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 few lines for each part is sufficient.

(a) Shift–And algorithm and BNDM algorithm.

Solution.
Both are exact string matching algorithms. Both are based on bit-parallel simulation of a nondeterministic automaton. Shift–Or automaton maintains information about which pattern prefixes match a suffix of the scanned string and accepts when the string ends with the full pattern, while the BNDM automaton maintains information about which factors of the reverse pattern match the scanned string and accepts suffixes of the reverse pattern. Shift–Or makes one, long left-to-right scan, while BNDM makes many right-to-left scans and shifts to the right between scans. Both require an integer alphabet. When pattern length $m$ is at most the machine word size $w$, the BNDM search time complexity is $O(mn)$ in the worst case, $O(n \log \sigma m / m)$ in the average case (which is optimal), and $O(n/m)$ in the best case, while the Shift-Or search time complexity is always $O(n)$ (which is optimal in the worst case). Both need $O(m + \sigma)$ time for pattern preprocessing. Both algorithms slow down when $m > w$.

(b) LSD radix sort and MSD radix sort.

Solution.
Both are string sorting algorithms for integer alphabet. Both use counting sort as a subroutine. Both process string one position at a time. LSD radix sort proceeds from the end to the beginning, while MSD radix sort proceeds in the opposite direction. MSD radix sort recurses into smaller and smaller subsets (and switches to string quicksort for small subsets), while LSD radix sort processes all strings in each step. LSD radix sort always processes the whole strings, while MSD radix sort processes only the shortest distinguishing prefixes, which makes MSD radix sort better for long and variable length strings. The time complexities are $O(||R|| + m\sigma)$ (LSD) and $O(\sum LCP(R) + n \log \sigma)$ (MSD).

(c) Suffix array and Burrows–Wheeler transform.

Solution.
Both are arrays computed from the text based on the lexicographic order of suffixes (SA) or rotations (BWT). They are connected by the equation

$$BWT[i] = \begin{cases} \text{"$" if } SA[i] = 0 \\ T[SA[i] - 1] \text{ otherwise} \end{cases}$$

Both have linear size and can be constructed in linear time. BWT is typically more space efficient, because it replaces the text (it is invertible) while SA is used with the text; because its elements are symbols while SA elements are integers; and because BWT can often be compressed. Both support indexed exact string matching over the text: SA using binary search and BWT using backward search.
2. [12 points] Construct the Aho–Corasick automaton for the pattern set \{string, ring, trie, log, ecology\}. Simulate the scanning of the text stringology with the automaton.

**Solution.** The automaton:

Failure links going to stage 0 have been omitted for clarity. The non-empty sets of patterns are:

<table>
<thead>
<tr>
<th>state</th>
<th>patterns(state)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>log</td>
</tr>
<tr>
<td>7</td>
<td>ecology</td>
</tr>
<tr>
<td>10</td>
<td>log</td>
</tr>
<tr>
<td>14</td>
<td>ring</td>
</tr>
<tr>
<td>20</td>
<td>ring, string</td>
</tr>
<tr>
<td>24</td>
<td>trie</td>
</tr>
</tbody>
</table>

Simulation:

<table>
<thead>
<tr>
<th>i</th>
<th>T[i]</th>
<th>transition(s)</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>0 → 15</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>t</td>
<td>15 → 16</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r</td>
<td>16 → 17</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>17 → 18</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>18 → 19</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>g</td>
<td>19 → 20</td>
<td>ring, string</td>
</tr>
<tr>
<td>6</td>
<td>o</td>
<td>20 → 14 → 0 → 0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>l</td>
<td>0 → 8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>o</td>
<td>8 → 9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>g</td>
<td>9 → 10</td>
<td>log</td>
</tr>
<tr>
<td>10</td>
<td>y</td>
<td>10 → 0 → 0</td>
<td></td>
</tr>
</tbody>
</table>
3. [6+6 points] Let $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$ be two strings over the alphabet of real numbers, i.e., $a_i, b_j \in \mathbb{R}$ for all $1 \leq i \leq m$, $1 \leq j \leq n$. Let us define a variant of edit distance for such strings. The edit operations are the standard insertion, deletion and substitution of single symbols. The cost of substituting $a_i$ with $b_j$ is $|a_i - b_j|$, i.e., the absolute value of the difference. There are two models for the cost of insertions and deletions (indels):

(a) The cost of inserting or deleting a symbol $c$ is $|c|$, i.e., the absolute value of the symbol.

(b) Indels have no cost, but the total number of indels must be at most $K$.

Describe algorithms for computing these edit distance variants. The time complexity should be $\mathcal{O}(mn)$ for (a)-part and $\mathcal{O}(mnK)$ for (b)-part. You may assume that all basic arithmetic operations on real numbers can be performed in constant time.

**Solution.**

(a) Modify the standard dynamic programming algorithm to use the stated costs:

1. $d_{00} = 0$
2. for $i \leftarrow 1$ to $m$ do $d_{i0} \leftarrow d_{i-1,0} + |a_i|$
3. for $j \leftarrow 1$ to $n$ do $d_{0j} \leftarrow d_{0,j-1} + |b_j|$
4. for $j \leftarrow 1$ to $n$ do
5. for $i \leftarrow 1$ to $m$ do
6. $d_{ij} \leftarrow \min\{d_{i-1,j-1} + |a_i - b_j|, d_{i-1,j} + |a_i|, d_{i,j-1} + |b_j|\}$
7. return $d_{mn}$

(b) Add a third dimension to the dynamic programming table representing the limit on the number of indels. That is, $d_{ijk}$ is the specified edit distance between $a_1 \cdots a_i$ and $b_1 \cdots b_j$ when the number of indels is at most $k$. If $|i - j| > k$, it is not possible to transform $a_1 \cdots a_i$ into $b_1 \cdots b_j$ using at most $k$ indels, and we define $d_{ijk} = \infty$.

1. for $k \leftarrow 0$ to $K$ do
2. for $j \leftarrow 0$ to $n$ do
3. for $i \leftarrow 0$ to $m$ do
4. $d_{ijk} = \infty$
5. $d_{000} = 0$
6. for $i \leftarrow 1$ to $\min(m, n)$ do
7. $d_{i00} = d_{i-1,0} + |a_i - b_1|$
8. for $k \leftarrow 1$ to $K$ do
9. for $i \leftarrow 0$ to $\min(k, m)$ do $d_{i0k} \leftarrow 0$
10. for $j \leftarrow 1$ to $\min(k, n)$ do $d_{0jk} \leftarrow 0$
11. for $j \leftarrow 1$ to $n$ do
12. for $i \leftarrow 1$ to $m$ do
13. $d_{ijk} \leftarrow \min\{d_{i-1,j-1,k} + |a_i - b_j|, d_{i-1,j,k-1}, d_{i,j-1,k-1}\}$
14. return $d_{mnK}$
4. [12 points] Let $T$ be a string of length $n$ over an alphabet $\Sigma$ of constant size. Describe an algorithm that finds the shortest string over the alphabet $\Sigma$ that occurs exactly $k$ times in $T$. The time complexity should be $O(n)$.

Solution. In the suffix tree of $T$, a node $v$ represents a string $S_v$ that occurs exactly $k$ times in $T$ if and only if the subtree rooted at $v$ has $k$ leafs. A locus $(v,d)$ between a node $v$ and the parent of $v$ represent a prefix of $S_v$ that has the same number of occurrences than $S_v$, and the locus one character below the parent represents the shortest such prefix. The shallowest (i.e., the least deep) such locus is represents the shortest such string. Thus the algorithm is:

1. Construct the suffix tree of $T$. This takes $O(n)$ time.
2. For each node $v$, compute the number of leaves in the subtree rooted at $v$. This can be done in linear time during a depth-first traversal: The leaf count of $v$ is 1 if $v$ is a leaf and sum of the leaf counts of $v$’s children if $v$ is an internal node.
3. Find the node with a leaf count $k$ and the shallowest parent, and report the string represented by the locus just below the parent. This can be done with any linear time traversal of the suffix tree. Return null if there are no nodes with leaf count exactly $k$.

5S. [12 points] (Separate exam) Describe one of these topics discussed in the first study group:
- burstsort
- weak prefix search
- sparse suffix sorting
- comparison-driven data structures for strings.

Try to answer the following questions:
- What is the problem solved in the paper?
- What are the main ideas of the solution?
- How is the solution related to the algorithms and data structures described on the lectures?

Solution. See study group material.

5R. [6+6 points] (Renewal exam only) Tries and data structures based on tries (e.g., suffix tree) involve an operation called child.

(a) Describe the child-operation and explain why its implementation is an important and non-trivial problem.

(b) List different approaches for dealing with the problem together with the main advantages and disadvantages of each approach.

Solution.

(a) For a node $v$ and a symbol $c \in \Sigma$, $\text{child}(v,c)$ is $u$ if $u$ is a child of $v$ and the edge $(v,u)$ is labeled with $c$ (trie) or starts with $c$ (compact trie), and $\text{child}(v,c) = \bot$ (null) if $v$ has no such child.

The child function stores the tree topology in the root-to-leaves direction, allowing downward traversals, which are essential to many queries on tries, e.g. prefix or lcp.
queries. Depending on the application, the data structure implementing the child function should support efficient lookup queries, updates (insertions and deletions) or listing of all elements.

A suitable implementation of child function depends also on the alphabet type, which defines the operations that are permitted on symbols, e.g., symbols from ordered alphabet can only be compared, thus we have to use comparison-based data structure (such as BST), but if the symbols are known to be integers, a faster data structure (e.g., a hash table) could be used.

Consider for example the string over ordered alphabet for which we want to build the suffix tree used to answer prefix queries. Using McCreight’s algorithm for the suffix tree construction requires efficient lookups and insertions, hence the self-balancing binary search tree (e.g., AVL tree) is the best choice. However, if we build the suffix tree from precomputed suffix array and LCP array, it suffices to use linked-list for implementing child, since the algorithm requires only insertions at the end of the children list. After constructing the tree using this method, we can easily rebuild the linked-list into balanced static BST (since at this point only efficient lookups are required).

Finally, note that the used child implementation can heavily affect the space consumption of a trie, e.g., when dealing with integer alphabet we can achieve constant lookup/update time by storing an array of size \( \sigma \) in each node. However, as \( \sigma \) grows this solution becomes ineffective due to large memory consumption.

(b) Array: Each node stores an array of size \( \sigma \). The space complexity is \( \mathcal{O}(\sigma N) \), where \( N \) is the number of nodes in the trie. The time complexity of the child operation is \( \mathcal{O}(1) \). Updates are also supported in constant time. Listing all children of a node, however, takes \( \mathcal{O}(\sigma) \) time. This child representation requires an integer alphabet. Due to space consumption growing proportionally to \( \sigma \), it is practical only for small alphabets.

• Linked-list: Each node stores a list of pointers. The space complexity is \( \mathcal{O}(N) \). The time is \( \mathcal{O}(\sigma) \) for child and \( \mathcal{O}(1) \) for updates. Listing takes time proportional to the number of children. Works for an ordered alphabet. This solution is attractive only if the child is not invoked very often or the alphabet is small.

• Binary search tree (balanced): Replace the list with the tree. The space complexity is \( \mathcal{O}(N) \). The query/update time is \( \mathcal{O}(\log \sigma) \). Listing takes time proportional to the number of children. Works for an ordered alphabet. This approach is best suited for large ordered alphabets but could be improved if the symbols are integers.

• Hash table: One hash table for the whole trie, storing the values \( \text{child}(v, c) \neq \perp \). Space complexity is \( \mathcal{O}(N) \), time complexity of \( \text{child} \) is \( \mathcal{O}(1) \). If we use sophisticated hashing methods (e.g., dynamic perfect hashing) we can achieve amortized \( \mathcal{O}(1) \) updates. Listing all children takes \( \mathcal{O}(\sigma) \) time (by asking a child query for all symbols of the alphabet in ascending order). Requires an integer alphabet. This solution should be chosen for large integer alphabets but it is not very practical if we often need to iterate over all children of a given node.