Data Compression

Applications of Data Compression

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.

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.

Encoding and Decoding

Message. Binary data M we want to compress.
Encode. Generate a "compressed" representation C(M).
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.

Lossy. M ∼ M', 10% or lower.
Ex. Images, sound, video.
Run-Length Encoding

**Natural encoding.** \((19 \times 51) + 6 = 975\) bits.

**Run-length encoding.** \((63 \times 6) + 6 = 384\) bits.

- 63 6-bit run lengths

| 000000000000000000000000000000000000000000000000011 | 28 14 9 |
| 000000000000000000000000000000000000000000000000011 | 26 18 7 |
| 000000000000000000000000000000000000000000000000011 | 23 24 4 |
| 000000000000000000000000000000000000000000000000011 | 22 26 3 |
| 000000000000000000000000000000000000000000000000011 | 20 30 1 |
| 000000000000000000000000000000000000000000000000011 | 19 7 18 7 |
| 000000000000000000000000000000000000000000000000011 | 19 5 22 5 |
| 000000000000000000000000000000000000000000000000011 | 19 3 26 3 |
| 000000000000000000000000000000000000000000000000011 | 19 3 26 3 |
| 000000000000000000000000000000000000000000000000011 | 19 3 26 3 |
| 000000000000000000000000000000000000000000000000011 | 19 3 26 3 |
| 000000000000000000000000000000000000000000000000011 | 20 4 23 3 1 |
| 000000000000000000000000000000000000000000000000011 | 22 3 20 3 3 |
| 000000000000000000000000000000000000000000000000011 | 1 50 |
| 000000000000000000000000000000000000000000000000011 | 1 50 |
| 000000000000000000000000000000000000000000000000011 | 1 50 |
| 000000000000000000000000000000000000000000000000011 | 1 50 |
| 000000000000000000000000000000000000000000000000011 | 1 2 46 2 |

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

---

**Run-Length Encoding**

**Ancient Ideas**

- Braille.
- Morse code.
- Natural languages.
- Mathematical notation.
- Decimal number system.

---

**Applications.**

- JPEG.
- ITU-T T4 fax machines. (black and white graphics)

---

**Natural encoding.**

\((19 \times 51) + 6 = 975\) bits.

---

**Run-length encoding (RLE).**

- Exploit long runs of repeated characters.
- Replace run by count followed by repeated character.
- Annoyance: how to represent counts.
- Runs in binary file alternate between 0 and 1, so output count only.
- “File inflation” possible if runs are short.

---

- JPEG.
- ITU-T T4 fax machines. (black and white graphics)
Fixed Length Coding

**Fixed length encoding.**
- Use same number of bits for each symbol.
- \( N \) symbols \( \Rightarrow \lceil \log N \rceil \) bits per symbol.

<table>
<thead>
<tr>
<th>char</th>
<th>dec</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>a</td>
<td>97</td>
<td>110001</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
<td>1100010</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
<td>1100011</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
<td>1100100</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>~</td>
<td>126</td>
<td>1111110</td>
</tr>
<tr>
<td>DEL</td>
<td>127</td>
<td>1111111</td>
</tr>
</tbody>
</table>

7-bit ASCII encoding

**7 \times 11 = 77 bits**

Variable Length Encoding

**Variable-length encoding.** Use different number of bits to encode different characters.

**Ex.** Morse code.

**Ambiguity.**
- • • • – – • • •
  - SOS
  - IAMIE
  - EEWNI
  - T70

7-bit ASCII encoding

<table>
<thead>
<tr>
<th>Letters</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>•••</td>
</tr>
<tr>
<td>C</td>
<td>••</td>
</tr>
<tr>
<td>D</td>
<td>•</td>
</tr>
<tr>
<td>E</td>
<td>-</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>•••</td>
</tr>
<tr>
<td>I</td>
<td>•••</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>•••</td>
</tr>
<tr>
<td>L</td>
<td>•••</td>
</tr>
<tr>
<td>M</td>
<td>•</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>•••</td>
</tr>
<tr>
<td>P</td>
<td>•••</td>
</tr>
<tr>
<td>Q</td>
<td>•••</td>
</tr>
<tr>
<td>R</td>
<td>•••</td>
</tr>
<tr>
<td>S</td>
<td>•••</td>
</tr>
<tr>
<td>T</td>
<td>•••</td>
</tr>
<tr>
<td>U</td>
<td>•••</td>
</tr>
<tr>
<td>V</td>
<td>•••</td>
</tr>
<tr>
<td>W</td>
<td>•••</td>
</tr>
<tr>
<td>X</td>
<td>•••</td>
</tr>
<tr>
<td>Y</td>
<td>•••</td>
</tr>
<tr>
<td>Z</td>
<td>•••</td>
</tr>
</tbody>
</table>

3-bit abracadabra encoding

<table>
<thead>
<tr>
<th>char</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>000</td>
</tr>
<tr>
<td>b</td>
<td>001</td>
</tr>
<tr>
<td>c</td>
<td>011</td>
</tr>
<tr>
<td>d</td>
<td>011</td>
</tr>
<tr>
<td>r</td>
<td>100</td>
</tr>
</tbody>
</table>

3 \times 11 = 33 bits

Variable Length Encoding

**Variable-length encoding.** Use different number of bits to encode different characters.

**Q.** How do we avoid ambiguity?
**A.**
- A1. Append special stop symbol to each codeword.
- A2. Ensure no encoding is a prefix of another.
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 corresponding to symbol, 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.
- Take left branch if bit is 0; right branch if 1.
- If leaf node, print symbol and return to root.

How to Transmit the Trie

How to transmit the trie?
- Send preorder traversal of trie.
  - we use * as sentinel for internal nodes
  - what if there is no sentinel?
- Send number of characters to decode.
- Send bits (packed 8 to the byte).

Prefix-Free Decoding Implementation

```java
public class HuffmanDecoder {
    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() {
        int N = StdIn.readInt();
        for (int i = 0; i < N; i++) {
            Node x = root;
            while (x.isInternal()) {
                char bit = StdIn.readChar();
                if (bit == '0') x = x.left;
                else if (bit == '1') x = x.right;
            }
            System.out.print(x.ch);
        }
    }
}
```

Huffman Coding

Q. How to construct a good prefix-free code?
    A. Huffman code. [David Huffman 1950]

To compute Huffman code:
* Count frequencies \( p_s \) for each symbol \( s \) in message.
* Start with a forest of trees, each consisting of a single node
  corresponding to each symbol \( s \) with weight \( p_s \).
* Repeat:
  - select two trees with min weight \( p_1 \) and \( p_2 \)
  - merge into single tree with weight \( p_1 + p_2 \)

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

Huffman Tree Implementation

```java
private class Node implements Comparable<Node> {
    char ch;
    int freq;
    Node left, right;
    // constructor
    Node(char ch, int freq, Node left, Node right) { ... }
    // print preorder traversal
    void preorder() {
        System.out.print(ch);
        if (!isInternal()) {
            left.preorder();
            right.preorder();
        }
    }
    // compare by frequency
    int compareTo(Node b) { return freq - b.freq; }
    ...}
```

Huffman Coding Example

<table>
<thead>
<tr>
<th>Char</th>
<th>Freq</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>125</td>
<td>110</td>
</tr>
<tr>
<td>T</td>
<td>93</td>
<td>011</td>
</tr>
<tr>
<td>A</td>
<td>80</td>
<td>000</td>
</tr>
<tr>
<td>O</td>
<td>76</td>
<td>001</td>
</tr>
<tr>
<td>I</td>
<td>73</td>
<td>1011</td>
</tr>
<tr>
<td>N</td>
<td>71</td>
<td>1010</td>
</tr>
<tr>
<td>S</td>
<td>65</td>
<td>1001</td>
</tr>
<tr>
<td>R</td>
<td>61</td>
<td>1000</td>
</tr>
<tr>
<td>H</td>
<td>55</td>
<td>1111</td>
</tr>
<tr>
<td>L</td>
<td>41</td>
<td>0111</td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>0100</td>
</tr>
<tr>
<td>C</td>
<td>31</td>
<td>11100</td>
</tr>
<tr>
<td>U</td>
<td>27</td>
<td>11110</td>
</tr>
<tr>
<td>Total</td>
<td>838</td>
<td>3.62</td>
</tr>
</tbody>
</table>

Huffman Encoding Implementation

```java
public HuffmanEncoder(String input) {
    private static int SYMBOLS = 128;  // alphabet size (ASCII)
    private Node root;

    // tabulate frequencies
    int[] freq = new int[SYMBOLS];
    for (int i = 0; i < input.length(); i++)
        freq[input.charAt(i)]++;

    // initialise priority queue with singleton elements
    MinPQ<Node> pq = new MinPQ<Node>();
    for (int i = 0; i < SYMBOLS; i++)
        if (freq[i] > 0)
            pq.insert(new Node((char) i, freq[i], null, null));

    // repeatedly merge two smallest trees
    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);
    }
}
```
Huffman Encoding

Theorem. [Huffman] Huffman coding is optimal prefix-free code.
Corollary. “Greed is good.”

Implementation.
- Pass 1: tabulate symbol frequencies and build trie
- Pass 2: encode file by traversing trie or lookup table
- Use heap for delete min and insert.
- \( O(M + N \log N) \).

\[ \begin{array}{|c|c|c|}
\hline
M & f & N \# \text{distinct symbols} \\
\hline
\end{array} \]

PQ implementation important if each symbol represents one English word or k-gram

Difficulties.
- Must have encoding (trie).
- Not optimal (unless block size grows to infinity!)

A Difficult File To Compress

One million pseudo-random characters (a–p)

A Difficult File To Compress

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

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

What Data Can Be Compressed?

Theorem. Impossible to losslessly compress all files.
Pf.
- Consider all 1,000 bit messages.
- \( 2^{1000} \) possible messages.
- Only \( 2^999 + 2^998 + \ldots + 1 \) can be encoded with \( \leq 999 \) bits.
- Only 1 in \( 2^{49} \) can even be encoded with \( \leq 500 \) bits.
Information Theory

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

Q. So how do we know if our algorithm is doing well?
A. Want lower bound on # bits required by any compression scheme.

Entropy

Entropy. [Shannon 1948] $H(S) = - \sum_{s \in S} p(s) \log_2 p(s)$
- Information content of symbol $s$ is proportional to $-\log_2 p(s)$.
- Weighted average of information content over all symbols.
- Interface between coding and model.

<table>
<thead>
<tr>
<th>p(o)</th>
<th>p(b)</th>
<th>H(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>Model 2</td>
<td>0.900</td>
<td>0.100</td>
</tr>
<tr>
<td>Model 3</td>
<td>0.990</td>
<td>0.010</td>
</tr>
<tr>
<td>Model 4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p(o)</th>
<th>p(r)</th>
<th>p(c)</th>
<th>p(d)</th>
<th>p(e)</th>
<th>p(f)</th>
<th>H(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fair die</td>
<td>1/6</td>
<td>1/6</td>
<td>1/6</td>
<td>1/6</td>
<td>1/6</td>
<td>2.585</td>
</tr>
</tbody>
</table>

Language Model

How compression algorithms work?
- Exploit biases of input messages.
- White patches occur in typical images.
- Word Princeton occurs more frequently than Yale.

Compression is all about 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: each symbol $s$ generated independently at random, with fixed probability $p(s)$.

Entropy and Compression

Theorem. [Shannon, 1948] If data source is an order 0 Markov model, any compression scheme must use $\geq H(S)$ bits per symbol on average.
- Cornerstone result of information theory.
  - Ex: to transmit results of fair die, need $\geq 2.58$ bits per roll.

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

Q. Is there any hope of doing better than Huffman coding?
A. Yes. Huffman wastes up to 1 bit per symbol.
  - if $H(S)$ is close to 0, this matters
  - can do better with "arithmetic coding"

A. 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 demtraroate. In a pubitiacon of New Scnieist you could randomise all the letters, kipeng the first two and last two the same, and reibadalty would hardly be ofctfeed. My ansaylis did not come to much beucase the theory at the time was for shape and senquence retigionon. Saberi's work sugasests we may have some pofrweal palralael prsoscers at work. The resion for this is suerly that identiyng coentnt by paarallel prsoscing speeds up regnicoiton. We only need the first and last two leteters to spot chganes in meniang."

A. Quite a bit.

Entropy of the English Language

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

Q. How can we measure it?

A. [Shannon’s 1951 experiment]

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

<table>
<thead>
<tr>
<th># of guesses</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>≥ 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction</td>
<td>0.79</td>
<td>0.08</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
<td>0.05</td>
</tr>
</tbody>
</table>

- Shannon’s estimate = 0.6 - 1.3.

Lossless Compression Ratio for Calgary Corpus

<table>
<thead>
<tr>
<th>Year</th>
<th>Scheme</th>
<th>Bits / char</th>
</tr>
</thead>
<tbody>
<tr>
<td>----</td>
<td>ASCII</td>
<td>7.00</td>
</tr>
<tr>
<td>1950</td>
<td>Huffman</td>
<td>4.70</td>
</tr>
<tr>
<td>1977</td>
<td>LZ77</td>
<td>3.94</td>
</tr>
<tr>
<td>1984</td>
<td>LZW</td>
<td>3.32</td>
</tr>
<tr>
<td>1987</td>
<td>LZH</td>
<td>3.30</td>
</tr>
<tr>
<td>1987</td>
<td>Move-to-front</td>
<td>3.24</td>
</tr>
<tr>
<td>1987</td>
<td>LZW</td>
<td>3.18</td>
</tr>
<tr>
<td>1987</td>
<td>Gzip</td>
<td>2.71</td>
</tr>
<tr>
<td>1988</td>
<td>PPMC</td>
<td>2.48</td>
</tr>
<tr>
<td>1988</td>
<td>SAKDC</td>
<td>2.47</td>
</tr>
<tr>
<td>1994</td>
<td>PPM</td>
<td>2.34</td>
</tr>
<tr>
<td>1995</td>
<td>Burrows-Wheeler</td>
<td>2.29</td>
</tr>
<tr>
<td>1997</td>
<td>BOA</td>
<td>1.99</td>
</tr>
<tr>
<td>1999</td>
<td>RK</td>
<td>1.89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entropy</th>
<th>Bits/char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char by char</td>
<td>4.5</td>
</tr>
<tr>
<td>8 chars at a time</td>
<td>2.4</td>
</tr>
<tr>
<td>Asymptotic</td>
<td>1.3</td>
</tr>
</tbody>
</table>

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]
- Associate an integer with each useful string.
- When input matches string in ST, output associated integer.

Encoding:
- Find longest string $s$ in ST that is a prefix of remaining part of string to compress.
- Output integer associated with $s$.
- Add $s \cdot x$ to dictionary, where $x$ is next char in string to compress.

Ex. Dictionary: $a, aa, ab, aba, abb, abaa, abaab, abaaa$.
- String to be compressed: $abaababb$.
- $s = abaab$, $x = a$.
- Output integer associated with $s$; insert $abaab$ into ST.

LZW Implementation

Implementation:
- Use trie to create symbol table on-the-fly.
- Note that prefix of every word is also in ST.

Encode:
- Lookup string suffix in trie.
- Output ST index at bottom.
- Add new node to bottom of trie.

Decode:
- Lookup index in array
- Output string
- Insert string + next letter.

LZW Encoder: Java Implementation

```java
public class LZWEncoder {
    public static void main(String[] args) {
        String text = StdIn.readAll();
        StringST<Integer> st = new StringST<Integer>();
        int i;
        for (i = 0; i < 256; i++) {
            String s = Character.toString((char) i);
            st.put(s, i);
        }
        while (text.length() > 0) {
            String s = st.prefix(text);
            System.out.println(st.get(s));
            int length = s.length();
            if (length < text.length())
                st.put(text.substring(0, length + 1), i++);
            text = text.substring(length);
        }
    }
}
```

LZW Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND</td>
<td>256</td>
</tr>
<tr>
<td>i</td>
<td>105</td>
</tr>
<tr>
<td>t</td>
<td>116</td>
</tr>
<tr>
<td>y</td>
<td>116</td>
</tr>
<tr>
<td>l</td>
<td>32</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>i</td>
<td>258</td>
</tr>
<tr>
<td>t</td>
<td>258</td>
</tr>
<tr>
<td>y</td>
<td>260</td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>262</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>258</td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>266</td>
</tr>
<tr>
<td>n</td>
<td>110</td>
</tr>
<tr>
<td>STOP</td>
<td>257</td>
</tr>
</tbody>
</table>

Dictionary

<table>
<thead>
<tr>
<th>Index</th>
<th>Word</th>
<th>Index</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>256</td>
<td>it</td>
<td></td>
</tr>
<tr>
<td></td>
<td>259</td>
<td>tt</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>_</td>
<td>ty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>261</td>
<td>y_</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td>263</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>264</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>z</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>SEND</td>
<td>268</td>
<td></td>
</tr>
<tr>
<td>257</td>
<td>STOP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LZW Decoder: Java Implementation

```java
public class LZWDecoder {
    public static void main(String[] args) {
        ST<Integer, String> st = new ST<Integer, String>();
        int i;
        for (i = 0; i < 256; i++) {
            String s = Character.toString((char) i);
            st.put(i, s);
        }
        int code = StdIn.readInt();
        String prev = st.get(code);
        System.out.print(prev);
        while (!StdIn.isEmpty()) {
            code = StdIn.readInt();
            String s = st.get(code);
            if (i == code) s = prev + prev.charAt(0);
            System.out.print(s);
            st.put(i++, prev + s.charAt(0)); // special case, e.g., for "ababababab"
            prev = s;
        }
    }
}
```

LZW in the Real World

**Lempel-Ziv and friends.**
- **LZ77.** LZ77 not patented — widely used in open source.
- **LZ78.** LZ78 patent #4,558,302 expired in US on June 20, 2003
- **LZW.** some versions copyrighted
- **Deflate** = LZ77 variant + Huffman

**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.

LZW Implementation Details

**What to do when ST gets too large?**
- Throw away and start over. **GIF**
- Throw away when not effective. **Unix compress**

**Lossless compression.**
- Simple approaches. **[RLE]**
- 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.

**Summary**

Theoretical limits closely match what we can achieve in practice!