



<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*



<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*

# Data compression

---

## Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

## Who needs compression?

- Moore's law: # transistors on a chip doubles every 18–24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, ...

*“ Everyday, 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. ” — IBM report on big data (2011)*

Basic concepts ancient (1950s), best technology recently developed.

# Applications

---

## Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, HFS+, ZFS.



## Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.



## Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.
- Skype.



Databases. Google, Facebook, ....



# Lossless compression and expansion

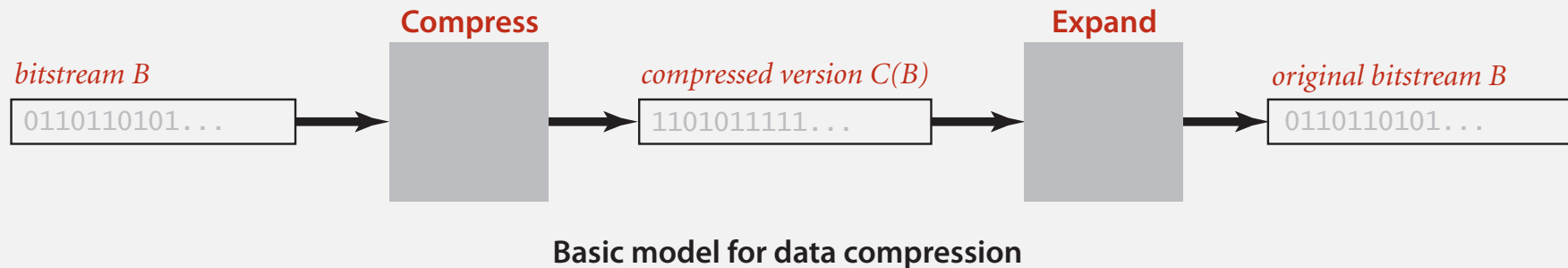
---

**Message.** Binary data  $B$  we want to compress.

**Compress.** Generates a "compressed" representation  $C(B)$ .

**Expand.** Reconstructs original bitstream  $B$ .

uses fewer bits (you hope)



**Compression ratio.** Bits in  $C(B)$  / bits in  $B$ .

**Ex.** 50–75% or better compression ratio for natural language.

# Food for thought

---

Data compression has been omnipresent since antiquity:

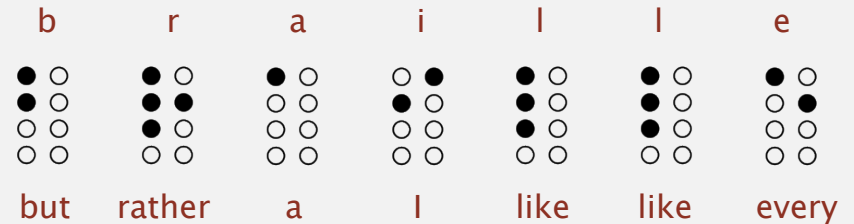
- Number systems.
- Natural languages.
- Mathematical notation.



$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

has played a central role in communications technology,

- Grade 2 Braille.
- Morse code.
- Telephone system.



and is part of modern life.

- MP3.
- MPEG.



Q. What role will it play in the future?

# Data representation: genomic code

---

**Genome.** String over the alphabet { A, C, T, G }.

**Goal.** Encode an  $N$ -character genome: A T A G A T G C A T A G . . .

**Standard ASCII encoding.**

- 8 bits per char.
- $8N$  bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

**Two-bit encoding.**

- 2 bits per char.
- $2N$  bits.

char	binary
A	00
C	01
T	10
G	11

**Fixed-length code.**  $k$ -bit code supports alphabet of size  $2^k$ .

**Amazing but true.** Initial genomic databases in 1990s used ASCII.

# Reading and writing binary data

---

Binary standard input and standard output. Libraries to read and write **bits** from standard input and to standard output.

```
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
```

```
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
```

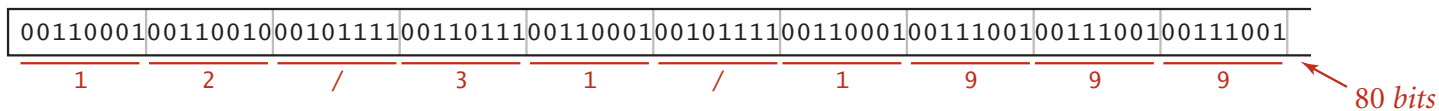


# Writing binary data

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

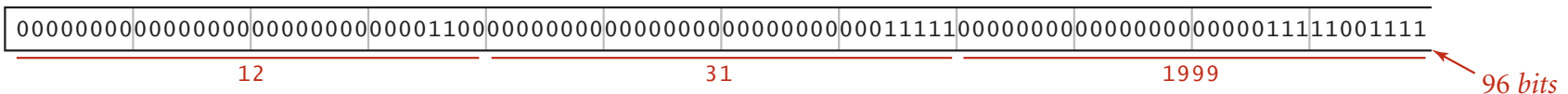
## A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year);
```



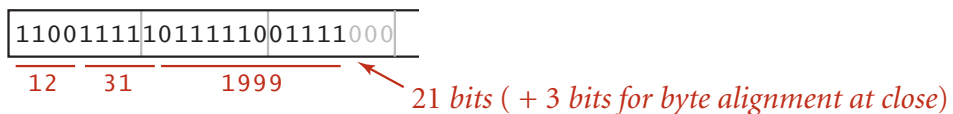
## Three ints (BinaryStdOut)

```
BinaryStdOut.write(month);  
BinaryStdOut.write(day);  
BinaryStdOut.write(year);
```



## A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```
BinaryStdOut.write(month, 4);  
BinaryStdOut.write(day, 5);  
BinaryStdOut.write(year, 12);
```



# Binary dumps

## Q. How to examine the contents of a bitstream?

### Standard character stream

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

### Bitstream represented as 0 and 1 characters

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

### Bitstream represented with hex digits

```
% java HexDump 4 < abra.txt  
41 42 52 41  
43 41 44 41  
42 52 41 21  
12 bytes
```

### Bitstream represented as pixels in a Picture

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



← 16-by-6 pixel window, magnified

96 bits

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal to ASCII conversion table

# Universal data compression

---

[US Patent 5,533,051](#) on "Methods for Data Compression", which is capable of compression **all** files.

[Slashdot](#) reports of the Zero Space Tuner™ and BinaryAccelerator™.

*“ ZeoSync has announced a breakthrough in data compression that allows for 100:1 lossless compression of **random** data. If this is true, our bandwidth problems just got a lot smaller.... ”*

# Universal data compression

---

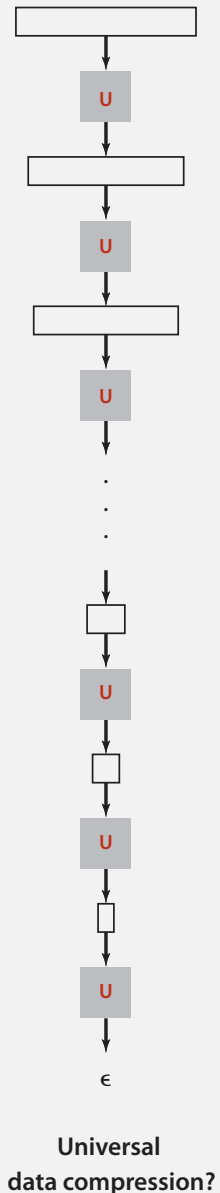
**Proposition.** No algorithm can compress every bitstring.

**Pf 1.** [by contradiction]

- Suppose you have a universal data compression algorithm  $U$  that can compress every bitstream.
- Given bitstring  $B_0$ , compress it to get smaller bitstring  $B_1$ .
- Compress  $B_1$  to get a smaller bitstring  $B_2$ .
- Continue until reaching bitstring of size 0.
- Implication: all bitstrings can be compressed to 0 bits!

**Pf 2.** [by counting]

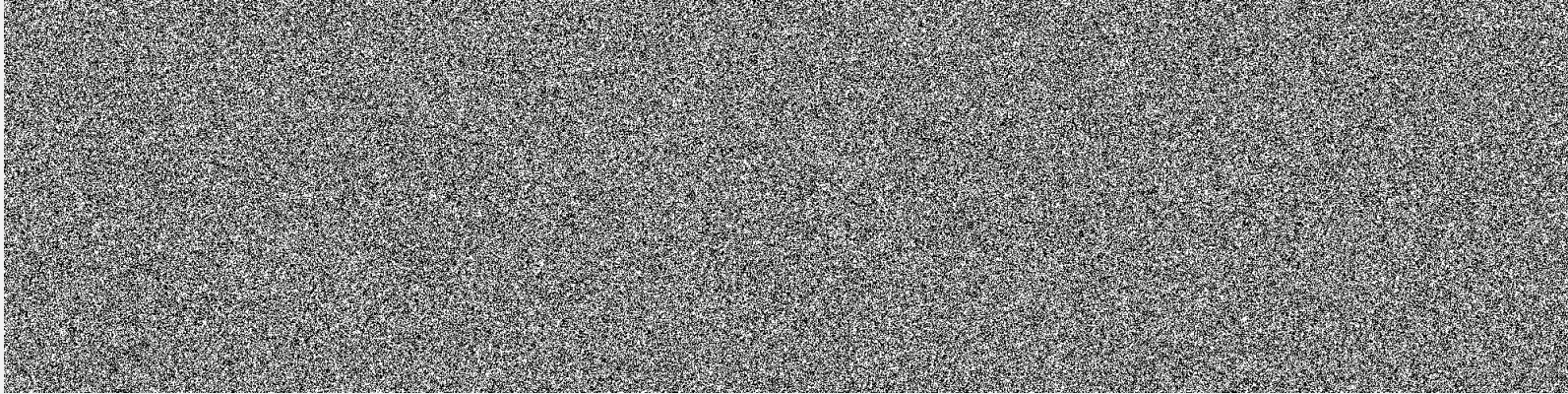
- Suppose your algorithm that can compress all 1,000-bit strings.
- $2^{1000}$  possible bitstrings with 1,000 bits.
- Only  $1 + 2 + 4 + \dots + 2^{998} + 2^{999}$  can be encoded with  $\leq 999$  bits.
- Similarly, only 1 in  $2^{499}$  bitstrings can be encoded with  $\leq 500$  bits!



# Undecidability

---

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



1000000 bits

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

```
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();
    }
}
```

# Rdenudcany in Enlgsih Inagugae

---

Q. How mcuh rdenudcany is in the Enlgsih Inagugae?

*“ ... randomising letters in the middle of words [has] little or no effect on the ability of skilled readers to understand the text. This is easy to demntrasote. In a pubiltacion of New Scnieitst you could ramdinose all the letetrs, keipeng the first two and last two the same, and reibadailty would hadrly be aftcfeed. My ansaylis did not come to much beucase the thoery at the time was for shape and senqeuce retigcionon. Saberi's work sugsegts we may have some pofrweul palrlael prsooscers at work. The resaon for this is suerly that idnetiyfing coentnt by paarllel prseocsing speeds up regnicoiton. We only need the first and last two letetrs to spot chganes in menieng. ” — Graham Rawlinson*

A. Quite a bit.



<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*



<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*



# Run-length encoding

---

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

000000000000000011111110000000111111111111

← 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.

1111011101111011 ← 16 bits (instead of 40)  
15      7      7      11

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, ...

# Run-length encoding: Java implementation

---

```
public class RunLength
{
    private final static int R    = 256;
    private final static int lgR  = 8;

    public static void compress()
    { /* see textbook */ }

    public static void expand()
    {
        boolean bit = false;
        while (!BinaryStdIn.isEmpty())
        {
            int run = BinaryStdIn.readInt(lgR);
            for (int i = 0; i < run; i++)
                BinaryStdOut.write(bit);
            bit = !bit;
        }
        BinaryStdOut.close();
    }
}
```

← maximum run-length count

← number of bits per count

← read 8-bit count from standard input

← write 1 bit to standard output

← pad 0s for byte alignment





<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*

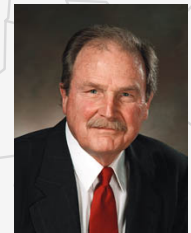


<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ ***Huffman compression***
- ▶ *LZW compression*



David Huffman

# Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: ••• — — — •••

Issue. Ambiguity.

SOS ?

V7 ?

IAMIE ?

EEWNI ?

In practice. Use a medium gap to separate codewords.

codeword for S is a prefix of codeword for V

Letters		Numbers	
A	•—	1	• — — — —
B	—•••	2	•• — — —
C	—•—•	3	••• — —
D	—••	4	•••• —
E	•	5	•••••
F	••—•	6	—••••
G	— —•	7	— —•••
H	••••	8	— — —••
I	••	9	— — — —•
J	• — — —	0	— — — — —
K	—• —		
L	• —••		
M	— —		
N	—•		
O	— — —		
P	• — —•		
Q	— —• —		
R	• —•		
S	•••		
T	—		
U	•• —		
V	••• —		
W	• — —		
X	—•• —		
Y	—• — —		
Z	— —••		

# Variable-length codes

---

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.

## Codeword table

<i>key</i>	<i>value</i>
!	101
A	0
B	1111
C	110
D	100
R	1110

## Compressed bitstring

0111111100110010001111111100101 ← 30 bits  
A B RA CA DA B RA !

## Codeword table

<i>key</i>	<i>value</i>
!	101
A	11
B	00
C	010
D	100
R	011

## Compressed bitstring

11000111101011100110001111101 ← 29 bits  
A B R A C A D A B R A !

# Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.

**Codeword table**

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

**Trie representation**

**Compressed bitstring**

0111111100110010001111111100101 ← 30 bits

A B RA CA DA B RA !

**Codeword table**

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

**Trie representation**

**Compressed bitstring**

11000111101011100110001111101 ← 29 bits

A B RA CA DA B RA !



# Prefix-free codes: compression and expansion

## Compression.

- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key-value pairs.

## Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.

**Codeword table**

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

**Trie representation**

**Compressed bitstring**

0111111100110010001111111100101 ← 30 bits

A B RA CA DA B RA !

**Codeword table**

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

**Trie representation**

**Compressed bitstring**

11000111101011100110001111101 ← 29 bits

A B RA CA DA B RA !

# Huffman trie node data type

---

```
private static class Node implements Comparable<Node>
{
    private final char ch;    // used only for leaf nodes
    private final int freq;   // used only for compress
    private final Node left, right;
```

```
    public Node(char ch, int freq, Node left, Node right)
    {
        this.ch    = ch;
        this.freq  = freq;
        this.left  = left;
        this.right = right;
    }
```

```
    public boolean isLeaf()
    { return left == null && right == null; }
```

```
    public int compareTo(Node that)
    { return this.freq - that.freq; }
```

```
}
```

← initializing constructor

← is Node a leaf?

← compare Nodes by frequency  
(stay tuned)

## Prefix-free codes: expansion

---

```
public void expand()
{
    Node root = readTrie();
    int N = BinaryStdIn.readInt();

    for (int i = 0; i < N; i++)
    {
        Node x = root;
        while (!x.isLeaf())
        {
            if (!BinaryStdIn.readBoolean())
                x = x.left;
            else
                x = x.right;
        }
        BinaryStdOut.write(x.ch, 8);
    }
    BinaryStdOut.close();
}
```

← read in encoding trie  
← read in number of chars

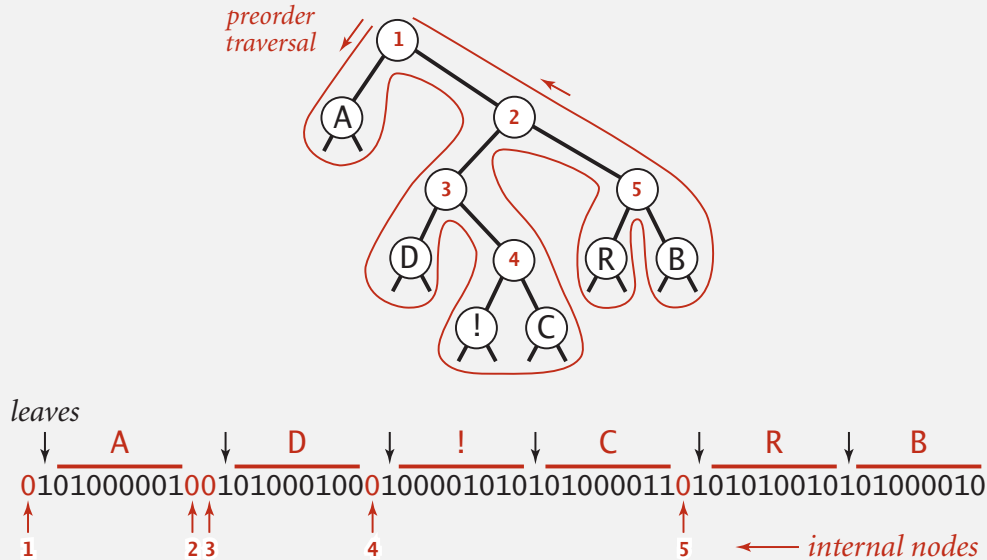
← expand codeword for  $i^{\text{th}}$  char

**Running time.** Linear in input size  $N$ .

# Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.



Using preorder traversal to encode a trie as a bitstream

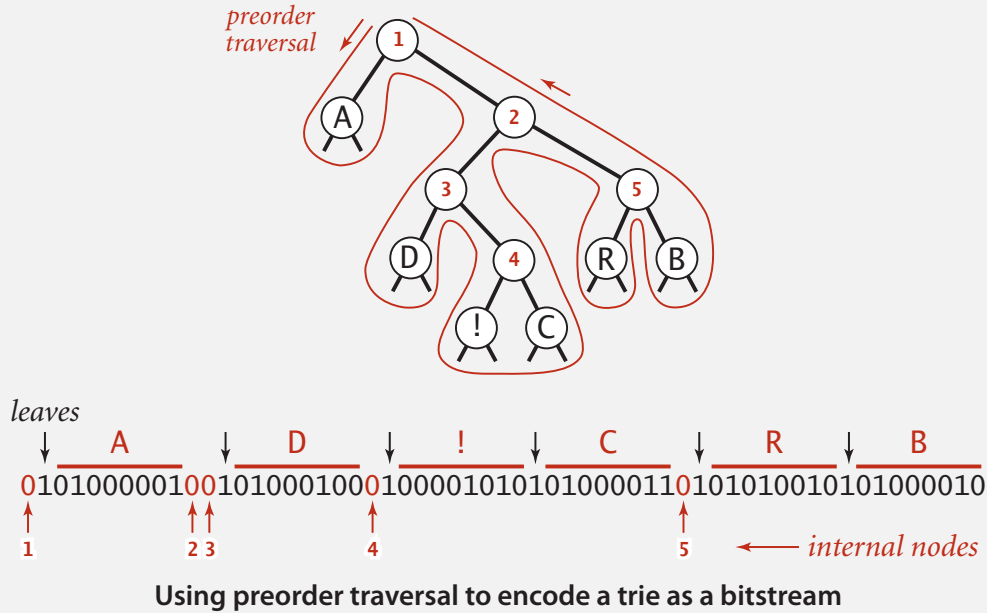
```
private static void writeTrie(Node x)
{
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

**Note.** If message is long, overhead of transmitting trie is small.

# Prefix-free codes: how to transmit

Q. How to read in the trie?

A. Reconstruct from preorder traversal of trie.



```
private static Node readTrie()
{
    if (BinaryStdIn.readBoolean())
    {
        char c = BinaryStdIn.readChar(8);
        return new Node(c, 0, null, null);
    }
    Node x = readTrie();
    Node y = readTrie();
    return new Node('\0', 0, x, y);
}
```

used only for leaf nodes

# Shannon-Fano codes

---

Q. How to find best prefix-free code?

Shannon-Fano algorithm:

- Partition symbols  $S$  into two subsets  $S_0$  and  $S_1$  of (roughly) equal freq.
- Codewords for symbols in  $S_0$  start with 0; for symbols in  $S_1$  start with 1.
- Recur in  $S_0$  and  $S_1$ .

char	freq	encoding
A	5	0...
C	1	0...

$S_0$  = codewords starting with 0

char	freq	encoding
B	2	1...
D	1	1...
R	2	1...
!	1	1...

$S_1$  = codewords starting with 1

Problem 1. How to divide up symbols?

Problem 2. Not optimal!

# Huffman algorithm demo

---

- Count frequency for each character in input.

<u>char</u>	<u>freq</u>	<u>encoding</u>
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



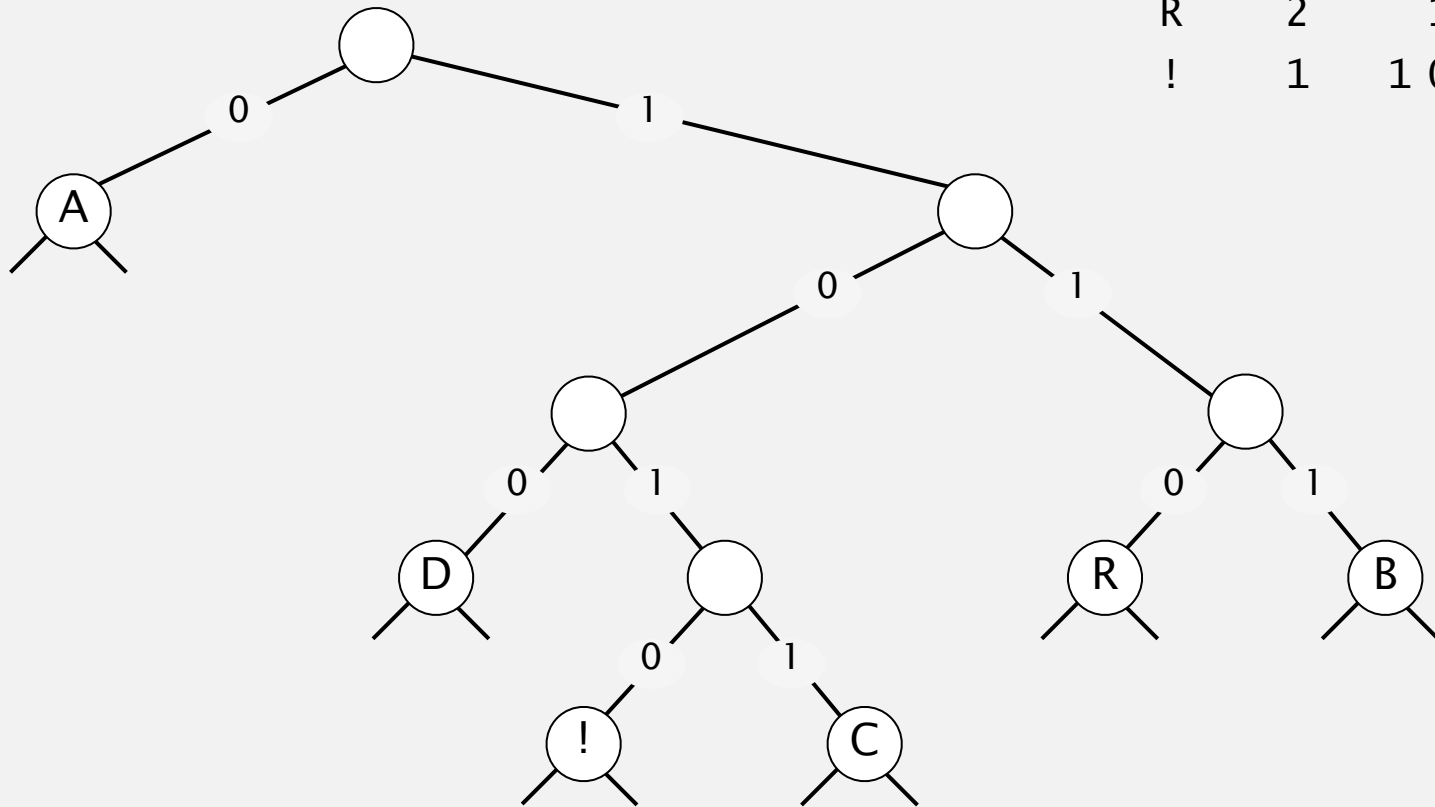
**input**

A B R A C A D A B R A !

# Huffman algorithm demo

---

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0





# Huffman codes

---

Q. How to find best prefix-free code?

## Huffman algorithm:

- Count frequency  $\text{freq}[i]$  for each char  $i$  in input.
- Start with one node corresponding to each char  $i$  (with weight  $\text{freq}[i]$ ).
- Repeat until single trie formed:
  - select two tries with min weight  $\text{freq}[i]$  and  $\text{freq}[j]$
  - merge into single trie with weight  $\text{freq}[i] + \text{freq}[j]$

## Applications:



# Constructing a Huffman encoding trie: Java implementation

---

```
private static Node buildTrie(int[] freq)
{
```

```
    MinPQ<Node> pq = new MinPQ<Node>();
    for (char i = 0; i < R; i++)
        if (freq[i] > 0)
            pq.insert(new Node(i, freq[i], null, null));
```

← initialize PQ with  
singleton tries

```
    while (pq.size() > 1)
    {
        Node x = pq.delMin();
        Node y = pq.delMin();
        Node parent = new Node('\0', x.freq + y.freq, x, y);
        pq.insert(parent);
    }
```

← merge two  
smallest tries

```
    return pq.delMin();
```

↑  
not used for  
internal nodes

↑  
total frequency

↑ ↑  
two subtrees

```
}
```

# Huffman encoding summary

---

**Proposition.** [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

**Pf.** See textbook.

↑  
no prefix-free code uses fewer bits

## Implementation.

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

**Running time.** Using a binary heap  $\Rightarrow N + R \log R$ .

↑                    ↑  
input                alphabet  
size                    size

**Q.** Can we do better? [stay tuned]



<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

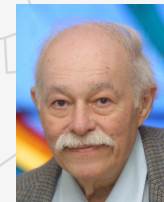
---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*

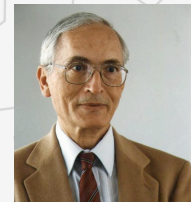
## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ ***LZW compression***



Abraham Lempel



Jacob Ziv

# Statistical methods

---

**Static model.** Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

**Dynamic model.** Generate model based on text.

- Preliminary pass needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

**Adaptive model.** Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Decoding must start from beginning.
- Ex: LZW.

# LZW compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D	A B		R A		B R		A B R			A
<i>value</i>	41	42	52	41	43	41	44	81		83		82		88			41 80

LZW compression for A B R A C A D A B R A B R A B R A

key	value
:	:
A	41
B	42
C	43
D	44
:	:

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

codeword table

# Lempel-Ziv-Welch compression

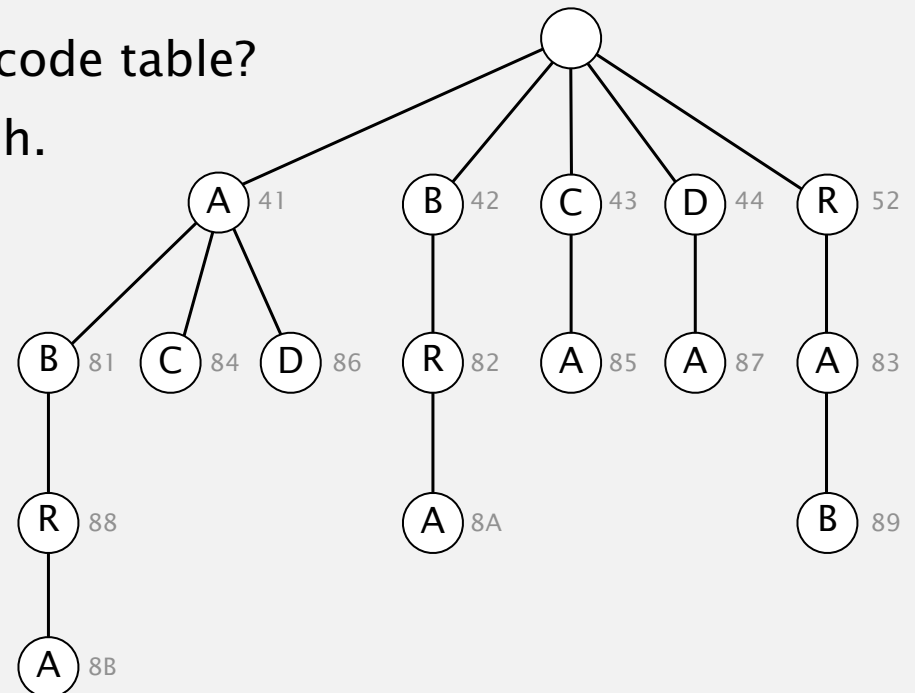
## LZW compression.

- Create ST associating  $W$ -bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string  $s$  in ST that is a prefix of unscanned part of input.
- Write the  $W$ -bit codeword associated with  $s$ .
- Add  $s + c$  to ST, where  $c$  is next char in the input.

longest prefix match

Q. How to represent LZW compression code table?

A. A trie to support longest prefix match.





# LZW compression: Java implementation

---

```
public static void compress()
{
    String input = BinaryStdIn.readString();

    TST<Integer> st = new TST<Integer>();
    for (int i = 0; i < R; i++)
        st.put("" + (char) i, i);
    int code = R+1;

    while (input.length() > 0)
    {
        String s = st.longestPrefixOf(input);
        BinaryStdOut.write(st.get(s), W);
        int t = s.length();
        if (t < input.length() && code < L)
            st.put(input.substring(0, t+1), code++);
        input = input.substring(t);

        BinaryStdOut.write(R, W);
        BinaryStdOut.close();
    }
}
```

← read in input as a string

← codewords for single-char, radix R keys

← find longest prefix match s

← write W-bit codeword for s

← add new codeword

← scan past s in input

← write "stop" codeword and close input stream

# LZW expansion example

---

*value* 41 42 52 41 43 41 44 81 83 82 88 41 80  
*output* A B R A C A D A B R A B R A B R A

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value
:	:
41	A
42	B
43	C
44	D
:	:

key	value
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD

key	value
87	DA
88	ABR
89	RAB
8A	BRA
8B	ABRA

codeword table

# LZW expansion

---

## LZW expansion.

- Create ST associating string values with  $W$ -bit keys.
- Initialize ST to contain single-char values.
- Read a  $W$ -bit key.
- Find associated string value in ST and write it out.
- Update ST.

Q. How to represent LZW expansion code table?

A. An array of size  $2^W$ .

key	value
:	:
65	A
66	B
67	C
68	D
:	:
129	AB
130	BR
131	RA
132	AC
133	CA
134	AD
135	DA
136	ABR
137	RAB
138	BRA
139	ABRA
:	:

# LZW example: tricky case

---

<i>input</i>	A	B	A	B	A	B	A
<i>matches</i>	A	B	A B		A B A		
<i>value</i>	41	42	81		83		80

LZW compression for ABABABA

key	value
⋮	⋮
A	41
B	42
C	43
D	44
⋮	⋮

key	value
AB	81
BA	82
ABA	83

codeword table

# LZW example: tricky case

---

<i>value</i>	41	42	81	83	80
<i>output</i>	A	B	A B	A B A	

need to know which  
key has value 83  
before it is in ST!

LZW expansion for 41 42 81 83 80

key	value
⋮	⋮
41	A
42	B
43	C
44	D
⋮	⋮

key	value
81	AB
82	BA
83	ABA

codeword table

# LZW implementation details

---

## How big to make ST?

- How long is message?
- Whole message similar model?
- [many other variations]

## What to do when ST fills up?

- Throw away and start over. [GIF]
- Throw away when not effective. [Unix compress]
- [many other variations]

## Why not put longer substrings in ST?

- [many variations have been developed]

# LZW in the real world

## Lempel-Ziv and friends.

- LZ77. LZ77 not patented ⇒ widely used in open source
- LZ78. LZW patent #4,558,302 expired in U.S. on June 20, 2003
- LZW.
- Deflate / zlib = LZ77 variant + Huffman.

**United States Patent** [19] [11] **Patent Number:** **4,558,302**  
**Welch** [45] **Date of Patent:** **Dec. 10, 1985**

[54] **HIGH SPEED DATA COMPRESSION AND DECOMPRESSION APPARATUS AND METHOD**

[75] Inventor: Terry A. Welch, Concord, Mass.

[73] Assignee: Sperry Corporation, New York, N.Y.

[21] Appl. No.: 505,638

[22] Filed: Jun. 20, 1983

[51] Int. Cl.<sup>4</sup> ..... G06F 5/00

[52] U.S. Cl. .... 340/347 DD; 235/310

[58] Field of Search ..... 340/347 DD; 235/310, 235/311; 364/200, 900

[56] **References Cited**

**U.S. PATENT DOCUMENTS**

4,464,650 8/1984 Eastman ..... 340/347 DD

**OTHER PUBLICATIONS**

Ziv, "IEEE Transactions on Information Theory", IT-24-5, Sep. 1977, pp. 530-537.

Ziv, "IEEE Transactions on Information Theory", IT-23-3, May 1977, pp. 337-343.

*Primary Examiner*—Charles D. Miller  
*Attorney, Agent, or Firm*—Howard P. Terry; Albert B. Cooper

[57] **ABSTRACT**

A data compressor compresses an input stream of data character signals by storing in a string table strings of data character signals encountered in the input stream. The compressor searches the input stream to determine

the longest match to a stored string. Each stored string comprises a prefix string and an extension character where the extension character is the last character in the string and the prefix string comprises all but the extension character. Each string has a code signal associated therewith and a string is stored in the string table by, at least implicitly, storing the code signal for the string, the code signal for the string prefix and the extension character. When the longest match between the input data character stream and the stored strings is determined, the code signal for the longest match is transmitted as the compressed code signal for the encountered string of characters and an extension string is stored in the string table. The prefix of the extended string is the longest match and the extension character of the extended string is the next input data character signal following the longest match. Searching through the string table and entering extended strings therein is effected by a limited search hashing procedure. Decompression is effected by a decompressor that receives the compressed code signals and generates a string table similar to that constructed by the compressor to effect lookup of received code signals so as to recover the data character signals comprising a stored string. The decompressor string table is updated by storing a string having a prefix in accordance with a prior received code signal and an extension character in accordance with the first character of the currently recovered string.

**181 Claims, 9 Drawing Figures**



# LZW in the real world

---

## Lempel-Ziv and friends.

- LZ77.
- LZ78.
- LZW.
- Deflate / zlib = LZ77 variant + Huffman.



Unix compress, GIF, TIFF, V.42bis modem: LZW.

zip, 7zip, gzip, jar, png, pdf: deflate / zlib.

iPhone, Sony Playstation 3, Apache HTTP server: deflate / zlib.





# Lossless data compression benchmarks

---

year	scheme	bits / char
1967	ASCII	7.00
1950	Huffman	4.70
1977	LZ77	3.94
1984	LZMW	3.32
1987	LZH	3.30
1987	move-to-front	3.24
1987	LZB	3.18
1987	gzip	2.71
1988	PPMC	2.48
1994	SAKDC	2.47
1994	PPM	2.34
1995	Burrows-Wheeler	2.29
1997	BOA	1.99
1999	RK	1.89

← next programming assignment

data compression using Calgary corpus

# Data compression summary

---

## Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

## Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3, ...
- FFT, wavelets, fractals, ...

**Theoretical limits on compression.** Shannon entropy:  $H(X) = - \sum_i^n p(x_i) \lg p(x_i)$

**Practical compression.** Use extra knowledge whenever possible.



<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*



<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*