Group 7
CSCI 313
HW1—Question 9: De Bruijn sequences

A De Bruijn sequence B(k,n) is a sequence over an alphabet of size k containing all substrings of length n exactly once. For B(k,n), the length of the sequence is k^n. This code creates a random binary string of length 8, and tests whether it is B(2,3). It then randomly mutates the string until it becomes a De Bruijn sequence. It does this 100 times, timing for each string, and then outputs the sum of all the timings.

This code uses the vector data structure for ease of use, because it is both of variable size and random access. However, because the code could not be modified to generalize for substrings of different length, an array would also function well. The code starts by creating the timing sum and seeding that random number generation. Following that, it enters a for loop to time the process 100 times. Once inside the for loop, the code creates a vector of binary digits of length 8. Calling the provided timing function, it then passes that vector to **createDeBruijn3**, the function that will do the majority of the work.

The function **createDeBruijn3** takes in a vector by reference, allowing it to modify that vector. Inside the function is a simple while loop, indicating that while the vector is not a De Bruijn sequence the following steps should be taken. First, the program iterates through the vector. At each point in the vector, it creates a 5% chance that the number will be changed from a 1 to a 0 or vice versa. It does this by generating a random number modulo 20, and then checking to see if that number is equal to 0. The program uses 0, but any single value between 0 and 19 has a 5% chance. Then it simply uses an if-else statement to change the digit. If it’s a 0, set it equal to 1, and vice versa. However, the entire while loop depends on a second function, **deBruin3**, to determine whether the vector is a De Bruijn sequence.

Inside the function **deBruijn3**, it starts by creating a 2-dimensional array of 8 bit-strings of length 3, covering all possible binary permutations. Additionally, it creates a boolean array of size 8 and initializes the booleans to false. Then, using a for loop to iterate through the vector, the code uses nested if-statements to analyze positions **i**, **i+1**, and **i+2** and switches the corresponding Boolean to true for whatever binary string it corresponds to. Note that the if statements actually use **i+1 modulo v.size()** and **i+2 modulo v.size()**, because De Bruijn sequences are circular, so the last element in the vector must be compared against the first element. Finally, the function uses a for loop to iterate through the list of booleans, and if any remain false, it returns false.

Ultimately, this code, while gratifying to build, was frustratingly difficult to generalize. The generation of bit strings can be done recursively fairly easily (generate 2 strings, a ‘1’ plus all bit strings of length n-1, and a ‘0’ plus all bit strings of length n-1). However, creating a general analysis of a De Bruijn sequence B(2,n) proved to be beyond the scope of this project. Unfortunately, this had some dismaying carry-on effects. Because the analysis is so quick with such small vectors and strings, the timing function for each call of **createDeBruijn3** often rounds down to 0 milliseconds, meaning the sum of the timings is neither accurate nor particularly illuminating. This method of building is also quite inefficient, as one could build such a function for strings of length 4 and probably 5 and 6, but beyond that it becomes so tedious as to be prohibitive. While we are confident that a graceful solution exists, we were unable to build one.