University of Pisa

Department of computer science

Algorithm design

Third hands-on: Karp-Rabin fingerprinting on strings

Domenico Erriquez

1. PROBLEM

Given a string $S \equiv S[0 \dots n-1]$, and two positions $0 \leq i < j \leq n$, the longest common extension $lce(i,j)$ is the length of the maximal run of matching characters from those positions, namely:$if\ S[i] \neq S[j]$ then $lce(i,j) = 0$ otherwise, $lce(i,j) = \max\{l \geq 1 : S[i \dots i+l-1] = S[j \dots j+l-1]\}$. For example if $S = abracadabra$, then $lce(1,2) = 0,\ lce(0,3) = 1$ and $lce(0,7) = 4$. 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:

- $lce(i,j)$ it computes the longest common extension at positions i and j in $O(\log n)$ time.
- $equal(i,j,l)$ 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 one-time preprocessing is performed, and then many online queries are to be answered on the fly.]

2. SOLUTION

The Karp-Rabin fingerprint on a string S is calculated as follows:

$$f(s[0 \dots n-1]) = s_0 d^{n-1} + s_1 d^{n-2} + \dots + s_{n-2} d + s_{n-1} \bmod p$$

Where $s_0, s_1 \dots s_n$ are the character that compose the string S, p is a prime number, d is a fixed constant (often chosen to be a large prime number) and n is the size of the string.

The first step to do to build a data structure for S based on Karp-Rabin fingerprint is to build an array where we will store the values of the hash function for each substring $s[i \dots n-1],\ 0 \leq i \leq n-1$ To build this array we take advantage of the rolling hash property, which allow us to build the hash function $h[i]$ in constant time exploiting using the hash function $h[i-1]$. The only hash function to compute is the first one $h[0]$, for the rest as said before, the cost to compute them is constant.

The hash function for the substring $s[i]$ is computed in this way:

$$h[i] = \left(h[i-1] - \left(\left(d^{n-i} * s_{i-1}\right) mod\ p\right)\right)$$

2.1. Equal (i, j, l)

Given the positions i and j, and the length of the substring l, the objective is to determine whether the two substrings of length l, starting at positions i and j respectively, are equivalent. Specifically, this requires verifying the equality of the substrings $s[i \dots i+l-1] = s[j \dots j+l-1]$. To accomplish this, the values of the hash functions calculated for these two substrings will be compared. The first step in this process is to establish a method for calculating these hash functions. Starting with the substring $s[i \dots i+l-1]$, we can take advantage of the hash functions in the array h, more specifically of the functions $h[i]$ and $h[i+l]$. We can notice that:

$$h[i] = f(s[i \dots n-1]) = s_i d^{n-(i+1)} + s_{i+1} d^{n-(i+2)} + \cdots + s_{n-2} d + s_{n-1}\ mod\ p$$
$$h[i+l] = f(s[i+l \dots n-1]) = s_{i+l} d^{n-(i+l+1)} + s_{i+l+1} d^{n-(i+l+2)} + \cdots + s_{n-2} d + s_{n-1}\ mod\ p$$

We can subtract these two functions to obtain the hash function for the substring $s[i \dots i+l-1]$

$$h[i] - h[i+l] = s_i d^{n-(i+1)} + s_{i+1} d^{n-(i+2)} + \cdots + s_{i+l-1} d^{\,n-(i+l)}\ mod\ p$$
$$= f(s[i \dots i+l-1]) * d^{n-(i+l)}\ mod\ p$$

However, this subtraction would not give us the correct hash value of the desired substring, because the power of the prime number is based on the position of the character in the original string, not the substring.

Since we want to compute $f(s[i \dots i+l-1])$ we have to divide the result of $h[i] - h[i+l]$ by $d^{n-(i+l)}$, to do this we can multiply directly $f(s[i \dots i+l-1])$ for the inverse of $d^{n-(i+l)}$ which is $d^{n-(i+l)^{-1}}$ so we will have that:

$$f(s[i \dots i+l-1]) = h[i] - h[i+l] * d^{n-(i+l)^{-1}} \bmod p$$

To avoid recomputing for each query the multiplicative inverse of $d^{n-(i+l)}$, we may precompute all of them $d^{n-k-1}\ 0 \le k \le n-1$, in an array so that the equal query can be done in constant time.

With the same approach we can compute the hash function for the substring $s[j \dots j+l-1]$.

2.2. lce (i, j, l)

The algorithm for computing the Longest Common Extension (lce) between indices i and j in a given string S, can be performed in $O(\log n)$ time. It is essential to note that the result can be at most equal to the minimum value between $j-i$ and $n-j$, where n is the length of the string S. We will call $l = \min((j-i), (n-j)$ the maximum common extension possible.

A binary search strategy can be employed to solve this problem.

```
def lce(i, j, l):
    if l <= 0 or i>=j or j>=n-1:
        return 0
    if equal(i, j, l):
         return l + lce(i+l, j+l, l)
    else:
        return lce(i, j, l/2)
```

2.3. Error probability

We know that the function $equal(i, j, l)$ gives us a wrong answer when the hash function f has a collision. So, we have to estimate the probability that $f(s[i \dots i+l]) = f(s[j \dots j+l])$ when $s[i \dots i+l] \neq s[j \dots j+l]$.

$$\Pr\left[f(s[i \dots i+l]) = f(s[j \dots j+l] \wedge s[i \dots i+l] \neq s[j \dots j+l]\right] \leq \frac{\#\ bad\ primes}{\#\ all\ primes} \leq \frac{n}{\tau/\ln \tau}$$

If we choose $\tau \sim n^{c+1} \ln n$ for a constant $c > 0$ we have

$$\Pr[error] \leq \frac{1}{n^c}$$

So the probability of error for the function $equal()$ is $\frac{1}{n^c}$, while the probability of error for the $lce()$ function is $\frac{1}{n^c} \log n$, this is because the lce function calls the function $equal()$ $\log n$ times.