

AULA 16

Compressão de dados



Fonte: [Best VPN for Data Compression](#)

Referências: Entrada e saída binárias (PF), Compressão de dados (PF), Data Compression (SW), slides (SW), video (SW)

Esquema

representar um dado **fluxo de bits** (*bitstream*) por outro mais curto.

Esquema básico de compressão de dados:

- ▶ um **compressor** transforma um fluxo de bits B em um fluxo $C(B)$ e
- ▶ um **expansor** transforma $C(B)$ de volta em B .



Basic model for data compression

Codificador e decodificador

Fluxo B é **original** e o fluxo $C(B)$ é **codificado**.

Fluxo produzido pelo **expansor** é **decodificado**.

Fluxo **decodificado** é **idêntico** ao **original**, a compressão não perde informação (*lossless compression*).

Notação: $|B|$ é o número de bits de B .

Taxa de compressão: $|C(B)| / |B|$.

Desafio: obter a **menor** taxa de compressão possível.

Considerações teóricas

Fato. Nenhum algoritmo pode garantir taxa de compressão estritamente menor que 1 para todo e qualquer fluxo de bits.

Fluxos de bits aleatórios são pouco compressíveis.

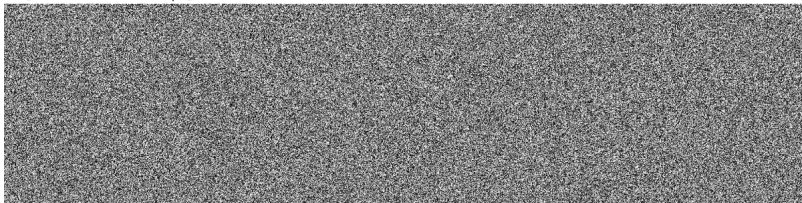
Mesmo fluxos pseudo-aleatórios podem ser difíceis de comprimir.

Considerações teóricas

Muitos fluxos de bits **parecem aleatórios** mas não são.

Exemplo: o fluxo de bits parece aleatório. . .

```
% java RandomBits | java PictureDump 2000 500
```



1000000 bits

A difficult file to compress: 1 million (pseudo-) random bits

Considerações teóricas

mas foi produzido pelo código

```
public class RandomBits {
    public static void main(String[] args){
        int x = 11111;
        for(int i = 0; i < 1000000; i++) {
            x = x * 314159 + 218281;
            BinaryStdOut.write(x>0);
        }
        BinaryStdOut.close();}
}
```

Taxa de compressão: 0.02

Considerações teóricas

Outro exemplo:

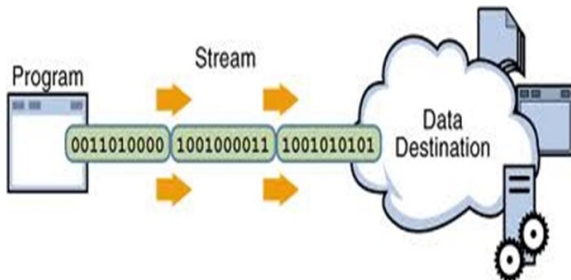
$$4 \left(\sum_{i=0}^{\infty} (-1)^i (1/(2i + 1)) \right)$$

é uma representação **muito comprimida** da expansão decimal do número π .

Teoria da compressão de dados tem ligações fascinantes com a **Teoria da Informação** e os conceitos de **Aleatoriedade** e **Entropia**.

Entrada e saídas binárias

Input / Output Streams



Fonte: [Input / Output Streams Byte streams](#)

Referências: [BinaryStdIn](#), [BinaryStdOut](#), [BinaryIn](#), [BinaryOut](#)

Cadeia de bits e fluxo de bits

Cadeia de bits (= *bitstring*) é uma sequência de bits:

```
110001000001110101010011111011001110001010000  
1100000000001111110000011101000111
```

fluxo de bits (= *bitstream*) é uma cadeia de bits na **entrada** ou na **saída** de um programa.

A classe **BinaryStdIn** lê um fluxo de bits a partir da **entrada padrão**.

A classe **BinaryStdOut** escreve um fluxo de bits na **saída padrão**.

BinaryStdIn

```
public class BinaryStdIn
```

<code>boolean</code>	<code>readBoolean()</code>	lê 1 bit e devolve o booleana correspondente
<code>char</code>	<code>readChar()</code>	lê 8 bits
<code>char</code>	<code>readChar(int r)</code>	lê <code>r</code> (entre 1 e 16) bits
<code>String</code>	<code>readString()</code>	lê fluxo em blocos de 8 bits
<code>int</code>	<code>readInt()</code>	lê 32 bits
<code>int</code>	<code>readInt(int r)</code>	lê <code>r</code> (entre 1 e 32) bits
<code>boolean</code>	<code>isEmpty()</code>	fluxo está vazio?
<code>void</code>	<code>close()</code>	fecha o fluxo

BinaryStdOut

<code>public class</code>	<code>BinaryStdOut</code>	
<code>void</code>	<code>write(boolean b)</code>	escreve o bit representado por <code>b</code>
<code>void</code>	<code>write(char c)</code>	escreve 8 bits da representação de <code>c</code>
<code>void</code>	<code>write(boolean c, int r)</code>	escreve os <code>r</code> (1 a 16) bits de <code>c</code> menos significativos
<code>void</code>	<code>close()</code>	fecha o fluxo

Para arquivos o `algs4` disponibiliza os irmãos `BinaryIn` e `BinaryOut`.

Exemplo

Representação de datas, como 15/05/2018.

```
StdOut.print(dia + "/" + mes + "/" + ano);
```

```
00110001 00110101 16 bits  
00101111          8 bits  
00110000 00110101 16 bits  
00101111          8 bits  
00110010 00110000 00110001 00111000 32 bits  
80 bits
```

representação: 10 chars = 10×8 bits

Exemplo... continuação

Representação de datas, como 15/05/2018.

```
BinaryStdOut.write(dia);
```

```
BinaryStdOut.write(mes);
```

```
BinaryStdOut.write(ano);
```

```
0000000000000000000000000000000001111 32 bits
```

```
000000000000000000000000000000000101 32 bits
```

```
00000000000000000000000011111100010 32 bits
```

96 bits

representação: 3 ints = 3×32 bits

Exemplo ... continuação

Representação de datas, como 15/05/2018.

```
BinaryStdOut.write((char) dia);  
BinaryStdOut.write((char) mes);  
BinaryStdOut.write((short) ano);
```

00001111 8 bits

00000101 8 bits

0000011111100010 16 bits

32 bits

representação: 2 **chars** + 1 short = $2 \times 8 + 16$ bits

Exemplo ...continuação

Representação de datas, como 15/05/2018.

```
BinaryStdOut.write(dia, 5);  
BinaryStdOut.write(mes, 4);  
BinaryStdOut.write(ano, 12);
```

01111 5 bits

0101 4 bits

011111100010 12 bits

000 3 bits

24 bits

representação: 5 + 4 + 12 bits + 3 bits de padding

BinaryDump: exemplo

```
% more abra.txt
```

```
ABRACADARRA!
```

```
% java BinaryDump 16 < abra.txt
```

```
0100000101000010
```

```
0101001001000001
```

```
0100001101000001
```

```
0100010001000001
```

```
0100001001010010
```

```
0100000100100001
```

```
96 bits
```

BinaryDump

Examina e converte uma cadeia de bits no caracteres 0 e 1.

```
public class BinaryDump{  
    public static void main(String[] args){  
        int width= Integer.parseInt(args[0]);  
        int cnt;
```

BinaryDump

```
for(cnt= 0;!BinaryStdIn.isEmpty();cnt++){
    if (width == 0) {
        BinaryStdIn.readBoolean();
        continue;
    }
    if (cnt != 0 && cnt % width == 0)
        StdOut.println();
    if (BinaryStdIn.readBoolean())
        StdOut.print("1");
    else StdOut.print("0");
}
if (width != 0) StdOut.println();
StdOut.println(cnt + "bits");
}
```

HexDump: exemplo

Divide a cadeia de bits em blocos de 4 bits e imprime cada bloco como um par de dígitos hexadecimais.

```
% more abra.txt
```

```
ABRACADARRA!
```

```
% java HexDump 4 < abra.txt
```

```
41 42 52 41
```

```
43 41 44 41
```

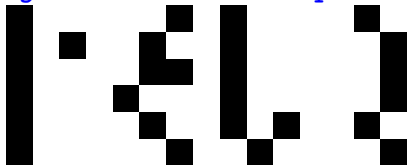
```
42 52 41 21
```

```
96 bits
```

PictureDump: exemplo

Exibe uma cadeia de bits como uma sequência de pequenos quadrados pretos e brancos dividida em **linhas** de **comprimento especificado**.

```
% java PictureDump 16 6 < abra.txt
```



Compressão de genomas



Fonte: [A New DNA Sequence Search - Compressive Genomics](#)

Genomas

Genomas são representados por cadeias **sobre o alfabeto** A, C, G e T.

```
GAATTGCTAGCAATTGTGTCCCCCTCACACTCCCAAATTCGCGGGG  
CCACACTCACCGGAGTGGCAAGCTAGCTTAAGTAACGCCACTTTGC  
GAAAGTTCAGATCAAGGTCAGGAACAAAGAAACAGCTGAATACCA
```

Podemos representar **cada caractere** por 1 byte = 8 bits.

Genomas

O genoma

ATAGATGCATAGCGCATAGCTAGATGTGCTAGC usaria

$33 \times 8 = 264$ bits

```
% more genomeTiny.txt
```

```
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

```
% java BinaryDump 56 < genomeTiny.txt
```

```
01000001010101000100000101000111010000010101010001000111
```

```
01000011010000010101010001000001010001110100001101000111
```

```
01000011010000010101010001000001010001110100001101010100
```

```
01000001010001110100000101010100010001110101010001000111
```

```
0100001101010100010000010100011101000011
```

264 bits

Genomas

Não precisamos usar 8 bits para representar os caracteres de um **alfabeto pequeno**.

Podemos usar só **2 bits** por caractere:

```
% more genomeTiny.txt
```

```
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

```
% java Genome - < genomeTiny.txt | java  
BinaryDump 32
```

```
00000000000000000000000000000000100001
```

```
00110010001110010011001001100100
```

```
11001001110010001110111001110010
```

```
01000000
```

```
104 bits
```

Genomas

Podemos usar só **2 bits** por caractere:

```
% more genomeTiny.txt
```

```
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

```
% java Genome - < genomeTiny.txt | java  
HexDump 8
```

```
00 00 00 21 23 2d 23 74
```

```
8d 8c bb 63 40
```

```
104 bits
```

Genomas

```
public static void compress() {  
    Alphabet DNA = new Alphabet("ACTG");  
    int w = DNA.lgR(); // w == 2  
    String s = BinaryStdIn.readString();  
    int n = s.length();  
    BinaryStdOut.write(n);  
    for(int i = 0; i < n; i++) {  
        int d = DNA.toIndex(s.charAt(i));  
        // escreve em 2 bits  
        BinaryStdOut.write(d, w);  
    }  
    BinaryStdOut.close();  
}
```

Genomas

```
public static void expand() {
    Alphabet DNA = new Alphabet("ACTG");
    int w = DNA.lgR(); // w == 2
    int n = BinaryStdIn.readInt();
    for (int i = 0; i < n; i++) {
        // leia 2 bits
        char c = BinaryStdIn.readChar(w);
        // escreva um char
        BinaryStdOut.write(DNA.toChar(c));;
    }
    BinaryStdOut.close();
}
```

Genomas

Não dá para ver o fluxo de bits diretamente na saída padrão:

```
% java Genome - < genomeTiny.txt  
??
```

Compressão seguida de **expansão** reproduz o **fluxo original**:

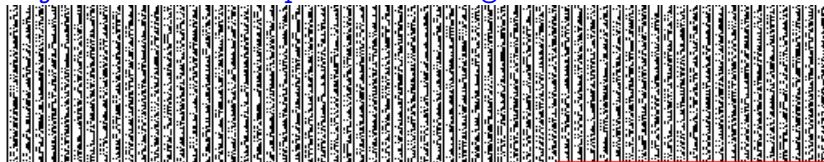
```
% java Genome- < genomeTiny.txt > genomeTiny.2bit  
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

```
% java Genome - < genomeTiny.txt | java Genome +  
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

Genomas

Virus de verdade, taxa de compressão de $12536/50000 = 0.25$:

```
% java PictureDump 512 100 < genomeVirus.txt
```



50000 bits

```
% java Genome - < genomeVirus.txt | java  
PictureDump 512 25
```



12536 bits

Codificação de comprimento de carreira

Lossless pixel compression



Fonte: [Lossy Image Compression with Run-Length Encoding](#)

Codificação de comprimento de carreira

Em inglês: *run-length encoding* (=RLE).

Exemplo: cadeia de bits abaixo tem uma carreira de 15 0s, uma carreira de 7 1s, uma de 7 0s, e uma de 11 1s:

0000000000000001111111000000011111111111

Pode ser representada pela sequência 15 7 7 11.

Usando **8 bits** para cada um desses números, teremos uma cadeia de apenas 32 bits (ignore os espaços):

00001111000001110000011100001011

Codificação de comprimento de carreira

Decisões de projeto:

- ▶ Use **8 bits** (valores de 0 a 255) para cada comprimento de **carreira**.
- ▶ Use **carreiras de comprimento 0** para **dividir carreiras muito longas** em blocos de comprimento menor que 256.
- ▶ A **primeira carreira** é sempre de 0s (e pode ser vazia).

Codificação de comprimento de carreira

Para texto **ASCII**, a compressão é ruim

Exemplo: "**ABRACADABRA!**" tem carreiras curtas

```
% java BinaryDump 32 < abra.txt  
01000001010000100101001001000001  
01000011010000010100010001000001  
01000010010100100100000100100001  
96 bits
```

Codificação de comprimento de carreira

Para texto **ASCII**, a compressão é ruim

Exemplo: "ABRACADABRA!" tem carreiras curtas

```
% java RunLength - < abra.txt | java  
HexDump 13
```

```
01 01 05 01 01 01 04 01 02 01 01 01 02  
01 02 01 05 01 01 01 04 02 01 01 05 01  
01 01 03 01 03 01 05 01 01 01 04 01 02  
01 01 01 02 01 02 01 05 01 02 01 04 01  
416 bits
```

Taxa de compressão: $416/96 = 4.33$

Codificação de comprimento de carreira

Para **mapas de bits** (*bitmaps*) é boa

Exemplo: `q32x48.bin` representa uma letra "q" em 32 por 48 pixels:

```
% java PictureDump 32 48 < q32x48.bin  
1536 bits
```



Codificação de comprimento de carreira

```
% java BinaryDump 32 < q32x48.bin
```

```
000000000000000000000000000000000000
```

```
000000000000000000000000000000000000
```

```
000000000000000011111110000000000000
```

```
000000000000111111111111111000000000
```

```
000000000011110000111111111000000000
```

```
000000001111000000000111111000000000
```

```
000000011100000000000011111000000000
```

```
000000111100000000000011111000000000
```

```
000001111000000000000011111000000000
```

```
000011110000000000000011111000000000
```

```
000011110000000000000011111000000000
```

```
000111100000000000000011111000000000
```

```
000111100000000000000011111000000000
```

Codificação de comprimento de carreira

```
% java RunLength - < q32x48.bin >
```

```
q32x48.bin.rle
```

```
% java HexDump 13 < q32x48.bin.rle
```

```
4f 07 16 0f 0f 04 04 09 0d 04 09 06 0c
```

```
03 0c 05 0b 04 0c 05 0a 04 0d 05 09 04
```

```
0e 05 09 04 0e 05 08 04 0f 05 08 04 0f
```

```
05 07 05 0f 05 07 05 0f 05 07 05 0f 05
```

```
07 05 0f 05 07 05 0f 05 07 05 0f 05 07
```

```
05 0f 05 07 05 0f 05 07 06 0e 05 07 06
```

```
0e 05 08 06 0d 05 08 06 0d 05 09 06 0c
```

```
05 09 07 0b 05 0a 07 0a 05 0b 08 07 06
```

```
0c 14 0e 0b 02 05 11 05 05 05 1b 05 1b
```

```
05 1b 05 1b 05 1b 05 1b 05 1b 05 1b 05
```

```
1b 05 1b 05 1b 05 1a 07 16 0c 13 0e 41
```

Codificação de comprimento de carreira

```
% java RunLength - < q32x48.bin >  
q32x48.bin.rle
```

```
% java HexDump 0 < q32x48.bin
```

1536 bits

```
% java HexDump 0 < q32x48.bin.rle
```

1144 bits

Taxa de compressão: $1144/1536 = 0.74$

Codificação de comprimento de carreira

```
public static void compress() {  
    char cnt = 0;  
    boolean b, old = false;  
    while (!BinaryStdIn.isEmpty()) {  
        b = BinaryStdIn.readBoolean();  
        if (b != old) {  
            BinaryStdOut.write(cnt);  
            cnt = 0;  
            old = !old;  
        }  
    }  
}
```

Codificação de comprimento de carreira

```
else {  
    if (cnt == 255) {  
        BinaryStdOut.write(cnt);  
        cnt = 0;  
        BinaryStdOut.write(cnt);  
    }  
}  
cnt++;  
}  
BinaryStdOut.write(cnt);  
BinaryStdOut.close();  
}
```

Codificação de comprimento de carreira

```
public static void expand() {
    boolean b = false;
    while(!BinaryStdIn.isEmpty()) {
        // comprimento de uma carreira
        char cnt=BinaryStdIn.readChar();
        for (int i = 0; i < cnt; i++)
            // sai um único bit
            BinaryStdOut.write(b);
        b = !b;
    }
    BinaryStdOut.close();
}
```

Codificação de comprimento de carreira

Para mapas de bits a taxa de compressão **diminui linearmente** com o **aumento da resolução!**

```
% java PictureDump 32 48 < q32x48.bin  
1536 bits
```

Q

```
% java RunLength - < q32x48.bin | java  
BinaryDump 0  
1144 bits
```

Codificação de comprimento de carreira

Para mapas de bits a taxa de compressão **diminui linearmente** com o **aumento da resolução!**

```
% java PictureDump 64 96 < q64x96.bin  
6144 bits
```



```
% java RunLength - < q64x96.bin | java  
BinaryDump 0  
2296 bits
```