LZVN decoder

Submit Solution

Write a C++ command line program that accepts the following options:

```
1. lzvn_decode <input file> <output file>
```

The program must manage the command line, load a stream compressed with the LZVN algorithm and decode it in the output file.

LZVN is an open-source lossless data compression algorithm created by Apple Inc. It was released together with a more complex algorithm called LZFSE (Lempel–Ziv Finite State Entropy). LZFSE was introduced by Apple at its Worldwide Developer Conference 2015. It shipped with that year's iOS 9 and OS X 10.11 releases.

The general Apple compressed stream is divided into blocks, each with different compression algorithms. The list of possible blocks is:

magic number	block type
`bvx-`	uncompressed
`bvxn`	LZVN compressed
`bvx1`	LZFSE compressed, uncompressed tables
`bvx2`	LZFSE compressed, compressed tables
`bvx$`	end of stream

In this exercise, only LZVN and end of stream will be used. After the `bvxn` header a 32 bits integer (little endian) provides the number of bytes that the output will require, followed by another 32 bits integer (little endian) which contains the block size, that is the number of encoded (source) bytes.

The LZVN stream is made up of commands (*opcodes* in Apple sources) which have a 1-byte size, possibly followed by more bytes. In general, every opcode instructs the decoder to first add some bytes (literals) to the output stream and then to go back in the already decoded stream of a certain *match distance* and append *match length* bytes to the decoded stream. Every opcode differs in the number of bits dedicated to every part (which could be even 0 bits, meaning that the literals or the distance/match information are missing).

The complete list of opcodes is provided in the following tables. Every letter (L, M, D) or number (0, 1) is one bit. Every value is encoded as a binary number from the most significant bit to the least significant bit.

opcode type	format	description
small distance (sml_d)	`LLMMMDDD DDDDDDDD LITERALS`	1) the length of the literal part (0-3 bytes) is encoded by the high 2 bits of the first byte. 2) the match length is encoded with a bias of +3 in the next 3 bits (3-10 bytes). 3) the match distance is encoded in the following 3+8 bits, but the first 3 bits cannot be `110` or `111`, so the effective range is 0-1535.
medium distance (med_d)	`101LLMMM DDDDDDMM DDDDDDDD LITERALS`	1) the length of the literal part (0-3 bytes) is encoded by 2 bits of the first byte. 2) the match length is encoded with a bias of +3 in the next 3 bits (msb) and in two bits (lsb) of the following byte (3-34 bytes). 3) the match distance is encoded in the following 8+6 bits (0-16383). **Notice that the last 8 bits are strangely the most significant part**.
large distance (lrg_d)	`LLMMM111 DDDDDDDD DDDDDDDD LITERALS`	1) the length of the literal part (0-3 bytes) is encoded by the high 2 bits of the first byte. 2) the match length is encoded with a bias of +3 in the next 3 bits (3-10 bytes). 3) the match distance is encoded in the following 8+8 bits (0-65535). **Notice that the last 8 bits are strangely the most significant part**

opcode type	format	description
previous distance (pre_d)	`LLMMM110 LITERALS`	1) the length of the literal part (0-3 bytes) is encoded by the high 2 bits of the first byte. 2) the match length is encoded with a bias of +3 in the next 3 bits (3-10 bytes). 3) the match distance is **not encoded**. It uses the last distance set.
small match (sml_m)	`1111MMMM`	1) there are no literals 2) the match length is encoded without bias in 4 bits (0-15 bytes). 3) the match distance is **not encoded**. It uses the last distance set.
large match (lrg_m)	`11110000 MMMMMMMM`	1) there are no literals 2) the match length is encoded with a bias of +16 in 8 bits (16-271 bytes). 3) the match distance is **not encoded**. It uses the last distance set.
small literal (sml_l)	`1110LLLL LITERALS`	1) the length of the literal part (0-15 bytes) is encoded by the low 4 bits of the first byte. 2) there is no copy, so no match length or distance is encoded. This opcode doesn't change the "previous" distance value.
large literal (lrg_l)	`11100000 LLLLLLLL LITERALS`	1) the length of the literal part is encoded with a bias of +16 by the 8 bits of the second byte (16-271 bytes). 2) there is no copy, so no match length or distance is encoded. This opcode doesn't change the "previous" distance value.
nop	`00001110`	do nothing
nop	`00010110`	do nothing
eos	`00000110 00000000 00000000 00000000 00000000 00000000 00000000 00000000`	end of the stream
udef	`0111xxxx`	should never happen
udef	`1101xxxx`	should never happen
udef	`00011110`	should never happen
udef	`00100110`	should never happen
udef	`00101110`	should never happen
udef	`00110110`	should never happen
udef	`00111110`	should never happen

IMPORTANT NOTE

The decoding of the first byte univocally determines the opcode type, and in the original Apple code, it is realized with a lookup table. Unfortunately, the patterns are not mutually exclusive (*a bad design in my opinion*). If you want to decode the first byte without building a table, first detect the fully specified patterns (8 explicit bits), then the partially specified ones (4 explicit bits), then medium distance, then large and previous distance, and finally small distance.

For your convenience during debugging, the following table lists all bytes and the associated opcodes:

byte	opcode
00000000	sml_d
00000001	sml_d
00000010	sml_d
00000011	sml_d
00000100	sml_d
00000101	sml_d
00000110	eos
00000111	lrg_d

byte	opcode
00001000	sml_d
00001001	sml_d
00001010	sml_d
00001011	sml_d
00001100	sml_d
00001101	sml_d
00001110	nop
00001111	lrg_d
00010000	sml_d
00010001	sml_d
00010010	sml_d
00010011	sml_d
00010100	sml_d
00010101	sml_d
00010110	nop
00010111	lrg_d
00011000	sml_d
00011001	sml_d
00011010	sml_d
00011011	sml_d
00011100	sml_d
00011101	sml_d
00011110	udef
00011111	lrg_d
00100000	sml_d
00100001	sml_d
00100010	sml_d
00100011	sml_d
00100100	sml_d
00100101	sml_d
00100110	udef
00100111	lrg_d
00101000	sml_d
00101001	sml_d
00101010	sml_d
00101011	sml_d
00101100	sml_d
00101101	sml_d
00101110	udef
00101111	lrg_d
00110000	sml_d

byte	opcode
00110001	sml_d
00110010	sml_d
00110011	sml_d
00110100	sml_d
00110101	sml_d
00110110	udef
00110111	lrg_d
00111000	sml_d
00111001	sml_d
00111010	sml_d
00111011	sml_d
00111100	sml_d
00111101	sml_d
00111110	udef
00111111	lrg_d
01000000	sml_d
01000001	sml_d
01000010	sml_d
01000011	sml_d
01000100	sml_d
01000101	sml_d
01000110	pre_d
01000111	lrg_d
01001000	sml_d
01001001	sml_d
01001010	sml_d
01001011	sml_d
01001100	sml_d
01001101	sml_d
01001110	pre_d
01001111	lrg_d
01010000	sml_d
01010001	sml_d
01010010	sml_d
01010011	sml_d
01010100	sml_d
01010101	sml_d
01010110	pre_d
01010111	lrg_d
01011000	sml_d
01011001	sml_d

byte	opcode
01011010	sml_d
01011011	sml_d
01011100	sml_d
01011101	sml_d
01011110	pre_d
01011111	lrg_d
01100000	sml_d
01100001	sml_d
01100010	sml_d
01100011	sml_d
01100100	sml_d
01100101	sml_d
01100110	pre_d
01100111	lrg_d
01101000	sml_d
01101001	sml_d
01101010	sml_d
01101011	sml_d
01101100	sml_d
01101101	sml_d
01101110	pre_d
01101111	lrg_d
01110000	udef
01110001	udef
01110010	udef
01110011	udef
01110100	udef
01110101	udef
01110110	udef
01110111	udef
01111000	udef
01111001	udef
01111010	udef
01111011	udef
01111100	udef
01111101	udef
01111110	udef
01111111	udef
10000000	sml_d
10000001	sml_d
10000010	sml_d

byte	opcode
10000011	sml_d
10000100	sml_d
10000101	sml_d
10000110	pre_d
10000111	lrg_d
10001000	sml_d
10001001	sml_d
10001010	sml_d
10001011	sml_d
10001100	sml_d
10001101	sml_d
10001110	pre_d
10001111	lrg_d
10010000	sml_d
10010001	sml_d
10010010	sml_d
10010011	sml_d
10010100	sml_d
10010101	sml_d
10010110	pre_d
10010111	lrg_d
10011000	sml_d
10011001	sml_d
10011010	sml_d
10011011	sml_d
10011100	sml_d
10011101	sml_d
10011110	pre_d
10011111	lrg_d
10100000	med_d
10100001	med_d
10100010	med_d
10100011	med_d
10100100	med_d
10100101	med_d
10100110	med_d
10100111	med_d
10101000	med_d
10101001	med_d
10101010	med_d
10101011	med_d

byte	opcode
10101100	med_d
10101101	med_d
10101110	med_d
10101111	med_d
10110000	med_d
10110001	med_d
10110010	med_d
10110011	med_d
10110100	med_d
10110101	med_d
10110110	med_d
10110111	med_d
10111000	med_d
10111001	med_d
10111010	med_d
10111011	med_d
10111100	med_d
10111101	med_d
10111110	med_d
10111111	med_d
11000000	sml_d
11000001	sml_d
11000010	sml_d
11000011	sml_d
11000100	sml_d
11000101	sml_d
11000110	pre_d
11000111	lrg_d
11001000	sml_d
11001001	sml_d
11001010	sml_d
11001011	sml_d
11001100	sml_d
11001101	sml_d
11001110	pre_d
11001111	lrg_d
11010000	udef
11010001	udef
11010010	udef
11010011	udef
11010100	udef

byte	opcode
11010101	udef
11010110	udef
11010111	udef
11011000	udef
11011001	udef
11011010	udef
11011011	udef
11011100	udef
11011101	udef
11011110	udef
11011111	udef
11100000	lrg_l
11100001	sml_l
11100010	sml_l
11100011	sml_l
11100100	sml_l
11100101	sml_l
11100110	sml_l
11100111	sml_l
11101000	sml_l
11101001	sml_l
11101010	sml_l
11101011	sml_l
11101100	sml_l
11101101	sml_l
11101110	sml_l
11101111	sml_l
11110000	lrg_m
11110001	sml_m
11110010	sml_m
11110011	sml_m
11110100	sml_m
11110101	sml_m
11110110	sml_m
11110111	sml_m
11111000	sml_m
11111001	sml_m
11111010	sml_m
11111011	sml_m
11111100	sml_m
11111101	sml_m

byte	opcode
11111110	sml_m
11111111	sml_m