We will write a Python 3 script that reads a single sequence from a FASTA file, computes its Burrows-Wheeler Transform and applies run-length encoding. Then, it inverts both the run-length encoding and BWT using LF mapping. If all functions are implemented correctly, this will produce the original sequence from the FASTA file again.

Use the `rle_bwt.py` script as a skeleton. The script can be executed with a single FASTA file as argument, to which it will apply the implemented functions. You can use the command `$ python3 rle_bwt.py small.fa` to test your code with a short sequence; the `data` folder contains larger files. If all functions are implemented, it will additionally check if the final result matches the original sequence. Alternatively, you can use `import rle_bwt` in a Python console to test your functions individually, e.g. `rle_bwt.string_rotations('banana$')`.

1. Reading the FASTA file

Implement the `read_fasta` function. This function takes a single argument, which is the path to a FASTA file containing one sequence, and returns said sequence as a string. The sequence's metadata can be ignored, and if the file contains multiple sequences only the first one should be returned. If the file is not in the correct format, an error message should be shown instead.

2. Applying the BWT

After reading our sequence, the first step is to compute its BWT. We will do this in two steps: first, we compute the sequence's rotations, then we use the rotations to compute the BWT. Note that the pre-written code appends a unique `$` to the end of our sequence, as this is necessary for the BWT to be performed correctly.

Implement the `string_rotations` and `bwt` functions. The former takes our sequence as input and returns a list containing all its rotation; the latter takes these rotations as input and returns the transformed string.

3. Run-length encoding and decoding

One useful property of the BWT is that it can often be more easily compressed than a raw sequence. We will apply one compression method, called run-length encoding, to the transformed sequence and then invert it.

Implement the `rle` and `rle_invert` functions. The former takes the BWT string and returns its run-length encoded version, whereas the latter takes the encoded string and returns the original. We will represent the RLE as a single string, consisting of single characters followed by numbers indicating the lengths of runs all the way through. For example, the RLE of `‘mississippi’` is `‘m1i1s2i1s2i1p2i1’`.

4. Inverting the BWT

Lastly, we will invert the BWT by applying LF-mapping. For this, we need two auxiliary data structures: the rank vector indicating the rank of each letter in the BWT string, and the F-vector that allows us to map each ranked letter to a row in the Burrows-Wheeler matrix.

First, implement the `compute_rank_vector` function. This should return a list giving the letter rank for each position in the given BWT string: the first occurrence of a letter gets a 0, the second a 1, and so on.

Next, we need a way to represent the F-column and efficiently look up ranked letters in it. Recall that equal letters appear consecutively in the F-column. We can store the index of the first occurrence (rank 0) of each letter, and then find other ranks by simply adding to this index. For example, to find the correct row for letter A with rank 2, we simply look up the first occurrence of A and move down two rows.

Implement the `compute_f_map` function, which takes the BWT string, computes the F-column and returns a dictionary that maps each distinct letter to the index at which it first occurs in the F-column.

With these two functions written, we can finally invert the BWT. Use the LF mapping algorithm to implement the `bwt_invert` function, which takes as arguments the BWT string, the rank vector and the F-column mapping and returns the original untransformed sequence.

Notes on implementation

The automatic grading tool will only use DNA sequences—i.e. strings over the alphabet `{A,C,G,T}`—to check your code. However, we encourage you to implement the functions to work with other alphabets as well. You can assume, for the BWT, that the input string ends with a $ which does not occur at any other positions in the string.

In the RLE functions, we encode run-lengths simply using numerical digits. Of course, there are more memory-efficient ways, but this method is readable for humans and therefore makes it easier to debug your code. To decode, you can use the Python function `isdigit` to differentiate between letters and run-lengths. You can assume that the input string does not contain any digits as part of its alphabet.

Extra assignments

If you are looking for an extra challenge, try your hand at some of the following tasks:

- Implement a search algorithm, that uses the BWT to find occurrences of some smaller pattern in the sequence.
- Implement some additional compression techniques on top of RLE. Can you adapt the search algorithm to work on the compressed sequence?

- The current method of computing the BWT is simple, but relatively slow. Try to implement an *O(n)*-time algorithm, e.g. by first constructing the suffix array.