Functionality

As it stands, *Colden* is fully functional. It can successfully compress and expand files up to 16 megabytes big (that's as far as it's been tested, but it most likely can compress files up to around 2^{29} bits. Past there, probably not, due to a limitation in the way bits are read). Given a malformed input file, *Colden* aborts on irrecoverable errors (such as no marker denoting the header from the body, or the absence of the magic number at the start of the file) but continues when it can (for example, when the file claims it has more compressed bytes in it than found, in which case *Colden* will output what it can find). If there is an error during either compression or expansion, the program will log it in the logging box, letting the user know what went wrong and at what stage. If the user tries to run compression without selecting files, the program will not do anything and instead will tell the user to select files. Overall, *Colden* is resistant to malformed input of any sort and is transparent to the user in its operations. In addition, the program fulfills its basic purpose: to compress and expand files.

As an addon to its functionality of compression and expansion, *Colden* can also output a visualization in pdf form of its internal Huffman tree in case one wants to inspect it. As it is too complicated to embed a pdf library into *Colden*, it instead outputs a file in GraphViz dot format that is transformed by the `dot` program into a pdf. While it does rely on an external program, `dot`, to do the actual production of the pdf, this program is relatively common and is free online. If the program isn't installed, the logs will warn the user and tell them to install it.

As well as being technically functional, *Colden* is also visually functional. The GUI is clean and easy to use and works exactly how you would expect. You can select files as input and output, see the logs of the program as it compresses and expands, and clear selected files. This is extremely important for any GUI program.

All in all, *Colden* is an exceptionally functional program. It completes its basic mission, while augmenting its functionality with useful tools and allowing easy access to all functionality through a clean GUI.

Design

The design of *Colden* is documented in the Program Specification document and is made specifically to work well. Complicated logic, such as handling individual bits in an array of bytes, is outsourced to a class made specifically for that purpose: `BitArray`. The Huffman tree data structure is contained within its own class, exposing functionality and concentrating all of its logic into a single file, preventing it from being spread across the actual consumers of various functionalities. Code is not repeated: instead of having an Expand class and a Compress class that essentially render the same GUI and repeating all of that code, both of these classes are subclasses of ColdenTab and only override the functionality they need to. This way, when the GUI is updated, it only needs to be updated in one spot to affect all relevant views.

All in all, these are only examples of a superb design. Code isn't repeated when it can be reused, and semantically separate functionalities are separated into classes. Utility classes, like `Artifact` and `BitArray`, handle speciality logic all in one place, making sure that consumers of that logic don't need to duplicate it across multiple locations. As a whole, the basic program logic is wholly contained within the `doCompression()` function of the subclasses of `ColdenTab`. This allows the basic program operation to be easily understood. When they need to be, details are abstracted away. And the hierarchical ownership structure of the classes—the GUI owns trees, data, and files, but not the other way around—allows side effects to be minimized and directed through return values, allowing for easy reasoning about what the code does. All of these design decisions make the code easy to understand, easy to follow, and easy to reason about. These are the hallmarks of good design.

Creativity

While we did not create the Huffman algorithm, and it has been implemented many, many times before, we were creative in the way we implemented various parts of the file. For example, there exists no BitArray in the Java standard library, so we—creatively—thought of one and designed it. We also went about making the Huffman tree algorithm to be as fast as possible, cleverly

thinking of optimizations over the original design. If you wish to, you can see the evolution of our design at https://github.com/cgm616/compression. In addition, I have never seen a compression tool that allows you to output a visualization of the internal data structure, as ours does. This is hugely useful as a teaching tool, and is really fun to play with.

Sophistication

Our project is very sophisticated, as a result of the algorithms we use. The Huffman tree algorithm is easy to understand, but difficult to implement correctly and robustly. Doing so was the bulk of the work on this project. In addition, speeding up the algorithm was difficult. To do this, we used a Java flamegraph profiler to see which functions took up most of the execution time. This allowed us to speed up the application immensely with some careful, tooling-driven optimizations. That is sophisticated. Not to mention that we created two new data structures to do the heavy lifting of our program: Huffman and BitArray. These two structures use careful and sophisticated logic to perform the operations we need. We also created a sophisticated error handling mechanism, where malformed input or use of the program cannot cause it to crash. We also report errors to the user, bubbling up exceptions and catching them to present user-readable and understandable information for troubleshooting. Nothing is more irritating than when a program you're using crashes with no error message, just a crash log, because that log isn't easily searchable on google. We present easy to search and easy to understand error messages, even for internal errors.

Broadness

We ticked boxes 1 through 6.

1: We use JavaFX, a newer graphics library that is easier to use, understand, and reason about, to drive our GUI. JavaFX is a better choice than Swing for complex applications, and it's easier to use out of the box. The builtin functionality and power of JavaFX allowed us to easily build a GUI that looks and feels good.

2: We use subclassing to prevent code duplication between the Expand and Compress tab by making both of the those classes subclasses of ColdenTab. This has already been discussed in this document and in the specification, but this reduces the work required in building the GUI by a huge amount. Now, changes in one class (ColdenTab) affect all relevant views, instead of requiring changes across a variety of classes. In addition, common functionality is abstracted away.

3: While we did not create an interface, I believe that we should still get credit for this box. We create a custom Node class for use in the Huffman tree, and at one point while building the tree use a PriorityQueue to order the Nodes. In order to do this, we had to implement Comparable<Node> on the Node class. While we didn't create our own interface (there was no need to for this particular program, and making a contrived reason to do so would be against the spirit of coding in the most readable way possible), we still relied on an implementation of one for one of the most important parts of our algorithm: building the Huffman tree. In fact, it becomes radically more difficult to build a Huffman tree without the use of a PriorityQueue.

4: We define two new data structure: Huffman, for holding the tree and its data; and BitArray, for holding arrays of individually indexed bits. Both of these are useful. Obviously, Huffman is incredibly important to our program: it holds the vast majority of the actual application logic used to compress and expand files. In addition, working with compressed data means working with information that is hopefully much smaller than the byte level. This means that we need an easy way to access bits. The BitArray class makes this possible, and takes less memory than an ArrayList of Booleans.

5: We also use built-in data structures, like PriorityQueue, HashMap, and ArrayList. The PriorityQueue is a big part of the algorithm used to build the Huffman tree, and is extremely important there. Because the tree is balanced, we need to combine nodes while keeping them ordered to obtain a best-case balanced tree. The hashmap is also important, because it allows us to flatten the tree into a simple map during compression, radically speeding up the compression algorithm. Instead of walking a tree for each byte to compress, it's a simple map lookup.

6: File input and output is the basic functionality of this program, so it's fitting that we used it. I think it's importance is relatively self-evident: without input and output, how could we

compress and expand files? In addition, we use file input and output to create the graph file when that option is checked.

Code Quality

We use the JavaDoc commenting scheme on classes and methods for consistency, documenting return values, parameters, and exceptions the method may throw. This allows the code base to be easily understood, especially with smart IDEs that can report the JavaDoc information when code is hovered over.

On a more fine-grained level, every line of code in the project is documented. Some lines are documented in groups because individual comments don't make sense, but in general the code is explained and justified by comments that exist alongside it. This allows the code to be easily read.

In terms of code quality, we wanted to make sure that everything worked as expected. To do this, we have a number of test cases in the tests/ directory that we use to manually run the program during testing. All of these files can be successfully compressed and expanded with the program. In addition, we created the Tester class, which tests the BitArray, Artifact, and Huffman classes when run. While this doesn't have as many test cases as we wanted, those are covered by the files in the tests/ directory.