EXERCISE 1: Compression Warm-up

A. The *compression ratio* is defined as: *compressed size* / *original size* . Consider a sequence of *N* characters, 8-bits each, what is the compression ratio achieved by *Huffman* coding in:

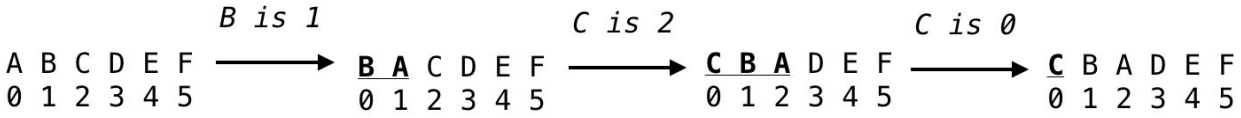
- the best case?

- 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



Move-To-Front encoding is typically used to convert a given text into one where *some characters appear much more frequently than others*.

- Give an example (using the ABCDEF alphabet) where Move-To-Front works best (low variability → high variability).

- Give an example (using the ABCDEF alphabet) where Move-To-Front does not work well (low variability → low variability)

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

Sorted

B. The Burrows-Wheeler transform is the last character of each 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 is needed to store the circular suffixes?

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. The character that follows W, is the first character in the circular suffix that follows row 6 in the original circular suffixes array! Convince the person sitting beside you!

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.

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
       if (used[j]) continue;
       if (s[i] == t[j]) {
           used[j] = true; //disallow reuse of this character
           next[i] = j;
       }
}
```

	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.