1. [4+4+4 points] Each of the following pairs of concepts are somehow connected. Describe the main connecting factors or commonalities as well as the main separating factors or differences.

(a) Horspool algorithm and BNDM algorithm.
   **Solution.** Both are exact string matching algorithms that start matching the pattern against a text factor from the end to achieve potentially long shift. Horspool bases the shift on the last symbol of the text factor, while BNDM uses the longest suffix of the text factor that is a pattern factor. Both algorithms are fast on average and in the best case but slow in the worst case. Horspool suffers from a small alphabet, BNDM does not.

(b) Edit distance computation and approximate string matching.
   **Solution.** Both are problems involving computing the smallest number of edit operations required to transform one string into either another string (edit distance) or into substrings of another string (approximate string matching). Both can be solved using similar dynamic programming calculation; the main differences are the initialization of the first row in the DP table and the interpretation of the resulting table to produce the output. There are also algorithms and variants that apply only to one problem.

(c) LCA (Lowest Common Ancestor) preprocessing ja RMQ (Range Minimum Query) preprocessing.
   **Solution.** Both preprocess a data structure in linear time so that it supports the query in question in constant time. The data structure is tree for LCA and an array for RMQ. The longest common prefix of two suffixes of a text can be found by LCA on the suffix tree or by RMQ on the LCP array augmenting the suffix array.

A few lines for each part is sufficient.

2. [12 points] A string $A$ is a subsequence of a string $B$ if $A$ can be obtained by deleting characters from $B$. For example, $abc$ is a subsequence of $abadc$ but it is not a subsequence of $acadb$.

Let $P$ be a pattern and $T$ a text. Describe an efficient algorithm for finding the length of the shortest factor of $T$ that contains $P$ as a subsequence. For example, if $P = abc$ and $T = cabadcabddc$, then the answer is 5 as $abc$ is a subsequence of $X = abadc$, and $X$ is shortest of such substrings of $T$. What is the time complexity of your algorithm in terms of the lengths of $P$ and $T$?

**Solution.**

We use dynamic programming. Let $P = P[1..m]$ and $T = T[1..n]$. For $i \in [0..m]$ and $j \in [0..n]$, let $\ell_{ij}$ be the length of the shortest suffix of $T[1..j]$ that contains $P[1..i]$ as a subsequence, or $\infty$ if there is no such suffix. The smallest number on the last row $\ell_{m*}$ is the answer.

Clearly, $\ell_{0j} = 0$ for all $j \in [0..n]$ and $\ell_{i0} = \infty$ for all $i \in [1..m]$. Other values can be computed using the recurrence

$$
\ell_{ij} = \begin{cases} 
\ell_{i-1, j-1} & \text{if } P[i] = T[j] \\
\ell_{i, j-1} + 1 & \text{if } P[i] \neq T[j]
\end{cases}
$$
3. \([4+8 \text{ points}]\)

(a) What is the lcp-comparison technique? Describe the main principles.

**Solution.**

The lcp-comparison technique is a method for comparing strings that can be used for speeding up comparison-based algorithms. The main principles are:

- When two strings are compared, the result is not only the order of the strings (\(<,=,>\)) but also the length of the longest common prefix (lcp). The additional information obtained compensates for the potentially long time needed for a string comparison. Computing the extra information does not increase the comparison time.
- Using results from previous comparisons, it may be possible to determine a lower bound for the lcp value and sometimes even the full result of the comparison. In the former case, the comparison can skip the known common prefix, and in the latter case, the comparison can be completely avoided.

(b) Give two examples of algorithms or data structures that use the lcp-comparison technique. Describe the role of the lcp-comparison technique in the algorithms.

**Solution.**

String mergesort uses lcp comparisons when merging two sorted sequences of strings. The sequences are augmented with the lcp values between adjacent elements providing information from the comparisons performed when sorting the sequences. The algorithm repeatedly compares the heads of the input sequences and moves the smaller one to the end of the output sequence. Let \(B\) and \(B'\) be the two heads and \(A\) the tail of the output. The algorithm knows the result of the comparisons between \(A\) and \(B\) and between \(A\) and \(B'\), and uses these to avoid or speed up the comparison between \(B\) and \(B'\).

String binary search performs \(\log n\) lcp comparisons between the query string \(Q\) and strings in a sorted array of \(n\) strings. Comparing \(Q\) against \(S_{\text{mid}}\) involves two other strings \(S_{\text{left}}\) and \(S_{\text{right}}\) such that \(S_{\text{left}} \leq Q, S_{\text{mid}} \leq S_{\text{right}}\). The algorithm knows the results from comparisons between \(Q, S_{\text{left}}\) and \(S_{\text{right}}\), and possibly also between \(S_{\text{mid}}, S_{\text{left}}\) and \(S_{\text{right}}\), and uses these to avoid or speed up the comparison between \(Q\) and \(S_{\text{mid}}\).

4. \([6+6 \text{ points}]\) Let \(\{a, b\}\) be the alphabet. For any integers \(k \geq 1\) and \(m \geq k\), describe a set of \(2^k\) strings of length \(m\) such that the number of nodes in the (uncompact) trie for the set is

(a) as large as possible

**Solution.**

The trie should branch as much as possible at the top of the trie so that the common prefixes between the strings are minimized. This is achieved with the set

\[
\mathcal{R} = \{Xa^{m-k} \mid X \in \Sigma^k\}
\]

The nodes of the trie represent the prefixes of the strings in \(\mathcal{R}\). Let \(\mathcal{P}_d\) be the set of prefixes of length \(d\). If \(d \leq k\), then \(\mathcal{P}_d = \Sigma^d\), and if \(d \in [k..m]\), then \(\mathcal{P}_d = \{Xa^{d-k} \mid X \in \Sigma^k\} \)
$X \in \Sigma^k \}$. Thus the number of nodes in the trie is

$$
\sum_{d=0}^{m} |P_d| = \sum_{d=0}^{k-1} |P_d| + \sum_{d=k}^{m} |P_d| = \sum_{d=0}^{k-1} 2^d + \sum_{d=k}^{m} 2^k
$$

$$
= (2^k - 1) + (m - k + 1)2^k = (m - k + 2)2^k - 1
$$

(b) as small as possible.

**Solution.**

Now the trie should branch as little as possible at the top so that the common prefixes are maximized. This is achieved with the set

$$
\mathcal{R} = \{a^{m-k}X | X \in \Sigma^k \}
$$

In this case, if $d \leq m - k$, then $P_d = a^d$, and if $d \in [m - k..m]$, then $P_d = \{a^{m-k}\Sigma^{d-m+k}\}$. Thus the number of nodes in the trie is

$$
\sum_{d=0}^{m} |P_d| = \sum_{d=0}^{m-k-1} |P_d| + \sum_{d=m-k}^{m} |P_d| = \sum_{d=0}^{m-k-1} 1 + \sum_{d=m-k}^{m} 2^{d-m-k}
$$

$$
= (m - k) + 2^{k+1} - 1 = 2^{k+1} + m - k - 1
$$

What is the number of nodes in each case? Note that all the strings in the set must be different.

5. **[12 points]** Let $S$ and $T$ be strings over the integer alphabet $[0..\sigma)$. Describe an algorithm that finds the shortest string that occurs in $S$ but does not occur in $T$. The time complexity should be $O(|S| + |T| + \sigma)$.

Build the suffix tree of the concatenation $S$§$T$, i.e., the generalized suffix tree of $S$ and $T$. Mark each leaf according to whether it starts in $S$-part or $T$-part.

Let $v$ be a node satisfying the following conditions:

- The string $V$ represented by $v$ does not contain the symbol $\$.
- $v$ has a child $u$ that has only $S$-type leaves in its subtree.
- The first character $c$ on the edge from $v$ to $u$ is not $\$.

Then the string $Vc$ occurs in $S$ but not in $T$. Furthermore, if $v$ is the highest node satisfying the above conditions, then $Vc$ is the shortest such string, which is what we want to find.

The suffix tree can be constructed in linear time for an integer alphabet using Farach’s algorithm or via suffix array construction. The branching in the internal nodes can be implemented as a linked list. Then, we can perform a linear time depth-first traversal, compute for every node whether it satisfies the above conditions, and select the highest such node.