

## COS 226 Lecture 14: File compression

**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

14-1

## 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

14-2

## 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!

14-3

## Run-length encoding example

```
0000000000000111111111111111000000000 13 14 9
000000000000111111111111111110000000 11 18 7
00000000111111111111111111111110000 8 24 4
0000000111111111111111111111111000 7 26 3
0000011111111111111111111111111110 5 30 1
00001111111100000000000000000000111111 4 7 18 7
0000111110000000000000000000000011111 4 5 22 5
00001110000000000000000000000000111 4 3 26 3
00001110000000000000000000000000111 4 3 26 3
00001110000000000000000000000000111 4 3 26 3
00001110000000000000000000000000111 4 3 26 3
000011110000000000000000000000001110 5 4 23 3 1
000000011100000000000000000000011100 7 3 20 3 3
0111111111111111111111111111111111 1 35
0111111111111111111111111111111111 1 35
0111111111111111111111111111111111 1 35
0111111111111111111111111111111111 1 35
0111111111111111111111111111111111 1 35
01100000000000000000000000000000011 1 2 31 2
```

14-4

## 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:

a	1	5	5
b	001	2	6
c	0000	1	4
d	0001	1	4
r	01	2	4

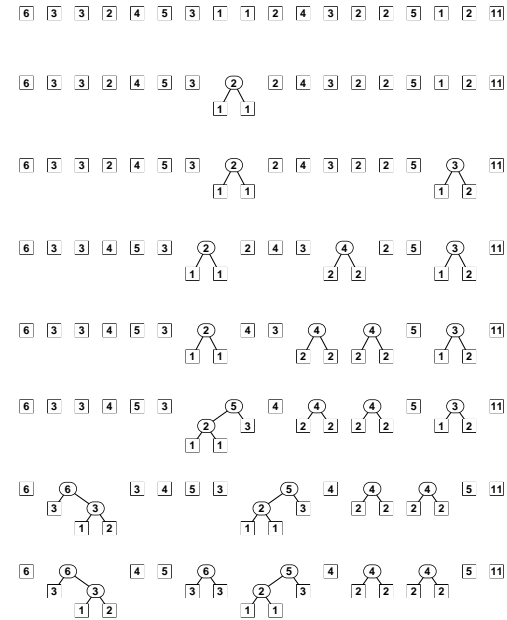
23 bits

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

23 bits

14-5

## Huffman code construction example



14-7

## Huffman code implementation

### ENCODE:

- count frequency of occurrence of characters
- build TRIE by combining two smallest frequencies
- trie defines code
- pass through source, output code

### DECODE:

- use bits to guide trie search
- output char when reached, restart at top

Have to transmit trie!

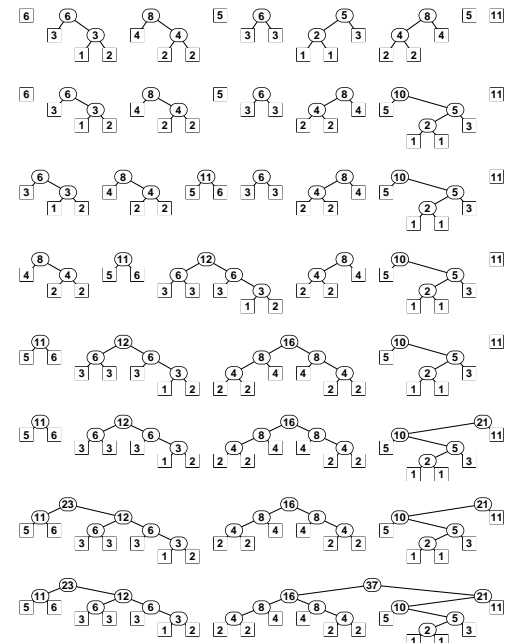
- suffices to transmit code
- relatively insignificant for big messages

Issues:

- trie representation
- data structure to hold nodes being processed

14-6

## Huffman code construction example (cont.)

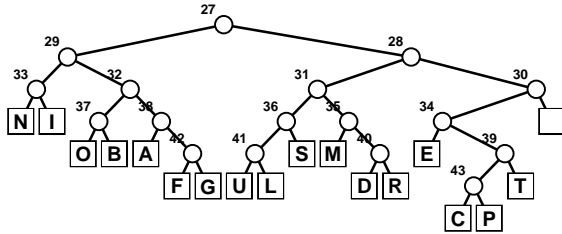


14-8

## Huffman trie representation (encoding)

Links point UP the tree

- one index per node to parent
- one bit per node (left or right child)



```
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
27 27 28 28 29 29 30 31 31 32 32 34 35 36 38 39
*   *   *   *   *   *   *   *   *
```

To get code for given char

- move up the trie
- PUSH left-right bits onto stack
- pop stack when top reached

14-9

## Huffman code implementation (continued)

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

14-10

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 1 2 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 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
```

14-11

## 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 (!?)

14-12

## 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

a	0.00000	.45454	5/11 chars are a's
b	.45454	.63636	2/11 chars are b's
c	.63636	.72727	
d	.72727	.81818	
r	.81818	1.00000	

14-13

## 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  $(0,1)$
- ```
code = (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

14-15

## Arithmetic coding (encode)

Rescale interval for each letter

Ex: code for "abracadabra" is .2787887

|   |           |           |              |              |
|---|-----------|-----------|--------------|--------------|
| a | .00000000 | .45454547 | (0, 4/11)    | in (0, 1)    |
| b | .20661159 | .28925622 | (5/11, 7/11) | in (0, 4/11) |
| r | .27422991 | .28925622 |              |              |
| a | .27422991 | .28106004 |              |              |
| c | .27857634 | .27919728 |              |              |
| a | .27857634 | .27885857 |              |              |
| d | .27878159 | .27880725 |              |              |
| a | .27878159 | .27879325 |              |              |
| b | .27878690 | .27878901 |              |              |
| r | .27878863 | .27878901 |              |              |
| a | .27878863 | .27878881 |              |              |

Rescale and output code digit when leading digits match

14-14

## 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 |                                      |

14-16

## 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

14-17

## A difficult file to compress

### One million random characters

```
ylculksedgrdivayjpgkredehwhrvvbbldkctqtoctnnhylmsgvw  
achzjvwzhazbrhsyqxnzgzkpufoxdytmoajjsorphchnxlygnezneof  
gidqfyqfmvldxreuseufgryyvuzsrdreilknwvordwsfblvhinawe  
zslumidtljqydtbprazipawngrubtqyjfiowyaktrwuopoohtkuae  
nzxcuivbjekdimqdtxxcwkauandcobuekavqepoxmidcmdrahuzd  
jmjxkeoyimflsoyxknqkqsgwgjofhakqrhmbfhgvtgsousmvcuwuu  
ngwqzzhllznusllaepdphwojuiorbpsuxxebnszltkzbpzpscprl  
ltgiklkgczvxmrgoywywdgapsqslilkpktzjarozmfvaqjmlqp  
esbvdgpkkxrxeaphrlfabegwprtqnywxhoysbionurepogfucqfn  
ehrqwudyiuurcrpylxzzkaystirtemsgaplcpfnwmgfokcfyigso  
...
```

14-19

## 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

14-18

## Compressing a difficult file

**130 chars!**

```
#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

14-20

## Pure Adaptive Coding (LZ)

To encode the next character(s)

- if no match of length  $\geq 1$  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
.  t h i s   i s   t h e   t h i r d   t i m e
.
.  t h i s * 3 3 t h e 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

14-21

## 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!

14-23

## 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 1 1 0 1 1 1 1 0 1
.
. 0 1 1 0 4 4 0      6 5 1   4 2 1   5 2 -
```

14-22

## LZW example

```
. 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
.  t h i s   i s   t h e   t h i r d   t i m e
.  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
.
.  t h i s * 2 s 4 t 1 e 6 h 2 r d 6 i m e
```

14-24

## 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 -
```

14-25

## 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"
```

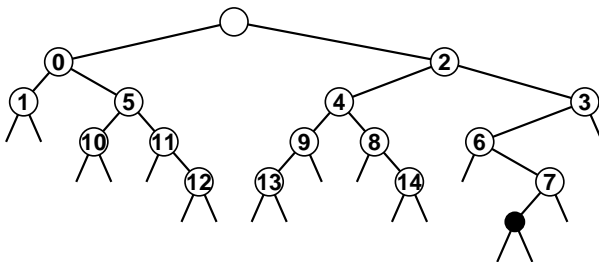
Experiment: Compress a 1MB file of random chars

```
. 5-bit 638K 63%
. Huffman 610K 60%
. LZ 695K 68%
. LZW 646K 63%
```

14-27

## LZW implementation

Use trie for dictionary (standard representation)



```
.0001111001110110110110001001101111000101111010
```

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)

14-26

## 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

14-28