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) Shift–Or 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.

(b) Aho–Corasick algorithm and suffix tree.

**Solution.** Both are based on the trie structure: AC on the basic trie for an arbitrary set of strings, ST on the compact trie for the set of all suffixes of a text. Failure links in AC automaton and suffix links in ST can be defined the same way using the notion of longest proper suffix. ST can be used for simulating an AC automaton for the set of suffixes. Both can be used for multiple exact string matching in linear time on constant alphabet, AC by preprocessing patterns, ST by preprocessing text.

(c) String quicksort and MSD radix sort.

**Solution.** Both are string sorting algorithm that work by partitioning the strings and then recursing on the partitions. Both partition by a single symbol following the known common prefix. String quicksort has three and MSD radix sort $\sigma$ partitions in each step. String quicksort works on ordered alphabet while MSD radix sort requires integer alphabet. Both have time complexity of the form $O(DP + n \log x)$, with $x = n$ for string quicksort and $x = \sigma$ for MSD radix sort.

(d) 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. [3+2+3 points]

(a) Explain what are ordered alphabet and integer alphabet.

**Solution.** An ordered alphabet is a finite ordered set. An integer alphabet is the set of integers in a range $[0..\sigma)$ for some $\sigma > 0$. An algorithm works on an ordered alphabet if it requires only symbol comparisons to perform its task. If the algorithm performs operations that rely on symbols being encoded by integers, it is said to require an integer alphabet. Examples of such operations are using a symbol to index a table and computing a hash value. Algorithms for integer alphabet are more restricted in their applicability.

(b) Give an example of an algorithm that works equally well with both kinds of alphabets.

**Solution.** Algorithms for ordered alphabet can be used on integer alphabet since integers can be compared. The (Knuth–)Morris–Pratt algorithm uses only symbol comparison and thus works with both kinds of alphabets.
(c) Give an example of an algorithm that works well with one type of alphabet but not the other. Explain why the algorithm requires a specific type of alphabet.

**Solution.** The Horspool algorithm uses a text symbol as an address to a table, where the corresponding shift length is stored. This operation requires the symbol to be an integer in a relatively small range. In principle, the lookup could be implemented using a binary search, for example, but this would significantly slow down the algorithm.

3. [3+3+3 points] Give

(a) the compact trie

**Solution.**

(b) the balanced ternary tree

**Solution.**

(c) the LLCP and RLCP arrays for efficient binary searching in the sorted array

**Solution.**
<table>
<thead>
<tr>
<th>left</th>
<th>LLCP</th>
<th>mid</th>
<th>RLCP</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>australia</td>
<td>0</td>
<td>latvia</td>
</tr>
<tr>
<td>australia</td>
<td>5</td>
<td>austria</td>
<td>0</td>
<td>latvia</td>
</tr>
<tr>
<td>latvia</td>
<td>1</td>
<td>liberia</td>
<td>2</td>
<td>lithuania</td>
</tr>
<tr>
<td>liberia</td>
<td>3</td>
<td>libya</td>
<td>2</td>
<td>lithuania</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>lithuania</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>lithuania</td>
<td>0</td>
<td>peru</td>
<td>0</td>
<td>span</td>
</tr>
<tr>
<td>peru</td>
<td>0</td>
<td>somalia</td>
<td>1</td>
<td>span</td>
</tr>
<tr>
<td>lithuania</td>
<td>0</td>
<td>span</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>spain</td>
<td>1</td>
<td>sudan</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>sudan</td>
<td>1</td>
<td>sweden</td>
<td>0</td>
<td>—</td>
</tr>
</tbody>
</table>

for the string set \{australia, austria, latvia, liberia, libya, lithuania, peru, somalia, spain, sudan, sweden\}.

4. [11 points] Let \( R = \{S_1, S_2, \ldots, S_k\} \) be a set of strings. Strings \( S_i \) and \( S_j \) are rotations of each other if \( S_i = uv \) and \( S_j = vu \) for some strings \( u \) and \( v \). Describe an algorithm for finding all strings in \( R \) that are rotations of another string in \( R \). The algorithm should report each string only once even if it is a rotation of many other strings. The time complexity should be linear on a constant size alphabet.

**Solution.**

Divide the strings into subsets by their length (using counting sort). Only strings of the same length can be rotations of each other.

Let \( R' = \{S'_1, S'_2, \ldots, S'_k\} \) be a set of strings with the same length. Build the Aho–Corasick automaton for \( R' \). For each string \( S'_i \) in \( R' \) form the string \( S'_i S'_i \) and scan it using the AC automaton. If an occurrence of \( S'_j, j \neq i \), is found, then \( S'_i \) is a rotation of \( S'_j \) and \( S'_i \) is reported.

The AC automaton can be constructed in linear time and the scanning of \( S'_i S'_i \) takes \( O(|S'_i|) \). Thus the time complexity is linear in the total length of the strings.

5. [10 points] The task is to find the longest string \( S \) that occurs at least three times in a text \( T \) of length \( n \). Describe how to find \( S \) in linear time given the suffix array of \( T \) and the associated LCP array without constructing any major additional data structures.

**Solution.**

If string \( S \) of length \( m \) occurs at least three times in \( T \), then at least three suffixes of \( T \) have \( S \) as a prefix, and the suffixes are consecutive in the suffix array. Thus there are two consecutive values larger or equal to \( m \) in the LCP array. This works in the other direction too. If \( LCP[i] \) and \( LCP[i + 1] \) are both larger or equal to \( m \), then the suffixes \( SA[i - 1], SA[i], \) and \( SA[i + 1] \) have the same prefix of length \( m \) and thus that prefix occurs at least three times in \( T \).

The algorithm is then:

(a) Find \( i \) that maximizes \( m = \min\{LCP[i], LCP[i + 1]\} \).

(b) Return \( T[SA[i]..SA[i] + m] \).