1. The Knuth–Morris–Pratt algorithm differs from the Morris–Pratt algorithm only in the failure function, which can be defined as

\[
\text{fail}_{\text{KMP}}[i] = k, \text{ where } k \text{ is the length of the longest proper border of } P[0..i] \text{ such that } P[k] \neq P[i], \text{ or } -1 \text{ if there is no such border.}
\]

(a) Compute both failure functions for the pattern \text{ananassana}.

\[
\begin{array}{|c|c|c|c|c|}
\hline
i & P[0..i] & \text{MP border} & \text{fail}_{\text{MP}}[i] & \text{KMP border} & \text{fail}_{\text{KMP}}[i] \\
\hline
0 & \varepsilon & -1 & -1 & -1 \\
1 & a & \varepsilon & 0 & \varepsilon & 0 \\
2 & an & \varepsilon & 0 & -1 & -1 \\
3 & ana & a & 1 & \varepsilon & 0 \\
4 & anan & an & 2 & -1 & -1 \\
5 & anana & ana & 3 & ana & 3 \\
6 & ananas & \varepsilon & 0 & \varepsilon & 0 \\
7 & ananass & \varepsilon & 0 & -1 & -1 \\
8 & ananassa & a & 1 & \varepsilon & 0 \\
9 & ananassan & an & 2 & -1 & -1 \\
10 & ananassana & ana & 3 & ana & 3 \\
\hline
\end{array}
\]

(b) Give an example of a text, where some text character is compared three times by the MP algorithm but only once by the KMP algorithm when searching for \text{ananassana}.

**Solution:** In \( T = \text{kananmuna} \), the symbol \text{m} is compared against \text{a} in positions 4, 2 and 0 in \text{ananassana} by the MP algorithm, but only against \text{a} in positions 4 by the KMP algorithm.

2. A pattern matching automaton for a string \( P \) is an automaton that recognizes the language \( \Sigma^*P \), i.e., all strings that end with \( P \). Draw the following types of pattern matching automata for \( P = \text{ananas} \):

(a) MP-automaton (see slide 79 in lecture notes)

**Solution:**

```
1 \Sigma
\downarrow
0 \rightarrow a \rightarrow 1
\rightarrow n \rightarrow 2 \rightarrow a
\rightarrow 3 \rightarrow n \rightarrow 4 \rightarrow a
\rightarrow 5 \rightarrow s \rightarrow 6
```

(b) KMP-automaton

**Solution:**

```
1 \Sigma
\downarrow
0 \rightarrow a \rightarrow 1
\rightarrow n \rightarrow 2 \rightarrow a
\rightarrow 3 \rightarrow n \rightarrow 4 \rightarrow a
\rightarrow 5 \rightarrow s \rightarrow 6
```
3. Let us analyze the average case time complexity of the Horspool algorithm, where the average is taken over all possible patterns of length $m$ and all possible texts of length $n$ for the integer alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ where $\sigma > 1$. This is the same as the expected time complexity when each pattern and text character is chosen independently and randomly from the uniform distribution over $\Sigma$.

(a) Show that the average time spend in the loop on line 7 is $O(1)$.

**Solution:** The probability of a match between $P[i]$ and $T[j + i]$ is $1/\sigma$. Therefore, the probability of performing a comparison in the loop is at most

- 1 for the first comparison
- $1/\sigma$ for the second comparison
- $(1/\sigma)^2$ for the third comparison, etc.

The expected number of comparisons is therefore at most

$$\sum_{k=0}^{\infty} \left(\frac{1}{\sigma}\right)^k = \frac{\sigma}{\sigma - 1} \leq 2$$

where we used the standard sum formula for geometric series.

(b) Show that the probability that the shift is shorter than $\min(m, \sigma/2)$ is at most $1/2$.

**Solution:** Let $\ell = \min(m, \sigma/2)$. If $P[m - \ell..m]$ does not contain the symbol $c = T[j + m - 1]$, then the shift is at least $\ell$. Since $\ell \leq \sigma/2$, $P[m - \ell..m]$ contains at most $\sigma/2$ distinct symbols. Therefore the probability of $P[m - \ell..m]$ containing the random symbol $c$ is at most $1/2$.

(c) Combine the above results to show that the average time complexity is $O(n/\min(m, \sigma))$.

**Solution:** By (b), the average shift is at least $\min(m, \sigma/2)/2$. Therefore, the total number of shifts is $O(n/\min(m, \sigma))$. By (a), the algorithm spends $O(1)$ time before a new shift.
4. Simulate the execution of the BNDM algorithm for the pattern \texttt{anna} and the text \texttt{bananamanna}.

\textbf{Solution:}

\begin{align*}
\begin{array}{cccc}
B[c], c \in \{a, b, m, n\} & D \text{ when scanning } \texttt{bana} \text{ backwards (} j = 0 \text{)} \\
a & \begin{array}{cccc}
b & 0 & 0 & 0 \\
n & 0 & 0 & 1 \\
a & 1 & 0 & 0 \\
\end{array} & a & \begin{array}{cccc}
b & 1 & 1 & 0 \\
n & 1 & 0 & 1 \\
a & 1 & 1 & 0 \\
\end{array} \\
\end{array}
\end{align*}

\begin{align*}
\begin{array}{cccc}
D \text{ when scanning } \texttt{anam} \text{ backwards (} j = 3 \text{)} & D \text{ when scanning } \texttt{anna} \text{ backwards (} j = 7 \text{)} \\
\begin{array}{cccc}
a & 1 & 0 \\
n & 1 & 0 \\
a & 1 & 0 \\
\end{array} & a & \begin{array}{cccc}
b & 1 & 1 & 0 \\
n & 1 & 0 & 1 \\
a & 1 & 0 & 0 \\
\end{array} & \Rightarrow shift = 3 \\
\end{array}
\end{align*}

\begin{align*}
\begin{array}{cccc}
D \text{ when scanning } \texttt{anam} \text{ backwards (} j = 3 \text{)} & D \text{ when scanning } \texttt{anna} \text{ backwards (} j = 7 \text{)} \\
\begin{array}{cccc}
a & 1 & 0 \\
n & 1 & 0 \\
a & 1 & 0 \\
\end{array} & a & \begin{array}{cccc}
b & 1 & 1 & 0 \\
n & 1 & 0 & 1 \\
a & 1 & 0 & 0 \\
\end{array} & \Rightarrow shift = 4 \\
\end{array}
\end{align*}

5. Show how the following (single) exact string matching algorithms can be modified to solve the \textit{multiple exact string matching problem}:

(a) Shift-And

(b) Karp-Rabin

The solution should be more efficient than the trivial one of searching each pattern separately.

\textbf{Solution:}

(a) We use Shift-And for the concatenation of the patterns with two additional changes:

- Replace “+1” with “| $M_{\text{begin}}$”, where $M_{\text{begin}}$ contains 1-bits at the starting positions of each individual pattern. This sets the bits that mark the occurrences of the prefixes of length one.
- Replace “& $2^m - 1$” with “& $M_{\text{end}}$”, where $M_{\text{end}}$ contains 1-bits at the ending positions of each individual pattern. This checks for an occurrence of any individual pattern.

\textbf{Algorithm:} Shift-And for multiple patterns

\textbf{Input:} text $T$, patterns $P_1, P_2, \ldots, P_k$

\textbf{Output:} text positions where an occurrence of $P_i$ ends for some $i$

\textbf{Preprocess:}

\begin{enumerate}
\item for $c \in \Sigma$ do $B[c] \leftarrow 0$
\item $P \leftarrow P_1 P_2 \ldots P_k$ // concatenation
\item for $i \leftarrow 0$ to $|P| - 1$ do $B[P[i]] \leftarrow B[P[i]] + 2^i$
\item // Compute bit masks marking the beginnings and ends of patterns
\item $M_{\text{end}} \leftarrow 0$; $m \leftarrow 0$
\item for $i \leftarrow 1$ to $k - 1$ do
\item $m \leftarrow m + |P_i|$; $M_{\text{end}} \leftarrow M_{\text{end}} | 2^m - 1$
\item $M_{\text{begin}} \leftarrow (M_{\text{end}} <<< 1) | 1$
\end{enumerate}

\textbf{Search:}

\begin{enumerate}
\item $D \leftarrow 0$
\item for $j \leftarrow 0$ to $n - 1$ do
\item $D \leftarrow ((D <<< 1) | M_{\text{begin}}) \& B[T[j]]$
\item if $D \& M_{\text{end}} \neq 0$ then output $j$
\end{enumerate}
The search time is $O(n[m/w])$, where $m$ is the total length of the patterns.
If we wanted to report for each position the list of matching strings, then we would have to check which of the ending position bits are set, which would take $O(k)$ time for each match. This can be improved if we assume there exists a constant-time function which returns the position of the leftmost one-bit in a machine word. Using gcc, this can be implemented for example with the function `builtin_clz`, which counts the number of leading zeroes in a machine word. There exists theoretical methods to do this even without the built in gcc extensions, see e.g. https://www.akalin.com/constant-time-mssb

(b) Assume first that all patterns have the same length $m$. Compute the hash value for all patterns and compare each text factor hash value against all patterns. Naively, this takes $O(nk)$ time. If we store the pattern hash values into a hash table, the time becomes $O(n)$.
Note that we may have to compute a “hash value of a hash value” to reduce the hash values to a smaller range.
If the patterns have different lengths, we can truncate them to the same length for the purpose of computing the hash values. If there are big difference in the lengths, it may be useful to divide the patterns into two or more groups according to their lengths. The worst case still remains bad.