



<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

---

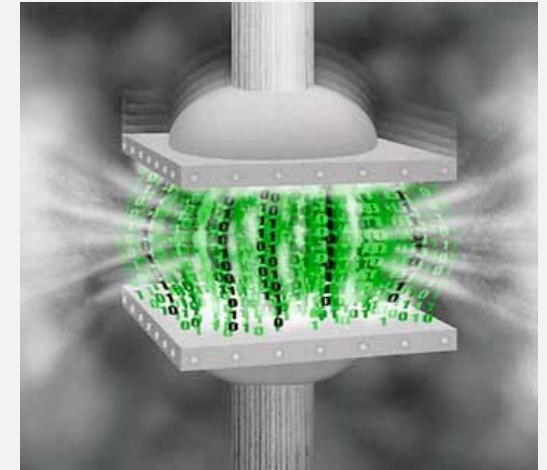
- ▶ *introduction*
- ▶ *run-length coding*
- ▶ *Huffman compression*
- ▶ *LZW compression*

# 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, ZFS, HFS+, ReFS, GFS.



## Multimedia.

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



## Communication.

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



Databases. Google, Facebook, NSA, ....



# Lossless compression and expansion

---

**Message.** Bitstream  $B$  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:

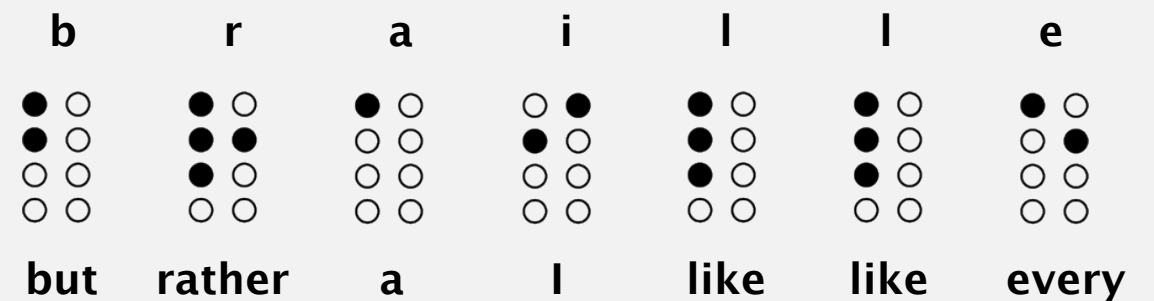
- Number systems.
- Natural languages.
- Mathematical notation.



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

has played a central role in communications technology,

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



and is part of modern life.

- JPEG.
- MP3.
- MPEG.



Q. What role will it play in the future?

# Data representation: genomic code

---

**Genome.** String over the alphabet { A, T, C, 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
'T'	54	01010100
'C'	43	01000011
'G'	47	01000111

**Two-bit encoding.**

- 2 bits per char.
- $2n$  bits (25% compression ratio).

char	binary
'A'	00
'T'	01
'C'	10
'G'	11

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

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



# 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
```

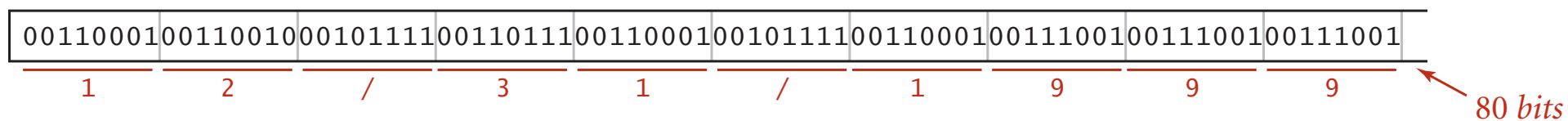


# Writing binary data

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

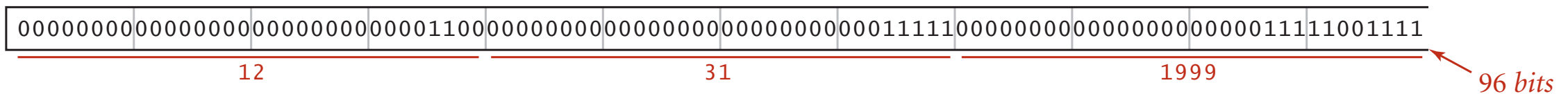
## A character stream (StdOut)

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



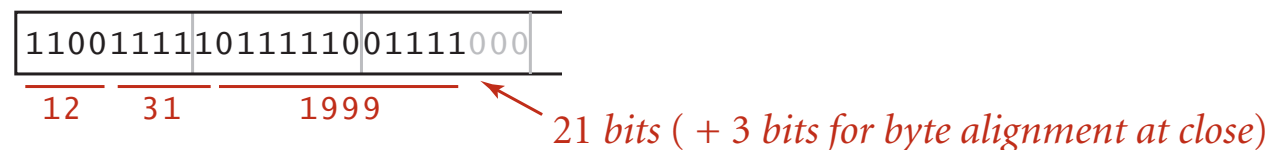
## Three ints (BinaryStdOut)

```
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

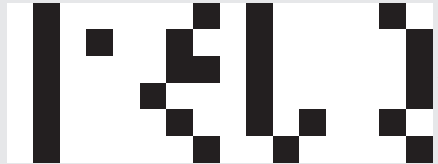
```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96 bits
```

## 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

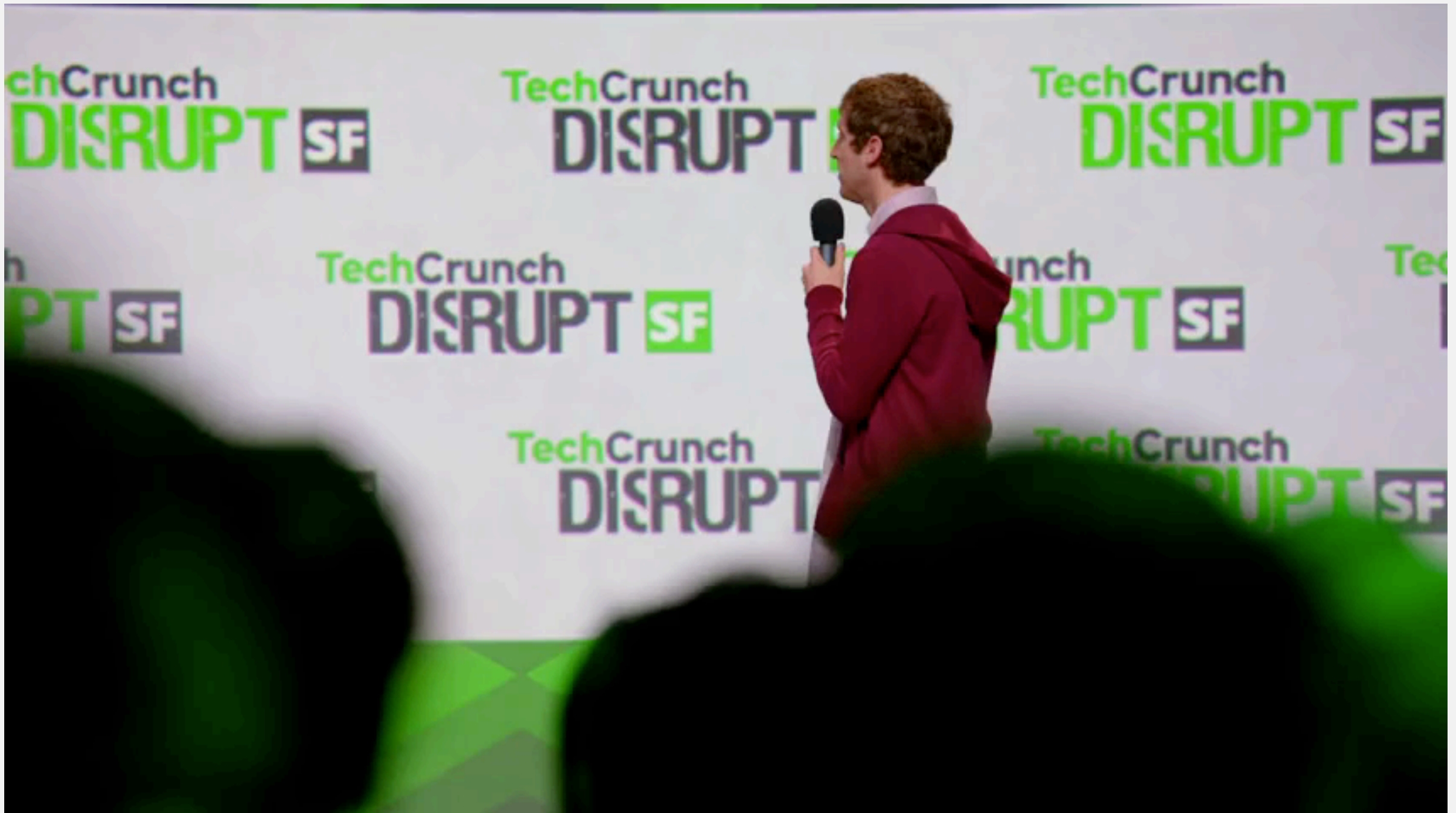
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	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

---

[Pied Piper](#). Claims 3.8:1 lossless compression of arbitrary data.



# Universal data compression

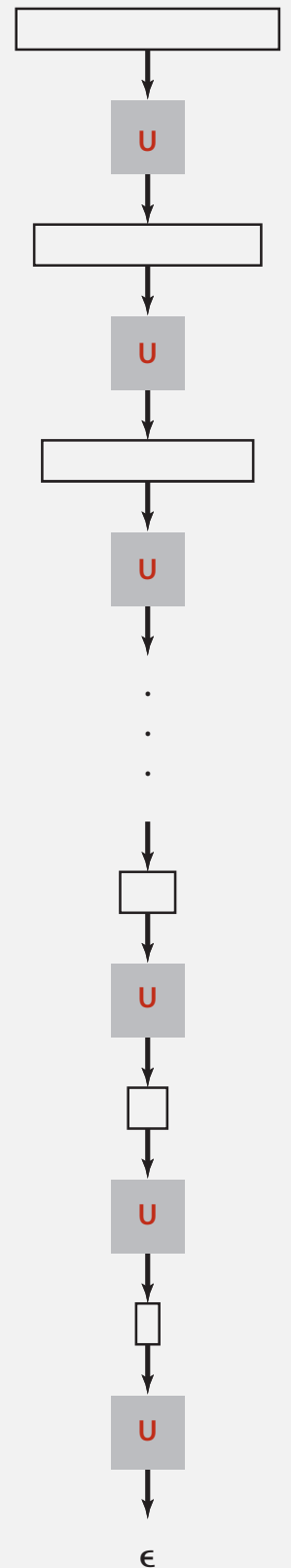
**Proposition.** No algorithm can compress every bitstring.

**Pf 1.** [by contradiction]



**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!

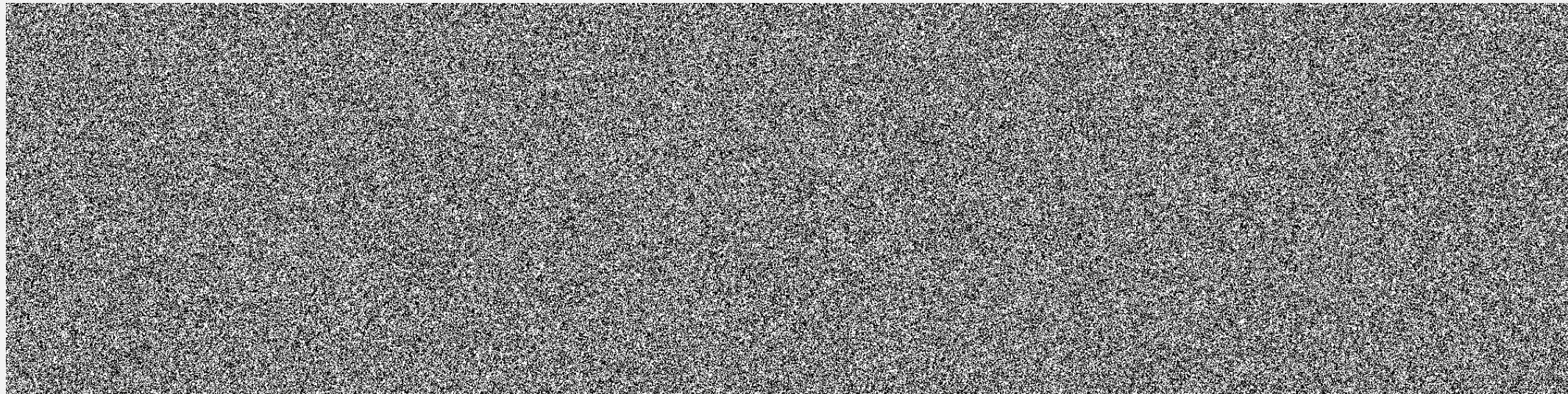




# Optimal data compression is undecidable

---

```
% 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();
    }
}
```

# Redundancy in English Language

---

Q. How much redundancy in the English language?

A. Quite a bit.

*“ ... 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 demonstrate. In a publication of New Scientist you could randomise all the letters, keeping the first two and last two the same, and readability would hardly be affected. My analysis did not come to much because the theory at the time was for shape and sentence recognition. Saberi's work suggests we may have some powerful parallel processors at work. The reason for this is surely that identifying content by parallel processing speeds up recognition. We only need the first and last two letters to spot changes in meaning. ” — Graham Rawlinson*

The goal of data compression is to identify redundancy and exploit it.





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

## 5.5 DATA COMPRESSION

---

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



# Run-length encoding (RLE)

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

0000000000000000001111111000000001111111111111111

run of length 7

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. Typically 8 bits (but 4 on this slide for brevity).

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

A. Intersperse runs of length 0.

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

# Run-length encoding: Java implementation

---

```
public class RunLength
{
```

```
    private static final int R    = 256;
    private static final 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

← initial runs are 0

← read 8-bit count from standard input

← write run of 0s or 1s to standard output

← flip bit (for next run)

← pad 0s for byte alignment

## Data compression: quiz 1

---

What is the best compression ratio achievable from run-length encoding when using 8-bit counts?

A.  $1 / 256$

B.  $1 / 16$

C.  $8 / 255$

D.  $24 / 510 = 4 / 85$

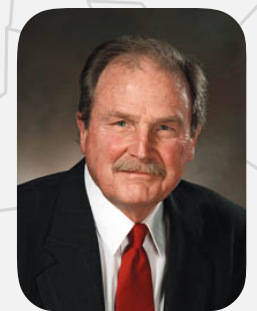


<http://algs4.cs.princeton.edu>

## 5.5 DATA COMPRESSION

---

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



David Huffman

# Variable-length codes

---

**Key idea.** Use different number of bits to encode different characters.

**Ex.** Morse code: ● ● ● — — — ● ● ●

**Issue.** Ambiguity.

S O S ?

V Z E ?

E E J I E ?

E E W N I ?

A ● ■	N ■ ●
B ■ ● ● ●	O ■ ■ ■
C ■ ● ■ ●	P ● ■ ■ ●
D ■ ● ●	Q ■ ■ ● ■
E ●	R ● ■ ●
F ● ● ■ ●	S ● ● ●
G ■ ■ ●	T ■
H ● ● ● ●	U ● ● ■
I ● ●	V ● ● ● ■
J ● ■ ■ ■	W ● ■ ■
K ■ ● ■	X ■ ● ● ■
L ● ■ ● ●	Y ■ ● ■ ■
M ■ ■	Z ■ ■ ● ●

codeword for S  
is a prefix of  
codeword for V

**In practice.** Use a short gap to separate characters.

# 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” character 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

011111110011001000111111100101 ← 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?

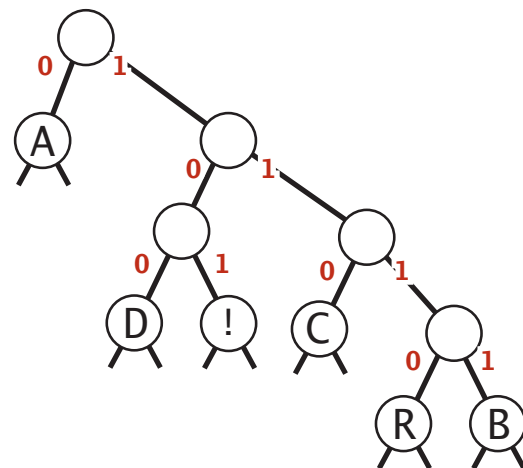
A. A binary trie!

- Characters 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



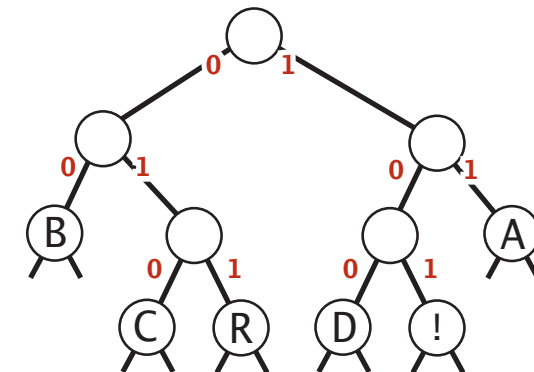
Compressed bitstring

011111110011001000111111100101 ← 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 R A C A D A B R A !



# Prefix-free codes: expansion

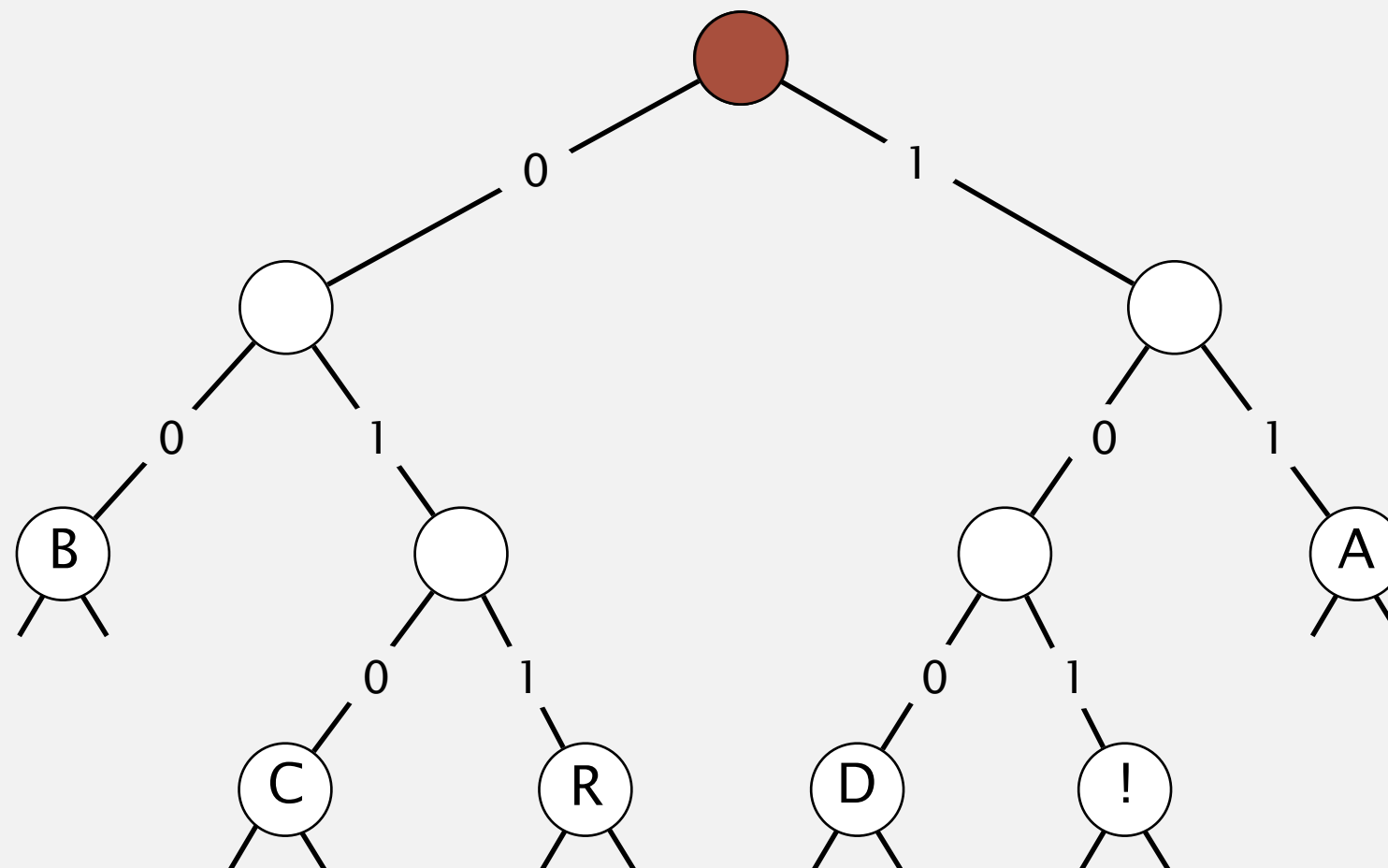
## Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, write character; return to root node; repeat.



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

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

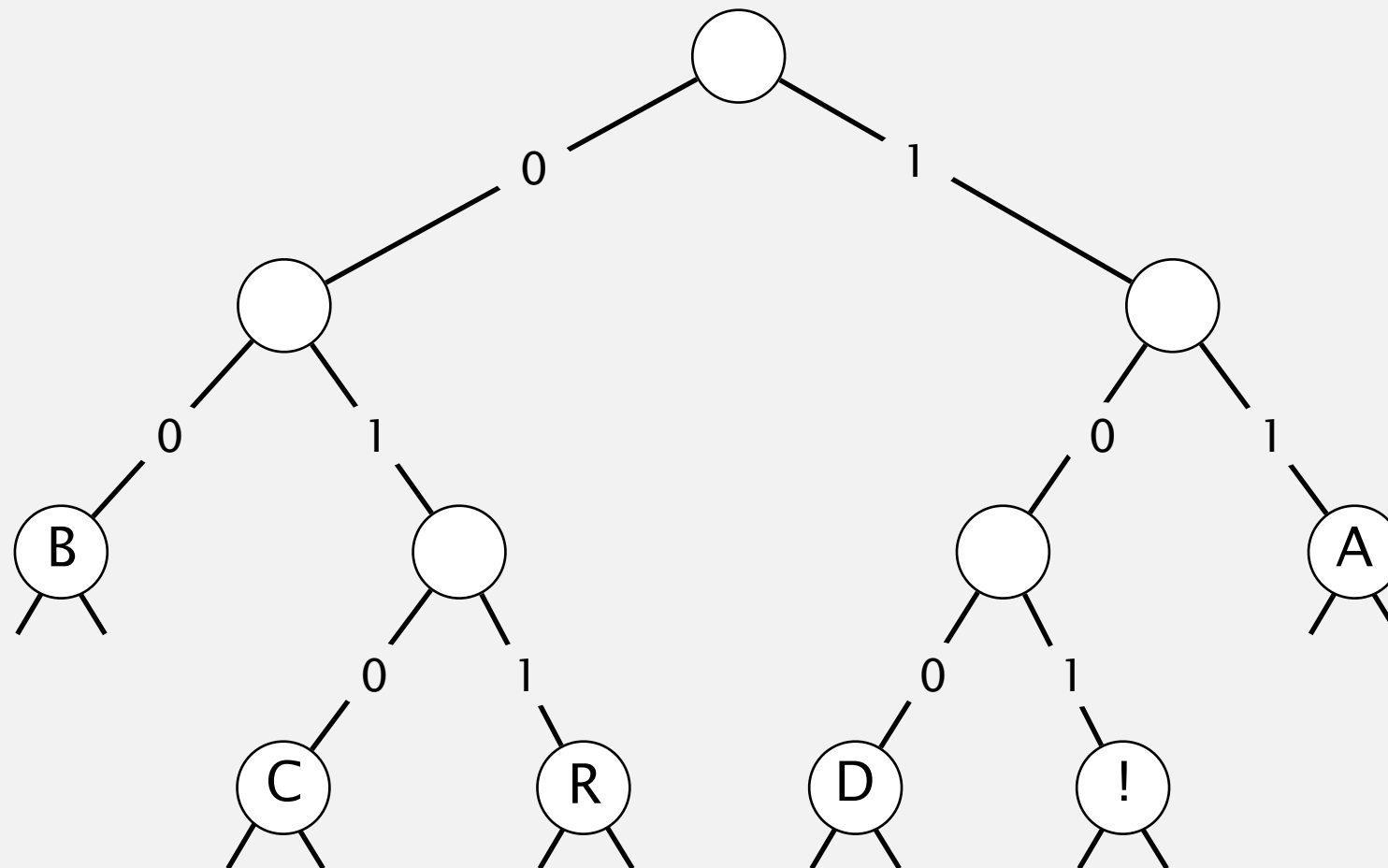


# Prefix-free codes: compression

---

## Compression.

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



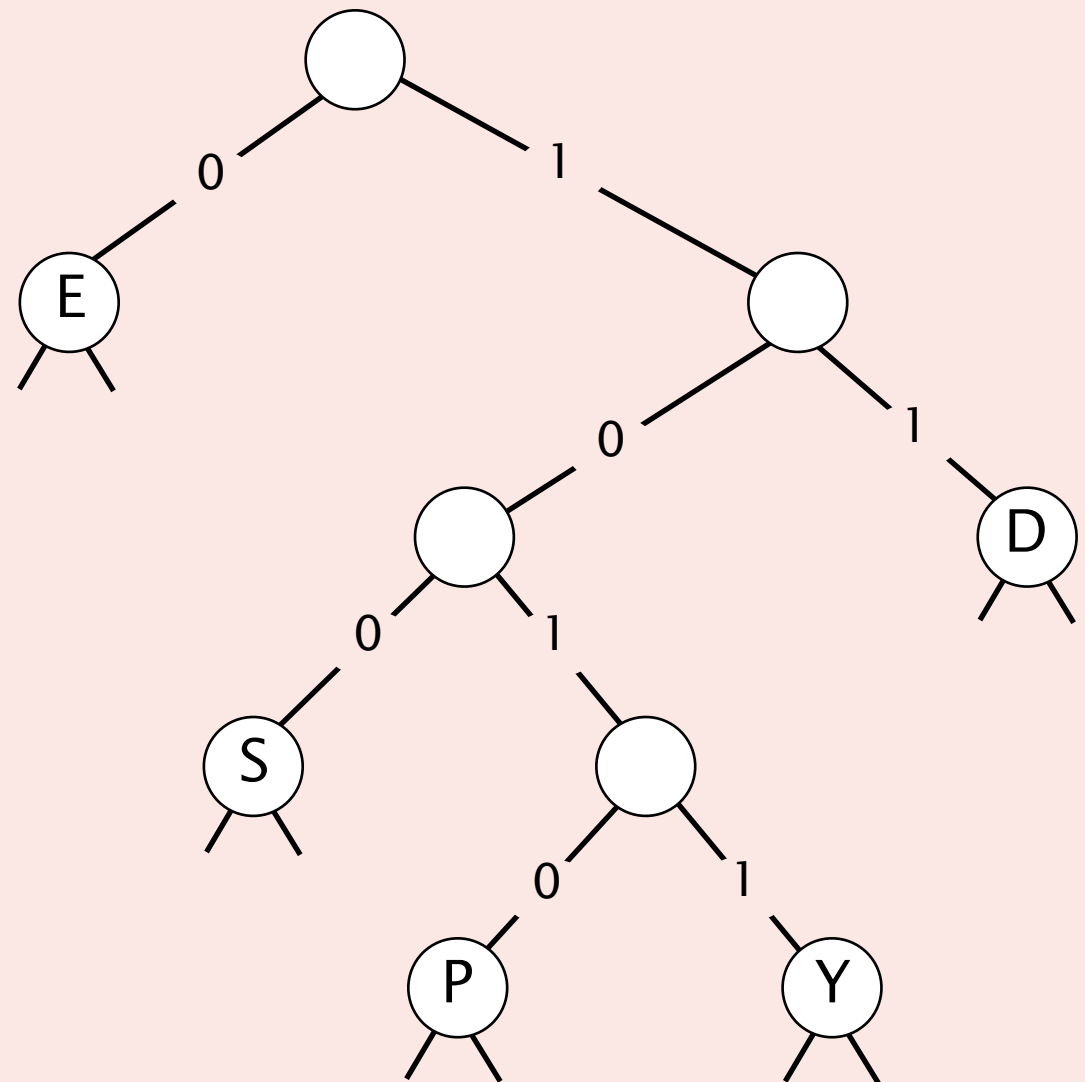
## Data compression: quiz 2

---

Consider the following trie representation of a prefix-free code.

Expand the compressed bitstring **100101000111011** ?

- A. PEED
- B. PESDEY
- C. SPED
- D. SPEEDY



# Huffman coding overview

---

**Static model.** Use the same prefix-free code for all messages.

**Dynamic model.** Use a custom prefix-free code for each message.

## Compression.

- Read message.
- Build **best prefix-free code** for message. How? [ahead]
- Write prefix-free code (as a trie).
- Compress message using prefix-free code.

## Expansion.

- Read prefix-free code (as a trie).
- Read compressed message and expand using trie.

# 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 by 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;
}
```

← initializing constructor

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

← is Node a leaf?

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

← 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 encoding trie

← read number of chars

← for each encoded character i

← follow path from root to leaf to determine character

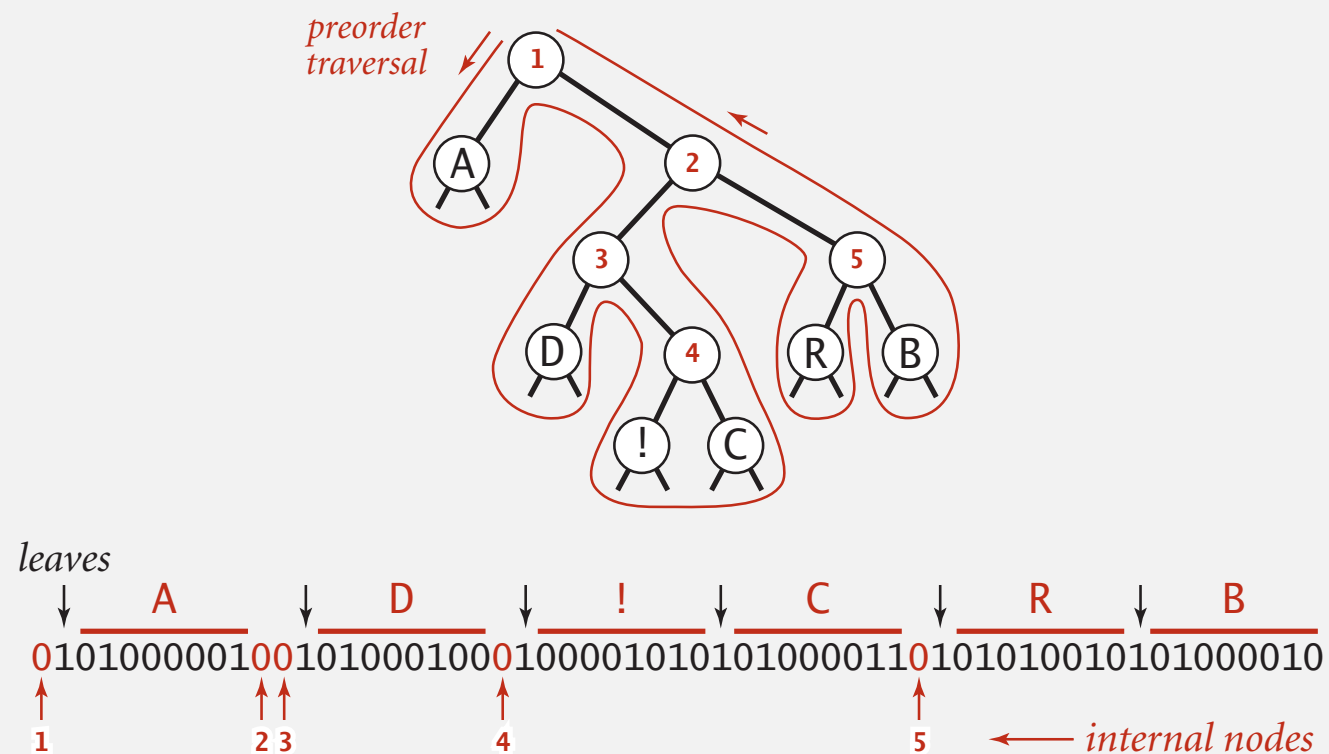
← write character (8 bits)

**Running time.** Linear in input size (number of bits).

# 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);
}
```

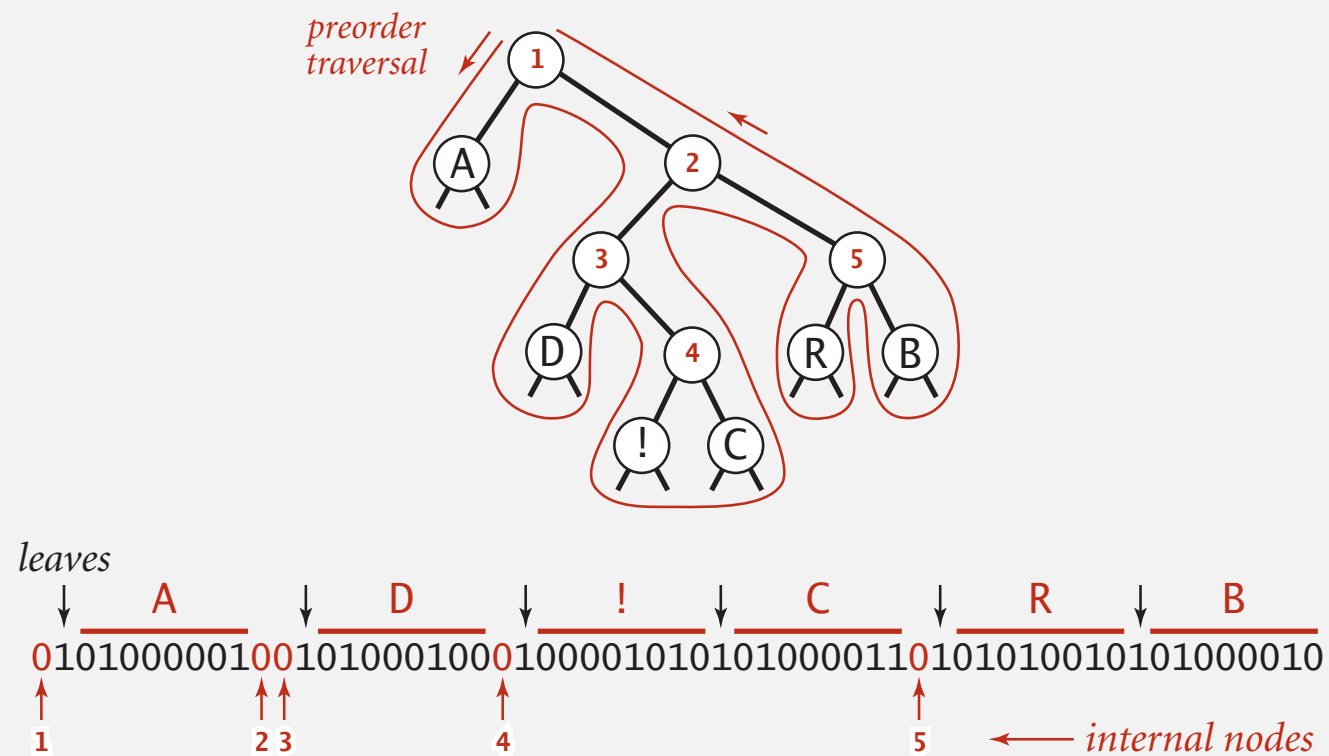
**Note.** If message is long, overhead of transmitting trie is small.



# Prefix-free codes: how to transmit

Q. How to read in the trie?

A. Reconstruct from preorder traversal of trie.



Using preorder traversal to encode a trie as a bitstream

```
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);
}
```

arbitrary value  
(value not used with internal nodes)

# 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 compression summary

---

**Proposition.** Huffman's algorithm produces an optimal prefix-free code.

**Pf.** See textbook.

no prefix-free code  
uses fewer bits

**Two-pass implementation (for compression).**

- Pass 1: tabulate character frequencies; build trie.
- Pass 2: encode file by traversing trie (or symbol table).

**Running time (for compression).** Using a binary heap  $\Rightarrow n + R \log R$ .

**Running time (for expansion).** Using a binary trie  $\Rightarrow n$ .

input  
size

alphabet  
size

**Q.** Can we do better (in terms of compression ratio)? [stay tuned]

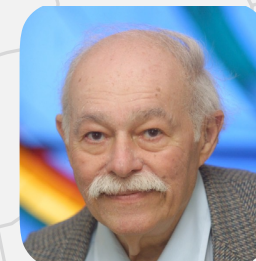
## 5.5 DATA COMPRESSION

---

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



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



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 demo

<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	key	value	key	value
:	:	AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85	ABRA	8B
:	:	AD	86		

codeword table



# LZW expansion demo

---

*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	key	value	key	value
:	:	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC	8A	BRA
44	D	85	CA	8B	ABRA
:	:	86	AD		

codeword table

## Data compression: quiz 3

---

What is the LZW compression of ABABABA ?

A. 41 42 41 42 41 42 80

B. 41 42 41 81 81

C. 41 42 81 81 41

D. 41 42 81 83 80

# LZW tricky case: compression

---

<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	key	value
:	:	AB	81
A	41	BA	82
B	42	ABA	83
C	43		
D	44		
:	:		

codeword table

# LZW tricky case: expansion

*value* 41 42 81 83 80  
*output* A B A B A B A

LZW expansion for 41 42 81 83 80

need to know code for 83 before it is in codeword table!

we can deduce that the code for 83 is ABx for some character x

now, we have deduced x!

key	value
⋮	⋮
41	A
42	B
43	C
44	D
⋮	⋮

key	value
81	AB
82	BA
83	ABA

codeword table

# 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. ← previously under patent

zip, 7zip, gzip, jar, png, pdf: deflate / zlib. ← not patented  
(widely used in open source)

iPhone, Wii, Apache HTTP server: deflate / zlib.



# Lossless data compression benchmarks

---

year	scheme	bits / char
1967	ASCII	7
1950	Huffman	4.7
1977	LZ77	3.94
1984	LZMW	3.32
1987	LZH	3.3
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
<b>1995</b>	<b>Burrows-Wheeler</b>	<b>2.29</b>
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/DCT, wavelets, fractals, ...

$$X_k = \sum_{i=0}^{n-1} x_i \cos \left[ \frac{\pi}{n} \left( i + \frac{1}{2} \right) k \right]$$

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

Practical compression. Exploit extra knowledge whenever possible.

