Project 1: Huffman Coding

Introduction

In this Project, you will be implementing a tool to compress and uncompress files using Huffman Coding.

Remember that this is a Project, not a PA, and as such, it is much more involved! To be successful, you **absolutely must** read through the **entire** write-up before even attempting to write a single line of code! Familiarize yourself with the components/sections of the write-up so that, when you finally start coding, you will have a better understanding of what the project entails, and when you inevitably get stuck, you will know where in this write-up to look for tips on how to progress.

Deadline: Tuesday, May 28 at 10:00 PM

Late submissions **will not be accepted!** Start early, start often.

Academic Integrity

This Project must be completed **100% independently!** You may only discuss the Project with the Tutors, TAs, and Instructors. You are free to use resources from the internet, but you are not allowed to blatantly copy-and-paste code. If you ever find yourself highlighting a code snippet, copying, and pasting into your Project, you are likely violating the Academic Integrity Policy. If you have any concerns or doubts regarding what you are about to do, *please* be sure to post on the discussion board first to ask us if it is okay.

Grading

- 8 points for empty file
- 8 points for small (10 KB) file in DNA alphabet
- 8 points for medium (20 KB) file in DNA alphabet
- 8 points for large file (50 KB) in DNA alphabet
- 8 points for small (10 KB) file in extended alphanumeric alphabet
- 8 points for medium (20 KB) file in extended alphanumeric alphabet
- 8 points for large file (50 KB) in extended alphanumeric alphabet
- 8 points for small (10 KB) binary file
- 8 points for medium (20 KB) binary file
- 8 points for large (50 KB) binary file

- 10 points for no memory leaks
- 5 points for beating the reference solution on 5 or more of the files
- 5 points for beating the reference solution every time

Overview

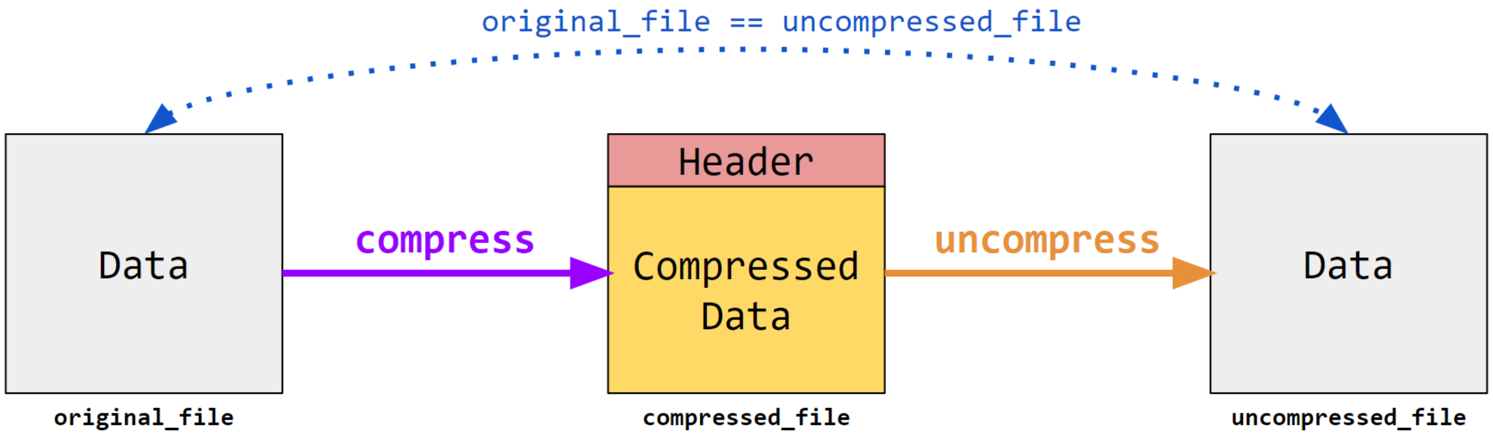
In the age of big data, there is a constant struggle with regard to storage. For example, a single Whole Genome Sequencing experiment can generate over 100 GB of raw data. For years, computer scientists have attempted to reduce the storage size of files without losing any information. In this assignment, we will be implementing our own file compression tool using the Huffman algorithm.

Task: File Compression and Uncompression

In this project, "all" you have to do is implement two programs: **`compress`** and **`uncompress`**. The usage should be as follows:

```
./compress <original_file> <compressed_file>
./uncompress <compressed_file> <uncompressed_file>
```

For grading purposes, you are guaranteed that `original_file` will be at most 10 MB.



File Compression

The `compress` program will take as input an arbitrary file (`original_file`) and will use Huffman Compression to create a compressed version of the file (`compressed_file`):

1. Construct a Huffman code for the contents of `original_file`
2. Use the constructed Huffman code to encode `original_file`
3. Write the results (with a header to allow for uncompression) to `compressed_file`

File Uncompression

The `uncompress` program will then take as input a compressed file created by your `compress` program (`compressed_file`) and will uncompress the file (`uncompressed_file`):

1. Construct a Huffman code by reading the header of `compressed_file`

2. Use the constructed Huffman code to decode `compressed_file`
3. Write the results to `uncompressed_file`

Note that `uncompressed_file` must be **completely identical** to `original_file`! In other words, calling `diff` should return no differences:

```
diff uncompressed_file original_file # this should not output anything
```

Reference Solution

You have been provided two programs (`refcompress` and `refuncompress`), which are a reference solution implementation of this Project. You can use them to help guide you. Note that the reference binaries were compiled to run on Ed, so if you attempt to run them on a different architecture, they will most likely not work.

Note: Your `compress` is **not** expected to work with `refuncompress`, and your `uncompress` is **not** expected to work with `refcompress`. The provided reference executable files are a matched pair.

Compiling and Running Your Code

We have provided a `Makefile`, but **you are free to modify *anything* in the starter code** that you'd like (including the `Makefile` and any of the `.cpp` or `.hpp` files) however you wish, and you are allowed to use anything in the C++ STL. Our only requirements are the following:

1. We need to be able to compile your code using the `make` command
2. Once compiled, we need to be able to run your `compress` and `uncompress` programs as described above

Project Workflow

As mentioned in the overview, as long as we can (1) compile your program via `make`, (2) run your program as described, and (3) successfully compress + uncompress a file, you are absolutely free to implement the project any way you think is best. However, here are some suggestions just in case you feel a bit lost.

Suggested Development Process

Before doing anything, it is a good idea to create all required `.cpp` files (see the `Makefile`) and write blank `main()` functions in your `compress.cpp` and `uncompress.cpp` files to get your project files to the point that you can compile successfully with `make` (even though the `compress` and `uncompress` executables don't do anything). Once you get things compiling, start working on `compress`, and once you are confident that you have that working reasonably well, start working on `uncompress`. Use print statements, `gdb`, etc. as you develop to make sure you're implementing things correctly.

Suggested Control Flow for `compress`

1. Parse the command line arguments and throw an error message if the user runs your program incorrectly
2. Open the input file for reading
3. Read bytes from the file. Count the number of occurrences of each byte value
4. Use the byte counts to construct a Huffman coding tree. Each unique byte with a non-zero count will be a leaf node in the Huffman tree
5. Open the output file for writing
6. Write enough information (a "file header") to the output file to enable the coding tree to be reconstructed when the file is read by your `uncompress` program
7. Move back to the beginning of the input file to be able to read it, again
8. Using the Huffman coding tree, translate each byte from the input file into its code, and append these codes as a sequence of bits to the output file, after the header
9. Close the input and output files (note that this is handled for you; see Design Notes)

Suggested Control Flow for `uncompress`

1. Open the input file for reading
2. Read the file header at the beginning of the input file, and use it to reconstruct the Huffman coding tree
3. Open the output file for writing

4. Using the Huffman coding tree, decode the bits from the input file into the appropriate sequence of bytes, writing them to the output file
5. Close the input and output files (note that this is handled for you; see Design Notes)

Potential Development Process

If you choose to use the code structure we have laid out in the starter, one reasonable potential development process is the following. Note that this is independent of the "Suggested Development Process" section, which gives an overview of how a functioning `compress.cpp` and `uncompress.cpp` (the source code) should *look* like. This section illustrates our suggested steps to go from the starter code to a functioning `compress` and `uncompress` (the executables):

1. Create `compress.cpp` and `uncompress.cpp` with `main()` functions that don't do anything for now
   - This will allow you to compile your program as you make changes to the other files
   - As you implement the components described below, you can use your `main()` functions in `compress.cpp` and `uncompress.cpp` to test them out one-by-one
   - For right now, think of `compress.cpp` and `uncompress.cpp` as blank canvases to be able to write code to test the other components, and you will write the actual implementation of both programs later
2. Create `HCTree.cpp` and add skeletons of the `HCTree` functions you will need to implement
   - Refer to `HCTree.h` to see what function signatures are declared in the `HCTree` class
3. Implement the `HCTree::build` function to construct a Huffman Tree from symbol frequencies
   - You will want to use `gdb` or print statements to help you trace your tree to make sure it matches what you would get if you constructed it manually by hand (on small example files)
4. Implement the `HCTree` destructor
5. Implement the `HCTree::encode` function to encode a given symbol
6. Implement the `HCTree::decode` function
7. Once you are confident that all of the individual components are working properly, begin to piece them together in the `main` functions of `compress.cpp` and `uncompress.cpp` according to the suggested control flows above
   - Incorporate one step of the control flow, then test it thoroughly to make sure things work to that point, then incorporate the next step, test thoroughly, etc.
   - How you choose to implement your compressed file header is up to you, but we have some recommendations in the "Design Notes" section of this write-up, so be sure to refer to that

Map Out *Your* Ideal Approach

As mentioned, the above suggested control flows and development processes are simply what *we* recommend if you feel lost and don't know where to start. However, *not everybody thinks/works the same!* What we suggest may not be the best approach for you personally. Before you even write a single line of code, read through this guide and make sure you fully understand the required tasks of the project, and then actually sit down and map out how *you personally* want to approach the problem. You are of course free to deviate from or modify the programming approach you design, but simply having a roadmap to help guide you can be *extremely* helpful when coding.

TL;DR: Try not to write a *single line of code* until you actually physically map out how you personally want/plan to approach this project! Feel free to chat with a tutor during Lab Hours if you want some feedback about your approach! :-)

Design Notes

Before you write even a single line of code, it's important to properly design the various components of your `compress` and `uncompress` programs. Here, we will discuss some aspects of the design that you should be sure to think about.

Parsing Command Line Arguments in C++

Remember that, in C++, you can parse command line arguments by writing a `main` function as follows:

```
int main(int argc, char* argv[]) {
    // your program's main execution code
}
```

The `argc` variable provides the number of command line arguments that were provided, and the `argv` variable is an array containing the command line arguments. See the following article to learn more about how to use them:

http://www.cplusplus.com/articles/DEN36Up4/

File I/O

For this assignment, you will have to read data from files and write data to files. To help you with this, in `Helper.hpp` and `Helper.cpp`, we have provided two helper classes that can make it easier to perform file I/O: `FancyInputStream` and `FancyOutputStream`. It is up to you to thoroughly read the header comments of the functions in these two classes to see how to use them. It is a good idea to have the "Project Workflow" open while you look at the function headers for `FancyInputStream` and `FancyOutputStream` to get a better idea of how you can apply various functions of those classes to the `compress` and `uncompress` workflows.

Note that, in C++, `ifstream` and `ofstream` objects are automatically closed by their destructors. Thus, when a `FancyInputStream` or `FancyOutputStream` object is destroyed (e.g. if you create it on the runtime stack and it goes out of scope), the internal `ifstream`/`ofstream` object will *also* automatically get destroyed (and thus will automatically close).

Huffman Tree

One crucial data structure you will need is a **binary trie** (i.e., "code tree" or "encoding tree") that represents a Huffman code. The `HCTree.hpp` header file provides a possible interface for this structure (included in your repo as skeleton code); you can modify this in any way you want.

Additionally, you will write a companion `HCTree.cpp` implementation file that implements the interface specified in `HCTree.hpp`, and you will then use it in your `compress` and `uncompress` programs. Note that, when you implement the `HCTree` class, you will want to use the `HCNode` class we have provided in `Helper.hpp` and `Helper.cpp`.

As you implement Huffman's algorithm, you will find it convenient to use multiple data structures. For example, a priority queue will assist in building the Huffman Tree. Feel free to utilize other beneficial data structures. However, you should use good object-oriented design in your solution. For example, since a Huffman code tree will be used by both your `compress` and `uncompress` programs, it makes sense to encapsulate its functionality inside a single class accessible by both programs. With a good design, the `main` functions in the `compress` and `uncompress` programs will be quite simple: they will create objects of other classes and call their methods to do the necessary work.

Be sure to refer to the "Honey, I Shrunk the File" section of the *Data Structures* Cogniterra text for more information about Huffman's algorithm.

Priority Queues in C++

A C++ `priority_queue` is a generic container that can hold any type, including `HCNode*` (wink wink). By default, a `priority_queue<T>` will use the `<` operator defined for objects of type `T`. Specifically, if `a < b`, then `b` is taken to have a **higher priority** than `a` (i.e., by default, it functions as a **max**-heap).

However, in the Huffman Tree building algorithm, we want to select symbols with ***lower* frequencies first** (i.e., we want to use a **min**-heap). Thus, we need to tell the `priority_queue` how to compare `HCNode*` objects by defining a custom "comparison class" that will dereference the pointers and compare the objects they point to. This has been done for you via the `HCNodePtrComp` class in `Helper.hpp`. You can use it to create a `priority_queue` as follows:

```
priority_queue<HCNode*, vector<HCNode*>, HCNodePtrComp> pq;
```

- The first argument of the template (`HCNode*`) tells the `priority_queue` that it will be storing `HCNode*` objects
- The second argument of the template (`vector<HCNode*>`) tells the `priority_queue` to use a `vector<HCNode*>` container behind-the-scenes to store its elements
- The third argument of the template (`HCNodePtrComp`) tells the `priority_queue` to use our custom `HCNodePtrComp` class when comparing `HCNode*` objects to determine priorities

Be sure to refer to the `priority_queue` section of the C++ documentation for more information about how to use a `priority_queue`.

Compressed File Header

In the "Suggested Control Flow" section, you will see references to a header that should be stored by the `compress` program and later retrieved by the `uncompress` program. Both the `compress` and `uncompress` programs need to construct the Huffman Tree before they can successfully encode and decode information, respectively. The `compress` program has access to the original file, so it can build the tree by first deciphering the symbol counts. However, the `uncompress` program only has access to the compressed file, not the original file, so it has to use some other information to build the tree. The information needed for the `uncompress` program to build the Huffman Tree needs to be stored in the header of the compressed file. In other words, the header information should be sufficient to reconstruct the tree.

Efficient Header Design

One extremely *inefficient* strategy for designing the header (which is what is used in the reference solution) is to simply store 256 integers (in 4-byte chunks as `unsigned int` objects) at the beginning of the compressed file (i.e., represent the symbol counts as the header), but note that this is not very efficient and is guaranteed to use up 256*4 = 1024 bytes! In order to receive full points, you must BEAT our reference solution by coming up with a more efficient way to represent this header!

However, we **strongly** encourage you to implement the naïve (256 `unsigned int` objects) approach first, and do not attempt to reduce the size of the header until you've gotten your `compress` and `uncompress` to work correctly for the provided inputs.

One way to reduce the size of the header is to take a frequency approach, but to also think about how many bytes you *really* need to store each integer. Answering this question relates to the maximum size of the files that you are required to encode: your `compress` program must work for input files up to 10 MB in size, so a particular byte value may occur up to ~10 million times in the file. This fact should help you determine the minimum number of bytes required to represent each frequency count in your header.

Alternative approaches may use arrays to represent the structure of the tree itself in the header. With some cleverness, it is possible to optimize the header size to about 10*M bits, where M is the number of distinct byte values that actually appear in the input file. This write-up may be helpful to learn about how to "serialize" the tree structure itself.

How to Debug

As you work on this Project, you will almost certainly run into bugs you will need to find and fix. Because of the bitwise nature of this Project, debugging might be more challenging than usual. Here, we will provide some general tips for debugging that may prove useful.

Printing a Huffman Tree

As you implement the Huffman Tree components of `compress` and `uncompress`, you will want to verify that the Huffman Tree you built is correct by working through the exact same example by hand and manually comparing the tree your code builds against the tree you manually build by hand. To compare the tree your code builds against the tree you built yourself, you will want to somehow view the Huffman Tree in your code. We suggest doing this in one of the following two ways:

- Write a helper function that traverses an `HCTree` object and outputs the edges in some human-readable format (e.g. as an edge list you can then draw out by hand)
- Use `gdb` to set a breakpoint just after the `HCTree` object is built, and use `gdb` commands to print the entire tree (e.g. as an edge list you can then draw out by hand)

Viewing a Binary File as Bits

As you implement `compress`, you will likely want to try compressing a small example file that you can trace by hand and then manually look at the compressed output file to verify that it looks like what you would expect. However, if you try to open the binary file in `vim` or similar, you will likely see garbage data: the text editor is trying to render each byte as a human-readable character (and is failing to do so).

Instead, you can view the bit representation of a binary file using a hex dump tool. One good choice is `xxd`, which is a command-line hex dump tool that exists on most Linux distributions. Given any arbitrary file (plain-text or binary!), you can call `xxd` with the `-b` flag (for "bits") to view the bit representation of the bytes in the file. For example, if I have a file `combooter.txt` that contains the text `boo` (3 bytes), I can view the binary representation of the file as follows:

```
$ xxd -b combooter.txt
00000000: 01100010 01101111 01101111                             boo
```

If the file you are trying to look at is very large, the output of `xxd` will be extremely long, so to make it easier to read through it, you can pipe the output of `xxd` to `less` to allow you to scroll up and down:

```
$ xxd -b combooter.txt | less
```

```
00000000: 01100010 01101111 01101111 00001010 01100010 01101111  boo.bo
00000006: 01101111 00001010 01100010 01101111 01101111 00001010  o.boo.
0000000c: 01100010 01101111 01101111 00001010 01100010 01101111  boo.bo
00000012: 01101111 00001010 01100010 01101111 01101111 00001010  o.boo.
00000018: 01100010 01101111 01101111 00001010 01100010 01101111  boo.bo
0000001e: 01101111 00001010 01100010 01101111 01101111 00001010  o.boo.
00000024: 01100010 01101111 01101111 00001010 01100010 01101111  boo.bo
0000002a: 01101111 00001010 01100010 01101111 01101111 00001010  o.boo.
00000030: 01100010 01101111 01101111 00001010 01100010 01101111  boo.bo
00000036: 01101111 00001010 01100010 01101111 01101111 00001010  o.boo.
0000003c: 01100010 01101111 01101111 00001010 01100010 01101111  boo.bo
:
```

The hexadecimal numbers to the left (before the colon) represent the file offset (i.e., 0-based index) of the first byte on the line. Then, we see a series of bytes starting at that offset in the middle. Last, to the right, we see an ASCII representation of each of those bytes. For example, the first line in the `xxd` output above is saying that, starting at offset `00000000` (aka index 0) of the file, the first 6 bytes are `01100010`, `01101111`, `01101111`, `00001010`, `01100010`, and `01101111`, which can be interpreted in ASCII as `boo.bo`.

Speeding Up Code

If your code is taking too long to run, you will want to investigate *which parts* of your code are slow so you can speed up those specific parts. On pretty much any Linux platform, you can use `gprof` (**G**NU **Prof**iler) to perform "per-function profiling". There are more complicated profiling tools that exist, so if you feel comfortable learning how to set up / use one of those, feel free, but `gprof` works for me. The basic workflow is as follows:

```
# Step 1: Compile your program with the -pg flag (we provide a Makefile target for convenience)
make gprof

# Step 2: Run your program (it will create a file gmon.out)
./MyProgram

# Step 3: Run gprof to profile (I pipe to less -S to make it easier to read)
gprof MyProgram | less -S
```

The `gprof` output tells you what percentage of your runtime was spent on each function that was called in your code. Many times, the runtime will be some built-in C++ STL function, so you may think "Well, that's not my code, so that's not something I can speed up", but that would be incorrect! If a built-in C++ STL function is taking a bunch of runtime, it's because *you're calling* that function :-) So try to find where in your code you're calling it a bunch of times, and try to refactor that part of your code!

Here's a lengthier tutorial about `gprof`:

https://www.thegeekstuff.com/2012/08/gprof-tutorial/

Common Issues

Plain-Text Files Ending with Newline Characters

Try using a text editor like `vim` to create a text file `niema.txt` that contains only the string `Niema` in it, and save the file. What is the file size? Intuitively, you might think that the size is 5 bytes (each character takes exactly 1 byte), but try running `ls -l`, or `du -sb` to get the file size in bytes, and the results might surprise you:

```
$ du -sb niema.txt
6       niema.txt
```

As can be seen, the file is 6 bytes, not 5! What's going on? Let's look at the file using `xxd`:

```
$ xxd -b niema.txt
00000000: 01001110 01101001 01100101 01101101 01100001 00001010  Niema.
```

This shows us the following bytes:

```
01001110 =  78 = N
01101001 = 105 = i
01100101 = 101 = e
01101101 = 109 = m
01100001 =  97 = a
00001010 =  10 = NEWLINE (\n)
```

We see that, even though we didn't explicitly add a newline character (`\n`) to the end of our file, `vim` went ahead and added one for us. The reason why `vim` and most text editors do this by default is that, according to the POSIX standard, a "line" of plain-text is defined as "a sequence of zero or more non-newline characters plus a terminating newline character."

In your final implementation, you will need to support *all* possible bytes, including the newline character, but as you begin the Project, you may want to only play around with visible human-readable symbols. If you want to create plain-text files without a newline character at the end, you will want to use the `echo` command with the `-n` flag (which tells `echo` not to add a newline character at the very end), and redirect the output to a file using `>`:

```
$ echo -n "Niema" > niema_no_newline.txt
$ du -sb niema_no_newline.txt
5       niema_no_newline.txt
$ xxd -b niema_no_newline.txt
00000000: 01001110 01101001 01100101 01101101 01100001           Niema
```

Unicode Characters

In general, we typically think of a single character in a plain-text file as a `char`, which is 1 byte (8 bits), meaning there are 2^8 = 256 possible characters. However, let's imagine creating a plain-text file containing the string `Ňĩēѫą`:

```
$ echo -n "Ňĩēѫą" > niema_fancy.txt
```

You may expect this file to be exactly 5 bytes, as there are exactly 5 characters, right? However, manually checking the file size may surprise us:

```
$ du -sb niema_fancy.txt
10      niema_fancy.txt
```

Why is our file 10 bytes instead of 5? The issue is that we are using symbols from the extended Unicode alphabet: basically, a single byte can only represent 256 possible symbols, so under the Unicode encoding, a single character in the extended alphabets can be represented using more than 1 byte. In this specific example, each of our 5 symbols (`Ňĩēѫą`) is represented using 2 bytes, which is why the file size is 10 bytes.

Just as a heads-up, you don't need to (and shouldn't try to) handle the multi-byte extended Unicode symbols as single symbols: if your `compress` and `uncompress` tools simply treat every file they encounter as files over the alphabet of all 256 possible bytes, they will be able to handle any arbitrary possible file completely fine.

No Space Left On Device

This message means that your virtual environment has run out of disk space. This typically implies that you've accidentally created a massive file that's filled up your disk space. One common cause (but not necessarily the only cause) is that a loop in your code within which you're writing to disk is running too long (e.g. potentially an infinite loop). You will need to find out which file(s) are too large and delete them using the `rm` command.

You find out which files are largest using the `du` command (`-h` is for human-readable file size units, e.g. `M` for megabyte, and `-a` is to view *all* files, including hidden files):

```
$ du -h -a *
```

You can also use the `ls` command to list the files in any given directory (`-h` is again for human-readable, and `-a` is again to view all files, including hidden files):

```
$ ls -h -a *
```

Valgrind Reports Memory Leak with `make gprof`

If your code was timing out because the runtime was too long, you likely investigated the runtime of the parts of your code using `gprof` (which is excellent!), and in doing so, you likely compiled your code using `make gprof`, which adds the `-pg` compilation flag to the compilation command (the `-pg` compilation flag is required in order to use `gprof`).

However, for reasons out-of-scope of this class, Valgrind doesn't play nicely with the `-pg` flag, and it will (potentially) incorrectly report a memory leak even if your code doesn't actually have a memory leak. Because of this, you will want to remove the `-pg` flag from your compilation command before checking for memory leaks (and before submitting your code, as the grader will be checking for memory leaks). For more information about Valgrind and `-pg`, see this post.

Code Works on Example Datasets but Fails Grader

The small example datasets we provide are by no means exhaustive, nor are they intended to be. If your code seems to be working correctly on all of the example datasets we provide but fails the grader, that means that there is a bug in your code that needs to be fixed, and the only way to find the bug is to **design your own test datasets** to try to cover any corner cases you can think of in order to trigger the bug. The ability to design appropriate test datasets is one of the intended learning outcomes of this assignment.

Project 1: Huffman Coding

Did you already read through the ***entire*** write-up before coming here to code? We know it's long, but we promise that you'll be way more ready to start coding once you've read everything and mapped out how you want to tackle the project. If you skip any part of the write-up, you'll make this poor kitty sad:

Figure: *The poor kitty you'll make sad if you start coding without reading the entire write-up*