Information Theory

Assignment 2
Deflate

Miguel Martins Duarte - 2012139146
Guilherme Graça – 2012138932
Informatics Engineering
University Of Coimbra

General Information:

All the code was programmed using C++.

The routines used and the main function are commented, specifying what every piece of code does.

Implementation;

I will explain the general implemtation of the algorithm.

The first thing that needs to be done for each block is to read HLIT, HDIST and HCLEN. To read them, we created a function to read a certain amount of bits. That is made by storing the already read bits into a integer, and every time with to not have there the needed amout of bits, we read one more byte and add it to our int. To select the needed bits, its used a simple mask.

Using the function implemented earlier, its easy to read the given sequence of HCLEN + 4 numbers, each one using 3 bits. This groups of bits are the lengths of the huffman codes that are used to read the information that will be needed to create the huffman trees that are used to read the data itself.

The code lengths are given in a strange order (16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15), but the order is that because they are ordered by how common each symbol is.

Now, using the length of the huffman codes, we have to obtain their value. That is made following a simple algorithm.

First, we initialize a counter to zero. We loop through all the possible code lengths, starting at one. Then we loop through all the lengths and for all that match we set them to the value of the counter and increment it. In the end of this loop, we shift the counter one bit to the left. This algorithm can be used with certainty that no huffman code is a prefix of another (unless something is wrong with the file). This algorithm is implemented in our function length2intcode.

Using the codes obtained right now, we can create a huffman tree. That is made by using the code implemented by the teacher, using is struct HuffmanTree, and in this case, the functions createTree and addNode. It's also needed to use an auxiliary function to create strings from the codes, in the integer type. We define a function to do this, intcodes2tree, and will be used later on.

Having this Huffman tree initialized, we can start reading from the file using it. First, the literals/lengths codes (what is read is the the code lengths, the codes themselves will be obtained later). The reading process is done reading one bit at a time a going left or right acording to it. We use the nextNode routine provided. When we arrive to a leaf, we get the value itself from the tree and write it in our array. To implement this reading, with created the function read_using_hufftree.

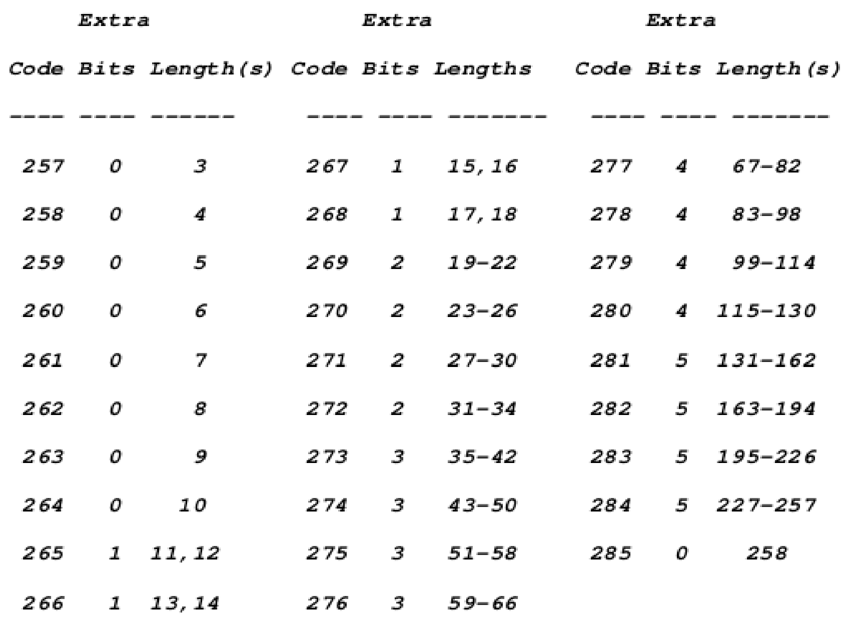
Second, the distance codes (we think there is a mistake in the text of the exercise). The tree is the same, and we can use the routine we have implemented.

Now we have the length of the huffman codes. To create the real codes, we can use the function defined in exercise 3, length2intcode, for both arrays previously read.

Using this codes, we create two trees, one for literal/length codes, and one for distance codes. That is made using our other routine from exercise 4, intcodes2tree.

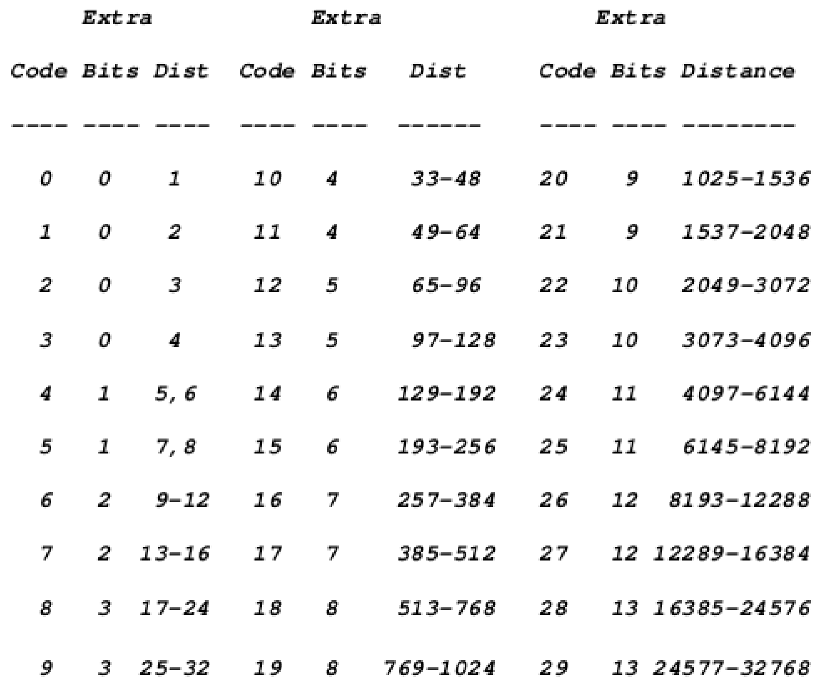
We now have everything that is needed to read the data itself, using our two trees. The deflate algorithm is simple:

- Read bit by bit until we get to a leaf. We use the routine read_using_hufftree for that. Then we get the value in the leaf.
- If the value read is lower than 256, we have the ascii value itself, and we can place it in the output buffer.
- If the value is 256, than it's the end of the block.
- If the value is greater than 256, then we expect a length/distance pair, similarly with lz77 algorithm. To get the length value, it might be needed to read some more bits. The length is calculated adding the extra_bits with a value extracted from the literal/length value, varying from 257 to 285. Our algorithm follows the following table:



	Extra			Extra			Extra	
Code	Bits	Length(s)	Code	Bits	Lengths	Code	Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

After reading the extra bits and determining the length, we read the distance. Now, we use the distances tree and read bit by bit. According to the value in the leaf, we might need to read some extra bits to know the distance. We use this table:



	Extra			Extra			Extra	
Code	Bits	Dist	Code	Bits	Dist	Code	Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

To calculate the length and the distance we “hardcode” the base distance for each code into an array.
Now that we have the length, we can go back in our buffer the number of characters specified by the distance and we copy the number of characters specified by length to the current position in the buffer.

It's almost done. We have only to print our buffer (that was allocated with the size of the file) to the final file. Our previous code was placed in the main do while loop, so it runs for every block of the file.