Homework #5

(5 points) Implement FM-index search using a wavelet tree with both fixed-length and Huffman encoding.

Recall that the FM-index consists of:

- $L = bwt(T)$ – the last column in the matrix of all rotations of T,
- F – the first column; $F[c]$ = position at which lines starting with the character c begin,
- a data structure supporting the computation of $\texttt{rank}_c$ over L
- SA – a sampled suffix array of T.

Download the input file from `https://kubokovac.eu/ds/du/bwt/SH.txt`. This single example will be used throughout the assignment.

Implementing a complete FM-index is a large project, so we will simplify a few aspects:

- You already implemented the suffix array / BWT in the previous assignment. You may use your own solution or one of the reference programs available at:

  `https://kubokovac.eu/ds/src/bwt.zip`

- In practice, FM-indices store only a sample of the suffix array to reduce memory usage. For simplicity, we will store the entire suffix array.
- The given text contains fewer than 32 distinct symbols. For fixed-length encoding, use any encoding where each symbol is represented by exactly 5 bits. For example: `0` → `00000`, ␣ → `00001`, `a` → `00010`, `b` → `00011`, ...
- For an efficient binary rank operation, you may use an existing library, e.g., SDSL (see `https://github.com/simongog/sdsl-lite`), which provides several rank implementations: `rank_support_v`, `rank_support_v5`, `rank_support_rrr`, `rank_support_hyb`,... Python bindings are also available. However, you must implement your own wavelet tree; using a library wavelet tree is not allowed.

Count the frequencies of all symbols in T and construct the corresponding Huffman code. The first column F can then be represented as an array of prefix sums of these frequencies.

Once you have the BWT string, the chosen encoding (fixed-length or Huffman), and a rank implementation, build a wavelet tree over the BWT text and implement $\texttt{rank}_c(L, i)$.

Finally, implement FM-index pattern search as follows. For a pattern $P = p_0 \dots p_{m-1}$, we proceed backwards (from the last character), maintaining an interval of rows $[s_i, e_i)$ corresponding to suffixes starting with $P_{i \dots m-1}$.

- Initialization – rows starting with the last symbol p_{m-1} are:
  $[s_{m-1}, e_{m-1}) \leftarrow [F[p_{m-1}], F[p_{m-1}+1])$
- Step – going from $[s_{i+1}, e_{i+1})$ to $[s_i, e_i)$:
  $[s_i, e_i) \leftarrow [F[p_i] + \texttt{rank}_{p_i}(L, s_{i+1}), F[p_i] + \texttt{rank}_{p_i}(L, e_{i+1}))$
- Termination – after processing all characters, $[s_0, e_0)$ contains all rows starting with pattern P.
- Using the suffix array, convert these row indices to positions in the original text T. Verify that your program correctly identifies the occurrences of P.

Submit:

- Your implementation.
- The Huffman and fixed-length encodings – explicitly list how each symbol is encoded.
- The memory footprint (in bytes) of your wavelet tree for both encodings.
- The search performance – measure search times, for example, on random strings and random substrings of T of lengths $10, 20, 30, \ldots, 100$ (compare that with linear string search in T without FM-index). Report average time per operation. Always specify the units of measurement.