ECE36800 Programming Assignment #2

Due Monday, February 21, 2022, 11:59pm

This assignment covers learning objective 1: An understanding of basic data structures, including stacks, queues, and trees; learning objective 3: An ability to apply appropriate sorting and search algorithms for a given application; learning objective 5: An ability to design and implement appropriate data structures and algorithms for engineering applications.

This assignment is to be completed on your own. The description is mainly taken from Professor Vijay Raghunathan. It deals with file compression and file decompression (similar to zip and unzip). They are based on the widely used algorithmic technique of Huffman coding, which is used in JPEG compression as well as in MP3 audio compression. In particular, you will utilize your knowledge about lists/arrays, trees and/or other necessary data structures learned in ECE26400 to **design program** to "decompress" (decode) a file that has been compressed (coded or encoded) using "Huffman coding."

You may have learned about Huffman coding earlier in ECE26400. **Your instructor may have provided you structures/functions that are relevant for this assignment. However, you are not allowed to use those structures/functions as your own work in this assignment. You should write your own structures/functions to replace those structures/functions for this assignment.**

For this assignment, you will decode a file that has been coded using a coding tree, similar to how a file is coded (or compressed) using a Huffman coding tree. Your program will have to decide whether the coding tree is a Huffman coding tree, i.e., whether the coding has been constructed optimally. We shall first present the concept of Huffman coding, which is essential for your development of a method to determine whether a given file has been coded optimally.

Although the description is based on Huffman coding tree, it can be applied to any coding tree that is a strictly binary tree (with ASCII characters as the leaf nodes). A strictly binary tree is a binary tree where a node has either 0 or 2 child nodes. A node with 0 child nodes is a leaf node and a node with 2 child nodes is an internal (non-leaf) node.

ASCII coding (and extended ASCII coding)

In (extended) ASCII coding, every character is encoded (represented) with the same number of bits (8-bits) per character. Since there are 256 different values that can be represented with 8-bits, there are potentially 256 different characters in the ASCII character set, as shown in the ASCII character table (and extended ASCII character table) available at http://www.asciitable.com/.

Using ASCII encoding (8 bits per character) the 13-character string `"go go gophers"` requires $13 \times 8 = 104$ bits. The table to the right shows how the coding works. **From left to right, the binary bits for each character are ordered from the most significant position to the least significant position.**

Character	ASCII value	8-bit binary value
`Space ' '`	32	00100000
`'e'`	101	01100101
`'g'`	103	01100111
`'h'`	104	01101000
`'o'`	111	01101111
`'p'`	112	01110000
`'r'`	114	01110010
`'s'`	115	01110011

The given string would be written in a file as 13 bytes, represented by the following stream of bits:

```
01100111 01101111 00100000 01100111 01101111 00100000 01100111 01101111 01110000
01101000 01100101 01110010 01110011
```

A more efficient coding

There is a more efficient coding scheme that uses fewer bits. As there are only 8 different characters in `"go go gophers"`, we can use 3 bits to encode each of the 8 different characters. We may, for example, use the coding shown in the following table (other 3-bit encodings are also possible). Again, we assume that **from left to right, the 3 bits for each character is ordered from the most significant position to the least significant position.**

Note that we assume that for each byte in this bit stream, we have the **most significant bit on the left and the least significant bit on the right**. In other words, the least significant bit in this bit stream is a 1-bit. Now the string `"go go gophers"` would be encoded using a total of 39 bits instead of 104 bits. We can store that as five 8-bit bytes in a file as follows (**in each byte, left to right is most significant to least significant**):

Character	Code value	3-bit binary value
`'g'`	0	000
`'o'`	1	001
`'p'`	2	010
`'h'`	3	011
`'e'`	4	100
`'r'`	5	101
`'s'`	6	110
`Space ' '`	7	111

```
11001000 10010001 00100011 00011010 01101011.
```

The first byte contains the 3 bits of `'g'` at the least significant positions, the 3 bits of `'o'` in the middle, and the less significant 2 bits of `Space`. The least significant bit of the second byte contains the most significant bit of `Space`. In positions of increasing significance, the second byte also contains `'g'`, `'o'`, and the least significant bit of `Space`. As five bytes contains 40 bits altogether, the most significant position of the last byte in the file is not used. **In this assignment, all unused bits should be assigned the value of 0.**

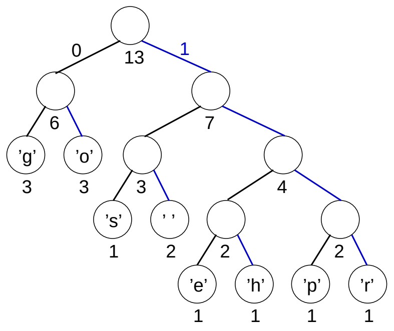
However, even in this improved coding scheme, we used the same number of bits to represent each character, regardless of how often the character appears in our string. Even more bits can be saved if we use fewer than three bits to encode characters like `'g'`, `'o'`, and `Space` that occur frequently and more than three bits to encode characters like `'e'`, `'h'`, `'p'`, `'r'`, and `'s'` that occur less frequently in `"go go gophers"`. This is the basic idea behind Huffman coding: use fewer bits for characters that occur more frequently. We'll see how this is done using a strictly binary tree that stores the characters as its leaf nodes, and whose root-to-leaf paths provide the bit sequences used to encode the characters.

Again, although the description is based on Huffman coding tree, it can be applied to any coding tree that is a strictly binary tree with the leaf nodes storing ASCII characters. The main difference is in the the number of bits required to represent (code/encode/compress) a given file.

A coding tree

Using a strictly binary tree for coding, all characters are stored at the leaf nodes of a tree. A left-edge is numbered 0 and a right-edge is numbered 1. The code for any character/leaf node is obtained by following the root-to-leaf path and concatenating the 0's and 1's. The specific structure of the tree determines the coding of any leaf node using the 0/1 edge convention described. As an example, the tree to the right yields the coding shown in the table. **Unlike before, from left to right, the binary bits for each character in the table are ordered from the least significant position to the most significant position.**

Character	Huffman code
`'g'`	00
`'o'`	01
`'s'`	100
`Space ' '`	101
`'e'`	1100
`'h'`	1101
`'p'`	1110
`'r'`	1111



Using this coding, `"go go gophers"` can be encoded with 37 bits, two bits fewer than the 3-bit coding scheme. **From the least significant position to the most significant position, the bit stream from left to right** is as follows:

```
00 01 101 00 01 101 00 01 1110 1101 1100 1111 100.
```

Of course, we still have to use 5 bytes to store the 37 bits in a file. To show the bytes stored in the file, we insert | to delimit every 8 bits as follows:

```
00 01 101 0|0 01 101 00| 01 1110 11|01 1100 11|11 100.
```

Then, we write in the **convention of the least significant bit appearing on the right for each byte** as follows:

```
01011000 00101100 11011110 11001110 00000111.
```

In this case, **the three most significant bits of the last byte in the file are unused and they are assigned the value of 0.**

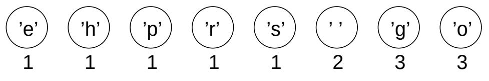
To decode a non-empty stream (from the least significant position to the most) that has been coded by the given tree, start at the root of the tree, and follow a left-branch if the next bit in the stream is a 0, and a right branch if the next bit in the stream is a 1. When you reach a leaf, write the character stored at the leaf, and start again at the top of the tree.

Huffman coding tree

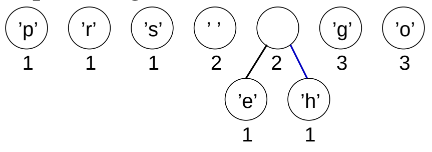
The coding tree in this example was constructed using an algorithm invented by David A. Huffman in 1952 when he was a Ph.D. student at MIT. We shall discuss how to construct the coding tree using Huffman's algorithm. We assume that associated with each character is a weight that is equal to the number of times the character occurs in a file. For example, in the string `"go go gophers"`, the characters `'g'` and `'o'` have weight 3, the `Space` has weight 2, and the other characters have weight 1.

When compressing a file, we first read the file and calculate these weights. Assume that all the character weights have been calculated. Huffman's algorithm assumes that we are building a single tree from a group (or forest) of trees. Initially, all the trees have a single node containing a character and the character's weight. Iteratively, a new tree is formed by picking two trees and making a new tree whose child nodes are the roots of the two trees. The weight of the new tree is the sum of the weights of the two sub-trees. This decreases the number of trees by one in each iteration. The process iterates until there is only one tree left. The algorithm is as follows:

1. Begin with a forest of trees. All trees have just one node, with the weight of the tree equal to the weight of the character in the node. Characters that occur most frequently have the highest weights. Characters that occur least frequently have the smallest weights.
2. Repeat this step until there is only one tree:
   - Choose two trees with the smallest weights; call these trees T_1 and T_2.
   - Create a new tree whose root has a weight equal to the sum of the weights of T_1 and T_2, and whose left sub-tree is T_1 and whose right sub-tree is T_2.
3. The single tree left after the previous step is a Huffman coding tree.

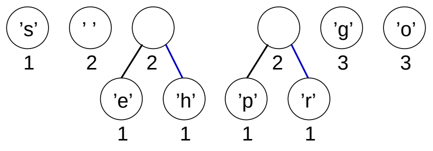


For the string `"go go gophers"`, we initially have the forest shown to the right. The nodes are shown with a weight that represents the number of times the node's character occurs in the given input string/file.

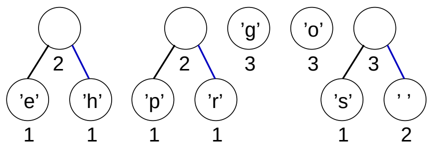


We pick two minimal nodes. There are five nodes with the minimal weight of 1. In this implementation, we maintain a priority queue with items arranged according to their weights, i.e., the items are sorted. When two items have the same weight, a leaf node (i.e., a node associated with an ASCII character) is always ordered first. If both nodes are leaf nodes, they are ordered according to their ASCII coding. If both nodes are non-leaf nodes, they are ordered according to the creation times of the nodes. We always pick the first two items in the priority queue, namely, nodes for characters `'e'` and `'h'`. We create a new tree whose root is weighted by the sum of the weights chosen. The order of the nodes in the priority queue also determines the left and right child nodes of the new root. We now have a forest of seven trees as shown here. Although the newly created node has the same weight as `Space`, it is ordered after Space in the priority queue because `Space` is an ASCII character.

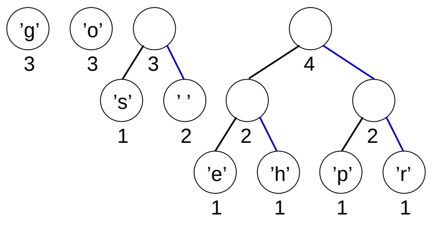
Choosing the first two (minimal) nodes in the priority queue yields another tree with weight 2 as shown here. There are now six trees in the forest of trees that will eventually build an encoding tree.



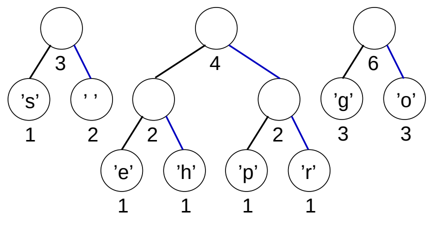
Again we must choose the first two (minimal) nodes in the priority queue. The lowest weight is the 'e'-node/tree with weight equal to 1. There are three trees with weight 2; the one chosen corresponds to an ASCII character because of the way we order the nodes in the priority queue. The new tree has a weight of 3, which will be placed last in the priority queue according to our ordering strategy.

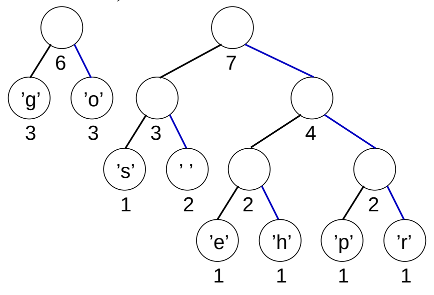


Now there are two trees with weight equal to 2. These are joined into a new tree whose weight is 4. There are four trees left, one whose weight is 4 and three with a weight of 3.



The first two minimal (weight 3) trees in the priority queue are joined into a tree whose weight is 6 (see the left figure below). There are three trees left. Now, the minimal trees have weights of 3 and 4; these are joined into a tree with weight 7 (see the right figure below).





Finally, the last two trees are joined into a final tree whose weight is 13, the sum of the two weights 6 and 7 (see page 2). Note that you can easily come up with **an alternative tree by using a different ordering strategy to order trees of the same weights**. In that case, the **bit patterns for each character are different, but the total number of bits used to encode** `"go go gophers"` **is the same**.

Implementing Huffman coding and decoding

Now, we will present the basic steps in Huffman coding and decoding.

Coding or compression

To compress a file (sequence of characters) you need a table of bit encodings, i.e., a table giving a sequence of bits used to encode each character. This table is constructed from a coding tree using root-to-leaf paths to generate the bit sequence that encodes each character. A compressed file is obtained using the following top-level steps:

1. Build a Huffman coding tree based on the number of occurrences of each ASCII character in the file.

2. Build a table of Huffman codes for all ASCII characters that appear in the file.

3. Read the file to be compressed (the plain file) and process one character at a time. To process each character find the bit sequence that encodes the character using the table built in the previous step and write this bit sequence to the compressed file.

The main challenge here is that when you encode an ASCII character read from the file, the Huffman code is typically shorter than 8 bits. However, most systems allow you to write to a file a byte or 8 bits at a time. It becomes necessary for you to accumulate the Huffman codes for a few ASCII characters before you write an 8-bit byte to the output file.

To compress the string `"go go gophers"` for example, we read from the string one character at a time. The Huffman code for `'g'` is `00` (**for Huffman code, left to right is least significant to most significant**), we cannot write to the file yet. We read the next character `'o'`, whose Huffman code is `01`. Again, we cannot write to the output file because the total number of bits is only 4. Now, we read the next character `Space`, whose Huffman code is `101`. We have now accumulated 7 bits. Now, we read `'g'` again. The least significant bit of the Huffman code of `'g'` (`0`) is used to complete the byte, and we can now print a byte of bit pattern `01011000` (**in the conventional notation that the left most bit is most significant**) to the output file.

The most significant bit of the Huffman code of `'g'` (`0`) is the least significant bit of the next byte in the file. This byte will also contain the bits of the next three characters `'o'` (`01`), `Space` (`101`), and `'g'` (`00`), and the 8-bit pattern written to the file is `00101100`.

The following byte contains bits of `'o'` (`01`) and `'p'` (`1110`), and the two least significant bits of `'h'` (`11`) for a bit pattern of `11011110`. The two most significant bits of `'h'` (`01`), the bits of `'e'` (`1100`), and the two least significant bits of `'r'` (`11`) form the next byte of bit pattern `11001110`.

The last byte in the file contains only 5 useful bits: the two most significant bits of `'r'` (`11`) are in the least significant positions of this byte and the 3 bits of `'s'` (`100`). As unused bits, the 3 most significant bits of the last byte are 0. The bit pattern is `00000111`.

(Of course, we could code (compress) a file based on any strictly binary coding tree that is not constructed with Huffman's algorithm. In other words, we could have constructed a non-Huffman strictly binary coding tree based on the ASCII characters that appear in the file in step 1. We could still build a table of codes for all ASCII characters that appear in the file and the strictly binary coding tree in step 2, and use the codes to build a coded file in step 3. It is likely that such a coded file will use more bits than a Huffman-coded file. Moreover, such an approach does not guarantee that the coded file will represent the original file with fewer bits.)

Header Information

The compression program must also store some header information in the compressed file that will be used by the decompression program. At the beginning of the compressed file, there are three long integers:

- The total number of bytes in the compressed (or coded) file, as a long integer.
- The total number of bytes storing the topology of the Huffman coding tree, as a long integer.
- The total number of bytes in the original uncompressed file, as a long integer.

Following the three long integers, we store the topology of the Huffman coding tree, followed by the encoding of the original text using Huffman codes. We have described the encoding of the original text in the earlier sections. Now, we focus on how the topology of a coding tree is stored.

To store the tree at the beginning of the file (after the three long integers), we use a pre-order traversal, writing each node visited as follows: when you encounter a leaf node, you write a 1 followed by the ASCII character of the leaf node. When you encounter a non-leaf node, you write a 0.

For our example, the topology information is stored as `"001g1o001s1 001e1h01p1r"`, based on pre-order traversal. In this example, we use characters `'0'` and `'1'` to distinguish between non-leaf and leaf nodes. As there are eight leaf nodes, there are eight `'1'` characters and seven `'0'` characters for non-leaf nodes. This approach used a total of 23 bytes.

The representation of a Huffman coding tree can be made more economical if we use bits 0 and 1 to distinguish between non-leaf and leaf nodes. In this example, there will be a total of 79 bits (64 bits for the ASCII codes of the eight leaf nodes, 8 1-bits for the leaf nodes, and 7 0-bits for the non-leaf nodes). The challenge here is that both the compression and decompression programs would have to handle bits instead of characters.

For example, in the bit-based approach, the first 16 bits (or the first 2 bytes **in the convention that left to right is most significant to least significant**) describing the topology of the Huffman tree constructed for encoding the string `"go go gophers"` are `00111100 11111011`

The least significant two bits of the first bytes are 0-bits (for non-leaf nodes). The next bit is a 1-bit (for a leaf node). It is followed by the ASCII code for `'g'`, which has a bit pattern of `01100111`. The 5 least significant bits (`00111`) are in this byte. The 3 most significant bits (`011`) are in the least significant position of the next byte. This is followed by another 1-bit, followed by `1111`, the 4 least significant bits of the ASCII code for `'o'`, which has a bit pattern of `01101111`.

Note that ASCII code for 'g' straddles two bytes. Similarly, the ASCII code for 'o' straddles two bytes.

In the bit-based representation of the Huffman coding tree, the last byte may not contain 8 bits. In this case, the most significant unused bits of the last byte should be assigned 0.

(If we use a non-Huffman strictly binary coding tree, the second long integer of the coded file would correspond to the total number of bytes storing the topology of the corresponding coding tree.)

Decoding or decompression

Given a compressed (or coded) file, which contains the header information, followed by a bit stream corresponding to the encoding of the original file, the decompression program should perform the following tasks:

1. Build a Huffman coding tree based on the header information. You would probably need the second long integer stored at the beginning of the compressed file to help with the construction.
2. To decode the file, we use the bit stream starting at the location after the header information. The decoding terminates when the number of characters decoded matches the number of characters stored as the third long integer at the beginning of the compressed file. We must initially start from the root node of the Huffman coding tree. When we are at a leaf node, we print the corresponding ASCII character to the output file and re-start from the root node again. When we are at a non-leaf node, we have to use a bit from the compressed file to decide how to traverse the Huffman coding tree (0 for left and 1 for right).

(If the coded file was "compressed" using a non-Huffman strictly binary coding tree, step 1 would reconstruct a strictly binary coding tree, and step 2 would use the strictly binary coding tree for decoding.)

Byte to bits and bits to byte

A challenge is in the reading of a bit from or writing a bit to a compressed file. Recall that the smallest unit that you can read from/write to a file is a byte. It is important that we use the same order to read a bit from/write a bit to a byte for a pair of compression/decompression programs to work correctly. For this assignment, **the order in which you read a bit from a byte or write a bit to a byte should be from the least significant position to the most significant position.**

A strategy to read a bit or to write a bit to a file is to always read a byte from or write a byte to the file. You should maintain an index to indicate where you are in that byte. For example, when you first read in a byte, the index should be pointing to the least significant position. After you have read in one bit, the index should be updated to point to the next more significant position. When the index is beyond the most significant position, you have exhausted the entire byte, and the next bit should be from the least significant position of the next byte read.

Similarly, for writing, you should have an "empty" byte to begin with, with the index initially pointing to the least significant position. After you have written in one bit, the index should be updated to point to the next more significant position. When the index is beyond the most significant position, you have filled in a byte, and you should write the filled byte to the file and use an new "empty" byte for the next bit.

You also have to write an 8-bit ASCII character that straddles two bytes to a compressed file to represent the Huffman coding tree. For decompression, you have to read an 8-bit ASCII character that straddles two bytes in the compressed file when you want to re-construct the Huffman coding tree (for decoding later).

Here are some bit-wise operations that may be useful for your program. Assume that you have two integers `a` and `b`.

```
b = a >> 5;
```

This statement takes the bits stored in `a` (**in the convention that left to right is most significant to least significant**), shifts them right by 5 bits, and stores the results in `b`. Bits shifted out of the least significant position are dropped. The most significant bit of `a` is replicated and shifted right.

```
b = a << 5;
```

This statement takes the bits stored in `a`, shifts them left by 5 bits, and stores the results in `b`. Bits shifted out of the most significant position are dropped. 0's are shifted into the least significant bit.

If you are not dealing with unsigned integers, the left shift may turn a positive number into a negative number (and vice versa). If you are dealing with unsigned integers, the right shift operation will shift 0's into the most significant bit.

To extract the most significant 3 bits of an ASCII character that is stored in an integer `a`, and store the 3 bits at the least significant positions of `b`, we can use shift operations.

```
b = a >> 5;
```

You may want to extract the second most significant bit and the third most significant bit of an ASCII character that is stored in an integer `a`. Moreover, you want to keep the two extracted in integer `b` at the same significant positions in `a`. You can use a mask to achieve that. (Note that I may not be showing you the most efficient way to do it. I am trying to illustrate different bit-wise operations that may be useful to you.)

```
int mask = (0xFF >> 6) << 5;
b = a & mask;
```

`0xFF` corresponds to a bit pattern of `11111111`. The right shift operation keeps two 1's at the least significant positions. The left shift operation moves the two 1's to the correct positions. The bit-wise AND operation gives us the correct two bits in `a` and stores them in `b`.

Now, assume that `a` and `b` are integers storing two ASCII characters; Now suppose you want to combine the 3 most significant bits of ASCII character in a and the 5 least significant bits of ASCII character in `b` to form a new 8-bit pattern to be stored in an integer c. Moreover, you want the 3 bits from a to occupy the less significant positions and the 5 bits from b to occupy the more significant positions. We can use the bit-wise OR operation as follows:

```
int mask = 0xFF >> 3;
c = (b & mask) << 3;
c = c | (a >> 5);
```

What your program should do

You will write a main function that takes in 6 arguments, which are names of input/output files. The argument `argv[1]` should be the name of the input (compressed/coded) file to be decoded. This input file will have the format as described earlier: three long integers, the topology of the coding tree stored using bit-based representation under pre-order traversal (with perhaps padding 0-bits at the last incomplete byte), encoding of the original file using the codes derived from the coding tree (with perhaps padding 0-bits at the last incomplete byte). Note that the coding tree in the given input file may or may not be a Huffman coding tree of the original text. If it is a Huffman coding tree, the encoding would be optimal in that it uses fewest number of bits for encoding.

Your program should re-construct the coding tree stored in the input file, and store the topology information of the tree using character-based representation under pre-order traversal in an output file specified in the argument `argv[2]`.

Using the re-constructed coding tree, your program should decode the input file and store the decoded file in an output file specified in the argument `argv[3]`. The decoded file should be an exact replica of the original file.

To decide whether the provided coding tree is a Huffman coding tree, it is easier to construct a Huffman coding tree of the original file (or decoded file) and determine the number of bits you would need to encode the original file. This number should not be larger than the number of bits required in the input file `argv[1]`. These intermediate steps are captured by the next two output files produced by your program.

`argv[4]` specifies the name of the output file that should contain the frequencies of occurrences of all 256 ASCII characters of the decoded file. The number of bytes in this file should be $256 \times$ `sizeof(long)`. Every `sizeof(long)` bytes correspond to the count of an ASCII value in the decoded file. The counts are ordered from ASCII value 0 to ASCII value 255. While it is fine to read from the decoded file specified in `argv[3]` to perform the counting, that is not necessary. You may also count as you decode the input file (`argv[1]`).

`argv[5]` specifies the name of the output file that should contain the topology information of the constructed Huffman coding trees using the character-based representation. Essentially, your program should construct a Huffman coding tree based on the frequencies of occurrences of ASCII characters in the decoded file and output the topology information to the file specified by `argv[5]`.

With the coding tree in the input file specified by `argv[1]` and the Huffman coding tree you have constructed, you will write four numbers to the file specified by `argv[6]`. The first pair of numbers together specify the number of bits required by the coding tree in the input file to encode the original file. The second pair of numbers together specify the number of bits required by the Huffman coding tree to encode the original file. Each pair of numbers is represented by a long integer `long` and an integer `int`. The long integer stores the number of full bytes required by the encoding bits, and the integer stores the number of encoding bits in the incomplete byte. For example, for the original file/string “go go gophers”, 37 encoding bits required by Huffman coding, the long integer would store $\lfloor 37/8 \rfloor = 4$ and the integer would store

37 mod 8 = 5. You should use `fwrite` to write the first pair of numbers (long integer `long` and integer `int`) and second pair of numbers to the output file specified by `argv[6]`.

Assuming that the input file `gophers_huff.hbt` resides in folder `encoded`, we can call your program `pa2` as follows:

```
    ./pa2 encoded/gophers_huff.hbt gophers.tree gophers.ori gophers.count gophers.htree
gophers.eval
```

The output stored in `gophers.tree` should be a character-based representation of the coding tree that has been stored in the input file `encoded/gophers_huff.hbt` using a bit-based representation. The decoded file is stored in `gophers.ori`. The frequencies of occurrences of ASCII characters in the original file (decoded file) are stored in `gophers.count`, and the Huffman coding tree constructed by your program is stored in `gophers.htree`. The numbers specifying the numbers of encoded bits required to encode the original file (decoded file) by the coding tree in the input file and the constructed Huffman coding tree are stored in `gophers.eval`.

The returned value of the main function should be `EXIT_SUCCESS` if the program is able to produce all required output files. Otherwise, the returned value should be `EXIT_FAILURE` if the number of arguments is incorrect, the input file does not exist, the program runs out of memory, a file cannot be opened for output, etc. Although you should technically check whether a read operation or a write operation is successful, you may assume for this assignment that if a file can be opened successfully, all reasonable file operations will be successful.

You may also assume that every input file, if it can be opened, is in a valid format. We will not use an input file that is of an invalid format to test your program. (Of course, as a competent programmer, you should check for that. However, that is not a requirement of the assignment.)

Evaluation of your program

You program will be evaluated based on the five output files produced for a number of test cases. The test cases altogether account for 100% for the assignment. The first output files of all test cases account for 30%, the second output files account for 30%, the third output files account for 5%, the fourth output files account for 30%, and the fifth output files account for 5%.

Up to 5 points will be deducted if your `main` function does not return the correct value. 50 points will be deducted if your program has memory issues (as reported by `valgrind`).

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case.

What you are given

Some samples are given. The folder `decoded` contains 5 original (and decoded) files (4 text files, namely, `gophers`, `lorum`, `stone`, and `woods`, and 1 binary file, `binary1`).

The folder `encoded` contains two versions of the encoded files of the files in the folder `decoded`. For example, the file `gopher` has two encoded versions, `gophers_huff.hbt` has been encoded using a Huffman coding tree and `gophers_nonhuff.hbt` has been encoded using an arbitrary, but valid coding tree. In each of these encoded files, the topology of the coding tree (Huffman or non-Huffman) is stored using the bit-based representation.

The corresponding character-based representations of the coding trees stored in the encoded files in the folder `encoded` are stored in the folder `tree`. For example, the character-based representation of the coding tree in `gophers_nonhuff.hbt` is stored in `gophers_nonhuff.tree` and the character-based representation of the Huffman coding tree in `gophers_huff.hbt` is stored in `gophers_huff.tree`. If there are $n > 0$

distinct ASCII characters in an original file (or a decoded file), there should be exactly $3n-1$ characters in the corresponding file in the folder tree. For an empty coding tree, the corresponding file to store its character-based representation should be an empty file.

Each file in the folder `count` contains the frequencies of occurrences of all 256 ASCII characters of the corresponding file in the folder `decoded`.

Each binary file in the folder `eval` contains information that gives us the numbers of encoding bits in the corresponding file in the `encoded` folder. For example, the file `gophers_nonhuff.eval` corresponds to the encoded file `gophers_nonhuff.hbt` and the file `gophers_huff.eval` corresponds to the encoded file `gophers_huff.hbt`. In the file `gophers_huff.eval`, the first pair of numbers should match the second pair of numbers.

Consider the following command:

```
./pa2 encoded/gophers_huff.hbt gophers.tree gophers.ori gophers.count gophers.htree
gophers.eval
```

The output file `gophers.tree` should match the provided file `tree/gophers_huff.tree`; `gophers.ori` should match `decoded/gophers`; `gophers.count` should match `count/gophers.count`; `gophers.eval` should match `eval/gophers_huff.eval`. Recall that there may be multiple Huffman coding trees. `gophers.htree` would match `tree/gophers_huff.tree` only if you construct the Huffman coding tree exactly the way we outlined earlier.

Consider the following command:

```
./pa2 encoded/gophers_nonhuff.hbt gophers.tree gophers.ori gophers.count gophers.htree
gophers.eval
```

The output file `gophers.tree` should match the provided file `tree/gophers_nonhuff.tree`; `gophers.ori` should match `decoded/gophers`; `gophers.count` should match `count/gophers.count`; `gophers.eval` should match `eval/gophers_nonhuff.eval`. Again, `gophers.htree` would match `tree/gophers_huff.tree` only if you construct the Huffman coding tree exactly the way we outlined earlier.

What you should turn in

You should turn in all the `.c` and `.h` files that you have created for this assignment. For example, I can imagine that you have a `pa2.c` file that contains the main function, a `huffman.c` file that contains all other important functions, and a `huffman.h` file that defines the structures and declares the functions in `huffman.c`. You should create a zip file called `pa2.zip` that contains the `.c` and `.h` files. **Your zip file should not contain a folder (that contains the source files)**.

```
zip pa2.zip *.c *.h
```

You should submit `pa2.zip` **to Brightspace.**

If you want to use a `Makefile` for your assignment, please include the `Makefile` in the zip file. In that case, you can create the zip file as follows:

```
zip pa2.zip *.c *.h Makefile
```

If the zip file that you submit contains a `Makefile`, we use that file to make your executable (by typing "`make pa2`" at the command line to create the executable called `pa2`).

Without a `Makefile`, we will use the following command to compile your submission:

```
gcc -std=c99 -Wall -Wshadow -Wvla -pedantic -O3 *.c -o pa2
```

Only `.c` files, `.h` files, and `Makefile` will be used to generate an executable. Other files in the submitted zip file will not be made available for compilation. If your program does not compile, you do not get any credit for the assignment. Code that contains memory problems (as reported by `valgrind`) will be subject to a penalty of 50% of the total possible points.