UVIC
Summer 2023
SENG 440: Embedded Systems

Huffman Encoding and Decoding Optimization for the S3C2440A ARM Machine

Group Members:
Salem Ait Ami (V00932871)
salemaitami@uvic.ca

Adam Tidball (V00936605)
adamtidball@uvic.ca

Submission Date: August 4th 2023

Table of Contents

1.0 - Introduction

In an increasingly digital world, where vast amounts of data are sent over the internet every second, strategies like Huffman coding are used to ensure data is transferred and stored efficiently [1]. Huffman coding is a lossless data compression technique created by David Huffman in the 1950's when he was studying electrical engineering at MIT [2]. Huffman's coding techniques are still utilized in many modern day applications such as image compression solutions like MPEG. The revolutionary idea of Huffman coding was the ability to assign variable length codes to symbols based on the symbols probability of occurrence, as a way to reduce the overall bit-rate of a system [3]. A reduced bit-rate is achieved by assigning shorter length codes to the most probable symbols and reserving longer codes for the less common symbols, the resulting encoded data is smaller on average than it would have been if it was coded using codes of the same bit length.

In this report, the implementation of Huffman Encoding and Decoding will be optimized for use on an S3C2440A ARM Machine. Optimization is a broad term, so in order to focus the scope of this project and provide context on project structure, there are a few critical specifications that must be clarified:

- Firstly, the majority of work was done on optimizing the relatively more complex Huffman decoding process rather than the encoding process.
- The optimization techniques used were designed for the S3C2440A ARM Machine and may not be transferable to machines with different architecture.
- Any optimized solution must not limit the functionality of a Huffman Encoding and Decoding solution. To ensure this requirement is met, correctness and performance metrics were used to analyze proposed optimizations.
- Finally, while hardware and firmware optimization solutions do exist, the primary optimizations in this project were done in the software, using C and assembly.

The report itself is structured as follows: Firstly, background information about Huffman Encoding and Decoding, as well as information about the microprocessor used in this project will be provided to give context about the solution. The majority of the report will discuss the optimization designs attempted and the rationale for why they could lead to an improvement in performance. The next section will discuss the performance metrics for the optimized solution using a naive solution as a baseline to evaluate our optimizations against. Finally, the results will be discussed in a conclusion along with higher level discussion about what our optimization solution did successfully and what could be improved.

2.0 - Theoretical Background

The following section discusses background information about the implementation of a Huffman Encoding and Decoding solution, as well as information about the microcontroller used in this project. It is the goal of this section to provide context about the project before the design process is discussed further.

2.1 - Huffman Encoding

Lossless encoding is the process of taking input data and transforming it into an encoded representation that can be decoded back into the original input, without losing any information about the original data. In order to accomplish this goal, there are two main components that must be present before any data can be encoded: The input data itself, and all the symbols in the input data along with their respective Huffman codes.

The unique part about Huffman encoding, compared to other basic approaches, is that the codes are derived from each symbols' probability of occurrence. Therefore, to encode data using Huffman encoding the first step is determining the probabilities of each symbol in the sample. The Huffman encoding process can be broken down into the following steps:

1) Determine the probability of each symbol in the input data alphabet
2) Create a symbol and code key based on the symbol probabilities
3) Use the symbol code key to encode the input text

For the purposes of providing background an example encoding the word “GALAXY” will be used to illustrate the Huffman encoding process. The following symbols and their probabilities can be seen below:

A - 0.4	**G** - 0.2	**L** - 0.25	**X** - 0.05	**Y** - 0.1

Figure 1 *- Example Alphabet and Probabilities*

In this example the symbols’ probabilities were chosen arbitrarily. However, calculating the real probabilities based on the input file or using credible probability data from the same alphabet is preferable for bit-rate reduction.

The most influential step of the encoding process is actually creating the unique codes for each symbol, rather than generating the encoding itself. Once the “alphabet” is determined, generating the encoding is a trivial process. To accomplish this, a minimum heap is created according to symbol probabilities, which is then used to create a

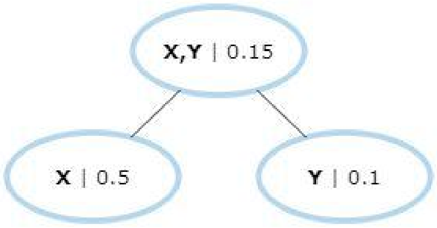
Huffman tree, where the path traversed to a leaf determines codes. The following algorithm is utilized to sort the values into a Huffman tree:

1. Pop the two nodes with the minimum frequency from the minimum heap.
2. Sum the two nodes' probabilities and add the result as a new internal node.
3. Make the first popped node the new node's left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat until the heap contains only one node, which is the root of the tree [4].

Continuing with the example of encoding the word "GALAXY", we can illustrate how a Huffman tree for our symbol probability pairs is created. The following figure shows the Huffman tree creation, based on our provided probabilities, as well as how the minimum heap changes as the tree is made:

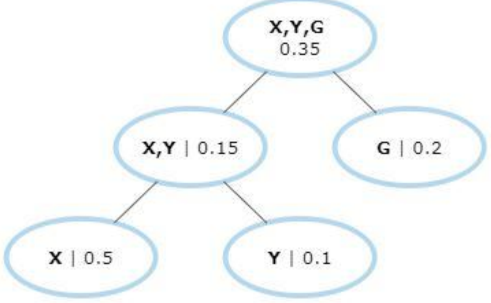
(1)

Node	Prob
X	0.05
Y	0.1
G	0.2
L	0.25
A	0.4



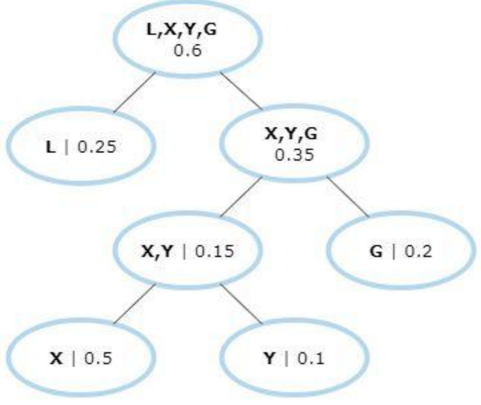
(2)

Node	Prob
X,Y	0.15
G	0.2
L	0.25
A	0.4



(3)

Node	Prob
L	0.25
X, Y, G	0.35
A	0.4



(4)

Node	Prob
A	0.4
L, X, Y, G	0.6

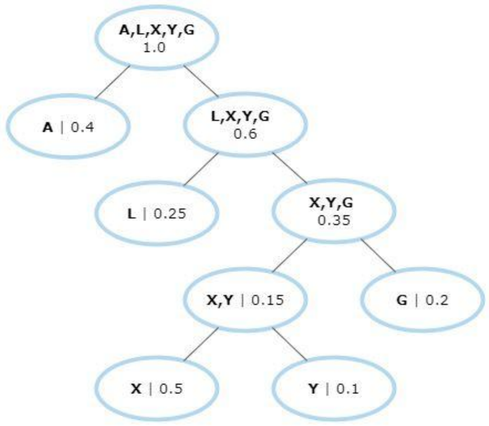


***Figure 2** - Example Huffman Tree Creation and Min Heap*

Notice how the resulting Huffman tree contains all of the symbols in the input data as leaf nodes at the bottom of the tree. This inherent property is what is used to define the Huffman codes for every leaf. That is, the path taken to reach a particular leaf node determines its Huffman code; where a left-child traversal corresponds to 0 and a right-child traversal corresponds to a 1. A complete table of the symbols in example distribution in Figure 1, along with their respective codes, and the Huffman tree that was traversed to determine those codes can be seen below:

Symbol	Huffman Code	Bit Length
A	0	1
L	10	2
G	111	3
X	1100	4
Y	1101	4

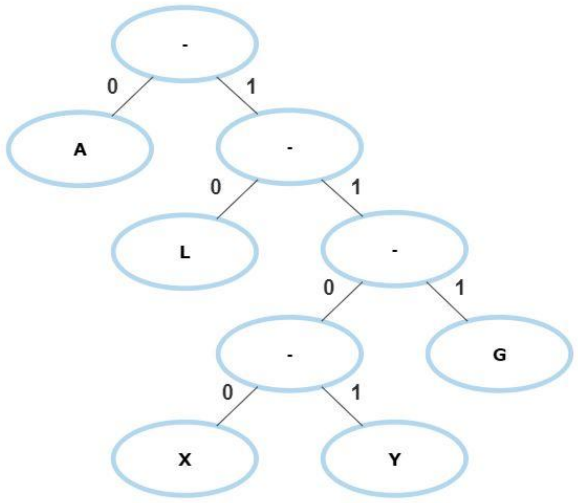


Figure 3 *- Example Huffman Codes and Complete Tree*

The final step of the encoding process involves using this symbol code key table, also known as a look up table, to encode the input. Using this table we can encode the word “GALAXY” to the following: “111010011001101”.

2.2 - Huffman Decoding

Huffman decoding is a fairly straightforward process at first, but it becomes a much more complicated issue when optimizations are considered. One of the main factors for this complexity has to do with the fact that there are no guard bits to separate encodings in the compressed data. That is, there is no way to determine the length of a given code until it is decoded in a bit-by-bit manner. In other words, it’s impossible to start decoding the next symbol until the length of the current symbol is known. If decoding is being done in a bit-by-bit manner, then the symbol length is only known once the symbol is fully decoded. Essentially, Huffman decoding is a completely sequential process with no opportunity for parallelization.

There are two different approaches to the Huffman Decoding process that are covered in this report. The first is a naive solution that involves traversing the Huffman Tree by reading the encoded data bit-by-bit. The second approach creates look up tables, by traversing the Huffman tree, before starting the decoding process. Both approaches are discussed below:

2.2.1 - Tree Decoding

For the bit-by-bit approach, three components are needed to perform the decoding: The compressed data, the length of the compressed data in bits, and the Huffman tree used to encode the data. The compressed data is iterated through, one bit at a time, where each bit determines whether the left child, or the right child of the current node is traversed [5]. The algorithm completes the following steps:

1. Set "cur" to point to the root of the Huffman Tree.
2. Read 1 bit from the encoded stream.
    a. If it is a 1, set "cur" to point to its right child.
    b. Else, set "cur" to point to its left child
3. Determine if "cur" is pointing to a leaf node.
    a. If true, retrieve the symbol stored at the node
    b. If false, go to step 2
4. Determine if there is more encoded data in the stream.
    a. If true, go to step 1
    b. If false, return

Because the Tree Decoding approach has to iterate through the encoded data bit-by-bit it is not viewed as the optimal decoding technique. However, it acts as a good baseline for comparing other decoding strategies.

2.2.2 - Look Up Table Decoding

The second and more complex decoding approach implemented in this project is Look Up Table Decoding. For this approach, two main components are needed to complete the decoding: The encoded data and the Look Up Tables. However, in order to create the Look Up Tables, the symbols along with their respective Huffman codes are required.

The Look Up Tables themselves have some important properties. Firstly, the tables have two columns, one stores the symbol and the other stores the code length of that symbol. The symbols in the table are indexed according to the binary value of that symbol's code. Using the previous example of encoding the symbols for the input "GALAXY" we can see what a LUT would look like based on the the following symbol code pairs:

Symbol	Huffman Code	Bit Length
A	0	1
L	10	2
G	111	3
X	1100	4
Y	1101	4

***Figure 4** - Example Huffman Codes*

LUT Part One			LUT Part Two		
Index [binary]	**Symbol**	**Bit Length**	**Index [binary]**	**Symbol**	**Bit Length**
0000	A	1	1000	L	2
0001	A	1	1001	L	2
0010	A	1	1010	L	2
0011	A	1	1011	L	2
0100	A	1	1100	X	4
0101	A	1	1101	Y	4
0110	A	1	1110	G	3
0111	A	1	1111	G	3

***Figure 5** - Example Look Up Table*

Notice that in figure 5, the symbols with code lengths shorter than the maximum code length have multiple entries in the table. This is because during the decoding process maximum code length sized sections of the encoded data are used to select the index of the table. In other words any possible combination of bit values after the shorter codes must still lead to the same symbol. An example of how the Look Up Table decoding would decode the “L” in “GALAXY” can be seen in the figure below:

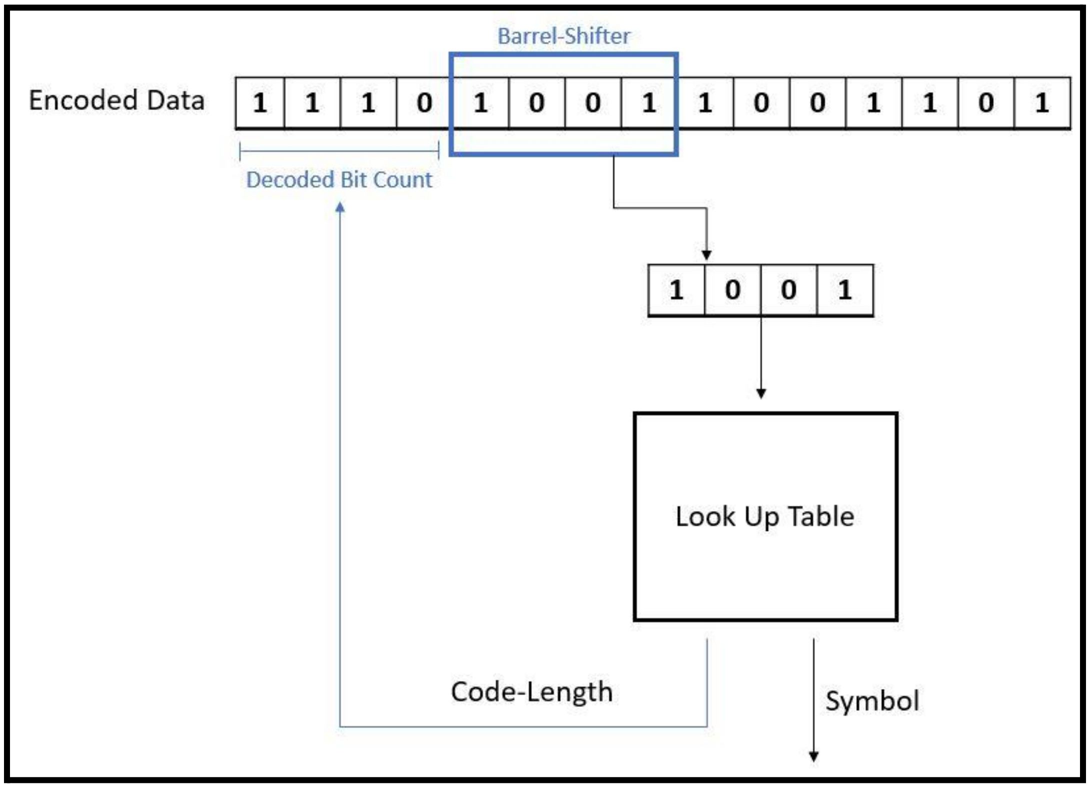


Figure 6 *- Barrel Shifting LUT Decoding*

Another important point to mention concerns the amount of entries in a Look Up Table. The amount of required entries is determined by the bit length of the longest symbol’s code. This is because the index of each table is related to the binary value of the symbols code, due to the fact the indexes range from 0 to the value of the maximum length code where every bit is set to 1. In the case of the “GALAXY” example, the maximum code length is 4 bits, this means the table indexes will range from zero to “1111” in binary. Based on this we know there will be 2^4 = 16 entries in the Look Up Tables. Because large alphabets generally have larger code lengths that can generate massive Look Up Tables, the Look Up Table solution is preferable for smaller alphabet data decoding. The negative impacts of large Look Up tables is that they require a lot of memory to store and must be broken up into smaller tables that can fit into cache memory, and even then if there are many smaller tables that cannot be stored in cache at the same time the chances of cache misses increase which hurts efficiency.

2.3 - Transmission Rate Theory

The purpose of encoding and decoding data using the Huffman approach is to reduce the bit-rate of the system and therefore improve transfer and storage efficiency. The theory behind this is that the encoded data will contain fewer bits than the original data while retaining the same information, thus increasing the ratio of useful data stored in a bit which improves the bit-rate.

In the previous sections we used the example of encoding and decoding the word “GALAXY”, let's continue with this example to prove the bit-rate reduction of Huffman encoding. The figure below compare the bit rates of the example Huffman approach with the bit rate of a standard same size code approach:

Required bits without Huffman Encoding

Symbol	Huffman Code	Bit Length
A	000	3
L	001	3
G	010	3
X	011	3
Y	100	3

Average bits per symbol = 3

Required bits with Huffman Encoding

Symbol	Huffman Code	Prob	Bit Length
A	0	0.4	1
L	10	0.25	2
G	111	0.2	3
X	1100	0.1	4
Y	1101	0.05	4

Average bits per symbol ≈ 2

$(0.4*1)+(0.25*2)+(0.2*3)+(0.1*4)+(0.05*4) = 2.1 \approx 2$

Figure 7 *- Required Bits for Alphabet with 5 Symbols*

As you can see from figure 7, the average bits required per symbol is 3 without the Huffman approach and approximately 2 when Huffman is used. It should be mentioned that this metric does not capture the efficiency cost of encoding and decoding the Huffman data. Which takes non negligible time and should be considered when assigning to over efficiency of the system.

2.4 - ARM Machine

The hardware that was used to run and test the code created during the project was the Uvic provided ARM machine: S3C2440A 32-BIT CMOS microcontroller. This microcontroller is developed with a ARM920T core which is a 16/32 bit RISC processor with separate 16KB instruction and 16KB data caches, each with an 8-word line, or 32 byte, length [6].

3.0 - Design Process

The following section discusses the different design and optimization approaches that were attempted in this project. The following sections are ordered sequentially, beginning with the initial code structure design, moving to code optimization, and finishing with investigating hardware optimizations.

3.1 - Initial Huffman Design

One encoding solution was designed in this project and two distinct decoding solutions were designed. Each of these designs are described below:

3.1.1 - Encoding Design

The huffman encoding design can be broken up into 3 main sections that are all separated in their own functions in the C code. The functions and their descriptions can be seen below:

1. **buildHuffmanEncTree** - This function takes in an alphabet of symbols and their respective probabilities and creates the Huffman tree. In our implementation this function returns the root node of the tree which is used for several different functions throughout the program.
2. **generatehuffmanCodesBitified** - This function takes a Huffman root node as input and recursively traverses that tree to generate the Huffman codes for each symbol in the tree. The symbols are stored in a Look-Up Table.
3. **GenerateEncodedFileBitified** - This function performs the Huffman encoding on a provided input text and stores the result in a ".huf" file. The Huffman encoding is done using a table containing the codes that was generated by the "generatehuffmanCodesBitified" function.

3.1.2 - Tree Decoding Design

The first decoding solution created was the Huffman tree decoding. The tree decoding solution was done in one function in the C code, that function is described below:

1. **treeDecodingBitByBit** - This function decodes encoded data from a compressed file by traversing a provided Huffman tree according to the values read bit by bit from the encoded data.

3.1.3 - Look Up Table Decoding Design

The Look Up Table implementation can be broken up into four distinct sections, which are each represented as functions inC code. Each of these functions and a description of what they do can be seen below, listed in sequential order:

1. **lutCreation** - This function allocates memory for the look-up tables based on the number of unique symbols in the original message. The number of look up tables that are created is provided as input by the user which indirectly determines the size of each individual look-up table.
2. **lutPopulate** - This function is called by the LutCreation function and uses the existing tree of symbols to fill in the look-up tables. Each table has two columns, one for the symbol and one for code length. Each index corresponds to the value of that symbol's code.
3. **lutDecoding** - This function uses the populated look-up tables to decode a given message by using the maximum code length sized indexes and then iterating through the message by the code length of each symbol it decodes.
4. **lutFreeAll** - This function simply frees the memory of all of the look-up tables.

3.2 - Code Optimizations

The following section contains information about code optimizations that were attempted in this project, this focuses on C code optimizations. Assembly code was generated using an arm-linux-gcc compiler. The complete C code can be seen in Appendix E and F. It should be noted that some of the following optimization attempts did not result in large increases of efficiency, but will be discussed regardless.

3.2.1 - Implementing BARR-C Standards

One of the first changes to our original Huffman encoding and decoding solution we made was to implement BARR-C code standards to the best of our ability. This meant going through all of the code and replacing standard variable definitions like “unsigned int” or “float” with “uint16_t” or “int32_t” declarations.

Original Code	Optimized Code
unsigned int tables_needed = 0;	uint32_t tables_needed = 0;

Figure 8 *- Example of BARR-C Standards*

3.2.2 - Declaring Register Variables

Another optimization technique implemented on the C code was defining loop counter variables as registers. This is an exception to the BARR-C code standards so they recommend not using register declarations, however, we decided to use them for loop counter variables, and certain constants. It should be noted that the register keyword only serves as a suggestion to the compiler, and at the end of the day the decision is up to the compiler as to whether or not it stores a variable as a register.

Original Code

```
int i;
for(i = 0; i < (*tables_needed); i++){
    free(all_luts[i]);
}
```

Optimized Code

```
register uint32_t i;
for(i = 0; i < (*tables_needed); i++){
    free(all_luts[i]);
}
```

Assembly

```
main:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #12
        mov     r3, #10
        str     r3, [fp, #-12]
        mov     r3, #0
        str     r3, [fp, #-8]
        b       .L2
.L3:
        ldr     r3, [fp, #-8]
        add     r3, r3, #1
        str     r3, [fp, #-8]
.L2:
        ldr     r3, [fp, #-12]
        ldr     r2, [r3, #0]
        ldr     r3, [fp, #-8]
        cmp     r2, r3
        bgt     .L3
        mov     r0, r3
        add     sp, fp, #0
        ldmfd   sp!, {fp}
        bx      lr
```

Optimized Assembly

```
main:
        stmfd   sp!, {r4, fp}
        add     fp, sp, #4
        sub     sp, sp, #8
        mov     r3, #10
        str     r3, [fp, #-8]
        mov     r4, #0
        b       .L2
.L3:
        add     r4, r4, #1
.L2:
        ldr     r3, [fp, #-8]
        ldr     r3, [r3, #0]
        cmp     r3, r4
        bgt     .L3
        mov     r0, r3
        sub     sp, fp, #4
        ldmfd   sp!, {r4, fp}
        bx      lr
```

***Figure 9** - Example of Register Variables & Assembly*

3.2.3 - Bit Shifting Operations

Again this is a C code level optimization technique that we utilized. We tried to reduce the number of computationally expensive operations like multiplications and divisions by replacing them with equivalent bit shifting operations. Another example of utilizing bit shifting would be to replace modulo two operations with simply checking the value of the first bit of the variable.

Original Code	Optimized Code
size = size / sizeof(uint16_t);	size >>= 1;

***Figure 10** - Example Bit Shifting Operations*

3.2.4 - Precomputed Constants

Since division is a costly operation, we did our best to avoid performing unnecessary divisions. There were not many opportunities for using precomputed values throughout our code, but the most obvious one was at the end of the prologue function, where we calculated probabilities from the accumulated frequencies. By precomputing the inverse sample size and storing it as a register, we can almost certainly guarantee better performance as multiplication is a cheaper operation than division and we only need to perform a singular division. It should be noted that this improvement in performance is negligible, as symbol_count will never exceed 128. Therefore, we don't spend enough time in the loop to achieve a noticeable difference in performance. However, in the spirit of this course we thought it best to try and experiment with as many different optimization strategies as we could.

Original Code	**Optimized Code**

`uint32_t j;`
`for (j = 0; j < symbol_count; j++) {`
`    probabilities[j] = frequencies[j]/(*sample_size)`
`}`
	`register double inv_sample_size = *sample_size;`
	`inv_sample_size = 1/inv_sample_size;`
	`register uint32_t j;`
	`for (j = 0; j < symbol_count; j++) {`
	`    probabilities[j] = frequencies[j]*inv_sample_size;`
	`}`

Figure 11 *- Example Precomputed value*

3.2.5 - Branch Reduction

The branch reduction strategies that we explored were loop unrolling, loop grafting and reducing redundant else if statements. When first attempting the loop optimization we were skeptical whether or not it would improve code efficiency because Huffman decoding is a sequential process. For this reason we focused on loop optimizations on the prologue and encoding sections of the code when looking to implement loop optimizations. Reducing redundant else if statements was an approach we were able to apply primarily to the decoding functions.

When looking for loops to apply loop unrolling to, we found that the loops that are used in the program are quite large and for that reason had the extra possibility of unrolling causing register overflow. For this reason loop unrolling was not explored in much detail and no substantial changes were implemented. For loop grafting it was also difficult to determine where it could be implemented in the code. Implementing it in the prologue function, which determines the symbol count and probabilities of the alphabet, was the only place we deemed possible. We grafted a small probability calculation loop onto the larger frequency determining loop. This loop grafting implementation required extra multiplication steps to account for incrementing symbol count after a probability has already been added to the index. The implementation also sacrificed a bit of accuracy of the calculated probabilities as it based the probabilities on the total symbol count at the time the character was found. In the end the full implementation was decided not to be included in the final optimized code, however the attempt can be seen below:

Original Code

```
    uint32_t i;
    for (i = 0; i < *sample_size; i++) {
cur_symbol = sample[i];
if (symbol_count > MAX_SYMBOLS) {
        printf("Symbol count exceeded!\n");
        exit(1);
   }
   else if (symbolIndex(symbols,
symbol_count, cur_symbol) == -1) {
        symbols[symbol_count] = cur_symbol;
        frequencies[symbol_count] = 1;
        symbol_count++;
   }
   else {
        frequencies[symbolIndex(symbols,
symbol_count, cur_symbol)]  += 1;
  }
}

double inv_sample_size = *sample_size;
inv_sample_size = 1/inv_sample_size;
register uint32_t j;
    for (j = 0; j < symbol_count; j++) {
         probabilities[j] =
frequencies[j]*inv_sample_size;
}
```

Attempted Optimized Code

```
double inv_sample_size = *sample_size;
inv_sample_size = 1/inv_sample_size;
register uint32_t i;
    for (i = 0; i < *sample_size; i++) {
cur_symbol = sample[i];
if (symbol_count > MAX_SYMBOLS) {
        printf("Symbol count exceeded!\n");
        exit(1);
   }
   else if (symbolIndex(symbols,
symbol_count, cur_symbol) == -1) {
        symbols[symbol_count] = cur_symbol;
        frequencies[symbol_count] = 1;
        probabilities[symbol_count] =
frequencies[symbol_count]*inv_sample_size;
        symbol_count++;
   }
   else {
        frequencies[symbolIndex(symbols,
symbol_count, cur_symbol)]  += 1;
        probabilities[symbol_count] =
frequencies[symbol_count]*(*sample_size);
        probabilities[symbol_count] =
frequencies[symbol_count] + 1;
        probabilities[symbol_count] =
frequencies[symbol_count]*inv_sample_size;
   }
}
```

Figure 12 *- Loop Grafting Optimization*

The second strategy used for branch reduction was reducing redundant else if statements. Redundant else if statements result in extra branch statements in the assembly code which can negatively impact performance. By reducing the number of else if statements the goal was to reduce the total branches in the final assembly code. These changes can be seen below in figure 12:

Original Code

```
for (int i = 0;i < barrel_shifter;
i++) {
    encoded_section <<= 1;
    if (encoded[i]  == 1) {
        encoded_section |= 1;
    }
    else if(encoded[i] == 0) {
        encoded_section |= 0;
    }
    else {
        printf("Tried to convert non 1
or 0 to bit!\n");
        exit(1);
    }
}
```

Optimized Code

```
register uint32_t j;
for (j = decoded_bit_count; j <
barrel_shifter + decoded_bit_count;
j++) {

    if (compressed[cur_index] & (1U <<
bit_pos )) {
         compressed_section |= 1u <<
((barrel_shifter - 1) - (j -
decoded_bit_count));
    }
    bit_pos++;
    if(bit_pos >= 16){
          bit_pos = 0;
          cur_index++;
    }
}
```

Figure 13 *- Example "else if" Statement Reduction*

3.2.6 - Memory Organization

For memory organization, a range of optimization strategies were implemented. The first of these strategies was in the choice of data structure used for the Look Up Tables. An array data structure was chosen instead of a linked list due to the fact arrays provide better spatial locality in memory.

A second memory optimization strategy was utilizing '#defines' and 'consts'. '#defines' were used for constraining variables that were not edited during the program such as the `MAX_ROWS_PER_TABLE` and `MAX_FILE_LENGTH`. This utilizes preprocessing to execute statements so that the constraints do not consume any memory space in the final executable.

A final memory organization optimization strategy that was employed was to implement a custom fwrite() function. This was done after testing revealed that calls to fwrite() were storing data to the internal buffer without flushing it. This impacted the final output of the program depending on what machine it was run on, as the internal buffer size for fwrite() is implementation dependent and there is no way to directly retrieve that information, programmatically. We also didn't want to perpetually fwrite() into fflush() during the decoding loops, as it would incur a lot of I/O overhead. Our solution was to create a wrapper function called custom_fwrite(), that would perform writes at a set chunk_size until all data was exhausted. This function is called at the end of the decoding functions, and though it may incur some overhead, it guarantees the correct result for decoding regardless of platform or architecture. The function can be seen below:

```
void custom_fwrite(const void* array, uint32_t element_size, uint32_t
total_elements, FILE* file) {
    uint32_t elements_written = 0;
    uint32_t chunk_size = 32000;

    while (elements_written < total_elements) {
        uint32_t remaining_elements = total_elements - elements_written;
        uint32_t elements_to_write = (remaining_elements < chunk_size) ?
remaining_elements : chunk_size;
        uint32_t bytes_written = fwrite(array + (elements_written *
element_size), element_size, elements_to_write, file);
        assert(bytes_written == elements_to_write);

        fflush(file);
        elements_written += elements_to_write;
    }
}
```

Figure 14 *- Custom fwrite() Function*

3.2.7 - Assembly inlining

The final software optimization approach that was attempted was utilizing assembly inlining. Within the Look Up Table decoding section of the code there is a calculation for determining the table number and table index of the barrel shifted section of code. These modulo and division operations are computationally expensive and need to be executed for each symbol in the message. These factors made this section of code an ideal candidate for assembly inlining, so that the operations could be completed by the compiler more efficiently. The original code and optimized code can be seen below:

Original Code	**Optimized Code**

table_index = compressed_section % max;
table_num = compressed_section / max;
	__asm__ __volatile__ (
	"mov r0, %[dividend]\n\t"
	"mov r1, %[divisor]\n\t"
	"mov r2, #0\n\t"
	"udiv r3, r0, r1\n\t"
	"mov %[quotient], r3\n\t"
	"mov %[remainder], r2\n\t"
	: [quotient] "=r" (table_index), [remainder] "=r" (table_num)
	: [dividend] "r" (compressed_section), [divisor] "r" (max)
	: "r0", "r1", "r2", "r3");

Assembly	**Optimized Assembly**

```
ldr     r3, [fp, #-16]
mov     r0, r3
ldr     r1, [fp, #-20]
bl      __aeabi_idivmod
mov     r3, r1
str     r3, [fp, #-8]
ldr     r0, [fp, #-16]
ldr     r1, [fp, #-20]
bl      __aeabi_idiv
mov     r3, r0
str     r3, [fp, #-12]
```

```
ldr     ip, [fp, #-24]
ldr     r4, [fp, #-28]
mov r0, ip
mov r1, r4
mov r2, #0
udiv r3, r0, r1
mov r5, r3
mov r4, r2

str     r5, [fp, #-16]
str     r4, [fp, #-20]
```

Figure 15 *- Example Assembly inlining*

3.3 - Theoretical Hardware Optimizations

The hardware optimizations explored in this project are done so from the perspective of determining the theoretical optimum hardware that could be used for Huffman Decoding. Throughout optimizing the software it was confirmed that cache memory is a critical part of making a Huffman encoding and decoding solution efficient. There are two ways in which cache improvements would also improve Huffman coding performance: Firstly, increasing the available cache. If there is more cache memory that would be larger Look Up Tables could be stored in the cache which would in theory reduce the likelihood for cache misses thus increasing the efficiency of the system. The second way in which cache could be improved is in its memory access speed. The faster the processor can read from cache the better.

4.0 - Performance and Cost Analysis

Performance analysis was completed using two techniques: analyzing the created assembly code and performing tests using a range of different text files as input. For the assembly analysis the assembly code from the unoptimized and optimized solutions were compared. Special interest was paid to the number of load and store instructions in each solution as well as the number of branching operations.

For the text file performance tests, a total of 9 test inputs were used and their results were analyzed together. These test files will be referred to as samples, as that is what they are labeled in the code. The samples were compared in three different ways:

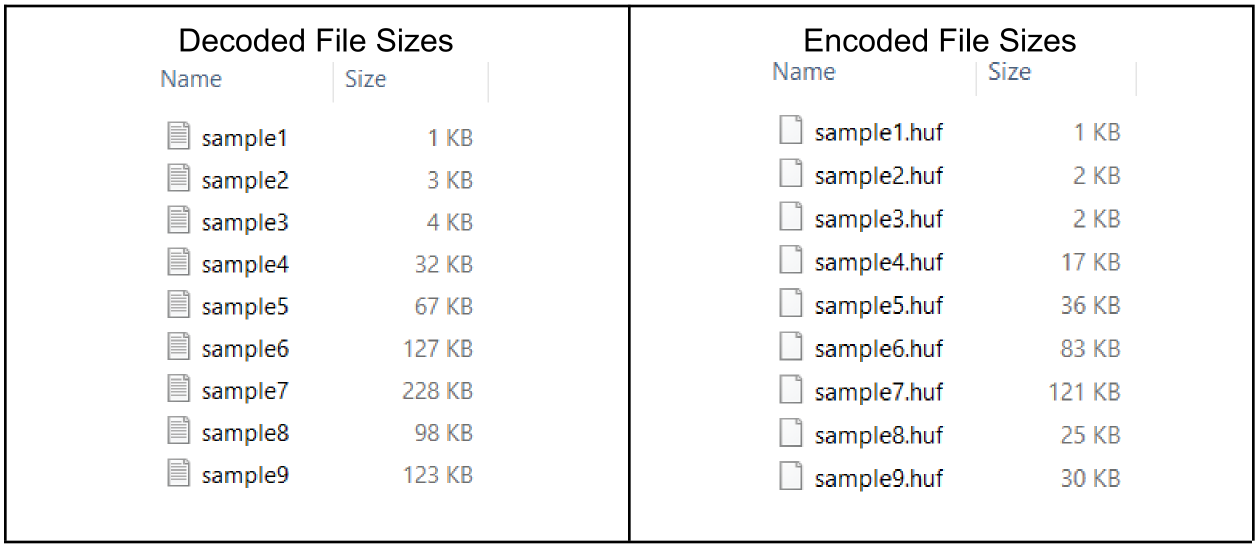
1. Comparing compression and decompression file sizes

2. Comparing cache efficiency metrics (using cachegrind)
3. Profiling Time spent in each function (using gprof)

Each of these metrics will be discussed individually in the following sections:

4.1 - Comparing Compression and Decompression File Sizes

The first metric that we looked at when assessing the performance of the Huffman Encoding and Decoding solution was the file sizes of the created files. Because Huffman Encoding and Decoding is a lossless coding technique we wanted to ensure that the unencoded input file and decoded output file were the same size and had the same content. Another size metric that we looked at was the size of the compressed file compared to the input file. We wanted to ensure that the compressed file was in fact smaller than the original file, as the whole point of Huffman encoding is to reduce the bit rate by reducing the data size. The screenshot of the file comparison for the optimized code can be seen below:



Decoded File Sizes

Name	Size
sample1	1 KB
sample2	3 KB
sample3	4 KB
sample4	32 KB
sample5	67 KB
sample6	127 KB
sample7	228 KB
sample8	98 KB
sample9	123 KB

Encoded File Sizes

Name	Size
sample1.huf	1 KB
sample2.huf	2 KB
sample3.huf	2 KB
sample4.huf	17 KB
sample5.huf	36 KB
sample6.huf	83 KB
sample7.huf	121 KB
sample8.huf	25 KB
sample9.huf	30 KB

Figure 16 *- File Comparison For Optimized Code*

4.2 - Comparing Cache Efficiency Metrics

The second metric that we looked at was comparing the cache efficiency of our Huffman coding solution. For this test the focus was on comparing the results of the 9 unique input samples. The intention was trying to see if there was a relationship between alphabet size or file length to the amount of cache misses. The samples were tested using cachegrind on a desktop computer so the results about the exact number of cache misses are not directly transferable to what they would have been on our ARM machine. However, the relative percentages and the ability to compare samples was still quite useful. For example, the results showed that for files containing smaller alphabets they had a lower percentage of data cache misses. This fact is supported by the reasoning that larger alphabets will produce symbols with longer code lengths which will result in the need for much bigger Look Up Tables and thus increase the chances for cache misses. Another interesting factor that we observed was that for samples with very small alphabets the percentage of data cache write misses was actually larger than that of data read misses.The complete cachegrind data can be seen in Appendix A.

4.3 - Profiling Time Spent in each Function

The final metric we looked at was the time spent in each function over the course of the program. This metric was obtained by compiling with the -pg flag and running gprof on our executable. This helped us compare several aspects of our program concretely such as, heap allocated LUT vs stack allocated LUT, LUT decoding vs bit-by-bit decoding, and how different sample distributions impacted our algorithms. Though it is known that heap memory is slower than stack memory, we were considering a dynamic approach for more flexibility if the trade-off wasn't too bad. However, our testing revealed that the heap implementation was 33% slower when compared to the stack implementation, so we decided it wasn't a worthwhile tradeoff. Furthermore, we were able to compare our decoding implementation in a relatively unambiguous manner which was very insightful for reasoning about our solutions. From these tests, we could observe that when we had a large alphabet with minimal redundancy the LUT approach was much slower than simply going bit-by-bit. We mostly attributed this to cache misses with the LUT, but other factors may also have been at play. When we tested sample8 and sample9 (randomly generated DNA data), we noticed that the LUT approach was getting much closer in performance to the bit-by-bit approach. Ultimately, the LUT approach and the bit-by-bit approach had the same performance time for sample9, but we suspect that with a larger sample the LUT approach would overtake the bit-by-bit approach. This is a downside of not scaling our code to handle arbitrary file lengths as files larger than sample9 would often not produce the correct output.

5.0 - Conclusion

This project implemented a Huffman Decoding and Encoding solution with two different approaches to Huffman Decoding: Huffman Tree Decoding and Look Up Table Decoding. The majority of work done on this project was optimizing the C code utilizing different optimization techniques including: implementing BARR-C standards, declaring register variables, using precomputed constants, implementing bit shift operations, attempting branch reduction techniques, designing for memory organization, and investigating assembly inlining. The final performance metrics compared how different alphabet characteristics of input data impact the performance of our solution.

Specifically, larger alphabets with longer codes result in a higher probability of data cache misses in the Look Up Table implementation.

As a final note, there are still more optimization techniques that could be applied to the Huffman Encoding and Decoding process that fall out of the scope of this project. If the project scope was increased and more time and resources were available a couple notable areas of future work could be: Firstly, implementing physical hardware improvements such as a microcontroller with a larger cache size and faster cache memory access speed. Another area for extra research would be into ways to keep the processor working while data is being decoded by giving it a parallel task, and scaling our program such that it can handle arbitrary file lengths.

6.0 - Bibliography

[1] R. Löffler, "How big is the internet?," Live Counter,
https://live-counter.com/how-big-is-the-internet/#:~:text=In%202016%20the%20amount

%20of%20data%20passing%20through,Internet%20grew%20to%20about%2026%2C500%20gigabytes%20per%20second.

[2] I. Pivkina, “Discovery of Huffman codes,” Discovery of Huffman Codes | Mathematical Association of America, https://www.maa.org/press/periodicals/convergence/discovery-of-huffman-codes.

[3] M. Sima “SENG 440 Embedded Systems - Lesson 105: Huffman Decoding”, 2023

[4] A. Barnwal, “Huffman coding: Greedy Algo-3,” GeeksforGeeks, https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

[5] H. Sidhwa, “Huffman decoding,” GeeksforGeeks, https://www.geeksforgeeks.org/huffman-decoding/?ref=lbp (accessed Aug. 4, 2023).

[6] “32-BIT CMOS MICROCONTROLLER USER’S MANUAL,” S3C2440A - keil, https://www.keil.com/dd/docs/datashts/samsung/s3c2440_um.pdf (accessed Aug. 4, 2023).

7.0 - Appendix

7.1 - Appendix A - LUT Decoding Cache Performance For All Samples

```
Note: Only 1st level cache metrics
Definitions:
Ir: Number of instruction reads (fetches) from the instruction cache for that line.
I1mr: Number of instruction cache misses for that line.
Dr: Number of data reads from the data cache for that line.
D1mr: Number of data cache misses for that line.
```

```
Dw: Number of data writes to the data cache for that line.
D1mw: Number of data cache misses due to writes for that line.

Cache Efficiency (CE) = (Cache Hits / (Cache Hits + Cache Misses))

Count   Rate
                            -------- sample1 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
79,847  (6.5%)          8 (0.5%)                 14,158 (14.1%)         184  (7.3%)

Dw________________      D1mw_______________      CE_______________
 6,689  (0.8%)           2  (0.0%)                0.991156753


                            -------- sample2 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
435,443 (19.2%)         8 (0.5%)                 77,390 (25.9%)         199  (7.4%)

Dw________________      D1mw_______________      CE_______________
37,179  (4.3%)          13  (0.1%)               0.998153004


                            -------- sample3 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
577,712 (21.4%)         8 (0.5%)                 102,886 (27.0%)        594 (17.8%)

Dw________________      D1mw_______________      CE_______________
49,586  (5.5%)          60  (0.4%)               0.998879517


                            -------- sample4 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
5,575,909 (36.6%)       8 (0.5%)                 994,522 (35.8%)        3,036 (40.4%)

Dw________________      D1mw_______________      CE_______________
481,137 (29.3%)         506  (3.1%)              0.997605464


                             -------- sample5 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
12,561,764 (39.6%)      8 (0.5%)                 2,244,494 (37.8%)      15,834 (70.2%)

Dw________________      D1mw_______________      CE_______________
1,088,156 (41.0%)       1,067  (5.3%)            0.994954249


                             -------- sample6 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
25,387,939 (31.0%)      7 (0.4%)                 4,551,422 (29.6%)      56,839 (83.5%)

Dw________________      D1mw_______________      CE_______________
2,210,607 (46.5%)       2,037  (7.6%)            0.991368300

                             -------- sample7 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
37,895,530 (37.8%)      7 (0.4%)                 6,751,020 (35.4%)      4,917 (26.1%)

Dw________________      D1mw_______________      CE_______________
3,259,030 (51.7%)       3,642 (16.9%)            0.998905360

                             -------- sample8 --------
Ir________________      I1mr_______________      Dr_______________      D1mr_______________
```

```
5,377,837 (30.5%)    7 (0.4%)              900,197 (26.9%)      514  (7.4%)

Dw________________   D1mw________________  CE________________
400,012 (23.7%)      1,565  (9.7%)         0.998403578


                              -------- sample9 --------
Ir________________   I1mr________________  Dr________________   D1mr________________
8,236,771 (35.9%)    7 (0.4%)              1,375,197 (31.4%)    603  (7.5%)

Dw________________   D1mw________________  CE________________
625,012 (30.8%)      1,956 (11.5%)         0.998722268
```

7.2 - Appendix B - Symbol Encodings For All Samples

```
-------- sample1 --------

Symbol: LF    Encoding: 11000010
Symbol: space Encoding: 101
Symbol: ,     Encoding: 111110
Symbol: .     Encoding: 111100
Symbol: :     Encoding: 011000011
Symbol: ?     Encoding: 11000011
Symbol: A     Encoding: 00110000
Symbol: C     Encoding: 011000100
Symbol: D     Encoding: 001100111
Symbol: H     Encoding: 011000101
```

```
Symbol: I     Encoding: 011000001
Symbol: L     Encoding: 001100110
Symbol: M     Encoding: 011000000
Symbol: P     Encoding: 00110110
Symbol: Q     Encoding: 11111100
Symbol: R     Encoding: 00110111
Symbol: S     Encoding: 11111101
Symbol: U     Encoding: 011000010
Symbol: a     Encoding: 1000
Symbol: b     Encoding: 1100000
Symbol: c     Encoding: 111101
Symbol: d     Encoding: 00111
Symbol: e     Encoding: 1110
Symbol: f     Encoding: 00110001
Symbol: g     Encoding: 1111111
Symbol: h     Encoding: 01100011
Symbol: i     Encoding: 010
Symbol: l     Encoding: 01101
Symbol: m     Encoding: 0111
Symbol: n     Encoding: 11001
Symbol: o     Encoding: 0010
Symbol: p     Encoding: 011001
Symbol: q     Encoding: 110001
Symbol: r     Encoding: 0000
Symbol: s     Encoding: 0001
Symbol: t     Encoding: 1101
Symbol: u     Encoding: 1001
Symbol: v     Encoding: 0011010
Symbol: x     Encoding: 00110010

-------- sample2 --------

Symbol: LF    Encoding: 10101010
Symbol: space Encoding: 110
Symbol: ,     Encoding: 00001
Symbol: -     Encoding: 0000011001
Symbol: .     Encoding: 011100
Symbol: :     Encoding: 00000101111
Symbol: ;     Encoding: 011101101
Symbol: ?     Encoding: 00000000
Symbol: A     Encoding: 011101111
Symbol: B     Encoding: 0000010001
Symbol: C     Encoding: 00000101010
Symbol: D     Encoding: 0000011100
Symbol: E     Encoding: 000001101
Symbol: F     Encoding: 0000010110
Symbol: H     Encoding: 00000110001
Symbol: I     Encoding: 011101100
Symbol: L     Encoding: 0000010000
Symbol: N     Encoding: 011101110
Symbol: P     Encoding: 00000001
Symbol: Q     Encoding: 10101011
Symbol: R     Encoding: 00000101110
```

```
Symbol: S     Encoding: 000001111
Symbol: T     Encoding: 000001001
Symbol: U     Encoding: 00000110000
Symbol: V     Encoding: 0000010100
Symbol: a     Encoding: 1000
Symbol: b     Encoding: 1010111
Symbol: c     Encoding: 01011
Symbol: d     Encoding: 01111
Symbol: e     Encoding: 1111
Symbol: f     Encoding: 1010110
Symbol: g     Encoding: 0111010
Symbol: h     Encoding: 0000001
Symbol: i     Encoding: 001
Symbol: l     Encoding: 01010
Symbol: m     Encoding: 11101
Symbol: n     Encoding: 0100
Symbol: o     Encoding: 0001
Symbol: p     Encoding: 101001
Symbol: q     Encoding: 101000
Symbol: r     Encoding: 11100
Symbol: s     Encoding: 0110
Symbol: t     Encoding: 1011
Symbol: u     Encoding: 1001
Symbol: v     Encoding: 1010100
Symbol: x     Encoding: 0000011101
Symbol: y     Encoding: 00000101011

-------- sample3 --------

Symbol: LF    Encoding: 11100110
Symbol: space Encoding: 110
Symbol: !     Encoding: 10100101110
Symbol: ,     Encoding: 111000
Symbol: -     Encoding: 10100101111
Symbol: .     Encoding: 1010011
Symbol: :     Encoding: 1110010010
Symbol: ;     Encoding: 111001010
Symbol: ?     Encoding: 111001011
Symbol: A     Encoding: 101001001
Symbol: B     Encoding: 10100101010
Symbol: C     Encoding: 1110010011
Symbol: D     Encoding: 0100011011
Symbol: E     Encoding: 1110010001
Symbol: F     Encoding: 10100101011
Symbol: H     Encoding: 10100101000
Symbol: I     Encoding: 1010010110
Symbol: L     Encoding: 0100011001
Symbol: M     Encoding: 0100011111
Symbol: N     Encoding: 1110010000
Symbol: P     Encoding: 101001010010
Symbol: Q     Encoding: 01000101
Symbol: R     Encoding: 0100011000
Symbol: S     Encoding: 101001000
```

```
Symbol: T     Encoding: 0100011110
Symbol: V     Encoding: 0100011010
Symbol: Z     Encoding: 101001010011
Symbol: a     Encoding: 1001
Symbol: b     Encoding: 1010111
Symbol: c     Encoding: 01100
Symbol: d     Encoding: 01101
Symbol: e     Encoding: 000
Symbol: f     Encoding: 01000100
Symbol: g     Encoding: 11100111
Symbol: h     Encoding: 1010110
Symbol: i     Encoding: 1111
Symbol: l     Encoding: 101010
Symbol: m     Encoding: 0010
Symbol: n     Encoding: 0101
Symbol: o     Encoding: 11101
Symbol: p     Encoding: 01001
Symbol: q     Encoding: 101000
Symbol: r     Encoding: 0011
Symbol: s     Encoding: 0111
Symbol: t     Encoding: 1000
Symbol: u     Encoding: 1011
Symbol: v     Encoding: 010000
Symbol: x     Encoding: 010001110

-------- sample4 --------

Symbol: LF    Encoding: 00000011
Symbol: CR    Encoding: 00000010
Symbol: space Encoding: 110
Symbol: ,     Encoding: 000110
Symbol: .     Encoding: 101011
Symbol: ;     Encoding: 0001110111101
Symbol: A     Encoding: 1010100010
Symbol: C     Encoding: 1010100011
Symbol: D     Encoding: 101010000
Symbol: E     Encoding: 0001110001
Symbol: F     Encoding: 0001110000
Symbol: I     Encoding: 000111100
Symbol: L     Encoding: 0001110111100
Symbol: M     Encoding: 0001110110
Symbol: N     Encoding: 00011111
Symbol: P     Encoding: 000111001
Symbol: Q     Encoding: 00011101110
Symbol: S     Encoding: 00000001
Symbol: U     Encoding: 000111011111
Symbol: V     Encoding: 000111101
Symbol: a     Encoding: 0111
Symbol: b     Encoding: 000011
Symbol: c     Encoding: 10110
Symbol: d     Encoding: 00010
Symbol: e     Encoding: 001
Symbol: f     Encoding: 1010101
```

```
Symbol: g     Encoding: 000001
Symbol: h     Encoding: 10101001
Symbol: i     Encoding: 1111
Symbol: j     Encoding: 000111010
Symbol: l     Encoding: 0100
Symbol: m     Encoding: 10100
Symbol: n     Encoding: 0101
Symbol: o     Encoding: 01101
Symbol: p     Encoding: 011001
Symbol: q     Encoding: 000010
Symbol: r     Encoding: 10111
Symbol: s     Encoding: 1001
Symbol: t     Encoding: 1000
Symbol: u     Encoding: 1110
Symbol: v     Encoding: 011000
Symbol: x     Encoding: 00000000

-------- sample5 --------

Symbol: LF    Encoding: 00010010
Symbol: CR    Encoding: 111010111
Symbol: space Encoding: 110
Symbol: ,     Encoding: 11101010110101
Symbol: .     Encoding: 111011
Symbol: A     Encoding: 111010100
Symbol: B     Encoding: 011110001011
Symbol: C     Encoding: 11101010111
Symbol: D     Encoding: 0111101001
Symbol: E     Encoding: 011110101
Symbol: F     Encoding: 0111100110
Symbol: G     Encoding: 1110101011011
Symbol: H     Encoding: 011110001010
Symbol: I     Encoding: 1110101101
Symbol: J     Encoding: 11101010110100
Symbol: L     Encoding: 0111101000
Symbol: M     Encoding: 011110000
Symbol: N     Encoding: 000100111
Symbol: O     Encoding: 01111000100
Symbol: P     Encoding: 1110101100
Symbol: Q     Encoding: 01111001111
Symbol: R     Encoding: 111010101100
Symbol: S     Encoding: 011110010
Symbol: T     Encoding: 1110101010
Symbol: U     Encoding: 0111100011
Symbol: V     Encoding: 000100110
Symbol: a     Encoding: 1000
Symbol: b     Encoding: 1110100
Symbol: c     Encoding: 01110
Symbol: d     Encoding: 00011
Symbol: e     Encoding: 001
Symbol: f     Encoding: 0001000
Symbol: g     Encoding: 011000
Symbol: h     Encoding: 01111011
```

```
Symbol: i     Encoding: 1111
Symbol: j     Encoding: 01111001110
Symbol: l     Encoding: 0000
Symbol: m     Encoding: 11100
Symbol: n     Encoding: 0101
Symbol: o     Encoding: 01101
Symbol: p     Encoding: 011111
Symbol: q     Encoding: 000101
Symbol: r     Encoding: 0100
Symbol: s     Encoding: 1001
Symbol: t     Encoding: 1011
Symbol: u     Encoding: 1010
Symbol: v     Encoding: 011001

-------- sample6 --------

Symbol: LF    Encoding: 0010
Symbol: space Encoding: 101
Symbol: !     Encoding: 0110101110
Symbol: (     Encoding: 000010110000
Symbol: )     Encoding: 000010110001
Symbol: ,     Encoding: 001110
Symbol: -     Encoding: 01101011110
Symbol: .     Encoding: 001111
Symbol: /     Encoding: 011010110111100
Symbol: 0     Encoding: 011011
Symbol: 1     Encoding: 011110
Symbol: 2     Encoding: 1110101
Symbol: 3     Encoding: 0110011
Symbol: 4     Encoding: 0101001
Symbol: 5     Encoding: 0111111
Symbol: 6     Encoding: 0000111
Symbol: 7     Encoding: 0011010
Symbol: 8     Encoding: 0000001
Symbol: 9     Encoding: 0000100
Symbol: :     Encoding: 0000101111100
Symbol: ;     Encoding: 0000101101
Symbol: ?     Encoding: 011001001
Symbol: A     Encoding: 0111110
Symbol: B     Encoding: 01010001
Symbol: C     Encoding: 0000110
Symbol: D     Encoding: 01010000
Symbol: E     Encoding: 10001000
Symbol: F     Encoding: 111011
Symbol: G     Encoding: 0110101100
Symbol: H     Encoding: 01101010
Symbol: I     Encoding: 11011100
Symbol: K     Encoding: 00001011001
Symbol: L     Encoding: 111101
Symbol: M     Encoding: 0011011
Symbol: N     Encoding: 111100
Symbol: O     Encoding: 00001010
Symbol: P     Encoding: 01101011010
```

```
Symbol: Q     Encoding: 011010111110
Symbol: R     Encoding: 110110010
Symbol: S     Encoding: 11011000
Symbol: T     Encoding: 01011
Symbol: U     Encoding: 011001000
Symbol: V     Encoding: 01101011011111
Symbol: W     Encoding: 01100101
Symbol: X     Encoding: 000010111101
Symbol: Y     Encoding: 1101100110
Symbol: a     Encoding: 11100
Symbol: b     Encoding: 1000101
Symbol: c     Encoding: 010101
Symbol: d     Encoding: 00010
Symbol: e     Encoding: 1001
Symbol: f     Encoding: 1110100
Symbol: g     Encoding: 1101101
Symbol: h     Encoding: 11010
Symbol: i     Encoding: 01110
Symbol: j     Encoding: 011010111111
Symbol: k     Encoding: 11011110
Symbol: l     Encoding: 00011
Symbol: m     Encoding: 011000
Symbol: n     Encoding: 11001
Symbol: o     Encoding: 11111
Symbol: p     Encoding: 0110100
Symbol: q     Encoding: 11011001110
Symbol: r     Encoding: 10000
Symbol: s     Encoding: 11000
Symbol: t     Encoding: 0100
Symbol: u     Encoding: 100011
Symbol: v     Encoding: 11011101
Symbol: w     Encoding: 000001
Symbol: x     Encoding: 0000101110
Symbol: y     Encoding: 001100
Symbol: z     Encoding: 0110101101110

-------- sample7 --------

Symbol: LF    Encoding: 011111111
Symbol: CR    Encoding: 011111110
Symbol: space Encoding: 110
Symbol: .     Encoding: 111011
Symbol: A     Encoding: 011110110
Symbol: B     Encoding: 011110000011
Symbol: C     Encoding: 0111101001
Symbol: D     Encoding: 0111101000
Symbol: E     Encoding: 011110101
Symbol: F     Encoding: 0100000001
Symbol: G     Encoding: 010000000001
Symbol: H     Encoding: 011110000010
Symbol: I     Encoding: 0111101111
Symbol: J     Encoding: 010000000000
Symbol: L     Encoding: 0111100111
```

```
Symbol: M     Encoding: 010000001
Symbol: N     Encoding: 011110001
Symbol: O     Encoding: 01111000000
Symbol: P     Encoding: 010000011
Symbol: Q     Encoding: 01111000011
Symbol: R     Encoding: 01000000001
Symbol: S     Encoding: 011110010
Symbol: T     Encoding: 0111101110
Symbol: U     Encoding: 0111100110
Symbol: V     Encoding: 010000010
Symbol: a     Encoding: 1000
Symbol: b     Encoding: 0111110
Symbol: c     Encoding: 01110
Symbol: d     Encoding: 01001
Symbol: e     Encoding: 001
Symbol: f     Encoding: 0100001
Symbol: g     Encoding: 011000
Symbol: h     Encoding: 01111110
Symbol: i     Encoding: 1111
Symbol: j     Encoding: 01111000010
Symbol: l     Encoding: 0000
Symbol: m     Encoding: 11100
Symbol: n     Encoding: 0101
Symbol: o     Encoding: 01101
Symbol: p     Encoding: 111010
Symbol: q     Encoding: 010001
Symbol: r     Encoding: 0001
Symbol: s     Encoding: 1010
Symbol: t     Encoding: 1001
Symbol: u     Encoding: 1011
Symbol: v     Encoding: 011001

-------- sample8 --------
Sample Distribution ~= (A: 25000, B: 25000, C:25000, D:25000)

Symbol: A     Encoding: 01
Symbol: C     Encoding: 00
Symbol: G     Encoding: 11
Symbol: T     Encoding: 10

-------- sample9 --------
Sample Distribution ~= (A: 50000, B: 30000, C:20000, D:25000)

Symbol: A     Encoding: 0
Symbol: C     Encoding: 10
Symbol: G     Encoding: 111
Symbol: T     Encoding: 110
```

7.3 - Appendix C - Profiling Times for Sample 9

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 33.33      0.02     0.02        1    20.00    20.00  lutDecoding
 33.33      0.04     0.02        1    20.00    20.00  
treeDecodingBitByBit
 16.67      0.05     0.01        1    10.00    10.00  
generateEncodedFileBitified
 16.67      0.06     0.01        1    10.00    10.00  prologue
  0.00      0.06     0.00        2     0.00     0.00  printCodesBitified
  0.00      0.06     0.00        1     0.00     0.00  
buildHuffmanEncTree
  0.00      0.06     0.00        1     0.00     0.00  freeHuffmanTree
  0.00      0.06     0.00        1     0.00     0.00  
```

```
generatehuffmanCodesBitified
  0.00      0.06     0.00        1     0.00     0.00  initMinHeap
  0.00      0.06     0.00        1     0.00     0.00  lutCreation
  0.00      0.06     0.00        1     0.00     0.00  lutPopulate
  0.00      0.06     0.00        1     0.00     0.00  maxTreeDepth

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
         else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
         function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
         the function in the gprof listing. If the index is
         in parenthesis it shows where it would appear in
         the gprof listing if it were to be printed.

                 Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 16.67% of 0.06 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.06                 main [1]
                0.02    0.00       1/1           treeDecodingBitByBit
[3]
                0.02    0.00       1/1           lutDecoding [2]
                0.01    0.00       1/1           prologue [5]
                0.01    0.00       1/1
generateEncodedFileBitified [4]
                0.00    0.00       1/1           buildHuffmanEncTree [7]
                0.00    0.00       1/1
generatehuffmanCodesBitified [9]
                0.00    0.00       1/1           maxTreeDepth [13]
                0.00    0.00       1/1           lutCreation [11]
                0.00    0.00       1/1           freeHuffmanTree [8]
```

```
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[2]     33.3    0.02    0.00       1         lutDecoding [2]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[3]     33.3    0.02    0.00       1         treeDecodingBitByBit [3]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[4]     16.7    0.01    0.00       1         generateEncodedFileBitified
[4]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[5]     16.7    0.01    0.00       1         prologue [5]
-----------------------------------------------
                0.00    0.00       2/2
generatehuffmanCodesBitified [9]
[6]      0.0    0.00    0.00       2         printCodesBitified [6]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[7]      0.0    0.00    0.00       1         buildHuffmanEncTree [7]
                0.00    0.00       1/1           initMinHeap [10]
-----------------------------------------------
                                   3             freeHuffmanTree [8]
                0.00    0.00       1/1           main [1]
[8]      0.0    0.00    0.00       1+3       freeHuffmanTree [8]
                                   3             freeHuffmanTree [8]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[9]      0.0    0.00    0.00       1
generatehuffmanCodesBitified [9]
                0.00    0.00       2/2           printCodesBitified [6]
-----------------------------------------------
                0.00    0.00       1/1           buildHuffmanEncTree [7]
[10]     0.0    0.00    0.00       1         initMinHeap [10]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[11]     0.0    0.00    0.00       1         lutCreation [11]
                0.00    0.00       1/1           lutPopulate [12]
-----------------------------------------------
                                   6             lutPopulate [12]
                0.00    0.00       1/1           lutCreation [11]
[12]     0.0    0.00    0.00       1+6       lutPopulate [12]
                                   6             lutPopulate [12]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[13]     0.0    0.00    0.00       1         maxTreeDepth [13]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
```

```
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:
    index  A unique number given to each element of the table.
           Index numbers are sorted numerically.
           The index number is printed next to every function name so
           it is easier to look up where the function in the table.

    % time This is the percentage of the `total' time that was spent
           in this function and its children.  Note that due to
           different viewpoints, functions excluded by options, etc,
           these numbers will NOT add up to 100%.

    self   This is the total amount of time spent in this function.

    children    This is the total amount of time propagated into this
           function by its children.

    called This is the number of times the function was called.
           If the function called itself recursively, the number
           only includes non-recursive calls, and is followed by
           a `+' and the number of recursive calls.

    name   The name of the current function.  The index number is
           printed after it.  If the function is a member of a
           cycle, the cycle number is printed between the
           function's name and the index number.


For the function's parents, the fields have the following meanings:

    self   This is the amount of time that was propagated directly
           from the function into this parent.

    children    This is the amount of time that was propagated from
           the function's children into this parent.

    called This is the number of times this parent called the
           function `/' the total number of times the function
           was called.  Recursive calls to the function are not
           included in the number after the `/'.

    name   This is the name of the parent.  The parent's index
           number is printed after it.  If the parent is a
           member of a cycle, the cycle number is printed between
           the name and the index number.

If the parents of the function cannot be determined, the word
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:
     self   This is the amount of time that was propagated directly
            from the child into the function.

     children    This is the amount of time that was propagated from the
            child's children to the function.

     called This is the number of times the function called
            this child `/' the total number of times the child
            was called.  Recursive calls by the child are not
            listed in the number after the `/'.

     name   This is the name of the child.  The child's index
            number is printed after it.  If the child is a
            member of a cycle, the cycle number is printed
            between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The `+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.


Index by function name

   [7] buildHuffmanEncTree    [10] initMinHeap            [13]
maxTreeDepth
   [8] freeHuffmanTree        [11] lutCreation             [6]
printCodesBitified
   [4] generateEncodedFileBitified [2] lutDecoding         [5] prologue
   [9] generatehuffmanCodesBitified [12] lutPopulate       [3]
treeDecodingBitByBit
```

7.4 - Appendix D - Profiling Times for Sample 7

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 44.12      0.15     0.15        1   150.00   150.00  lutDecoding
 26.47      0.24     0.09        1    90.00    90.00  prologue
 14.71      0.29     0.05        1    50.00    50.00
treeDecodingBitByBit
 11.76      0.33     0.04        1    40.00    40.00
generateEncodedFileBitified
```

```
  2.94      0.34     0.01        1    10.00    10.00  lutPopulate
  0.00      0.34     0.00        2     0.00     0.00  printCodesBitified
  0.00      0.34     0.00        1     0.00     0.00  
buildHuffmanEncTree
  0.00      0.34     0.00        1     0.00     0.00  freeHuffmanTree
  0.00      0.34     0.00        1     0.00     0.00  
generatehuffmanCodesBitified
  0.00      0.34     0.00        1     0.00     0.00  initMinHeap
  0.00      0.34     0.00        1     0.00    10.00  lutCreation
  0.00      0.34     0.00        1     0.00     0.00  maxTreeDepth

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
         else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
         function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
         the function in the gprof listing. If the index is
         in parenthesis it shows where it would appear in
         the gprof listing if it were to be printed.

                Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 2.94% of 0.34 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.34                 main [1]
                0.15    0.00       1/1           lutDecoding [2]
                0.09    0.00       1/1           prologue [3]
                0.05    0.00       1/1           treeDecodingBitByBit
[4]
                0.04    0.00       1/1           
generateEncodedFileBitified [5]
                0.00    0.01       1/1           lutCreation [6]
                0.00    0.00       1/1          buildHuffmanEncTree [9]
                0.00    0.00       1/1
generatehuffmanCodesBitified [11]
                0.00    0.00       1/1          maxTreeDepth [13]
                0.00    0.00       1/1          freeHuffmanTree [10]
-----------------------------------------------
                0.15    0.00       1/1          main [1]
[2]     44.1    0.15    0.00       1        lutDecoding [2]
-----------------------------------------------
                0.09    0.00       1/1          main [1]
[3]     26.5    0.09    0.00       1        prologue [3]
-----------------------------------------------
                0.05    0.00       1/1          main [1]
[4]     14.7    0.05    0.00       1        treeDecodingBitByBit [4]
-----------------------------------------------
                0.04    0.00       1/1          main [1]
[5]     11.8    0.04    0.00       1        generateEncodedFileBitified
[5]
-----------------------------------------------
                0.00    0.01       1/1          main [1]
[6]      2.9    0.00    0.01       1        lutCreation [6]
                0.01    0.00       1/1          lutPopulate [7]
-----------------------------------------------
                                  90            lutPopulate [7]
                0.01    0.00       1/1          lutCreation [6]
[7]      2.9    0.01    0.00       1+90     lutPopulate [7]
                                  90            lutPopulate [7]
-----------------------------------------------
                                  28            printCodesBitified [8]
                0.00    0.00       2/2
generatehuffmanCodesBitified [11]
[8]      0.0    0.00    0.00       2+28     printCodesBitified [8]
                                  28            printCodesBitified [8]
-----------------------------------------------
                0.00    0.00       1/1          main [1]
[9]      0.0    0.00    0.00       1        buildHuffmanEncTree [9]
                0.00    0.00       1/1          initMinHeap [12]
-----------------------------------------------
                                  42            freeHuffmanTree [10]
                0.00    0.00       1/1          main [1]
[10]     0.0    0.00    0.00       1+42     freeHuffmanTree [10]
                                  42            freeHuffmanTree [10]
-----------------------------------------------
                0.00    0.00       1/1          main [1]
[11]     0.0    0.00    0.00       1
generatehuffmanCodesBitified [11]
                0.00    0.00       2/2          printCodesBitified [8]
-----------------------------------------------
                0.00    0.00       1/1          buildHuffmanEncTree [9]
[12]     0.0    0.00    0.00       1        initMinHeap [12]
-----------------------------------------------
                                  46            maxTreeDepth [13]
                0.00    0.00       1/1          main [1]
```

```
[13]     0.0    0.00    0.00       1+46      maxTreeDepth [13]
                                  46           maxTreeDepth [13]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index  A unique number given to each element of the table.
            Index numbers are sorted numerically.
            The index number is printed next to every function name so
            it is easier to look up where the function in the table.

     % time This is the percentage of the `total' time that was spent
            in this function and its children.  Note that due to
            different viewpoints, functions excluded by options, etc,
            these numbers will NOT add up to 100%.

     self   This is the total amount of time spent in this function.

     children    This is the total amount of time propagated into this
            function by its children.

     called This is the number of times the function was called.
            If the function called itself recursively, the number
            only includes non-recursive calls, and is followed by
            a `+' and the number of recursive calls.

     name   The name of the current function.  The index number is
            printed after it.  If the function is a member of a
            cycle, the cycle number is printed between the
            function's name and the index number.


 For the function's parents, the fields have the following meanings:

     self   This is the amount of time that was propagated directly
            from the function into this parent.

     children    This is the amount of time that was propagated from
            the function's children into this parent.

     called This is the number of times this parent called the
            function `/' the total number of times the function
            was called.  Recursive calls to the function are not
            included in the number after the `/'.

     name   This is the name of the parent.  The parent's index
            number is printed after it.  If the parent is a
```

```
           member of a cycle, the cycle number is printed between
           the name and the index number.

 If the parents of the function cannot be determined, the word
 `<spontaneous>' is printed in the `name' field, and all the other
 fields are blank.

 For the function's children, the fields have the following meanings:

     self   This is the amount of time that was propagated directly
           from the child into the function.

     children    This is the amount of time that was propagated from the
           child's children to the function.

     called This is the number of times the function called
           this child `/' the total number of times the child
           was called.  Recursive calls by the child are not
           listed in the number after the `/'.

     name   This is the name of the child.  The child's index
           number is printed after it.  If the child is a
           member of a cycle, the cycle number is printed
           between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The `+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.


Index by function name

   [9] buildHuffmanEncTree    [12] initMinHeap            [13]
maxTreeDepth
  [10] freeHuffmanTree         [6] lutCreation             [8]
printCodesBitified
   [5] generateEncodedFileBitified [2] lutDecoding         [3] prologue
  [11] generatehuffmanCodesBitified [7] lutPopulate        [4]
treeDecodingBitByBit
```

7.5 - Appendix E - Optimized C Code Header File

```
#ifndef _HUFFMAN_HEADER_
```

```
#define _HUFFMAN_HEADER_
// Macros
#define MAX_FILE_LENGTH 512000        // Constraint for input file (256
kB)
#define MAX_COMPRESSED_LENGTH 128000  // Constraint for encoded file
(128 kB)
#define MAX_PATH_LENGTH 256           // Constraint for file path length
#define MAX_FILENAME_LENGTH 64        // Constraint for filename
#define MAX_SYMBOLS 128               // ASCII
#define MAX_CODE_LENGTH 16            // Should be a multiple of 8
#define MAX_TABLES 2048               // For Max symbols min table size
is over 50
#define MAX_ROWS_PER_TABLE 64         // Sets the maximum number of rows
per LUT

// Structs & Typedefs
typedef struct node_t {
    uint8_t symbol;
    float probability;
    struct node_t *leftChild;
    struct node_t *rightChild;
} node_t;                       // Each node_t contains a symbol, its
probability in the sample text, and pointers to its children in a tree

typedef struct {
    node_t **treeArray;
    uint32_t numNodes;
    uint32_t maxNodes;
} minHeap_t;                    // Only one minHeap_t should exist at a
time: contains a pointer to the root node, # of nodes in the tree, and
the max # of nodes

typedef struct {
    uint8_t symbol;
    uint16_t bitcode;
    uint32_t code_length;
} huffCode_t;                   // Each huffCode_t contains a symbol,
its Huffman Encoding, and the length of its encoding

typedef struct {
    uint8_t symbol;
    uint32_t code_length;
} lut;                          // Each lut contains a symbol paired
with its huffman encoding length

/***** Min Heap *****/
node_t* allocateNewNode(             // Dynamically allocates a min-heap
node containing a symbol-probability pair
    uint8_t symbol,
    float probability);
uint32_t isLeaf(                // Returns 1 if the given node is a leaf
node; returns -1 otherwise
    node_t* node);
```

```
void swapNodes(                     // Swaps two node pointers with
each other
    node_t** n1,
    node_t** n2);
void heapify(                       // Standard min-heapify
    minHeap_t* minHeap,
    uint32_t i);
node_t* popMin(                     // Pop the root, set the last node
as the new root, and heapify
    minHeap_t* minHeap);
minHeap_t* createMinHeap(           // Dynamically allocates memory for
min-heap and tree array
    uint32_t maxNodes);
void insertMinHeap(                 // Insert a new node into the
correct position of the min-heap
    minHeap_t* minHeap,
    node_t* newNode);
minHeap_t* initMinHeap(             // Create and initialize the
min-heap according to symbols and their probabilities
    uint8_t alph[],
    float prob[],
    uint32_t size);

/***** Huffman Encoding *****/
node_t* buildHuffmanEncTree(       // Construct the huffman tree from
the min-heap, by adding the appropriate internal nodes
    uint8_t alph[],
    float prob[],
    uint32_t size);
void printCodesBitified(           // Generate the huffman code for
every character in the tree, recusively, and store it in a table
    node_t* root,
    uint8_t* buff,
    uint32_t bit_count,
    huffCode_t* asciiToHuffman);
void generatehuffmanCodesBitified(  // Uses the huffman tree to populate
a table that maps each symbol to its huffman encoding in one memory
access (Don't have to traverse tree anymore in order to encode)
    node_t* root,
    huffCode_t* asciiToHuffman);
void generateEncodedFileBitified(   // Generates the huffman encoding of
the given sample and stores it in a file
    const char* sample_filename,
    const char* encoded_filename,
    huffCode_t* asciiToHuffman);

/***** LUT *****/
void lutPopulate(                    //
    node_t* root,
    uint8_t* buff,
    uint32_t bit_count,
    uint32_t longest_code_exp,
    lut all_luts[MAX_TABLES][MAX_ROWS_PER_TABLE],
    uint32_t* lut_num,
    uint32_t* row_count);
void lutCreation(                    //
    node_t* root,
    lut all_luts[MAX_TABLES][MAX_ROWS_PER_TABLE],
    uint32_t longest_code_exp);

/***** Huffman Decoding *****/
void treeDecodingBitByBit(                //
    char* huf_filename,
    char* decompressed_filename,
    node_t* root);
void lutDecoding(                         //
    const char* huf_filename,
    const char* decompressedLUT_filename,
    lut all_luts[MAX_TABLES][MAX_ROWS_PER_TABLE],
    const int barrel_shifter);

/***** Misc Helpers *****/
void printArr(
    int arr[],
    int n);
int symbolIndex(
    uint8_t* symbols,
    int symbol_count,
    uint8_t symbol);
int prologue(
    const char* original_filename,
    uint8_t* symbols,
    float* probabilities,
    uint32_t* sample_size);
void printSymbolEncodingBitified(
    huffCode_t* asciiToHuffman);
int maxTreeDepth(node_t* root);
void freeHuffmanTree(node_t* node);  // Free the memory allocated for
the huffman tree recursively

void custom_fwrite(             // Wrapper for fwrite to avoid problems
with fwrite's interal buffer, that is implementation dependent
    const void* array,
    uint32_t element_size,
    uint32_t total_elements,
    FILE* file);

#endif
```

7.6 - Appendix F - Optimized C Code Main File

```c
// Includes
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <assert.h>
#include <stdint.h>
#include "HuffmanHeader.h"


/***** Min Heap *****/
node_t* allocateNewNode(uint8_t symbol, float probability) {
    // Dynamically allocate memory for a new node
    node_t* temp = (node_t*)malloc(sizeof(node_t));
    assert(temp != NULL);
    // printf("Memory Allocated: %p\n", temp);
    temp->leftChild = NULL;
    temp->rightChild = NULL;
    temp->symbol = symbol;
    temp->probability = probability;
    return temp;
}

void swapNodes(node_t** n1, node_t** n2) {
    // Swap two nodes
    node_t* temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}

inline uint32_t isLeaf(node_t* node) {
    // Returns 1 if a node is a leaf; returns 0 otherwise
    return !(node->leftChild) && !(node->rightChild);
}

void heapify(minHeap_t* minHeap, uint32_t i) {
    // Standard min-heapify
    uint32_t minI = i;
    uint32_t leftI = 2*i+1;
    uint32_t rightI = 2*i+2;

    if (leftI < minHeap->numNodes &&
minHeap->treeArray[leftI]->probability <
minHeap->treeArray[minI]->probability) {
        minI = leftI;
    }

    if (rightI < minHeap->numNodes &&
minHeap->treeArray[rightI]->probability <
minHeap->treeArray[minI]->probability) {
        minI = rightI;
    }
    if (minI != i) {
        // Recurse
        swapNodes(&minHeap->treeArray[minI],&minHeap->treeArray[i]);
        heapify(minHeap, minI);
    }
}

node_t* popMin(minHeap_t* minHeap) {
    // Pop the min and heapify before returning
    node_t* temp = minHeap->treeArray[0];
    minHeap->treeArray[0] = minHeap->treeArray[minHeap->numNodes - 1];

    minHeap->numNodes--;
    heapify(minHeap, 0);

    return temp;
}

void insertMinHeap(minHeap_t* minHeap, node_t* newNode) {
    // Insert a new node into the heap
    minHeap->numNodes++;
    register uint32_t i = minHeap->numNodes - 1;

    while (i && newNode->probability < minHeap->treeArray[(i - 1) /
2]->probability) {

        minHeap->treeArray[i] = minHeap->treeArray[(i - 1) / 2];
        i = (i - 1) / 2;
    }

    minHeap->treeArray[i] = newNode;
}

minHeap_t* createMinHeap(uint32_t maxNodes) {
    // Create minHeap object and allocate necessary memory
    minHeap_t* minHeap = (minHeap_t*)malloc(sizeof(minHeap_t));
    assert(minHeap != NULL);

    minHeap->numNodes = 0;
    minHeap->maxNodes = maxNodes;

    minHeap->treeArray = (node_t**)malloc(minHeap->maxNodes *
sizeof(node_t*));
    assert(minHeap->treeArray != NULL);
    return minHeap;
}

minHeap_t* initMinHeap(uint8_t alph[], float prob[], uint32_t size) {
    // NOTE: If there are n symbols in our alphabet,
    // then there are n-1 internal nodes for Huffman Coding, for a total
of 2n - 1 nodes for a tree
    uint32_t numLeaves = size;
    minHeap_t* minHeap = createMinHeap(2*numLeaves - 1);
    register uint32_t i;
    for (i = 0; i < size; i++){
        minHeap->treeArray[i] = allocateNewNode(alph[i], prob[i]);
    }

    minHeap->numNodes = numLeaves;

    // Finish building min-heap by calling heapify on a subset of nodes
    // NOTE: j is signed to avoid undefined behavior when j == 0
    register int32_t j;
    for (j = (numLeaves - 2)/2; j >= 0; j--) {
        heapify(minHeap, j);
    }

    return minHeap;
}

/***** Huffman Encoding *****/
node_t* buildHuffmanEncTree(uint8_t alph[], float prob[], uint32_t size)
{
    node_t *left, *right, *top;

    // Create, initialize, and build the min heap
    minHeap_t* minHeap = initMinHeap(alph, prob, size);

    // Iterate while size of heap doesn't become 1
    while (!(minHeap->numNodes == 1)) {

        // Extract the two minimum probability nodes
        left = popMin(minHeap);
        right = popMin(minHeap);

        // Create new node: Internal nodes identified by '~', set
probability to the sum of its children, set child pointers
        top = allocateNewNode('~', left->probability +
right->probability);

        top->leftChild = left;
        top->rightChild = right;

        insertMinHeap(minHeap, top);
    }

    minHeap_t* lost_soul = minHeap;
    node_t* root = popMin(minHeap);

    free(lost_soul->treeArray);
    free(lost_soul);

    // Return the root of the Huffman tree
    return root;
}
```

```
void printCodesBitified(node_t* root, uint8_t* buff, uint32_t bit_count,
huffCode_t* asciiToHuffman) {
    // Assign 0 for left edge and recur
    if (root->leftChild) {
        buff[bit_count] = 0;
        printCodesBitified(root->leftChild,
        buff, bit_count + 1,
        asciiToHuffman);
    }

    // Assign 1 for right edge and recur
    if (root->rightChild) {
        buff[bit_count] = 1;
        printCodesBitified(root->rightChild,
        buff, bit_count + 1,
        asciiToHuffman);
    }

    // If this is a leaf node, then it contains a symbol
    if (isLeaf(root)) {
        uint8_t ascii_index = root->symbol;                 // Index the
table at the symbol's acsii code and,
        asciiToHuffman[ascii_index].code_length = bit_count;     //
store the length of its encoding in the table
          register uint32_t i;
        for (i = 0; i < bit_count; i++) {
            asciiToHuffman[ascii_index].bitcode +=  buff[i] * (1U << i);
// Represent the encoding with the right amount of bits
        }
    }
}

void generatehuffmanCodesBitified(node_t* root, huffCode_t*
asciiToHuffman) {
    // Initialize each symbol in the table such that its index matches
its ascii encoding (i.e. asciiToHuffman[65].symbol = 'A')
    uint8_t buff[128];
    register uint32_t i;
    for(i = 0; i < MAX_SYMBOLS; i++){
        asciiToHuffman[i].symbol = i;
        asciiToHuffman[i].bitcode = 0;
        asciiToHuffman[i].code_length = 0;
    }

    printCodesBitified(root, buff, 0, asciiToHuffman);
}

void generateEncodedFileBitified(const char* sample_filename, const
char* encoded_filename, huffCode_t* asciiToHuffman) {
    uint8_t sample[MAX_FILE_LENGTH];                        // Contains
orginal data
    uint16_t encodedSample[MAX_COMPRESSED_LENGTH] = {0};  // Initialized
at zero so only 1s have to be set
    uint8_t ascii_index;                                   // Contains
the ASCII code of the current symbol
    uint32_t cur_Index = 0;                                // # of shorts
in compressed data
    uint32_t sample_size;                                  // # of
symbols in original data
    uint32_t bit_position = 0;                             // Keeps track
of bit position
    uint32_t bit_count = 0;                                // # of bits
in compressed data
    FILE* fp_sample = fopen(sample_filename, "r");              // Points
to the original file
    FILE* fp_compressed = fopen(encoded_filename, "wb");        // Poitns
to the compressed file
    assert(fp_sample != NULL);
    assert(fp_compressed != NULL);

    sample_size = fread(sample, sizeof(uint8_t), MAX_FILE_LENGTH,
fp_sample);  // Retrieve original data
    register uint32_t i;
    for (i = 0; i < sample_size; i++) {  // Iterate 1 symbol per loop
iteration
        ascii_index = sample[i];         // Get ASCII code for current
symbol
        register uint32_t j;
          for (j = 0; j < asciiToHuffman[ascii_index].code_length; j++)
{  // Iterate through code length
                if (asciiToHuffman[ascii_index].bitcode & (1U << j)) {
// If a bit is set in the bitcode, then
                    encodedSample[cur_Index] |= (1U << bit_position);
// set the matching bit in the encoding
                }
                bit_count++;               // Increment bit count
                bit_position++;            // Increment bit position
                if(bit_position >= 16) {   // If bit_postion == 16
                    bit_position = 0;      // Reset bit_position
                    cur_Index++;           // Move to next short
                }
            }
    }

    fwrite(&bit_count, sizeof(uint32_t), 1,  fp_compressed);
// First 4 bytes of the compressed file contain the compressed size in
bits
    custom_fwrite(encodedSample, sizeof(uint16_t), cur_Index + 1,
fp_compressed);  // Rest of file contains encoding

    fclose(fp_sample);
    fclose(fp_compressed);
}

/***** LUT *****/
```

```
void lutPopulate(node_t* root,  uint8_t* buff, uint32_t bit_count,
uint32_t longest_code_exp, lut all_luts[MAX_TABLES][MAX_ROWS_PER_TABLE],
uint32_t* lut_num, uint32_t* row_count) {
    // Search for leaf to the left
    if (root->leftChild) {
        buff[bit_count] = 0;
        lutPopulate(root->leftChild, buff,  bit_count + 1,
longest_code_exp, all_luts, lut_num, row_count);
    }

    // Search for leaf to the right
    if (root->rightChild) {
        buff[bit_count] = 1;
        lutPopulate(root->rightChild, buff, bit_count + 1,
longest_code_exp, all_luts, lut_num, row_count);
    }

    // If this is a leaf node, then it contains a symbol
    if (isLeaf(root)) {
        // The buffer is the start of the index for the symbol and
code_length
        uint32_t bit_val = 0;
        register uint32_t i;
        for (i = 0; i < bit_count; i++) {
            bit_val <<= 1;
            if (buff[i]  == 1) {
                bit_val |= 1;
            }
            else if(buff[i] == 0) {
                bit_val |= 0;
            }
        }

        // Still need to fill in all possible combos of bits after the
buffer code
        uint32_t min_bit_val = bit_val;
        uint32_t max_bit_val = bit_val;
        uint32_t bit_dif = longest_code_exp - bit_count;
        register uint32_t j;
        for(j = 0; j < bit_dif; j++) {
            min_bit_val <<= 1;
            min_bit_val |= 0;
            max_bit_val <<= 1;
            max_bit_val |= 1;
        }

        while(min_bit_val <= max_bit_val) {
            all_luts[*lut_num][min_bit_val - ((*lut_num) *
(MAX_ROWS_PER_TABLE))].symbol = root->symbol;
            all_luts[*lut_num][min_bit_val - ((*lut_num) *
(MAX_ROWS_PER_TABLE))].code_length = bit_count;
            (*row_count) = (*row_count) + 1;
            if ((*row_count) == MAX_ROWS_PER_TABLE) {
                (*lut_num) = (*lut_num) + 1;
                (*row_count) = 0;
            }
            min_bit_val += 1;

        }
    }
}

void lutCreation(node_t* root, lut
all_luts[MAX_TABLES][MAX_ROWS_PER_TABLE], uint32_t longest_code_exp) {
    // Populate the tables with the symbols from the tree
    uint8_t buff[100];
    uint32_t lut_num = 0;
    uint32_t row_counter = 0;

    lutPopulate(root, buff , 0, longest_code_exp, all_luts, &lut_num,
&row_counter);
}

/***** Huffman Decoding *****/
void treeDecodingBitByBit(char* huf_filename, char*
decompressed_filename, node_t* root) {
    node_t* cur = root;                                         //
Cursor for traversing Huffman tree
    uint16_t compressed[MAX_COMPRESSED_LENGTH];            // Contains
compressed data
    uint8_t decompressed[MAX_FILE_LENGTH] = {0};           // Contains
decompressed data
    uint32_t compressed_bits = 0;                          // # of bits
in compressed data
    uint32_t cur_Index = 0;                                // Indexes
the current short
    uint32_t num_symbols = 0;                              // Counts #
of decompressed symbols
    uint32_t bit_position = 0;                             // Keeps
track of bit position
    FILE* fp_compressed= fopen(huf_filename, "rb");             //
Points to the compressed file
    FILE* fp_decompressed = fopen(decompressed_filename, "w");  //
Points to the decompressed file, or creates it if it doesn't exist
    assert(fp_compressed != NULL);
    assert(fp_decompressed != NULL);

    assert(fread(&compressed_bits, sizeof(uint32_t), 1, fp_compressed)
!= 0);                                 // Retrieve # of compressed
bits
    assert(fread(compressed, sizeof(uint16_t), MAX_COMPRESSED_LENGTH >>
1, fp_compressed) != 0);  // Retrieve compressed data
    register uint32_t i;
    for (i = 0; i < compressed_bits; i++) {  // Iterate through 1
compressed bit per loop iteration
```

```
        if (!isLeaf(cur)) {                                     // If
the current node is not a leaf,
            if (compressed[cur_Index] & (1U << bit_position)) {  // and
if the current compressed bit is a 1,
                cur = cur->rightChild;                          // then
traverse the right edge of the current node
            }
            else {                                              //
Else, we know the current compressed bit is a 0,
                cur = cur->leftChild;                           // so
traverse the left edge of the current node
            }

            bit_position++;            // Increment the bit position for
the current word
            if (bit_position >= 16) {  // If bit position exceeds
word-length,
                bit_position = 0;      // then reset bit position,
                cur_Index++;           // and move on to the next word
            }
        }
        else {                                     // Else, we know
the current node is a leaf and we have sucessfully decoded a symbol,
            decompressed[num_symbols] = cur->symbol;  // so store the
symbol in the decompressed array,
            num_symbols++;                            // increment the
decompressed symbol count,
            cur = root;                               // set the current
node back to the root of the huffman tree,
            i--;                                      // and decrement
the loop counter, since 1 dedicated loop iteration is required to decode
a symbol (Otherwise we skip 1 encoded bit after each symbol
decompression)
        }
    }

    // Decompress the last symbol, if it exists
    if (isLeaf(cur)) {
        decompressed[num_symbols] = cur->symbol;
        num_symbols++;
    }

    custom_fwrite(decompressed, sizeof(uint8_t), num_symbols,
fp_decompressed);
    fclose(fp_compressed);
    fclose(fp_decompressed);
}

void lutDecoding(const char* huf_filename, const char*
decompressedLUT_filename, lut all_luts[MAX_TABLES][MAX_ROWS_PER_TABLE],
const int barrel_shifter) {
    FILE* fp_compressed = fopen(huf_filename, "rb");
    FILE* fp_decompressed = fopen(decompressedLUT_filename, "wb");
    uint32_t compressed_bits = 0;       // # of bits in compressed data
    uint16_t compressed[MAX_COMPRESSED_LENGTH];
    uint8_t decompressed[MAX_FILE_LENGTH];

    assert(fp_compressed != NULL);
    assert(fp_decompressed != NULL);

    // Go through the compressed file by barrel_shifter length sections
(longest code length)
    // and search each table for that index

    assert(fread(&compressed_bits, sizeof(uint32_t), 1, fp_compressed)
!= 0);
    assert(fread(compressed, sizeof(uint16_t), MAX_COMPRESSED_LENGTH >>
1, fp_compressed) != 0);

    uint32_t decoded_bit_count = 0;
    uint32_t decoded_symbol_count = 0;
    uint32_t cur_index = 0;
    int32_t bit_pos = 0;
    while(decoded_bit_count < compressed_bits) {     // Iterate through
until all bits are decoded

    // Get barrel shifter size section of compressed message
        uint32_t compressed_section = 0;
        register uint32_t j;
        for (j = decoded_bit_count; j < barrel_shifter +
decoded_bit_count; j++) {

            if (compressed[cur_index] & (1U << bit_pos )) {
                compressed_section |= 1u << ((barrel_shifter - 1) - (j -
decoded_bit_count));
            }
            bit_pos++;
            if(bit_pos >= 16){
                bit_pos = 0;
                cur_index++;
            }
        }

        // Find the index that is the same value as the
compressed_section
        // use all_luts[i][x] and the MAX_ROWS_PER_TABLE variable
        uint32_t table_index = 0;
        uint32_t table_num = 0;

        // TODO: Posible optimization by inlining assembly division and
retrieving result and remainder registers from one division operation
        table_index = compressed_section % MAX_ROWS_PER_TABLE;
        table_num = compressed_section / MAX_ROWS_PER_TABLE;

        // Set to the code_length at that index and print the symbol
        uint32_t temp_bit_count =
all_luts[table_num][table_index].code_length;
        uint8_t symbol = all_luts[table_num][table_index].symbol;

        decompressed[decoded_symbol_count] = symbol;
        decoded_symbol_count++;

        bit_pos -= (barrel_shifter - temp_bit_count);      // update the
bit position to account for not decoded bits
        if (bit_pos < 0) {                                 // if the
bit_pos is negative we need to go back to previous index
            cur_index--;
            bit_pos = 16 + bit_pos;
        }

        decoded_bit_count = decoded_bit_count + temp_bit_count;
    }

    custom_fwrite(decompressed, sizeof(uint8_t), decoded_symbol_count,
fp_decompressed);
    fclose(fp_compressed);
    fclose(fp_decompressed);
}

/***** Misc Helpers *****/
void printArr(int arr[], int n) {
    register int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);

    printf("\n");
}

int symbolIndex(uint8_t* symbols, int symbol_count, uint8_t symbol) {
    // If a symbol exists in symbols, return its index
    // Otherwise return -1
    int i;
    for (i = 0; i < symbol_count; i++) {
            if (symbols[i] == symbol) {
                return i;
            }
        }

    return -1;
}

int prologue(const char* original_filename, uint8_t* symbols, float*
probabilities, uint32_t* sample_size) {
    FILE* fp = fopen(original_filename, "r");  // File pointer
    uint8_t cur_symbol;                // Holds the current symbol
    uint32_t symbol_count = 0;               // # of unique ASCII values
    uint32_t frequencies[MAX_SYMBOLS];       // Frequencies of each
symbol
    uint8_t sample[MAX_FILE_LENGTH];   // Buffer for sample text
    assert(fp != NULL);
    *sample_size = fread(sample, sizeof(uint8_t), MAX_FILE_LENGTH, fp);
// Retrieve sample text
    register uint32_t i;
    for (i = 0; i < *sample_size; i++) {
// Iterates through sample text 1 char at a time
        cur_symbol = sample[i];
        if (symbol_count > MAX_SYMBOLS) {
// If # of symbols exceeds MAX_SYMBOLS,
            printf("Symbol count exceeded!\n");
// then abort
            exit(1);
        }
        else if (symbolIndex(symbols, symbol_count, cur_symbol) == -1) {
// Else if a new symbol is encountered,
            symbols[symbol_count] = cur_symbol;
// then store it,  and increment
            frequencies[symbol_count] = 1;
// initialize its frequency,
            symbol_count++;
// and increment symbol_count
        }
        else {
// Else retrieve index for cur_symbol,
            frequencies[symbolIndex(symbols, symbol_count, cur_symbol)]
+= 1;  // and incrememt its frequency
        }
    }

    register double inv_sample_size = *sample_size;
    inv_sample_size = 1/inv_sample_size;  // Precomputed for efficiency
    register uint32_t j;
    for (j = 0; j < symbol_count; j++) {
        probabilities[j] = frequencies[j]*inv_sample_size;
    }

    fclose(fp);
    return symbol_count;
}

void printSymbolEncodingBitified(huffCode_t* asciiToHuffman) {
    uint8_t symbol;
    char *control_characters[] = {
        "NUL","SOH", "STX", "ETX", "EOT", "ENQ", "ACK", "BEL",
        "BS", "HT", "LF", "VT", "FF", "CR", "SO", "SI", "DLE",
        "DC1", "DC2","DC3", "DC4", "NAK", "SYN", "ETB", "CAN",
        "EM", "SUB", "ESC", "FS", "GS", "RS", "US"};

    register uint32_t i;
    for (i = 0; i < MAX_SYMBOLS; i++) {
        if (asciiToHuffman[i].code_length > 0) {
            // Print symbol appropriately
            symbol = asciiToHuffman[i].symbol;
            if (symbol < 8 || (symbol > 15 && symbol < 25) || symbol ==
26 || symbol == 27) {
                printf("Symbol: %s   Encoding: ",
control_characters[(uint8_t) symbol]);
            }
            else if ((symbol > 7 && symbol < 16) || symbol == 25
||(symbol > 27 && symbol < 32)) {
                printf("Symbol: %s    Encoding: ",
control_characters[(uint8_t) symbol]);
            }
            else if (symbol == 32) {
                printf("Symbol: space Encoding: ");
            }
            else if (symbol == 127) {
                printf("Symbol: DEL   Encoding: ");
            }
            else {
                printf("Symbol: %c     Encoding: ", symbol);
            }

            uint8_t code[MAX_CODE_LENGTH] = {0};
            register uint32_t j;
            for (j = 0; j < asciiToHuffman[i].code_length; j++) {
                if (asciiToHuffman[i].bitcode & (1U << j)) {
                    code[j] = '1';
                }
                else {
                    code[j] = '0';
                }

            }
            printf("%s\n", code);
        }
    }
}

int maxTreeDepth(node_t* root) {
    if (root == NULL) {
        return 0;
    }
    else {
        int leftDepth = maxTreeDepth(root->leftChild);
        int rightDepth = maxTreeDepth(root->rightChild);
        int max = (leftDepth > rightDepth) ? leftDepth : rightDepth;
        return max + 1;
    }
}

void freeHuffmanTree(node_t* node) {
    if (node == NULL) {
        return;
    }
    freeHuffmanTree(node->leftChild);
    freeHuffmanTree(node->rightChild);

    free(node);
}

void custom_fwrite(const void* array, uint32_t element_size, uint32_t
total_elements, FILE* file) {
    uint32_t elements_written = 0;
    uint32_t chunk_size = 32000;

    while (elements_written < total_elements) {
        uint32_t remaining_elements = total_elements - elements_written;
        uint32_t elements_to_write = (remaining_elements < chunk_size) ?
remaining_elements : chunk_size;
        uint32_t bytes_written = fwrite(array + (elements_written *
element_size), element_size, elements_to_write, file);
        assert(bytes_written == elements_to_write);

        fflush(file);
        elements_written += elements_to_write;
    }
}


int main(int argc, char* argv[]) {
    /******************************** Variables
********************************/
    char input_filename[MAX_FILENAME_LENGTH];        // Filename
provided by user input
    char original_filename[MAX_PATH_LENGTH];         // Filename for
original file
    char compressed_filename[MAX_PATH_LENGTH];       // Filename for
compressed file
    char decompressed_filename[MAX_PATH_LENGTH];     // Filename for
decompressed file (Bitified)
    char decompressedLUT_filename[MAX_PATH_LENGTH];  // Filename for
decompressed file (LUT)
    uint8_t symbols[MAX_SYMBOLS];               // Every unique ASCII
symbol in the sample (unsorted)
    float probabilities[MAX_SYMBOLS];                // Corresponding
probability for each symbol
    uint32_t sample_size;                       // Size of the sample in
bits
    uint32_t symbol_count;                      // Size of the alphabet
    node_t* huffmanRoot;                             // Root node of the
huffman tree
    huffCode_t asciiToHuffman[MAX_SYMBOLS];          // Huffman code of
each symbol is stored at the index of its ascii encoding
    lut all_luts[MAX_TABLES][MAX_ROWS_PER_TABLE];
// An array to store all the look up tables
    //uint32_t tables_needed = 0;                 // Used to keep track
of the LUTs needed for an alphabet
```

```
    /******************************* Handle User Input
*******************************/
    if (argc == 1) {
        fprintf(stderr, "Too few arguments!\nusage: %s <filename>\n",
argv[0]);
        exit(1);
    }
    else if (argc == 2 && strlen(argv[1]) < 64) {
        strncpy(input_filename, argv[1], 63);
    }
    else if (strlen(argv[1]) >= 64) {
        fprintf(stderr, "Filename is too long!\nusage: %s <filename>\n",
argv[0]);
        exit(1);
    }
    else {
        fprintf(stderr, "Too many arguments!\nusage: %s <filename>\n",
argv[0]);
        exit(1);
    }

    /******************************* Initialization
*******************************/
    if (input_filename != NULL) {
        // Assign directory paths
    strncpy(original_filename, "original/", MAX_PATH_LENGTH - 1);
    strncpy(compressed_filename, "compressed/", MAX_PATH_LENGTH - 1);
    strncpy(decompressed_filename, "decompressed/", MAX_PATH_LENGTH -
1);
    strncpy(decompressedLUT_filename, "decompressed/", MAX_PATH_LENGTH -
1);

    // Null-terminate the destination buffers
    original_filename[MAX_PATH_LENGTH - 1] = '\0';
    compressed_filename[MAX_PATH_LENGTH - 1] = '\0';
    decompressed_filename[MAX_PATH_LENGTH - 1] = '\0';
    decompressedLUT_filename[MAX_PATH_LENGTH - 1] = '\0';

    // Retrieve filename without file extension
    char stripped_filename[MAX_FILENAME_LENGTH+1];
    strncpy(stripped_filename, input_filename, MAX_FILENAME_LENGTH - 1);
    stripped_filename[MAX_FILENAME_LENGTH - 1] = '\0'; // Ensure
null-termination

    // Find the position of the last dot in the filename
    char* dot_pos = strrchr(stripped_filename, '.');

    // If there is a dot, remove the extension
    if (dot_pos != NULL) {
        *dot_pos = '\0';
    }
    // Append filename to file paths
    strncat(original_filename, stripped_filename, MAX_PATH_LENGTH -
strlen(original_filename) - 1);
    strncat(compressed_filename, stripped_filename, MAX_PATH_LENGTH -
strlen(compressed_filename) - 1);
    strncat(decompressed_filename, stripped_filename, MAX_PATH_LENGTH -
strlen(decompressed_filename) - 1);
    strncat(decompressedLUT_filename, stripped_filename, MAX_PATH_LENGTH
- strlen(decompressedLUT_filename) - 1);

    // Append file extension
    strncat(original_filename, ".txt", MAX_PATH_LENGTH -
strlen(original_filename) - 1);
    strncat(compressed_filename, ".huf", MAX_PATH_LENGTH -
strlen(compressed_filename) - 1);
    strncat(decompressed_filename, ".txt", MAX_PATH_LENGTH -
strlen(decompressed_filename) - 1);
    strncat(decompressedLUT_filename, "LUT.txt", MAX_PATH_LENGTH -
strlen(decompressed_filename) - 1);
    }
    else {
        fprintf(stderr, "Error retrieving filename!\nusage: %s
<filename>\n", argv[0]);
        exit(1);
    }


    symbol_count = prologue(original_filename, symbols,  // Retrieves
sample text and size, determines symbols
                   probabilities, &sample_size);         // and their
probabilities, and return symbol_count


    /******************************* Encoding
*******************************/
    huffmanRoot = buildHuffmanEncTree(symbols,
// Construct a Huffman encoding tree
                                      probabilities,
// for the given symbol and probability pairs
                                      symbol_count);

    generatehuffmanCodesBitified(huffmanRoot,
// Recursively traverse the tree, generate the huffman code for every
leaf's symbol,
                                 asciiToHuffman);
// and populate the encoding table

    //printSymbolEncodingBitified(asciiToHuffman);
// Print the huffman encoding of every symbol in the alphabet


    generateEncodedFileBitified(original_filename,     // Perform
huffman encoding on the sample text and,
                                compressed_filename,   // store the
result in a .huf file
                                asciiToHuffman);

    /******************************** Decoding
******************************/
    treeDecodingBitByBit(compressed_filename, decompressed_filename,
huffmanRoot);  // Decode the compressed file in a bit-by-bit manner

    uint32_t code_max_bits = 0;
    code_max_bits = maxTreeDepth(huffmanRoot);

    lutCreation(huffmanRoot, all_luts, code_max_bits);

    lutDecoding(compressed_filename, decompressedLUT_filename, all_luts,
code_max_bits);

    freeHuffmanTree(huffmanRoot);

    return 0;
}
```