Karp-Rabin fingerprinting on strings

Given a string $S = S[0 \dots n-1]$, and two positions $0 \leq i < j \leq n-1$, the longest common extension $lce_S(i,j)$ is the length of the maximal run of matching characters from those positions, namely: if $S[i]! = S[j]$ then $lce_S(i,j) = 0$; otherwise, $lce_S(i,j) = max\{l \geq 1 : S[i \dots i+l-1] = S[j \dots j+l-1]\}$. For example, if $S = abracadabra$, then $lce_S(1,2) = 0$, $lce_S(0,3) = 1$, and $lce_S(0,7) = 4$. Given S in advance for preprocessing, build a data structure for S based on the Karp-Rabin fingerprinting, in O(n log n) time, so that it supports subsequent online queries of the following two types:

- $lce_S(i,j)$: it computes the longest common extension at positions i and j in O(log n) time.
- $equalS(i,j,l)$: it checks if $S[i \dots i+l-1] = S[j \dots j+l-1]$ in constant time.

Analyze the cost and the error probability. The space occupied by the data structure can be O(n log n) but it is possible to use O(n) space. [Note: in this exercise, a onetime preprocessing is performed, and then many online queries are to be answered on the fly.]

SOLUTION
Karp-Rabin hashing on strings (i.e. $str[n]$)for solving the longest common extension problem. We have the following steps:

1. Create a data structure holding the hashes, with hashes of previous characters being held as a prefix. We fix a sufficient big prime p and we use the Karp-Rabin hash (i.e. for a string k and a base b: $h(k) = (k[0]b^{L-1} + k[1]b^{L-2} + \dot{+}k[L-1]b^0)\ mod\ p)$:

$$
\begin{aligned}
H[0] &= h(str[0]) \\
H[1] &= H[0]p + h(str[1]) \\
H[2] &= H[1]p + h(str[2]) \ = \ h(str[0])p^2 + h(str[1])p^1 + h(str[2])p^0 \\
&\dots \\
H[n-1] &= H[n-2]p + h(str[n-1]) \ = \ h(str[0])p^{n-1} + \dots + h(str[n-2])p^1 + h(str[n-1])p^0
\end{aligned}
$$

   Therefore the space used here is just O(n).

2. Firstly we care about equality. To compare equality of a substring of length l at indexes i, j, we need to know the sub-hash of the string. Thus, we take hashes of $H[i]$ and $H[i+l]$ and we compare calculate the hash of the sub-string between i and l:

$$
\begin{aligned}
H[Substring_i] =& H[i+l] - H[i] * p^{(l-1)} \\
=& h(str[0])p^{i+l-1} + \dots + h(str[i+l-2])p^1 + h(str[i+l-1])p^0 \\
&-(h(str[0])p^{i-1} + \dots + h(str[i-2])p^1 + h(str[i-1])p^0) * p^{(l-1)} \\
=& h(str[i+l-1])p^0 + h(str[i+l-2])p^1 \cdots + h(str[0])p^{i+l-1} \\
&-(h(str[i-1])p^{(l-1)} + h(str[i-2])p^l \cdots + h(str[0])p^{i+l-2}) \\
=& h(str[i+l-1])p^0 + h(str[i+l-2])p^1 \cdots + h(str[i])p^{i+l-1}
\end{aligned}
$$

   We do the same procedure for the sub-string between j and l (i.e. $H[Substring_j]$), the we can simply compare the two hash. We introduce a base case (a sanity check) in the case $l = 1$ and we have $str[i] \neq str[j]$. The cost of this operation i O(1) since we did just simple operation $(+\ *)$.
   We choose a random prime number $p \in [2, \cdots, \tau]$ where $\tau > n$. Since the prime number in the interval $[2, \cdots, \tau]$ are approximately $\frac{\tau}{ln(\tau)}$, and we have a collision when $Substring_i = Substring_j$ but $H[Substring_i] \neq H[Substring_j]$, thus when $c = Substring_i - Substring_j\ mod\ p = 0$. We can conclude that $P_r[error] \leq \frac{\#\text{BAD PRIME}}{\#\text{PRIME}} = \frac{n}{\frac{\tau}{ln(\tau)}}$ because there are at most n distinct prime p that divide c (Chinese Theorem of residual). If we choose $\tau \approx n^{a+1}ln(n)$ then we have $P_r[error] \leq \frac{1}{n^a}$.

3. Finally to compute the the longest common extension $lce_S(i,j)$, we just do a binary search of the index l. The cost to do so is O(ln(n)) since the check of the equality is constant.

SOLUTION in O(N LOG N) works for every kind of hash function

The proposed data structure that maintain a series of trees. At each level we encode power of two elements (THIS IS IMPOSSIBLE TO EXPLAIN... LOOK THE CODE). The space occupied by this data structure is $O(nlog(n))$ where n is the length of the input array.

function CREATETREE(S,n,p)
  $A \leftarrow NEW\ Array[log_2(n)+1]$
  $TEMP \leftarrow NEW\ Array[n]$
  for $i\ IN\ (0,n)$ **do**
    $TEMP[i] \leftarrow S[i]\ mod\ p$
  end for
  $A[0] \leftarrow TEMP$
  for $i\ IN\ (1, log_2(n))$ **do**
    $TEMP \leftarrow NEW\ Array[n-i]$
    for $j\ IN\ (0, n-i)$ **do**
      $TEMP[j] \leftarrow A[i-1][j] + A[i-1][j+2^{i-1}]\ mod\ p$
    end for
    $A[i] \leftarrow TEMP$
  end for
  return A
end function

Now to have $lce_S(i,j)$ we build the implement the following procedure, where $h = \lfloor log_2(n-j) \rfloor$

function LCE(i,j,h)
  if $A[h][i] == A[h][j]$ **then**
    return 2^h
  else
    if $h \neq 0$ **then**
      return 0
    else
      return LCE$(i,j,h-1)$+LCE$(i+2^{h-1}, j+2^{h-1}, h-1)$
    end if
  end if
end function

Notice that, this procedure work in $O(log(n))$ since the array is length is at most log(n) and we are doing two recursive call with an array one unit smaller each time. Finally to obtain $equalS(i,j,l)$ in cost $O(1)$ we simply check whether $A[\lceil log_2(l) \rceil)][i] == A[\lceil log_2(l) \rceil][j]$ is true.

Notice that all this algorithm work for arrays in which their length n is a power of two. If we have an array that is not of the latter length, we are doing the following: create the same data structure as before but the part of the array that is not in the tree, that it's at most long $2^{i+1} - 2^i - 1$ where $i = \lfloor log_2(n) \rfloor$, is store in simple array. In this case whether we need to check if $A[\lfloor log_2(n-j) \rfloor][i] == A[\lfloor log_2(n-j) \rfloor][j]$, i.e. the longest possible string, we need also to check, manually, whether there is a matching in the array. Last but not least, the calculation of the error probability is exactly the same to the one analysed during the course.