Longest common substring using rolling hash approach

Reading time: 30 minutes | Coding time: 15 minutes

A substring is a contiguous sequence of characters within a string.For example,open is a substring of opengenus. Here,we have presented an approach to find the longest common substring in two strings using rolling hash. **Rolling hash** is used to prevent rehashing the whole string while calculating hash values of the substrings of a given string. In rolling hash,the new hash value is rapidly calculated given only the old hash value.Using it, two strings can be compared in constant time.

To understand the basic approaches to solve this problem, go through this article Longest Common Substring in two strings by Ashutosh Singh.

> *With the rolling hash technique, we will be able to solve this problem in O(N * log(N)^2) time and O(N) space.*

Examples

```
str1 = opengenus
str2 = genius
Output = gen
```

```
The longest common substring of str1(opengenus) and str2(genius)

str1 = carpenter
str2 = sharpener
Output = arpen
The longest common substring of str1(carpenter) and str2(sharpen
```

Implementation

Firstly we need to calculate polynomial hashes on prefixes of strings **A** and **B**. Suppose that we have found the largest common substring of length len, starting with the position pos of any of the strings. Then the common substring is any substring of length ** len-1, len-2, ..., 1, ** starting with **pos**, but len + 1 will be not a common substring. We see that the **binary search** conditions are satisfied.

Pseudo Code for binary search:-

```
 l = 0 , r = min(s1.length(),s2.length())
while (l <= r){
    mid = l + (r-l)/2
    if(p(s1, s2, mid)) // p(s1,s2,len) checks for common substrin
        l = mid + 1
    else
        r = mid - 1
}
return l-1
```

At each iteration of the search, we add all hashes of the line mid of the string A to the vector, sort it, and then go through the hash of the substrings with length mid of the string B and search them in sorted array of hashes substrings of string A with length mid.

To find the hash of a substring,we find the hash of previous substring and calculate the hash of next substring.

Pseudo Code for rolling hash

```
// Part of OpenGenus IQ
// computes the hash value of the input string s
long long compute_hash(string s) {
    const int p = 31;   // base
    const int m = 1e9 + 9; // large prime number
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
// finds the hash value of next substring given nxt as the ending
// and the previous substring prev
long long rolling_hash(string prev,char nxt)
{
   const int p = 31;
   const int m = 1e9 + 9;
   long long H=compute_hash(prev);
   long long Hnxt=( ( H - pow(prev[0],prev.length()-1) ) * p + (i
   return Hnxt;
}
```

You can refer to this **article** to learn more about rolling hashes.

Let's consider an example, we want to find the longest commmon substring between the strings **iit** and **iiitian**.Applying the rolling hash formula,the hash value of string iit would be **7955**.So **pref1[3]=7955** which refers to the prefix hash upto 3 letters from start of the string **iit** .

When we find the hash values of all 3-letters substrings of **iiitian**, the hash value of the first substring **iii** would be **7944**,so **pref2[3]=7944** when our considered string starts from first letter and that when the string under consideration starts from second letter would be the hash corresponding to the next substring iit ,i.e, **7955**.Therefore **pref2[3]=7955**.We can easily find out using binary search in the **pref2[]** array during the second iteration we get equal hash to that of the considered hash of string **iit** which is of maximum length ,i.e., equal to the length of smallest substring(3 in this case), therefore **iit** would be the required answer.

Following is the complete implementation in cpp using structure storing the prefixes of the strings:-

```
#include <stdio.h>
#include <cassert>
#include <algorithm>
#include <vector>
#include <random>
#include <chrono>
#include <string>

typedef unsigned long long ull;

// Generate random base in (before, after) open interval:
int gen_base(const int before, const int after) {
    auto seed = std::chrono::high_resolution_clock::now().time_si
    std::mt19937 mt_rand(seed);
    int base = std::uniform_int_distribution<int>(before+1, after
    return base % 2 == 0 ? base-1 : base;
}

struct RollingHash {
    // -------- Static variables --------
    static const int mod = (int)1e9+123; // prime mod of polynomi
    static std::vector<int> pow1;        // powers of base modulo
    static std::vector<ull> pow2;        // powers of base modulo
    static int base;                     // base (point of hashin

    // --------- Static functons --------
    static inline int diff(int a, int b) {
        // Diff between `a` and `b` modulo mod (0 <= a < mod, 0 <
        return (a -= b) < 0 ? a + mod : a;
    }
```

```
    // -------------- Variables of class -------------
    std::vector<int> pref1; // Hash on prefix modulo mod
    std::vector<ull> pref2; // Hash on prefix modulo 2^64

    // Cunstructor from string:
    RollingHash(const std::string& s)
        : pref1(s.size()+1u, 0)
        , pref2(s.size()+1u, 0)
    {
        assert(base < mod);
        const int n = s.size(); // Firstly calculated needed powe
        while ((int)pow1.size() &lt= n) {
            pow1.push_back(1LL * pow1.back() * base % mod);
            pow2.push_back(pow2.back() * base);
        }
        for (int i = 0; i < n; ++i) { // Fill arrays with polynom
            assert(base > s[i]);
            pref1[i+1] = (pref1[i] + 1LL * s[i] * pow1[i]) % mod;
            pref2[i+1] = pref2[i] + s[i] * pow2[i];
        }
    }

    // Rollingnomial hash of subsequence [pos, pos+len)
    // If mxPow != 0, value automatically multiply on base in nee
    inline std::pair<int, ull> operator()(const int pos, const in
        int hash1 = pref1[pos+len] - pref1[pos];
        ull hash2 = pref2[pos+len] - pref2[pos];
        if (hash1 &lt 0) hash1 += mod;
        if (mxPow != 0) {
            hash1 = 1LL * hash1 * pow1[mxPow-(pos+len-1)] % mod;
            hash2 *= pow2[mxPow-(pos+len-1)];
        }
        return std::make_pair(hash1, hash2);
    }
};

// Init static variables of RollingHash class:
int RollingHash::base((int)1e9+7);
```

```cpp
std::vector<int> RollingHash::pow1{1};
std::vector<ull> RollingHash::pow2{1};

int main() {
    // Input:
    int n;
    scanf("%d", &n);
    char buf[1+100000];
    scanf("%100000s", buf);
    std::string a(buf);
    scanf("%100000s", buf);
    std::string b(buf);

    // Calculate max neede power of base:
    const int mxPow = std::max((int)a.size(), (int)b.size());

    // Gen random base of hashing:
    RollingHash::base = gen_base(256, RollingHash::mod);

    // Create hashing objects from strings:
    RollingHash hash_a(a), hash_b(b);

    // Binary search by length of same subsequence:
    int pos = -1, low = 0, high = std::min(a.size(), b.size())+1;
    while (high - low > 1) {
        int mid = (low + high) / 2;
        std::vector<std::pair<int,ull>> hashes;
        for (int i = 0; i + mid <= n; ++i) {
            hashes.push_back(hash_a(i,mid,mxPow));
        }
        std::sort(hashes.begin(), hashes.end());
        int p = -1;
        for (int i = 0; i + mid <= n; ++i) {
            if (std::binary_search(hashes.begin(), hashes.end(),
                p = i;
                break;
            }
        }
        if (p >= 0) {
```

```
            low = mid;
            pos = p;
        } else {
            high = mid;
        }
    }
    assert(pos >= 0);
    // Output answer:
    printf("%s", b.substr(pos, low).c_str());

    return 0;
}
```

Complexity

- Worst case time complexity: `O(n*log(n)^2)`
- Average case time complexity: `O(n*log(n)^2)`
- Best case time complexity: `O(n*log(n)^2)`
- Space complexity: `O(n)`

Sorting the hashes require **O(nlog(n))** time and binary search requires another **O(log(n)** time,therefore, the total time complexity of finding the longest common substring using rolling hash approach would be **O(n * log(n)^2)**.Since the prefixes and hashes are to be stored in arrays/vectors, therefore, space complexity would be **O(n)**.

Further reading

- Rolling hash technique by Ashutosh Singh
- Longest Common Substring in two strings by Ashutosh Singh
- Practice problem ADAPHOTO on SPOJ

With this, you will have a strong idea of the problem. Enjoy.

Ashutosh Singh

Intern at OpenGenus | Pursuing B. Tech in Computer Science at Indian Institute of Information Technology (IIIT) Sricity

Read More

OpenGenus Foundation

Tags

Algorithms | String Algorithms

Start Discussion 0 replies

Start Discussion 0 replies

Start Discussion 0 replies

— OpenGenus IQ: Learn Computer Science —

Algorithms

Transitive Closure Of A Graph using Floyd Warshall Algorithm

Gale Shapley Algorithm for Stable Matching problem

Minimum Increment and Decrement operations to make array elements equal

See all 297 posts →

CS HISTORY

Computer History on 16th February

Today (16th February) is a major day in Computer History as today the World's first Bulletin board system was launched and Ian Clarke was born who is behind FreeNet. These events shaped the communication on the

Internet.

UE KIAO

MACHINE LEARNING (ML)

Tensorflow.js: Machine Learning through JavaScript

Tensorflow.js is an open-source library with which we can implement machine learning in the browser with the help of JavaScript. It is powered by WebGL and provides a high-level layers API for defining models, and a low-level API for linear algebra and automatic differentiation.

PRANJAL SRIVASTAVA