5.5 DATA COMPRESSION

- introduction
- run-length coding
- Huffman compression
- LZW compression

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, ...

"Everyday, we create 2.5 quintillion bytes of data—so much that 90% of the data in the world today has been created in the last two years alone." — IBM report on big data (2011)

Applications

Generic file compression.
- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, ZFS, HFS+, ReFS, GFS.

Multimedia.
- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.

Communication.
- ITU-T T4 Group 3 Fax.
- V.42bis modem.
- Skype, Google hangout.

Databases. Google, Facebook, NSA, ....
Lossless compression and expansion

**Message.** Bitstream $B$ we want to compress.

**Compress.** Generates a “compressed” representation $C(B)$.

**Expand.** Reconstructs original bitstream $B$.

![Basic model for data compression](image)

**Compression ratio.** Bits in $C(B)$ / bits in $B$.

**Ex.** 50–75% or better compression ratio for natural language.

---

**Food for thought**

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

has played a central role in communications technology,

- Grade 2 Braille.
- Morse code.
- Telephone system.

and is part of modern life.

- MP3.
- MPEG.

Q. What role will it play in the future?

---

**Data representation: genomic code**

**Genome.** String over the alphabet \{ A, C, T, G \}.


**Standard ASCII encoding genome.**

- 8 bits per char.
- $8N$ bits.

**Two-bit encoding.**

- 2 bits per char.
- $2N$ bits (25% compression ratio).

<table>
<thead>
<tr>
<th>char</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>41</td>
<td>01000001</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
<td>01000011</td>
</tr>
<tr>
<td>T</td>
<td>54</td>
<td>01010100</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
<td>01000111</td>
</tr>
</tbody>
</table>

**Fixed-length code.** $k$-bit code supports alphabet of size $2^k$.

**Amazing but true.** Some genomic databases in 1990s used ASCII.

---

**Reading and writing binary data**

**Binary standard input.** Read bits from standard input.

```java
public class BinaryStdIn

boolean readBoolean() read 1 bit of data and return as a boolean value
char readChar() read 8 bits of data and return as a char value
char readChar(int r) read r bits of data and return as a char value
[similar methods for byte (8 bits), short (16 bits), int (32 bits), long and double (64 bits)]
boolean isEmpty() is the bitstream empty?
void close() close the bitstream
```

**Binary standard output.** Write bits to standard output

```java
public class BinaryStdOut

void write(boolean b) write the specified bit
void write(char c) write the specified 8-bit char
void write(char c, int r) write the r least significant bits of the specified char
[similar methods for byte (8 bits), short (16 bits), int (32 bits), long and double (64 bits)]
void close() close the bitstream
```
Writing binary data

Date representation. Three different ways to represent 12/31/1999.

A character stream (StdOut)

```java
StdOut.println(month + "/" + day + "/" + year);
```

Three ints (BinaryStdOut)

```java
BinaryStdOut.write(month);
BinaryStdOut.write(day);
BinaryStdOut.write(year);
```

A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```java
BinaryStdOut.write(month, 4);
BinaryStdOut.write(day, 5);
BinaryStdOut.write(year, 12);
```

Universal data compression

ZeoSync. Announced 100:1 lossless compression of random data using Zero Space Tuner™ and BinaryAccelerator™ technology.

Universal data compression

Proposition. No algorithm can compress every bitstring.

Pf 1. [by contradiction]
- Suppose you have a universal data compression algorithm \( U \) that can compress every bitstring.
- Given bitstring \( B_0 \), compress it to get smaller bitstring \( B_1 \).
- Compress \( B_1 \) to get a smaller bitstring \( B_2 \).
- Continue until reaching bitstring of size 0.
- Implication: all bitstrings can be compressed to 0 bits!

Pf 2. [by counting]
- Suppose your algorithm that can compress all 1,000-bit strings.
- \( 2^{1000} \) possible bitstrings with 1,000 bits.
- Only \( 1 + 2 + 4 + \ldots + 2^{999} + 2^{999} \) can be encoded with \( \leq 999 \) bits.
- Similarly, only 1 in \( 2^{999} \) bitstrings can be encoded with \( \leq 500 \) bits!
Undecidability

A difficult file to compress: one million (pseudo-) random bits

```java
public class RandomBits {
    public static void main(String[] args) {
        int x = 11111;
        for (int i = 0; i < 1000000; i++) {
            x = x * 314159 + 218281;
            BinaryStdOut.write(x > 0);
        }
        BinaryStdOut.close();
    }
}
```

Rdenudcany in Enlgsh Inagugae

Q. How much redundancy in the English language?
A. Quite a bit.

“... 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 demntrasote. In a pubilication of New Scnieitst you could randmide all the letetrs, keipeng the first two and last two the same, and reibadaitly would hardly be afeefed. My ansaylis did not come to much beucase the theory at the time was for shape and seenque retigicon. Saberi's work sugsegts we may have some pofrweul palrlael prsooscers at work. The resaon for this is suerly that identityfing coenntnt by paarllel prseocsing speeds up regniciton. We only need the first and last two letetrs to spot chganes in meniang. ” — Graham Rawlinson

The gaol of data cmperisosn is to inetidy rdenudcany and exploit it.

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

```
0000000000000111111100000001111111111
```

40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s:

```
111101111011110111
15 7 7 11
```

16 bits (instead of 40)

Q. How many bits to store the counts?
A. We typically use 8 (but 4 in the example above).

Q. What to do when run length exceeds max count?
A. Intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

5.5 DATA COMPRESSION

- introduction
- run-length coding
- Huffman compression
- LZW compression

http://algs.cs.princeton.edu
Run-length encoding: Java implementation

```java
public class RunLength {
    private final static int R = 256;
    private final static int lgR = 8;

    public static void compress() {
        /* see textbook */
    }

    public static void expand() {
        boolean bit = false;
        while (!BinaryStdIn.isEmpty()) {
            int run = BinaryStdIn.readInt(lgR);
            for (int i = 0; i < run; i++)
                BinaryStdOut.write(bit);
            bit = !bit;
        }
        BinaryStdOut.close();
    }
}
```

Maximum run-length count:
- number of bits per count
- read 8-bit count from standard input
- write 1 bit to standard output
- pad 0s for byte alignment

Data compression: quiz 1

What is the best compression ratio achievable from run-length coding using 8-bit counts?

- A. 1 / 256
- B. 1 / 16
- C. 8 / 255
- D. 24 / 510 = 4 / 85
- E. I don’t know.

Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: • • • – – – • •

**Issue.** Ambiguity.

SOS?  
V?  
IAMIE?  
EEWNI?

**In practice.** Use a medium gap to separate codewords.

Codeword for S is a prefix of codeword for V

5.5 Data Compression

- introduction
- run-length coding
- Huffman compression
- LZW compression
Variable-length codes

Q. How do we avoid ambiguity?
A. Ensure that no codeword is a prefix of another.

Ex 1. Fixed-length code.
Ex 2. Append special stop char to each codeword.
Ex 3. General prefix-free code.

Prefix-free codes: trie representation

Q. How to represent the prefix-free code?
A. A binary trie!
   • Chars in leaves.
   • Codeword is path from root to leaf.

Prefix-free codes: compression

Compression.
   • Method 1: start at leaf; follow path up to the root; print bits in reverse.
   • Method 2: create ST of key-value pairs.

Prefix-free codes: expansion

Expansion.
   • Start at root.
   • Go left if bit is 0; go right if 1.
   • If leaf node, print char and return to root.
Consider the following trie representation of a prefix-free code. Which string is encoded by the compressed bit string 100101000111011?

A. PEED
B. PESDEY
C. SPED
D. SPEEED
E. I don’t know.

Huffman trie node data type

```java
private static class Node implements Comparable<Node> {
    private final char ch; // used only for leaf nodes
    private final int freq; // used only by compress()
    private final Node left, right;

    public Node(char ch, int freq, Node left, Node right) {
        this.ch = ch;
        this.freq = freq;
        this.left = left;
        this.right = right;
    }

    public boolean isLeaf() {
        return left == null && right == null;
    }

    public int compareTo(Node that) {
        return this.freq - that.freq;
    }
}
```

Prefix-free codes: expansion

```java
public void expand() {
    Node root = readTrie();
    int N = BinaryStdIn.readInt();

    for (int i = 0; i < N; i++) {
        Node x = root;
        while (!x.isLeaf()) {
            if (BinaryStdIn.readBoolean())
                x = x.left;
            else
                x = x.right;
        }
        BinaryStdOut.write(x.ch, 8);
    }
    BinaryStdOut.close();
}
```

Running time. Linear in input size $N$. 

Data compression: quiz 2

Dynamic model. Use a custom prefix-free code for each message.

Compression.
- Read message.
- Built best prefix-free code for message. How?
- Write prefix-free code (as a trie) to file.
- Compress message using prefix-free code.

Expansion.
- Read prefix-free code (as a trie) from file.
- Read compressed message and expand using trie.
Prefix-free codes: how to transmit

Q. How to write the trie?
A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.

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

Prefix-free codes: how to transmit

Q. How to read in the trie?
A. Reconstruct from preorder traversal of trie.

Huffman codes

Q. How to find best prefix-free code?

Huffman algorithm:

- Count frequency \( \text{freq}[i] \) for each char \( i \) in input.
- Start with one node corresponding to each char \( i \) (with weight \( \text{freq}[i] \)).
- Repeat until single trie formed:
  - select two tries with min weight \( \text{freq}[i] \) and \( \text{freq}[j] \)
  - merge into single trie with weight \( \text{freq}[i] + \text{freq}[j] \)

Applications:

Huffman algorithm demo

- Count frequency for each character in input.

input

\[ A \ B \ R \ A \ C \ A \ D \ A \ B \ R \ A \ ! \]
Huffman encoding summary

**Proposition.** [Huffman 1950s] Huffman’s algorithm produces an optimal prefix-free code.

**Pf.** See textbook.

**Implementation.**
- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

**Running time.** Using a binary heap $\Rightarrow N + R \log R$.

Q. Can we do better? [stay tuned]
**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.

**Lempel-Ziv-Welch compression**

**LZW compression.**
- Create ST associating $M$-bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string $s$ in ST that is a prefix of unscanned part of input.
- Write the $M$-bit codeword associated with $s$.
- Add $s+c$ to ST, where $c$ is next char in the input.

**Q.** How to represent LZW compression code table?

**A.** A trie to support longest prefix match.

---

**LZW compression demo**

| input  | A | B | R | A | C | A | D | A | B | R | A | B | R | A | B | R | A | B | R | A |
| matches| A | B | R | A | C | A | D | A | B | R | A | B | R | A | B | R | A | B | R | A |
| value  | 41 | 42 | 52 | 41 | 43 | 41 | 44 | 81 | 83 | 82 | 88 | 41 | 80 |

**LZW expansion demo**

<table>
<thead>
<tr>
<th>value</th>
<th>41</th>
<th>42</th>
<th>52</th>
<th>41</th>
<th>43</th>
<th>41</th>
<th>44</th>
<th>81</th>
<th>83</th>
<th>82</th>
<th>88</th>
<th>41</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>B</td>
<td>R</td>
</tr>
</tbody>
</table>

---

**LZW compression for A B R A C A D A B R A B R A B R A**

**codeword table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

**LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80**

**codeword table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

---

**LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80**

**codeword table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

---
LZW expansion

**LZW expansion.**
- Create ST associating string values with \(W\)-bit keys.
- Initialize ST to contain single-char values.
- Read a \(W\)-bit key.
- Find associated string value in ST and write it out.
- Update ST.

**Q.** How to represent LZW expansion code table?

**A.** An array of size \(2^W\).

---

**LZW tricky case: compression**

<table>
<thead>
<tr>
<th>input</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>value</td>
<td>41</td>
<td>42</td>
<td>81</td>
<td>83</td>
<td>80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LZW tricky case: expansion**

<table>
<thead>
<tr>
<th>value</th>
<th>41</th>
<th>42</th>
<th>81</th>
<th>83</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

**LZW tricky case: expansion**

**LZW tricky case: expansion**

- Need to know code for 83 before it is in ST!
- We can deduce that the code for 83 is ABx for some character x.
- Now we have deduced x!

---

**LZW implementation details**

**How big to make ST?**
- How long is message?
- Whole message similar model?
  - [many other variations]

**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 / zlib = LZ77 variant + Huffman.

Unix compress, GIF, TIFF, V.42bis modem: LZW.
zip, 7zip, gzip, jar, png, pdf: deflate / zlib.
iPhone, Wii, Apache HTTP server: deflate / zlib.

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, ...

**Theoretical limits on compression.** Shannon entropy: \( H(X) = - \sum p(x_i) \log p(x_i) \)

**Practical compression.** Exploit extra knowledge whenever possible.

---

Lossless data compression benchmarks

<table>
<thead>
<tr>
<th>Year</th>
<th>Scheme</th>
<th>Bits / Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>1967</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>1994</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>

data compression using Calgary corpus

---

next programming assignment