Data Compression

Introduction

Reasons

- save space when storing
- save time when transmitting

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, ...

Lossless compression and expansion

- message---binary data B we want to compress
- compress---generate a "compressed" representation $c(B)$
- expand---reconstructs original bitstream B
- compression ratio---Bits in $C(B)$/bits in B

Reading and writing binary data

- BinaryStdIn

`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*

- BinaryStdOut

`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*

Universal data compression

- No algorithm can compress every bitstream
  - proof by contradiction
    - Suppose you have a universal data compression algorithm U that can compress every bitstream.
    - 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!
  - proof by counting
    - Suppose your algorithm that can compress all 1,000-bit strings.
    - 2^{1000} possible bitstrings with 1,000 bits.
    - Only $1 + 2 + 4 + \ldots + 2^{998} + 2^{999}$ can be encoded with ≤ 999 bits.
    - Similarly, only 1 in 2^{499} bitstrings can be encoded with ≤ 500 bits!
- Undecidability---Impossible to find the best compression algorithm

Run-length Coding

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

    // Do not instantiate.
    private RunLength() { }

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

    public static void compress() {
        char run = 0;
        boolean old = false;
        while (!BinaryStdIn.isEmpty()) {
            boolean b = BinaryStdIn.readBoolean();
            if (b != old) {
                BinaryStdOut.write(run, LG_R);
                run = 1;
```

```
                    old = !old;
                }
                else {
                    if (run == R-1) {
                        BinaryStdOut.write(run, LG_R);
                        run = 0;
                        BinaryStdOut.write(run, LG_R);
                    }
                    run++;
                }
            }
            BinaryStdOut.write(run, LG_R);
            BinaryStdOut.close();
        }

        public static void main(String[] args) {
            if      (args[0].equals("-")) compress();
            else if (args[0].equals("+")) expand();
            else throw new IllegalArgumentException("Illegal command line argument");
        }

    }
```

Huffman Compression

> produce an optimal prefix-free code

Variable-length Codes

- How to avoid ambiguity?
  - Ensure that no codeword is a prefix of another
    - fixed-length code
    - append special stop char to each codeword
    - general prefix-free code
- How to represent the pre-fix code?
  - A binary trie!
    - Chars in leaves
    - Codewords is path from root to leaf

Shannon-Fano Codes

> How to find best prefix-free code?

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

- Return in S_0 and S_1

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 mim weight freq$[i]$ and freq$[j]$
  - merge into single trie with weight freq$[i]$+freq$[j]$

Implementation

Huffman Trie

```java
    private static class Node implements Comparable<Node> {
        private final char ch;
        private final int freq;
        private final Node left, right;

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

        // is the node a leaf node?
        private boolean isLeaf() {
            assert ((left == null) && (right == null)) || ((left != null) && (right != null));
            return (left == null) && (right == null);
        }

        // compare, based on frequency
        public int compareTo(Node that) {
            return this.freq - that.freq;
        }
    }
```

Build Trie

```java
    private static Node buildTrie(int[] freq) {

        // initialize priority queue with singleton trees
        MinPQ<Node> pq = new MinPQ<Node>();
        for (char c = 0; c < R; c++)
            if (freq[c] > 0)
                pq.insert(new Node(c, freq[c], null, null));

```

```
        // merge two smallest trees
        while (pq.size() > 1) {
            Node left  = pq.delMin();
            Node right = pq.delMin();
            Node parent = new Node('\0', left.freq + right.freq, left, right);
            pq.insert(parent);
        }
        return pq.delMin();
    }
```

Expansion

> Running time---linear in input size N

```
    public static void expand() {

        // read in Huffman trie from input stream
        Node root = readTrie();

        // number of bytes to write
        int length = BinaryStdIn.readInt();

        // decode using the Huffman trie
        for (int i = 0; i < length; i++) {
            Node x = root;
            while (!x.isLeaf()) {
                boolean bit = BinaryStdIn.readBoolean();
                if (bit) x = x.right;
                else     x = x.left;
            }
            BinaryStdOut.write(x.ch, 8);
        }
        BinaryStdOut.close();
    }
```

Complete Code

```
public class Huffman {

    // alphabet size of extended ASCII
    private static final int R = 256;

    // Do not instantiate.
    private Huffman() { }

    // Huffman trie node
    private static class Node implements Comparable<Node> {}

    public static void compress() {
        // read the input
        String s = BinaryStdIn.readString();
```

```
        char[] input = s.toCharArray();

        // tabulate frequency counts
        int[] freq = new int[R];
        for (int i = 0; i < input.length; i++)
            freq[input[i]]++;

        // build Huffman trie
        Node root = buildTrie(freq);

        // build code table
        String[] st = new String[R];
        buildCode(st, root, "");

        // print trie for decoder
        writeTrie(root);

        // print number of bytes in original uncompressed message
        BinaryStdOut.write(input.length);

        // use Huffman code to encode input
        for (int i = 0; i < input.length; i++) {
            String code = st[input[i]];
            for (int j = 0; j < code.length(); j++) {
                if (code.charAt(j) == '0') {
                    BinaryStdOut.write(false);
                }
                else if (code.charAt(j) == '1') {
                    BinaryStdOut.write(true);
                }
                else throw new IllegalStateException("Illegal state");
            }
        }

        // close output stream
        BinaryStdOut.close();
    }

    // build the Huffman trie given frequencies
    private static Node buildTrie(int[] freq) {}

    // write bitstring-encoded trie to standard output
    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);
    }
```

```java

    // make a lookup table from symbols and their encodings
    private static void buildCode(String[] st, Node x, String s) {
        if (!x.isLeaf()) {
            buildCode(st, x.left,  s + '0');
            buildCode(st, x.right, s + '1');
        }
        else {
            st[x.ch] = s;
        }
    }

    public static void expand() {}

    private static Node readTrie() {
        boolean isLeaf = BinaryStdIn.readBoolean();
        if (isLeaf) {
            return new Node(BinaryStdIn.readChar(), -1, null, null);
        }
        else {
            return new Node('\0', -1, readTrie(), readTrie());
        }
    }

    public static void main(String[] args) {
        if      (args[0].equals("-")) compress();
        else if (args[0].equals("+")) expand();
        else throw new IllegalArgumentException("Illegal command line argument");
    }

}
```

LZW Compression

> Lempel-Ziv-Welch compression

Steps for 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

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

Compression

```java
    public static void compress() {
        String input = BinaryStdIn.readString();
        TST<Integer> st = new TST<Integer>();

        // since TST is not balanced, it'd be better to insert in a different order
        for (int i = 0; i < R; i++)
            st.put("" + (char) i, i);

        int code = R+1;  // R is codeword for EOF

        while (input.length() > 0) {
            String s = st.longestPrefixOf(input);  // Find max prefix match s.
            BinaryStdOut.write(st.get(s), W);      // Print s's encoding.
            int t = s.length();
            if (t < input.length() && code < L)    // Add s to symbol table.
                st.put(input.substring(0, t + 1), code++);
            input = input.substring(t);            // Scan past s in input.
        }
        BinaryStdOut.write(R, W);
        BinaryStdOut.close();
    }
```

Expansion

```
    public static void expand() {
        String[] st = new String[L];
        int i; // next available codeword value

        // initialize symbol table with all 1-character strings
        for (i = 0; i < R; i++)
            st[i] = "" + (char) i;
        st[i++] = "";                        // (unused) lookahead for EOF

        int codeword = BinaryStdIn.readInt(W);
        if (codeword == R) return;           // expanded message is empty string
        String val = st[codeword];

        while (true) {
            BinaryStdOut.write(val);
            codeword = BinaryStdIn.readInt(W);
            if (codeword == R) break;
            String s = st[codeword];
            if (i == codeword) s = val + val.charAt(0);   // special case hack
            if (i < L) st[i++] = val + s.charAt(0);
            val = s;
        }
        BinaryStdOut.close();
    }
```

Complete Implementation

```
public class LZW {
    private static final int R = 256;        // number of input chars
    private static final int L = 4096;       // number of codewords = 2^W
    private static final int W = 12;         // codeword width

    // Do not instantiate.
    private LZW() { }

    public static void compress() {}

    public static void expand() {}

    public static void main(String[] args) {
        if      (args[0].equals("-")) compress();
        else if (args[0].equals("+")) expand();
        else throw new IllegalArgumentException("Illegal command line argument");
    }
}
```

Other versions

- LZ77
- LZ78
- Deflate/zlib = LZ77+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

year	scheme	bits / char
1967	ASCII	7.00
1950	Huffman	4.70
1977	LZ77	3.94
1984	LZMW	3.32
1987	LZH	3.30
1987	move-to-front	3.24
1987	LZB	3.18
1987	gzip	2.71
1988	PPMC	2.48
1994	SAKDC	2.47
1994	PPM	2.34
1995	Burrows-Wheeler	2.29
1997	BOA	1.99
1999	RK	1.89

Summary

- Lossless compression
  - Huffman---represent fixed-length symbols with variant-length codes
  - LZW---represent variable-lengh symbols with fixed-length codes
- Lossy compression
  - JPEG, MPEG, MP3,...
  - FFT, wavelets, fractals,...