1. [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) Morris–Pratt algorithm and Crochemore algorithm.

Solution. Both are exact string matching algorithms. Both find the longest prefix of the pattern that occurs at a given position in the text and then shift the pattern forward based on the length of the matching prefix. After the shift Morris–Pratt always skips the overlapping part between the preceding match and the new alignment, while Crochemore does this skip only if a long match is followed by a short shift, i.e., the overlapping part is long. Both run in linear time in the best and the worst case. Both work for the ordered alphabet. Morris–Pratt needs $O(m)$ extra space for a pattern of length $m$, but Crochemore needs only constant extra space.

(b) Shift–And algorithm and Myers’ bitparallel algorithm.

Solution. Both are bitparallel algorithms for string matching, Shift-Or for exact string matching and Myers’ algorithm for approximate string matching. Both have the same search time complexity $O(n\lceil m/w \rceil)$ on an integer alphabet.

(c) LCA (Lowest Common Ancestor) preprocessing and 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. [10 points] Construct the Aho–Corasick automaton for the pattern set \{angel, angry, chapel, gel, michael\}. Simulate the scanning of the text michelangelo with the automaton.

Solution. The automaton is below. The missing failure links point to state 0.

The patterns associated with the states are:
3. \(4+7 \text{ points}\) A \(q\)-gram of a string is its factor of length \(q\). Let \(G_q(A, B)\) denote the number of \(q\)-grams shared by the strings \(A\) and \(B\).

For example, for \(A = \text{varaurat}\) the 2-grams are \(\text{va}, \text{ar}, \text{ra}, \text{au}, \text{ur}, \text{ra}\) and \(\text{at}\), and for \(B = \text{ararat}\) they are \(\text{ar}, \text{ra}, \text{ar}, \text{ra}\) and \(\text{at}\). The shared 2-grams are \(\text{ra}\) twice, \(\text{ar}\) and \(\text{at}\), and thus \(G_2(A, B) = 4\).

(a) Show that if \(ed(A, B) \leq k\), then \(G_q(A, B) \geq |A| - q + 1 - kq\).

Solution.
The number of \(q\)-grams in \(A\) is \(|A| - q + 1\). Each edit operation changes at most \(q\) different \(q\)-grams. Thus after \(k\) edit operations to transform \(A\) into \(B\), there must remain at least \(|A| - q + 1 - kq\) unchanged \(q\)-grams, all of which are shared by \(A\) and \(B\).

(b) Design a filtering algorithm for approximate string matching based on the result of (a)-part.

Solution.
Let \(m\) be the length of the pattern. Find and mark the exact occurrences of all pattern \(q\)-grams in the text using a multiple exact string matching algorithm such as Aho–Corasick. Slide a window of size \(m + k\) (the maximum length of an occurrence) over the text and count the number of marked \(q\)-grams within each window position. If some window has at least \(m - q + 1 - kq\) marked \(q\)-grams, verify the window using, for example, the standard dynamic programming algorithm. For this to work, we have to choose \(q\) so that \(m - q + 1 - kq > 0\), which happens when \(q < (m + 1)/(k + 1)\). Thus we can choose \(q = \lfloor m/(k + 1) \rfloor\).

4. \(4+6 \text{ points}\)

(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 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 $S_{\text{mid}}$.

5. [10 points] Let $S$ and $T$ be strings over an alphabet of constant size. Describe an algorithm that given $S$, $T$ and an integer $k$ finds if there exists a string that occurs exactly $k$ times in $S$ and exactly $k$ times in $T$. The time complexity should be linear.

Solution.

Build the suffix tree of the concatenation $S \$ T \$ \ell$, i.e., the generalized suffix tree of $S$ and $T$. Mark each leaf according to whether it starts in $S$-part or $T$-part. For each internal node, count the number of $S$-type leaves and the number of $T$-type leaves in subtree. This can be done in linear time by depth-first traversal of the tree. If there exists a node, where both numbers are $k$, the algorithm answers “yes”. Otherwise the algorithm answers “no”.