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)

Basic concepts ancient (1950s), best technology recently developed.

Applications

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

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

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

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

**Message.** Binary data $B$ we want to compress.
**Compress.** Generates a "compressed" representation $C(B)$.
**Expand.** Reconstructs original bitstream $B$.

[Diagram: Basic model for data compression]

<table>
<thead>
<tr>
<th>bitstream $B$</th>
<th>Compress</th>
<th>compressed version $C(B)$</th>
<th>Expand</th>
<th>original bitstream $B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110110101...</td>
<td></td>
<td></td>
<td></td>
<td>0110110101...</td>
</tr>
</tbody>
</table>

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

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

Data representation: genomic code

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

**Goal.** Encode an $N$-character genome: ATAGATGCATAG...

**Standard ASCII encoding.**
- 8 bits per char.
- 8 $N$ bits.

<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>01001111</td>
</tr>
</tbody>
</table>

**Two-bit encoding.**
- 2 bits per char.
- 2 $N$ bits.

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
</tr>
<tr>
<td>T</td>
<td>10</td>
</tr>
<tr>
<td>G</td>
<td>11</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.

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?

Reading and writing binary data

**Binary standard input and standard output.** Libraries to read and write bits from standard input and to standard output.

```java
public class BinaryStdIn {
    public boolean readBoolean() {
        // read 1 bit of data and return as a boolean value
    }
    public char readChar() {
        // read 8 bits of data and return as a char value
    }
    public 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)
    public boolean isEmpty() {
        // is the bitstream empty?
    }
    public void close() {
        // close the bitstream
    }
}

public class BinaryStdOut {
    public void write(boolean b) {
        // write the specified bit
    }
    public void write(char c) {
        // write the specified 8-bit char
    }
    public 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)
    public void close() {
        // close the bitstream
    }
}
```
Writing binary data

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

A character stream (StdOut)

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

Three ints (BinaryStdOut)

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

US Patent 5,533,051 on "Methods for Data Compression", which is capable of compression all files.

Slashdot reports of the Zero Space Tuner™ and BinaryAccelerator™.

"ZeoSync has announced a breakthrough in data compression that allows for 100:1 lossless compression of random data. If this is true, our bandwidth problems just got a lot smaller.... "

Binary dumps

Q. How to examine the contents of a bitstream?

Standard character stream

```java
% more abra.txt
ABRACADABRA!
```

Bitstream represented as 0 and 1 characters

```java
% java BinaryDump 16 < abra.txt
0100000101000010
0100010101000001
0100011001000000
0100000100000001
0100000100000000
0000000000000000
96 bits
```

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 bitstream.
- Given bitstring $B_i$, compress it to get smaller bitstring $B_i$.
- Compress $B_i$ 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 RandomBits | java PictureDump 2000 500
```

```
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 Inaguag

Q. How much rdenudcany is in the Enlgsh Inaguag?

“... 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 demonstrate. In a publication of New Scnieitst you could ramdinose all the letetrs, keipeng the first two and last two the same, and reibadaility would hadrly be aficfeed. My ansaylis did not come to much heucase the thoery at the time was for shape and senqence retigcion. Saberi’s work sugsegts we may have some pofrweul paarlal prsooscers at work. The resaon for this is suery that idnetifyng coentnt by paarlal prseocsing speeds up regnicoito. We only need the first and last two letetrs to spot chganes in meniag.” — Graham Rawlinson

A. Quite a bit.

Run-length encoding

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

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
```

40 bits

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

15 0s, then 7 1s, then 7 0s, then 11 1s.

```
1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1
```

16 bits (instead of 40)

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

Q. What to do when run length exceeds max count?
A. If longer than 255, intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...
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();
    }
}
```

Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: ••• – – – •

Issue. Ambiguity.

SOS? V? IAMIE? EENWI?

In practice. Use a medium gap to separate codewords.

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.

Huffman coding overview

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.

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 for 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;
  }
}
```

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

Expansion.
  - Start at root.
  - Go left if bit is 0; go right if 1.
  - If leaf node, print char and return to root.
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 $N$.

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.

```java
private static void writeTrie(Node x)
{
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

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

Shannon-Fano codes

Q. How to find best prefix-free code?

Shannon-Fano algorithm:
- Partition symbols $S$ into two subsets $S_0$ and $S_1$ of (roughly) equal freq.
- Codewords for symbols in $S_0$ start with 0; for symbols in $S_1$ start with 1.
- Recur in $S_0$ and $S_1$.

<table>
<thead>
<tr>
<th>char</th>
<th>freq</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>0...</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0...</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>1...</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1...</td>
</tr>
<tr>
<td>R</td>
<td>2</td>
<td>1...</td>
</tr>
<tr>
<td>!</td>
<td>1</td>
<td>1...</td>
</tr>
</tbody>
</table>

$S_0 = \text{codewords starting with } 0$
$S_1 = \text{codewords starting with } 1$

Problem 1. How to divide up symbols?
Problem 2. Not optimal!
Huffman algorithm demo

- Count frequency for each character in input.

<table>
<thead>
<tr>
<th>char</th>
<th>freq</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

input

A B R A C A D A B R A !

Huffman codes

Q. How to find best prefix-free code?

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

Applications:

Constructing a Huffman encoding trie: Java implementation

```java
private static Node buildTrie(int[] freq)
{
    MinPQ<Node> pq = new MinPQ<Node>();
    for (char i = 0; i < R; i++)
        if (freq[i] > 0)
            pq.insert(new Node(i, freq[i], null, null));

    while (pq.size() > 1)
    {
        Node x = pq.delMin();
        Node y = pq.delMin();
        Node parent = new Node(‘\0’, x.freq + y.freq, x, y);
        pq.insert(parent);
    }

    return pq.delMin();
}
```
Huffman encoding summary

**Proposition.** [Huffman 1950s] Huffman 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.

LZW compression demo

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

codeword table

LZW compression demo

5.5 Data Compression

- introduction
- run-length coding
- Huffman compression
- LZW compression
Lempel-Ziv-Welch compression

LZW compression.
- Create ST associating $W$-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 $W$-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 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

LZW expansion demo

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

LZW compression for ABABABA
LZW tricky case: expansion

LZW expansion for 41 42 81 83 80

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, Sony Playstation 3, Apache HTTP server: deflate / zlib.
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>LZB</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 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: \( H(X) = - \sum_{i} p(x_i) \ln p(x_i) \)

Practical compression. Use extra knowledge whenever possible.