow)
- Real compressors take shortcuts to deal with these situations…

Workaround (i.e. hack) for big inputs

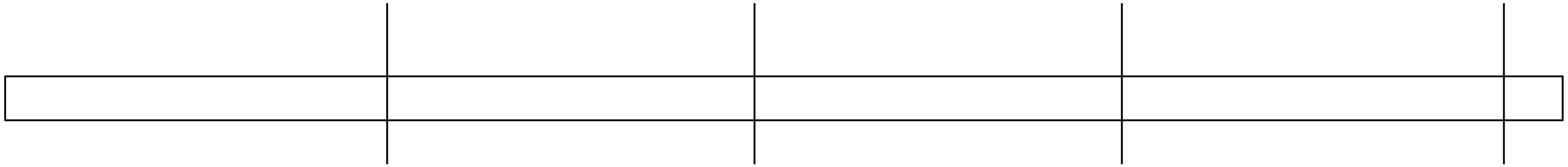
At compression time, break file into blocks, compress each block separately, source & phrase must be in same block

Workaround (i.e. hack) for big inputs

At compression time, break file into blocks, compress each block separately, source & phrase must be in same block

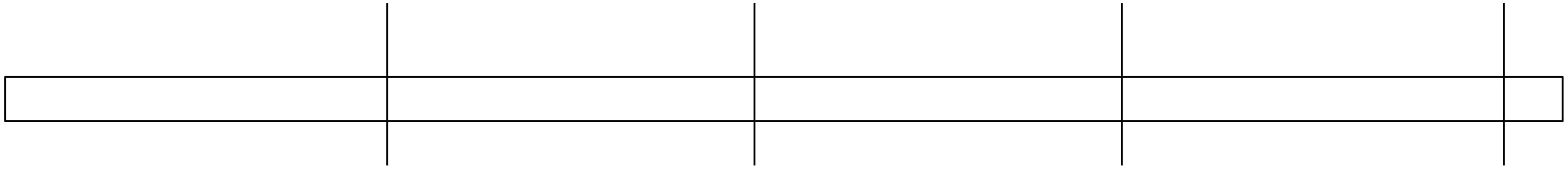
Workaround (i.e. hack) for big inputs

At compression time, break file into blocks, compress each block separately, source & phrase must be in same block



Workaround (i.e. hack) for big inputs

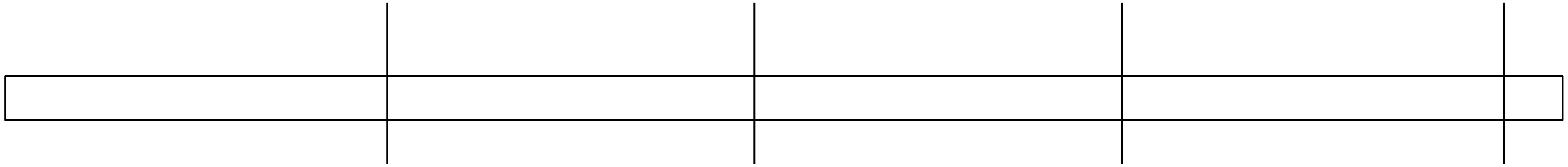
At compression time, break file into blocks, compress each block separately, source & phrase must be in same block



(.,.)(.,.)(.,.)…

Workaround (i.e. hack) for big inputs

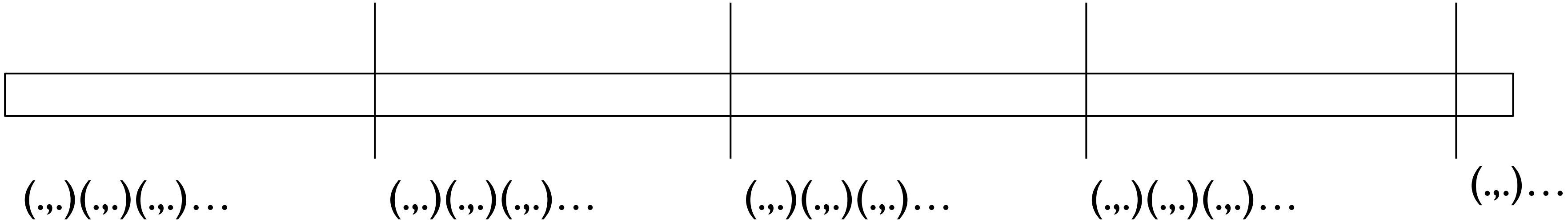
At compression time, break file into blocks, compress each block separately, source & phrase must be in same block



(.,.)(.,.)(.,.)…　　　　(.,.)(.,.)(.,.)…

Workaround (i.e. hack) for big inputs

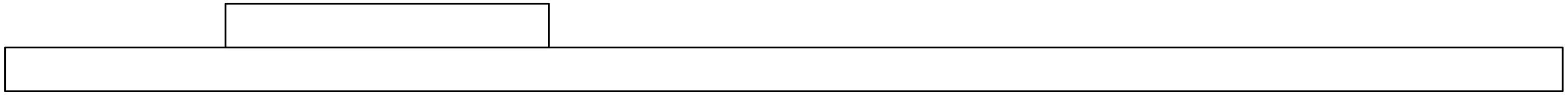
At compression time, break file into blocks, compress each block separately, source & phrase must be in same block



Similar idea: use a sliding window

A fixed-length window slides over the text as we parse

- Source for the next phrase must be inside the window



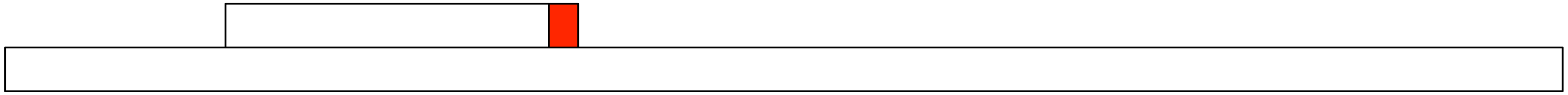
(.,.)(.,.)(.,.)…

(.,.)…

Similar idea: use a sliding window

A fixed-length windows slides over the text as we parse

- Source for the next phrase must be inside the window



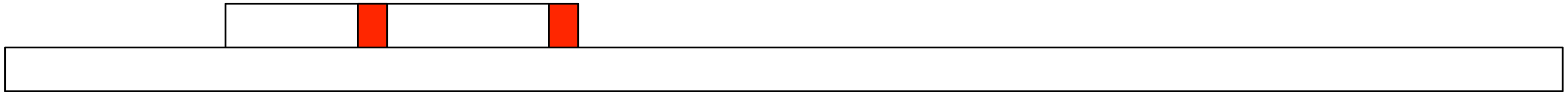
(.,.)(.,.)(.,.)…

(.,.)…

Similar idea: use a sliding window

A fixed-length windows slides over the text as we parse

- Source for the next phrase must be inside the window



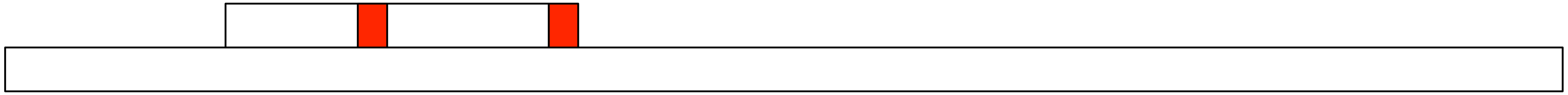
(.,.)(.,.)(.,.)…

(.,.)…

Similar idea: use a sliding window

A fixed-length windows slides over the text as we parse

- Source for the next phrase must be inside the window



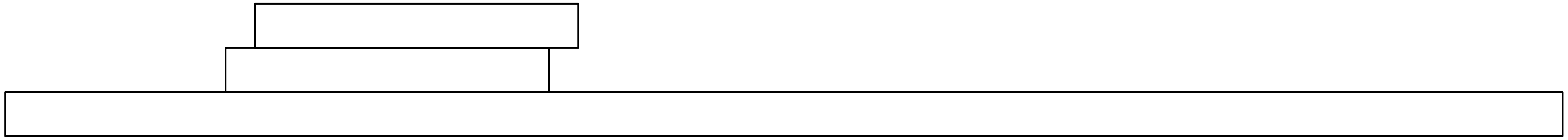
(.,.)(.,.)(.,.)…

(.,.)

Similar idea: use a sliding window

A fixed-length windows slides over the text as we parse

  – Source for the next phrase must be inside the window



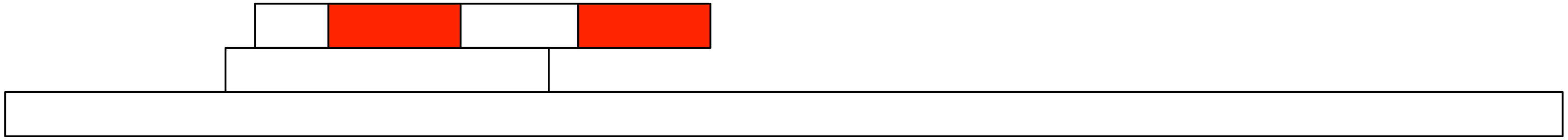
(.,.)(.,.)(.,.)…

(.,.)

Similar idea: use a sliding window

A fixed-length windows slides over the text as we parse

- Source for the next phrase must be inside the window



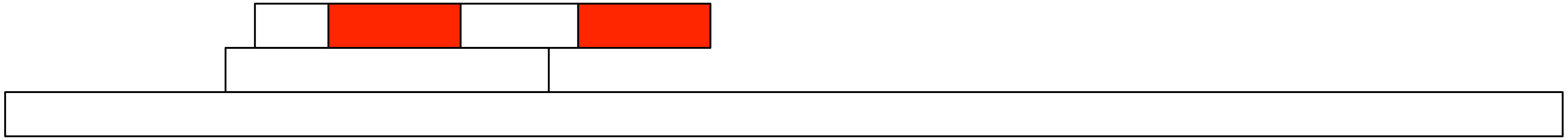
(.,.)(.,.)(.,.)…

(.,.)

Similar idea: use a sliding window

A fixed-length windows slides over the text as we parse

- Source for the next phrase must be inside the window



(.,.)(.,.)(.,.)…

(.,.)(.,.)

Similar idea: use a sliding window

A fixed-length windows slides over the text as we parse

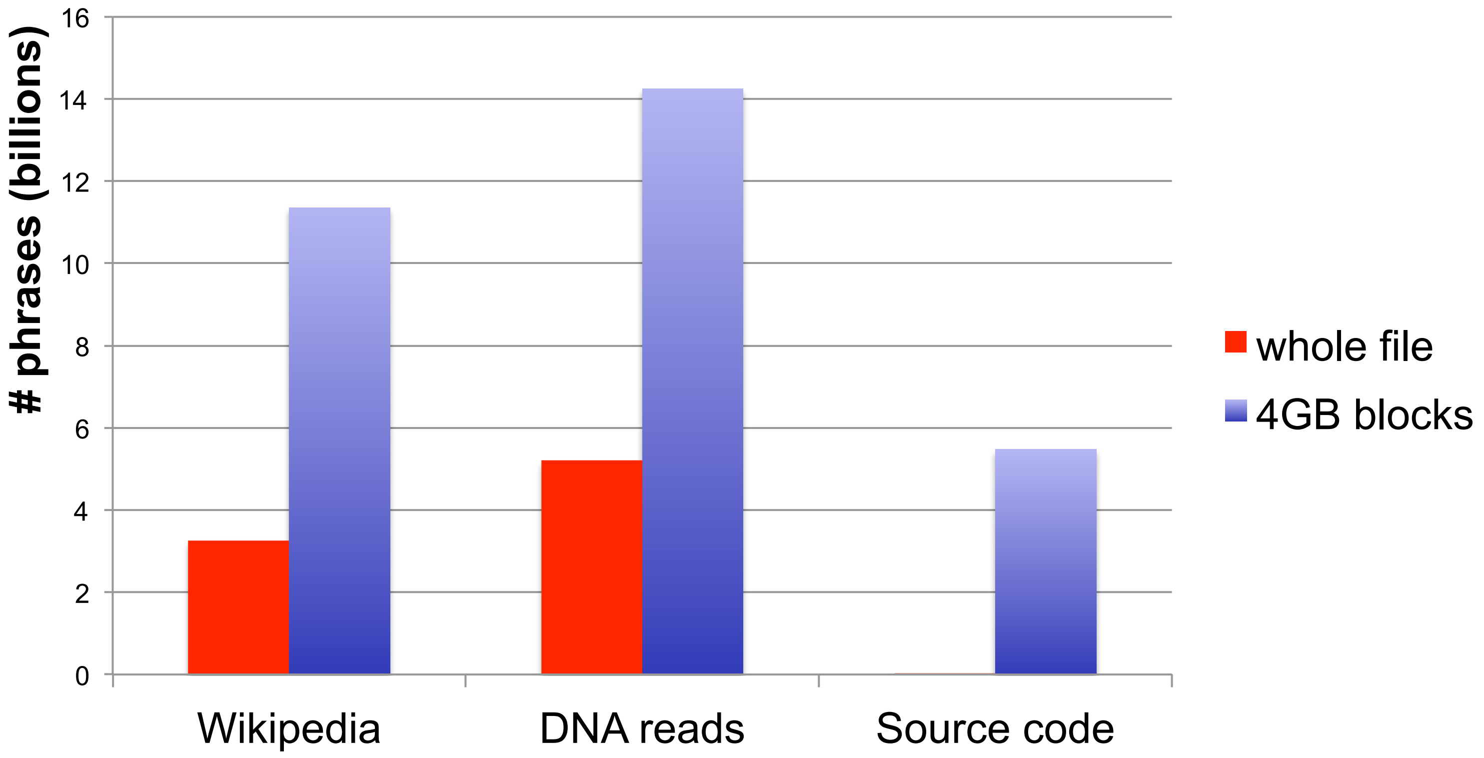
  – Source for the next phrase must be inside the window

(.,.)(.,.)(.,.)…

(.,.)(.,.)

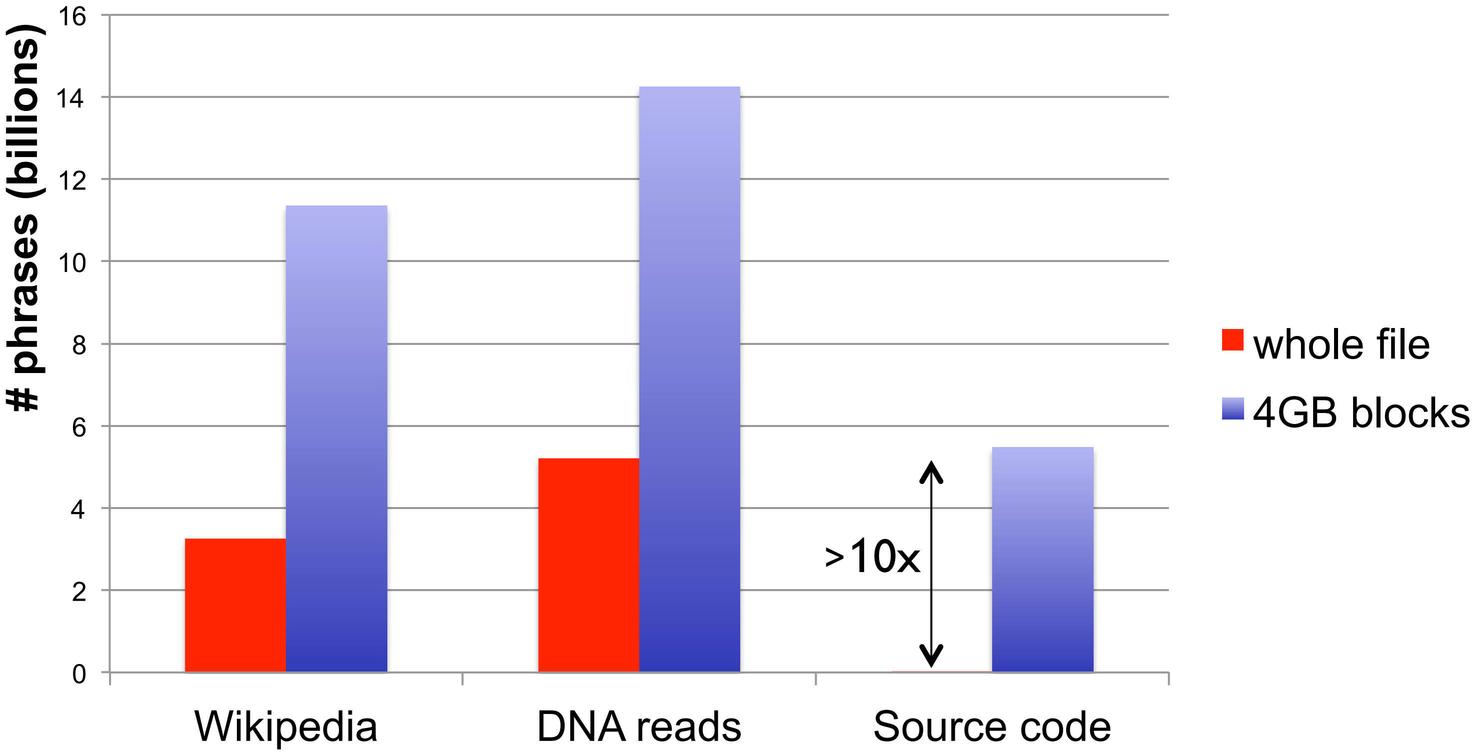
Workaround (i.e. hack) for big inputs

No free lunch…



Workaround (i.e. hack) for big inputs

No free lunch…



Variations to the basic scheme…

Variations

- The greedy parsing algorithm minimizes # of phrases…
- …but often we are more interested in minimizing the total size of the compressed file.
- This forces us to think about how (pos,len) pairs are encoded,
  - and whether we should use the (pos,len) representation of a phrase at all.

Variations: encode short strings as literals

- Naive encoding: store (pos,len) as two 32-bit integers
  - This equates to 8 bytes per phrase
  - If we assume each character of the input needs 1 byte...
  - ...it’s only worth encoding a phrase as (pos,len) if its “fairly” long

Variations: encode short strings as literals

- Naive encoding: store (pos,len) as two 32-bit integers
  - This equates to 8 bytes per phrase
  - If we assume each character of the input needs 1 byte...
  - ...it’s only worth encoding a phrase as (pos,len) if its “fairly” long

… (14,3),(4,12),(19,3),(14,26),(26,18),(22,2)…

Variations: encode short strings as literals

- Naive encoding: store (pos,len) as two 32-bit integers
  - This equates to 8 bytes per phrase
  - If we assume each character of the input needs 1 byte...
  - ...it's only worth encoding a phrase as (pos,len) if its "fairly" long

… (14,3),(4,12),(19,3),(14,26),(5,18),(22,2)…

… (-3, ktt),(12,4),(-3, vav),(26,14),(18,5),(-2, nc)…

Variations: encode triples rather than pairs

- Often in practice, a long match is broken by a single mismatching character
  - In the parsing we've discussed, this will produce three phrases

actcgcagagcgcgcagagccctac ……... actcgcagagcgcacagagccctat

Variations: encode triples rather than pairs

- Often in practice, a long match is broken by a single mismatching character
  - In the parsing we've discussed, this will produce three phrases

actcgcagagcgcgcagagccctac …..... actcgcagagcgcacagagccctat

(1,13),(1, 2),(12,10)

Variations: encode triples rather than pairs

- Often in practice, a long match is broken by a single mismatching character
  - In the parsing we've discussed, this will produce three phrases

actctgcagagcgcgcagagccctac ……... actcgcagagcgcacagagccctat

(1,13),(1, 2),(12,10)

(1,13,a), (12,10,t) …

Variations…

- Many more tricks used in actual compressors…
- E.g., store distance to previous match rather than position of previous match
  - Numbers tend to be smaller, so use less bits
- E.g., store positions relative to the previous phrase's position
  - These tend to be correlated

Variations…

- Many more tricks used in actual compressors…
- E.g., store distance to previous match rather than position of previous match
  - Numbers tend to be smaller, so use less bits
- E.g., store positions relative to the previous phrase's position
  - These tend to be correlated
- Whatever the output of the parsing algorithm (some stream of integers and chars) it needs to be <u>coded</u> efficiently…