1. [3+3+3+3 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) (Knuth–)Morris–Pratt algorithm and Shift-and algorithm.

Solution.
Both are exact string matching algorithms that operate like automata in the sense that they read the text sequentially one character at a time, never moving backwards and never skipping characters. With KMP, the automaton is deterministic, while Shift-and simulates a non-deterministic automaton. The time complexities of KMP are $O(m)$ for preprocessing and $O(n)$ for searching on an ordered alphabet in the best, average and worst case. Shift-and is theoretically worse algorithm in that it needs an integer alphabet, uses $O(\sigma + m)$ time for preprocessing and $O(n\lceil m/w \rceil)$ time for searching in the worst case. The average search time for Shift-and is $O(n)$, and it is often faster in practice.

(b) String quicksort and ternary tree.

Solution.
Both operate on a set of strings: string quicksort sorts the set while ternary tree stores the set so that efficient insertions, deletions and searches are supported. Both are based on repeatedly dividing the set into three parts: It is the structure of recursive calls in string quicksort, and the actual tree structure of the ternary tree.

(c) String binary search and backward search.

Solution.
Indexed exact string matching can be done by performing a string binary search on the suffix array of the text or by a backward search on the Burrows-Wheeler transform (BWT) of the text. In both cases, the result is the suffix array range corresponding to the pattern occurrences. String binary search is a more general procedure that operates on any sorted set of strings. Backward search can operate on a compressed representation of the BWT with a size close to the compressed text.

(d) Compact trie and suffix tree.

Solution.
The compact trie is a data structure for storing a set of strings. The suffix tree is a special case of the compact trie for the set of suffixes of a text. The suffix tree can have suffix links, which are not available in general compact tries, though adding failure links to make the compact trie into an Aho-Corasick automaton is somewhat equivalent. The suffix tree can be constructed in linear time, while a generic compact trie construction could require even quadratic time.

A few lines for each part is sufficient.

2. [6+6 points] Let $S[0..n]$ be a string over an alphabet $\Sigma$ of size $\sigma$. A $q$-gram is a string of length $q$. Let $occ[0..\sigma^q]$ be an array that for each $q$-gram $Q$, in lexicographic order, tells how many times $Q$ occurs in $S$. For example, if $\Sigma = \{a, b, c\}$, $q = 2$ and $S = acbbbac$, then $occ = (0, 0, 2, 1, 3, 0, 0, 1, 0)$. Describe algorithms for computing $occ$ given $q$ and $S$:

(a) $O(n + \sigma^q)$ time algorithm for integer alphabet $[0..\sigma)$.
Solution.
We use the Karp–Rabin rolling hash function without modulo operation to map any $q$-gram into its position in the occ array:

1. \( \text{occ} \gets (0, 0, \ldots, 0) \)
2. \( h t \gets 0 \)
3. for \( i \gets 0 \) to \( q - 2 \) do \( h t \leftarrow h t \cdot \sigma + T[i] \)
4. for \( i \gets 0 \) to \( n - q - 1 \) do
   1. \( h t \leftarrow h t \cdot \sigma - T[i] \cdot \sigma^q + T[i + q] \)
5. \( \text{occ}[ht] \leftarrow \text{occ}[ht] + 1 \)
6. return \( \text{occ} \)

The time complexity is clearly \( O(n + \sigma^q) \).

(b) \( O(n \log \sigma + \sigma^q) \) time algorithm for an ordered alphabet.

Solution.
We can use the algorithm of part (a) if we have an order preserving mapping \( \delta : \Sigma \rightarrow [0..\sigma] \): just replace each \( T[i] \) with \( \delta(T[i]) \). Such a mapping can be implemented by a binary search on an array containing all symbols of \( \Sigma \) in order. The additional time due to the binary searches is \( O(n \log \sigma) \) and thus the time complexity of the algorithm is \( O(n \log \sigma + \sigma^q) \).

If \( \Sigma \) is not given explicitly, we can collect all distinct symbols from the text and use that set as \( \Sigma \). This can be done in \( O(n \log \sigma) \) time by representing the set by a balanced binary tree and inserting all symbols of the text into the tree (and discarding duplicates).

In both cases, prove the time complexity.

3. [6+6 points]

(a) Compute the edit distance between strings \texttt{tukholma} and \texttt{stockholm} using the dynamic programming algorithm described on the course.

Solution.

<table>
<thead>
<tr>
<th>d</th>
<th>s</th>
<th>t</th>
<th>o</th>
<th>c</th>
<th>k</th>
<th>h</th>
<th>o</th>
<th>l</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>t</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

The edit distance is 4.
(b) Give all optimal alignments between tukholma and stockholm, i.e., alignments with the same cost as the edit distance.

| INSINNNNND | INISNNNNND |
| -tu-kholma  | -t-ukholma  |
| stockholm–  | stockholm–  |

Solution.

4. [6+6 points] Let $A$, $B$, $B'$ and $C$ be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$.

(a) Prove that $\text{lcp}(B, B') \geq \text{lcp}(A, C)$. You may assume only basic definitions from the course to be known, i.e., do not use any lemmas or theorems from the course.

Solution.

Let $\ell = \text{lcp}(A, C)$ and let $P$ be the common prefix, i.e., $L = A[0..\ell] = C[0..\ell]$. We will first show the following lemma.

Lemma. If $A \leq X \leq C$, then $X[0..\ell] = P$.

Proof. Suppose this is not the case. Let $k < \ell$ be the first position, where $X$ differs from $L$, i.e., $X[0..k] = L[0..k]$ but $X[k] \neq L[k]$ (or $|X| = k$). Now, if $|X| = k$ or $X[k] < L[k] = A[k]$, then $X \prec A$ which is a contradiction. On the other hand, if $X[k] > L[k] = C[k]$, then $X \succ C$ which again contradicts the assumption. Thus we have shown the lemma. \hfill \square

The lemma means that $B[0..\ell] = L = B'[0..\ell]$. Thus $\text{lcp}(B, B') \geq \ell = \text{lcp}(A, C)$.

(b) Describe how the above result can be used to speed up string binary searching.

Solution.

Each step in a string binary search involves four strings, the query string $P$ and three strings $S_{\text{left}}$, $S_{\text{left}}$ and $S_{\text{right}}$ from the sorted array. We know that $S_{\text{left}} \leq P \leq S_{\text{right}}$ and $S_{\text{left}} \leq S_{\text{mid}} \leq S_{\text{right}}$, and we want to compare $P$ to $S_{\text{mid}}$. We obtain a lower bound on $\text{lcp}(P, S_{\text{mid}})$ as follows:

$$\text{lcp}(P, S_{\text{mid}}) \geq \text{lcp}(S_{\text{left}}, S_{\text{right}}) = \min\{\text{lcp}(S_{\text{left}}, P), \text{lcp}(P, S_{\text{right}})\}$$

where the inequality follows from the above result. The equality is another result shown on the lectures and the right-hand side values $\text{lcp}(S_{\text{left}}, P)$ and $\text{lcp}(P, S_{\text{right}})$ are known since we have previously compared $P$ to $S_{\text{left}}$ and $S_{\text{right}}$. The lower bound on $\text{lcp}(P, S_{\text{mid}})$ can speed up the comparison of $P$ and $S_{\text{mid}}$.

5. [12 points] The reverse of the string $A = a_1a_2\ldots a_m$ is the string $A^R = a_m\ldots a_1$. Describe an algorithm that, given two strings $S$ and $T$, finds the shortest string $X$ such that $X$ occurs in $S$ but neither $X$ nor $X^R$ occurs in $T$. The time complexity should be linear on a constant size alphabet.

Solution.

Build the suffix tree of the concatenation $S \$ T \$ T^R$, where $\$ is a unique symbol. Mark each leaf according to whether it starts in $S$-part, $T$-part or $T^R$-part of the concatenation.

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 $T$ or $T^R$. The latter means that $(Vc)^R$ does not occur 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. Using a linear time depth-first traversal, we can compute for every node, whether it satisfies the above conditions.