Lempel-Ziv-Stac

Submit Solution

Lempel–Ziv–Stac (LZS, or Stac compression) is a lossless data compression algorithm that uses a combination of the LZ77 sliding-window compression algorithm and fixed Huffman coding. It was originally developed by Stac Electronics for tape compression, and subsequently adapted for hard disk compression and sold as the Stacker disk compression software. It was later specified as a compression algorithm for various network protocols.

Assignment

Create files `lzs.h` and `lzs.cpp` that allow to use the following function declaration:

```
void lzs_decompress(std::istream& is, std::ostream& os);
```

The `.h` file should contain only the non defining declaration of the function, while the definition should be in the `.cpp` file. Don`t provide a `main()` function, because one is already defined in the system and will be linked to your code.

The function should read from the input stream `is` a sequence compressed with the LZS algorithm and write the decoded sequence to the output stream `os`.

The input sequence will always be valid, so no error checking is required.

Algorithm

LZS compression and decompression uses an LZ77 type algorithm. It uses the last 2 KB of uncompressed data as a sliding-window dictionary.

An LZS compressor looks for matches between the data to be compressed and the last 2 KB of data. If it finds a match, it encodes an offset/length reference to the dictionary. If no match is found, the next data byte is encoded as a `literal` byte. The compressed data stream ends with an end-marker.

Compressed data format

Data is encoded into a stream of variable-bit-width tokens.

Literal byte

A literal byte is encoded as a `0` bit followed by the 8 bits of the byte.

Offset/length reference

An offset/length reference is encoded as a `1` bit followed by the encoded offset, followed by the encoded length. One exceptional encoding is an end marker, described below.

An **offset** can have a minimum value of 1 and a maximum value of 2047. A value of 1 refers to the most recent byte in the history buffer, immediately preceding the next data byte to be processed. An offset is encoded as:

- If the offset is less than 128: a `1` bit followed by a 7-bit offset value.
- If the offset is greater than or equal to 128: a `0` bit followed by an 11-bit offset value.

A **length** is encoded as:

Length	Bit encoding
2	00
3	01
4	10
5	1100
6	1101

Length	Bit encoding
7	1110
length > 7	(1111 repeated N times) xxxx, where N is integer result of (length + 7) / 15, and xxxx is length - (N*15 − 7)

For example if length is in [8,22] it gets encoded as `1111 xxxx`, where `xxxx` is $length - 8$, if length is in [23,37] it gets encoded as `1111 1111 xxxx`, where `xxxx` is $length - 23$, and so on.

End marker

An end marker is encoded as the 9-bit token 110000000. Following the end marker, 0 to 7 extra `0` bits are appended as needed, to pad the stream to the next byte boundary.

Example

Given the input file (shown as a sequence of 2 hex digits per byte)

```
30 98 8C 26 3C 23 82 30 38 78 C6 18 00
```

that is in binary

```
0011.0000 1001.1000 1000.1100 0010.0110 0011.1100 0010.0011 1000.0010 0011.0000
0011.1000 0111.1000 1100.0110 0001.1000 0000.0000
```

It must be interpreted as follows:

```
0 01100001        Literal: 97 'a'
0 01100010        Literal: 97 'b'
0 01100001        Literal: 97 'a'
0 01100011        Literal: 97 'c'
1 10000100 01     Off/Len: 4/3
1 10000010 00     Off/Len: 2/2
1 10000001 1100   Off/Len: 1/5
0 01111000        Literal: 120 'x'
1 10001100 00     Off/Len: 12/2
1 10000000        end marker
0000              padding
```

Leading to the following output sequence:

```
abacababaaaaaxca
```