Run Length Encoding

Mark Matthews PhD

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1

40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s: 15 0s, then 7 1s, then 7 0s, then 11 1s.

$$\underbrace{1\ 1\ 1\ 1}_{15}\ \underbrace{0\ 1\ 1\ 1}_{7}\ \underbrace{0\ 1\ 1\ 1}_{7}\ \underbrace{1\ 0\ 1\ 1}_{11}$$

16 bits (instead of 40)

Q. How many bits to store the counts?

A. We'll use 8 (but 4 in the example above).

Q. What to do when run length exceeds max count?

A. If longer than 255, intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

Simple implementation of Run Length Encoding

```
// Java program to implement run length encoding

public class RunLength_Encoding {
    public static void printRLE(String str)
    {
        int n = str.length();
        for (int i = 0; i < n; i++) {

            // Count occurrences of current character
            int count = 1;
            while (i < n - 1 &&
                   str.charAt(i) == str.charAt(i + 1)) {
                count++;
                i++;
            }

            // Print character and its count
            System.out.print(str.charAt(i));
            System.out.print(count);
        }
    }

    public static void main(String[] args)
    {
        String str = "wwwwaaadexxxxxxywww";
        printRLE(str);
    }
}
```

Reading and writing binary data

Binary standard input. Read bits from standard input.

`public class BinaryStdIn`

`boolean`	`readBoolean()`	*read 1 bit of data and return as a* `boolean` *value*
`char`	`readChar()`	*read 8 bits of data and return as a* `char` *value*
`char`	`readChar(int r)`	*read* `r` *bits of data and return as a* `char` *value*
[similar methods for `byte` *(8 bits);* `short` *(16 bits);* `int` *(32 bits);* `long` *and* `double` *(64 bits)]*		
`boolean`	`isEmpty()`	*is the bitstream empty?*
`void`	`close()`	*close the bitstream*

Binary standard output. Write bits to standard output

`public class BinaryStdOut`

`void`	`write(boolean b)`	*write the specified bit*
`void`	`write(char c)`	*write the specified 8-bit* `char`
`void`	`write(char c, int r)`	*write the* `r` *least significant bits of the specified* `char`
[similar methods for `byte` *(8 bits);* `short` *(16 bits);* `int` *(32 bits);* `long` *and* `double` *(64 bits)]*		
`void`	`close()`	*close the bitstream*

Writing binary data

Date representation. Three different ways to represent 31/12/1999.

Character Stream Output

We could use a character stream which would give us 8 bits per character:
So with 10 characters (including the / char) that gives us 80 bits

Use Three Ints

We could use an int to represent the day, month and year respectively.
But ints in java are 32 bits each totally 96 bits (we're getting worse)

Binary Output

Looking at the limits of the date representation we can see that we only need:

- 5 bits to represent days
- 4 bits to represent months
- 12 bits to represent the years

This equals 21 bits (which rounds up to 24 bits given underlying representation of 8 bit units.

Run-length encoding: Java implementation

```
public class RunLength
{
   private final static int R   = 256;      ← maximum run-length count
   private final static int lgR = 8;        ← number of bits per count

   public static void compress()

   public static void expand()
   {
      boolean bit = false;
      while (!BinaryStdIn.isEmpty())
      {
         int run = BinaryStdIn.readInt(lgR);      ← read 8-bit count from standard input
         for (int i = 0; i < run; i++)
            BinaryStdOut.write(bit);          ← write 1 bit to standard output
         bit = !bit;
      }
      BinaryStdOut.close();               ← pad 0s for byte alignment
   }

}
```

An application: compress a bitmap

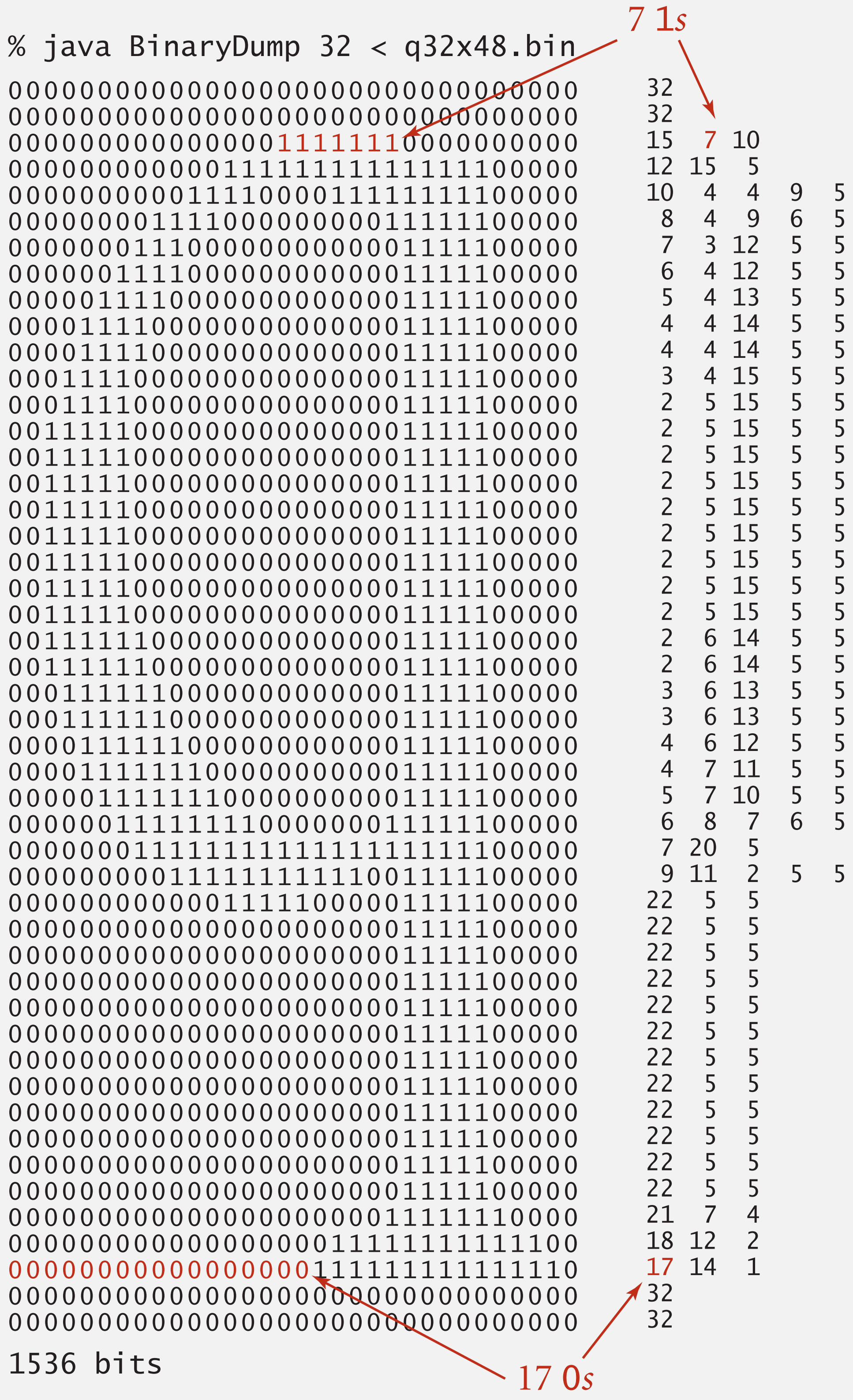
Typical black-and-white-scanned image.

- 300 pixels/inch.
- 8.5-by-11 inches.
- 300 × 8.5 × 300 × 11 = 8.415 million bits.

Observation. Bits are mostly white.

Typical amount of text on a page.

40 lines × 75 chars per line = 3,000 chars.



```
% java BinaryDump 32 < q32x48.bin
00000000000000000000000000000000    32
00000000000000000000000000000000    32
00000000000000011111110000000000    15  7 10
00000000000011111111111111100000    12 15  5
00000000001111000011111111100000    10  4  4  9  5
00000000111100000000011111100000     8  4  9  6  5
00000001110000000000001111100000     7  3 12  5  5
00000011110000000000001111100000     6  4 12  5  5
00000111100000000000001111100000     5  4 13  5  5
00001111000000000000001111100000     4  4 14  5  5
00001111000000000000001111100000     4  4 14  5  5
00011110000000000000001111100000     3  4 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111110000000000000001111100000     2  5 15  5  5
00111111000000000000001111100000     2  6 14  5  5
00111111000000000000001111100000     2  6 14  5  5
00011111100000000000001111100000     3  6 13  5  5
00011111100000000000001111100000     3  6 13  5  5
00001111110000000000001111100000     4  6 12  5  5
00001111111000000000001111100000     4  7 11  5  5
00000111111100000000001111100000     5  7 10  5  5
00000011111111000000011111100000     6  8  7  6  5
00000001111111111111111111100000     7 20  5
00000000011111111111001111100000     9 11  2  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000001111100000    22  5  5
00000000000000000000011111110000    21  7  4
00000000000000000011111111111100    18 12  2
00000000000000000111111111111110    17 14  1
00000000000000000000000000000000    32
00000000000000000000000000000000    32
1536 bits
```

7 1*s*

17 0*s*

A typical bitmap, with run lengths for each row

Black and white bitmap compression: another approach

Fax machine (~1980).

- Slow scanner produces lines in sequential order.
- Compress to save time (reduce number of bits to send).

Electronic documents (~2000).

- High-resolution scanners produce huge files.
- Compress to save space (reduce number of bits to save).

Idea.

- use OCR to get back to ASCII (!)
- use Huffman on ASCII string (!)

Bottom line. Any extra information about file can yield dramatic gains.