Compression

Prepared by Mark on April 26, 2024

Instructor's Handout

This file contains solutions and notes.
Compile with the "nosolutions" flag before distributing.
Click [here] for the latest version of this handout.

Part 1: Introduction

Definition 1:
An *alphabet* is a set of symbols. Two examples are $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ and $\{\texttt{0}, \texttt{1}\}$.

Definition 2:
A *string* is a sequence of symbols from an alphabet.
For example, `CBCAADDD` is a string over the alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$.

Problem 3:
Say we want to store a length-n string over the alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ as a binary sequence.
How many bits will we need?
Hint: Our alphabet has four symbols, so we can encode each symbol using two bits, mapping $\texttt{A} \to \texttt{00}$, $\texttt{B} \to \texttt{01}$, $\texttt{C} \to \texttt{10}$, and $\texttt{D} \to \texttt{11}$.

Solution

$2n$ bits.

Problem 4:
Similarly, we can encode an n-symbol string over an alphabet of size k using $n \times \lceil \log_2 k \rceil$ bits. Show that this is true.
Note: We'll call this the *naïve coding scheme*.

As you might expect, this isn't ideal: we can do much better than $n \times \lceil \log_2 k \rceil$. We will spend the rest of this handout exploring more efficient ways of encoding such sequences of symbols.

Part 2: Run-length Coding

Problem 5:
Using the naïve coding scheme, encode `AAAA·AAAA·BCD·AAAA·AAAA` in binary.
Note: We're still using the four-symbol alphabet {`A`, `B`, `C`, `D`}.
Dots (·) in the string are drawn for readability. Ignore them.

Solution

There are eight `A`s on each end of that string. Mapping symbols as before,
we get `[00 00 00 00 00 00 00 00 01 10 11 00 00 00 00 00 00 00 00]`

Note for Instructors

In this handout, all encoded binary is written in square brackets.
Spaces, dashes, dots, and etc are added for readability, and should be ignored.

In Problem 5—and often, in the real world—the strings we want to encode have fairly low *entropy*.
That is, they have predictable patterns, sequences of symbols that don't contain a lot of information.
For example, consider the text in this document.
The symbols `e`, `t`, and `<space>` are much more common than any others.
Also, certain subsequences are repeated: `th`, `and`, `encode`, and so on.
We can exploit this fact to develop encoding schemes that need relatively few bits per letter.

Example 6:
A simple example of such a coding scheme is *run-length encoding*. Instead of simply listing letters of a string in their binary form, we'll add a *count* to each letter, shortening repeated instances of the same symbol.

We'll encode our string into a sequence of 6-bit blocks, interpreted as follows:

Bits	0	0	1	1	0	1
Meaning	number of copies				symbol	

So, the sequence `BBB` will be encoded as `[0011-01]`.
Notation: Just like dots, dashes and spaces are added for readability. Pretend they don't exist.
Encoded binary sequences will always be written in square brackets. `[]`.

Problem 7:
Decode `[010000001111]` using this scheme.

Solution

`AAAADDD`

Problem 8:
Encode `AAAA·AAAA·BCD·AAAA·AAAA` using this scheme.
Is this more or less efficient than Problem 5?

Solution

`[1000-00 0001-01 0001-10 0001-11 1000-00]`
This requires 30 bits, as compared to 38 in Problem 5.

Problem 9:
Give an example of a message on $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ that uses n bits when encoded with a naïve scheme, and *fewer* than $n/2$ bits when encoded using the scheme described on the previous page.

Problem 10:
Give an example of a message on $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ that uses n bits when encoded with a naïve scheme, and *more* than $2n$ bits when encoded using the scheme described on the previous page.

Problem 11:
Is run-length coding always more efficient than naïve coding?
When does it work well, and when does it fail?

Problem 12:
Our coding scheme wastes a lot of space when our string has few runs of the same symbol.
Fix this problem: modify the scheme so that single occurrences of symbols do not waste space.
Hint: We don't need a run length for every symbol. We only need one for *repeated* symbols.

Solution

One idea is as follows:

- Encode single symbols naïvely: `ABCD` becomes `[00 01 10 11]`
- Signal runs using two copies of the same symbol: `AAAAAA` becomes `[00 00 0110]`.
  When our decoder sees two copies of the same symbol, it will interpret the next four bits as a run length.

`BDC·DDDDD·AADBDC` will be encoded as `[01 11 10 11-11-0101 01-01-0010 11 01 11 10]`.

Problem 13:
Consider the following string: `ABCD·ABCD·BABABA·ABCD·ABCD`.

- How many bits do we need to encode this naïvely?
- How about with the (unmodified) run-length scheme described on the previous page?

Hint: You don't need to encode this string—just find the length of its encoded form.

Solution

Naïvely: 22 bits
Run-length: $6 \times 21 = 126$ bits. Watch out for the two repeated `A`s!

Neither solution to Problem 13 is ideal. Run-length is very wasteful due to the lack of runs, and naïve coding does not take advantage of repetition in the string. We'll need a better coding scheme.

Part 3: LZ Codes

The LZ-family[1] of codes (LZ77, LZ78, LZSS, LZMA, and others) take advantage of repeated subsequences in a string. They are the basis of most modern compression algorithms, including DEFLATE, which is used in the ZIP, PNG, and GZIP formats.

The idea behind LZ is to represent repeated substrings as *pointers* to previous parts of the string. Pointers take the form <pos, len>, where pos is the position of the string to repeat and len is the number of symbols to copy.

For example, we can encode the string ABRACADABRA as [ABRACAD<7, 4>].
The pointer <7, 4> tells us to look back 7 positions (to the first A), and copy the next 4 symbols.
Note that pointers refer to the partially decoded output—*not* to the encoded string.
This allows pointers to reference other pointers, and ensures that codes like A<1,9> are valid.
For example, [B<1,2>] decodes to BBB.

Problem 14:
Encode ABCD·ABCD·BABABA·ABCD·ABCD using this scheme.
Then, decode the following:

- [ABCD<4,4>]
- [A<1,9>]
- [DAC<3,5>]

Solution

ABCD·ABCD·BABABA·ABCD·ABCD becomes [ABCD<4, 4> BA<2,4> ABCD<4,4>].

In parts two and three, remember that we're reading the *output string.*
The ten As in part two are produced one by one,
with the decoder's "read head" following its "write head."

- ABCD·ABCD
- AAAAA·AAAAA
- DACDACDA

Problem 15:
Convince yourself that LZ is a generalization of the run-length code we discussed in the previous section. *Hint:* [A<1,9>] and [00-1001] are the same thing!

Remark 16:
Note that we left a few things out of this section: we didn't discuss the algorithm that converts a string to an LZ-encoded blob, nor did we discuss how we should represent strings encoded with LZ in binary. We skipped these details because they are problems of implementation—they're the engineer's headache, not the mathematician's.

---

[1] Named after Abraham Lempel and Jacob Ziv, the original inventors.
LZ77 is the algorithm described in their first paper on the topic, which was published in 1977.
LZ78, LZSS, and LZMA are minor variations on the same general idea.

Part 4: Huffman Codes

Example 17:
Now consider the alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}\}$.
With the naïve coding scheme, we can encode a length n string with $3n$ bits, by mapping...

- `A` to `000`
- `B` to `001`
- `C` to `010`
- `D` to `011`
- `E` to `100`

For example, this encodes `ADEBCE` as `[000 011 100 001 010 100]`.
It is easy to see that this scheme uses an average of three bits per symbol.

However, one could argue that this coding scheme is wasteful:
we're not using three of the eight possible three-bit sequences!

Example 18:
There is, of course, a better way.
Consider the following mapping:

- `A` to `00`
- `B` to `01`
- `C` to `10`
- `D` to `110`
- `E` to `111`

Problem 19:

- Using the above code, encode `ADEBCE`.
- Then, decode `[110011001111]`.

Solution

`ADEBCE` becomes `[00 110 111 01 10 111]`,
and `[110 01 10 01 111]` is `DBCBE`.

Problem 20:
How many bits does this code need per symbol, on average?

Solution

$$\frac{2+2+2+3+3}{5} = \frac{12}{5} = 2.4$$

Problem 21:
Consider the code below. How is it different from the one on the previous page?
Is this a good way to encode five-letter strings?

- `A` to `00`
- `B` to `01`
- `C` to `10`
- `D` to `110`
- `E` to `11`

Solution

No. The code for `E` occurs inside the code for `D`, and we thus can't decode sequences uniquely.
For example, we could decode the fragment `[11001···]` as `EA` or as `DB`.

Remark 22:
The code from the previous page can be visualized as a full binary tree:
Every node in a *full binary tree* has either zero or two children.

- A encodes as 00
- B encodes as 01
- C encodes as 10
- D encodes as 110
- E encodes as 111

```
          / \
         0   1
        /     \
      /\       /\
     0  1     0  1
    /    \   /    \
   A      B C     /\
                 0  1
                /    \
               D      E
```

You can think of each symbol's code as it's "address" in this tree. When decoding a string, we start at the topmost node. Reading the binary sequence bit by bit, we move down the tree, taking a left edge if we see a 0 and a right edge if we see a 1. Once we reach a letter, we return to the top node and repeat the process.

Definition 23:
We say a coding scheme is *prefix-free* if no whole code word is a prefix of another code word.

Problem 24:
Convince yourself that trees like the one above always produce a prefix-free code.

Problem 25:
Decode [110111001001110110] using the tree above.

Solution

This is [110·111·00·10·01·110·110], which is DEACBDD

Problem 26:
Encode ABDECBE using this tree.
How many bits do we save over a naïve scheme?

Solution

This is [00 01 110 111 10 01 111], and saves four bits.

Problem 27:
In Problem 25, we needed 18 bits to encode `DEACBDD`.
Note that we'd need $3 \times 7 = 21$ bits to encode this string naïvely.

Draw a tree that encodes this string more efficiently.

Solution

Two possible solutions are below.

- The left tree encodes `DEACBDD` as `[00·111·110·10·01·00·00]`, using 16 bits.
- The right tree encodes `DEACBDD` as `[0·111·101·110·100·0·0]`, using 15 bits.

```
        / \
       0   1
      /     \
     /\     /\
    0  1   0  1
   /    \ /    \
  D     B C    /\
              0  1
             /    \
            A      E
```

```
     / \
    0   1
   /     \
  D      /\
        0  1
       /    \
      /\    /\
     1  0  0  1
    /    \/    \
   A     B C    E
```

Problem 28:
Now, do the opposite: draw a tree that encodes `DEACBDD` *less* efficiently than before.

Solution

Bury `D` as deep as possible in the tree, so that we need four bits to encode it.

Remark 29:
As we just saw, constructing a prefix-free code is fairly easy.
Constucting the *most efficient* prefix-free code for a given message is a bit more difficult.

Remark 30:
Let's restate our problem.
Given an alphabet A and a frequency function f, we want to construct a binary tree T that minimizes

$$\mathcal{B}_f(T) = \sum_{a \in A} f(a) \times d_T(a)$$

Where...

- a is a symbol in A
- $d_T(a)$ is the "depth" of a in our tree.
  In other words, $d_T(a)$ is the number of bits we need to encode a
- $f(a)$ is a frequency function that maps each symbol in A to a value in $[0, 1]$.
  You can think of this as the distribution of symbols in messages we expect to encode.
  For example, consider the alphabet $\{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$:
  - In `AAA`, $f(\mathtt{A}) = 1$ and $f(\mathtt{B}) = f(\mathtt{C}) = 0$.
  - In `ABC`, $f(\mathtt{A}) = f(\mathtt{B}) = f(\mathtt{C}) = 1/3$.

  Note that $f(a) \geq 0$ and $\sum f(a) = 1$.

Also notice that $\mathcal{B}_f(T)$ is the "average bits per symbol" metric we saw in previous problems.

Problem 31:
Let f be fixed frequency function over an alphabet A.
Let T be an arbitrary tree for A, and let a, b be two symbols in A.
Construct T' by swapping a and b in T. Show that

$$\mathcal{B}_f(T) - \mathcal{B}_f(T') = \Big(f(b) - f(a)\Big) \times \Big(d_T(a) - d_T(b)\Big)$$

Solution

$\mathcal{B}_f(T)$ and $\mathcal{B}_f(T')$ are nearly identical, and differ only at $d_T(a)$ and $d_T(b)$. So, we get...

$$\begin{aligned}
\mathcal{B}_f(T) - \mathcal{B}_f(T') &= f(a)d_T(a) + f(b)d_T(b) - f(a)d_T(b) - f(b)d_T(a) \\
&= f(a)\big(d_T(a) - d_T(b)\big) + f(b)\big(d_T(b) - d_T(a)\big) \\
&= \Big(f(b) - f(a)\Big) \times \Big(d_T(a) - d_T(b)\Big)
\end{aligned}$$

Problem 32:
Show that there is an optimal tree in which the two symbols with the lowest frequencies have the same parent. *Hint:* You may assume that an optimal tree exists. There are a few cases.

Solution

Let T be an optimal tree, and let a, b be the two symbols with the lowest frequency.
If there is a tie among three or more symbols, pick a, b to be those with the greatest depth.
Label a and b so that that $d_T(a) \geq d_T(a)$.

If a and b share a parent, we're done. If a and b do not share a parent, we have three cases:

- There is a node x with $d_T(x) > d_T(a)$.
  Create T' by swapping a and x. By definition, $f(a) < f(x)$, and thus by Problem 31 $\mathcal{B}_f(T) > \mathcal{B}_f(T')$. This is a contradiction, since we chose T as an optimal tree—so this case is impossible.
- a is an only child. Create T' by removing a's parent and replacing it with a.
  Then $\mathcal{B}_f(T) > \mathcal{B}_f(T')$, same contradiction as above.
  If we assume T is a full binary tree, this case doesn't exist.
- a has a sibling x, and x isn't b.
  Let T' be the tree created by swapping x and b (thus making a and b siblings).
  By Problem 31, $\mathcal{B}_f(T) \geq \mathcal{B}_f(T')$. T is optimal, so there cannot be a tree with a better average length—thus $\mathcal{B}_f(T) = \mathcal{B}_f(T')$ and T' is also optimal.

Problem 33:
Devise an algorithm that builds an optimal tree given an alphabet A and a frequency function f. Then, use the previous two problems to show that your algorithm indeed produces an ideal tree.
Hint: First, make an algorithm that makes sense intuitively.
Once you have something that looks good, start your proof.
Hint: Build from the bottom.

Solution

The Algorithm:
Given an alphabet A and a frequency function f...

- If $|A| = 1$, return a single node.
- Let a, b be two symbols with the smallest frequency.
- Let $A' = A - \{a, b\} + \{x\}$ (Where x is a new "placeholder" symbol)
- Let $f'(x) = f(a) + f(b)$, and $f'(s) = f(s)$ for all other symbols s.
- Compute T' by repeating this algorithm on A' and f'
- Create T from T' by adding a and b as children of x.

In plain english: pick the two nodes with the smallest frequency, combine them, and replace them with a "compound symbol". Repeat until you're done.

The Proof:
We'll proceed by induction on $|A|$.
Let f be an arbitrary frequency function.

Base case: $|A| = 1$. We only have one vertex, and we thus only have one tree.
The algorithm above produces this tree. Done.

Induction: Assume that for all A with $|A| = n - 1$, the algorithm above produces an ideal tree. First, we'll show that $\mathcal{B}_f(T) = \mathcal{B}_{f'}(T') + f(a) + f(b)$:

$$\begin{aligned}
\mathcal{B}_f(T) &= \sum_{x \in A - \{a,b\}} \Big(f(x) d_T(x) \Big) + f(a) d_T(a) + f(b) d_T(b) \\
&= \sum_{x \in A - \{a,b\}} \Big(f(x) d_T(x) \Big) + \Big(f(a) + f(b) \Big) \Big(d_{T'}(x) + 1 \Big) \\
&= \sum_{x \in A - \{a,b\}} \Big(f(x) d_T(x) \Big) + f'(z) d_{T'}(z) + f(a) + f(b) \\
&= \sum_{x \in A'} \Big(f'(x) d_{T'}(x) \Big) + f(a) + f(b) \\
&= \mathcal{B}_{f'}(T') + f(a) + f(b)
\end{aligned}$$

Now, assume that T is not optimal. There then exists an optimal tree U with a and b as siblings (by Problem 32). Let U' be the tree created by removing a, b from U. U' is a tree for A' and f', so we can repeat the calculation above to find that $\mathcal{B}_f(U) = \mathcal{B}_{f'}(U') + f(a) + f(b)$.

So, $\mathcal{B}_{f'}(T') \;=\; \mathcal{B}_f(T) - f(a) - f(b) \;>\; \mathcal{B}_f(U) - f(a) - f(b) \;=\; \mathcal{B}_{f'}(U')$.
Since T' is optimal for A' and f', this is a contradition. T must therefore be optimal.

Part 5: Bonus problems

Problem 34:
Make sense of the document on the next page.
What does it describe, and how does it work?

Problem 35:
Given a table with a marked point, O, and with 2013 properly working watches put down on the table, prove that there exists a moment in time when the sum of the distances from O to the watches' centers is less than the sum of the distances from O to the tips of the watches' minute hands.

Problem 36: A Minor Inconvenience
A group of eight friends goes out to dinner. Each drives his own car, checking it in with valet upon arrival. Unfortunately, the valet attendant forgot to tag the friends' keys. Thus, when the group leaves the restaurant, each friend is handed a random key.

- What is the probability that everyone gets the correct set of keys?
- What is the probability that each friend gets the wrong set?

Problem 37: Bimmer Parking
A parking lot has a row of 16 spaces, of which a random 12 are taken.
Ivan drives a BMW, and thus needs two adjacent spaces to park.
What is the probability he'll find a spot?

THE QUITE OK IMAGE FORMAT

Specification **Version 1.0**, 2022.01.05 – qoiformat.org – Dominic Szablewski

A QOI file consists of a 14-byte header, followed by any number of data “chunks” and an 8-byte end marker.

```
qoi_header {
    char     magic[4];   // magic bytes "qoif"
    uint32_t width;      // image width in pixels (BE)
    uint32_t height;     // image height in pixels (BE)
    uint8_t  channels;   // 3 = RGB, 4 = RGBA
    uint8_t  colorspace; // 0 = sRGB with linear alpha
                         // 1 = all channels linear
};
```

The colorspace and channel fields are purely informative. They do not change the way data chunks are encoded.

Images are encoded row by row, left to right, top to bottom. The decoder and encoder start with **{r: 0, g: 0, b: 0, a: 255}** as the previous pixel value. An image is complete when all pixels specified by **width * height** have been covered. Pixels are encoded as:

- a run of the previous pixel
- an index into an array of previously seen pixels
- a difference to the previous pixel value in r,g,b
- full r,g,b or r,g,b,a values

The color channels are assumed to not be premultiplied with the alpha channel (“un-premultiplied alpha”).

A running **array[64]** (zero-initialized) of previously seen pixel values is maintained by the encoder and decoder. Each pixel that is seen by the encoder and decoder is put into this array at the position formed by a hash function of the color value. In the encoder, if the pixel value at the index matches the current pixel, this index position is written to the stream as **QOI_OP_INDEX**. The hash function for the index is:

```
index_position = (r * 3 + g * 5 + b * 7 + a * 11) % 64
```

Each chunk starts with a 2- or 8-bit tag, followed by a number of data bits. The bit length of chunks is divisible by 8 - i.e. all chunks are byte aligned. All values encoded in these data bits have the most significant bit on the left. The 8-bit tags have precedence over the 2-bit tags. A decoder must check for the presence of an 8-bit tag first.

The byte stream's end is marked with 7 **0x00** bytes followed by a single **0x01** byte.

The possible chunks are:

QOI_OP_RGB

Byte[0]	Byte[1]	Byte[2]	Byte[3]
7 6 5 4 3 2 1 0	7 .. 0	7 .. 0	7 .. 0
1 1 1 1 1 1 1 0	red	green	blue

8-bit tag b11111110
8-bit   red channel value
8-bit green channel value
8-bit blue channel value

The alpha value remains unchanged from the previous pixel.

QOI_OP_RGBA

Byte[0]	Byte[1]	Byte[2]	Byte[3]	Byte[4]
7 6 5 4 3 2 1 0	7 .. 0	7 .. 0	7 .. 0	7 .. 0
1 1 1 1 1 1 1 1	red	green	blue	alpha

8-bit tag b11111111
8-bit   red channel value
8-bit green channel value
8-bit blue channel value
8-bit alpha channel value

QOI_OP_INDEX

Byte[0]	
7 6	5 4 3 2 1 0
0 0	index

2-bit tag b00
6-bit index into the color index array: 0..63

A valid encoder must not issue 2 or more consecutive QOI_OP_INDEX chunks to the same index. QOI_OP_RUN should be used instead.

QOI_OP_DIFF

Byte[0]			
7 6	5 4	3 2	1 0
0 1	dr	dg	db

2-bit tag b01
2-bit   red channel difference from the previous pixel -2..1
2-bit green channel difference from the previous pixel -2..1
2-bit blue channel difference from the previous pixel -2..1

The difference to the current channel values are using a wraparound operation, so **1 - 2** will result in **255**, while **255 + 1** will result in **0**.

Values are stored as unsigned integers with a bias of **2**. E.g. **-2** is stored as **0** (**b00**). **1** is stored as **3** (**b11**).

The alpha value remains unchanged from the previous pixel.

QOI_OP_LUMA

Byte[0]		Byte[1]	
7 6	5 4 3 2 1 0	7 6 5 4	3 2 1 0
1 0	diff green	dr - dg	db - dg

2-bit tag b10
6-bit green channel difference from the previous pixel -32..31
4-bit   red channel difference minus green channel difference -8..7
4-bit blue channel difference minus green channel difference -8..7

The green channel is used to indicate the general direction of change and is encoded in 6 bits. The red and blue channels (dr and db) base their diffs off of the green channel difference. I.e.:

```
dr_dg = (cur_px.r - prev_px.r) - (cur_px.g - prev_px.g)
db_dg = (cur_px.b - prev_px.b) - (cur_px.g - prev_px.g)
```

The difference to the current channel values are using a wraparound operation, so **10 - 13** will result in **253**, while **250 + 7** will result in **1**.

Values are stored as unsigned integers with a bias of **32** for the green channel and a bias of **8** for the red and blue channel.

The alpha value remains unchanged from the previous pixel.

QOI_OP_RUN

Byte[0]	
7 6	5 4 3 2 1 0
1 1	run

2-bit tag b11
6-bit run-length repeating the previous pixel: 1..62

The run-length is stored with a bias of **-1**. Note that the run-lengths **63** and **64** (**b111110** and **b111111**) are illegal as they are occupied by the **QOI_OP_RGB** and **QOI_OP_RGBA** tags.