Assignment 5
Chris Lee

Edited code in DE10B.java:

```
void deBW(){
 for (int i = 0; i < length; i++){
   int j = i - 1; for (; j >= 0; j--)
     if (L[i] < F[j]) F[j + 1] = F[j];
     else break;
   F[j + 1] = L[i];
 }
 int j = 0;
 for (int i = 0; i < length; i++){
  if (i > 0 && F[i] > F[i - 1]) j = 0;
  for (; j < length; j++) if (L[j] == F[i]) break;
  T[i] = j++;
 }
 // Now we have I, L, F, and T
 // Your code here for printing the decoded block using System.out.write().
 // Write one byte a time
   int current_value = I;
 System.out.write(F[I]);

   for (int i = 0; i < length; i++) {
      current_value = T[current_value];
      System.out.write(F[current_value]);
   }
   System.out.flush();
}
```

Text created from running "/home/chris/.jdks/openjdk-21.0.1/bin/java -javaagent:/home/chris/Documents/intellij/lib/idea_rt.jar=39625:/home/chris/Documents/intellij/bin -Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath /home/chris/projects/data-encoding/assignment-5/out/production/assignment-5 DE10B < DE10test1 > original**":**

```
The most widely used data compression algorithms are based on the sequential
data compressors of Lempel and Ziv. Statistical modelling techniques may
produce superior compression, but are significantly slower.
In this paper, we present a technique that achieves compression within a percent
or so of that achieved by statisticalmodelling techniques, but at speeds comparable
to those of algorithms based on Lempel and Ziv's.
Our algorithm does not process its input sequentially, but instead processes a
block of text as a single unit. The idea is to apply a reversible transformation to a
block of text to form a new block that contains the same characters, but is easier
to compress by simple compression algorithms. The transformation tends to group
characters together so that the probability of finding a character close to another
instance of the same character is increased substantially. Text of this kind can
easily be compressed with fast locally-adaptive algorithms, such as move-to-front
coding in combination with Huffman or arithmetic coding.
Briefly, our algorithm transforms a string S of N characters by forming the N
rotations (cyclic shifts) of S, sorting them lexicographically, and extracting the last
character of each of the rotations. Astring L is formed fromthese characters, where
the ith character of L is the last character of the ith sorted rotation. In addition to
L, the algorithm computes the index I of the original string S in the sorted list of
rotations. Surprisingly, there is an efficient algorithm to compute the original string
```

S given only L and I .
The sorting operation brings together rotations with the same initial characters. Since the initial characters of the rotations are adjacent to the final characters, consecutive characters in L are adjacent to similar strings in S. If the context of a character is a good predictor for the character, L will be easy to compress with a simple locally-adaptive compression algorithm.
The algorithm described here was discovered by one of the authors (Wheeler) in 1983 while he wasworking at AT&T Bell Laboratories, though it has not previously been published.

To see why this might lead to effective compression, consider the effect on a single letter in a common word in a block of English text. We will use the example of the letter ‘t’ in the word ‘the’, and assume an input string containing many instances of ‘the’.
When the list of rotations of the input is sorted, all the rotations starting with ‘he ’ will sort together; a large proportion of them are likely to end in ‘t’. One region of the string L will therefore contain a disproportionately large number of ‘t’ characters, intermingled with other characters that can proceed ‘he ’ in English, such as space, ‘s’, ‘T’, and ‘S’.

We have described a compression technique that works by applying a reversible transformation to a block of text to make redundancy in the input more accessible to simple coding schemes. Our algorithm is general-purpose, in that it does well on both text and non-text inputs. The transformation uses sorting to group characters together based on their contexts; this technique makes use of the context on only one side of each character.
To achieve good compression, input blocks of several thousand characters are needed. The effectiveness of the algorithm continues to improve with increasing block size at least up to blocks of several million characters.
Our algorithm achieves compression comparable with good statistical modellers, yet is closer in speed to coders based on the algorithms of Lempel and Ziv. Like Lempel and Ziv’s algorithms, our algorithm decompresses faster than it compresses.T