**Building:**

We took advantage of the GNU Make tool chain for our project. Running “make” or “make –f Makefile” in the project directory will build two targets: LZ and EXPAND. Alternatively, you could run make LZ and make EXPAND separately.

**Data Structures:**

We decided too implement this using a hash table, specifically std::unordered\_map. The reason we choose too use this data structure was because we could have constant find and insertion times, and those were the only two operations we ended up needing to do our compression. The major disadvantage of this data structure was the space and in worst case we have linear lookup size equal to the container size.

The approach we took was too convert the input from a character array too a string of bits. The advantage of this was we could string operations such as substring and string length with still maintaining constant accesses too the string. For each character in the current window, then for some look ahead amount descending too 2 bytes, we attempt too find a matching string in the map. If we do find it, then we add a character double. If we don’t find a match, then we insert the current string into the map, including all possible permutations of the string.

Before we write the tuplets out to stdout, we also combined character references that are next too each other into one tuplet, increasing the length. These combination tuplets we limit too be of size . After we have done this consolidation step, we iterate over all the tuplets, and encode them properly according too the instructions.

Decode was a rather simple implementation. We first go over the input string and build recover out tuplets, then for each tuplet, we either append the new addition or we do a repetition of existing characters based off the information from the tuplet.

**Analysis:**

Compress and decompress have two different complexities. Compression first has too iterate over each window, W. In the worst case, over each window we end up doing the entire look ahead amount, meaning that we always find unique bit sequences. This would end up having a complexity of time. However, we find that in the average case we have roughly had 50% repetition that appear exponentially distributed. This means that we get out of the method with time with a hash table of space.

Decompression will have a better case when we have just have new strings since string addition is roughly linear up too the length of the total content. In that case, we have complexity. In the case where there are mostly string references, we have to do an additional iteration of length offset. Offset in max can be F, arriving at time.