DEFLATE & gzip

Artur Jamro

- DEFLATE

Algorytm kompresji danych bazujący na LZ77 oraz Kodach Huffmana.

- DEFLATE

Algorytm kompresji danych bazujący na LZ77 oraz Kodach Huffmana.

- gzip (*.gz)

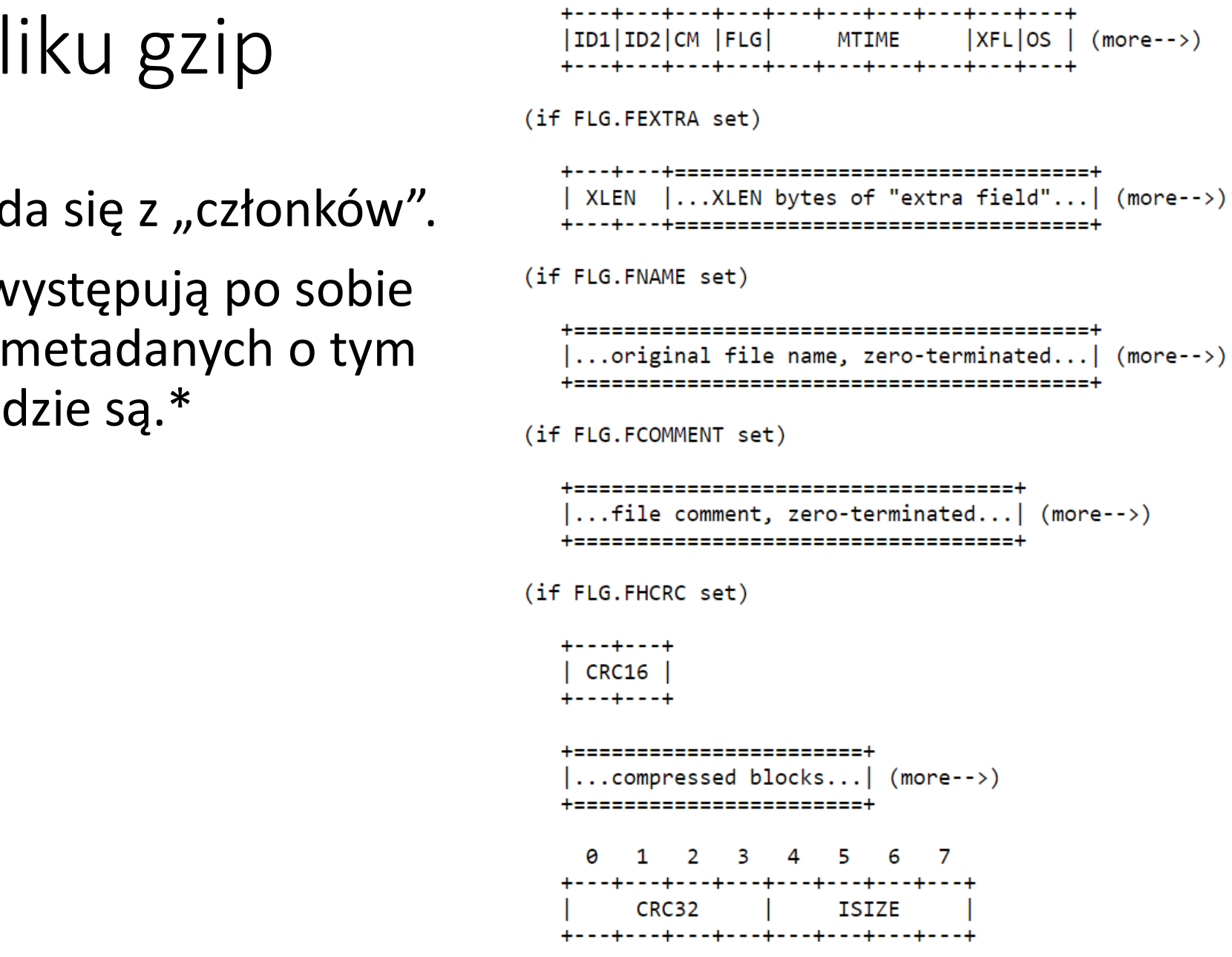
Format pliku przechowujący plik spakowany algorytmem DEFLATE.

Format pliku gzip

- Plik gzip składa się z „członków”.
- Członkowie występują po sobie bez żadnych metadanych o tym ile ich jest i gdzie są.*

2.3. Member format

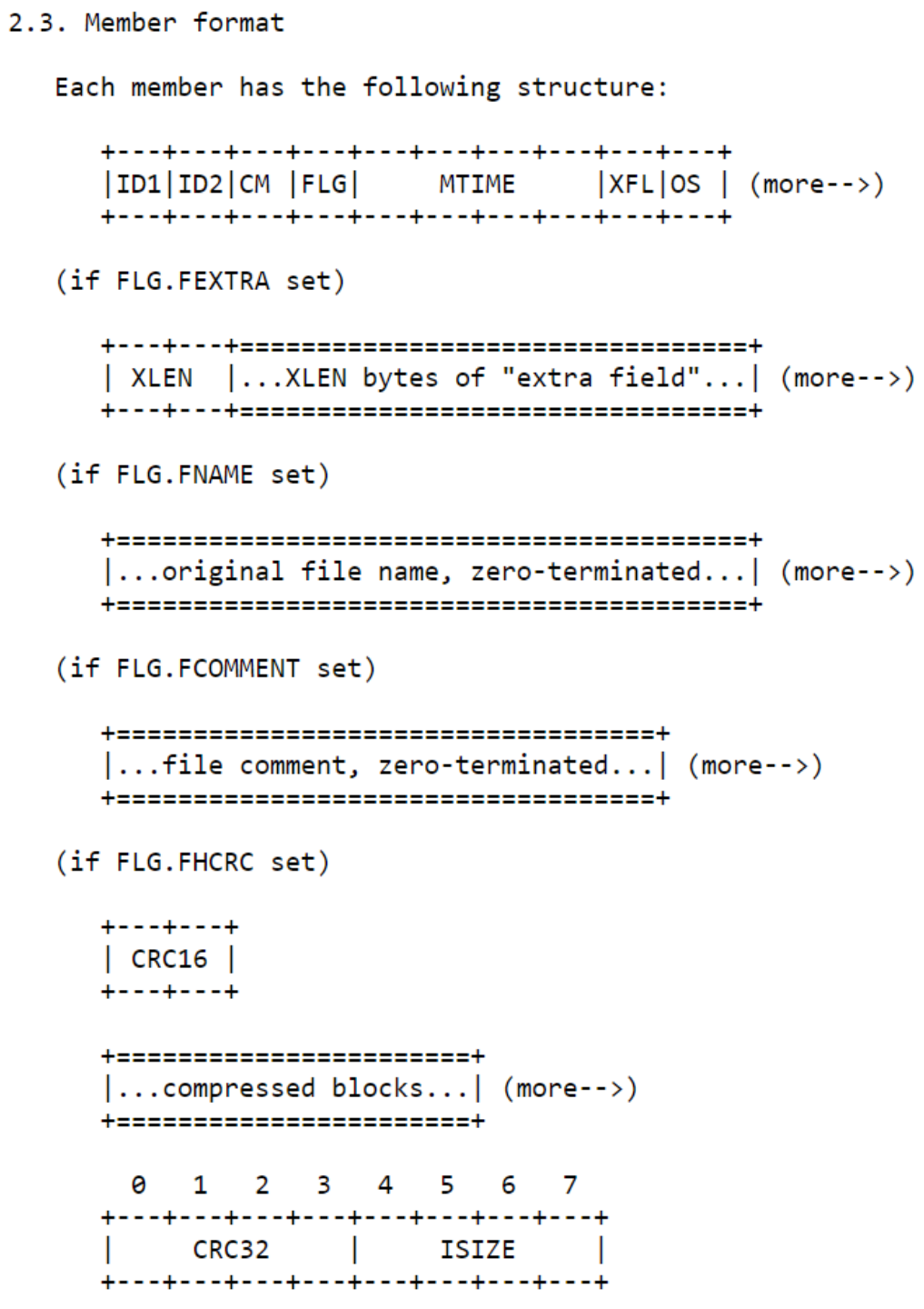
Each member has the following structure:



Format pliku gzip

- Plik gzip składa się z „członków”.
- Członkowie występują po sobie bez żadnych metadanych o tym ile ich jest i gdzie są.*

** Programy 7-zip oraz gzip wypakowują pojedynczy plik o zawartości powstałej przez połączenie wszystkich członków wewnątrz, ponadto 7-zip zachowuje pierwszą spotkaną nazwę pliku, a gzip sugeruje się nazwą pliku archiwum*



```
2.3. Member format

   Each member has the following structure:

      +---+---+---+---+---+---+---+---+---+---+
      |ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
      +---+---+---+---+---+---+---+---+---+---+

   (if FLG.FEXTRA set)

      +---+---+=================================+
      | XLEN  |...XLEN bytes of "extra field"...| (more-->)
      +---+---+=================================+

   (if FLG.FNAME set)

      +=========================================+
      |...original file name, zero-terminated...| (more-->)
      +=========================================+

   (if FLG.FCOMMENT set)

      +===================================+
      |...file comment, zero-terminated...| (more-->)
      +===================================+

   (if FLG.FHCRC set)

      +---+---+
      | CRC16 |
      +---+---+

      +=======================+
      |...compressed blocks...| (more-->)
      +=======================+

        0   1   2   3   4   5   6   7
      +---+---+---+---+---+---+---+---+
      |     CRC32     |     ISIZE     |
      +---+---+---+---+---+---+---+---+
```

Przykład

$ cat file.txt

abcabcxxxxxbcabcxxx

Przykład

$ cat file.txt

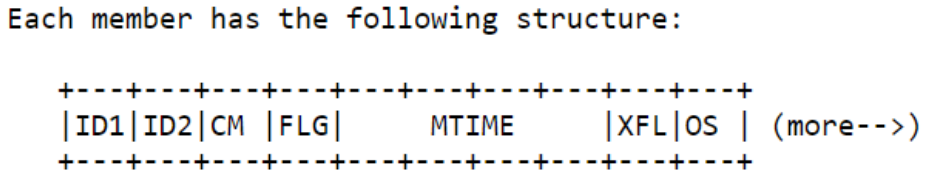
abcabcxxxxxbcabcxxx

$ gzip -9 file.txt

Przykład

```
$ cat file.txt
abcabcxxxxxbcabcxxx
$ gzip -9 file.txt
$ xxd file.txt.gz
00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74  ....!dOZ..file.t
00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00  xt.KLJNLJ.......
00000020: cc00 1e9d 1400 0000                      ........
```

Przykład



$ xxd file.txt.gz

00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ..!dOZ..file.t

00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.....

00000020: cc00 1e9d 1400 0000 ....

Magiczna liczba (identyfikator formatu pliku)

Przykład

Each member has the following structure:

```
+---+---+---+---+---+---+---+---+---+---+
|ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
+---+---+---+---+---+---+---+---+---+---+
```

$ xxd file.txt.gz

00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.......

00000020: cc00 1e9d 1400 0000   ....

Metoda kompresji:

- 0-7 – zarezerwowane
- 8 – DEFLATE

Przykład

Each member has the following structure:

```
+---+---+---+---+---+---+---+---+---+---+
|ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
+---+---+---+---+---+---+---+---+---+---+
```

$ xxd file.txt.gz

00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

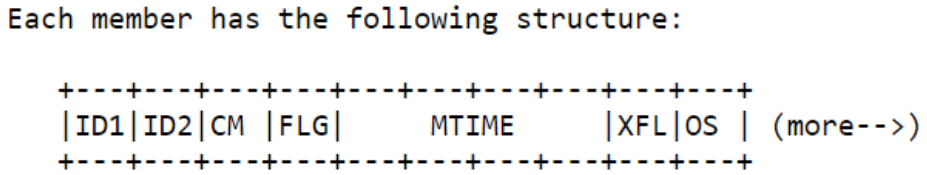
00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.......

00000020: cc00 1e9d 1400 0000   ....

Flagi:

Bit	0 – FTEXT	1 – FHCRC	2 – FEXTRA	3 – FNAME	4 – FCOMMENT	5-7 – reserved
Znaczenie	Kodowany zapewne jako plik ASCII	Obecne CRC16 dla nagłówka gzip	Obecne dodatkowe pola	Obecna nazwa kompresowanego pliku	Obecny komentarz	

Przykład



$ xxd file.txt.gz

00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.......

00000020: cc00 1e9d 1400 0000   ....

Czas modyfikacji pliku

Tutaj: 2018-01-05 12:32:51 GMT

Przykład

Each member has the following structure:

```
+---+---+---+---+---+---+---+---+---+---+
|ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
+---+---+---+---+---+---+---+---+---+---+
```

$ xxd file.txt.gz

00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.......

00000020: cc00 1e9d 1400 0000   ....

Dodatkowe flagi:

- 2 – użyto najwolniejszy i najlepiej kompresujący algorytm
- 4 – użyto najszybszy algorytm

Przykład

Each member has the following structure:

```
+---+---+---+---+---+---+---+---+---+---+
|ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
+---+---+---+---+---+---+---+---+---+---+
```

$ xxd file.txt.gz

00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

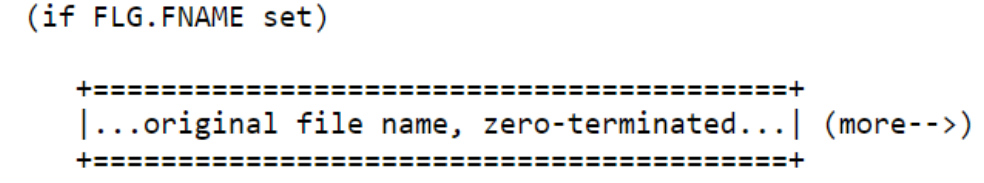
00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.......

00000020: cc00 1e9d 1400 0000   ....

System plików na którym wykonano kompresję *(przydatne przy określaniu znaków końca linii)*.

Tutaj: Unix.

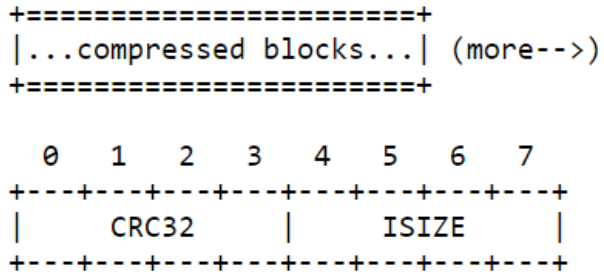
Przykład




```
$ xxd file.txt.gz
00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74  ....!dOZ..file.t
00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00  xt.KLJNLJ.......
00000020: cc00 1e9d 1400 0000                      ........
```

Nazwa pliku zakończona bajtem zerowym.

Przykład



$ xxd file.txt.gz

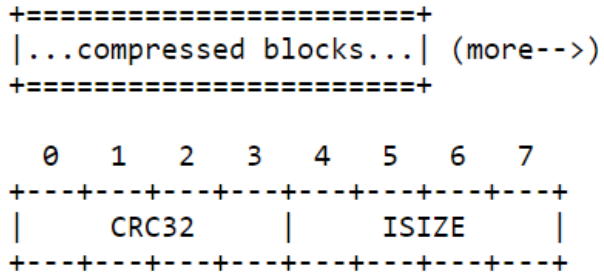
00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.....

00000020: cc00 1e9d 1400 0000 ....

Skompresowane dane. (o tym później)

Przykład



$ xxd file.txt.gz

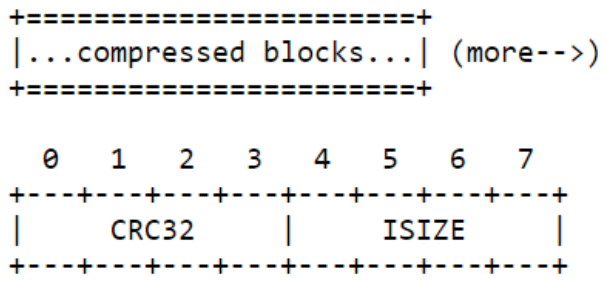
00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.......

00000020: cc00 1e9d 1400 0000 ....

CRC32 danych przed skompresowaniem.

Przykład



$ xxd file.txt.gz

00000000: 1f8b 0808 2164 4f5a 0203 6669 6c65 2e74 ....!dOZ..file.t

00000010: 7874 004b 4c4a 4e4c 4aae 0001 188b 0b00 xt.KLJNLJ.......

00000020: cc00 1e9d 1400 0000 ....

Rozmiar danych przed skompresowaniem (modulo 2^32).

Tutaj 20 bajtów.

Zanim przyjrzymy się skompresowanym danym…

Użycie kodów Huffmana w algorytmie DEFLATE

Dwie dodatkowe zasady:

- Wszystkie kody tej samej długości mają leksykograficznie kolejne wartości, w tej samej kolejności co symbole, które reprezentują.
- Krótsze kody występują leksykograficznie przed dłuższymi.

Użycie kodów Huffmana w algorytmie DEFLATE

Mając długość bitową wszystkich kodów odpowiadającym symbolom, można wyznaczyć te kody w następujący sposób:

Użycie kodów Huffmana w algorytmie DEFLATE

1. Policz bl_count[i] – liczba kodów o długości bitowej i

Użycie kodów Huffmana w algorytmie DEFLATE

1. Policz bl_count[i] – liczba kodów o długości bitowej i

Dla alfabetu ABCDEFGH i długości bitowych (3, 3, 3, 3, 3, 2, 4, 4):

i	bl_count[i]
2	1
3	5
4	2

Użycie kodów Huffmana w algorytmie DEFLATE

2. Dla każdej długości bitowej, znajdź najmniejszy kod bitowy:

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
    code = (code + bl_count[bits-1]) << 1;
    next_code[bits] = code;
}
```

Użycie kodów Huffmana w algorytmie DEFLATE

2. Dla każdej długości bitowej, znajdź najmniejszy kod bitowy:

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
    code = (code + bl_count[bits-1]) << 1;
    next_code[bits] = code;
}
```

i	bl_count[i]
2	1
3	5
4	2

i	next_code[i]
1	0
2	0
3	2
4	14

Użycie kodów Huffmana w algorytmie DEFLATE

3. Przypisz numeryczne wartości wszystkim kodom w drzewie:

```
for (n = 0;  n <= max_code; n++) {
    len = tree[n].Len;
    if (len != 0) {
        tree[n].Code = next_code[len];
        next_code[len]++;
    }
```

Długość bitowa symbolu

Pomiń puste kody

Przypisz kod danemu symbolowi

Użycie kodów Huffmana w algorytmie DEFLATE

3. Przypisz numeryczne wartości wszystkim kodom w drzewie:

```
for (n = 0;  n <= max_code; n++) {
    len = tree[n].Len;
    if (len != 0) {
        tree[n].Code = next_code[len];
        next_code[len]++;
    }
```

Długość bitowa symbolu

Pomiń puste kody

Przypisz kod danemu symbolowi

i	next_code[i]
1	0
2	0
3	2
4	14

Symbol	Długość	Kod
A	3	010
B	3	011
C	3	100
D	3	101
E	3	110
F	2	00
G	4	1110
H	4	1111

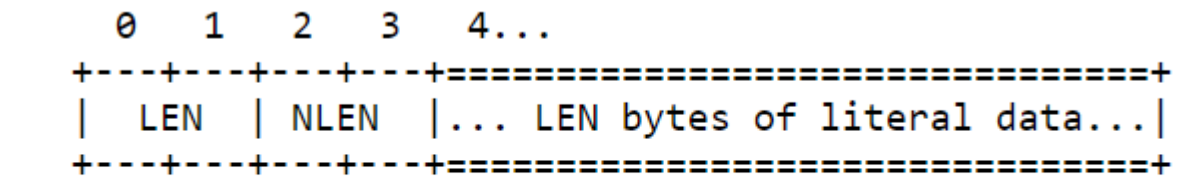
Format bloku DEFLATE’a

Nagłówek:

- 1 bit – BFINAL – czy to ostatni blok
- 2 bity – BTYPE – typ bloku:
  - 00 – bez kompresji
  - 01 – użyto statycznych kodów Huffmana
  - 10 – użyto dynamicznych kodów Huffmana
  - 11 – zarezerwowane (błąd)

Format bloków DEFLATE

- Blok bez kompresji (BTYPE=00):



LEN is the number of data bytes in the block. NLEN is the one's complement of LEN.

Skompresowane bloki

3 alfabety:

- Literały (0..255)
- Długości (3..258)
- Dystans (1..32,768)

Skompresowane bloki

3 alfabety:

- Literały (0..255)
- Długości (3..258)
- Dystanse (1..32,768)

Gdzie alfabety literałów i długości są połączone w jeden:

- 0..255 – literał
- 256 – koniec bloku
- 257..285 – długość

Code	Extra Bits	Length(s)	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

The extra bits should be interpreted as a machine integer stored with the most-significant bit first, e.g., bits 1110 represent the value 14.

Skompresowane bloki

Alfabet dystansu także ma tabelkę z extra bitami:

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Skompresowane bloki

Statyczne kody Huffmana (BTYPE=01):

- Nie przechowywane jawnie w danych

Skompresowane bloki

Statyczne kody Huffmana (BTYPE=01):

- Nie przechowywane jawnie w danych
- Alfabet literałów/długości: *(wystarczy znać długości bitowe)*

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Symbole 286-287 nie występują w skompresowanych danych.

Skompresowane bloki

Statyczne kody Huffmana (BTYPE=01):

- Alfabet dystansów:

Kody 0-31 reprezentowane przez 5 bitów z możliwością extra bitów (patrz tabelka wcześniej).

Kody 30-31 w praktyce nie występują w danych.

Skompresowane bloki

Dynamiczne kody Huffmana (BTYPE=10):

```
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)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    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)
```

```
We can now define the format of the block:

      5 Bits: HLIT, # of Literal/Length codes - 257 (257 - 286)
      5 Bits: HDIST, # of Distance codes - 1        (1 - 32)
      4 Bits: HCLEN, # of Code Length codes - 4     (4 - 19)

      (HCLEN + 4) x 3 bits: code lengths for the code length
         alphabet given just above, in the order: 16, 17, 18,
         0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

         These code lengths are interpreted as 3-bit integers
         (0-7); as above, a code length of 0 means the
         corresponding symbol (literal/length or distance code
         length) is not used.

      HLIT + 257 code lengths for the literal/length alphabet,
         encoded using the code length Huffman code

      HDIST + 1 code lengths for the distance alphabet,
         encoded using the code length Huffman code

      The actual compressed data of the block,
         encoded using the literal/length and distance Huffman
         codes

      The literal/length symbol 256 (end of data),
         encoded using the literal/length Huffman code
```

Skompresowane bloki

Dynamiczne kody Huffmana (BTYPE=10):

```
We can now define the format of the block:

      5 Bits: HLIT, # of Literal/Length codes - 257 (257 - 286)
      5 Bits: HDIST, # of Distance codes - 1        (1 - 32)
      4 Bits: HCLEN, # of Code Length codes - 4     (4 - 19)

      (HCLEN + 4) x 3 bits: code lengths for the code length
         alphabet given just above, in the order: 16, 17, 18,
         0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

         These code lengths are interpreted as 3-bit integers
         (0-7); as above, a code length of 0 means the
         corresponding symbol (literal/length or distance code
         length) is not used.
```

```
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)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    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)
```

```
HLIT + 257 code lengths for the literal/length alphabet,
   encoded using the code length Huffman code

HDIST + 1 code lengths for the distance alphabet,
   encoded using the code length Huffman code

The actual compressed data of the block,
   encoded using the literal/length and distance Huffman
   codes

The literal/length symbol 256 (end of data),
   encoded using the literal/length Huffman code
```

Skompresowane bloki

Dynamiczne kody Huffmana (BTYPE=10):

We can now define the format of the block:

```
5 Bits: HLIT, # of Literal/Length codes - 257 (257 - 286)
5 Bits: HDIST, # of Distance codes - 1        (1 - 32)
4 Bits: HCLEN, # of Code Length codes - 4     (4 - 19)

(HCLEN + 4) x 3 bits: code lengths for the code length
   alphabet given just above, in the order: 16, 17, 18,
   0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

   These code lengths are interpreted as 3-bit integers
   (0-7); as above, a code length of 0 means the
   corresponding symbol (literal/length or distance code
   length) is not used.
```

```
HLIT + 257 code lengths for the literal/length alphabet,
   encoded using the code length Huffman code

HDIST + 1 code lengths for the distance alphabet,
   encoded using the code length Huffman code

The actual compressed data of the block,
   encoded using the literal/length and distance Huffman
   codes

The literal/length symbol 256 (end of data),
   encoded using the literal/length Huffman code
```

```
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)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    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)
```

Algorytm dekompresji DEFLATE’a

Dla każdego bloku:
- przeczytaj nagłówek bloku
- jeśli blok przechowany bez kompresji:
  - odczytaj LEN, NLEN z wejścia
  - skopiuj LEN bajtów z wejścia na wyjście
- w przeciwnym przypadku:
  - jeśli skompresowany z użyciem dynamicznych kodów Huffmana:
    - odczytaj reprezentację drzewa
  - pętla [1]:
    - odczytaj wartość „literal/length” z wejścia
    - jeśli wartość < 256:
      - wypisz wartość na wyjście
    - wpp:
      - jeśli wartość = koniec bloku (256):
        - wyskocz z pętli [1]
      - wpp (wartość = 257..285):
        - zdekoduj odległość z wejścia
        - przesuń się o odległość w wyjściu do tyłu i skopiuj „length” bajtów na wyjście

Algorytm dekompresji DEFLATE'a

```
Dla każdego bloku:
        przeczytaj nagłówek bloku
        jeśli blok przechowany bez kompresji:
                odczytaj LEN, NLEN z wejścia
                skopiuj LEN bajtów z wejścia na wyjście
        w przeciwnym przypadku:
                jeśli skompresowany z użyciem dynamicznych kodów Huffmana:
                        odczytaj reprezentację drzewa
                pętla [1]:
                        odczytaj wartość „literal/length" z wejścia
                        jeśli wartość < 256:
                                wypisz wartość na wyjście
                        wpp:
                                jeśli wartość = koniec bloku (256):
                                        wyskocz z pętli [1]
                                wpp (wartość = 257..285):
                                        zdekoduj odległość z wejścia
                                        przesuń się o odległość w wyjściu do tyłu i skopiuj „length" bajtów na wyjście
```

Różnica między wykorzystaniem dynamicznych a statycznych kodów Huffmana

Przypomina LZ77, ale dodatkowo korzysta z kodów Huffmana

Przykład

Przyjrzyjmy się teraz skompresowanym danym:

```
$ xxd -b -s 0x13 -l 13 file.txt.gz
00000013: 01001011 01001100 01001010 01001110 01001100 01001010  KLJNLJ
00000019: 10101110 00000000 00000001 00011000 10001011 00001011  ......
0000001f: 00000000                                               .
```

Przykład

```
$ xxd -b -s 0x13 -l 13 file.txt.gz
00000013: 01001011 01001100 01001010 01001110 01001100 01001010  KLJNLJ
00000019: 10101110 00000000 00000001 00011000 10001011 00001011  ......
0000001f: 00000000                                               .
```

BFINAL – czy ostatni blok z danymi

U nas – tak.

Przykład

```
$ xxd -b -s 0x13 -l 13 file.txt.gz
00000013: 01001011 01001100 01001010 01001110 01001100 01001010  KLJNLJ
00000019: 10101110 00000000 00000001 00011000 10001011 00001011  ......
0000001f: 00000000                                               .
```

BTYPE – określa typ kompresji danych:

- 00 – bez kompresji
- 01 – użyto statycznych kodów Huffmana
- 10 – użyto dynamicznych kodów Huffmana
- 11 – zarezerwowane (błąd)

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ......

0000001f: 00000000   .

10010001 ----> literał ‘a’ (len codes)

Zdekodowano: a

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ......

0000001f: 00000000   .

10010010 ----> literał ‘b’ (len codes)

Zdekodowano: ab

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ......

0000001f: 00000000   .

10010011 ----> literał ‘c’ (len codes)

Zdekodowano: abc

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ......

0000001f: 00000000   .

10010001 ----> literał ‘a’ (len codes)

Zdekodowano: abca

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ......

0000001f: 00000000   .

10010010 ----> literał ‘b’ (len codes)

Zdekodowano: abcab

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ……

0000001f: 00000000   .

10010011 ----> literał ‘c’ (len codes)

Zdekodowano: abcabc

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ……

0000001f: 00000000   .

10101000 ----> literał ‘x’ (len codes)

Zdekodowano: abcabcx

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ……

0000001f: 00000000   .

0000010 ----> długość 4 (len codes)

Zdekodowano: abcabcx****

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ......

0000001f: 00000000   .

0000010 ----> długość 4 (len codes)

00000 ----> dystans 1 (dist codes)

Zdekodowano: abcabcxxxxx

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ……

0000001f: 00000000   .

0000110 ----> długość 8 (len codes)

Zdekodowano: abcabcxxxxx********

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ……

0000001f: 00000000   .

0000110 ----> długość 8 (len codes)

00110 + 01 ----> dystans 9+1 (dist codes)

Zdekodowano: abcabcxxxxxbcabcxxx

Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ……

0000001f: 00000000   .

00111010 ----> literał ‘\n’ (len codes)

Zdekodowano: abcabcxxxxxbcabcxxx\n

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

$ xxd -b -s 0x13 -l 13 file.txt.gz

00000013: 01001011 01001100 01001010 01001110 01001100 01001010 KLJNLJ

00000019: 10101110 00000000 00000001 00011000 10001011 00001011 ……

0000001f: 00000000 .

0000000 ----> koniec bloku (len codes)

Zdekodowano: abcabcxxxxxbcabcxxx\n

Lit Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

Przykład

```
$ xxd -b -s 0x13 -l 13 file.txt.gz
00000013: 01001011 01001100 01001010 01001110 01001100 01001010  KLJNLJ
00000019: 10101110 00000000 00000001 00011000 10001011 00001011  ......
0000001f: 00000000                                               .
```

Koniec dekompresji.

Zdekodowano: abcabcxxxxxbcabcxxx\n

Algorytm kompresji

- Nietrywialny problem
- Różne rozwiązania: od szybkich do lepszej kompresji, ale wolniejszych

```
local const config configuration_table[10] = {
/* good lazy nice chain */
/* 0 */ {0, 0, 0, 0, deflate_stored}, /* store only */
/* 1 */ {4, 4, 8, 4, deflate_fast}, /* max speed, no lazy matches */
/* 2 */ {4, 5, 16, 8, deflate_fast},
/* 3 */ {4, 6, 32, 32, deflate_fast},
/* 4 */ {4, 4, 16, 16, deflate_slow}, /* lazy matches */
/* 5 */ {8, 16, 32, 32, deflate_slow},
/* 6 */ {8, 16, 128, 128, deflate_slow},
/* 7 */ {8, 32, 128, 256, deflate_slow},
/* 8 */ {32, 128, 258, 1024, deflate_slow},
/* 9 */ {32, 258, 258, 4096, deflate_slow}}; /* max compression */
```

Algorytm kompresji

- Koniec bloku, gdy:
  - kompresor stwierdzi, że warto wygenerować nowe drzewa lub
  - gdy kompresorowi skończy się miejsce w buforze

Algorytm kompresji

- Koniec bloku, gdy:
  - kompresor stwierdzi, że warto wygenerować nowe drzewa lub
  - gdy kompresorowi skończy się miejsce w buforze
- Znajdowanie powtórzeń:
  - hash chain po 3 bajtach, przechodzimy się wybierając najdłuższe dopasowanie
  - preferujemy bliskie odległości, by wykorzystać sposób kodowania odległości (mniej extra bits)
  - parametr czasu wykonania określający jak daleko w hash chainie sięgamy w tył

Algorytm kompresji

- Lazy matching:
  - Jeśli znaleziono dopasowanie, sprawdź czy nie byłoby lepszego, gdybyśmy szukali powtórzenia bajt dalej
  - Kontrolowany parametrem czasu wykonania
  - W normalnym przypadku, gdy dopasowanie jest „wystarczająco długie”, kompresor nie szuka dłuższych wystąpień
  - Gdy liczy się szybkość, kompresor dodaje nowe elementy do hash chaina gdy:
    - Nie znaleziono dopasowania
    - Dopasowania nie są „zbyt długie”

Bibliografia

- RFC 1951 - DEFLATE Compressed Data Format Specification version 1.3, Peter Deutsch [https://tools.ietf.org/html/rfc1951]
- RFC 1952 - GZIP file format specification version 4.3, Peter Deutsch [https://tools.ietf.org/html/rfc1952]
- Opis algorytmów DEFLATE i INFLATE, Jean-loup Gailly & Mark Adler [http://www.gzip.org/algorithm.txt]
- Kod źródłowy biblioteki zlib, Jean-loup Gailly & Mark Adler [https://github.com/madler/zlib/]

Pytania?

PS to jeszcze nie koniec

Extra: FLIF

FLIF [http://flif.info]

- Free Lossless Image Format

FLIF [http://flif.info]

- Free Lossless Image Format
- Dlaczego?
  - Teraz - formatów z różnym przeznaczeniem (JPG do zdjęć, PNG do do szkicy technicznych, itp.)

FLIF [http://flif.info]

- Free Lossless Image Format
- Dlaczego?
  - Teraz - formatów z różnym przeznaczeniem (JPG do zdjęć, PNG do do szkicy technicznych, itp.)
- Co oferuje?
  - Kompresję zarówno stratną i bezstratną
  - Lepszą kompresję od wszystkich będących w użyciu formatów
  - Progresywne dekodowanie

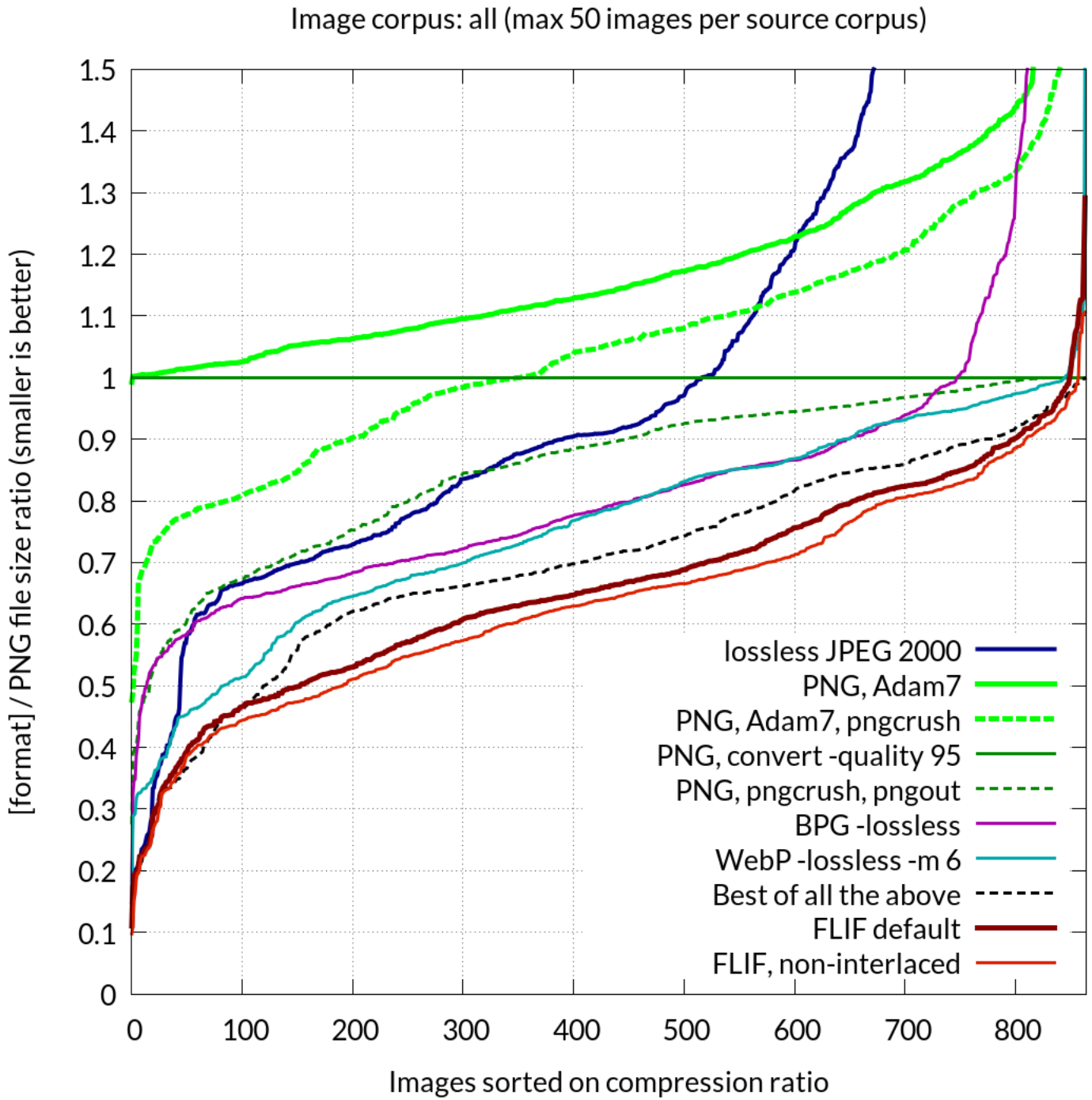
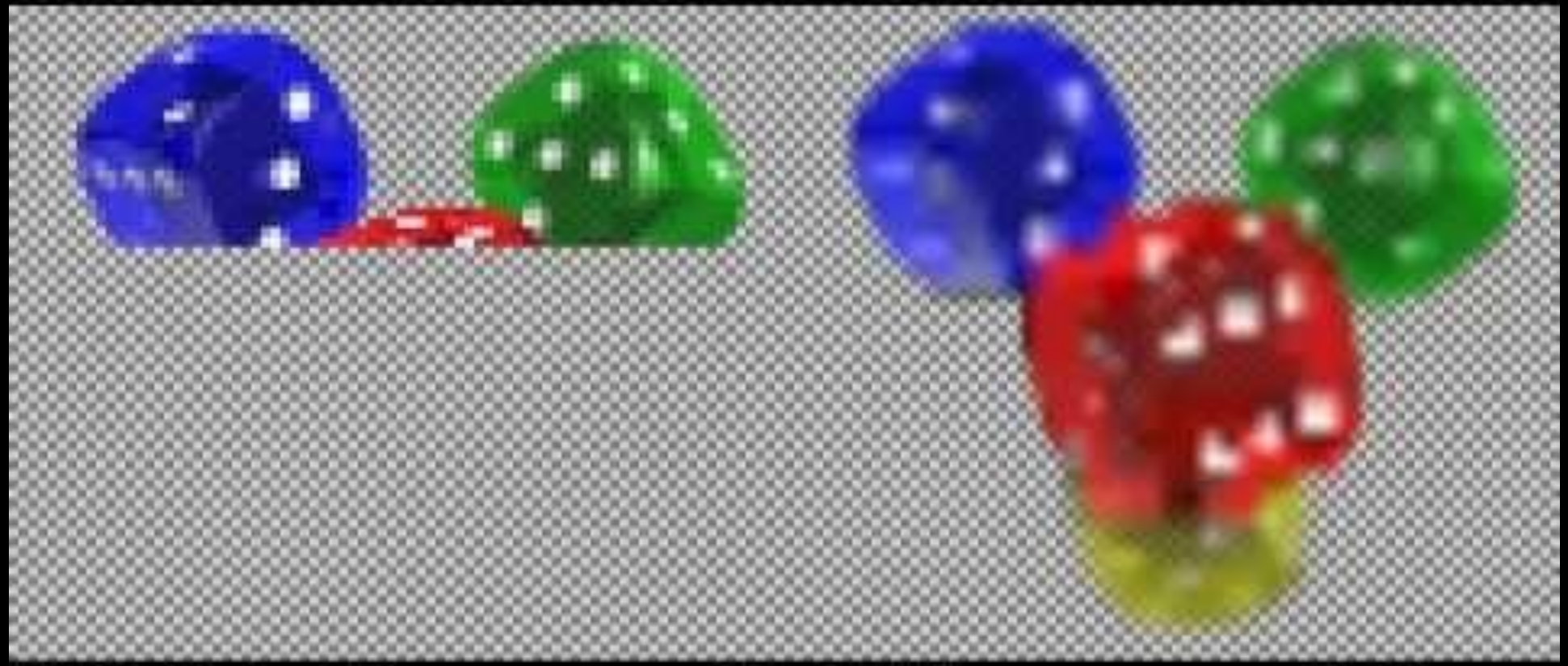


Image corpus: all (max 50 images per source corpus)
[format] / PNG file size ratio (smaller is better)
Images sorted on compression ratio
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
100
200
300
400
500
600
700
800
lossless JPEG 2000
PNG, Adam7
PNG, Adam7, pngcrush
PNG, convert -quality 95
PNG, pngcrush, pngout
BPG -lossless
WebP -lossless -m 6
Best of all the above
FLIF default
FLIF, non-interlaced



PNG (Adam7)
289924 bytes
(1.37%)

partial file
4000 bytes

FLIF
158184 bytes
(2.52%)

FLIF [http://flif.info]

- Jak?
  - FLIF is based on MANIAC compression. MANIAC (Meta-Adaptive Near-zero Integer Arithmetic Coding) is an algorithm for entropy coding developed by Jon Sneyers and Pieter Wuille. It is a variant of CABAC (context-adaptive binary arithmetic coding), where instead of using a multi-dimensional array of quantized local image information, the contexts are nodes of decision trees which are dynamically learned at encode time. This means a much more image-specific context model can be used, resulting in better compression.

FLIF [http://flif.info]

- Jak?
  - FLIF is based on MANIAC compression. MANIAC (Meta-Adaptive Near-zero Integer Arithmetic Coding) is an algorithm for entropy coding developed by Jon Sneyers and Pieter Wuille. It is a variant of CABAC (context-adaptive binary arithmetic coding), where instead of using a multi-dimensional array of quantized local image information, the contexts are nodes of decision trees which are dynamically learned at encode time. This means a much more image-specific context model can be used, resulting in better compression.
- Kiedy?
  - Jeszcze nie wiadomo. Format jest nadal rozwijany i niewiele programów aktualnie go obsługuje.

Dziękuję za uwagę.

A może więcej pytań?