Chapter 6

Compressing Data

Often we need to attempt to reduce the amount of space taken by some data, for storage or transmission. If data were arbitrary, it could not be compressed – to compress data we need to recognise and exploit patterns. However, most data is not arbitrary at all, but structured and containing repeated patterns. No compression scheme can guarantee to reduce the size of all inputs, but we would like to achieve good compression on typical data.

We will look at a simple byte-by-byte compression scheme, implementing it using our `input` and `output` data types so it can be applied to different kinds of inputs and outputs. Then, we will consider a more complex but efficient scheme for compressing data on a bit-by-bit basis.

A byte-by-byte compression scheme

The following extract from ISO-32000 (the PDF standard) defines a byte-by-byte compression scheme:

> The encoded data shall be a sequence of *runs*, where each run shall consist of a *length* byte followed by 1 to 128 bytes of data. If the *length* byte is in the range 0 to 127, the following *length* + 1 (1 to 128) bytes shall be copied literally during decompression. If the *length* is in the range 129 to 255, the following single byte shall be copied 257 - *length* (2 to 128) times during decompression. A *length* value of 128 shall denote EOD [end of data].

For example, consider the text "`((5.000000, 4.583333), (4.500000,5.000000))`", which can be represented in ASCII as the list of integers `[40; 40; 53; 46; 48; 48; 48; 48; 48; 48; 44; 32; 52; 46; 53; 56; 51; 51; 51; 51; 41; 44; 32; 40; 52; 46; 53; 48; 48; 48; 48; 48; 44; 53; 46; 48; 48; 48; 48; 48; 48; 41; 41]` of length 43. This will be compressed as shown in Figure 6.1. That is to say, as `[255; 40; 1; 53; 46; 251; 48; 5; 44; 32; 52; 46; 53; 56; 253; 51; 6; 41; 44; 32; 40; 52; 46; 53; 252; 48; 2; 44; 53; 46; 251; 48; 255; 41; 128]`, of length 35. The maximum compression ratio achieved with this method (for long constant data) is 64:1. The worst case (alternating bytes) is 127:128 – a slight expansion.

Let us begin with two utility functions to allow us to convert between strings and lists of integers – this will make it easier to understand our examples when we evaluate them in OCaml's top level:

Input data	Type of run	Length	Output header	Output data
40 40	same	2	257 - 2 = 255	40
53 46	different	2	2 - 1 = 1	53 46
48 48 48 48 48 48	same	6	257 - 6 = 251	48
44 32 52 46 53 56	different	6	6 - 1 = 5	44 32 52 46 53 56
51 51 51 51	same	4	257 - 4 = 253	51
41 44 32 40 52 46 53	different	7	7 - 1 = 6	41 44 32 40 52 46 53
48 48 48 48 48	same	5	257 - 5 = 252	48
44 53 46	different	3	3 - 1 = 2	44 53 46
48 48 48 48 48 48	same	6	257 - 6 = 251	48
41 41	same	2	257 - 2 = 255	41
-	EOD	-	-	128

Figure 6.1

```
string_of_int_list : int list → string
int_list_of_string : string → int list

let string_of_int_list l =
  let b = Bytes.create (List.length l) in
    List.iteri (fun n x -> Bytes.set b n (char_of_int x)) l;
    Bytes.to_string b

let int_list_of_string s =
  let l = ref [] in
    for x = String.length s - 1 downto 0 do
      l := int_of_char s.[x] :: !l
    done;
    !l
```

Note the use of the Standard Library function `List.iteri` which is like `List.iter`, but it passes an additional integer to the function each time, representing the position in the list, starting at 0. Note also that we use **`downto`** in `int_list_of_string` to avoid a list reversal.

Now, we shall build a little abstraction to allow our compression and decompression functions to share a common basis. Our `compress` and `decompress` function will need to read from any `input`, and write to any `output`. However, for our little tests, we shall be reading from and writing to strings. Since we do not know the size of the compressed or decompressed `output` in advance, we cannot allocate an appropriately sized string. So, let us build an `output` from a **`Buffer.t`**. This way, we can write a function `process` which, given a compression or decompression function, creates a buffer, builds an `output` from it, calls the function to process the data, and then extracts the final string from the buffer.

```
output_of_buffer : Buffer.t → output
process : (input → output → unit) → string → string

let output_of_buffer b =
  {output_char = Buffer.add_char b;
   out_channel_length = fun () -> Buffer.length b}

let process f s =
  let b = Buffer.create (String.length s) in
    f (input_of_string s) (output_of_buffer b);
    Buffer.contents b
```

Now, given some suitable function `decompress`, say, we can call `process` giving the function and a string, and get a new string back.

Decompression in this scheme, as is often the case, is simpler than compression, so we shall address it first. The whole thing is wrapped in a loop which terminates only upon an exception. If the input data is well-formed, that exception will be `EOD`, which we have defined for this purpose. Inside the loop, we read a byte from the input. It is either in the range $0...127$, in which case it is a "different" run, and we copy some bytes from input to output. Or, it is in the range $129...255$, in which case it is a "same" run, and we output a number of copies of the next byte. Otherwise, the byte must have value 128, and we raise the `EOD` exception. The `decompress_string` function is as simple as we claimed it would be, because we wrote `process`.

```
decompress : input → output → unit
decompress_string : string → string

exception EOD

let decompress i o =
  try
    while true do
      match int_of_char (i.input_char ()) with
        x when x >= 0 && x <= 127 ->
          for p = 1 to x + 1 do o.output_char (i.input_char ()) done
      | x when x > 128 && x <= 255 ->
          let c = i.input_char () in
            for p = 1 to 257 - x do o.output_char c done
      | _ -> raise EOD
    done
  with
    EOD -> ()

let decompress_string = process decompress
```

For compression, we will first write functions to recognise a "same" run and a "different" run in the input, and then a main function which uses them as appropriate. The function `get_same` returns the first character, and an integer representing the number (one or more) of "same" characters starting with the first one. It leaves the input pointing at the last "same" character. If 128 same characters have been found,

we must stop early. If we are at the end of input when the function is called, `End_of_file` is raised as usual.

```
get_same : input → char × int

let get_same i =
  let rec getcount ch c =
    if c = 128 then 128 else
      try
        if i.input_char () = ch
          then getcount ch (c + 1)
          else (rewind i; c)
      with
        End_of_file -> c
  in
    let ch = i.input_char () in (ch, getcount ch 1)
```

The `get_different` function is rather more awkward. It will return the non-empty list of "different" characters starting at the current position. We must stop as soon as we notice two like characters, rewinding twice and removing a character from our accumulator. So, for example, if we are reading "`An Accumulation`" we want to return `['A'; 'n'; ' '; 'A']` but we do not know this until we read the second `'c'`. Again, we must stop after 128 differing characters. As before, `End_of_file` is raised if we are at the end of the input on the initial call.

```
get_different : input → char list

let get_different i =
  let rec getdiffinner a c =
    if c = 128 then List.rev a else
      try
        let ch' = i.input_char () in
          if ch' <> List.hd a
            then getdiffinner (ch' :: a) (c + 1)
            else (rewind i; rewind i; List.rev (List.tl a))
      with
        End_of_file -> List.rev a
  in
    getdiffinner [i.input_char ()] 1
```

The compression function is now relatively simple. We repeatedly call `get_same`. If it indicates a run of length one, we call `get_different` instead. In each case we write appropriate data. Then, on `End_of_file`, we write the EOD marker. The `compress_string` function is built just like `decompress_string`.

```
compress : input → output → unit

let compress i o =
  try
    while true do
      match get_same i with
        (_, 1) ->
          rewind i;
          let cs = get_different i in
            o.output_char (char_of_int (List.length cs - 1));
            List.iter o.output_char cs
      | (b, c) ->
          o.output_char (char_of_int (257 - c));
          o.output_char b
    done
  with
    End_of_file -> o.output_char (char_of_int 128)

let compress_string = process compress
```

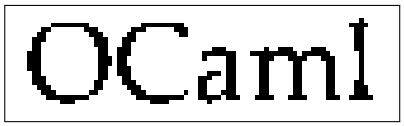
Let us try with our sample data:

```
       OCaml

# open Examples;;
# example;;
- : string = "((5.000000, 4.583333), (4.500000,5.000000))"

# int_list_of_string example;;
- : int list =
[40; 40; 53; 46; 48; 48; 48; 48; 48; 48; 44; 32; 52; 46; 53; 56; 51; 51; 51;
 51; 41; 44; 32; 40; 52; 46; 53; 48; 48; 48; 48; 48; 44; 53; 46; 48; 48; 48;
 48; 48; 48; 41; 41]

# let smaller = compress_string example;;
val smaller : string = "?(\0015.?0\005, 4.58?3\006), (4.5?0\002,5.?0?)\128"

# int_list_of_string smaller;;
- : int list =
[255; 40; 1; 53; 46; 251; 48; 5; 44; 32; 52; 46; 53; 56; 253; 51; 6; 41; 44;
 32; 40; 52; 46; 53; 252; 48; 2; 44; 53; 46; 251; 48; 255; 41; 128]

# decompress_string (compress_string example) = example;;
- : bool = true
```

A bit-by-bit compression scheme

Our previous method required whole bytes to be equal to one another to compress well. Now we consider compression bit-by-bit, but on the same principle – encoding the lengths of runs of data. We will build a

compressor and decompressor for the CCITT Group 3 compression scheme. This is the basic scheme used by the fax machines since the 1970s, and for 1 bit-per-pixel TIFF files. It is also used in the PDF format, as we shall see later in the book. Here is our example data, an 80x21, one-bit-per-pixel image of a scanned word from printed paper:



We can put it into our source file using the characters `0` and `1` to ease readability. A string can run over several lines if a backslash is added at the end of each line. Spaces at the beginning of the next line are also skipped, so we can line it up nicely:

```
let input_data =
"00000000000000000000000000000000000000000000000000000000000000000000000000000000\
 00000000000000000000000000000000000000000000000000000000000000000000000000000001000000\
 00000000111111110000000000011111111100000000000000000000000000000000000000111100000\
 00000011000000011100000001110000001110000000000000000000000000000000000000011000000\
 00000110000000001110000011000000000110000000000000000000000000000000000000011000000\
 00001110000000000111000111000000000000000000000000000000000000000000000000011000000\
 00001100000000000111000110000000000000000000000000000000000000000000000000011000000\
 00001100000000000011001110000000000000000011100000000100111000011100000011000000\
 00011100000000000011001110000000000000000111111100011111111110111111000000001000000\
 00011100000000000011101100000000000000000100001100000111000111100011100000001000000\
 00011100000000000011101100000000000000000011000001100000110000011000001000000\
 00011100000000000011001110000000000000000011000001100000110000011000001000000\
 00001100000000000011001110000000000000000111011000001100000110000011000001000000\
 00001110000000000011000110000000000000000111000110000011000001100000110000001000000\
 00001110000000000011000111000000000000000110000110000011000001100000110000011000000\
 00000111000000000110000111000000000000001100001100000110000011000001100000111000000\
 00000011100000001100000011100000000010010001110000011000001100000110000011100000000\
 00000011111111110000000001111111111000111110111000111100011110000111001111100000\
 00000000011100000000000000011110000000010000000000000000000000000000000000000000\
 00000000000000000000000000000000000000000000000000000000000000000000000000000000\
 00000000000000000000000000000000000000000000000000000000000000000000000000000000"
```

You can see that it consists of runs of 0 (white) and 1 (black) pixels, 80 to each line, with 21 lines.

Preliminaries

First, we had better define a function to convert the string of zeros and ones to a string containing the actual binary data. We can do this by building an `output_bits` with an `output` built from a buffer. Then, after flushing, we can use `Buffer.contents` to extract the data we have written.

```
packedstring_of_string : string → string

let packedstring_of_string s =
  let b = Buffer.create (String.length s / 8 + 1) in
  let o = output_bits_of_output (output_of_buffer b) in
    for x = 0 to String.length s - 1 do putbit o (s.[x] = '1') done;
    flush o;
    Buffer.contents b
```

Note that, since the string will always be a whole number of bytes (possibly padded with zeros by `flush`) we must remember the width and height of our image (80 and 21 here) and pass one or both of them to some of our other functions. For example, we can write a function to print one of these packed binary strings. This needs the width of the image to know when to print newlines:

```
print_packedstring : int → string → unit

let print_packedstring w s =
  let ibits = input_bits_of_input (input_of_string s) in
    try
      while true do
        for column = 1 to w do print_int (getbitint ibits) done;
        print_newline ()
      done
    with
      End_of_file -> ()
```

For simplicity here, we do not worry about padding at the end (in our example, since the width is 80, it will consist of whole bytes anyway).

Fax compression explained

We have said that the compression will proceed by encoding the lengths of the runs of zeros and ones in our data. How do we encode them? We need a set of binary sequences which can be distinguished from one another (i.e. no sequence can be a prefix of another sequence). In the case of fax compression, there are codes for run lengths 0 to 63, and different codes for white and black runs, as shown in Figure 6.2. Shorter codes are chosen for the more common cases, based on a statistical analysis of real documents.
Notice that no white symbol is a prefix of another white symbol, and no black symbol is a prefix of another black symbol. These codes for runs from 0 to 63 are called terminating codes. When a run is longer, we use one "make-up code", shown in Figure 6.3, followed by a terminating code. Thus, the longest run is of length 1728 + 63 = 1791.

The reason for distinguishing between white and black codes is to do with error correction in unreliable transport mechanisms (such as phone lines that fax machines operate over) – otherwise we could just have one set of codes and assume runs alternate. The reason for the existence of a zero-length run is that each line is defined to begin with a white run. If it is really black, a zero-length white run is output first. No run is longer than a line. As an example, let us compress the first three lines of our example data:

Run	White	Black	Run	White	Black	Run	White	Black
0	00110101	0000110111	22	0000011	00000110111	44	00101101	000001010100
1	000111	010	23	0000100	00000101000	45	00000100	000001010101
2	0111	11	24	0101000	00000010111	46	00000101	000001010110
3	1000	10	25	0101011	00000011000	47	00001010	000001010111
4	1011	011	26	0010011	000011001010	48	00001011	000001100100
5	1100	0011	27	0100100	000011001011	49	01010010	000001100101
6	1110	0010	28	0011000	000011001100	50	01010011	000001010010
7	1111	00011	29	00000010	000011001101	51	01010100	000001010011
8	10011	000101	30	00000011	000001101000	52	01010101	000000100100
9	10100	000100	31	00011010	000001101001	53	00100100	000000110111
10	00111	0000100	32	00011011	000001101010	54	00100101	000000111000
11	01000	0000101	33	00010010	000001101011	55	01011000	000000100111
12	001000	0000111	34	00010011	000011010010	56	01011001	000000101000
13	000011	00000100	35	00010100	000011010011	57	01011010	000001011000
14	110100	00000111	36	00010101	000011010100	58	01011011	000001011001
15	110101	000011000	37	00010110	000011010101	59	01001010	000000101011
16	101010	0000010111	38	00010111	000011010110	60	01001011	000000101100
17	101011	0000011000	39	00101000	000011010111	61	00110010	000001011010
18	0100111	0000001000	40	00101001	000001101100	62	00110011	000001100110
19	0001100	00001100111	41	00101010	000001101101	63	00110100	000001100111
20	0001000	00001101000	42	00101011	000011011010	(make up codes required		
21	0010111	00001101100	43	00101100	000011011011	for longer runs)		

Figure 6.2

Run	White	Black	Run	White	Black
64	11011	0000001111	960	011010100	0000001110011
128	10010	000011001000	1024	011010101	0000001110100
192	010111	000011001001	1088	011010110	0000001110101
256	0110111	000001011011	1152	011010111	0000001110110
320	00110110	000000110011	1216	011011000	0000001110111
384	00110111	000000110100	1280	011011001	0000001010010
448	01100100	000000110101	1344	011011010	0000001010011
512	01100101	0000001101100	1408	011011011	0000001010100
576	01101000	0000001101101	1472	010011000	0000001010101
640	01100111	0000001001010	1536	010011001	0000001011010
704	011001100	0000001001011	1600	010011010	0000001011011
768	0 11001101	0000001001100	1664	011000	0000001100100
832	011010010	0000001001101	1728	010011011	0000001100101
896	011010011	0000001110010			

Figure 6.3

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000
000000001111111100000000000011111111110000000000000000000000000000000000000111100000
```

We have the following:

Length	Colour	Code
80	White	11011 + 101010
73	White	11011 + 10100
1	Black	010
6	White	1110
8	White	1011
8	Black	000101
11	White	01000
9	Black	000100
35	White	00010100
4	Black	011
5	White	1100

We have reduced the data from 240 bits to 64 bits, a reduction of about three quarters.

Compressing fax data

First, we need to encode the terminating and make-up codes. We will use arrays for direct indexing, but lists for each element, since we do not need random access to each bit. This is shown in Figure 6.4. Now, we can write a function which, given a length of run and a colour, gives the appropriate code as a list of bits:

```
code : bool → int → int list

let rec code isblack length =
  if length > 1791 || length < 0 then
    raise (Invalid_argument "code: bad length")
  else
    if length > 64 then
      let m =
        if isblack
          then black_make_up_codes.(length / 64 - 1)
          else white_make_up_codes.(length / 64 - 1)
      in
        m @ code isblack (length mod 64)
    else
      if isblack
        then black_terminating_codes.(length)
        else white_terminating_codes.(length)
```

Now, a function which, given the current colour, an input_bits, the current number of like bits read, and the width of the image, returns a pair of the number of like bits, and the colour (`peekbit` is a generally useful function – it returns one bit without advancing):

```
peekbit : input_bits → int
read_up_to : int → input_bits → int → int → int × int

let peekbit b =
  if b.bit = 0 then
    begin
      let byte = int_of_char (b.input.input_char ()) in
        rewind b.input;
        byte land 128 > 0
    end
  else
    b.byte land b.bit > 0

let rec read_up_to v i n w =
  if n >= w then (n, v) else
    if peekbit i = v
      then (ignore (getbit i); read_up_to v i (n + 1) w)
      else (n, v)
```

```
white_terminating_codes : int list array
black_terminating_codes : int list array
white_make_up_codes : int list array
black_make_up_codes : int list array

let white_terminating_codes =
  [|[0; 0; 1; 1; 0; 1; 0; 1];
    [0; 0; 0; 1; 1; 1];
    [0; 1; 1; 1];
    [1; 0; 0; 0];
    [1; 0; 1; 1];
    [1; 1; 0; 0];                                          and so on...

let black_terminating_codes =
  [|[0; 0; 0; 1; 1; 0; 1; 1; 1];
    [0; 1; 0];
    [1; 1];
    [1; 0];
    [0; 1; 1];
    [0; 0; 1; 1];                                          and so on...

let white_make_up_codes =
  [|[1; 1; 0; 1; 1];
    [1; 0; 0; 1; 0];
    [0; 1; 0; 1; 1; 1];
    [0; 1; 1; 0; 1; 1; 1];
    [0; 0; 1; 1; 0; 1; 1; 0];
    [0; 0; 1; 1; 0; 1; 1; 1];                              and so on...

let black_make_up_codes =
  [|[0; 0; 0; 0; 0; 0; 1; 1; 1; 1];
    [0; 0; 0; 0; 1; 1; 0; 0; 1; 0; 0; 0];
    [0; 0; 0; 0; 1; 1; 0; 0; 1; 0; 0; 1];
    [0; 0; 0; 0; 0; 1; 0; 1; 1; 0; 1; 1];
    [0; 0; 0; 0; 0; 0; 1; 1; 0; 0; 1; 1];
    [0; 0; 0; 0; 0; 0; 1; 1; 0; 1; 0; 0];                  and so on...
```

Figure 6.4

For example, the first call to `read_up_to` will calculate (80, 0). Now we can write the main function. Given an `input_bits` and `output_bits`, and the width and height of the image, for each line we check if a zero-width white run needs to be added, and then call `encode_fax_line`.

```
encode_fax : input_bits → output_bits → int → int → unit

let encode_fax i o w h =
  let rec encode_fax_line i o w =
    if w > 0 then
      let n, isblack = read_up_to (peekbit i) i 0 w in
        List.iter (putbitint o) (code isblack n);
        encode_fax_line i o (w - n)
  in
    for x = 1 to h do
      if peekbit i then List.iter (putbitint o) (code false 0);
      encode_fax_line i o w
    done
```

Now we can write a function `process`, just like we did in the byte-by-byte example, but for `input_bits` and `output_bits`. We must be sure to flush the output. The main `compress_string_ccitt` function is then simple:

```
process : (input_bits → output_bits → int → int → unit) →
          string → int → int → string
compress_string_ccitt : (int → int → unit) → string → int → int → string

let process f s w h =
  let b = Buffer.create (String.length s) in
  let ibits = input_bits_of_input (input_of_string s) in
  let obits = output_bits_of_output (output_of_buffer b) in
    f ibits obits w h;
    flush obits;
    Buffer.contents b

let compress_string_ccitt = process encode_fax
```

For our full input data, the input string is $80 \times 21 = 1680$ bits long, and the compressed string is 960 bits long, a compression ratio of 7:4.

Decompressing fax data

For decompression, we will write two functions for reading white and black codes. It is easiest to directly encode the decision tree:

```
read_white_code : input_bits → int
read_black_code : input_bits → int

let rec read_white_code i =
  let a = getbitint i in
  let b = getbitint i in
  let c = getbitint i in
  let d = getbitint i in
    match a, b, c, d with
      0, 1, 1, 1 -> 2
    | 1, 0, 0, 0 -> 3
    | 1, 0, 1, 1 -> 4
    | 1, 1, 0, 0 -> 5
    | 1, 1, 1, 0 -> 6
    | 1, 1, 1, 1 -> 7
    | _ ->
  let e = getbitint i in
    match a, b, c, d, e with
      1, 0, 0, 1, 1 -> 8
    | 1, 0, 1, 0, 0 -> 9
    | 0, 0, 1, 1, 1 -> 10
    | 0, 1, 0, 0, 0 -> 11
    | 1, 1, 0, 1, 1 -> 64 + read_white_code i
    | 1, 0, 0, 1, 0 -> 128 + read_white_code i
    | _ ->
  let f = getbitint i in
    match a, b, c, d, e, f with
      0, 0, 0, 1, 1, 1 -> 1                                    and so on...

let rec read_black_code i =
  let a = getbitint i in
  let b = getbitint i in
    match a, b with
      1, 1 -> 2
    | 1, 0 -> 3
    | _ ->
  let c = getbitint i in
    match a, b, c with
      0, 1, 0 -> 1
    | 0, 1, 1 -> 4
    | _ ->
  let d = getbitint i in
  match a, b, c, d with
    0, 0, 1, 1 -> 5                                            and so on...
```

Now, decoding is relatively simple. We decode runs for each line, until the width is all used up, and do this for each line. We read white and black codes alternately – each line begins on white. We can use `process` again:

```
decode_fax : input_bits → output_bits → int → int → unit
decompress_string_ccitt : (int → int → unit) → string → int → int → string

let decode_fax i o w h =
  let lines = ref h in
  let pixels = ref w in
  let iswhite = ref true in
    while !lines > 0 do
      while !pixels > 0 do
        let n =
          (if !iswhite then read_white_code else read_black_code) i
        in
          for x = 1 to n do
            putbitint o (if !iswhite then 0 else 1)
          done;
          pixels := !pixels - n;
          iswhite := not !iswhite
        done;
        iswhite := true;
        pixels := w;
        lines := !lines - 1
      done

let decompress_string_ccitt = process decode_fax
```

We can verify our code by evaluating `s = decompress_string_ccitt (compress_string_ccitt s)` for our example data.

Questions

1. How much complexity did using the `input` and `output` types add to `compress` and `decompress` in our byte-by-byte example? Rewrite the functions so they just operate over lists of integers, in functional style, and compare the two.
2. Replace our manual tree of codes with a tree automatically generated from the lists of codes used for compression. The tree will have no data at its branches (since no code is a prefix of another), and will have data at only some of its leaves. Define a suitable data type first.
3. What happens if we compress our data as a single line of 1680 bits instead of 21 lines of 80? What happens if we try to compress already-compressed data?
4. Write a function which, given input data, will calculate a histogram of the frequencies of different runs of white and black. This could be used to build custom codes for each image, improving compression.