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

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
- Ex: blanks in text files
- Ex: black-and-white graphics

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

General representation problem
- annoying practical problems
- usually work with either
  * bitstream/bytestream
  * separate language entirely

Special hacks
- Ex: encode /usr/dict/words

Run-length encoding

Remove blanks in files

ex: tabular data without tabs

ex: (ancient) card images

Record lengths of 0s and 1s in binary files

ex: black-and-white graphics

```
.000000000000011111111111111000000000 13 14  9
.000000000001111111111111111110000000 11 18  7
.000000001111111111111111111111110000  8 24  4
.000000011111111111111111111111111000  7 26  3
.000001111111111111111111111111111110  5 30  1
.000011111110000000000000000001111111  4  7 18 7
.000011111000000000000000000000011111  4  5 22 5
.000011100000000000000000000000000111  4  3 26 3
.000011100000000000000000000000000111  4  3 26 3
.000011100000000000000000000000000111  4  3 26 3
.000011100000000000000000000000000111  4  3 26 3
.000001111000000000000000000000001110  5  4 23 3 1
.000000011100000000000000000000111000  7  3 20 3 3
.011111111111111111111111111111111111  1 35
.011111111111111111111111111111111111  1 35
.011111111111111111111111111111111111  1 35
.011111111111111111111111111111111111  1 35
.011111111111111111111111111111111111  1 35
.011000000000000000000000000000000011  1  2 31 2
```

Compression factor increases with resolution

No help for random files!

Huffman code

Variable length code, defined by trie

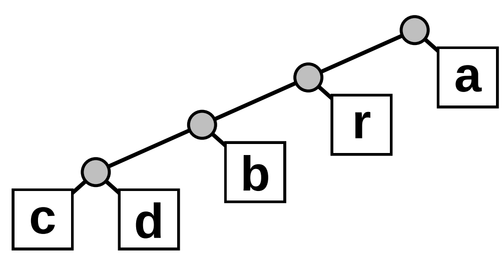
Ex: abracadabra

88 bits in byte code

55 bits in 5-bit code

33 bits in 3-bit code (only 5 different letters)

Huffman code:

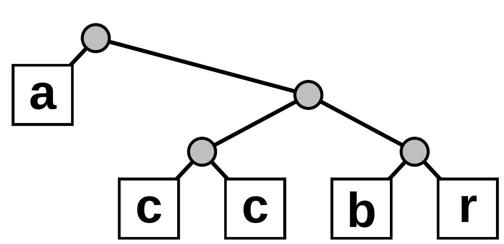


a	**1**	**5**	**5**
b	**001**	**2**	**6**
c	**0000**	**1**	**4**
d	**0001**	**1**	**4**
r	**01**	**2**	**4**
			23 bits

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

10010110000100011001011

not necessarily unique



a	**0**	**5**	**5**
b	**110**	**2**	**6**
c	**100**	**1**	**3**
d	**101**	**1**	**3**
r	**111**	**2**	**6**
			23 bits

01101110100010101101110

Depends on encoding frequently used chars

with fewer bits than infrequently used chars

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

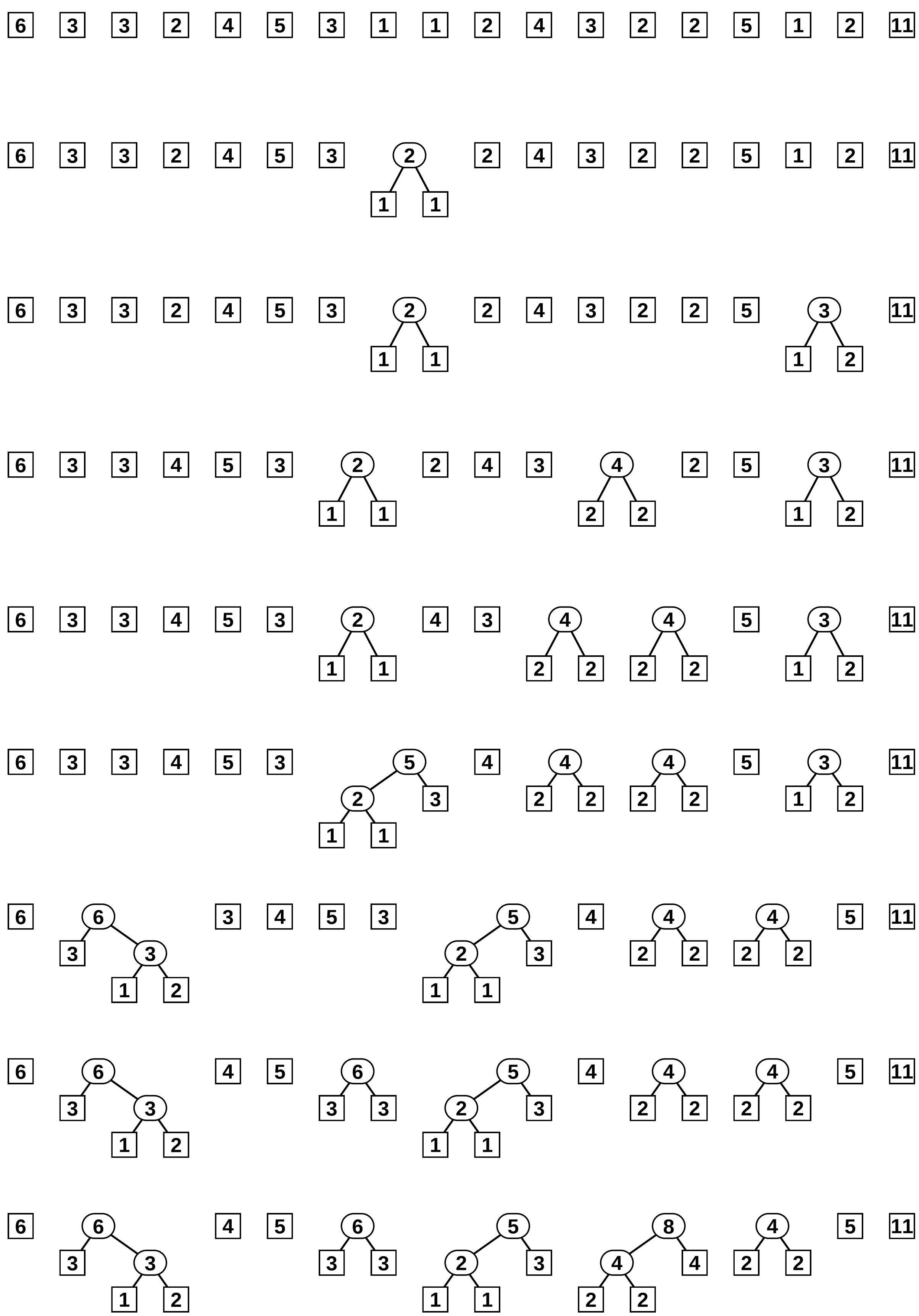
Have to transmit trie!

- suffices to transmit code
- relatively insignificant for big messages

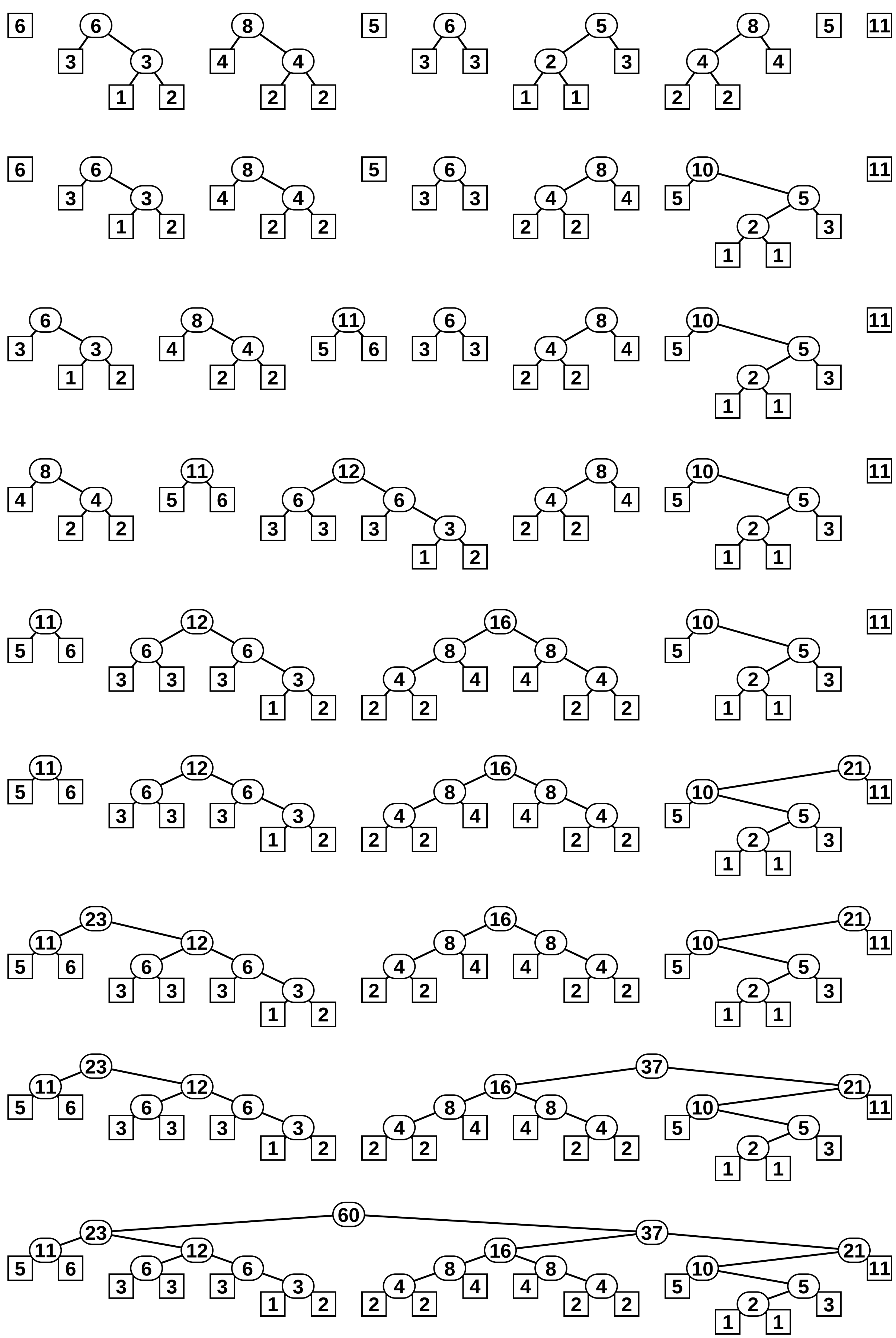
Issues:

- trie representation
- data structure to hold nodes being processed

Huffman code construction example



Huffman code construction example (cont.)



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

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

Problems:
- have to transmit code
- expensive computationally
- two-pass
- not optimal (!)

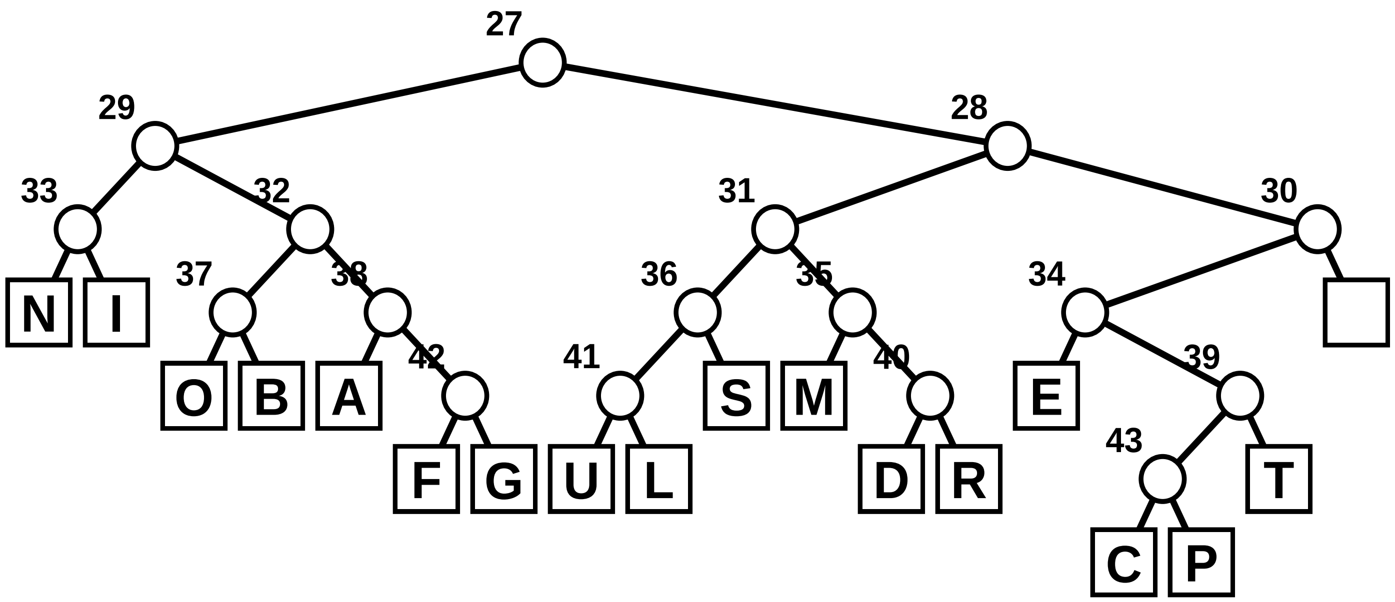
a simple string to be encoded using a minimal number of bits

```
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
```

Huffman trie representation

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
- * collect left-right bits
- * reverse string when top reached

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 compress further by another
* no guarantee of specific performance on all data

BUT

* "average" case results
* depends on mathematical model
* real data may not fit any model

Example:

    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

```
ylculksedgrdivayjpgkredehwhrvvbbltdkctqtoctnnhylmsgvwachzjvwhzabzrhsyqxnzgkpufx
dytmoajjsorphchnxlygnezneofgidqfyqfmvldxreuseufgryyvuzsrdreilknwvordwsfblvhinaw
ezslumidtljqydtbprazipawngrubtqyjfiowyaktrwuopoohtkuaenzxcuivbjekdimqdqtxxcwkau
ndcobuekavqepoxmidcmdrahhuzdjmjxkeoyimflsoyxknqkqsgwgjofhakqrhmbfhgvtgsousmvcuw
uungwqzzhllznusllaepdphwojuiorbpsuxxebnszlktzbapzpscprlltgiklkgczvxmrgoywywjdga
pspqslilkpktzjarozmfvaqjmnlqpesbvdgpkkxrxeaphrlfabegwprtqnywwxhoysbionurepogfuc
qfnehrquwdyiuurcrpylxzzkaystirtemsgaplcpcfnwmgfokcfyigsoudbsemxcenhofagogdidfqn
...
```

A lower bound for compression: 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 file??

Arithmetic Coding

Represent source with 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
```

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
```

Can rescale and output code digit when
leading digits of intervals match
(have to do so to handle long messages)

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

Ex:

.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

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

Pure Adaptive Coding (LZ)

To encode the next character(s)

- if no match of length >1 in previous string, output the next char
- otherwise, output
  - pointer 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 * 2 3 t h e 4 3 i r d 6 2 i m e
.
.
.  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 3
```

Problems:

big dictionary (N^2 prev substrings)

online dictionary update not easy

Dictionary-based Adaptive Coding (LZW)

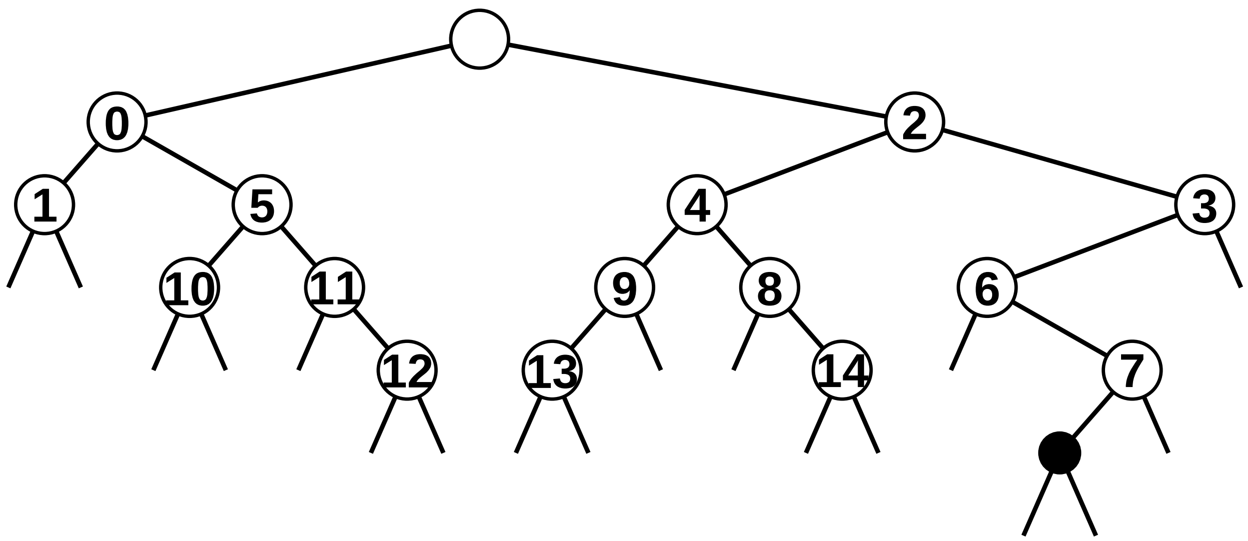
```
.  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
.  t h i s   i s   t h e   t h i r d   t i m e
.
.   0 t       t
.   1 h       h
.   2 i       i
.   3 s       s
.   4 *       *
.   5 is    2 s
.   6 *t    4 t
.   7 he    1 e
.   8 *th   6 h
.   9 ir    2 r
.  10 d       d
.  11 *ti   6 i
.  12 m       m
.  13 e       e
.
.  t h i s * 2 s 4 t 1 e 6 h 2 r d 6 i m e
```

Binary example of LZW

```
.  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  0 1 0
.  1  1 1 1
.  2  2 2 10    1 0
.  3  4 2 01    0 1
.  4  2 3 100   2 0
.  5  9 2 00    0 0
.  6 11 2 11    1 1
.  7 13 3 011   3 1
.  8 16 3 110   6 1
.
.  1 0 0 1 2 0 0 0 1 1 3 1 6 1
```

LZW implementation

Use trie for dictionary (standard representation)



.000111100111011011011000100110111110001011111010

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"

original	**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

original	**1012K**	
5-bit	**638K**	**63%**
Huffman	**610K**	**60%**
LZ	**695K**	**68%**
LZW	**646K**	**63%**

Summary

Don’t ignore the obvious

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