COMP 211 : Principles of Imperative Computation

Fall 2014

Assignment 9: Huffman Coding

Due: Friday, 5 Dec, 2014 by 11:59pm

For the programming portion of this week's homework, you'll write a C_0 implementation of a popular compression technique called Huffman coding.

1 Getting the code

The code handout for this assignment is available from the course web page. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.

To download and unzip the code, do the following:

1.1 Mac OS

1. On the assignments page of the course web site, right-click the Programming 8 `Code` link, and then click "Save Link as...", and choose to save it in `Documents/comp211/`.

2. Start a new terminal window and do the following:

   ```
   cd comp211
   tar -xzvkf hw9-handout.tgz
   ```

   This will make a directory called `Documents/comp211/hw9-handout` and extract all of the code for the assignment.

3. The file `hw9-handout/README.txt` contains a description of the various files, and instructions on running them.

1.2 Windows

1. On the assignments page of the course web site, right-click the Programming 8 `Code` link, and then click "Save Link as...", and choose to save it in your `comp211/`. To find this directory in your web browser, look in

   ```
   C:\cygwin\home\YOURNAME\comp211\
   ```

2. Start a new terminal window and do the following:

```
cd comp211
tar -xzkvf hw9-handout.tgz
```

   This will make a directory called `comp211/hw9-handout` and extract all of the code for the assignment.

3. The file `hw9-handout/README.txt` contains a description of the various files, and instructions on running them.

2 Huffman Coding

2.1 Data Compression: Overview

Whenever we represent data in a computer, we have to choose some sort of *encoding* with which to represent it. When representing strings in C_0, for instance, we use ASCII codes to represent the individual characters. Other encodings are possible as well. The UNICODE standard (as another encoding example) defines a variety of character encodings with a variety of different properties. The simplest, UTF-32, uses 32 bits per character.

Under the ASCII encoding, each character is represented using 7 bits, so a string of length n requires $7n$ bits of storage, which we usually round up to $8n$ bits or n bytes. For example, consider the string `"more free coffee"`; ignoring spaces, it can be represented in ASCII as follows with $14 \times 7 = 98$ bits:

$$1101101 \cdot 1101111 \cdot 1110010 \cdot 1100101 \cdot 1100110 \cdot$$
$$1110010 \cdot 1100101 \cdot 1100101 \cdot 1100011 \cdot 1101111 \cdot$$
$$1100110 \cdot 1100110 \cdot 1100101 \cdot 1100101$$

This encoding of the string is rather wasteful, though. In fact, since there are only 6 distinct characters in the string, we should be able to represent it using a custom encoding that uses only $\lceil \log 6 \rceil = 3$ bits to encode each character. If we were to use the custom encoding shown in Figure 1, the string would be represented with only $14 \times 3 = 42$ bits:

$$011 \cdot 100 \cdot 101 \cdot 001 \cdot 010 \cdot$$
$$101 \cdot 001 \cdot 001 \cdot 000 \cdot 100 \cdot$$
$$010 \cdot 010 \cdot 001 \cdot 001$$

In both cases, we may of course omit the separator "$\cdot$" between codes; they are included only for readability.

If we confine ourselves to representing each character using the same number of bits, i.e., a *fixed-length encoding*, then this is the best we can do. But if we allow ourselves a *variable-length encoding*, then we can take advantage of special properties of the data: for instance, in the sample string, the character `'e'` occurs very frequently while the characters `'c'` and `'m'` occur very infrequently, so it would be worthwhile to use a smaller bit pattern to encode

Character	Code
`'c'`	000
`'e'`	001
`'f'`	010
`'m'`	011
`'o'`	100
`'r'`	101

Figure 1: A custom fixed-length encoding for the non-whitespace characters in the string `"more free coffee"`.

Character	Code
`'e'`	0
`'o'`	100
`'m'`	1010
`'c'`	1011
`'r'`	110
`'f'`	111

Figure 2: A custom variable-length encoding for the non-whitespace characters in the string `"more free coffee"`. Note: this example is also provided in the handout code, but there the character set additionally includes the newline character, so the codes for each character will be different.

---

the character `'e'` even at the expense of having to use longer bit patterns to encode `'c'` and `'m'`. The encoding shown in Figure 2 employs such a strategy, and using it, the sample string can be represented with only 34 bits:

$$1010 \cdot 100 \cdot 110 \cdot 0 \cdot 111 \cdot$$
$$110 \cdot 0 \cdot 0 \cdot 1011 \cdot 100 \cdot$$
$$111 \cdot 111 \cdot 0 \cdot 0$$

Since this encoding is *prefix-free*—no code word is a prefix of any other code word—the "$\cdot$" separators are redundant here, too.

It can be proven that this encoding is optimal for this particular string: no other encoding can represent the string using fewer than 34 bits. Moreover, the encoding is optimal for *any* string that has the same distribution of characters as the sample string. In this assignment, you will implement a method for constructing such optimal encodings developed by David Huffman.

2.2 Huffman Coding: A Brief History

"Huffman coding" is an algorithm for constructing optimal prefix-free encodings given a frequency distribution over characters. It was developed in 1951 by David Huffman when he was a Ph.D student at MIT taking a course on information theory taught by Robert Fano. It was towards the end of the semester, and Fano had given his students a choice: they could

either take a final exam to demonstrate mastery of the material, or they could write a term paper on something pertinent to information theory. Fano suggested a number of possible topics, one of which was efficient binary encodings: while Fano himself had worked on the subject with his colleague Claude Shannon, it was not known at the time how to efficiently construct optimal encodings.

Huffman struggled for some time to make headway on the problem and was about to give up and start studying for the final when he hit upon a key insight and invented the algorithm that bears his name, thus outdoing his professor, making history, and attaining an "A" for the course. Today, Huffman coding enjoys a variety of applications: it is used as part of the DEFLATE algorithm for producing ZIP files and as part of several multimedia codecs like JPEG and MP3.

2.3 Huffman Trees

Recall that an encoding is *prefix-free* if no code word is a prefix of any other code word. Prefix-free encodings can be represented as binary *full* trees with characters stored at the leaves: a branch to the left represents a 0 bit, a branch to the right represents a 1 bit, and the path from the root to a leaf gives the code word for the character stored at that leaf. For example, the encodings from Figures 1 and 2 are represented by the binary trees in Figures 3 and 4, respectively.

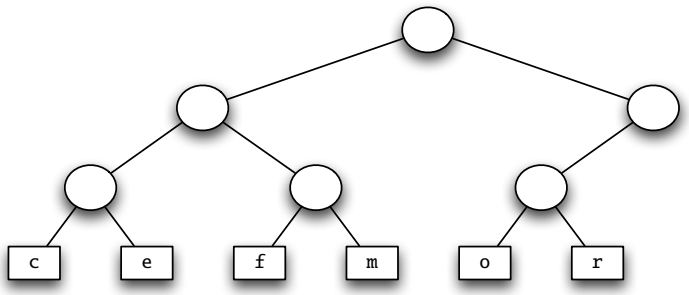


Figure 3: The custom encoding from Figure 1 as a binary tree.

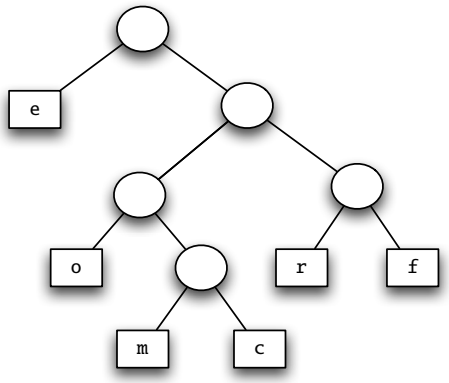


Figure 4: The custom encoding from Figure 2 as a binary tree.

The tree representation makes it clear that *frequently-occurring characters have shorter paths to the root*. We can see this property clearly if we label each subtree with the total frequency of the characters occurring at its leaves, as shown in Figure 5. A frequency-annotated tree is called a *Huffman tree*.

Huffman trees have a recursive structure: a Huffman tree is either a leaf containing a character and its frequency, or an interior node containing the combined frequency of two child Huffman trees. Since only the leaves contain character data, we draw them as rectangles to distinguish them from the interior nodes, which we draw as circles.

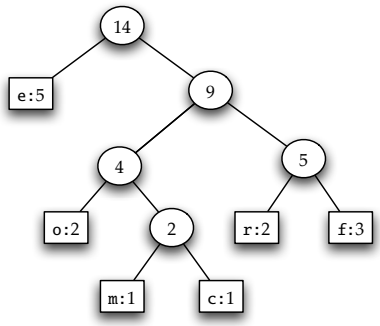


Figure 5: The custom encoding from Figure 2 as a binary tree annotated with frequencies, i.e., a Huffman tree.

We represent both kinds of Huffman tree nodes in C_0 using a `struct hufftree_node`:

```
typedef struct hufftree_node* hufftree;
struct hufftree_node {
    char value;  // '\0' except at leaves
    int frequency;
    hufftree left;
    hufftree right;
};
```

The `value` field of an `hufftree` should consist of a character `'\0'` everywhere except at the leaves of the tree, and every interior node should have exactly two non-NULL children. These criteria give rise to the following definitions:

> An `hufftree` is a *valid hufftree* if it is non-`NULL`, its `frequency` is strictly positive, and it is either a *valid hufftree leaf* or a *valid hufftree interior node*.
>
> An `hufftree` is a *valid hufftree leaf* if its `value` is *not* `'\0'` and its `left` and `right` children are `NULL`.
>
> An `hufftree` is a *valid hufftree interior node* if its `value` is `'\0'`, its `left` and `right` children are *valid hufftrees*, and its `frequency` is the sum of the `frequency` of its children.

Note that these definitions are *recursive*: being a valid hufftree refers to being a valid hufftree node, and a valid hufftree node has valid hufftrees as its children. Since we do not expect you to write recursive functions this semester (you will get lots of practice with that in COMP 212), we have provided the following spec functions

Function:	**Returns true iff...**
`bool is_hufftree_leaf(hufftree H)`	the node is a leaf
`bool is_hufftree_internal_node(hufftree H)`	the node is an internal node
`bool is_hufftree(hufftree H)`	the tree is a Huffman tree

We have also provided the the following utility functions, since they are simplest to define recursively:

`int hufftree_size(hufftree H)`	return the number all nodes in the tree
`int hufftree_count_leaves(hufftree H)`	return the number of leaves in the tree

2.4 Finding Optimal Encodings

Huffman's key insight was to use the frequencies of characters to build an optimal encoding tree from the bottom up. Given a set of characters and their associated frequencies, we can build an optimal Huffman tree as follows:

1. Construct leaf Huffman trees for each character/frequency pair.

2. Repeatedly choose two minimum-frequency Huffman trees and join them together into a new Huffman tree whose frequency is the sum of their frequencies.

3. When only one Huffman tree remains, it is the result.

This is an example of a *greedy algorithm* since it makes locally optimal choices that nevertheless yield a globally optimal result at the end of the day. Selection of a minimum-frequency tree in step 2 can be accomplished using a *priority queue* based on a heap. A sample run of the algorithm is shown in Figure 6.

Task 1 (10 pts) *Write a function*

```
hufftree build_hufftree(char[] chars, int[] freqs, int n)
//@requires 1 < n && n <= \length(chars) && \length(chars) == \length(freqs);
//@ensures is_hufftree(\result);
//@ensures hufftree_count_leaves(\result) == n;
```

that constructs an optimal encoding for an ***n****-character alphabet given by the first n characters of* `chars` *and* `freqs`*. Like in Homework 3, the* ***chars*** *array contains the characters of the alphabet and the* ***freqs*** *array their frequencies, where* ***chars[i]*** *occurs with frequency* ***freqs[i]****.*

See`lib/heaps.c0` for an implementation of priority queues; you do not need to modify the heaps implementation, and you should only use functions described in the interface. The file `pq-client.c0` instantiates the priority queue library so that the priority queue's elements are Huffman trees, ordered by their frequency.

2.5 Encoding

Once you successfully constructed a Huffman tree, the next task is encoding a given string. To encode a *character* you traverse the tree to the character you want, writing down a 0 every time you take a lefthand branch, and an 1 every time you take a righthand branch. As an example, we encode "roomforcreme" using the encoding from Figure 7:

```
110 100 100 1010 111 100 110 1011 110 0 1010 0
r   o   o   m    f   o   r   c    r   e m    e
```

Bitstrings can be represented in C_0 as ordinary strings composed only of characters `'0'` and `'1'`.

To encode a *string*, we need to convert each character to its bitstring encoding, returning the resulting string. Given a Huffman tree, we can retrieve the bitstring encoding corresponding to a particular character by traversing the tree. Encoding a string via this method, however, is inefficient, because we have to search the entire tree once for each character we encode.

Instead, we optimize this process by using a hashtable to remember the encoding of each character. This way, we need to traverse the tree only once.

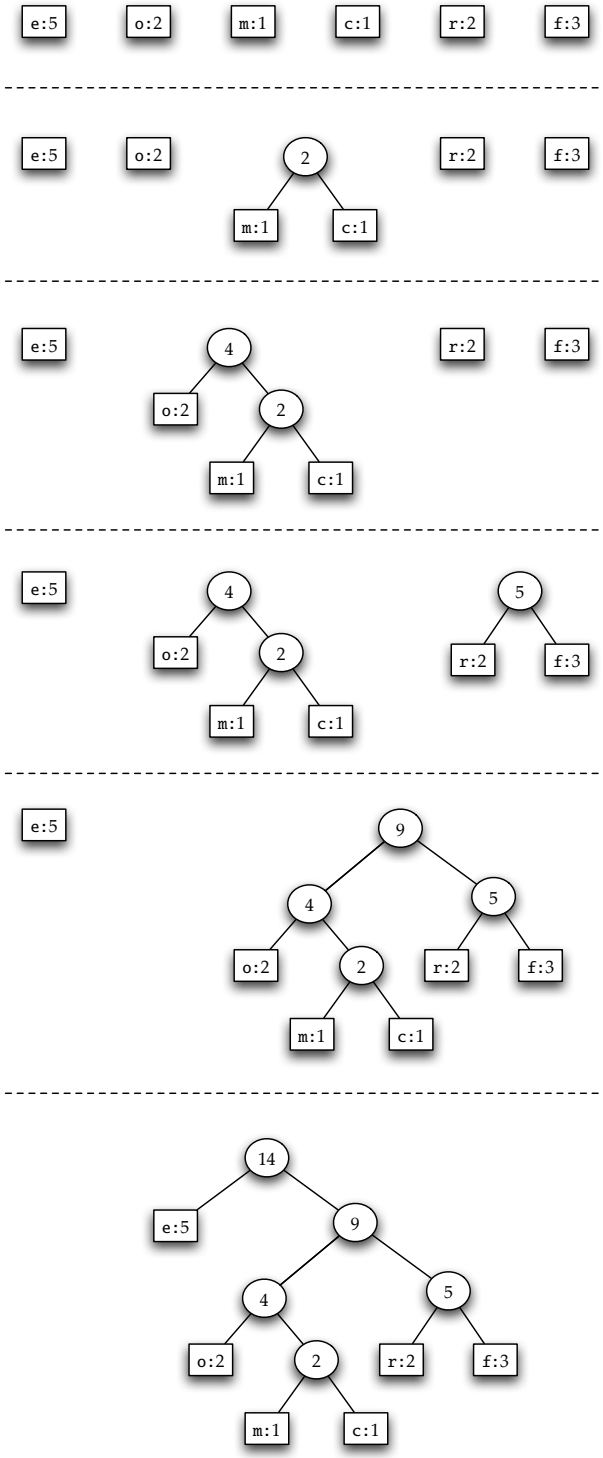
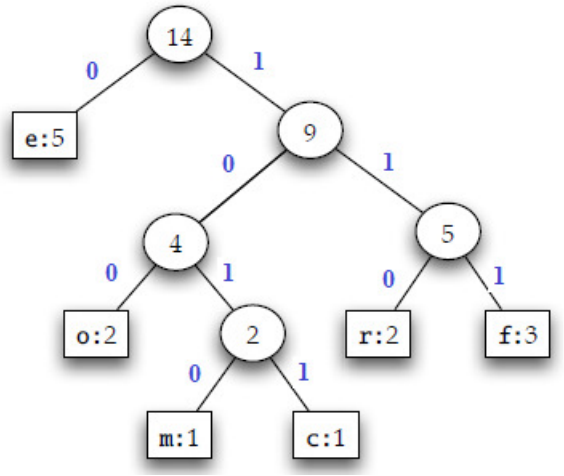


Figure 6: Building an optimal encoding using Huffman's algorithm. Note that in step 4, we could pick either the 'e' leaf or the 'r-f' tree, because both have the same priority.



char	encoding
m	1010
o	100
r	110
e	0
f	111
c	1011

Figure 7: A Huffman tree annotated with 0's and 1's.

Task 2 (7 pts) *Write a function*

```
hashtable hufftree_to_hashtable(hufftree H)
```

mapping characters to their corresponding Huffman-encoded bitstrings.

Use the code in the included `lib/hashtable.c0` as your implementation of hash tables. The file `hashtable-client.c0` instantiates the library with the necessary client types and functions for a table mapping characters to strings.

Hint: Use a stack. The file `lib/stack.c0` contains our implementation of stacks from lecture, and the client instantiation `stack-client.c0` defines the stack's elements to be a pair of a Huffman tree and a string (`struct hufftree_with_path`). The stack's entries will be Huffman trees paired with a bit-string representing the path to that tree. Create an initial stack with one element, the pair `(H,"")`. While the stack is non-empty, pop off the top of the stack, and if the top of the stack is a leaf, then insert the leaf's character and the path into the hashtable you are creating. Otherwise, if it's an internal node, then push the node's children (paired with the paths to them) onto the stack.

Task 3 (2 pts) *Write a function*

```
bitstring encode_map(hashtable M, string input)
```

that ***efficiently*** *encodes the given string* `input`*, assuming* `M` *maps the characters in* `input` *to their codes. Your implementation should have* $O(n)$ *asymptotic complexity, where* n *is the length of* `input`*.*

Task 4 (1 pts) *Write a function*

```
bitstring encode(hufftree H, string input)
```

that encodes the given string `input`*, by using* `encode_map` *and creating an appropriate hash map.*

2.6 Decoding Bitstrings

Huffman trees are well-suited to the task of decoding encoded data. Given an encoded bitstring and the Huffman tree that was used to encode it, decode the bitstring as follows:

1. Start at the roof of the Huffman tree.
2. Repeatedly read bits from the bitstring: when you read a 0 bit, follow the left branch, and when you read a 1 bit, follow the right branch.
3. Whenever you reach a leaf, output the character at that leaf and reset the pointer to the root of the Huffman tree.

As an example, we can use the encoding from Figure 7 to decode the following message:

```
1 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 0
    r     o     o       m     f     o     r       c     r e       m e
```

Task 5 (6 pts) *Implement a function*

```
struct decode_result {
  char decoded;
  string leftovers;
};

// Decodes the bit string  based on the Huffman tree H.
struct decode_result* decode(hufftree H, string bits)
//@requires is_hufftree(H);
//@requires is_bitstring(bits);
;
```

that decodes `bits` *based on the Huffman tree* `H`*.*

If some prefix of `bits` *is a valid code for the Huffman tree, then the function should return a* `decode_result` *whose* `decoded` *field contains the character that that prefix decodes to, and whose* `leftovers` *is the rest of* `bits` *after removing that prefix. Otherwise, if the provided bits do not lead to a leaf in the tree, the function should return* `NULL`*.*

The reason we have `decode` work on a prefix of the string is that, for a variable length code, when you are given a sequence of bits representing a whole file, you don't know when the code for one character ends and when the code for the next one begins.

2.7 Summary of data structures

This homework is a good example of programming by combining different data structures. Here is a summary to help you keep them straight:

- Building a Huffman tree converts a dictionary (represented as a pair of arrays) to a Huffman tree, using a priority queue (whose elements are Huffman trees).
- To encode, we convert a Huffman tree to a hashtable (mapping characters to bitstrings) using a stack (of pairs of Huffman trees and bitstrings).
- To decode, we just use the Huffman tree.
- The hashtable elements are a struct `char_with_string` that pairs a character with a bitstring.
- The priority queue elements are Huffman trees.
- The stack elements are a struct `hufftree_with_path` that pairs a Huffman tree with a bitstring.
- Decode returns a `decode_result`, which is either `NULL` or a pair of a character and a bitstring.

2.8 Testing

When you compile your code according to the instructions in `README.txt`, it will create two executables named `encode` and `decode`.

You can run `encode` as follows:

```
$ ./encode -f freqs/morefreecoffee.txt -i texts/coffee.txt
01101011001100100111101010100001111 0111
```

Encode takes two arguments: The first is a frequency file, in this case `freqs/moreforeecoffee.txt`, whose contents are

```
m 1
o 2
r 2
e 5
f 3
c 1
\n 1
```

This gives the frequency of each character (`\n` is newline, which must also be listed—because of this, the codes for each letter will be different than the ones used in the examples earlier in the handout).

The second is a text file to encode, which in this case contains

```
morefreecoffee
```

(with a newline at the end).

Encode prints the bitstring encoding of the file. You can put this in a file as follows:

```
% ./encode -f freqs/morefreecoffee.txt -i texts/coffee.txt > coffee.txt.bits
```

The `>` in the unix command line redirects output to the file `coffee.txt.bits`. **BE CAREFUL BECAUSE THIS WILL OVERWRITE A FILE THAT ALREADY EXISTS.**

The second executable, `decode`, takes two arguments, the first of which is a frequency file, and the second of which is a file containing a bitstring to decode. For example:

```
/decode -f freqs/morefreecoffee.txt -i coffee.txt.bits
morefreecoffee
3
```

Note that you must decode with the same frequency file that the bitstring was encoded with, because you need to decode using the same Huffman tree.

We have also provided additional frequency files

- `freqs/english.txt` which has all letters (and newline) with the frequency that they occur in english text.
- `freqs/english-punc.txt` which has all letters (and newline) and punctuation—but it is given only by a ranked order, rather than actual frequencies (see the task below)

We have also included the Twitter data from Homework 3 in `texts`.

2.9 Bonus Task

The frequency table `freqs/english-punc.txt` is gives only a ranked order of characters. Consequently, the codes it generates are not very efficient. In this extra credit task, you will use your code from Homework 4 to generate a better frequency table and investigate how it improves the Huffman codes.

1. Generate a frequency table from `texts/twitter_200k.txt` using your Homework 4 code. In `count-main.c0`, you will will need to change change `read_words` to a different function named `read_chars` for both the `vocab_filename` and the `tweet_filename`—these are the only two calls to `read_words` in the code. Then use `texts/vocab_ascii.txt` as the vocabulary file, and use your Homework 4 code to generate a table listing the frequencies of the characters in this vocabulary as they occur in `twitter_200k.txt`.

   Note: the Homework 4 code does not produce a frequency for the newline character. You should manually add an entry

   ```
   \n 200000
   ```

   to the frequency file that you create.

2. Encode `texts/twitter_200k.txt` using both `freq/english-punc.txt` and your frequency file.

3. Look at the sizes of the resulting files (here they are named `t2k-english-punc.bits` and `t2k-myfrequencytable.bits` for the included frequency table and the one I generated) using the command `ls -lh`:

   ```
   % ls -lh texts/twitter_200k.txt t2k-english-punc.bits t2k-myfrequencytable.bits
   -rw------- 1 dlicata 410922006 9.6M Mar 20 13:35 texts/twitter_200k.txt
   -rw------- 1 dlicata 410922006  ??M Mar 22 12:55 t2k-english-punc.bits
   -rw------- 1 dlicata 410922006  ??M Mar 22 12:55 t2k-myfrequencytable.bits
   ```

   The `9.6M` means that the original Twitter file is 9.6 megabytes (roughly 9.6 million bytes/characters, except a megabyte is really `1024` characters). You will see sizes for the resulting bitstring files as well. The size for your frequency table should be less than for the one we provided.

4. However, you should notice that both of the bit files are *much larger than the original data!* What went wrong? Isn't compression supposed to make files smaller?

5. The problem is that these files containing the bits of the code are *text files*. Thus, if a character `c` is coded as `0101`, it will take 4 characters to represent a single character! To see any savings, we need to create *binary files*, where each eight bits of the bitstring are stored on disk as a single byte. We have provided utilites `pack` and `unpack` for converting bitstrings to and from binary.

   Run `'make pack'` to compile the C code for pack/unpacking. To pack a file, run `./pack <file>` This creates `<file>.pack` To unpack a file, run `./unpack <file>`. This creates `<file>.unpack`.

   For example,

   ```
   ./pack t2k-english-punc.bits
   ```

   will create a binary file `t2k-english-punc.bits.pack`.

   If you have such a binary file, you can unpack it to extract the bitstring file:

   ```
   ./unpack t2k-english-punc.bits.pack
   ```

   will create a binary file `t2k-english-punc.bits.pack.unpack`, which will be the same as `t2k-english-punc.bits` (so if I sent you a compressed binary file, you could decode it).

Task 6 (Extra Credit) *Create a file* `sizes.txt` *containing the sizes of the packed versions of* `texts/twitter_200k.txt` *encoded with both our frequency table and yours. E.g.*

```
size with freqs/english-punc.txt: X MB
size with my frequency table: Y MB
```

Also, calculate how big the compressed version of the file is, as a percantage of the original size—which shows how much compression you are getting out of your Huffman coding algorithm!