1. Outline algorithms that find the most frequent symbol in a given string

(a) in general alphabet model, and
(b) in integer alphabet model.

The algorithms should be as fast as possible. What are their (worst case) time complexities? Consider also the case where \( \sigma \gg n \).

**Solution:** With integer alphabet, we have the following algorithm

**Input:** string \( S = S[0\ldots n] \)

**Output:** most frequent symbol in \( S \)

1. for \( c \in \Sigma \) do \( C[c] \leftarrow 0 \)
2. \( \text{maxfreq} \leftarrow -1 \)
3. for \( i \leftarrow 0 \) to \( n - 1 \) do
4. \( c \leftarrow S[i] \)
5. \( C[c] \leftarrow C[c] + 1 \)
6. if \( C[c] > \text{maxfreq} \) then \( \text{maxfreq} \leftarrow C[c]; \text{maxsym} \leftarrow c \)
7. return \( \text{maxsym} \)

The time complexity is \( \mathcal{O}(n + \sigma) \). If \( \sigma \gg n \), we would like to reduce the time to \( \mathcal{O}(n) \). The time complexity is in fact \( \mathcal{O}(n) \) except for the initialization of the counters. If we use a hash table to store only non-zero counters, the time complexity is reduced to \( \mathcal{O}(n) \) with high probability but not in the worst case.

There is a way to avoid the cost of initialization. We allocate the space for the counter array \( C[0..\sigma] \) and another array \( F[0..\sigma] \) but do not initialize them. (We assume that it is possible to allocate any amount of uninitialized memory in constant time.) Once we have encountered a symbol \( c \) at least once, \( C[c] \) will be the number of occurrences of \( c \) and \( F[c] \) will be the position of the first occurrence. If \( c \) has not appered yet, \( C[c] \) and \( F[c] \) are uninitialized and either \( F[c] \) is not a position in the processed part of \( S \) or \( S[F[c]] \neq c \). Incrementing the counter on line (5) is now replaced with the following code:

\[
\begin{align*}
(5.1) & \quad \text{if } F[c] \in [0..i] \text{ and } S[F[c]] = c \text{ then } C[c] \leftarrow C[c] + 1 \\
(5.2) & \quad \text{else } C[c] \leftarrow 1; \quad F[c] \leftarrow i
\end{align*}
\]

The time complexity is now \( \mathcal{O}(n) \) in the worst case. A general version of this “lazy array initialization trick” is described for example here: [http://eli.thegreenplace.net/2008/08/23/initializing-an-array-in-constant-time](http://eli.thegreenplace.net/2008/08/23/initializing-an-array-in-constant-time)

In general alphabet model, we can use the same approach but the counters are stored in a balanced search tree using the symbol as a key. The time complexity is \( \mathcal{O}(n \log \sigma) \), since each update of the counters takes \( \mathcal{O}(\log \sigma) \) time. More precisely, the time complexity is actually \( \mathcal{O}(n \log \sigma_S) \), where \( \sigma_S \) is the number of distinct symbols that appears in \( S \).

An alternative approach is to sort the symbols and then scan to count the number of occurrences. In fact, the above solution for an integer alphabet is similar to counting sort. We can treat each integer as a base-\( d \) number for any constant \( d \) and LSD radix sort the list in \( \log_d \sigma \) rounds.

In general alphabet model, we can use any comparison based sorting algorithms to achieve \( \mathcal{O}(n \log n) \) time, but by using ternary quicksort the time complexity is \( \mathcal{O}(n \log \sigma_S) \).

(The time complexity of ternary quicksort is in fact \( \mathcal{O}(n H_0(S)) \), where \( H_0(S) \leq \log \sigma_S \) is the zeroth order empirical entropy of \( S \). The same time complexity can be achieved by using a splay tree as the search tree for counters.)
2. Let $\mathcal{R} = \{\text{manne, manu, minna, salla, saul, sauli, vihtori}\}$.

(a) Give the compact trie of $\mathcal{R}$.

Solution:

(b) Give the balanced compact ternary trie of $\mathcal{R}$.

Solution:
3. What is the time complexity of prefix queries for

(a) trie in constant alphabet model
(b) compact trie in constant alphabet model
(c) compact trie in general alphabet model using a binary tree implementation of the child function
(d) balanced compact ternary trie?

The queries should return the resulting strings as a list of pointers or other identifiers rather than the full strings.

Solution:

Let \( \mathcal{R} \) be the set of strings, \( P \) be the query argument, and \( O \) the query result (a set of strings). The prefix query time complexities are the following.

(a) Trie with constant alphabet. Any reasonable implementation of child function allows computing it in \( O(1) \) time, thus finding the node representing \( P \) in the trie takes \( O(|P|) \) time. Locating all leaves however can take as much as \( O(||O||) \) steps. Consequently, the total query time is \( O(|P| + ||O||) = O(||\mathcal{R}||) \). We can reduce the time to \( O(|P| + |O|) \) with additional data structures: a linked list connecting the leaves in a lex order and, for each node, pointers to the first and the last node in the subtree.

(b) Compact trie with constant alphabet. The child function can be computed in \( O(1) \) time similarly to case (a). This time path compression allows listing all leaves in the subtree in time proportional to its size, resulting in \( O(|P| + |O|) \) query time.

(c) Compact trie with general alphabet and binary tree implementation of the child function. The time complexity to compute the child function is \( O(\log \sigma) \) (assuming the tree is balanced or on average) and locating all the leaves is identical as in (b). Total query time is therefore \( O(|P| \log \sigma + |O|) \). If the binary trees are not balanced the query time increases to \( O(|P|\sigma + |O|) \).

(d) Balanced compact ternary trie. Descending down in the tree takes at most \( O(|P| + \log |\mathcal{R}|) \) operations in total. Listing all the leaves works similarly to compacted tries. Total query time is thus \( O(|P| + \log |\mathcal{R}| + |O|) \).

In all cases we could avoid the additive \( |O| \) factor by storing in each node a list of pointers to the strings in the leaves below. However this would take \( O(|\mathcal{R}|^2) \) space in a compact trie and \( O(|\mathcal{R}|^3) \) space in a non-compact trie.
4. Prove

(a) Lemma 1.14: For $i \in [2..n]$, $LCP_R[i] = lcp(S_i, \{S_1, \ldots, S_{i-1}\})$.

**Solution:**
Since $LCP_R[i] = lcp(S_i, S_{i-1})$ and $lcp(S_i, \{S_1, \ldots, S_{i-1}\}) = \max\{lcp(S_i, S_1), \ldots, lcp(S_i, S_{i-1})\}$, it is enough to show that $lcp(S_{i-1}, S_i) \geq lcp(S_j, S_i)$ for all $j < i - 1$. Assume to the contrary that $lcp(S_{i-1}, S_i) < lcp(S_j, S_i)$ for some $j$. Letting $\ell = lcp(S_{i-1}, S_i) < lcp(S_j, S_i)$, we must have $S_{i-1}[0..\ell] = S_i[0..\ell] = S_j[0..\ell]$ and $S_{i-1}[\ell] \neq S_i[\ell] = S_j[\ell]$. Thus either $S_{i-1} < S_i$ or $S_{i-1} > S_i$, both of which contradict the fact that $S_j \leq S_{i-1} \leq S_i$.

(b) Lemma 1.15: $\Sigma LCP(R) \leq \Sigma lcp(R) \leq 2 \cdot \Sigma LCP(R)$.

**Solution:**
Let $R = \{S_1, S_2, \ldots, S_n\}$ and $S_1 < S_2 < \ldots < S_n$. For convenience we define $S_{n+1}$ to be a string that have a common prefix of length 0 with any other string so that $LCP_R[n+1] = 0$.

Let us consider the lower bound first. By using the result from (a) we have

$$LCP_R[i] = lcp(S_i, \{S_1, \ldots, S_{i-1}\}) \leq lcp(S_i, R \setminus \{S_i\}).$$

Taking the sum over all $i$ gives

$$\sum_{i \in [1..n]} LCP_R[i] \leq \sum_{i \in [1..n]} lcp(S_i, R \setminus \{S_i\})$$

which proves the claim.

The prove the upper bound, observe that using a symmetrical argument to the one used in (a), we have

$$lcp(S_i, \{S_{i+1}, \ldots, S_{n+1}\}) = lcp(S_i, S_{i+1}).$$

Combining the above equality with (a) therefore shows that

$$lcp(S_i, R \setminus \{S_i\}) = \max\{lcp(S_i, S_{i-1}), lcp(S_i, S_{i+1})\} = \max\{LCP_R[i], LCP_R[i+1]\}.$$ 

Since we have $\max\{LCP_R[i], LCP_R[i+1]\} \leq LCP_R[i] + LCP_R[i+1]$, we can write

$$\sum_{i \in [1..n]} lcp(S_i, R \setminus \{S_i\}) \leq \sum_{i \in [1..n]} (LCP_R[i] + LCP_R[i+1]).$$

In the last sum, each $LCP_R[i]$ appears at most twice, thus we obtain

$$\sum_{i \in [1..n]} lcp(S_i, R \setminus \{S_i\}) \leq 2 \cdot \sum_{i \in [1..n]} LCP_R[i]$$

which proves the claim.
5. Show how to construct the compact trie for a set $\mathcal{R}$ in $O(|\mathcal{R}|)$ time (rather than $O(||\mathcal{R}||)$ time) given the string set $\mathcal{R}$ in lexicographical order and the LCP array $LCP_\mathcal{R}$.

Solution: Let $\mathcal{R} = \{S_1, \ldots, S_n\}$. For any node $u$ of compact trie $\text{trie}(\mathcal{R})$ let $P_u$ be the string obtained by concatenating the labels of edges on the path from the root of $\text{trie}(\mathcal{R})$ to $u$. We represent $\text{trie}(\mathcal{R})$ using the following functions:

- $\text{child}(u, c)$ is the child $v$ of node $u$ such that the label of edge $(u, v)$ starts with $c$.
- $\text{depth}(u)$ is the length of $P_u$.
- $\text{index}(u)$ is any integer $i$ such that $P_u$ is a prefix of $S_i$.
- $\text{parent}(u)$ is the parent of $u$.

**Algorithm** ConstructCompactTrie

**Input:** Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings in lexicographical order and the array $LCP = LCP_\mathcal{R}$.

**Output:** Compact trie $\text{trie}(\mathcal{R})$.

1. create new node $\text{root}; \text{depth}(\text{root}) \leftarrow 0$
2. $u \leftarrow \text{root}$
3. for $i \leftarrow 1$ to $n$ do // insert string $S_i$
   4. while $u \neq \text{root}$ and $LCP[i] \leq \text{depth}(\text{parent}(u))$ do
   5. $u \leftarrow \text{parent}(u)$
   6. if $LCP[i] < \text{depth}(u)$ then // we are in the middle of an edge
       7. create new node $v$
       8. $\text{index}(v) \leftarrow i; \text{depth}(v) \leftarrow LCP[i]$
       9. $p \leftarrow \text{parent}(u); j \leftarrow \text{index}(u)$
      10. $\text{child}(p, S_i[\text{depth}(p)]) \leftarrow v; \text{parent}(v) \leftarrow p$
      11. $\text{child}(v, S_j[\text{depth}(v)]) \leftarrow u; \text{parent}(u) \leftarrow v$
      12. $u \leftarrow v$
     13. create new leaf $w$ // $w$ represents string $S_i$
     14. $\text{index}(w) \leftarrow i; \text{depth}(w) \leftarrow |S_i|$
     15. $\text{child}(u, S_i[LCP[i]]) \leftarrow w; \text{parent}(w) \leftarrow u$
     16. $u \leftarrow w$

The correctness of this algorithm follows from the definition of a trie. To prove that its worst case time complexity is $O(|\mathcal{R}|)$ we observe that insertion of each string $S_i$ takes constant time, except lines (4)-(5). Each step of this loop decreases the node-depth (the number of nodes on the path from root) of $u$. Since in each iteration of the for loop the node-depth of $u$ can increase by at most one, the total number of steps performed in lines (4)-(5) is bounded by $2n = O(|\mathcal{R}|)$ (including the cost of checking the condition even when it evaluates to false).

It is possible to reduce the memory requirements of this algorithm by eliminating the usage of $\text{parent}$ function. It is used to navigate along the rightmost path in the trie. Such path however could be easily maintained using a stack.