

Data Compression

- ▶ introduction
- ▶ basic coding schemes
- ▶ an application
- ▶ entropy
- ▶ LZW codes

References:

Algorithms 2nd edition, Chapter 22

<http://www.cs.princeton.edu/introalgsds/65compression>

▶ introduction

- ▶ basic coding schemes
- ▶ an application
- ▶ entropy
- ▶ LZW codes

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.



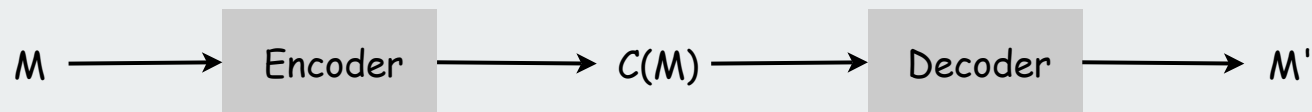
Encoding and decoding

Message. Binary data M we want to compress.

Encode. Generate a "compressed" representation $C(M)$.

uses fewer bits (you hope)

Decode. Reconstruct original message or some approximation M' .



Compression ratio. Bits in $C(M)$ / bits in M .

Lossless. $M = M'$, 50-75% or lower.

Ex. Natural language, source code, executables.

← this lecture

Lossy. $M \approx M'$, 10% or lower.

Ex. Images, sound, video.

"Poetry is the art of lossy data compression."

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.

- zip.
- MP3.
- MPEG.

What role will it play in the future?

Ex: If memory is to be cheap and ubiquitous, why are we doing lossy compression for music and movies??

- ▶ introduction
- ▶ **basic coding schemes**
- ▶ an application
- ▶ entropy
- ▶ LZW codes

Fixed length encoding

- Use same number of bits for each symbol.
- k -bit code supports 2^k different symbols

Ex. 7-bit ASCII

char	decimal	code
NUL	0	0
...	...	
a	97	1100001
b	98	1100010
c	99	1100011
d	100	1100100
...	...	
~	126	1111110
	127	1111111

this lecture:
special code for
end-of-message



a	b	r	a	c	a	d	a	b	r	a	!
1100001	1100010	1110010	1100001	1100011	1100001	1100100	1100001	1100010	1110010	1100001	1111111

12 symbols \times 7 bits per symbol = 84 bits in code

Fixed length encoding


- Use same number of bits for each symbol.
- k -bit code supports 2^k different symbols

Ex. 3-bit custom code

char	code
a	000
b	001
c	010
d	011
r	100
!	111

a	b	r	a	c	a	d	a	b	r	a	!
000	001	100	000	010	000	011	000	001	100	000	111

12 symbols \times 3 bits
36 bits in code



Important detail: decoder needs to know the code!

Fixed length encoding: general scheme

- count number of different symbols.
- $\lceil \lg M \rceil$ bits suffice to support M different symbols

Ex. genomic sequences

- 4 different codons
- 2 bits suffice

char	code
a	00
c	01
t	10
g	11

a	c	t	a	c	a	g	a	t	g	a
00	01	10	00	01	00	11	00	10	11	00

2N bits to encode
genome with N codons



- Amazing but true: initial databases in 1990s did **not** use such a code!

Decoder needs to know the code

- can amortize over large number of files with the same code
- in general, can encode an N-char file with $N \lceil \lg M \rceil + 16 \lceil \lg M \rceil$ bits

Variable-length encoding

Use different number of bits to encode different characters.

Ex. Morse code.

Issue: ambiguity.

• • • _ _ _ • • •

SOS ?

IAMIE ?

EEWNI ?

V7O ?

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 encoding

Use different number of bits to encode different characters.

Q. How do we avoid ambiguity?

A1. Append special stop symbol to each codeword.

A2. Ensure that **no** encoding is a **prefix** of another.

Ex. custom prefix-free code

char	code
a	0
b	111
c	1010
d	100
r	110
!	1011

S	...	← prefix of V
E	.	← prefix of I, S
I	..	← prefix of S
V	...-	

28 bits in code

a	b			r			a	c			a	d			a	b			r	a	!						
0	1	1	1	1	1	0	0	1	0	1	0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1	1

Note 1: fixed-length codes are prefix-free

Note 2: can amortize cost of including the code over similar messages

Prefix-free code: Encoding and Decoding

How to represent? Use a **binary trie**.

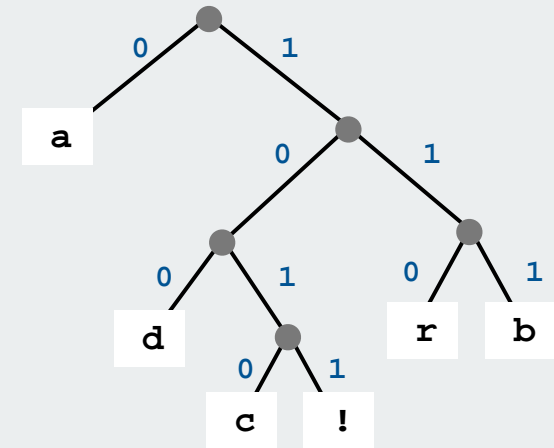
- Symbols are stored in leaves.
- Encoding is path to leaf.

Encoding.

- Method 1: start at leaf; follow path up to the root, and print bits in reverse order.
- Method 2: create ST of symbol-encoding pairs.

Decoding.

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

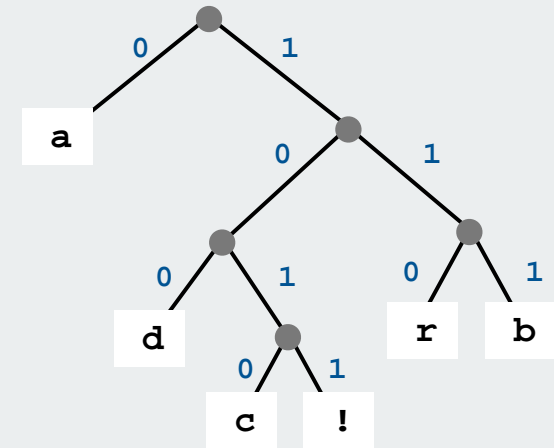


char	encoding
a	0
b	111
c	1010
d	100
r	110
!	1011

Providing the code

How to transmit the trie?

- send preorder traversal of trie.
 - we use * as sentinel for internal nodes
 - [what if no sentinel is available?]
- send number of characters to decode.
- send bits (packed 8 to the byte).



preorder traversal
chars to decode
the message bits

```
*a**d*c!*rb
12
0111110010100100011111001011
```

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

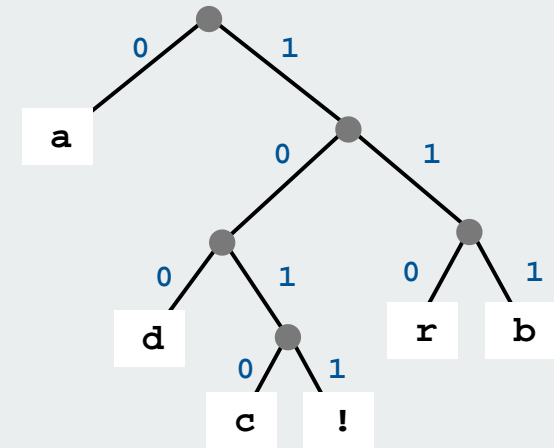
a	b	r	a	c	a	d	a	b	r	a	!
0	1	1	1	1	0	0	1	0	1	0	0
1	1	1	0	0	1	0	0	0	1	1	1

char	encoding
a	0
b	111
c	1010
d	100
r	110
!	1011

Prefix-free decoding implementation

```
public class PrefixFreeDecoder
{
    private Node root = new Node();
    private class Node
    {
        char ch;
        Node left, right;
        Node()
        {
            ch = StdIn.readChar();
            if (ch == '*')
            {
                left = new Node();
                right = new Node();
            }
        }
        boolean isInternal() { }
    }

    public void decode()
    {
        /* See next slide. */
    }
}
```

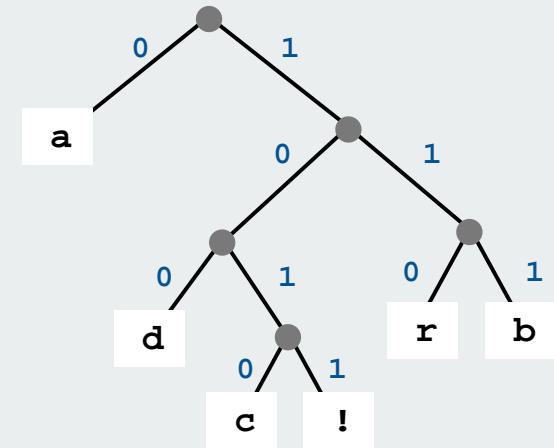


build tree from
preorder traversal

***a**d*c!*rb**

Prefix-free decoding iImplementation

```
public void decode()
{
    int N = StdIn.readInt();
    for (int i = 0; i < N; i++)
    {
        Node x = root;
        while (x.isInternal())
        {
            use bits, not chars → char bit = StdIn.readChar();
            in actual applications    if (bit == '0') x = x.left;
                                    else if (bit == '1') x = x.right;
        }
        System.out.print(x.ch);
    }
}
```



```
more code.txt
12
0111110010100100011111001011

% java PrefixFreeDecoder < code.txt
abacadabra!
```


Introduction to compression: summary

Variable-length codes can provide better compression than fixed-length

a	b	r	a	c	a	d	a	b	r	a	!
1100001	1100010	1110010	1100001	1100011	1100001	1100100	1100001	1100010	1110010	1100001	1111111

a	b	r	a	c	a	d	a	b	r	a	!
000	001	100	000	010	000	011	000	001	100	000	111

a	b	r	a	c	a	d	a	b	r	a	!														
0	1	1	1	1	0	0	1	0	1	0	0	1	0	0	0	1	1	1	1	0	0	1	0	1	1

Every trie defines a variable-length code

Q. What is the **best** variable length code for a given message?

Huffman coding

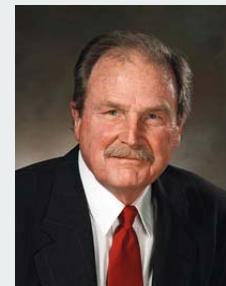
Q. What is the **best** variable length code for a given message?

A. Huffman code. [David Huffman, 1950]

To compute Huffman code:

- count frequency p_s for each symbol s in message.
- start with one node corresponding to each symbol s (with weight p_s).
- repeat until single trie formed:
 - select two tries with min weight p_1 and p_2
 - merge into single trie with weight $p_1 + p_2$

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

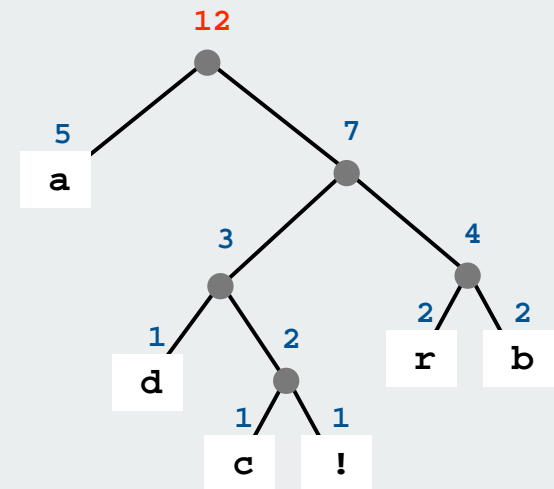
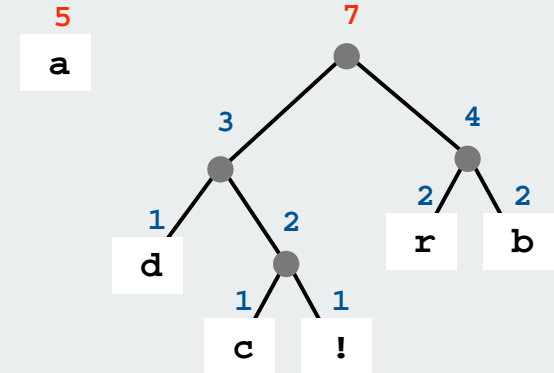
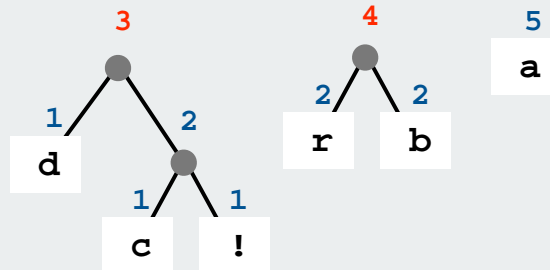
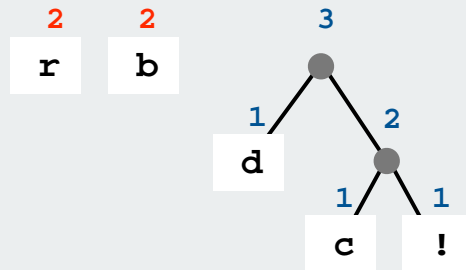
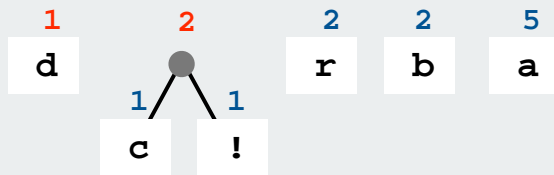


David Huffman

Huffman coding example

a b r a c a d a b r a !

1 1 1 2 2 5
c ! d r b a



Huffman trie construction code

```
int[] freq = new int[128];  
for (int i = 0; i < input.length(); i++)  
{ freq[input.charAt(i)]++; }
```

← tabulate
frequencies

```
MinPQ<Node> pq = new MinPQ<Node>();  
for (int i = 0; i < 128; i++)  
    if (freq[i] > 0)  
        pq.insert(new Node((char) i, freq[i], null, null));
```

← initialize
PQ

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

← merge
trees

two subtrees

↑
internal node
marker

↑
total
frequency

Huffman encoding summary

Theorem. Huffman coding is an **optimal** prefix-free code.



no prefix-free code uses fewer bits

Implementation.

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

Running time. Use binary heap $\Rightarrow O(M + N \log N)$.



output
bits



distinct
symbols

Can we do better? [\[stay tuned\]](#)

- ▶ introduction
- ▶ basic coding schemes
- ▶ **an application**
- ▶ entropy
- ▶ LZW codes

An application: compress a bitmap

Typical black-and-white-scanned image

300 pixels/inch

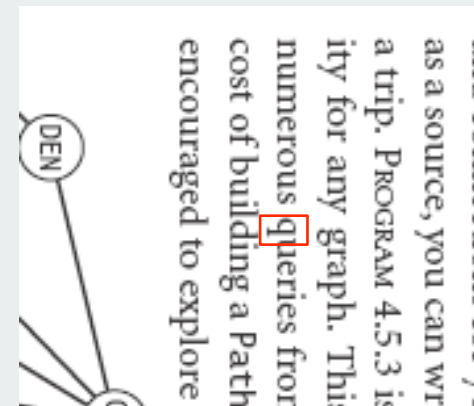
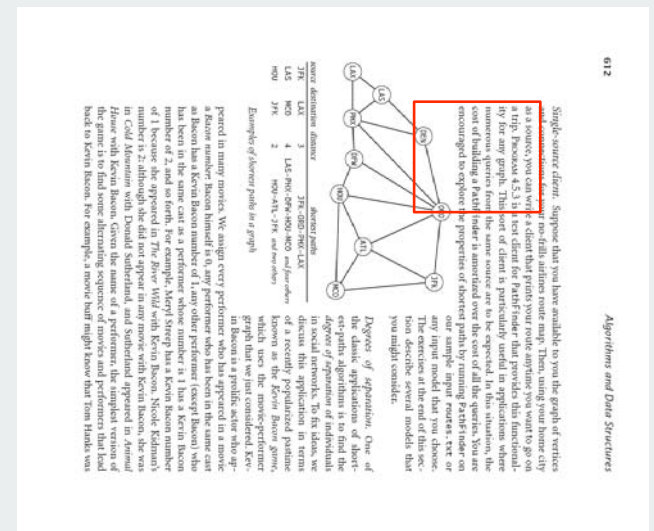
8.5 by 11 inches

$$300 \times 8.5 \times 300 \times 11 = 8.415 \text{ million bits}$$

Bits are mostly white

Typical amount of text on a page:

40 lines * 75 chars per line = 3000 chars



Natural encoding of a bitmap

one bit per pixel

[illegible]

19-by-51 raster of letter 'q' lying on its side

Run-length encoding of a bitmap

natural encoding. $(19 \times 51) + 6 = 975$ bits. ↗ to encode number of bits per line

run-length encoding. $(63 \times 6) + 6 = 384$ bits.

63 6-bit run lengths

[illegible]

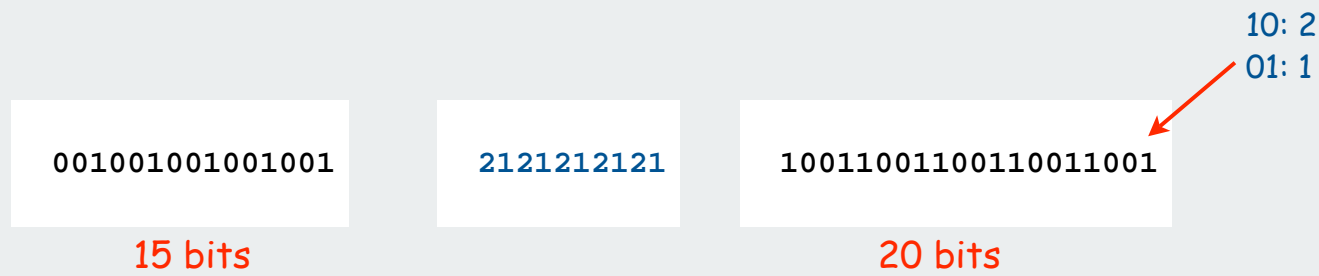
51					
28	14	9			
26	18	7			
23	24	4			
22	26	3			
20	30	1			
19	7	18	7		
19	5	22	5		
19	3	26	3		
19	3	26	3		
19	3	26	3		
19	3	26	3		
20	4	23	3	1	
22	3	20	3	3	
1	50				
1	50				
1	50				
1	50				
1	50				
1	2	46	2		

19-by-51 raster of letter 'q' lying on its side

RLE

Run-length encoding

- Exploit long runs of repeated characters.
- Bitmaps: runs alternate between 0 and 1; just output run lengths.
- Issue: how to encode run lengths (!)



- Does not compress when runs are short.

Runs are **long** in typical applications (such as black-and-white bitmaps).

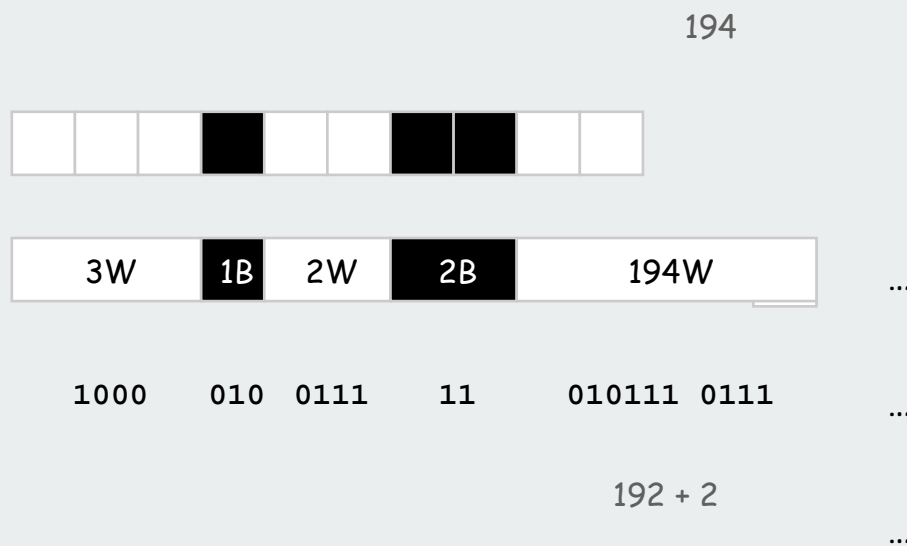
Run-length encoding and Huffman codes in the wild

ITU-T T4 Group 3 Fax for black-and-white bitmap images (~1980)

- up to 1728 pixels per line
- typically mostly white.

Step 1. Use run-length encoding.

Step 2. Encode run lengths using **two** Huffman codes. ← one for white and one for black



run	white	black
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
...
63	00110100	000001100111
64+	11011	0000001111
128+	10010	000011001000
...
1728+	010011011	0000001100101

Huffman codes built from
frequencies in huge sample

BW bitmap compression: another approach

Fax machine (~1980)

- slow scanner produces lines in sequential order
- compress to save **time** (reduce number of bits to send)

Electronic documents (~2000)

- high-resolution scanners produce huge files
- compress to save **space** (reduce number of bits to save)

Idea:

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

Ex. Typical page

- 40 lines, 75 chars/line ~ 3000 chars
- compress to ~ 2000 chars with Huffman code
- reduce file size by a **factor of 500 (! ?)**

Bottom line: **Any** extra information about file can yield dramatic gains

- ▶ introduction
- ▶ basic coding schemes
- ▶ an application
- ▶ **entropy**
- ▶ LZW codes

What data can be compressed?

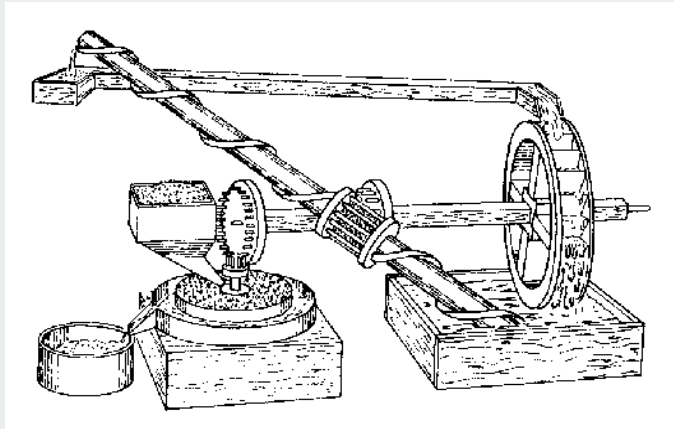
US Patent 5,533,051 on "Methods for Data Compression", which is capable of compression **all** files.

Slashdot reports of the Zero Space Tuner™ and BinaryAccelerator™.

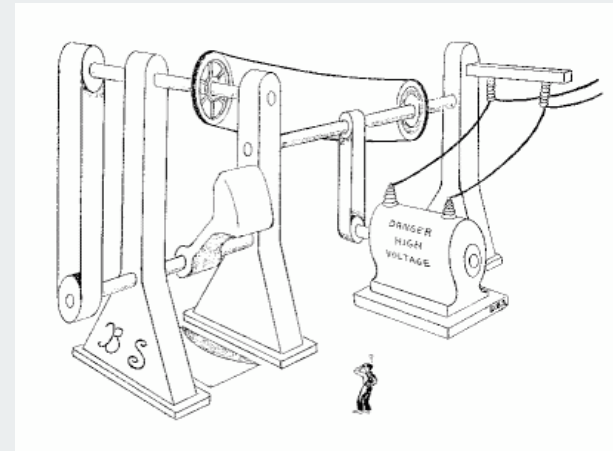
"ZeoSync has announced a breakthrough in data compression that allows for 100:1 lossless compression of **random** data. If this is true, our bandwidth problems just got a lot smaller...."

Perpetual Motion Machines

Universal data compression algorithms are the analog of perpetual motion machines.



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>

What data can be compressed?

Theorem. Impossible to losslessly compress **all** files.

Pf 1.

- consider all 1,000 bit messages.
- 2^{1000} possible messages.
- only $2^{999} + 2^{998} + \dots + 1$ can be encoded with ≤ 999 bits.
- only 1 in 2^{499} can be encoded with ≤ 500 bits!

Pf 2 (by contradiction).

- given a file M , compress it to get a smaller file M_1 .
- compress that file to get a still smaller file M_2 .
- continue until reaching file size 0.
- implication: **all** files can be compressed with **0** bits!

Practical test for any compression algorithm:

- given a file M , compress it to get a (smaller, you hope) file M_1
- compress that file to get a still smaller file M_2 .
- continue until file size does not decrease

A difficult file to compress

One million pseudo-random characters (a - p)

fclkkacifobjofmkgdcoiicnfmcpjcjfccabckjamolnihkbgobcjbngjiceeelpfgcjiihppenefllhglfemdemgahlbpi
ggmllmnefnhjelmgjncjcidlhkglhcnenidmmgnobkeglpnadanfbecoonbiehglmpnhkkamdffpacjmgojmcaabpcjce
cplfbgamlidceklhfkkmiojldnoaagiheiapaimlcnlljniggpeanbmogjkccogpkmkmoifioeikefjidbadgdcepnhdpfj
aeapdjefoklpdeghidbgcaiemajllhnnidigeihbeibifemacfadnknhlbgincpmimdogimgeeomgeljffjgklkdgnhafoho
npjbmllkapddhmepdnckeajebmeknmeejnmenbmnnfefdhhpmigbbjknjmobimamjjaaaffhlhiggaljbajnebidpaeigd
goghcihodnlhahllhhoojdfacnhadhgkfahmeaebccacgeojgikcoapknlomfignanedmajinlompjoaifiaejbcjcdibp
kofcbmjiobbpdhfilfajkhfmppcngdneeinpnfafaeladbhhifechinknpdnplamackphekokigpddmmjnbngklhibohdf
eaggmclllmdhafklmdmimdbplggbbejckmhlkjocjjlcngckfpfakmnpiaanfddjdlleiniilaenbnikgfnjfcophbgkhdg
mfpoehfmbpiaignphogbkelpobonmfghpdgmkfedkfkchceeldkcofaldinljcgafimaanelmfkokcjekefkbmegcgj
ifjcpjppnabldjoaafpbdafigcoibbcmoffbbgigmngefpkmbhbghlbdjngenldhgnfbdldcmjdmoflhocgfjoldfjpaok
epndejmnbiealkaofifekdjkgedgdlgbioacflfjlaafbcaemgpjlagbdgilhcfdcamhfmpffgohjphlmhegjechgdppklj
pndphfcnnganmbmnggpphnckbieknjhilafkegboilajdppcodpeoddldjfcpialoalfeomjbphkmhnpdmcpkggeaohfdm
cnegmibjkaajcdcpjcpjgminhhakihfgiiaachfepffnilcooiciepoapmdjniimfbolchkbkmbhbkkgconimkdchahcnhap
fdkiapikencegcjapkjkljgdlmgncpbakhjidapblcgeekkjaoihbnnbighboengpmedliofoofdcphelapijcegej
gcldcfodikaléhccpbccfakkblmoobdmgdgkafbbkjnidokfakjclbchambcpaepfeinmenmpoodadoecbgbmfkkeabi
laoeoggghoekamaibhjibefmoppbhfbhffapjnodlofeihmjahmeipejlfhloefgmjhjnlomapjakhhjpncomippeanbik
khekpcfgbgkmklipfbiikdkdcbolofhelipbkbjmfjoempccneaebklibmcaddldmjdcajpmhhaeedbbfpjafcnlianlfcj
mmbfncpdcceodeldhmnbdjmeajmboclkggogjghlohblbhgjkhkmclohkjamfmcchkchmiadggjhjehflcbklfifackbecg
joggpbkhlcmfhipflhmnmi fpmcoldbeghpcekhgmna hjpabnomnokldjcpppbcpgcjofngmbdcpeeeiiiclmbbmfjkh
anckidhmbearmlabncncpbhoafajjicnfeenppoekmlddholnbdjapbfcajblbooiapfmmeoafedflmdcbaodgeahimc
gpcammjljoebpfmghogfckgmomecdipmodbcempidfnlcggpgbfffoncajpncomalgooiikeolmiglikjkolgolfkdgiijj
iooiokdihjbbfoioibakadjnedlodeeiijkliicnioimablfdpjiafcfineecbafaamheiipegegibioocmlmhjekfikf
effmddhoakllnifdhckmbonbchfhhclecjamjildonjjdpifngbojianpljahpkindkdoanlldcbmlmhjfomifhmncikol
jjhebidjdphpdepibfgdonjljfgifimniipogockpidamnkcpipglafmlmoacjibognbplejnikdoefccdpfkomkimffgj
gielocdemnblimfmbkfbhkelkpfoheokfofochbmiflee cbglmnfbnfncjmfenihdcoieflllemnohlfdcmbdfebdmbeeb
balggfba jdamp lphdgiimehglpikbipnkkecekhilchhhfaeafbbfdmcjojfhppongkfdmhjpcieofcnjgkpi bcbiblf
njlejkc ppbhopohdghljlcokhdoahfmlglbdkliajbmnnkfkcklhlhjhhoiginaimgcabcfemjdnbfhohkjphnklcbhc
jpgbadakoecbkjcaebbanhnfhpnfkfbfpohmnkligpgfkjadomdjnhlnfailfpcmnololdjekeolhdkebi ffeba j jpc l g
hllmemegncknmk keogilijmmkomllbkkabelmodcohdhppdakbelmlejdnmbfmcjdebefnjihnejmnogeeafldabjcgfo
aehldcmkbnbafpciefhlopici fadbppgmfngecjhefnkbjmliodhelhicnfoongngemdddepchkokdjafegnpgledakmbcp
cmkckhbffei hpkajginfhdolfnlgnadefamlfocdibhfkiaofeegppcjilndepleihkpkkgkphbnkggjiaolnolbjpobjd
cehglelckbhjilafccfipgebp....

A difficult file to compress

```
public class Rand
{
    public static void main(String[] args)
    {
        for (int i = 0; i < 1000000; i++)
        {
            char c = 'a';
            c += (char) (Math.random() * 16);
            System.out.print(c);
        }
    }
}
```

231 bytes, but output is hard to compress
(assume random seed is fixed)

```
% javac Rand.java
% java Rand > temp.txt
% compress -c temp.txt > temp.Z
% gzip -c temp.txt > temp.gz
% bzip2 -c temp.txt > temp.bz2
```

```
% ls -l
      231 Rand.java
1000000 temp.txt
 576861 temp.Z
 570872 temp.gz
 499329 temp.bz2
```

resulting file sizes (bytes)

Information theory

Intrinsic difficulty of compression.

- Short program generates large data file.
- Optimal compression algorithm has to discover program!
- Undecidable problem.

Q. How do we know if our algorithm is doing well?

A. Want **lower bound** on # bits required by **any** compression scheme.

Language model

Q. How do compression algorithms work?

A. They exploit statistical biases of input messages.

- ex: white patches occur in typical images.
- ex: ord Princeton occurs more frequently than Yale.

Basis of compression: probability.

- Formulate probabilistic model to predict symbols.
simple: character counts, repeated strings
complex: models of a human face
- Use model to encode message.
- Use same model to decode message.

Ex. Order 0 Markov model

- R symbols generated independently at random
- probability of occurrence of i th symbol: p_i (fixed).

Entropy

A measure of information. [Shannon, 1948]

$$H(M) = p_0/\lg p_0 + p_1/\lg p_1 + p_2/\lg p_2 + \dots + p_{R-1}/\lg p_{R-1}$$

- information content of symbol s is proportional to $1/\lg_2 p(s)$.
- weighted average of information content over all symbols.
- interface between coding and model.

Ex. 4 binary models ($R = 2$)

	p_0	p_1	$H(M)$
1	1/2	1/2	1
2	0.900	0.100	0.469
3	0.990	0.010	0.0808
4	1	0	0



Claude Shannon

Ex. fair die ($R = 6$)

$p(1)$	$p(2)$	$p(3)$	$p(4)$	$p(5)$	$p(6)$	$H(M)$
1/6	1/6	1/6	1/6	1/6	1/6	2.585

Entropy and compression

Theorem. [Shannon, 1948] If data source is an order 0 Markov model, **any** compression scheme must use $\geq H(M)$ bits per symbol on average.

- Cornerstone result of information theory.
- Ex: to transmit results of fair die, need ≥ 2.58 bits per roll.

Theorem. [Huffman, 1952] If data source is an order 0 Markov model, Huffman code uses $\leq H(M) + 1$ bits per symbol on average.

Q. Is there any hope of doing better than Huffman coding?

A1. Yes. Huffman wastes up to 1 bit per symbol.
if $H(M)$ is close to 0, this difference matters
can do better with "arithmetic coding"

A2. Yes. Source may not be order 0 Markov model.

Entropy of the English Language

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 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."

A. Quite a bit.

Entropy of the English Language

Q. How much information is in each character of the English language?

Q. How can we measure it?

↖ model = English text

A. [Shannon's 1951 experiment]

- Asked subjects to predict next character given previous text.
- The number of guesses required for right answer:

# of guesses	1	2	3	4	5	≥ 6
Fraction	0.79	0.08	0.03	0.02	0.02	0.05

- Shannon's estimate: about 1 bit per char [0.6 - 1.3].

Compression less than 1 bit/char for English ? If not, keep trying!

- ▶ introduction
- ▶ basic coding schemes
- ▶ an application
- ▶ entropy
- ▶ **LZW codes**

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.

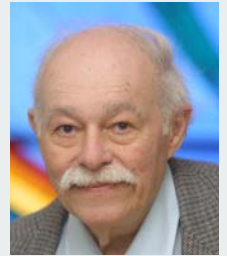
Adaptive model. Progressively learn and **update** model as you read text.

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

LZW Algorithm

Lempel-Ziv-Welch. [variant of LZ78]

- Create ST associating a fixed-length codeword with some previous substring.
- When input matches string in ST, output associated codeword.
- length of strings in ST grows, hence compression.



To send (encode) M .

- Find longest string s in ST that is a prefix of unsent part of M
- Send codeword associated with s .
- Add $s \cdot x$ to ST, where x is next char in M .

Ex. ST: $a, aa, ab, aba, abb, abaa, \text{abaab}, abaaa,$

- unsent part of M : $\text{abaababbbb} \dots$
- $s = \text{abaab}, x = a$.
- Output integer associated with s ; insert abaaba into ST.

LZW encoding example

input	code	add to ST
a	97	ab
b	98	br
r	114	ra
a	97	ac
c	99	ca
a	97	ad
d	100	da
a		
b	128	abr
r		
a	130	rac
c		
a	132	cad
d		
a	134	dab
b		
r	129	bra
a	97	
STOP	255	

input: 7-bit ASCII
output: 8-bit codewords

ASCII		ST	
key	value	key	value
	0	ab	128
		br	129
	...	ra	130
		ac	131
a	97	ca	132
b	98	ad	133
c	99	da	134
d	100	abr	135
	...	rac	136
r	114	cad	137
		dab	138
		bra	139

	127	STOP	255

To send (encode) M .

- Find longest string s in ST that is a prefix of unsent part of M
- Send integer associated with s .
- Add $s \cdot x$ to ST, where x is next char in M .

LZW encoding example

input: 7-bit ASCII
19 chars
133 bits

input	code
a	97
b	98
r	114
a	97
c	99
a	97
d	100
a	
b	128
r	
a	130
c	
a	132
d	
a	134
b	
r	129
a	97
STOP	255

output: 8-bit codewords
14 chars
112 bits

Key point: no need to send ST (!)

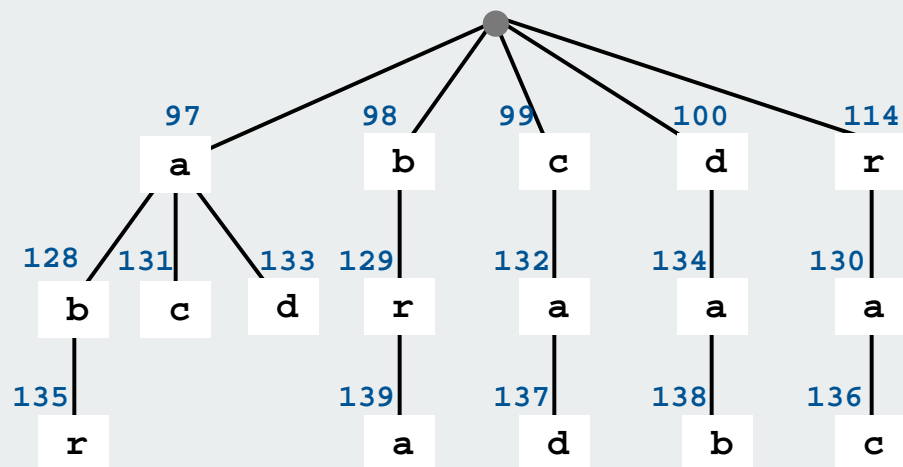
LZW encode ST implementation

Q. How to do longest prefix match?

A. Use a **trie** for the ST

Encode.

- lookup string suffix in trie.
- output ST index at bottom.
- add new node to bottom of trie.



Note that **all** substrings are in ST

ASCII		ST	
key	value	key	value
	0	ab	128
		br	129
	...	ra	130
		ac	131
a	97	ca	132
b	98	ad	133
c	99	da	134
d	100	abr	135
	...	rac	136
r	114	cad	137
		dab	138
		bra	139

	127	STOP	255

LZW encoder: Java implementation

```
public class LZWEncoder
{
    public static void main(String[] args)
    {
        LookAheadIn in = new LookAheadIn();

        LZWst st = new LZWst();

        while (!in.isEmpty())
        {
            int codeword = st.getput(in);
            StdOut.println(codeword);
        }
    }
}
```

input stream
with lookahead

specialized
TST

encode text
and build
TST

postprocess
to encode in
binary

Use specialized TST

- initialized with ASCII chars and codes
- `getput()` method returns code of longest prefix *s*
and adds *s* + next char to symbol table

Need input stream with `backup` [stay tuned]

LZW encoder: Java implementation (TST scaffolding)

```
public class LZWst
{
```

```
    private int i;
    private int codeword;
    private Node[] roots;
```

← next codeword to assign

← codeword to return

← array of TSTs

```
    public LZWst()
    {
        roots = new Node[128];
        for (i = 0; i < 128; i++)
            roots[i] = new Node((char) i, i);
    }
```

← initialize
with ASCII

```
    private class Node
    {
        Node(char c, int codeword)
        { this.c = c; this.codeword = codeword; }
        char c;
        Node left, mid, right;
        int codeword;
    }
```

← standard
node code

```
    public int getput(LookAheadIn in)
    // See next slide.
```

```
}
```


LZW encoder: Java implementation (TST search/insert)

```
public int getput(LookAheadIn in)
{
    char c = in.readChar();
    if (c == '!') return 255;
    roots[c] = getput(c, roots[c], in);
    in.backup();
    return codeword;
}
```

← longest prefix
codeword

caution:
tricky
recursive
code

```
public Node getput(char c, Node x, LookAheadIn in)
{
    if (x == null)
    {
        x = new Node(c, i++); return x;
    }

    if (c < x.c) x.left = getput(c, x.left, in);
    else if (c > x.c) x.right = getput(c, x.right, in);
    else
    {
        char next = in.readChar();
        codeword = x.codeword;
        x.mid = getput(next, x.mid, in);
    }

    return x;
}
```

← recursive
search and
insert

check for codeword overflow omitted

LZW encoder: Java implementation (input stream with lookahead)

```
public class LookAheadIn
{
    In in = new In();
    char last;
    boolean backup = false;

    public void backup()
    { backup = true; }

    public char readChar()
    {
        if (!backup)
        { last = in.readChar(); }
        backup = false;
        return last;
    }

    public boolean isEmpty()
    { return !backup && in.isEmpty(); }
}
```

Provides input stream with **one-character lookahead**.

backup() call means that last **readChar()** call was lookahead.

LZW Algorithm

Lempel-Ziv-Welch. [variant of LZ78]

- Create ST and associate an integer with each **useful** string.
- When input matches string in ST, output associated integer.
- length of strings in ST grows, hence compression.
- **decode by rebuilding ST from code**

To send (encode) M .

- Find longest string s in ST that is a prefix of unsent part of M
- Send integer associated with s .
- Add $s \cdot x$ to ST, where x is next char in M .

To **decode** received message to M .

- Let s be ST entry associated with received integer
- Add s to M .
- Add $p \cdot x$ to ST, where x is first char in s , p is previous value of s .

LZW decoding example

codeword	output	add to ST
97	a	
98	b	ab
114	r	br
97	a	ra
99	c	ac
97	a	ca
100	d	ad
128	a	
	b	da
130	r	
	a	abr
132	c	
	a	rac
134	d	
	a	cad
129	b	
	r	dab
97	a	bra
255	STOP	

role of keys and values switched

key	value	key	value
0		128	ab
		129	br
		130	ra
		131	ac
		132	ca
97	a	133	ad
98	b	134	da
99	c	135	abr
100	d	136	rac
		137	cad
		138	dab
		139	bra
		...	
127		255	

Use an **array**
to implement ST

To decode received message to M.

- Let s be ST entry associated with received integer
- Add s to M.
- Add $p \cdot x$ to ST, where x is first char in s , p is previous value of s .

LZW decoder: Java implementation

```
public class LZWDecoder
{
    public static void main(String[] args)
    {
```

```
        String[] st = new String[256];
        int i;
        for (i = 0; i < 128; i++)
        { st[i] = Character.toString((char) i); }
        st[255] = "!";
```

← initialize
ST with
ASCII

```
        String prev = "";
        while (!StdIn.isEmpty())
        {
            int codeword = StdIn.readInt();
            String s;
            if (codeword == i) // Tricky situation!
                s = prev + prev.charAt(0);
            else s = st[codeword];
            StdOut.print(s);
            if (prev.length() > 0)
            { st[i++] = prev + s.charAt(0); }
            prev = s;
        }
        StdOut.println();
    }
```

← preprocess
to decode
from binary

← decode text
and build ST

← Ex: ababababab


LZW decoding example (tricky situation)

input	code	add to ST
a	97	ab
b	98	ba
a		
b	128	aba
a		
b		
a	130	abab
b		
STOP	255	

key	value
128	ab
129	ba
130	aba
131	abab
...	
255	

codeword	output	add to ST
97	a	
98	b	ab
128	a	
	b	ba
130	a	
	b	
	a	aba
98	b	
255	STOP	

needed before
added to ST!



To send (encode) M.

- Find longest prefix
- Send integer associated with s.
- Add $s \cdot x$ to ST, where
x is next char in M.

To decode received message to M.

- Let s be ST entry for integer
- Add s to M.
- Add $p \cdot x$ to ST where
x is first char in s
p is previous value of s.

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 \Rightarrow 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 compression ratio benchmarks

Calgary corpus: standard data compression benchmark

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
1988	SAKDC	2.47
1994	PPM	2.34
1995	Burrows-Wheeler	2.29
1997	BOA	1.99
1999	RK	1.89

Entropy	Bits/char
Char by char	4.5
8 chars at a time	2.4
Asymptotic	1.3

← next assignment

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, SVD, ...

Limits on compression. Shannon entropy.

Theoretical limits closely match what we can achieve in practice.

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



Butch: I don't mean to be a sore loser, but when it's done, if I'm dead, kill him.

Sundance: Love to.

Butch: No, no, not yet. Not until me and Harvey get the rules straightened out.

Harvey: Rules? In a knife fight? **No rules.**

Butch: Well, if there ain't going to be any rules, let's get the fight started...