Exercises 3, solutions

1. Use the lcp comparison technique to modify the standard insertion sort algorithm so that it sorts strings in \(\mathcal{O}(\Sigma \text{LCP}(R) + n^2)\) time.

**Solution:** The following algorithm produces the same output as StringMergesort.

**Algorithm** StringInsertionSort

**Input:** Set \(R = \{S_1, S_2, \ldots, S_n\}\) of strings.

**Output:** \(R\) sorted and augmented with lcp information.

1. \((T_1, \ell_1) \leftarrow (S_1, 0)\)
2. **for** \(i \leftarrow 2 \text{ to } n\) **do**
3. \(j \leftarrow i - 1\)
4. \((x, h) \leftarrow \text{LcpCompare}(S_i, T_j, 0)\)
5. **while** \(x = " < \) **do**
6. \((T_{j+1}, \ell_{j+1}) \leftarrow (T_j, h) \text{ // Make room for } S_i\)
7. **if** \(j = 1\) **then** \(x \leftarrow " > \); h \leftarrow 0\)
8. **else if** \(h > \ell_j\) **then** \(x \leftarrow " > \); h \leftarrow \ell_j\)
9. **else if** \(h < \ell_j\) **then** \(x \leftarrow " < \); \ell_{j+1} \leftarrow \ell_j\)
10. **else** \((x, h) \leftarrow \text{LcpCompare}(S_i, T_{j-1}, h)\)
11. \(j \leftarrow j - 1\)
12. \((T_{j+1}, \ell_{j+1}) \leftarrow (S_i, h)\)
13. **return** \{\((T_1, \ell_1), (T_2, \ell_2), \ldots, (T_n, \ell_n)\)\}

The logic of the algorithm is based on the following result:

If \(B, B' \leq C\), then

(a) If \(\text{lcp}(B', C) > \text{lcp}(B, C)\) then \(B' > B\) and \(\text{lcp}(B', B) = \text{lcp}(B, C)\).

(b) If \(\text{lcp}(B', C) < \text{lcp}(B, C)\) then \(B' < B\) and \(\text{lcp}(B', B) = \text{lcp}(B', C)\).

(c) If \(\text{lcp}(B', C) = \text{lcp}(B, C)\) then \(\text{lcp}(B', B) \geq \min\{\text{lcp}(B', C), \text{lcp}(B, C)\}\).

which is easily proven using the results from the lectures.

When the algorithm enters the while loop, the correspondence between the above result and the variables in the algorithm is:

\[
\begin{align*}
B' &= S_i \\
B &= T_{j-1} \\
C &= T_j \\
\text{lcp}(B', C) &= h \\
\text{lcp}(B, C) &= \ell_j
\end{align*}
\]

The correctness of the algorithm is easy to check from this.

The time complexity excluding the calls to LcpCompare is \(\mathcal{O}(n^2)\) (and can be smaller if the input is already almost in order).

Consider then the time spent in LcpCompare when inserting \(S_i\). Each call to LcpCompare that makes \(1 + t\) symbol comparisons increases \(h\) by \(t\). Since \(h\) never decreases and never increases beyond \(\text{lcp}(S_i, R \setminus \{S_i\})\), the number of extra comparisons is at most \(\text{lcp}(S_i, R \setminus \{Q\})\). In total, the extra time in LcpCompare is \(\Sigma \text{lcp}(R) = \mathcal{O}(\Sigma \text{LCP}(R))\).
2. $\Omega(\Sigma LCP(\mathcal{R}))$ is a lower bound for string sorting for any algorithm in the simple string model, i.e., if characters can be accessed only one at a time. However, the packed string model allows accessing $\Theta(\log_\sigma n)$ characters at a time.

Develop a version of MSD radix sort for the packed string model. What is the time complexity?

**Solution:**

We assume that $\sigma = 2^b$, where $b$ is the number of bits in the encoding of characters. Choose $s \leq \log_\sigma n = (\log n)/b$. Then $s$ characters fit into $\log n$ bits. Let $\mathcal{R}_s$ be $\mathcal{R}$ as strings over the super-alphabet $\Sigma^s$. We will treat the strings in $\mathcal{R}$ as strings over the super-alphabet $\Sigma^s$. Let $\mathcal{R}_s$ be the set of super-alphabet strings.

Some details need to be taken care of:

- A standard assumption is that the machine word size is at least $\log n$ bits because pointers need at least $\log n$ bits. Thus super-characters can be manipulated in constant time.
- The lengths of the strings are generally not multiples of $s$ and the last super-character of a string may be incomplete. The missing part is filled with zeros, which can be done in constant time. If two strings differ only in the number of trailing zeros, the last super-character completion may make the two strings equal. To correct for this, strings that are equal under the super-alphabet, are sorted by their normal length. This can be done in $O(n)$ time.
- Sometimes characters are not stored in words in the order we would like. For example, in little endian machines, the first byte of a word is the least significant one, while in lexicographical order, the first character is the most significant one. We can correct such problems by using a lookup table of size $\sigma^s$ to replace an incorrect super-character value with a correct one. This takes constant time per super-character.

Now we are left with the problem of sorting $\mathcal{R}_s$. Note that $\Sigma LCP(\mathcal{R}_s) \leq n + \Sigma LCP(\mathcal{R})/s$.

- Using MSD radix sort from the lectures, the time complexity is $O(\Sigma LCP(\mathcal{R})/s + n \log \sigma^s) = O(\Sigma LCP(\mathcal{R})/s + sn \log \sigma)$. Optimizing $s$ can be difficult because we do not know $\Sigma LCP(\mathcal{R})$ in advance. Setting $s = \lfloor \log_\sigma n \rfloor$ gives the time complexity $O(\Sigma LCP(\mathcal{R})/\log_\sigma n + n \log n)$. This is, in fact, no better than the super-alphabet version of string quicksort.
- Using the optimal MSD radix sort mentioned in the lectures, we can do better. The time complexity is $O(\Sigma LCP(\mathcal{R})/s + n + \sigma^s)$. Here we can always set $s = \lfloor \log_\sigma n \rfloor$ and get the time complexity $O(\Sigma LCP(\mathcal{R})/\log_\sigma n + n)$.

3. Give an example showing that the worst case time complexity of string binary search without precomputed lcp information is $\Omega(m \log n)$.

**Solution:** Let $P = a^m$ and let the string set $\mathcal{R}$ contain $n$ strings with a prefix $P$. Now the binary search always goes left and thus $llcp$ remains 0. Without precomputed lcps, $mlcp$ will be 0 before each comparison and thus each of the $\log n$ comparisons takes $O(m)$ time.
4. Define

\[
MLCP[mid] = \max\{LLCP[mid], RLCP[mid]\}
\]

\[
D[mid] = \begin{cases} 
0 & \text{if } MLCP[mid] = LLCP[mid] \\
1 & \text{otherwise}
\end{cases}
\]

Show that, if we store the arrays \(MLCP\) and \(D\) instead of \(LLCP\) and \(RLCP\), we can compute \(LLCP[mid]\) and \(RLCP[mid]\) when needed during the string binary search.

**Solution:** Assume \(D[mid] = 0\) at some point in the binary search (the case \(D[mid] = 1\) is symmetric). Then \(MLCP[mid] = LLCP[mid] \geq RLCP[mid]\) and the question is how can we compute \(RLCP[mid] = \text{lcp}(S[mid], S[right])\).

Let \(\ell = \text{lcp}(S_{\text{left}}, S_{\text{right}})\). By Lemma 1.30(b), \(\ell = \min\{LLCP[mid], RLCP[mid]\}\), and since \(LLCP[mid] \geq RLCP[mid]\), we must have \(\ell = RLCP[mid]\). On the other hand, \(\ell = \min\{llcp, rlc\}\), and thus \(RLCP[mid] = \min\{llcp, rlc\}\).

In summary:

\[
\begin{align*}
LLCP[mid] &= \begin{cases} 
MCLP[mid] & \text{if } D[mid] = 0 \\
\min\{llcp, rlc\} & \text{if } D[mid] = 1
\end{cases} \\
RLCP[mid] &= \begin{cases} 
MCLP[mid] & \text{if } D[mid] = 1 \\
\min\{llcp, rlc\} & \text{if } D[mid] = 0
\end{cases}
\end{align*}
\]

5. Let \(S[0..n]\) be a string over an integer alphabet. Show how to build a data structure in \(O(n)\) time and space so that afterwards the Karp–Rabin hash function \(H(S[i..j])\) for the factor \(S[i..j]\) can be computed in constant time for any \(0 \leq i \leq j \leq n\).

**Solution:** We observe that \(H(S[i..j])\) can be expressed as a combination of \(H(S[0..i])\) and \(H(S[0..j])\). More precisely, recall that

\[
H(B) = (H(AB) - H(A) \cdot r^{|B|}) \mod q
\]

Thus, with \(A = S[0..i]\) and \(B = S[i..j]\), we have

\[
H(S[i..j]) = (H(S[0..j]) - H(S[0..i]) \cdot r^{j-i}) \mod q
\]

The data structure therefore consists of two integer arrays \(H_{\text{pref}}[0..n]\) and \(\text{Pow}[0..n]\) where

\[
H_{\text{pref}}[k] = H(S[0..k]),
\]

\[
\text{Pow}[k] = r^k \mod q
\]

for any \(k \in \{0, \ldots, n\}\). These arrays are initialized \(H_{\text{pref}}[0] = 0\), \(\text{Pow}[0] = 1 \mod q\) and for \(k > 0\) can be computed as follows:

\[
H_{\text{pref}}[k] = (r \cdot H_{\text{pref}}[k-1] + s_{k-1}) \mod q,
\]

\[
\text{Pow}[k] = (r \cdot \text{Pow}[k-1]) \mod q.
\]

Overall, computing both arrays takes \(O(n)\) time and space. Now \(H(S[i..j])\) for \(0 \leq i \leq j \leq n\) can be computed in constant time:

\[
H(S[i..j]) = (H_{\text{pref}}[j] - \text{Pow}[j-i] \cdot H_{\text{pref}}[i]) \mod q.
\]