The task is to implement a compression/decompression algorithm, say ALG-C

Compression

ALG-C iterates sequentially through the input string and stores any new match into a search buffer. The process of compression can be divided in 3 steps:

1. Find the longest match of a string that starts at the current position with a pattern available in the search buffer.
2. Output a triple (o, l, c) where,
- *o*: offset, represents the number of positions that we would need to move backwards in order to find the start of the matching string.
- *l*: length, represents the length of the match.
- *c*: character, represents the character that is found after the match.
1. Move the cursor l+1 positions to the right.

Let’s get a deeper insight with an example:

```
a b a b c b a b a b a a
```

Initially, our search buffer is empty and we start from the left, where we find an ‘a’. Given that there are not any matching patterns in our search buffer, we output the triple (0, 0, a), since we are not moving backwards (o = 0) and there is not a matching pattern in the search buffer (hence “matching” an empty string: l = 0). After this (non-)match, we find the character ‘a’, so c = a. We move l+1 positions to the right and find ourselves in the second position. We’ll be indicating the position of the cursor using the square brackets [].

```
a [b] a b c b a b a b a a
```

```
ALG-C encoding: (0,0,a)
```

So far, we do not have any pattern in our search buffer that starts with ‘b’. Therefore, the encoding process is similar to the previous step: (0,0,b). At this point, things start to get interesting:

```
a b [a] b c b a b a b a a
```

```
ALG-C encoding: (0,0,a), (0,0,b)
```

We’ve previously found an ‘a’ and even ‘ab’, but not ‘abc’ so we need to move 2 positions to the left (o = 2) and read 2 characters (l = 2). The next character that we can find is a ‘c’, therefore the output triple would be (2,2,c). We move our cursor l+1 positions to the right and find ourselves in the character ‘b’.

```
a b a b c [b] a b a b a a
```

```
ALG-C encoding: (0,0,a), (0,0,b), (2,2,c)
```

We’ve already found a ‘b’, even ‘ba’ and even ‘bab’ but not ‘baba’, so we’ll be moving 4 positions to the left (o = 4) and read 3 characters (l

= 3). The next character that we can find is an ‘a’, and hence the output triple would be (4,3,a). We move our cursor l+1 positions to the right and find ourselves in the character ‘b’.

```
a b a b c b a b a [b] a a
```

```
ALG-C encoding: (0,0,a), (0,0,b), (2,2,c),  (4,3,a)
```

We’re almost done! We’ve already seen a ‘b’ and a ba’, but not a ‘baa’. We need to move 2 positions to the left (o = 2) and read 2 characters (l = 2). After this match, we find an ‘a’, so the last output triple would be (2,2,a).

```
a b a b c b a b a b a a
```

```
ALG-C encoding: (0,0,a), (0,0,b), (2,2,c), (4,3,a),
(2,2,a)
```

You may have noticed that the time complexity in the compression phase does not seem to be too good considering that, in the worst case, we need to go back to the beginning of the input string to find a matching pattern (if any). This means that, in a 0-index position p, we need to move p positions to the left in the worst case. Thinking of an edge case in which every character of the string is different (and hence we do not take advantage of data compression), we would need to process 0 characters for the first position + 1 for the second + 2 for the third… + n-1 for the last position = n(n-1) / 2 = O(n2) time complexity. This is one of the reasons why it is common to predefine a limit on the size of the search buffer, allowing us to reuse the content of up to, for instance, 6 positions to the left of the cursor. The following example may help you illustrate this concept, where the parentheses indicate the content inside the search buffer.

```
a b a b c (b a b a b a) [c] b a a a
```

In this case, we would not find the ‘c’ in the search buffer and, hence, the output triple would be (0,0,c) instead of (7,3,a). However, we would not have to potentially pay the price in every processed character to find a match, in the worst case, at the beginning of the string. All in all, selecting the size of the search buffer becomes a tradeoff between the compression time and the required memory: a small search buffer will generally allow us to complete the compression phase faster, but the resulting encoding will require more memory; on the opposite side, a large search buffer will generally take longer to compress our data, but it will be more effective in terms of memory usage.

It is also common to limit the size of the lookahead buffer, which is the substring that starts at the cursor. Let’s illustrate this concept with an example, where the lookahead buffer is represented between two * symbols.

```
a b a b c (b a b a c a) *[b] a b a* c a a
```

In this case, we have a search buffer of size 6 and a lookahead buffer of size 4. Given that the content of our lookahead buffer is ‘baba’ and it is contained in the search buffer, the ALG-C encoding at this position would be (6,4,c). Note that, in this example, if our lookahead buffer was bigger, the output triple in this position would be different. For instance, if our lookahead buffer also had a size of 6 it would contain the string ‘babaca’, which is fully contained in the search buffer and, hence, the output triple would be (6,6,a).

It is worth mentioning that this algorithm is also known as the “sliding windows” algorithm, given that both the search buffer and the lookahead buffer get updated as the cursor “slides” through the input text.

Decompression

Let’s see how ALG-C uses its encoded form to reproduce the original string. ALG-C is categorized as a lossless data-compression algorithm, which means that we should be able to fully recover the original string. It is also worth mentioning that, in the case of ALG-C, we cannot start decompressing from a random ALG-C triple: instead, we need to start decompressing from the initial triple. The reason is, simply, that the encoded triples are based on the search buffer.

In order to illustrate the decompression process, let’s attempt to decompress the obtained encoding in the previous section, aiming to obtain the original string. Therefore, our encoding in this example would be the following:

```
(0,0,a), (0,0,b), (2,2,c), (4,3,a), (2,2,a)
```

Starting with (0,0,a), we need to move o = 0 positions to the left and read l = 0 characters (that is just an empty string). After that, write c = ‘a’. Hence, the decompressed value of this triple is ‘a’. At this point, our decompression string looks like this:

```
Current string: a
```

```
Remaining ALG-C encoding: (0,0,b), (2,2,c),
(4,3,a), (2,2,a)
```

The next triple that we find is (0,0,b) which means the following: move o = 0 positions to the left and read l = 0 characters (empty string). After that, write c = ‘b’. Hence, the decompressed value of this triple is ‘b’. Our decompression string now looks like this:

```
Current string: a b
```

```
Remaining ALG-C encoding: (2,2,c), (4,3,a), (2,2,a)
```

The next triple that we find is (2,2,c), which is a bit more interesting. Now it means the following: move o = 2 positions to the left and read l = 2 characters (‘ab’). After that, write c = ‘c’. Hence, the decompressed value of this triple is ‘abc’. Our decompression string now looks like this:

```
Current string: a b a b c
```

```
Remaining ALG-C encoding: (4,3,a), (2,2,a)
```

The next triple that we find is (4,3,a), which means the following: move o = 4 positions to the left and read l = 3 characters (‘bab’). After that, write c = ‘a’. Hence, the decompressed value of this triple is ‘baba’. Our decompression string now looks like this:

```
Current string: a b a b c b a b a
```

```
Remaining ALG-C encoding: (2,2,a)
```

The last triple that we find is (2,2,a), which means the following: move o = 2 positions to the left and read l = 2 characters (‘ba’). After that, write c = ‘a’. Hence, the decompressed value of this triple is ‘baa’. Our decompression string now looks like this:

```
Fully decompressed string: a b a b c b a b a b a a
```

If you check the original to-be-compressed string in the previous section, you will see that they are the same!

Tasks:

1. **Design your own encoding format in Rust**
2. **Implement an encode function that encodes arbitrary strings to the encoding format.**
3. **Implement a decode function that decodes encoded data back to string.**
4. **Design a series of unit tests, for smaller data and big data, test both the correctness and efficiency of your code.**