Online version: https://us.edstem.org/courses/77/lessons/1118/

EXERCISE 1: Compression Warm-up

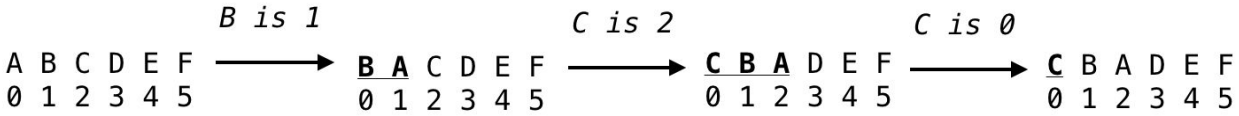
A. The *compression ratio* is defined as $\frac{\text{compressed size}}{\text{original size}}$. Consider a sequence of n characters, 8-bits each, what is the compression ratio achieved by *Huffman* coding in the following cases:

- If there are only two characters?
- the worst case?

B. Move-To-Front is a lossless encoding algorithm that works as follows:

> *Maintain an ordered sequence of the characters in the alphabet by repeatedly reading a character from the input message; printing the position in the sequence in which that character appears; and moving that character to the front of the sequence.*

Example. Input: BCC
Output: 120



Assuming the alphabet is A–Z, where A has code 0, B code 1, etc.:

- Encode "A A A B B B C C C D D D E E E".
- Encode "A E A B E C A D"

C. Move-To-Front encoding is typically used to convert a given text into one where *some characters appear much more frequently than others*. Based on the examples above, when does MoveToFront achieve this goal well?

D. The goal of the **Burrows-Wheeler** lossless transform is to convert a given text into text where *sequences of the same character occur near each other many times*.

How should Move-To-Front, Huffman and Burrows-Wheeler be used together in order to achieve a good compression ratio?

EXERCISE 2: Burrows-Wheeler Transform

A. List the *circular suffixes* of the word "`W E E K E N D`" and then sort them in lexicographical order.

	Original	Sorted
0	W E E K E N D	
1	E E K E N D W	
2		
3		
4		
5		
6		

B. The Burrows-Wheeler transform is the last character of each of the sorted circular suffixes, preceded by the row number in which the original string ends up when considered in sorted order.

What is the Burrows-Wheeler Transform of "`W E E K E N D`"?

C. How much memory (as a function of the number of characters in the String n) is needed to store the circular suffixes? Using Big-Theta notation.

EXERCISE 3: Burrows-Wheeler Inverse-Transform

A. Given only the last character of each of the circular suffixes when considered in sorted order, can we infer the first character in each of these suffixes? Explain your answer.

```
? _ _ _ _ _ N
? _ _ _ _ _ W
? _ _ _ _ _ E
? _ _ _ _ _ K
? _ _ _ _ _ E
? _ _ _ _ _ E
? _ _ _ _ _ D *
```

B. We know from Exercise 2.B that the Burrows-Wheeler transform stores where the original string is. The goal of the inverse-transform is to find the characters of the original string, i.e. the characters between `W` and `D` in row 6.

```
      s[]          t[]
    0| D _ _ _ _ _ N
+-->1| E _ _ _ _ _ W
|   2| E _ _ _ _ _ E
|   3| E _ _ _ _ _ K
|   4| K _ _ _ _ _ E
|   5| N _ _ _ _ _ E
+---6| W _ _ _ _ _ D *
```

Observation. A circular suffix that ends with a W comes directly after a circular suffix that begins with a W in the original (un-sorted) circular suffix array. Look at your answer for 2.A!

Use the following (very slow) algorithm to construct the array `next[]` to keep track of where the next circular suffix is for each of the sorted circular suffixes.

(`n` is the number of circular suffixes and `used[]` is of size `n` and is initialized to `false`)

```
1  for (int i = 0; i < n; i++) {
2      for (int j = 0; j < n; j++) {
3         if (used[j]) continue;
4         if (s[i] == t[j]) {
5             used[j] = true; // disallow reuse of this character
6             next[i] = j;
7             break;
8         }
9      }
10 }
```

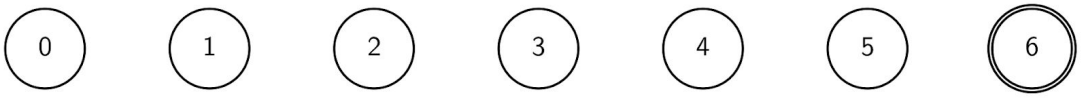
	Original
0	W E E K E N D
1	E E K E N D W
2	E K E N D W E
3	K E N D W E E
4	E N D W E E K
5	N D W E E K E
6	D W E E K E N

	S[] t[]	next[]
0	D - - - - - N	6
1	E - - - - - W	
2	E - - - - - E	
3	E - - - - - K	
4	K - - - - - E	
5	N - - - - - E	
6	W - - - - - D	

C. Trace the array `next[]` starting at row 6 to reconstruct the original string.

Exercise 4: Knuth-Morris-Pratt

Construct the Knuth-Morris-Pratt DFA for the string `PAPAYA` over the alphabet `{A, P, Y}`. Complete the transition diagram and the corresponding DFA table.



	0	1	2	3	4	5
A						
P						
Y						

Exercise 5: Circular Substrings

Implement method `search(String s, String t)`, which returns `true` if the non-empty string ***s*** is a substring of a non-empty **circular** string ***t***. The examples in the table below explain what is meant by a substring of a circular string.

string s	*circular string t*	*substring*
ABBA	BBBBBBABBABBBBB	*yes*
ABBA	BABBBBBABBBBBAB	*yes*
BBAABBAABBAABB	ABBA	*yes*
ABBA	BBBBBBBABABBBB	*no*
BAABAAB	ABBA	*no*

Assuming the binary alphabet `{A, B}`, Your algorithm should take $\Theta(m+n)$ time in the worst case, where ***n*** is the length of ***t*** and ***m*** is the length of ***s***.

Note. You have access to all the data structures and algorithms in algs4. Do not reimplement an algorithm or a data structure if you can use it as is. Simply import it and use it!