String Processing Algorithms 2015 - Week 3 Exercises

Rodion Efremov

November 8, 2015

Exercise 1

Describe how to modify the LSD radix sort algorithm to handle strings of varying length. The time complexity should be the one given in Theorem 1.27.

Solution

The time complexity mentioned in the Theorem 1.27 is $\mathcal{O}(||\mathcal{R}|| + m\sigma)$. All we need to do is to modify the COUNTING-SORT procedure:

Now, the desired LSD radix sort for variable-length strings is

Exercise 2

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

Solution

Why do we set LCP to zero?

Consider the following:

i	S_i	$\mathrm{LCP}_{\mathcal{R}}[i]$
1	$aaaba$	0
2	$abaaa$	1
3	aaa	1
4	$baaa$	0
5	$aabba$	0
6	$abaaa$	1

Exercise 3

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

Algorithm 1: COUNTING-SORT($\mathcal{R} = \{S_1, S_2, \ldots, S_n\}, \ell$)

```
for i = 0 to σ − 1 do
    C[i] = 0
s = 0
for i = 1 to n do
    if |S_i| < ℓ then
        s = s + 1
    else
        C[S_i[ℓ]] = C[S_i[ℓ]] + 1
sum = s
for i = 0 to σ − 1 do
    tmp = C[i]
    C[i] = sum
    sum = sum + tmp
p = 0
for i = 1 to n do
    if |S_i| < ℓ then
        J[p] = S_i
        p = p + 1
    else
        J[C[S_i[ℓ]]] = S_i
        C[S_i[ℓ]] = C[S_i[ℓ]] + 1
R = J
```

Algorithm 2: LSDRadixSort($\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$)

```
m = max_{i=1,2,...,n}{|S_i|}
for ℓ = m to 1 do
    COUNTING-SORT(R, ℓ)
```

Algorithm 3: INSERTIONSORT($\mathcal{R}, \mathrm{LCP}_{\mathcal{R}}$)

```
for i = 2 to n do
    s = S_i
    j = i − 1
    if LCP_R[i − 1] = 0 then
        LCP_R[i] = 0
    while j > 0 and lcpCOMPARE(S_j, s, LCP_R[i]) > 0 do
        S_{j+1} = S_j
        j = j − 1
    S_{j+1} = s
```

Solution

Suppose the sorted list of strings is

$$\begin{aligned}
\langle \overbrace{aaa\ldots aaa}^{m} a_1, \\
\overbrace{aaa\ldots aaa}^{m} a_2, \\
\ldots \\
\overbrace{aaa\ldots aaa}^{m} a_n \rangle
\end{aligned}$$

and the string to search for is $\overbrace{aaa\ldots aaa}^{m} a'$, where $a' \neq a_i$ for any $i \in \{1, 2, \ldots, m\}$ and $a_1 \leq a_2 \leq \cdots \leq a_n$. Now as there is no match, the binary search will do $\Omega(\log n)$ string comparisons, and as we assume the naïve implementation of the string comparison, each comparison will have to iterate through m first characters a before getting to the last character that fails the search, which leads to the worst case time complexity of $\Omega(m \log n)$.