A brief tour of practical data compression

DEFLATE

francois.cayre@grenoble-inp.fr

July 16th, 2015

Why ?

Data compression typical use cases

- Archiving :
  `$ tar cvf - directory | gzip > archive.tar.gz`
- Transparent bandwith optimization : `HTTP/1.1` [RFC2616, Sec. 3.5].
- Image compression (`GIF`, `PNG`).

Why ?

Data compression typical use cases

- Archiving :
  `$ tar cvf - directory | gzip > archive.tar.gz`
- Transparent bandwith optimization : `HTTP/1.1` [RFC2616, Sec. 3.5].
- Image compression (`GIF`, `PNG`).

Design consequences

- Separate *pure compression* methods from *formatting* specs.
  - Auxiliary functions like CRC provided above the formatting level.
- `DEFLATE` is an ubiquitous compression method, valid for : `gzip`, `ZIP`, `PDF`, `PNG`, `PPP`, `HTTP/1.1`, *etc.*

What ?

Archivers

- **Archivers group files together into a single one : the archive.**
  Programs : `ar`, `tar`, `cpio`, `pax`.
  - Basically, bytes concatenation and metadata addition.
  - Enable *solid* compression (references to bytes in other files).

What ?

Archivers vs. compressors

- **Archivers group files together into a single one : the archive.**
  Programs : `ar`, `tar`, `cpio`, `pax`.
  - Basically, bytes concatenation and metadata addition.
  - Enable *solid* compression (references to bytes in other files).
- **Compressors perform the actual removal of redundancy.**
  Programs : `pack`, `compress`, `gzip`, `bzip2`, `xz`.
  Incipit tragœdia.

Unix KISS motto : treat separate issues with dedicated tools.

When ?

Timeline of relevant Unix compression programs

Name	Technology	Extension	Date released
`pack`	Huffman	`.z`	<1990
`compact`	Adaptive Huffman	`.C`	?
`compress`	LZW	`.Z`	1984

When ?

Timeline of relevant Unix compression programs

Name	Technology	Extension	Date released
`pack`	Huffman	`.z`	<1990
`compact`	Adaptive Huffman	`.C`	?
`compress`	LZW	`.Z`	1984
`gzip`	LZ77, Huffman	`.gz`	1996
`bzip2`	RLE, BWT, MTF, Huffman	`.bz2`	1997
`xz`	LZMA, range encoding	`.xz`	2009

How ?

Redundancy ?

Let X be a source of symbols drawn from $\mathcal{A}$, its relative redundancy is :

$$R = 1 - \frac{H(X)}{\log|\mathcal{A}|}.$$

"Maximum achievable compression rate".

How ?

Redundancy ?

Let X be a source of symbols drawn from $\mathcal{A}$, its relative redundancy is :

$$R = 1 - \frac{H(X)}{\log|\mathcal{A}|}.$$

"Maximum achievable compression rate".

– *Thanks for the thermometer, but I really need to* `DEFLATE`.

How ?

Types of redundancy [Welch, 1984]

- Character distribution : Huffman, range encoding ;
- Character repetition : RLE ;
- High-usage patterns : Lempel-Ziv ;
- Positional redundancy : Lempel-Ziv, if at all.

How ?

Types of redundancy [Welch, 1984]

- Character distribution : Huffman, range encoding ;
- Character repetition : RLE ;
- High-usage patterns : Lempel-Ziv ;
- Positional redundancy : Lempel-Ziv, if at all.

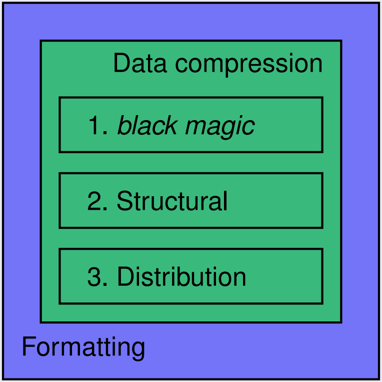
Engineering a compressor

Since LZ can emulate RLE (cf. *infra*) and it also produces symbols :

1. Optionally, pre-process data with black magic (BWT, MTF) ;
2. Remove structural redundancy (frequent patterns) with LZ ;
3. Remove character distribution redundancy on LZ symbols.

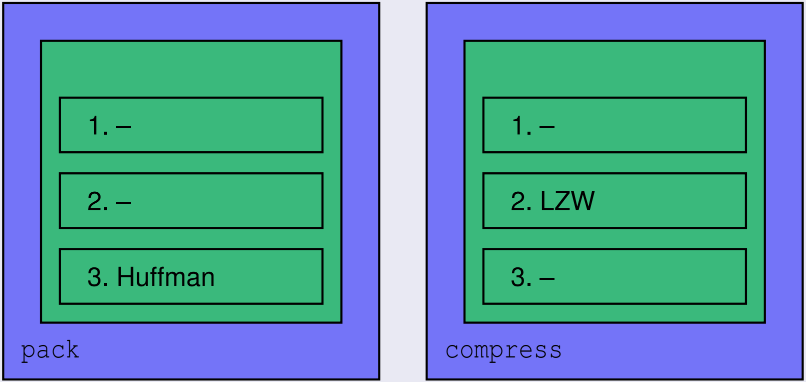
Architecture of a compressor

A compressor is better seen as a *pipeline* with optional stages



Gallery

Compression, *ca.* 1985



Gallery

Compression, *ca.* 2010

DEFLATE

1. –
2. LZ77
3. Huffman

`gzip`, `ZIP`

LZMA

1. BWT, MTF...
2. LZ77 *on steroids*
3. Range encoding

`xz`, `7z`

Why DEFLATE ?

A handful of excellent reasons

- It's a complete tool, it just lacks top-notch features of LZMA ;
- It's fully documented [RFC1951, RFC1952] ;
- It's free (as in free beer *and* as in free world, allegedly).

Easiest access to real-world use of LZ.

Why DEFLATE ?

A handful of excellent reasons

- It's a complete tool, it just lacks top-notch features of LZMA ;
- It's fully documented [RFC1951, RFC1952] ;
- It's free (as in free beer *and* as in free world, allegedly).

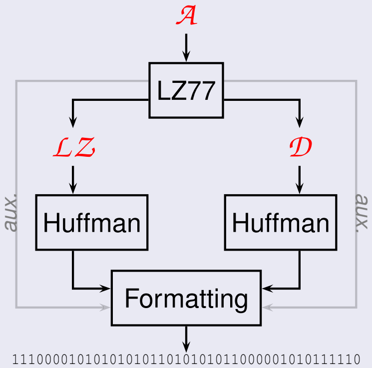
Easiest access to real-world use of LZ.

A good way of learning a lot (the hard way)

- How is Huffman coding really done ? How to tweak the LZ stage ?
- *"One point that appears to be little appreciated in the literature is that there is no disadvantage incurred, and considerable benefit to be gained, from mapping the source alphabet onto integer symbol numbers [...]"* [Moffat & Turpin, 1997].

Structure of DEFLATE

Use two Huffman coders for different data series



The Lempel-Ziv split

Timeline of papers

- J. Ziv & A. Lempel, “A Universal Algorithm for Sequential Data Compression", 1977.
- J. Ziv & A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding", 1978.

The Lempel-Ziv split

Timeline of papers and implementations

- J. Ziv & A. Lempel, “A Universal Algorithm for Sequential Data Compression", 1977.
- J. Ziv & A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding", 1978.
  LZ78 : Explicit construction of dictionary.
  Implementation : T. Welch, “A Technique for High-Performance Data Compression", 1984 (`LZW`).

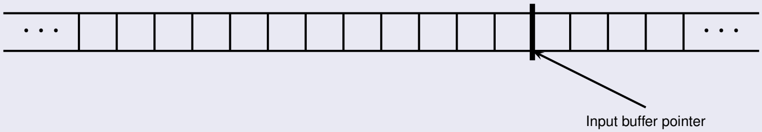
The Lempel-Ziv split

Timeline of papers and implementations

- J. Ziv & A. Lempel, “A Universal Algorithm for Sequential Data Compression", 1977.
  LZ77 : Sliding window.
  Implementation : P. Katz, `DEFLATE`, 1993 (PKZIPv2).
- J. Ziv & A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding", 1978.
  LZ78 : Explicit construction of dictionary.
  Implementation : T. Welch, “A Technique for High-Performance Data Compression", 1984 (`LZW`).
  Rationale : LZ78 easiest to implement on memory-limited devices than LZ77 (on-chip transparent compression, explicit flush of bounded-size dictionary).

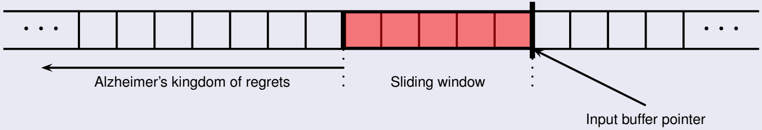
LZ77 in two lines

The big picture



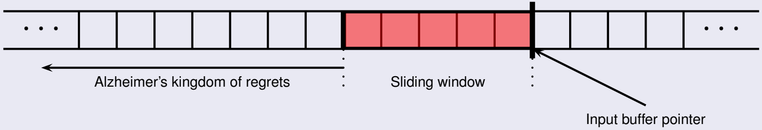
LZ77 in two lines

The big picture



LZ77 in two lines

The big picture



Algorithm

1. Search for known past strings starting with the same next literals.
2. If none is found, output the next literal.
   Otherwise, output a reference to the *best* past string.

LZ77 in practice

Symbols

- Input
  Sequence of literals in $\mathcal{A} = [\texttt{0x00..0xff}]$

LZ77 in practice

Symbols

- Input
  Sequence of literals in $\mathcal{A} = [\texttt{0x00..0xff}]$
- Output
  Sequence of :

`{L(lit)}`	:	The literal `lit` $\in \mathcal{A}$
`{Z(-dist->len)}`	:	A reference to `len` bytes that are `dist` bytes backwards

LZ77 in practice

Symbols

- **Input**
  Sequence of literals in $\mathcal{A} = [\texttt{0x00..0xff}]$
- **Output**
  Sequence of :

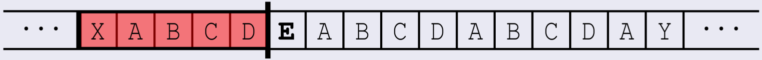
`{L(lit)}`	:	The literal `lit` $\in \mathcal{A}$
`{Z(-dist->len)}`	:	A reference to `len` bytes that are `dist` bytes backwards

Notes

- if `len` > `dist`, repeat modulo `dist` from `-dist` (RLE++ for free).
- `dist` is limited to the last 32KiB (size of the sliding window).
- `xz`/LZMA : Sliding window size of 4GiB (1st shot of steroids).

LZ77 in pictures

Emit literal {L('E')}



LZ77 in pictures

Emit literal {L('E')}

···	X	A	B	C	D	**E**	A	B	C	D	A	B	C	D	A	Y	···

Emit reference {Z(-5->4)}

···	X	A	B	C	D	E	**A**	**B**	**C**	**D**	A	B	C	D	A	Y	···

LZ77 in pictures

Emit literal {L('E')}

···	X	A	B	C	D	**E**	A	B	C	D	A	B	C	D	A	Y	···

Emit reference {Z(−5−>4)}

···	X	A	B	C	D	E	**A**	**B**	**C**	**D**	A	B	C	D	A	Y	···

Emit reference {Z(−4−>5)} – RLE emulation

···	X	A	B	C	D	E	A	B	C	D	**A**	**B**	**C**	**D**	**A**	Y	···

LZ77 in details

Dictionary search

- Various *best* strings :
  - Latest string : favor Huffman stage
  - Longest string
  - Lazy : does emitting literals first allow to find a longer string ?
    - Kind of future-aware lookup.

LZ77 in details

Dictionary search

- Various *best* strings :
  - Latest string : favor Huffman stage
  - Longest string
  - Lazy : does emitting literals first allow to find a longer string ?
    - Kind of future-aware lookup.
- Internal data structure : hash chains (array of linked lists).
- Hash computed on the next 3 literals.

LZ77 in details

Dictionary search

- Various *best* strings :
  - Latest string : favor Huffman stage
  - Longest string
  - Lazy : does emitting literals first allow to find a longer string ?
    - Kind of future-aware lookup.
- Internal data structure : hash chains (array of linked lists).
- Hash computed on the next 3 literals.
- `xz`/LZMA (2nd shot of steroids) :
  - Data structure : array of binary search trees.
  - Multi-threaded search based on the next 2, 3 or 4 literals.
  - Hundreds of MiBs of RAM frequently needed.
  - Cannot always afford it (kernel-specific stripped-down version).

Sample run (LZ77 stage)

Strategy : newest string (to be continued)

a {L(0x61)} b {L(0x62)} c {L(0x63)} d {L(0x64)} abc {Z(-4->3)} abc {Z(-3->3)} dabc {Z(-7->4)} bcd {Z(-6->3)}

Encoding LZ symbols

Encoding literals and reference lengths (producing up to 5 aux. bits)

`lit`'s and `len`'s share the same alphabet $\mathcal{LZ} = \mathcal{A} \cup [256..285]$

$\mathcal{A}$	Literal		266	Lengths 13-14	+1 bit	277	Lengths 67-82	+4 bits
256	`EndOfBlock`		267	Lengths 15-16	+1 bit	278	Lengths 83-98	+4 bits
257	Length 3		268	Lengths 17-18	+1 bit	279	Lengths 99-114	+4 bits
258	Length 4		269	Lengths 19-22	+2 bits	280	Lengths 115-130	+4 bits
259	Length 5		270	Lengths 23-26	+2 bits	281	Lengths 131-162	+5 bits
260	Length 6		271	Lengths 27-30	+2 bits	282	Lengths 163-194	+5 bits
261	Length 7		272	Lengths 31-34	+2 bits	283	Lengths 195-226	+5 bits
262	Length 8		273	Lengths 35-42	+3 bits	284	Lengths 227-257	+5 bits
263	Length 9		274	Lengths 43-50	+3 bits	285	Length 258	
264	Length 10		275	Lengths 51-58	+3 bits			
265	Lengths 11-12	+1 bit	276	Lengths 59-66	+3 bits			

Encoding LZ symbols

Encoding reference distances (producing up to 13 aux. bits)

`dist`'s enjoy their separate alphabet $\mathcal{D} = [0..29]$

0	1	
1	2	
2	3	
3	4	
4	5-6	+1 bit
5	7-8	+1 bit
6	9-12	+2 bits
7	13-16	+2 bits
8	17-24	+3 bits
9	25-32	+3 bits
10	33-48	+4 bits
11	49-64	+4 bits
12	65-96	+5 bits
13	97-128	+5 bits
14	129-192	+6 bits

15	193-256	+6 bits
16	257-384	+7 bits
17	385-512	+7 bits
18	513-768	+8 bits
19	769-1024	+8 bits
20	1025-1536	+9 bits
21	1537-2048	+9 bits
22	2049-3072	+10 bits
23	3073-4096	+10 bits
24	4097-6144	+11 bits
25	6145-8192	+11 bits
26	8193-12288	+12 bits
27	12289-16384	+12 bits
28	16385-24576	+13 bits
29	24577-32768	+13 bits

Huffman coding

Timeline of papers

- D.A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes", 1952
  The basic idea. Problem : How to (avoid to ?) transmit frequency tables ?

Huffman coding

Timeline of papers

- D.A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes", 1952
  The basic idea. Problem : How to (avoid to ?) transmit frequency tables ?
- D. Knuth, “Dynamic Huffman Coding", 1985
  Adaptive Huffman : Start from an empty p.m.f., use `NewSymbol` and update frequencies and codes from time to time.

Huffman coding

Timeline of papers

- D.A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes", 1952
  The basic idea. Problem : How to (avoid to ?) transmit frequency tables ?
- D. Knuth, “Dynamic Huffman Coding", 1985
  Adaptive Huffman : Start from an empty p.m.f., use `NewSymbol` and update frequencies and codes from time to time.
- A. Moffat & A. Turpin, “On the Implementation of Minimum Redundancy Prefix Codes", 1997
  Canonical Huffman : It is sufficient to send the lengths of the code.

Canonical Huffman codes

Apply an *ordering* constraint on classical Huffman code lengths

The Huffman codes used for each alphabet in the `DEFLATE` format have two additional rules [RFC1951, Sec. 3.2.2] :

1. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent ;
2. Shorter codes lexicographically precede longer codes.

Canonical Huffman codes

Apply an *ordering* constraint on classical Huffman code lengths

The Huffman codes used for each alphabet in the DEFLATE format have two additional rules [RFC1951, Sec. 3.2.2] :

1. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent ;
2. Shorter codes lexicographically precede longer codes.

Computing Huffman codes for real

1. Get classical Huffman codes

A	00
B	1
C	011
D	010

Canonical Huffman codes

Apply an *ordering* constraint on classical Huffman code lengths

The Huffman codes used for each alphabet in the `DEFLATE` format have two additional rules [RFC1951, Sec. 3.2.2] :

1. All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent ;
2. Shorter codes lexicographically precede longer codes.

Computing Huffman codes for real

1. Get classical Huffman codes → **2. Canonicalize**

A	00	10
B	1	0
C	011	110
D	010	111

Sample run ($\mathcal{LZ}$ excerpt)

Yes, it's ASCII art. Important things are in red, and so are codes.

```
Queue of 7 freqs: [ 1 1 1 1 1 1 3 [
Huffman tree 0x1706f08:
              ┌──0──1
        ┌──0──+
        │     └──1──1
  ┌──0──+
  │     │     ┌──0──1
  │     └──1──+
  │           └──1──1
  +
  │           ┌──0──1
  │     ┌──0──+
  │     │     └──1──1
  └──1──+
        └──1──3
Max code length: 3 bits
Freq/len: 1/3 1/3 1/3 1/3 1/3 1/3 3/2
```

```
Huffman encoder 0x1703dc0:
  - 286 symbols with freqs/codelen  0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1/3 1/3 1/3 1/3 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1/3 3/2 1/3 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
  - Codes: 0x61::010(3) 0x62::011(3)
0x63::100(3) 0x64::101(3) 0x100::110(3)
0x101::00(2) 0x102::111(3)
```

Encoding canonical Huffman codes

Another real world example of code lengths sequence

- 0 0 0 0 0 0 0 0 0 0 **2** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . . 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 **2** **2** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . . 0 0 0 0 0 0 0 0 0 0 **3**
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **3**
- Now how *redundant* is *this*?

Encoding canonical Huffman codes

Another real world example of code lengths sequence

- `0 0 0 0 0 0 0 0 0 0` **2** `0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0`
  `0 0 0 0 0 0 0 0 0 0` **2 2** `0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0` **3**
  `0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0` **3**
- Now how *redundant* is *this*?

Canonical Huffman codes alphabet $\mathcal{H}$

0-15	*Represent code* lengths of `0-15`.
16	*Copy the previous code* length 3-6 times. The next 2 bits indicate repeat length (0 = 3, ... , 3 = 6).
17	*Repeat a code* length of 0 for 3-10 times (3 bits of length).
18	*Repeat a code* length of 0 for 11-138 times (7 bits of length).

Sample run ($\mathcal{H}$ excerpt)

Huffman codes for coding Huffman codes ;-)

```
Queue of 4 freqs: [ 2 2 3 5 [
Huffman tree 0x1709488:
┌──0──5
+
│       ┌──0──3
└──1──+
        │       ┌──0──2
        └──1──+
                └──1──2
Max code length: 3 bits
Freq/len: 5/1 3/2 2/3 2/3
Huffman encoder 0x1706ea0:
  - 19 symbols with freqs/codelen 0 0 2/3 3/2 0 0 0 0 0 0 0 0 0 0 0 0 2/3 0 5/1
```

DEFLATE block format

Formatting imposes most of the implementation structure

1. Block header

1 bit	BFINAL (1 iff last block)
2 bits	BTYPE (10 for dynamic Huffman codes)
5 bits	HLIT $= \|\mathcal{LZ}\| - 257$ (257-286)
5 bits	HDIST $= \|\mathcal{D}\| - 1$ (1-32)
4 bits	HCLEN $= \|\mathcal{H}\| - 4$ (4-19)
(HCLEN + 4) x 3 bits	Lengths for $\mathcal{H}$ (*)

   HLIT + 257 code lengths for $\mathcal{LZ}$, using Huffman codes for $\mathcal{H}$

   HDIST + 1 code lengths for $\mathcal{D}$, using Huffman codes for $\mathcal{H}$

(*) "These code lengths are interpreted as 3-bit integers (0-7), in the order : 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15."

DEFLATE block format

Formatting imposes most of the implementation structure

1. Block header

1 bit	BFINAL (1 iff last block)
2 bits	BTYPE (10 for dynamic Huffman codes)
5 bits	HLIT $= \|\mathcal{LZ}\| - 257$ (257-286)
5 bits	HDIST $= \|\mathcal{D}\| - 1$ (1-32)
4 bits	HCLEN $= \|\mathcal{H}\| - 4$ (4-19)
(HCLEN + 4) x 3 bits	Lengths for $\mathcal{H}$ (*)

   HLIT + 257 code lengths for $\mathcal{LZ}$, using Huffman codes for $\mathcal{H}$

   HDIST + 1 code lengths for $\mathcal{D}$, using Huffman codes for $\mathcal{H}$

2. Compressed block data, using Huffman codes for $\mathcal{LZ}$ and $\mathcal{D}$

(*) "These code lengths are interpreted as 3-bit integers (0-7), in the order : 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15."

DEFLATE block format

Formatting imposes most of the implementation structure

1. Block header

1 bit	BFINAL (1 iff last block)
2 bits	BTYPE (10 for dynamic Huffman codes)
5 bits	HLIT $= \|\mathcal{LZ}\| - 257$ (257-286)
5 bits	HDIST $= \|\mathcal{D}\| - 1$ (1-32)
4 bits	HCLEN $= \|\mathcal{H}\| - 4$ (4-19)
(HCLEN + 4) x 3 bits	Lengths for $\mathcal{H}$ (*)

   HLIT + 257 code lengths for $\mathcal{LZ}$, using Huffman codes for $\mathcal{H}$
   HDIST + 1 code lengths for $\mathcal{D}$, using Huffman codes for $\mathcal{H}$

2. Compressed block data, using Huffman codes for $\mathcal{LZ}$ and $\mathcal{D}$
3. `EndOfBlock`, using the Huffman codes of $\mathcal{LZ}$

(*) "These code lengths are interpreted as 3-bit integers (0-7), in the order : 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15."

Sample run (putting it all together)

1/3 Writing block header info (final, dynamic Huffman)

```
{1} {10}
```

Sample run (putting it all together)

1/3 Writing block header parms (HLIT, HDIST, HCLEN)

{**1**} {**10**} 29+257 {**10111|**} 29+1 {**10111**} 12+4 {**001|1**}

Sample run (putting it all together)

1/3 Writing block header, fixed-width Huffman lengths

{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]

Sample run (putting it all together)

1/3 Writing block header, variable-width Huffman lengths

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1)
```

Sample run (putting it all together)

1/3 Writing block header, variable-width Huffman lengths

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10]
```

Sample run (putting it all together)

1/3 Writing block header, variable-width Huffman lengths

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|)
```

Sample run (putting it all together)

1/3 Writing block header, variable-width Huffman lengths

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|) (18,127) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0] (1111111|)
```

Sample run (putting it all together)

1/3 Writing block header, variable-width Huffman lengths

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|) (18,127) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0] (1111111|) (18,6) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0]
(0110000|) 3 [10] 2 [110] 3 [10] (18,18) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0|] (0100100) 2 [1|10] (16,0) * * * [111] (00) (18,13) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 [0|] (1011000) (wrote 286+30 code lengths) >ed >9d >87 >00 >00 >00 >00 >c2 >58 >3b
>fe >0c >2d >92 >1d
```

Sample run (putting it all together)

2/3 Writing compressed data

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|) (18,127) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0] (1111111|) (18,6) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0]
(0110000|) 3 [10] 2 [110] 3 [10] (18,18) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0|] (0100100) 2 [1|10] (16,0) * * * [111] (00) (18,13) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 [0|] (1011000) (wrote 286+30 code lengths) >ed >9d >87 >00 >00 >00 >00 >c2 >58 >3b
>fe >0c >2d >92 >1d
{L(0x61)} [0|10] >0d
```

Sample run (putting it all together)

2/3 Writing compressed data

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|) (18,127) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0] (1111111|) (18,6) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0]
(0110000|) 3 [10] 2 [110] 3 [10] (18,18) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0|] (0100100) 2 [1|10] (16,0) * * * [111] (00) (18,13) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 [0|] (1011000) (wrote 286+30 code lengths) >ed >9d >87 >00 >00 >00 >00 >c2 >58 >3b
>fe >0c >2d >92 >1d
{L(0x61)} [0|10] >0d {L(0x62)} [011] {L(0x63)} [100|] >39 {L(0x64)} [101] {Z(-4->3)} [00] [01]
{Z(-3->3)} [0|0] [00] >45
```

Sample run (putting it all together)

2/3 Writing compressed data

{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|) (18,127) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0] (1111111|) (18,6) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0]
(0110000|) 3 [10] 2 [110] 3 [10] (18,18) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0|] (0100100) 2 [1|10] (16,0) * * * [111] (00) (18,13) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 [0|] (1011000) (wrote 286+30 code lengths) **>ed >9d >87 >00 >00 >00 >00 >c2 >58 >3b**
>fe >0c >2d >92 >1d
{L(0x61)} [0|10] **>0d** {L(0x62)} [011] {L(0x63)} [100|] **>39** {L(0x64)} [101] {Z(-4->3)} [00] [01]
{Z(-3->3)} [0|0] [00] **>45** {Z(-7->4)} [111] [11|] (0) **>f8**

Sample run (putting it all together)

2/3 Writing compressed data

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|) (18,127) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0] (1111111|) (18,6) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0]
(0110000|) 3 [10] 2 [110] 3 [10] (18,18) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0|] (0100100) 2 [1|10] (16,0) * * * [111] (00) (18,13) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 [0|] (1011000) (wrote 286+30 code lengths) >ed >9d >87 >00 >00 >00 >00 >c2 >58 >3b
>fe >0c >2d >92 >1d
{L(0x61)} [0|10] >0d {L(0x62)} [011] {L(0x63)} [100|] >39 {L(0x64)} [101] {Z(-4->3)} [00] [01]
{Z(-3->3)} [0|0] [00] >45 {Z(-7->4)} [111] [11|] (0) >f8 {Z(-6->3)} [00] [10] (1)
```

Sample run (putting it all together)

3/3 Writing EndOfBlock

```
{1} {10} 29+257 {10111|} 29+1 {10111} 12+4 {001|1}
3 [110] 0 [000] 1 [1|00] 0 [000] 0 [000|] 0 [000] 0 [000] 0 [00|0] 0 [000] 0 [000] 0 [0|00] 0
[000] 0 [000|] 2 [010] 0 [000] 3 [11|0]
(18,86) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0] (011010|1) 3 [10] (16,0) * * * [111] (00|) (18,127) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0] (1111111|) (18,6) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [0]
(0110000|) 3 [10] 2 [110] 3 [10] (18,18) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 [0|] (0100100) 2 [1|10] (16,0) * * * [111] (00) (18,13) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 [0|] (1011000) (wrote 286+30 code lengths) >ed >9d >87 >00 >00 >00 >00 >c2 >58 >3b
>fe >0c >2d >92 >1d
{L(0x61)} [0|10] >0d {L(0x62)} [011] {L(0x63)} [100|] >39 {L(0x64)} [101] {Z(-4->3)} [00] [01]
{Z(-3->3)} [0|0] [00] >45 {Z(-7->4)} [111] [11|] (0) >f8 {Z(-6->3)} [00] [10] (1) {L(0x100)}
[11|0] {Flushing 7 bits} 0000000| >e8 >00
```

Building blocks and formatting

Setting the block size and chaining blocks

- Use fixed-size blocks of LZ symbols, may flush block early based on dictionary analysis.
- References may spawn across blocks.
- Blocks are output one after another.

Building blocks and formatting

Setting the block size and chaining blocks

- Use fixed-size blocks of LZ symbols, may flush block early based on dictionary analysis.
- References may spawn across blocks.
- Blocks are output one after another.

Formatting mantra

“In other words, if one were to print out the compressed data as a sequence of bytes, starting with the first byte at the *right* margin and proceeding to the *left*, with the most-significant bit of each byte on the left as usual, one would be able to parse the result from right to left, with fixed-width elements in the correct MSB-to-LSB order and Huffman codes in bit-reversed order (i.e., with the first bit of the code in the relative LSB position).” [RFC1951]

Building blocks and formatting

Setting the block size and chaining blocks

- Use fixed-size blocks of LZ symbols, may flush block early based on dictionary analysis.
- References may spawn across blocks.
- Blocks are output one after another.

Formatting mantra

*“In other words, if one were to print out the compressed data as a sequence of bytes, starting with the first byte at the *right* margin and proceeding to the *left*, with the most-significant bit of each byte on the left as usual, one would be able to parse the result from right to left, with fixed-width elements in the correct MSB-to-LSB order and Huffman codes in bit-reversed order (i.e., with the first bit of the code in the relative LSB position)."* [RFC1951]

Sample run (summary)

Printing some interesting figures

```
Gzip (39 runs)

INFLATE (167 runs)
  - 1 blocks
  - Compression summary (80.38% preamble bits):
      Preamble: 15*8 + 7 bits
      Data    : 3*8 + 7 bits
      Total   : 19*8 + 6 bits
      Ratio   : 109.72% (18->20 bytes)

Lempel-Ziv coder (8 runs)
  - 17 bytes -> 7 LZ symbols
  - Dictionnary (16 hash chains, 15 keys): 0 1 0 0 1 0 0 0 2 0 0 7 0 0 3 1
  - Longest string: 4 bytes

CRC (17 runs)
  - Value: b0fdc05f
```

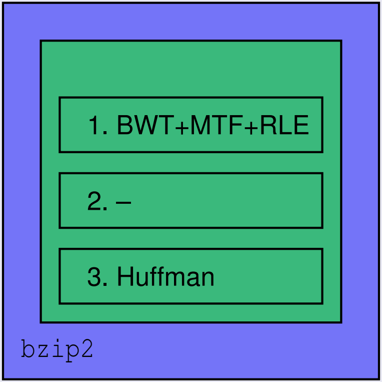
EndOfTalk

Closing words

A reader finally reaching this point (sweating profusely with such deep concentration on so many details) may respond with the single word "insane." This scheme of Phil Katz for compressing the two prefix-code tables per block is devilishly complex and hard to follow, but it works ![Salomon, p. 238]

Structure of bzip2

Roots dating back to 1983



How Kolmogorov rescued Shannon (take #2)

Assessing entropy coding alone

- *– I should definitely exploit local correlations !*

How Kolmogorov rescued Shannon (take #2)

Assessing entropy coding alone

- *– I should definitely exploit local correlations !*
  Structural vs. *combinatorial* processing before entropy coding.
- *Combinatorial* black magic
  Often advertised as *free lunch* (from Shannon's point of view).
  In practice, you often have to pay a small $\log(n)$ fee.

How Kolmogorov rescued Shannon (take #2)

Assessing entropy coding alone

- *– I should definitely exploit local correlations !*
  Structural vs. *combinatorial* processing before entropy coding.
- *Combinatorial* black magic
  Often advertised as *free lunch* (from Shannon's point of view).
  In practice, you often have to pay a small $\log(n)$ fee.

How to make local correlations shine ?

1. Block-sorting transform (BWT)
   Bijectively re-order data flow favoring local correlations.

How Kolmogorov rescued Shannon (take #2)

Assessing entropy coding alone

- – *I should definitely exploit local correlations !*
  Structural vs. *combinatorial* processing before entropy coding.
- *Combinatorial* black magic
  Often advertised as *free lunch* (from Shannon's point of view).
  In practice, you often have to pay a small $\log(n)$ fee.

How to make local correlations shine ?

1. Block-sorting transform (BWT)
   Bijectively re-order data flow favoring local correlations.
2. Entropic regularization (MTF)
   Adapt encoding of the above ***to exhibit a distribution more amenable to compression***.

The Burrows-Wheeler transform (BWT)

Forward re-ordering of $n = 11$ input literals ($n = |\mathtt{ABRACADABRA}|$)

Init

ABRACADABRA

The Burrows-Wheeler transform (BWT)

Forward re-ordering of $n = 11$ input literals ($n = |\texttt{ABRACADABRA}|$)

Compute all circular permutations

```
ABRACADABRA
AABRACADABR
RAABRACADAB
BRAABRACADA
ABRAABRACAD
DABRAABRACA
ADABRAABRAC
CADABRAABRA
ACADABRAABR
RACADABRAAB
BRACADABRAA
```

The Burrows-Wheeler transform (BWT)

Forward re-ordering of $n = 11$ input literals ($n = |\texttt{ABRACADABRA}|$)

Sort lexicographically

ABRACADABRA	AABRACADABR
AABRACADABR	ABRAABRACAD
RAABRACADAB	ABRACADABRA
BRAABRACADA	ACADABRAABR
ABRAABRACAD	ADABRAABRAC
DABRAABRACA	BRAABRACADA
ADABRAABRAC	CADABRAABRA
CADABRAABRA	DABRAABRACA
ACADABRAABR	BRACADABRAA
RACADABRAAB	RAABRACADAB
BRACADABRAA	RACADABRAAB

The Burrows-Wheeler transform (BWT)

Forward re-ordering of $n = 11$ input literals ($n = |\texttt{ABRACADABRA}|$)

Prepend #perm to **output**

ABRACADABRA	AABRACADAB**R**
AABRACADABR	ABRAABRACA**D**
RAABRACADAB	ABRACADABR**A** **#2**
BRAABRACADA	ACADABRAAB**R**
ABRAABRACAD	ADABRAABRA**C**
DABRAABRACA	BRAABRACAD**A**
ADABRAABRAC	CADABRAABR**A**
CADABRAABRA	DABRAABRAC**A**
ACADABRAABR	BRACADABRA**A**
RACADABRAAB	RAABRACADA**B**
BRACADABRAA	RACADABRAA**B**

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Init

R
D
A
R
C
A
A
A
A
B
B

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\text{RDARCAAAABB}|$)

Sort

A
A
A
A
A
B
B
C
D
R
R

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

```
RA
DA
AA
RA
CA
AB
AB
AC
AD
BR
BR
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

```
AA
AB
AB
AC
AD
BR
BR
CA
DA
RA
RA
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

```
RAA
DAB
AAB
RAC
CAD
ABR
ABR
ACA
ADA
BRA
BRA
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

AAB
ABR
ABR
ACA
ADA
BRA
BRA
CAD
DAB
RAA
RAC

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

RAAB
DABR
AABR
RACA
CADA
ABRA
ABRA
ACAD
ADAB
BRAA
BRAC

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\mathtt{RDARCAAAABB}|$)

Sort

AABR
ABRA
ABRA
ACAD
ADAB
BRAA
BRAC
CADA
DABR
RAAB
RACA

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

```
RAABR
DABRA
AABRA
RACAD
CADAB
ABRAA
ABRAC
ACADA
ADABR
BRAAB
BRACA
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

```
AABRA
ABRAA
ABRAC
ACADA
ADABR
BRAAB
BRACA
CADAB
DABRA
RAABR
RACAD
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

```
RAABRA
DABRAA
AABRAC
RACADA
CADABR
ABRAAB
ABRACA
ACADAB
ADABRA
BRAABR
BRACAD
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

AABRAC
ABRAAB
ABRACA
ACADAB
ADABRA
BRAABR
BRACAD
CADABR
DABRAA
RAABRA
RACADA

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\text{RDARCAAAABB}|$)

Paste

RAABRAC
DABRAAB
AABRACA
RACADAB
CADABRA
ABRAABR
ABRACAD
ACADABR
ADABRAA
BRAABRA
BRACADA

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

```
AABRACA
ABRAABR
ABRACAD
ACADABR
ADABRAA
BRAABRA
BRACADA
CADABRA
DABRAAB
RAABRAC
RACADAB
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

```
RAABRACA
DABRAABR
AABRACAD
RACADABR
CADABRAA
ABRAABRA
ABRACADA
ACADABRA
ADABRAAB
BRAABRAC
BRACADAB
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

AABRACAD
ABRAABRA
ABRACADA
ACADABRA
ADABRAAB
BRAABRAC
BRACADAB
CADABRAA
DABRAABR
RAABRACA
RACADABR

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

```
RAABRACAD
DABRAABRA
AABRACADA
RACADABRA
CADABRAAB
ABRAABRAC
ABRACADAB
ACADABRAA
ADABRAABR
BRAABRACA
BRACADABR
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

```
AABRACADA
ABRAABRAC
ABRACADAB
ACADABRAA
ADABRAABR
BRAABRACA
BRACADABR
CADABRAAB
DABRAABRA
RAABRACAD
RACADABRA
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

RAABRACADA
DABRAABRAC
AABRACADAB
RACADABRAA
CADABRAABR
ABRAABRACA
ABRACADABR
ACADABRAAB
ADABRAABRA
BRAABRACAD
BRACADABRA

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Sort

```
AABRACADAB
ABRAABRACA
ABRACADABR
ACADABRAAB
ADABRAABRA
BRAABRACAD
BRACADABRA
CADABRAABR
DABRAABRAC
RAABRACADA
RACADABRAA
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Paste

RAABRACADAB
DABRAABRACA
AABRACADABR
RACADABRAAB
CADABRAABRA
ABRAABRACAD
ABRACADABRA
ACADABRAABR
ADABRAABRAC
BRAABRACADA
BRACADABRAA

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\mathrm{RDARCAAAABB}|$)

Sort

```
AABRACADABR
ABRAABRACAD
ABRACADABRA
ACADABRAABR
ADABRAABRAC
BRAABRACADA
BRACADABRAA
CADABRAABRA
DABRAABRACA
RAABRACADAB
RACADABRAAB
```

The Burrows-Wheeler transform (BWT)

Reverse re-ordering of n input literals ($n = |\texttt{RDARCAAAABB}|$)

Reached n literals, selecting output block #2

```
AABRACADABR
ABRAABRACAD
ABRACADABRA
ACADABRAABR
ADABRAABRAC
BRAABRACADA
BRACADABRAA
CADABRAABRA
DABRAABRACA
RAABRACADAB
RACADABRAAB
```

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output

Encoding RDARCAAAABB

Stack ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output	0

Encoding RDARCAAAABB

Stack	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output	17

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack ABCDEFGHIJKLMNOPQRSTUVWXYZ

Output 0,1

Encoding RDARCAAAABB

Stack RABCDEFGHIJKLMNOPQSTUVWXYZ

Output 17,4

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	BACDEFGHIJKLMNOPQRSTUVWXYZ
Output	0,1,17

Encoding RDARCAAAABB

Stack	DRABCEFGHIJKLMNOPQSTUVWXYZ
Output	17,4,2

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack RBACDEFGHIJKLMNOPQSTUVWXYZ
Output 0,1,17,2

Encoding RDARCAAAABB

Stack ADRBCEFGHIJKLMNOPQSTUVWXYZ
Output 17,4,2,2

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding `ABRACADABRA`

Stack	`ARBCDEFGHIJKLMNOPQSTUVWXYZ`
Output	0,1,17,2,3

Encoding `RDARCAAAABB`

Stack	`RADBCEFGHIJKLMNOPQSTUVWXYZ`
Output	17,4,2,2,4

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	CARBDEFGHIJKLMNOPQSTUVWXYZ
Output	0,1,17,2,3,1

Encoding RDARCAAAABB

Stack	CRADBEFGHIJKLMNOPQSTUVWXYZ
Output	17,4,2,2,4,2

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	ACRBDEFGHIJKLMNOPQSTUVWXYZ
Output	0,1,17,2,3,1,4

Encoding RDARCAAAABB

Stack	ACRDBEFGHIJKLMNOPQSTUVWXYZ
Output	17,4,2,2,4,2,0

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	DACRBEFGHIJKLMNOPQSTUVWXYZ
Output	0,1,17,2,3,1,4,1

Encoding RDARCAAAABB

Stack	ACRDBEFGHIJKLMNOPQSTUVWXYZ
Output	17,4,2,2,4,2,0,0

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	ADCRBEFGHIJKLMNOPQSTUVWXYZ
Output	0,1,17,2,3,1,4,1,5

Encoding RDARCAAAABB

Stack	ACRDBEFGHIJKLMNOPQSTUVWXYZ
Output	17,4,2,2,4,2,0,0,0

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	BADCREFGHIJKLMNOPQSTUVWXYZ
Output	0,1,17,2,3,1,4,1,5,4

Encoding RDARCAAAABB

Stack	ACRDBEFGHIJKLMNOPQSTUVWXYZ
Output	17,4,2,2,4,2,0,0,0,4

Move-To-Front coding (MTF) – *a.k.a.* Ryabko's book stack

Encoding ABRACADABRA

Stack	RBADCEFGHIJKLMNOPQSTUVWXYZ
Output	0,1,17,2,3,1,4,1,5,4,2

Encoding RDARCAAAABB

Stack	BACRDEFGHIJKLMNOPQSTUVWXYZ
Output	17,4,2,2,4,2,0,0,0,4,0

LZ77 vs. LZ78

LZ77 deficiencies leading to LZ78[Salomon]

1. *Limitation of look-ahead buffer size*.
   Does not hold for `DEFLATE`.

LZ77 vs. LZ78

LZ77 deficiencies leading to LZ78[Salomon]

1. *Limitation of look-ahead buffer size.*
   Does not hold for `DEFLATE`.
2. *Patterns in the input data should occur close together.*
   - How *universal* is that ?

LZ77 vs. LZ78

LZ77 deficiencies leading to LZ78[Salomon]

1. *Limitation of look-ahead buffer size.*
   Does not hold for `DEFLATE`.
2. *Patterns in the input data should occur close together.*
   - How *universal* is that ?
   - True for `DEFLATE` (well, up to 32KiB), very much less for `xz`/LZMA.
   - In *practical* implementations of LZ78, memory is limited too (dictionary freeze, reset or update) – but it's used more wisely.

Shannon

Shannon

Information is a measure of innovation in the data flow coming in.

Shannon's fat vs. Kolmogorov's memories

Shannon

Information is a measure of innovation in the data flow coming in.

Kolmogorov

Information is a measure of complexity in the data flow seen so far.

Deep trends

Zurek's physical entropy

The sum of Shannon and Kolmogorov informations is (almost) constant.

An easy (awkward ?) way : self-extracting archives (LZMA).

Deep trends

Zurek's physical entropy

The sum of Shannon and Kolmogorov informations is (almost) constant.
An easy (awkward ?) way : self-extracting archives (LZMA).

Bennett

Information is a measure of time complexity when executing Kolmogorov's "code".

- In short : Space (Kolmogorov) complexity shouldn't ignore time complexity so badly.
- Link with Martin-Löf's definition of "reasonable" means of testing for randomness ?

Space complexities

Choose the one you need !

- Kolmogorov
  - Length of the smallest computer program that will generate the input.
- Lempel-Ziv
  - Size of LZ78 dictionary, approximates the above.

Space complexities

Choose the one you need !

- Kolmogorov
  - Length of the smallest computer program that will generate the input.
- Lempel-Ziv
  - Size of LZ78 dictionary, approximates the above.
- Subword
  - Number of distinct subwords of length n
- Palindromic
  - Number of distinct palindroms of length n
- Abelian
  - Subword, up to permutations

The Figure in the Carpet

What information ?

- *Fundamentally* dependent on the *goal* !
  - Let $S_1 = A_1 \| D_1$ and $S_2 = A_2 \| D_2$ with data D_x ($|D_1| \approx |D_2|$), "decompressors" A_x ($|A_1| > |A_2|$) with resp. time complexities $O(n)$ vs. $O(n \log n)$.
  - From Bennett's point of view, which is best ?

The Figure in the Carpet

What information ?

- *Fundamentally* dependent on the *goal* !
  - Let $S_1 = A_1 \| D_1$ and $S_2 = A_2 \| D_2$ with data D_x ($|D_1| \approx |D_2|$), "decompressors" A_x ($|A_1| > |A_2|$) with resp. time complexities $O(n)$ vs. $O(n \log n)$.
  - From Bennett's point of view, which is best ?
- *– I shall study DNA code with Shannon's information !*

The Figure in the Carpet

What information ?

- *Fundamentally* dependent on the *goal* !
  - Let $S_1 = A_1 \| D_1$ and $S_2 = A_2 \| D_2$ with data D_x ($|D_1| \approx |D_2|$), "decompressors" A_x ($|A_1| > |A_2|$) with resp. time complexities $O(n)$ vs. $O(n \log n)$.
  - From Bennett's point of view, which is best ?
- *– I shall study DNA code with Shannon's information !*
  DNA machinery & code target *survival* in a given environment.
  (maybe on a slightly deeper ground than *perfect transmission* ;-))

The Figure in the Carpet

What information ?

- *Fundamentally* dependent on the *goal* !
  - Let $S_1 = A_1 || D_1$ and $S_2 = A_2 || D_2$ with data D_x ($|D_1| \approx |D_2|$), "decompressors" A_x ($|A_1| > |A_2|$) with resp. time complexities $O(n)$ vs. $O(n \log n)$.
  - From Bennett's point of view, which is best ?
- *– I shall study DNA code with Shannon's information !*
  DNA machinery & code target *survival* in a given environment.
  (maybe on a slightly deeper ground than *perfect transmission* ;-))
- As for transmission...
  - skimming the entropy fat off back references proved efficient ;
  - automata for transforming bits and copying bytes.