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) (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 string mergesort.

**Solution.** Both are algorithms for sorting strings. Both are adaptations of standard (non-string) sorting algorithms. Both work on an ordered alphabet and have time complexity $O(n \log n + L)$ for sorting $n$ strings with total lcp length of $L$. Both are divide-and-conquer algorithms: string quicksort partitions by value and concatenates sorted parts, while string mergesort partitions arbitrarily and merges sorted parts. Both use lcp information. String quicksort compares strings using only one symbol at a given stage, while string mergesort compares strings using the lcp-comparison technique. String quicksort needs only $O(\log n)$ extra space, while string mergesort needs $O(n)$ extra space.

(c) Aho–Corasick automaton and suffix tree.

**Solution.** Both can solve the multiple exact string matching problem in linear time for a constant alphabet, but the solutions are very different: AC algorithm preprocess the patterns, while suffix tree solution preprocesses the text. AC algorithm is particularly designed for solving multiple exact string matching, but suffix tree can be used for solving many other problems, too.

Both are based on the trie structure. AC automaton is a trie for the set of pattern, while the suffix tree is the compact trie for the set of suffixes of the text. Both tries can be augmented with special links defined similarly, failure links in AC automaton and suffix links in suffix tree. Suffix tree can be used as an AC automaton for the set of suffixes.

A few lines for each part is sufficient.
2. [6+6 points] Let \( A = a_1a_2 \cdots a_m \) and \( B = b_1b_2 \cdots b_n \) 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.

**Solution.** 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) Indels have no cost, but the total number of indels must be at most \( K \).

**Solution.** 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 \( \min(m, n) \) do 
10. \( d_{i0k} \leftarrow 0 \)
11. for \( j \leftarrow 1 \) to \( \min(k, n) \) do 
12. \( d_{0jk} \leftarrow 0 \)
13. for \( j \leftarrow 1 \) to \( n \) do 
14. for \( i \leftarrow 1 \) to \( m \) do 
15. \( d_{ijk} \leftarrow \min\{d_{i-1,j-1,k} + |a_i - b_j|, d_{i-1,j,k-1} + |a_i|, d_{i,j-1,k-1} + |b_j|\} \)
16. return \( d_{mnK} \)

Describe algorithms for computing these edit distance variants. The time complexity should be \( O(mn) \) for (a)-part and \( O(mnK) \) for (b)-part. You may assume that all basic arithmetic operations on real numbers can be performed in constant time.
3. [3+3+4 points] Give

(a) the compact trie

\[3+3+4 \text{ points}\] Give

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></td>
<td>0</td>
<td>latvia</td>
<td>1</td>
<td>lithuania</td>
</tr>
<tr>
<td>latvia</td>
<td>1</td>
<td>libanon</td>
<td>2</td>
<td>lithuania</td>
</tr>
<tr>
<td>libanon</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>mexico</td>
<td>0</td>
<td>spain</td>
</tr>
<tr>
<td>mexico</td>
<td>0</td>
<td>singapore</td>
<td>1</td>
<td>spain</td>
</tr>
<tr>
<td>lithuania</td>
<td>0</td>
<td>spain</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, libanon, libya, lithuania, mexico, singapore, spain, sudan, sweden\}.
4. [9 points] Define the suffix link in suffix trees and describe briefly its role in a linear time suffix tree construction algorithm.

**Solution:**

Let $S_v$ be the string represented by a node $v$ in a suffix tree. For every node $u$ (except the root), there is a suffix link from $u$ to another node $v$ such that $S_u = cS_v$ for some symbol $c$.

McCreight’s linear time suffix array construction algorithm adds suffixes one at a time to the suffix tree. The brute force algorithm would find the insertion point by following a path form the root. McCreight’s algorithm uses suffix links as short cuts. More precisely, consider the situation immediately after adding the leaf $w_i$ representing the suffix $T_i$. Let $u_i$ be the parent of $w_i$ and let $v_i$ be the node where suffix link from $u_i$ goes to. Next the algorithm adds $T_{i+1}$. Since $u_i$ represents a prefix of $T_i$, $v_i$ represents a prefix of $T_{i+1}$. Thus the leaf $w_{i+1}$ is added somewhere in the subtree rooted at $v_i$, and McCreight’s algorithm starts the insertion of $T_{i+1}$ from $v_i$. Without the short cut the time complexity would be $\Theta(L(T_{[0..n]}))$.

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 $T[SA[i]..SA[i] + m]$.