5.5 Data Compression

- basics
- run-length encoding
- 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, …

> *" All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value. "* — *Carl Sagan*

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

Applications

Generic file compression.

- Files: GZIP, BZIP, BOA.
- Archivers: PKZIP.
- File systems: NTFS.

Multimedia.

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

Communication.

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

Databases. Google.

Lossless compression and expansion

Message. Binary data B we want to compress.

Compress. Generates a "compressed" representation $C(B)$. (uses fewer bits (you hope))

Expand. Reconstructs original bitstream B.

Basic model for data compression

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.

has played a central role in communications technology,

- Braille.
- Morse code.
- Telephone system.

and is part of modern life.

- MP3.
- MPEG.

Q. What role will it play in the future?

- binary I/O
- genomic encoding
- run-length encoding
- Huffman compression
- LZW compression

Reading and writing binary data

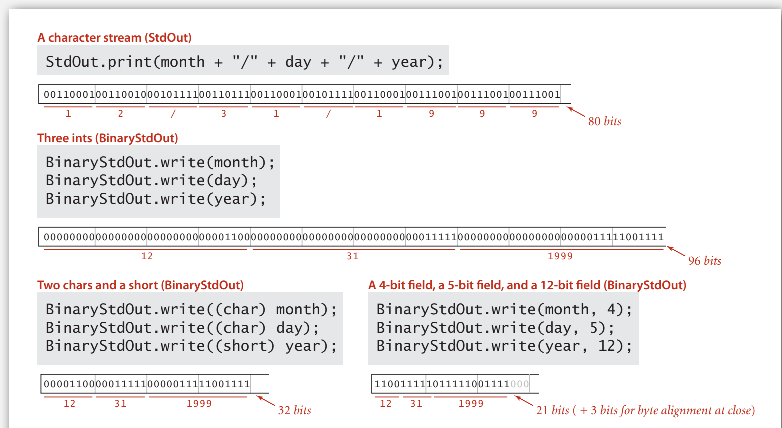
Binary standard input and standard output. Libraries to read and write bits from standard input and to standard output.

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. Different ways to represent 12/31/1999.



Four ways to put a date onto standard output

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
96 bits
```

Bitstream represented as pixels in a Picture

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


```
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

- binary I/O
- **limitations**
- genomic encoding
- run-length encoding
- Huffman compression
- LZW compression

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

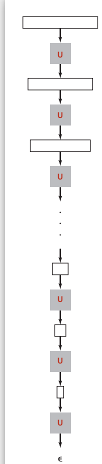
Proposition. No algorithm can compress every bitstring.

Pf 1. [by contradiction]

- Suppose you have a universal data compression algorithm U that can compress every bitstream.
- Given bintstring 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 with 0 bits!

Pf 2. [by counting]

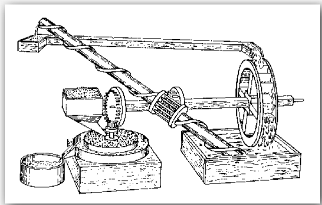
- Suppose your algorithm that can compress all 1,000-bit strings.
- 2^{1000} possible bitstrings with 1000 bits.
- Only $1 + 2 + 4 + … + 2^{998} + 2^{999}$ can be encoded with ≤ 999 bits.
- Similarly, only 1 in 2^{499} bitstrings can be encoded with ≤ 500 bits!



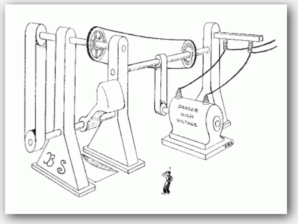
Universal data compression?

Perpetual motion machines

Universal data compression is the analog of perpetual motion.



Closed-cycle mill by Robert Fludd, 1618

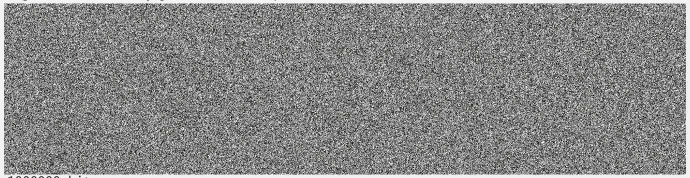


Gravity engine by Bob Schadewald

Reference: Museum of Unworkable Devices by Donald E. Simanek
http://www.lhup.edu/~dsimanek/museum/unwork.htm

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 much redundancy is in the English language?

> *" ... 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 denmtrasote. 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 meniang."* — *Graham Rawlinson*

A. Quite a bit.

- genomic encoding
- run-length encoding
- Huffman compression
- LZW compression

Genomic code

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

Goal. Encode an N-character genome: **ATAGATGCATAG...**

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 encoding.

- 2 bits per char.
- 2N bits.

char	binary
A	00
C	01
T	10
G	11

Amazing but true. Initial genomic databases in 1990s did not use such a code!

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

Genomic code

```
public class Genome {

   public static void compress() {
      Alphabet DNA = new Alphabet("ACTG");
      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));
         BinaryStdOut.write(d, 2);
      }
      BinaryStdOut.close();
   }

   public static void expand() {
      Alphabet DNA = new Alphabet("ACTG");
      int N = BinaryStdIn.readInt();
      for (int i = 0; i < N; i++) {
         char c = BinaryStdIn.readChar(2);
         BinaryStdOut.write(DNA.toChar(c));
      }
      BinaryStdOut.close();
   }
}
```

`Alphabet` data type converts between symbols { A, C, T, G } and integers 0—3.

read genomic string from stdin; write to stdout using 2-bit code

read 2-bit code from stdin; write genomic string to stdout

Genomic code: test client and sample execution

```
public static void main(String[] args)
{
    if (args[0].equals("-")) compress();
    if (args[0].equals("+")) expand();
}
```

Tiny test case (264 bits)

```
% more genomeTiny.txt
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC

java BitsDump 64 < genomeTiny.txt
0100000101010100010000010100011101000001010101000100011101000011
0100000101010100010000010100011101000011010001110100001101000001
0101010001000001010001110100001101010100010000010100011101000001
0101010001000111010101000100011101000011010101000100000101000111
01000011
264 bits

% java Genome - < genomeTiny.txt
??  ← cannot see bitstream on standard output

% java Genome - < genomeTiny.txt | java BinaryDump 64
0000000000000000000000000010000100100011001011010010001101110100
1000110110001100101110110110001101000000
104 bits

% java Genome - < genomeTiny.txt | java HexDump 8
00 00 00 21 23 2d 23 74
8d 8c bb 63 40
104 bits

% java Genome - < genomeTiny.txt | java Genome +
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC  ← compress-expand cycle produces original input
```


- genomic encoding
- **run-length encoding**
- Huffman compression
- LZW compression

Run-length encoding

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

```
0000000000000001111111000000011111111111
```

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

```
1111 0111 0111 1011   ← 16 bits (instead of 40)
 15    7    7   11
```

Q. How many bits to store the counts?
A. We'll use 8.

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;

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

   public static void expand()
   {
      boolean b = false;
      while (!BinaryStdIn.isEmpty())
      {
         char run = BinaryStdIn.readChar();   ← read 8-bit count from standard input
         for (int i = 0; i < run; i++)
            BinaryStdOut.write(b);            ← write 1 bit to standard output
         b = !b;
      }
      BinaryStdOut.close();
   }
}
```

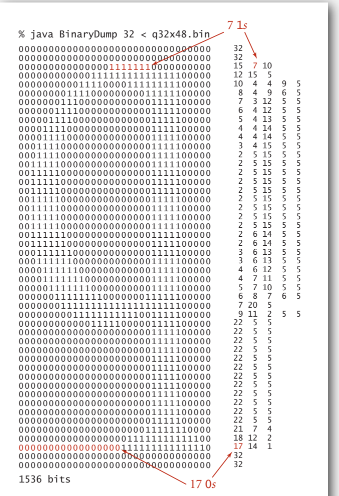

An application: compress a bitmap

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.



A typical bitmap, with run lengths for each row

- genomic encoding
- run-length encoding
- **Huffman compression**
- LZW compression

Variable-length codes

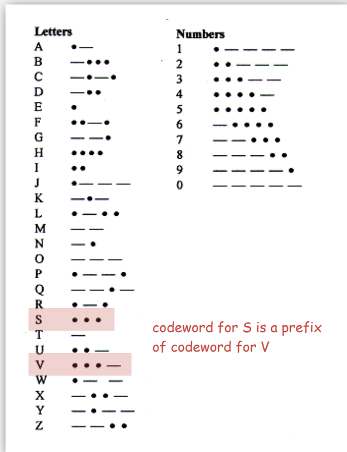
Use different number of bits to encode different chars.

Ex. Morse code: • • • – – – • • •

Issue. Ambiguity.

SOS ?
IAMIE ?
EEWNI ?
V7 ?

In practice. Use a medium gap to separate codewords.



Variable-length codes

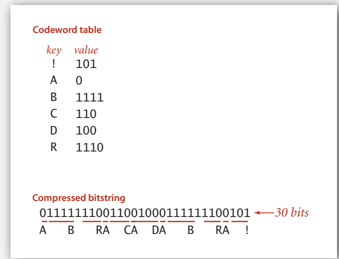
Q. How do we avoid ambiguity?

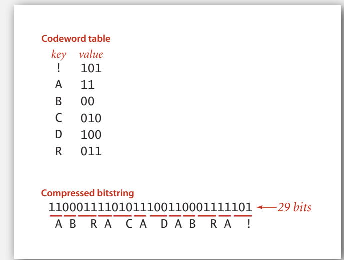
A. Ensure that no codeword is a prefix of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.



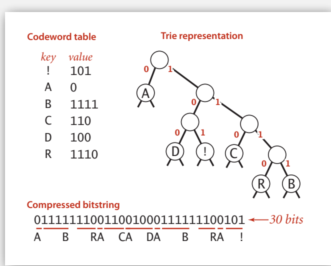


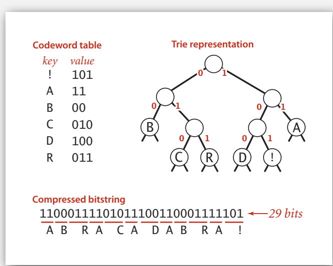
Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

A. A binary trie!

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





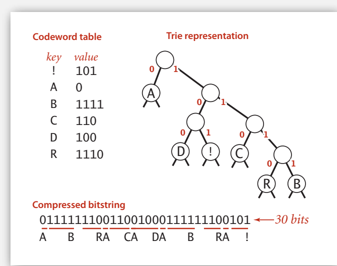
Prefix-free codes: compression and expansion

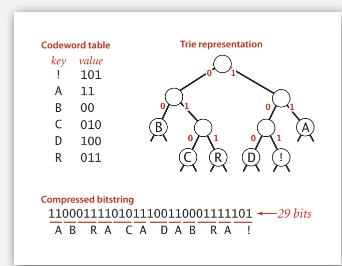
Compression.

- 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.





Huffman trie node data type

```
private static class Node implements Comparable<Node>
{
   private char ch;   // Unused for internal nodes.
   private int freq;  // Unused for expand.
   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;  }
}
```


Prefix-free codes: expansion

```
public void expand()
{
   Node root = readTrie();                    // read in encoding trie
   int N = BinaryStdIn.readInt();             // read in number of chars

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

read in encoding trie

read in number of chars

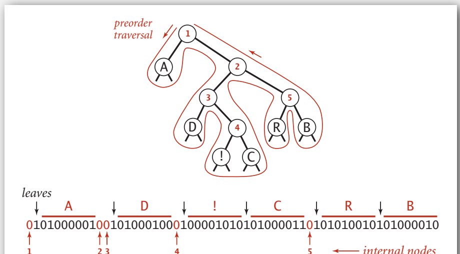
expand codeword for i^{th} char

Running time. Linear in input size (constant amount of work per bit read).

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

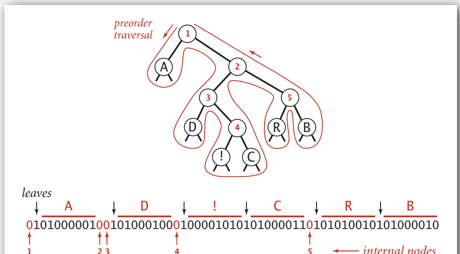
```
private static void writeTrie(Node x)
{
   if (x.isLeaf())
   {
      BinaryStdOut.write(true);
      BinaryStdOut.write(x.ch);
      return;
   }
   BinaryStdOut.write(false);
   writeTrie(x.left);
   writeTrie(x.right);
}
```

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.



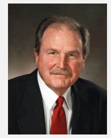
Using preorder traversal to encode a trie as a bitstream

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

Huffman codes

Q. How to find best prefix-free code?

A. Huffman algorithm.



David Huffman

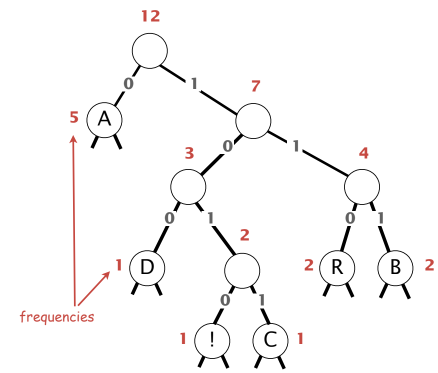
Huffman algorithm (to compute optimal prefix-free code):

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

Applications. JPEG, MP3, MPEG, PKZIP, GZIP, …

Constructing a Huffman encoding trie

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 code construction for A B R A C A D A B R A !

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));

    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);
    }

    return pq.delMin();
}
```

initialize PQ with singleton tries

merge two smallest tries

not used — total frequency — two subtries

Huffman encoding summary

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

no prefix-free code uses fewer bits

Pf. See textbook.

Implementation.

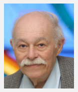
- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

Running time. Using a binary heap ⇒ $O(N + R \log R)$.

input size — alphabet size

Q. Can we do better? [stay tuned]

- genomic encoding
- run-length encoding
- 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.

Lempel-Ziv-Welch compression example

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	A B		R A		B R		A B R			A
value	41	42	52	41	43	41	44	81		83		82		88			41

LZW compression for ABRACADABRABRABRA

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

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

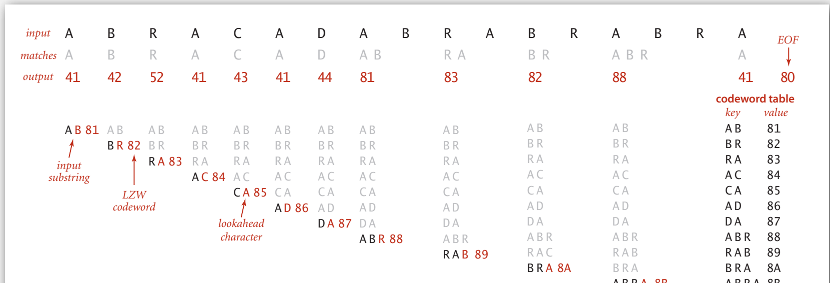
key	value
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

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.

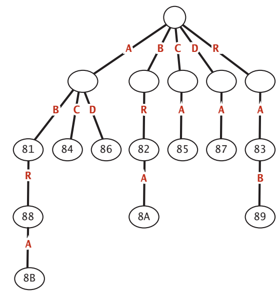


LZW compression for ABRACADABRABRABRA

Representation of LZW code table

Q. How to represent LZW code table?

A. A trie: supports efficient longest prefix match.



Trie representation of LZW code table

Remark. Every prefix of a key in encoding table is also in encoding table.

LZW compression: Java implementation

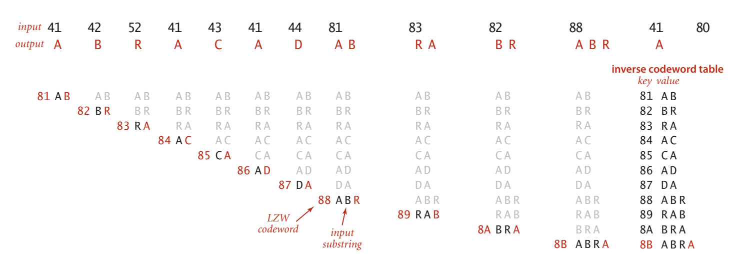
```
public static void compress()
{
  String input = BinaryStdIn.readString();

   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 last codeword and close input stream

LZW expansion

LZW expansion.

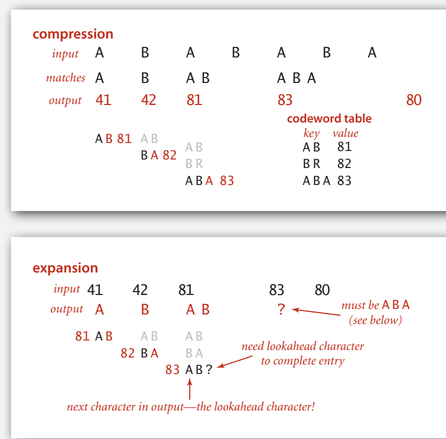
- Create ST associating string values with W-bit keys.
- Initialize ST to contain with single-char values.
- Read a W-bit key.
- Find associated string value in ST and write it out.
- Update ST.



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

LZW expansion: tricky situation

Q. What to do when next codeword is not yet in ST when needed?



LZW implementation details

How big to make ST?

- How long is message?
- Whole message similar model?
- [many variations have been developed]

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

LZ77 not patented ⇒ widely used in open source
LZW patent #4,558,302 expired in US on June 20, 2003
some versions copyrighted

PNG: LZ77.
Winzip, gzip, jar: deflate.
Unix compress: LZW.
Pkzip: LZW + Shannon-Fano.
GIF, TIFF, V.42bis modem: LZW.
Google: zlib which is based on deflate.

never expands a file

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.

Practical compression. Use extra knowledge whenever possible.