CS201c Programming Evaluation 3
Instructor: Apurva Mudgal
Topic: String matching using hashing
Due Date: Nov 3, 2019 by 11:59 pm

Problem description.

Input Format:

The first line is a single positive integer n.

The second line contains a bit string x of length exactly n.

The third line contains a second positive integer m.

This is followed by m lines, with p-th (1<=p<=m) line containing four positive integers i_p, j_p, k_p, and l_p between 1 and n:

```
i_1 j_1 k_1 l_1
i_2 j_2 k_2 l_2
.
.
.
i_m j_m k_m l_m
```

Output Format:

The output consists of m lines.

The p-th line has the single integer 1 if and only if substring x[i_p … j_p] is equal to substring x[k_p … l_p]. Otherwise, the p-th line has the single integer 0.

Note. You can use a randomized algorithm i.e., your C++ program can have a random supply of random bits using srand() and rand() functions. *Be sure to initialize your random number generator with a fresh seed using srand() at the start of your C++ program.*

Hint. Find an efficient randomized, hashing scheme for substrings such that two unequal substrings can have the same hash value only with some maximum probability.

Requirements:

1. *Running time.* Worst-case time taken by your C++ program should be

~~O((n+m) \log_2(n))~~ O((n+m) * [\log_2(n)]^c) in the RAM model, where c is a fixed positive constant.

2. *Probability of giving a correct answer.* Further, for each 1 <= p <= m, your C++ should satisfy the following condition:

   Pr [the p-th line of your output is correct] >= 1-(1/n)

3. You cannot use any in-built libraries (including standard template library). All data structures should be implemented in C++ from scratch.

4. **Collaboration is not permitted on this assignment. Your submitted code should be completely your own.**

   See section titled ``Honor Code" in course outline already shared with you.

<u>Example.</u>

<u>Sample Input.</u>

10
1011011011
6
1 4 7 10
2 5 7 10
2 4 5 7
2 4 8 10
3 7 6 10
1 5 6 10

<u>Correct Sample Output.</u>

1
0
1
1
1
0

Note. Since your C++ code is randomized, you are allowed to make an error on each output with probability at most ~~⅙=0.167~~ 0.1.

Note on random prime selection.

Your implementation may require you to generate a random prime in a range [1,n^d], where d is some positive integer. For testing whether the random number generated is in fact a prime, you will need to implement a fast primality test such as Miller-Rabin. The following method will allow you to proceed without implementing a fast primality testing algorithm, such as Miller-Rabin.

It is well known that the number of primes in this range is at least (n^d)/(d ln(n)).
(see: https://en.wikipedia.org/wiki/Prime-counting_function)

Consider the following prime-picking algorithm (in bold):

> **Set S is initially equal to the empty set.**
>
> **K = d ((ln (n))^3) [you can choose a higher power such as 4, 5, etc.]**
> **For i = 1 to K:**
> > **Pick a random integer r_i in the range [3,n^d]**
> > **Add integer r to set S.**

Now, what is the probability that S will contain *at least one* random prime in the range [3, n^d]?

Pr [r_i is a random prime] = (# of primes in [3,n^d]) / (total numbers in [3,n^d])
>= 1 / (d ln (n))
Pr [at least one of r_1, r_2, …, r_K is a random prime] >= 1 - [1 - (1 / (d ln(n)))]^K ---- (*)
(Simplify the right hand side of this equation yourself.)

Thus, you can choose such a random set S, and then run your Prog Eval 3 algorithm |S| times, using r_i as a “supposed prime” for the i-th run. By (*) above, one of these |S| runs will be with a random prime with high probability.