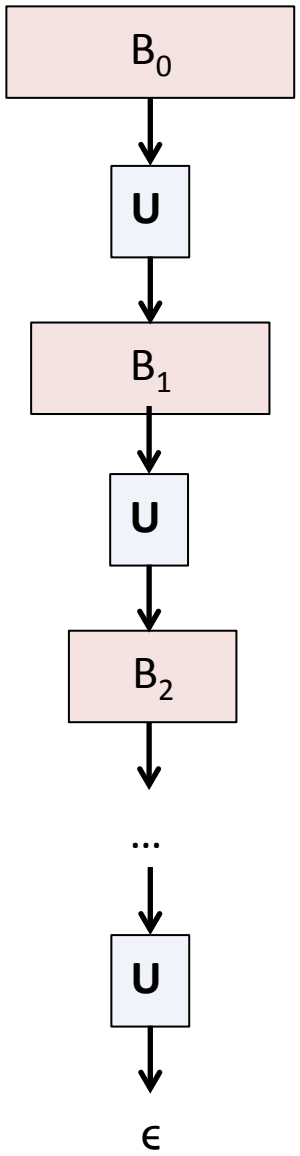
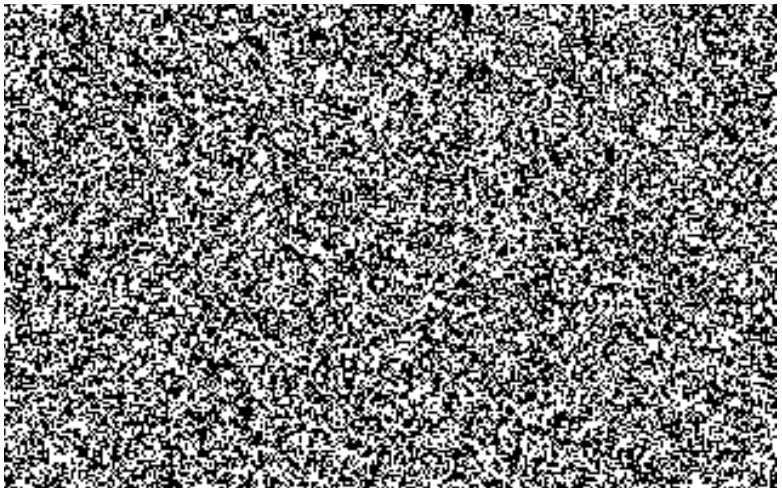
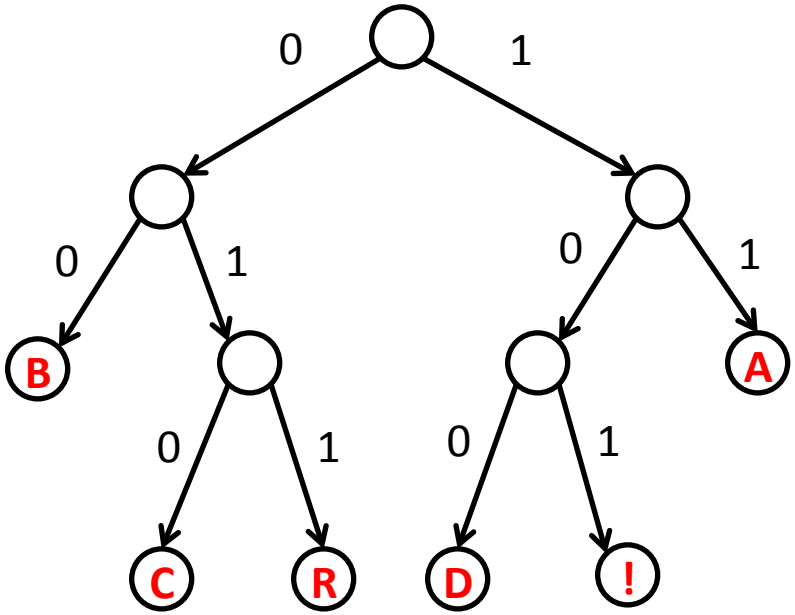
Lossless compression







Overview

Section 5.5

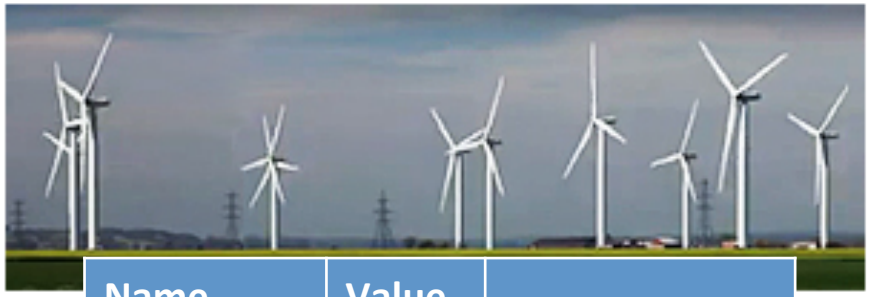
- Lossless compression
  - Motivation
  - Rules and limits of the game
  - Things to exploit
- Run-length encoding (RLE)
  - Exploit runs of same character
- Huffman coding
  - Variable-length codeword for each pattern (character)
  - Transmit codewords plus compressed data

Motivation

- Lossless compression
  - Reduce size of a file
  - Save space while storing it
    - Data always expands to fill available drive space
  - Save space while transmitting it
    - Bandwidth growing rapidly, but so are files!
    - HD video:
      - (1920 * 1080) pixels/frame * 30 frames/sec * 24 bits/pixel = 1.5Gbps!
  - Lossless = get back exactly what you put in (e.g. zip)
- Lossly compression (stay tuned)
  - Information is lost (e.g. JPEG, MP3)

What is big data?

Every day, we create 2.5 quintillion bytes of data — so much that 90% of the data in the world today has been created in the last two years alone. This data comes from everywhere: sensors used to gather climate information, posts to social media sites, digital pictures and videos, purchase transaction records, and cell phone GPS signals to name a few. This data is **big data.**



Name	Value	
Million	10^6	megabyte
Billion	10^9	gigabyte
Trillion	10^{12}	terabyte
Quadrillion	10^{15}	petabyte
Quintillion	10^{18}	exabyte

Big data spans three dimensions: Volume, Velocity and Variety.

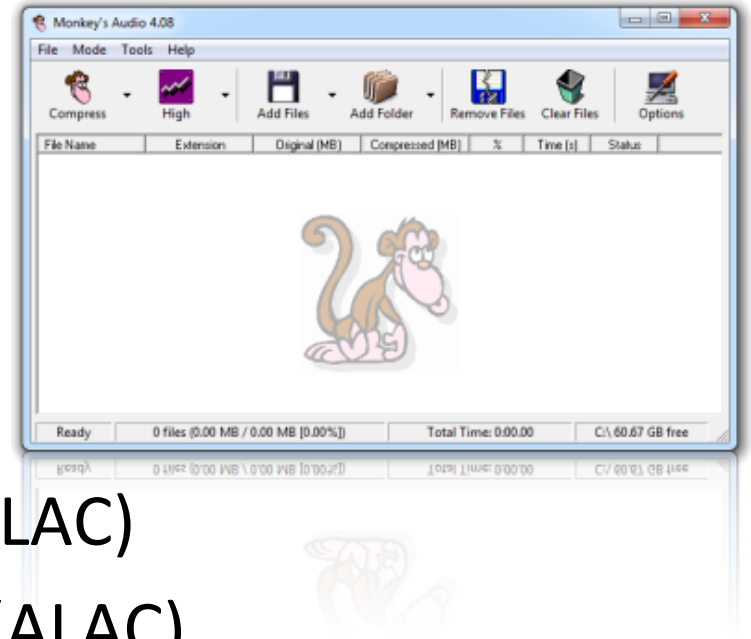
Volume: Enterprises are awash with ever-growing data of all types, easily amassing terabytes —even petabytes—of information.

- Turn 12 terabytes of Tweets created daily into improved product sentiment analysis
- Convert 350 billion meter readings per annum to better predict power consumption

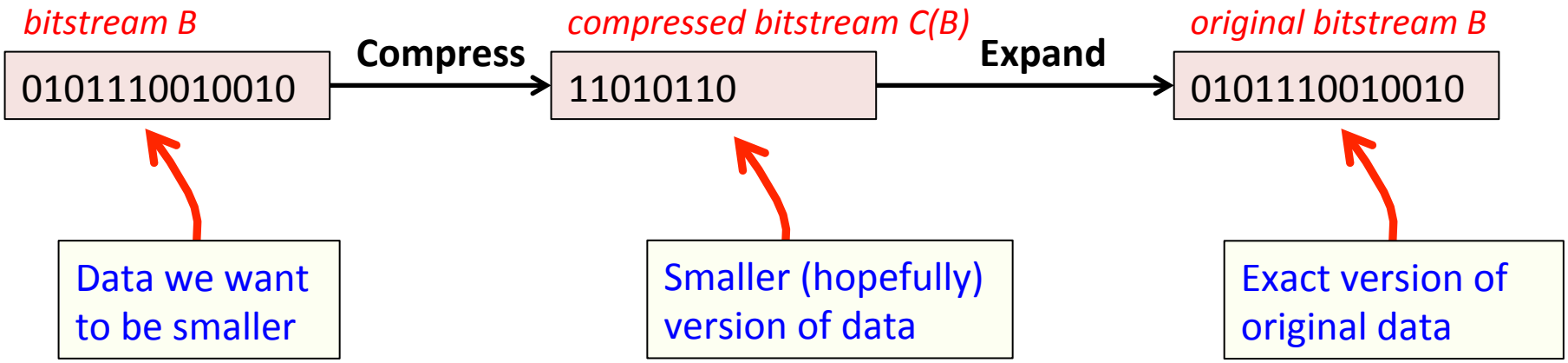
http://www-01.ibm.com/software/data/bigdata/

Lossless compression: applications

- Generic file compression
  - compress, gzip, zip, bzip2, 7z, xz
  - NTFS, HFS+, ZFS
- Image files
  - GIF, PNG, TIFF
- Audio files
  - Free Lossless Audio Codec (FLAC)
  - Apple Lossless Audio Codec (ALAC)
- Data transmission
  - HTTP, PPP, SSH, fax machines, v.92 modems



Compression and expansion



Compression ratio:
bits in C(B) / bits in B

Example:
17 ASCII characters, 7 bits each = 119 bits
Output 12 codewords, 8 bits/codeword = 96 bits
Compression ratio = 81%

```
219,743,396 katie30390.txt          plaintext
 52,583,716 katie30390.txt.Z        compress    LZW
 31,875,422 katie30390.txt.zip      zip         DEFLATE
 23,862,152 katie30390.txt.bz2      bzip2       Burrows-Wheeler
 21,814,286 katie30390.txt.lzma2.7z 7-zip       LZMA2
 21,579,605 katie30390.txt.7z       7-zip       LZMA
 21,320,608 katie30390.txt.xz       xz          LZMA2
 21,271,440 katie30390.txt.ppmd.7z  7-zip       PPMd
```

United States Patent [19]

James

[11] **Patent Number: 5,533,051**

[45] **Date of Patent: Jul. 2, 1996**

[54] **METHOD FOR DATA COMPRESSION**

"A second aspect of the present invention which further enhances its ability to achieve high compression percentages, is its ability to be applied to data recursively. Specifically, the methods of the present invention are able to make multiple passes over a file, each time further compressing the file. Thus, a series of recursions are repeated until the desired compression level is achieved."

H03M 7/30; H03M 7/34

341/87; 51; 348/415; 409; 590; 384; 382/244; 364/715.02; 380/42, 49

"the direct bit encode method of the present invention is effective for reducing an input string by one bit regardless of the bit pattern of the input string."

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,694,813	9/1972	Loh et al.	340/172.5
4,369,463	1/1983	Anastassiou .	
4,491,934	1/1985	Heinz	364/900
4,545,032	10/1985	Mak	364/900
4,560,976	12/1985	Finn	341/51
4,597,057	6/1986	Snow	364/900
4,633,490	12/1986	Mitchell .	
4,652,856	2/1986	Mohiuddin .	
4,672,539	6/1987	Goertzel	364/300
4,725,884	2/1988	Gonzales .	
4,748,577	5/1988	Marchant	364/722
4,796,003	1/1989	Bentley	341/95
4,881,075	11/1989	Weng	341/87

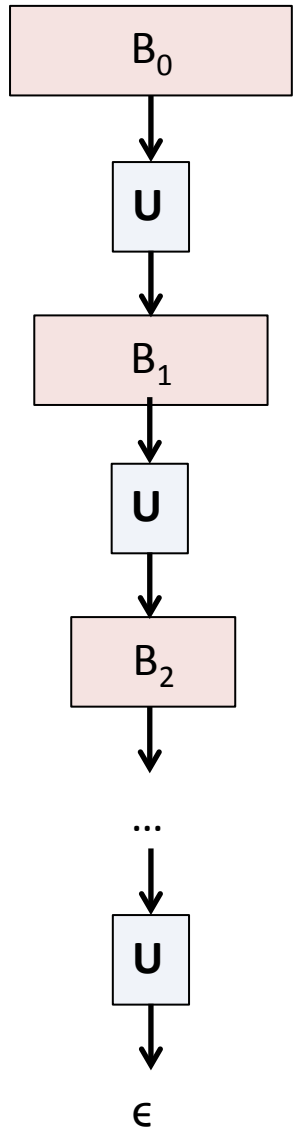
Attorney, Agent, or Firm—Dykema Gossett

[57] **ABSTRACT**

Methods for compressing data including methods for compressing highly randomized data are disclosed. Nibble encode, distribution encode, and direct bit encode methods are disclosed for compressing data which is not highly randomized. A randomized data compression routine is also disclosed and is very effective for compressing data which is highly randomized. All of the compression methods disclosed operate on a bit level and accordingly are insensitive to the nature or origination of the data sought to be compressed. Accordingly, the methods of the present invention are universally applicable to any form of data regardless of its source of origination.

Universal data compression?

- **Reality:** No algorithm can compress *every* bitstream
- Proof 1 (by contradiction)
  - Suppose you have a universal compressor U
  - Given bitstream B_0, use U to compress to smaller B_1
  - Compress B_1 to get smaller B_2
  - Continue until bitstream is of size 0
  - Thus all bitstream can be compressed to 0 bits!
- Proof 2 (counting)
  - Suppose you can compress all 1000-bit strings
  - 2^{1000} possible bit strings with 1000 bits
  - How many possible shorter encodings, ≤ 999 bits?
    - (# of 1 bit numbers) + (# of 2 bit numbers) + … + (# of 999 bit numbers)
    - $1 + 2 + 4 + \ldots + 2^{999} = 2^{1000} - 1$
  - Thus fewer than the 2^{1000} we need for unique mapping



1024 x 768 = 786,432 bits

```
java PictureDump 1024 768 < bits.bin
```

- Compressed with standard compression utilities
- My top-secret method!

```
98,304  bits.bin
98,346  bits.bin.gz
98,571  bits.bin.zip
99,080  bits.bin.bz2
98,368  bits.bin.xz
   232  bits.bin.kdv
```

232 * 8 / 786,432
0.24% of original!

```
public class RandomBits
{
   public static void main(String [] args)
   {
      int x = 1111;
      for (int i = 0; i < 786432; i++)
      {
         x = x * 314159 + 218291;
         BinaryStdOut.write(x > 0);
      }
      BinaryStdOut.close();
   }
}
```

Another set of 1024 x 768 = 786,432 bits

What is the optimal compressor for this image?

Undecideable!

In fact this image is completely random.

```
java PictureDump 1024 768 < 2012-04-22.bin
```

```
1,048,576 2012-04-22.bin
1,053,488 2012-04-22.bz2
1,048,769 2012-04-22.gz
1,049,000 2012-04-22.zip
```

RANDOM.ORG

Search RANDOM.ORG

Google™ Custom Search | Search

True Random Number Service

What's this fuss about *true* randomness?

Perhaps you have wondered how predictable machines like computers can generate randomness. In reality, most random numbers used in computer programs are *pseudo-random*, which means they are a generated in a predictable fashion using a mathematical formula. This is fine for many purposes, but it may not be random in the way you expect if you're used to dice rolls and lottery drawings.

RANDOM.ORG offers *true* random numbers to anyone on the Internet. The randomness comes from atmospheric noise, which for many purposes is better than the pseudo-random number algorithms typically used in computer programs. People use RANDOM.ORG for holding drawings, lotteries and sweepstakes, to drive games and gambling sites, for scientific applications and for art and music. The service has existed since 1998 and was built and is being operated by Mads Haahr of the School of Computer Science and Statistics at Trinity College, Dublin in Ireland.

As of today, RANDOM.ORG has generated 1,108 billion random bits for the Internet community.

True Random Number Generator

Min: 1

Max: 100

Generate

Result:

Powered by RANDOM.ORG

Like RANDOM.ORG?

Get the Newsletter

Redundancy in English

- How redundant is written English?

> Yet aoccdrnig to a sudty at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a ttoal mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

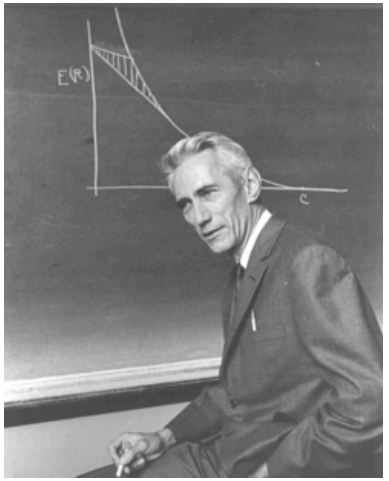
- Answer: very!
  - Shannon estimate: 0.6 - 1.3 bits per letter

Prediction and Entropy of Printed English

By C. E. SHANNON

(*ManuscriptReceived Sept. 15, 1950*)

A new method of estimating the entropy and redundancy of a language is described. This method exploits the knowledge of the language statistics possessed by those who speak the language, and depends on experimental results in prediction of the next letter when the preceding text is known. Results of experiments in prediction are given, and some properties of an ideal predictor are developed.



Approaches to compression

- Exploit 1+ of the following:
  1) Small alphabets
  2) Long sequences of identical bits
  3) Frequently used characters
  4) Long reused bit sequences (next time)
- We'll look at an example of each
  - Including Java implementation
  - Using support classes for:
    - Binary file input/output
    - Data structures

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Reading and writing binary data

```
public class BinaryStdIn
-----------------------------------------------------
boolean readBoolean()   // Read 1 bit of data, return as a boolean value
   char readChar()      // Read 8 bits of data, return as a char value
   char readChar(int r) // Read r bits of data, return as a char value
                        // Read r bits for byte, short, int, long, double
boolean isEmpty()       // Is the bitstream empty?
   void close()         // Close the bitstream
```

```
public class BinaryStdOut
-----------------------------------------------------
   void write(boolean b)     // Write the specified bit
   void writeChar(char c)    // Write the specified 8-bit char
   void write(char c, int r) // Write r least significant bits of char c
                             // Write r LSB of byte, short, int, long, double
   void close()              // Close the bitstream
```

Visualizing a bitstream

- How to view a bitstream?

```
% more abra.txt
ABRACADABRA!
```

Bitstream as characters

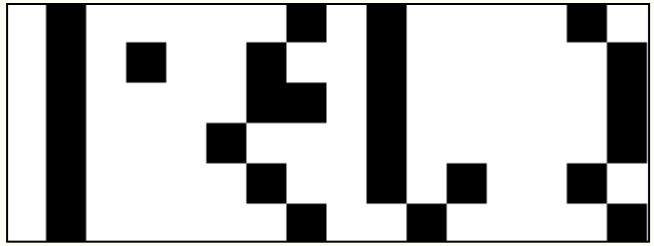
```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
00001010
104 bits
```

Bitstream represented by 0 and 1's

```
% java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
0a
104 bits
```

Bitstream represented by 2-digit hex numbers

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



Bitstream as pixels in a picture

Method 1: Small alphabets

```
public class OutputDate
{
   public static void main(String [] args)
   {
      int month = 12;
      int day = 31;
      int year = 1999;
      int mode = Integer.parseInt(args[0]);
      if (mode == 0)
      {
         System.out.print(month+"/"+day+"/"+year);
      }
      else if (mode == 1)
      {
         BinaryStdOut.write(month);
         BinaryStdOut.write(day);
         BinaryStdOut.write(year);
         BinaryStdOut.close();
       }
       else
       {
          BinaryStdOut.write(month, 4);
          BinaryStdOut.write(day, 5);
          BinaryStdOut.write(year, 12);
          BinaryStdOut.close();
       }
    }
}
```

```
% java OutputDate 0
| java BinaryDump 8

1  00110001
2  00110010
/  00101111
3  00110011
1  00110001
/  00101111
1  00110001
9  00111001
9  00111001
9  00111001
80 bits
```

Mode 0: output as ASCII text
Mode 1: output four 32-bit ints
Mode 2: output variable length bit fields

Method 1: Small alphabets

```
public class OutputDate
{
   public static void main(String [] args)
   {
      int month = 12;
      int day = 31;
      int year = 1999;
      int mode = Integer.parseInt(args[0]);
      if (mode == 0)
      {
         System.out.print(month+"/"+day+"/"+year);
      }
      else if (mode == 1)
      {
         BinaryStdOut.write(month);
         BinaryStdOut.write(day);
         BinaryStdOut.write(year);
         BinaryStdOut.close();
       }
       else
       {
          BinaryStdOut.write(month, 4);
          BinaryStdOut.write(day, 5);
          BinaryStdOut.write(year, 12);
          BinaryStdOut.close();
       }
    }
}
```

```
% java OutputDate 1 | java BinaryDump 32

00000000000000000000000000001100  12
00000000000000000000000000011111  31
00000000000000000000011111001111  1999
96 bits
```

Mode 0: output as ASCII text
Mode 1: output four 32-bit ints
Mode 2: output variable length bit fields

Method 1: Small alphabets

```
public class OutputDate
{
   public static void main(String [] args)
   {
      int month = 12;
      int day = 31;
      int year = 1999;
      int mode = Integer.parseInt(args[0]);
      if (mode == 0)
      {
         System.out.print(month+"/"+day+"/"+year);
      }
      else if (mode == 1)
      {
         BinaryStdOut.write(month);
         BinaryStdOut.write(day);
         BinaryStdOut.write(year);
         BinaryStdOut.close();
       }
       else
       {
          BinaryStdOut.write(month, 4);
          BinaryStdOut.write(day, 5);
          BinaryStdOut.write(year, 12);
          BinaryStdOut.close();
       }
    }
}
```

Compressing by using a small alphabet: e.g. use 2-bit code for nucleotides {A, C, T, G}

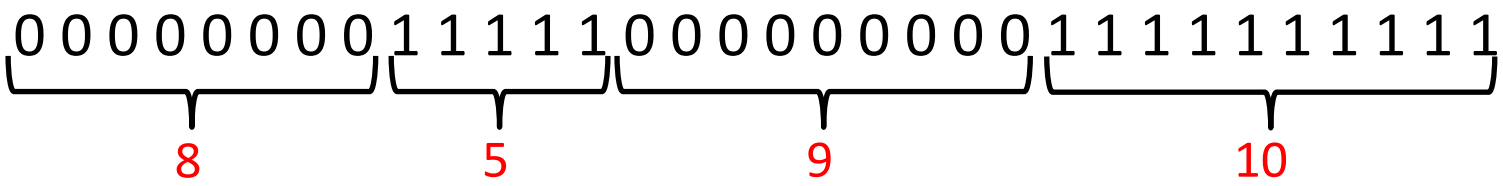
```
% java OutputDate 2 | java BinaryDump 32
12  31  1999
11001111101111100111 1000
24 bits
```

Padding since we must end on a byte boundary.

Mode 0: output as ASCII text
Mode 1: output four 32-bit ints
Mode 2: output variable length bit fields

Method 2: Long sequences

- Run Length Encoding (RLE)
  - Exploits simple form of redundancy
  - Long runs of same bit value



  - Store the count using 8-bits (0-255)
  - Alternate between 0 and 1
  - If >255, put run of length 0 of other bit, continue

```
public class RunLength
{
    private static final int R   = 256;            // Maximum run-length count
    private static final int lgR = 8;              // Number of bits per count

    public static void compress()
    {
        char run = 0;
        boolean old = false;                       // Start out with 0-bit
        while (!BinaryStdIn.isEmpty())
        {
            boolean b = BinaryStdIn.readBoolean();
            if (b != old)                          // Did the bit value change?
            {
                BinaryStdOut.write(run, lgR);      // Write out the count for completed run
                run = 1;
                old = !old;
            }
            else
            {
                if (run == R-1)                    // We have reached 255, time to output
                {
                    BinaryStdOut.write(run, lgR);  // Write run of 255
                    run = 0;
                    BinaryStdOut.write(run, lgR);  // Write a run of 0 of the other bit
                }
                run++;
            }
        }
        BinaryStdOut.write(run, lgR);
        BinaryStdOut.close();
    }
```

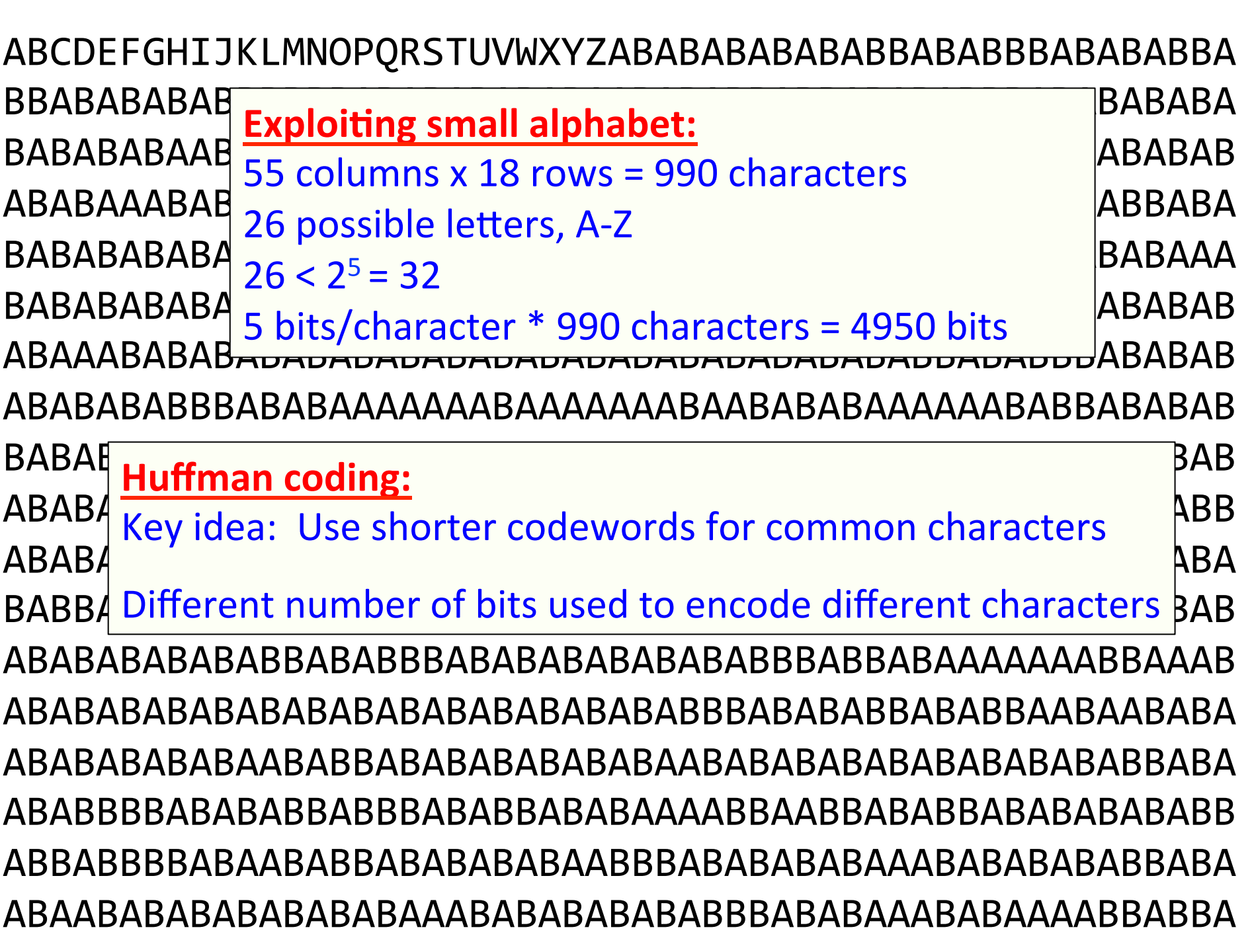
```
    public static void expand()
    {
        boolean b = false;
        while (!BinaryStdIn.isEmpty())
        {
            int run = BinaryStdIn.readInt(lgR); // Read 8-bit count from stdin
            for (int i = 0; i < run; i++)
                BinaryStdOut.write(b);          // Write 1-bit to stdout
            b = !b;
        }
        BinaryStdOut.close();                   // Pads 0s to get to byte boundary
    }

    public static void main(String[] args)
    {
        if      (args[0].equals("-")) compress();
        else if (args[0].equals("+")) expand();
        else throw new RuntimeException("Illegal command line argument");
    }
}
```

```
% java BinaryDump 32 < q32x48.bin
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000001111111000000000
00000000000011111111111111100000
00000000001111000011111111100000
00000000111100000000011111100000
00000011110000000000001111100000
00000111100000000000001111100000
00001111000000000000001111100000
00011110000000000000001111100000
00011110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00111110000000000000001111100000
00011111000000000000001111100000
00011111000000000000001111100000
00001111100000000000001111100000
00000111111000000000001111100000
00000011111111100000011111100000
00000001111111111111111111100000
00000000111111111111001111100000
00000000000011111000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111100000
00000000000000000000001111110000
00000000000000000011111111111100
00000000000000000111111111111110
00000000000000000000000000000000
00000000000000000000000000000000
1536 bits
```

```
bit-0 run = 79 (01001111)
bit-1 run = 7  (00000111)
bit-0 run = 22 (00010110)
bit-1 run = 15 (00001111)
...
```

```
% java RunLength - < q32x48.bin | java
BinaryDump 8
01001111
00000111
00010110
00001111
00001111
00000100
00000100
00001001
00001101
00000100
...
1144 bits
```

```
Compression ratio:
1144 / 1536 = 74%
```


ABCDEFGHIJKLMNOPQRSTUVWXYZABABABABABBABABBBABABABBA
BBABABABAB
BABABABAAB
ABABAAABAB
BABABABABA
BABABABABA
ABAAABABAB
ABABABABBBABABAAAAAAABAAAAAAABAABABABAAAAAABABBABABAB
BABAB
ABABA
ABABA
BABBA
ABABABABABABBABABBBABABABABABABABBBABBABAAAAAAABBAAAB
ABABABABABABABABABABABABABABABBBABABABBABABBAABAABABA
ABABABABABAABABBABABABABABABABAABABABABABABABABABABBABA
ABABBBBABABABBABBBABABBABABAAAABBAABBABABBABABABABABB
ABBABBBBABAABABBABABABABAABBBABABABABAAABABABABABBABA
ABAABABABABABABABAAABABABABABABBBABABAAABABAAAABBABBA
Exploiting small alphabet:
55 columns x 18 rows = 990 characters
26 possible letters, A-Z
26 < 2^5 = 32
5 bits/character * 990 characters = 4950 bits
Huffman coding:
Key idea: Use shorter codewords for common characters
Different number of bits used to encode different characters

Method 3: Frequency of characters

- Variable-length prefix-free codes
  - Map from characters to bit strings (codewords)
  - Choose codewords so none is a prefix of another

ABRACADABRA!

Text to be compressed

key	value
!	101
A	0
B	1111
C	110
D	100
R	1110

key	value
!	101
A	11
B	00
C	010
D	100
R	011

```
01111111001100100011111100101
A   B   RA  CA  DA   B   RA  !
```

Scheme 1: ***30 bits***

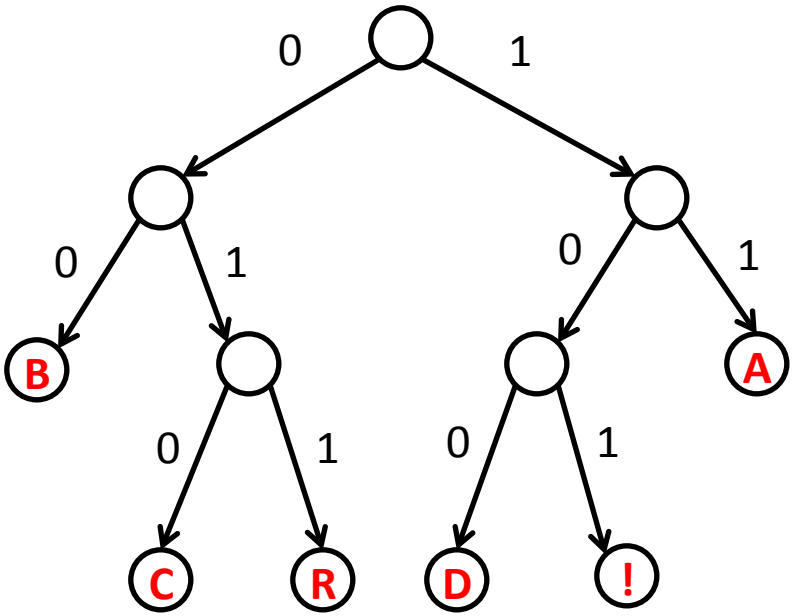
```
11000111101011100110001111101
 A B  R A  C A  D A B  R A  !
```

Scheme 2: ***29 bits***

Trie representation

- Represent prefix-free code as a binary trie
  - Characters at leafs
  - Codeword is path from root to leaf
  - 0 = left, 1 = right

key	value
!	101
A	11
B	00
C	010
D	100
R	011




```
11000111101011100110001111101
 A B  R A  C A  D A B  R A  !
```

29 bits

Using the trie

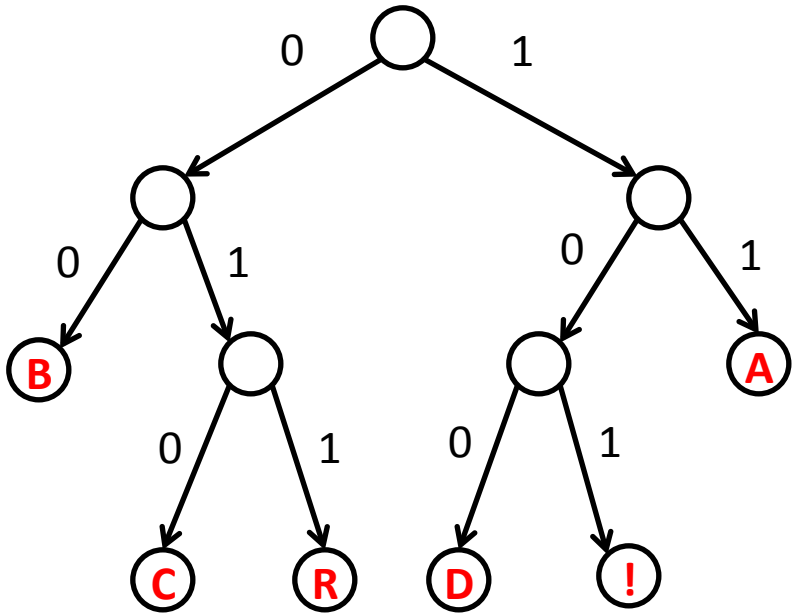
- **Compression**
  - Start at leaf of target character
  - Follow path to root, print bits in reverse order
  - …or create a symbol table

- **Expansion**
  - Start at root
  - Go left if bit = 0, go right if bit = 1
  - If leaf node, output character and return to root

key	value
!	101
A	11
B	00
C	010
D	100
R	011

```
11000111101011100110001111101
 A B  R A  C A  D A B  R A  !
```

29 bits

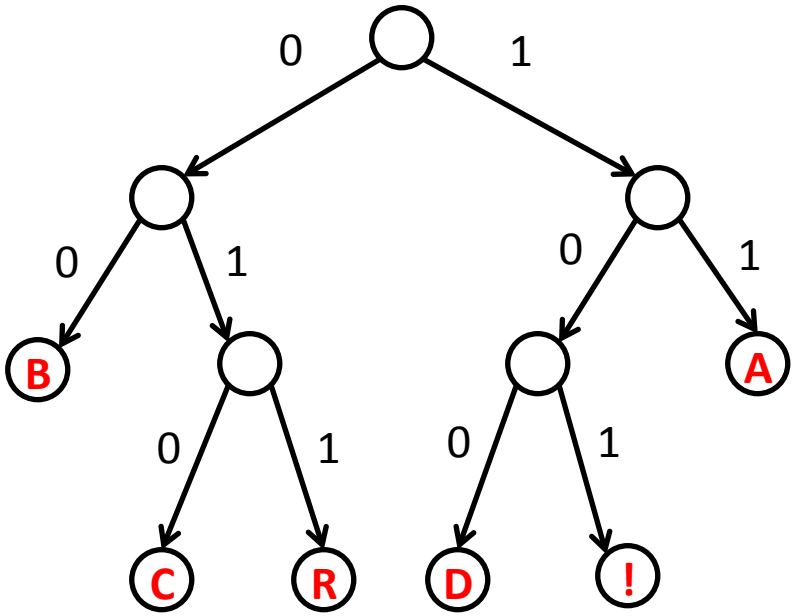


```
private static class Node implements Comparable<Node>
{
   private final char ch;
   private final int freq;
   private final Node left, right;

   // Initialize a new Node
   Node(char ch, int freq, Node left, Node right)
   {
      this.ch    = ch;
      this.freq  = freq;
      this.left  = left;
      this.right = right;
   }

   // Is this node a leaf?
   private boolean isLeaf()
   {
      return (left == null && right == null);
   }

   // Compare Nodes by frequency
   public int compareTo(Node that)
   {
      return this.freq - that.freq;
   }
}
```

Expansion

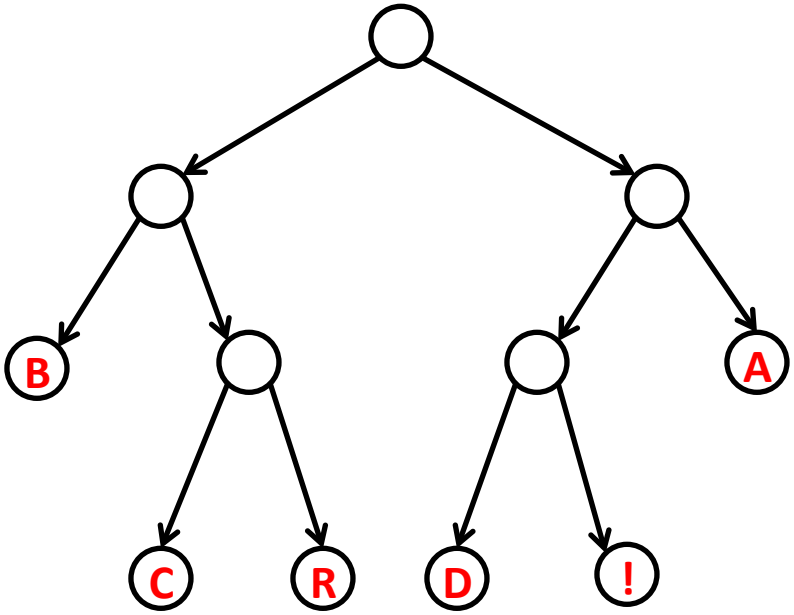
```
public static void expand()
{
   // Read in the encoding trie
   Node root = readTrie();

   // Number of bytes to write
   int length = BinaryStdIn.readInt();

   // Decode using the Huffman trie
   for (int i = 0; i < length; i++)
   {
      Node x = root;

      // Expand codeword for the ith character
      while (!x.isLeaf())
      {
         boolean bit = BinaryStdIn.readBoolean();
         if (bit)
            x = x.right;
         else
            x = x.left;
      }
      BinaryStdOut.write(x.ch);
   }
   BinaryStdOut.flush();
}
```

- Expansion
  - Start at root
  - Go left if bit=0, go right if bit=1
  - If leaf node, output character and return to root



Transmitting the trie

- Trie needed in order to expand
  - Must be sent along with the data
    - Causes overhead, but small if message is long
  - Write preorder traversal of trie
  - Mark leaf and internal nodes with a bit



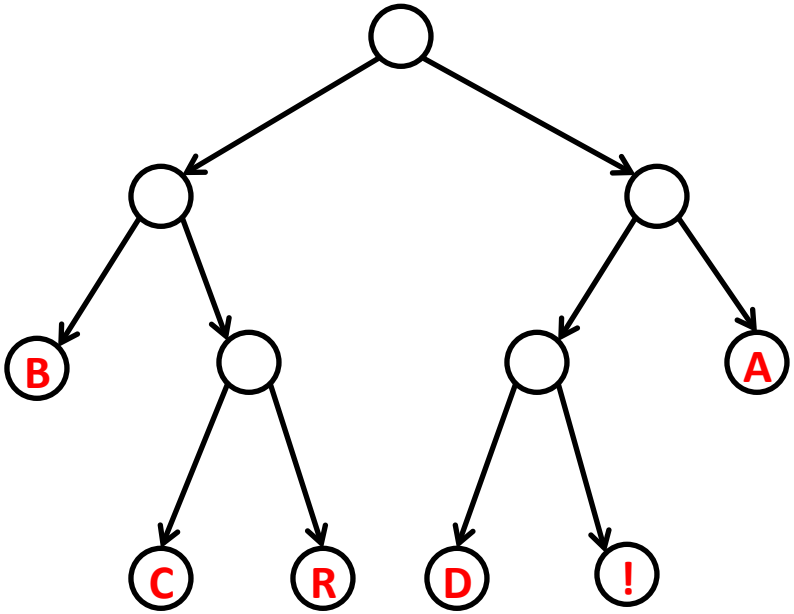

```
private static void writeTrie(Node x)
{
   if (x.isLeaf())
   {
      BinaryStdOut.write(true);
      BinaryStdOut.write(x.ch);
      return;
   }
   BinaryStdOut.write(false);
   writeTrie(x.left);
   writeTrie(x.right);
}
```

00101000010010100001110101001000101000100100001010101000001

B C R D ! A

Reading the trie

- Reconstructing from preorder traversal
  - Use 0/1 bits to decide if internal or leaf node



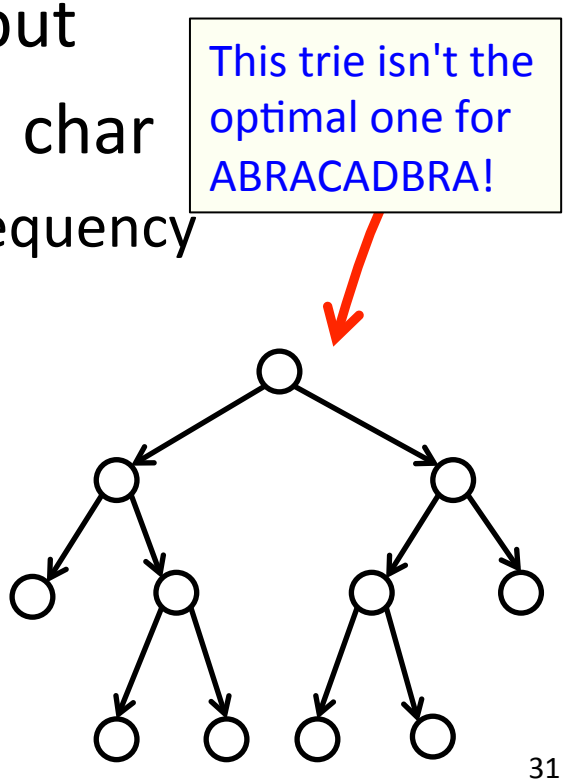

```
private static Node readTrie()
{
   boolean isLeaf = BinaryStdIn.readBoolean();
   if (isLeaf)
   {
      return new Node(BinaryStdIn.readChar(),
                      -1,
                      null,
                      null);
   }
   else
   {
      return new Node('\0',
                       -1,
                       readTrie(),
                       readTrie());
   }
}
```

00101000010010100001110101001000101000100100001010101000001

B C R D ! A

Building the trie

- Can we always find optimal prefix-free code?
  - Yes!
  - Discovered by David Huffman while a PhD student
- Huffman algorithm:
  - Count frequency of each char in input
  - Create forest of leaf nodes for each char
    - Each leaf weighted according to its frequency
  - Repeat until single trie:
    - Select 2 tries with min sum of weights
    - Merge into trie with sum of weights
  - Provably optimal



ABRACADABRA!

6) Join last two tries

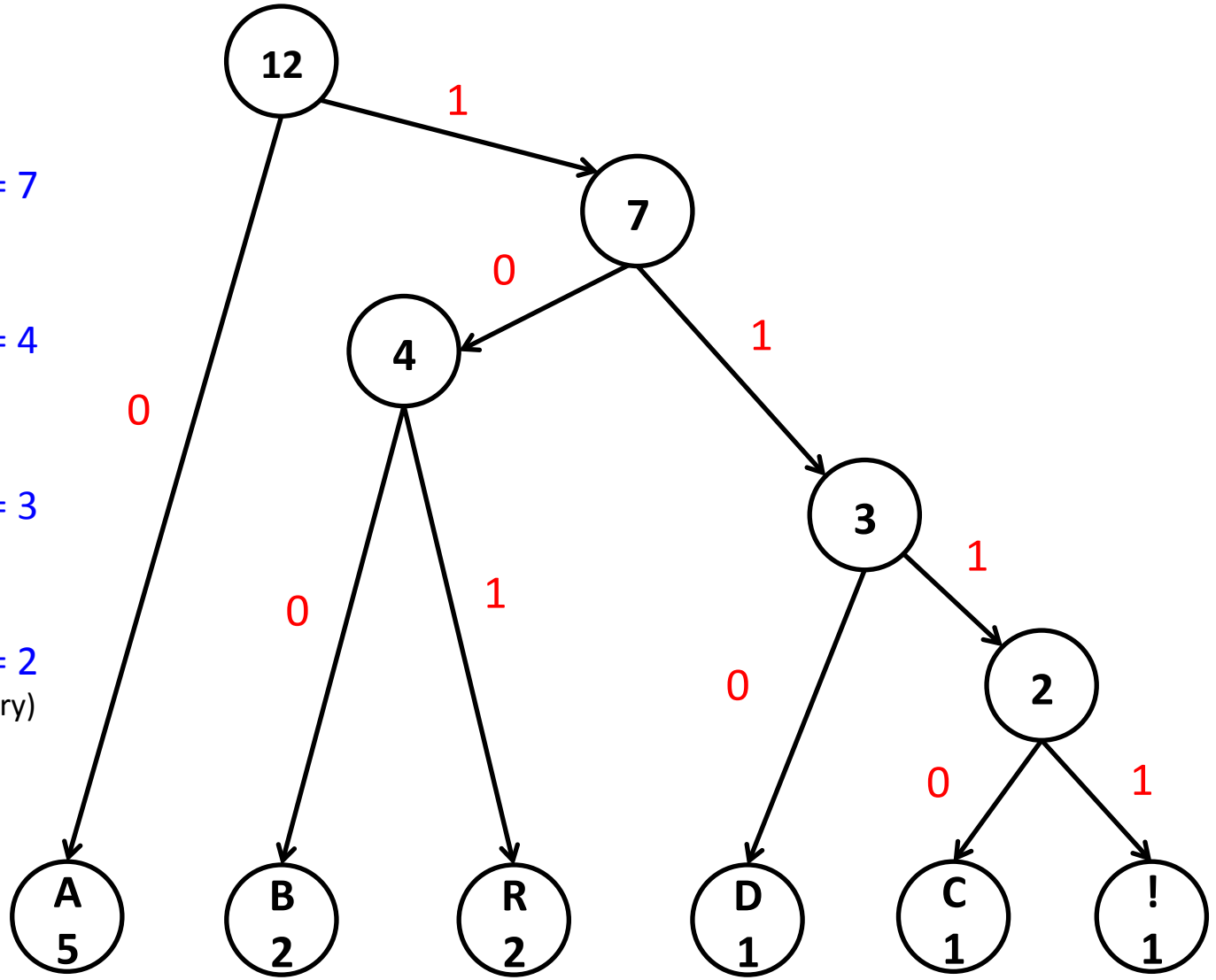
5) Join two with sum = 7

4) Join two with sum = 4

3) Join two with sum = 3

2) Join two with sum = 2
(in a tie, exact choice arbitrary)

1) Create leafs with weight = frequency in entire text

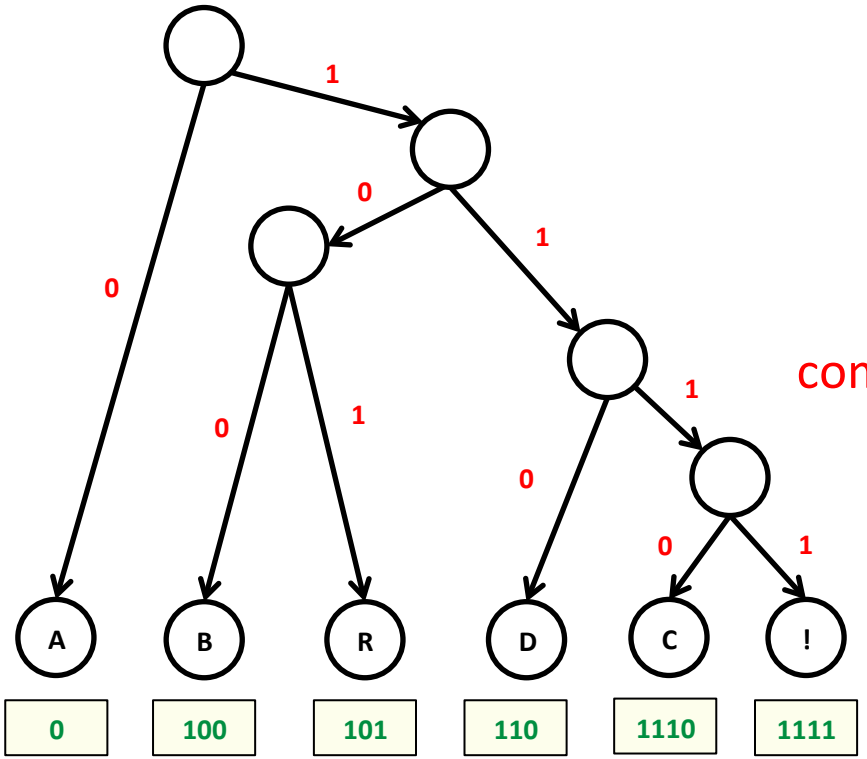


Codewords:

0	100	101	110	1110	1111

ABRACADABRA!

```
% java Huffman - < abra.txt | java
BinaryDump 32
01010000010010100010001001000011
01000011010101001010100001000000
00000000000000000000000110001111
10010110100011111010100
120 bits
```



trie

```
01        01000001   65 = A
001       01000100   68 = D
01        00100001   33 = !
1         01000011   67 = C
01        01010010   82 = R
1         01000010   66 = B
```

size of text

```
          00000000   int = 12
          00000000
          00000000
          00001100
```

compressed data

```
0                    A
111                  B
110                  R
0                    A
1011                 C
0                    A
100                  D
0                    A
111                  B
110                  R
0                    A
1010                 !
0                    padding
```

Summary

- Lossless compression
  - Universal compression: impossible
  - Optimal data compression: undecidable
- Exploiting:
  - Small alphabets
    - Use only as many bits as needed to represent data
  - Repeated symbols
    - Run length encoding (RLE)
  - Frequency of symbols
    - Prefix-free codes
    - Huffman coding