Author: Brandon John-Freso
Task: Facebook Data Compression Algorithm

Brief overview of Files/Classes:

- **Compressor.kt -** This is the main compressor class that coordinates the rest of the process. It can be injected with configurable params (CompressionParams) that will cascade throughout the other classes and optimize performance and output of various facets of compression. The algorithm used for searching the buffers is also injectable here.
- **CompressorTest -** A suite of tests that check for a few happy trails in the Compressor. Seeking mainly to assert that files do not become corrupted throughout the process and that files size is reduced for our sample files.
- **Search -** The SearchContract sets up an interface that allowed me to play with various algorithms, subbing them in here and there. I have set the compressor tests and driver class up to use my Sliding Window technique by default because it proved to be the fastest given the level of effort required to code a truly optimal search for this sort of compression. I have included a Knuth-Morris-Pratt implementation of a search as well, though performance is slower.
    - **SlidingWindow -** The main search algorithm.
    - **KMP -** Included because academically it was an interesting challenge.
- **Models**
    - **Block -** An intermediary format that the compressor works in. There is, of course, a tradeoff converting to and from binary to an intermediary format, I have done so here to allow for optimal readability of the code. Having the intermediary data representation also allows for different types of output to be experimented with.
    - **CompressionParams -** Compression parameters used by many classed (IO, Search, Compressor...) involved in our compression process. The defaults are set to what I found to be the optimal settings (window and code length size) for most files that still works within the bounds of our original spec.
    - **FileConfig -** Convenience data class used to set up file system configurations for the IO streams in our application.
    - **InOutStreamPair -** Creates a pair of In/Out Streams given paths. A convenience data class.
    - **MatchResult -** A data class for holding referential data to other bytes used by the compressor itself as well as the search algorithms.
    -
- **Lib**
    - **BinaryIn/Out -** Using a small lib here for writing directly out to binary with Kotlin on a bit level.
- **IO**
    - **CompressedIOStream -** Compressed format contract for writing data out and reading it in using our compressed format of references and literals. This

    interface can be implemented by different IO streams so that one could for instance write a stream that just outputs to a log or standard out. In practice for the final product, I wrote a CompressedIOStream for reading and writing of our binary files.

  - **BinaryCompression Input/Output -** The CompressedIOStream class that I created for writing to binary classed that implement our particular compressed format.
- **Samples**
  - **Pristine -** Any files you want compressed should be put in this folder. Try not to remove the "emptyfile" as it is specifically needed for one of the unit tests. I have pre-included files from the Calgary Corpus that represent a range of format patterns and types. Running the test suite will convert these all over automatically.
  - **Compressed -** The files in the pristine folder will be compressed and stored here automatically with a ".compressed" extension on them. The is also where the test suite will pull files from for decompression.
  - **Uncompressed -** Once decompressed, the files from Compressed will end up here. It is of note that all of the paths can be configured by updating the FileConfig paths found in the Main or CompressorTest classes.

Notes:

What type of compression are we using?
From the spec of the format that is given in the assignment, the mechanism of compression described seems to belong to the LZ family of compression algorithms, wherein repeated code sequences are searched for within a sliding window of the file and replaced by a reference to the original location.

What language is the compressor written in?
I set about to write my version of the compressor using Kotlin, mostly because very few compressors are written in the language and I thought it posed a unique opportunity to play with some of the APIs.

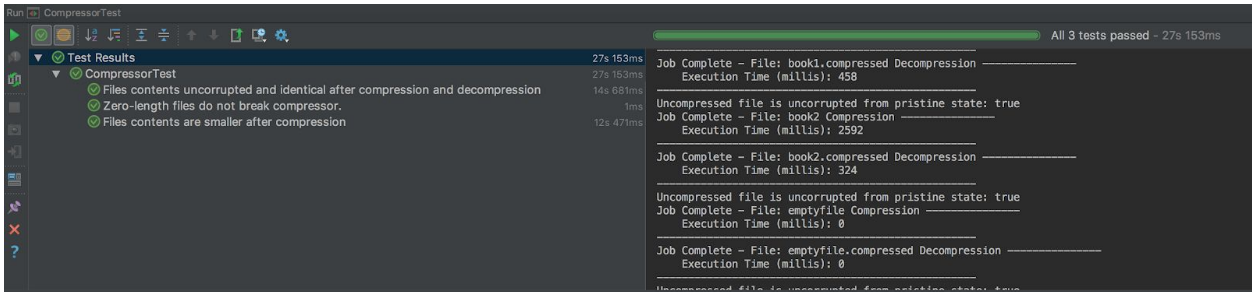
What was my main goal in writing the application?
My main goal was to write a compressor application that is architected in a sane and easily configurable way rather than simply shoving all of the code into the fewest amount of files possible. I had a really great time piecing out the logical partitions for my classes and methods.

How optimal is the compressor?
The assignment FAQ states that "we didn't specify that the compression has to be optimal, because we want to see a decision made about making the tradeoff between

optimality and a reasonable amount of work.” Given the limited amount of time I had to work with I erred on a getting a well-constructed application out that weighs more heavily on the optimal compression speed than it does on the extent of compression. In the end, my compressor on average produces a 47% reduction in file size converting most of the individual files of the corpus over in a few 100ms

What was used for test cases?

I knew I would need to have a solid reference of sample files to compress, as compression efficacy varies greatly with the content of the files. With a bit of research, I found the Calgary Corpus (http://www.data-compression.info/Corpora/CalgaryCorpus/) “The Calgary Corpus is the most referenced corpus in the data compression field especially for text compression and is the de facto standard for lossless compression evaluation.” Given that the intent here is to write a lossless compression algorithm, the corpus seems fitting. The test suite is running using JUnit5.



```
▼ Test Results                                                                              27s 153ms
  ▼ CompressorTest                                                                          27s 153ms
      Files contents uncorrupted and identical after compression and decompression          14s 681ms
      Zero-length files do not break compressor.                                                  1ms
      Files contents are smaller after compression                                           12s 471ms
```

```
--------------------------------------------------------------
Job Complete - File: book1.compressed Decompression ------------------
    Execution Time (millis): 458
--------------------------------------------------------------
Uncompressed file is uncorrupted from pristine state: true
Job Complete - File: book2 Compression ------------------
    Execution Time (millis): 2592
--------------------------------------------------------------
Job Complete - File: book2.compressed Decompression ------------------
    Execution Time (millis): 324
--------------------------------------------------------------
Uncompressed file is uncorrupted from pristine state: true
Job Complete - File: emptyfile Compression ------------------
    Execution Time (millis): 0
--------------------------------------------------------------
Job Complete - File: emptyfile.compressed Decompression ------------------
    Execution Time (millis): 0
--------------------------------------------------------------
Uncompressed file is uncorrupted from pristine state: true
```

What are the constraints of the format and the compressor?

Using only 16 bits to store the offset and 6 bits to store the match length constrains the compressor to some upper bounds. But playing around with different combinations that fell within the allowable ranges allowed me to come to the best speed/compression ratio that I eventually configured into the default parameters of the compressor. This was mostly achieved by reducing the window (lookback buffer) size which sped up the match search portion of the algorithm. The compressor itself is naturally limited in speed by the shortcomings of the language and framework, but still does a pretty good job! With a fair bit more time and work, there are certain optimizations that can be made to the search algorithms, and using ring buffers would likely go a long way.

How does it run?

The Kotlin program was written using IntelliJ, but given that it all compiles down to the same thing, will run on the JVM. For the sake of ease though, I suggest running it within IntelliJ. The Main.kt class will compress all pristine files and uncompress them. The CompressorTest.kt file will run the test suite written for the compressor.