Karp-Rabin fingerprinting on strings

Given a string $S = S[0 \dots n-1]$, and two positions $0 \le i < j \le n-1$, 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, $lce_S(i,j) = max\{l \ge 1 : S[i \dots i+l-1] = S[j \dots j+l-1]\}$. 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
Consider a string S of length n, we call $S[i]$ the i-th element in S starting from 0, and $S[i,l]$ the substring of S with length l starting at i (i.e. $S[i] \cdot S[i+1] \dots S[i+l-1]$, where $\cdot$ is the string concatenation). The idea is to build an array R with the same length of the string S such that for each index i we have that $R[i]$ contains the prefix of S of length $i+1$. $R[0]$ contains the first word of S, $R[1]$ contains the concatenation of the first two wolds of S etc. In general we will have that $R[i] = S[0, i+1]$. This representation has of course the problem to be too much expensive, but we will see later how to really implement the data structure, first have a look at how it works for checking the equality of two substring and for finding the lce_S of two indexes.

Given a pair of indexes i and j, and a length l we want to check whether $S[i,l] = S[j,l]$, looking at R. Considering that $\forall i, l . R[i+l] = S[0,i] \cdot S[i+1,l] = R[i] \cdot S[i+1,l]$, we can prove that

$$\forall i, l \, . \, S[i,l] = R[i+l-1] - R[i-1]$$

where we denote by $\alpha - \beta$ the string α without the prefix β (i.e. $\alpha\gamma - \alpha = \gamma$). Now we can simply check whether $s[i,l] = s[j,l]$ by comparing $R[i+l-1] - R[i-1]$ and $R[j+l-1] - R[j-1]$. The problem of finding the longest common extension is quite simple, it is sufficient to use some form of binary search, exploiting the array R. We use a recursive function in order to find the $lce_S(i,j)$, the input are i, j, the string S and its length n. We consider to have a function $is\_equal(A,x,y,l)$ such that return true if $A[x,l] = A[y,l]$. The main difference between this algorithm and a binary search is that we will continue calling recursively the function with a string of halved length regardless of the result of the equality check.

```
function LCE(S,i,j,n)
    if n = 1 then
        if is_equal(S, i, j, l) then
            return 1
        else
            return 0
        end if
    end if
    l = n/2
    if is_equal(S, i, j, l) then
        return l + LCE(S,i + l,j + l,n − l)
    else
        return LCE(S,i,j,l)
    end if
end function
```

Note that assuming the is_equal function to have constant cost we have that the cost of lce is $O(log(n))$ since at each iteration only a recursive call is reached and the length of the input is halved each time.

In the real implementation, we will use an array H which is the hash fingerprint of R. For each index i $H[i] = h(R[i]) = R[i] \; mod \; p$, of course we don't really built the array R, then we can simply fill $H[i]$ with $S[0,i+1] \; mod \; p$. An implicit assumption in this step (and in Karp-Rabin fingerprint in fact) is to

see each string as a number, in particular we see S as the representation of an integer number with base b (depending on the number of possible word in the alphabet). In the following we will call S the string and $(S)_b$ the number for which S is the representation in base b. We will also distinguish between $S[i]$ the word in position i of S and $S[i]_b$ the number between 0 and $b-1$ which represents. Note that we have to redefine h as $h(\alpha) = (\alpha)_b \; mod \; p$.
Then the number $(S)_b$ will be $S[0]_b \times b^{n-1} + S[1]_b \times b^{n-2} \cdots + S[n-1]_b \times b^0$. With this representation we have the beautiful feature that $(S[0, i+l])_b = (S[0, i] \cdot S[i, l])_b = (S[0, i])_b \times b^l + (S[i, l])_b$, that we can exploit to prove $h(S[i, l]) = H[i + l - 1] - H[i - 1] \times b^l \; mod \; p$.

$$
\begin{aligned}
H[i + l - 1] = {} & h(R[i + l - 1]) \\
& h(S[0, i + l]) \\
& (S[0, i + l])_b \; mod \; p \\
& (S[0, i])_b \times b^l + (S[i, l])_b \; mod \; p \\
& ((S[0, i])_b \; mod \; p) \times b^l + ((S[i, l])_b \; mod \; p) \; mod \; p \\
& h(S[0, i]) \times b^l + h(S[i, l]) \; mod \; p \\
& h(R[i - 1]) \times b^l + h(S[i, l]) \; mod \; p \\
& H[i - 1] \times b^l + h(S[i, l]) \; mod \; p
\end{aligned}
$$

This way the cost of compare two arbitrary substring of the same length $S[i, l]$ and $S[j, l]$ is the cost of acceding to four elements of H plus a constant number of arithmetic operations, i.e. $O(1)$. Since we have found a good approximation of the procedure is_equal, we can compute lce_S with the algorithm shown before in $O(log \; n)$ time.
The space occupied by the data structure is quasi $O(n)$ where n is the length of S. Actually the size of the input in bit is n times the size of the word (i.e. $n \times log \; b$) and the size of the output is n times the size of an element of H (i.e. $n \times log \; p$). Note that we can consider the use of R as a particular case in which p is equal to b^{n+1} (the maximum number that a string of length n with alphabet of size b can represent plus one), in this case the modulus in h is redundant and we have a data structure of size $O(n^2)$.
It is not difficult to see that we can use the compositional property of the h function in order to achieve a cost of $O(n)$ to fill H. The code speaks for itself probably, the idea is to compute $H[0]$ as $S[0] \; mod \; p$, $H[1]$ as $S[1] + S[0] \times b \; mod \; p$ etc. We use only a constant number of operations for each word in S.

$H[0] = S[0] mod p$
for $i \in \{1, \dots n - 1\}$ **do**
    $H[i] = H[i - 1] \times b + S[i] \; mod \; p$
end for

We take p as a random prime number $p \in [2, \cdots, \tau]$, where $\tau > n$. Let's have a look at the collision cases: we have a collision when $S[i, l] \neq S[j, l]$ but $h(S[i, l]) = h(S[j, l])$. This is the same as

$$
\begin{aligned}
& H[i + l - 1] - H[i - 1] \times b^l \; mod \; p = H[j + l - 1] - H[j - 1] \times b^l \; mod \; p \\
& H[i + l - 1] - H[i - 1] \times b^l - (H[j + l - 1] - H[j - 1] \times b^l) \; mod \; p = 0 \\
& h(S[i, l]) - h(S[j, l]) \; mod \; p = 0 \\
& ((S[i, l])_b \; mod \; p) - (S[j, l])_b \; mod \; p) \; mod \; p = 0 \\
& (S[i, l])_b - (S[j, l])_b \; mod \; p = 0 \\
& p \; divides \; (S[i, l])_b - (S[j, l])_b
\end{aligned}
$$

We define $c = (S[i, l])_b - (S[j, l])_b$, we want to count how many bad choices we have for p (how many choices for p such that p divides c). Note that as $(S)_b$ also c is a number representable with a string of length n with alphabet of size b; then $0 \leq c \leq b^n$. Let k be the number of prime number dividing c, then $c = p_1^{i_1} \times p_2^{i_2} \times \cdots \times p_k^{i_k}$ and, since p_x is grater or equal than 2 and i_x is greater or equal than 1 for all x, we can say that $c \geq 2^k$ (as seen in class). Finally $2^k \leq c \leq b^n \leq b^{n \times log \; b}$, and $k \leq n \times log \; b$.Since the possible choices for a prime number in the interval $[2, \cdots, \tau]$ are approximately $\frac{\tau}{ln(\tau)}$, the probability of error is less or equal to $\frac{\#\text{BAD PRIMES}}{\#\text{PRIMES}} = \frac{n \times log \; b}{\frac{\tau}{ln(\tau)}}$. If we choose $\tau \approx n^{a+1} ln(n)$ then we have that the probability of error is less or equal than $\frac{log \; b}{n^a}$.

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 ← NEW Array[log_2(n) + 1]
    TEMP ← NEW Array[n]
    for i IN (0, n) do
        TEMP[i] ← S[i] mod p
    end for
    A[0] ← TEMP
    for i IN (1, log_2(n)) do
        TEMP ← NEW Array[n − i]
        for j IN (0, n − i) do
            TEMP[j] ← A[i − 1][j] + A[i − 1][j + 2^(i−1)] mod p
        end for
        A[i] ← 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 ≠ 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.