Compression: reduce the size of a file
- to save TIME when transmitting the file
- to save SPACE when storing it away

Widely used despite technological advances
- (Why?)

Basic concepts ancient (1952)
Best technology recently developed

Simple methods
Huffman codes
Information theory
Arithmetic codes
Adaptive (LZ) codes
Simple methods

Step 1: avoid doing something dumb
Step 2: try to do something clever

Character count encoding
Ex: ASCII binary file
Ex: ACGT molecular biology sequences
Ex: five-bit code for text

Run length encoding

Special hacks
Ex: encode /usr/dict/words
Run-length encoding

Ex: Blanks in text files
- tabular data without tabs
- (ancient) card images

Ex: Black-and-white graphics

How to represent counts? (Typical annoyance.)
- escape characters
- separate alphabet (may waste bits)
- different language entirely

Compression factor increases with resolution

No help for random files!
**Run-length encoding example**

<table>
<thead>
<tr>
<th>Binary Representation</th>
<th>Length of Run</th>
<th>Value of Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000001111111111111100000000</td>
<td>13 14 9</td>
<td></td>
</tr>
</tbody>
</table>
Huffman code

VARIABLE LENGTH code
  - use TRIE
  - encode frequently-used chars with fewer bits

Ex: abracadabra
  - 88 bits in byte code
  - 55 bits in 5-bit code
  - 33 bits in 3-bit code (only 5 different letters)

Huffman code:

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
<th>Frequency</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>001</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>0000</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>0001</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>r</td>
<td>01</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

23 bits

1*001*01*0*0000*0*0001*0*001*01*0
10010110000100011001011

23 bits
Huffman code implementation

**ENCODER:**
- Count frequency of occurrence of characters
- Build Trie by combining two smallest frequencies
- Trie defines code
- Pass through source, output code

**DECODER:**
- Use bits to guide trie search
- Output char when reached, restart at top

Have to transmit trie!
- Suffices to transmit code
- Relatively insignificant for big messages

**Issues:**
- Trie representation
- Data structure to hold nodes being processed
Huffman code construction example

6 3 3 2 4 5 3 1 1 2 4 3 2 2 5 1 2 11

6 3 3 2 4 5 3 2 2 4 3 2 2 5 1 2 11

6 3 3 2 4 5 3 2 2 4 3 2 2 5 3 1 2

6 3 3 4 5 3 2 2 4 3 4 4 2 5 3 1 2

6 3 3 4 5 3 2 2 4 3 4 4 2 2 2 1 2

6 3 3 4 5 3 5 4 4 4 5 3 1 2

6 6 3 4 5 3 2 2 3 2 2 2 5 1 1

6 6 4 5 3 6 5 4 4 4 5 1 1

6 6 4 5 3 6 5 4 4 4 5 1 1

14.7
Huffman code construction example (cont.)
Huffman trie representation (encoding)

Links point UP the tree
- one index per node to parent
- one bit per node (left or right child)

To get code for given char
- move up the trie
- PUSH left-right bits onto stack
- pop stack when top reached
Trie representation
- parent links
- right-left bit

Data structure for nodes being processed
- "indirect" priority queue

To build code:
- put all chars with nonzero counts in PQ
- while PQ has two or more nodes
  take two smallest off PQ
  make new node with sum of counts
  put on PQ

Another algorithm to build code:
- sort initial frequencies
- merge with generated frequencies
- no PQ needed
a simple string to be encoded using a minimal number of bits

c  f  p  d  g  l  r  u  a  b  o  t  m  s  e  n  i  *
1  1  2  2  2  2  3  3  3  3  4  4  5  5  6  11

1  1  1  2  2  2  2  2  3  3  3  3  4  4  5  5  6  11
1  2  2  2  2  2  3  3  3  3  4  4  5  5  6  11  *  2
2  2  2  2  2  2  3  3  3  3  4  4  5  5  6  11  *  2  3
2  2  3  3  3  3  4  4  5  5  6  11  *  2  3  4
3  3  3  3  4  4  5  5  6  11  *  2  3  4  4

3  3  3  4  4  5  5  6  11  *  3  4  4  5
3  4  4  5  5  6  11  *  3  4  4  5  6
4  4  5  5  6  11  *  4  4  5  6  6
5  5  6  11  *  4  4  5  6  6  8
5  5  6  11  *  5  6  6  8  8
6  11  *  5  6  6  8  8  10
11  *  6  6  8  8  10  11
11  *  8  8  10  11  12
11  *  10  11  12  16
*  11  12  16  21

16  21  23
37  23
60
Huffman code overview

**THM (Huffman, 1952):** Huffman code is optimal
(no variable-length code uses fewer bits)

Details omitted:
- have to transmit code
- need conventional tree representation for decoding

Problems:
- expensive computationally
- two-pass
- not optimal (!?)
Arithmetic coding

Represent source with a single number!

Improves upon Huffman by using FRACTIONAL numbers of bits

(Fraction of message taken by each letter is a real number)

Ex: abracadabra

<table>
<thead>
<tr>
<th>Letter</th>
<th>Start Fraction</th>
<th>End Fraction</th>
<th>Fraction of Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.00000</td>
<td>.45454</td>
<td>5/11 chars are a’s</td>
</tr>
<tr>
<td>b</td>
<td>.45454</td>
<td>.63636</td>
<td>2/11 chars are b’s</td>
</tr>
<tr>
<td>c</td>
<td>.63636</td>
<td>.72727</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>.72727</td>
<td>.81818</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>.81818</td>
<td>1.00000</td>
<td></td>
</tr>
</tbody>
</table>
### Arithmetic coding (encode)

**Rescale interval for each letter**

Ex: code for “abracadabra” is .2787887

<table>
<thead>
<tr>
<th>Letter</th>
<th>Start</th>
<th>End</th>
<th>Interval</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>.00000000</td>
<td>.45454547</td>
<td>(0, 4/11)</td>
<td>in (0, 1)</td>
</tr>
<tr>
<td>b</td>
<td>.20661159</td>
<td>.28925622</td>
<td>(5/11, 7/11)</td>
<td>in (0, 4/11)</td>
</tr>
<tr>
<td>r</td>
<td>.27422991</td>
<td>.28925622</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>.27422991</td>
<td>.28106004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>.27857634</td>
<td>.27919728</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>.27857634</td>
<td>.27885857</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>.27878159</td>
<td>.27880725</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>.27878159</td>
<td>.27879325</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>.27878690</td>
<td>.27878901</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>.27878863</td>
<td>.27878901</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>.27878863</td>
<td>.27878881</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rescale and output code digit when leading digits match**
Arithmetic coding (decode)

Given code (real number between 0 and 1)
- it falls in one of the intervals \((l, r)\)
  associated with the characters
  - output that character
  - rescale to \((0,1)\)

\[
\text{code} = \frac{\text{code} - l}{r - l};
\]
- repeat to get next char

Need to adjust algorithm to match rescaling
- used on decode (can be tricky to do)

Bottom line:
- slightly better than Huffman
- like having fractional bits
Arithmetic coding decode example

.27878872 a since .27... is in (0, 5/11)
.61333513 b (.27...- 0)/(5/11); in (5/11, 7/11)
.87334329 r (.61...- 5/11)/(2/11); in (9/11, 1)
.30338812 a
.66745383 c
.34199208 a
.75238258 d
.27620819 a
.60765803 b
.84211922 r
.13165572 a
Entropy

Basic concept of information theory
- formal model of source
- formal concept of "information content"

First order model: chars generated randomly
- with fixed probability (independently)

**THM:** Huffman is optimal in first order model

Entropy concept confirms intuition
- random data cannot be compressed
- data compressed by one optimal compressor won't be compressed further by another
- no guarantee of specific performance on all data
Entropy limitations

“average” case results

Depends on mathematical model

Real data may not fit any model

Ex: short program generates large data file
   • compression alg has to discover the program!

Statistical modeling
   • find short program (or model parameters)

Ex: speech recognition
A difficult file to compress

One million random characters

ylculksedgrdivayjpgkredehwhrvvbbltdkctqtocthnnhylmsgvw
achzjvwhzabzrhhsyqxnzgkpufxdytmoaajsorphpchnxlygnezeofo
igidqfyqfvmvldxreuseufgryyvuzsrdrilknwvordwsfblvhinawe
zslumidtljqydtbpazipawngrubtqyjfiowyaktrwuopoohtkuae
nxxcuiivbjeckdimqdtxxcwaundcobuekavqepoxmidcmdrahhuze
djmjxkeoyimflsoyxknnqkqsgwgjofhakqrhmfbfhgvtsousmvuewu
ngwqzhhllznusllaeppdphwojuiorbpsuxxebnszlktzbapzpscrpl
ltgiklkgczvxmrogywywjdqspqslilkpktzjarozmfaqvjmlqpp
esbvdgppkkxxeaphrlfabegwprtqnywwxhoysbionurepogfucqfn
ehrquwdyiuurcrpylexzkkaystirtemsgaplcpcfnwmfgfokcfyigso

...
Compressing a difficult file

#include "math.h"
main ()
{
    int i, j; char x;
    for (i = 0; i < 12500; i++)
    {
        for (j = 0; j < 80; j++)
        {
            x = 'a' + 26*drand48();
            printf("%c", x);
        }
    }
}

Ex: Postscript vs bitmap
Pure Adaptive Coding (LZ)

To encode the next character(s)

- if no match of length >1 in previous string
  output the next char by itself
- otherwise output a triple (two numbers and a char)
  number of chars back to longest previous match
  length of longest previous match
  char causing the mismatch

```
. 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
. this is the third time
.
. this * 3 3 the 4 3 i r d 6 2 i m e
```

Problems:

- big dictionary (N^2 prev substrings)
- online dictionary update not easy
- could be long way back to match
LZ encoding a binary file

. 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
. 0 1 1 0 0 1 1 0 0 0 0 1 1 0 1 1 1 1 0 1
.
. 0 1 1 0 4 4 0 6 5 1 4 2 1 5 2 –
Dictionary-based Adaptive Coding (LZW)

Keep a "best possible" DICTIONARY of substrings

N = number of substrings in dictionary

Set N = 0

Process the NEW substring beginning at the next character:
  • if no match of length 1 or more in dictionary
    output the next char by itself
  • otherwise output a pair (a number and a char)
    index to longest match in dictionary
    char causing the mismatch (or next char if at end)
    AND create new entry for dictionary!
LZW example

. 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
. this is the third time
  * dict output
. 0 t   t
. 1 h   h
. 2 i   i
. 3 s   s
. 4 *   *
. 5 is  2 s
. 6 *t  4 t
. 7 he  1 e
. 8 *th 6 h
. 9 ir  2 r
. 10 d   d
. 11 *ti 6 i
. 12 m   m
. 13 e   e
.
. this * 2 s 4 t 1 e 6 h 2 r d 6 i m e
LZW binary example

. 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
. 0 1 1 0 0 1 1 0 0 0 0 1 1 0 1 1 1 1 0 1

*dict output*

. 0 0
. 1 1
. 2 10 1 0
. 3 01 0 1
. 4 100 2 0
. 5 00 0 0
. 6 11 1 1
. 7 011 3 1
. 8 110 6 1
. 1 -

. 1 0 0 1 2 0 0 0 1 1 3 1 6 1 1 -
LZW implementation

Use trie for dictionary (standard representation)

ENCODE loop
- lookup string suffix in trie
- output dictionary index at bottom
- add new node to trie

DECODE options
- build trie (faster, takes space)
- build dictionary (slower)
Empirical results

Methods are available as UNIX tools
- **pack**: Huffman
- **compress**: LZ
- **gzip**: LZW

**Experiment**: Compress the text of “Moby Dick” (1220K)
- 5-bit 778K 63%
- Huffman 692K 56%
- LZ 523K 42%
- LZW 493K 40%

**gzip the results** | **pack the results**
--- | ---
Huffman 692K 624K 10% | 692K 687K 1%
LZ 523K 520K | 523K "no savings"
LZW 493K 493K | 493K "no savings"

**Experiment**: Compress a 1MB file of random chars
- 5-bit 638K 63%
- Huffman 610K 60%
- LZ 695K 68%
- LZW 646K 63%
Summary

Huffman:
  • represent FIXED-LENGTH bitstrings with variable-length codes

Lempel-Ziv:
  • represent VARIABLE-LENGTH bitstrings with fixed-length codes

Not covered: codes like JPEG
  • "lossy" codes for graphics
  • (don’t really need all the bits)

Other questions
  • compute on compressed data?
  • archiving
    compression code must be truly portable
    Ex: replace processor
    Ex: send files to different processor
  • massive compression needed on some data