CS201c: Programming Evaluation 3 solutions

Let S[1...n] be the given bit string of n bits.

One can view any bit string as an integer (for example, 1011 = 1 + 2.0 + 4.1 + 8.1 = 13).
In general, substring S[i … j] (1 <= i <= j <= n) corresponds to the following integer N (S[i … j]):

$$N (S [i \dots j]) = S[i] + S[i+1].2 + S[i+2].4 + \dots + S[j].2^{j-i}$$

Let p be a random prime in the range [3, B]. (B will be computed later.)

Then, hash of substring S[i...j] equals:

$$h_p (S[i...j]) = [S[i] + S[i+1].2 + S[i+2].4 + \dots + S[j].2^{j-i}] \pmod p$$

Preprocessing step.

First compute a table of 2^i (mod p), where 0<=i<=(p-1).

Pick a random (odd) prime p in [3, B].

Compute the hash values of the n prefixes: S[1 … i] (1 <= i <= n). Let G_p[i] denote the hash value of prefix S[1 … i].

Note that:

$$G_p[i+1] = G_p[i] + (2^{i} * S[i+1]) \pmod p$$

Thus, array G_p can be computed time O(n ([log(p)]^{c1})) time, where c1 is a fixed constant.

Key idea.

Given the array G_p, the hash value of any substring can be computed as follows:

$$2^{i-1} * h_p (S [i \dots j]) = G_p[j] - G_p[i-1] \pmod p$$

Since p is an odd prime, the multiplicative inverse of 2^{i-1} exists and is equal to 2^{p-i} (https://en.wikipedia.org/wiki/Fermat%27s_little_theorem . Alternatively, you can compute the inverse using Euclid's gcd algorithm.). Thus,

$$h_p (S [i \dots j]) = 2^{p-i} * (G_p[j] - G_p[i-1]) \pmod p$$

This computation takes O((log (p))^{c2}) time, where c2 is a fixed constant.

The algorithm is described below (here ln(n) denotes the natural logarithm of n):

Algorithm (in blue):

Set B = 8 * n * ln(n)
Set K = 8 * (ln(n))^2
Initialize output array O[1 … m] to all zeroes.

Repeat the following K times:

    Pick a random odd number r in [3, B]
    Compute the array G_r [1 … n].
    For z = 1 to m:

        For each input pair S[i_z … j_z] and S[k_z … l_z] of substrings:
        Compute hashes q_1 and q_2 of both substrings with respect to h_r.

        If (q_1 != q_2):
            // strings are unequal
            Set O[z]=0

For y = 1 to m:
    print O[y] followed by a new line

Proof of running time. Note that the random odd number r takes O(log (n)) bits and the outermost loop is repeated O ((log (n))^2) time. Fill in the remaining details in analysis of running time yourself.

Proof of correctness:

The above algorithm will make an error in its z-th output line if and only if the two substrings S[i_z … j_z] and S[k_z … l_z] are unequal, but their hashes turn out to be equal for all K random numbers r_1, r_2, ... , r_K. Let:

A_z = absolute value of (N (S[i_z … j_z]) - N (S[k_z … l_z]))

A_z has magnitude at most 2^n. Therefore, A has at most n distinct prime factors (since any prime factor is at least 2). Let T_A be the set of distinct prime factors of A. Thus,

| T_A | <= n

We have set B = 8 n ln (n). Therefore, the number of odd primes in [3 … B] is ~ 8 n.
(https://en.wikipedia.org/wiki/Prime_number_theorem)

Probability (hash of S[i_z … j_z]) does not equal hash of S[k_z … l_z] with respect to r_i)

= Probability (r_i does not divide A)
>= Pr (r_i is a random prime) . Pr (r_i does not divide A | r_i is a random prime)
>= (1 / ln (n)) . (1 - (|T_A| / (8n)))
>= 7 / (8 * ln(n))

Prob (Algorithm makes an error on z-th pair of substrings, when they are unequal)
=
Prob (hash of S[i_z … j_z]) equals hash of S[k_z … l_z] with respect to all of r_1, … r_K)

=
Pr (hash is equal w.r.t. r_1) * Pr (hash is equal w.r.t. r_2) * … * Pr (hash is equal w.r.t. r_K)

<= [1 - 7 / (8*ln(n))] ^(8*(ln (n))^2)
<= [(1 - 1 / (8*ln(n)))^(8 * ln(n))] ^ (ln(n))
<= e^{-ln(n)} (Since (1-1/x)^x ---> e^{-1} as x tends to infinity)
= 1 / n