inzva Algorithm Programme 2018-2019

Bundle 14

Algorithms - 5

Editor
Mehmet Altuner

Reviewer
Osman Karaketir

Contents

1 Hashing

We can find a unique integer representation for each string. This process is called Hashing. There are a lot of hashing functions, we look into one of them: Polynomial rolling hash.

Hash of string s with length k + 1 is: $hash(s) = (s[0] * p^{(k)} + s[1] * p^{(k-1)} + ... + s[k] * p^0) \pmod b$

Choose p as a prime number that is bigger than the size of the alphabet, choose m as a large prime number to ensure that the hash value of the string is unique for that particular string.

Popular p and m values are $p = 31$ and $m = 1e9 + 7$.

```
long long int hash_func(string text){
        int p = 271; //
        long long int result_hash = 0;
        /*
        a prime number that is bigger than the size of
        the alphabet. Alphabet size for char is 256.
        */
        long long int m = 1e9 + 7; // a very large number

        long long int p_pow = 1;

        for(int i= text.size() - 1; i>=0; i--){
                result_hash += (s[i]*p_pow) %m;
                result_hash = result_hash %m
                p_pow = p * p_pow;
        }

        return result_hash;
}
```

We will make use Polynomial rolling hash in Rabin-Karp Algorithm for pattern finding.

2 String Matching Algorithms

2.1 Basics

In string matching, our goal is to find a pattern in a given text. Sometimes we are asked only to find out if there is at least one matching substring or how many matching substring there are.

2.2 Naive Approach

In naive approach, we simply traverse through all possible substrings of length `m` of the given text of length `n`. We check each substring for whether it is matching with the given pattern or not. Therefore, our complexity is `O(m*n)`.

```
bool find(string pattern, string text){
    for(int i=0; i<text.size()-pattern.size(); i++){
        for(int j=0; j<pattern.size(); j++){
            if(text[i+j] != pattern[j]) break;
            if(j == pattern.size()-1) return true;
        }
    }
    return false;
}
```

In the code above, we make sure that the substring that starts with `text[i]` and has the length of `m` is equal to the pattern. One can observe that while traversing, we are processing characters of text multiple times. Popular string matching algorithms are focused on reducing this cost.

What if our pattern was something like `abcdef` ? No `re-occuring` characters or substrings. In such cases, the code below will work with a complexity of `O(n)`.

```
bool find(string pattern, string text){
    int i = 0, matched_so_far = 0;
    while(i < text.size()){
        if(text[i] == pattern[matched_so_far]){
            if(matched_so_far == pattern.size()-1)
                return true;
            i++;
            matched_so_far++;
        }else{
            i++;
            matched_so_far = 0;
        }
    }
    return false;
}
```

Why is the complexity for finding patterns that have no re-occurent substrings `O(n)`? Because we only need to process each character just for once.

Let's say that our pattern is `abcdef` and we are in such a state that so far we have reached `abc` in our text. If the current character that we will check is not `d`, we do not need to start over because we have already matched `abc` and we know that our starting character `a` does not exist in the text except the last time we matched `a`.

So this observation leads us to another concept, Prefix Function.

2.3 Prefix Function and Proper Prefixes

As we have discussed before, string matching algorithms work in the worst way in the case of re-occurent substrings. In a more formal way, we can define these substrings as `prefixes that are also the suffixes` of a string (proper prefixes). Prefix `abc` in text `abcdeabc` is an example of proper prefixes since `abc` is a prefix and a suffix at the same time.

You are given a string s and the prefix function is an array that holds the length of the longest proper prefix of an interval `[0, i]` which is also the suffix of this substring.

$$\pi[i] = \max_{k=0\ldots i} \{k : s[0 \ldots k-1] = s[i-(k-1) \ldots i]\}$$

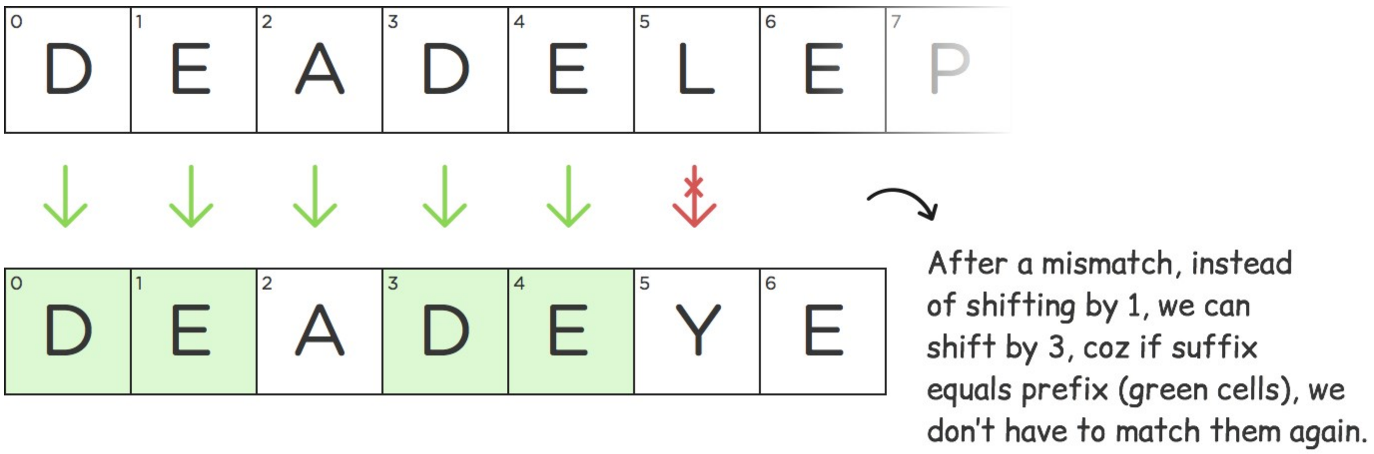
For example, for the pattern `abcdeabc`, the prefix function array will be [0, 0, 0, 0, 0, 1, 2, 3]

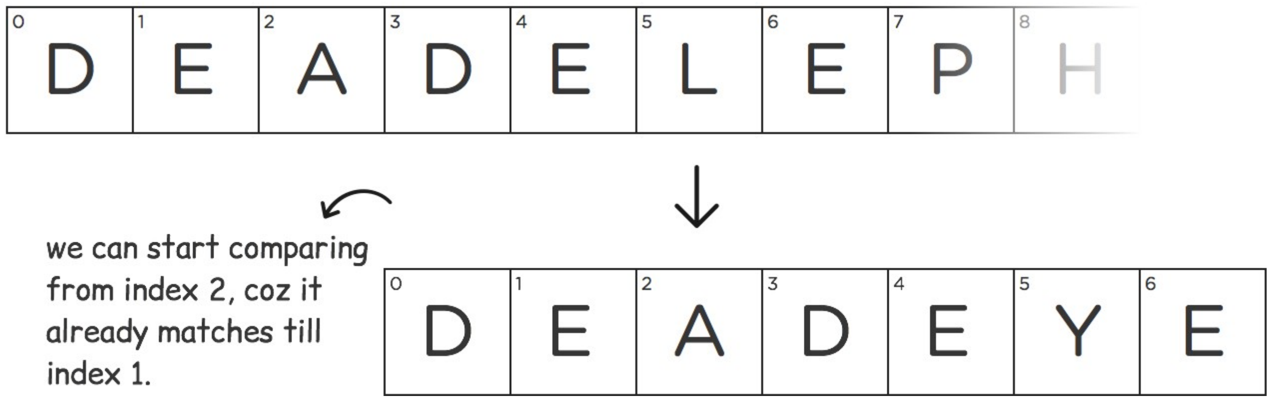
Most of the string matching algorithms are using prefix function to minimize the cost of the algorithm.

```
vector <int> prefix_array;
void calculatePrefixArray(string pattern){
        int len = 0; // length of the current proper prefix
        int i = 1;   // pointer that traverse on pattern
        prefix_array.resize(pattern.size());
        prefix_array[0] = 0;

        while(i < pattern.size()){
                if(pattern[i] == pattern[len]){
                        prefix_array[i] = ++len;
                        i++;
                }else{
                        if(len != 0){ // if i had a proper prefix before
                                len = prefix_array[len-1];
                        }else{
                                prefix_array[i] = 0;
                                i++;
                        }
                }
        }
}
```

2.4 KMP

KMP string matching algorithm is performed very similarly to the naive approach. In the case of a mismatch while checking a substring for whether it is equal to the pattern or not, we do not start over. Instead, we start from the head of the proper prefix we have matched so far if there is any. Because if we have a proper prefix, means that we have a suffix that is equal to the prefix. The pattern we am looking for might be starting with this very suffix we just found.


0 D 1 E 2 A 3 D 4 E 5 L 6 E 7 P
0 D 1 E 2 A 3 D 4 E 5 Y 6 E
After a mismatch, instead of shifting by 1, we can shift by 3, coz if suffix equals prefix (green cells), we don't have to match them again.

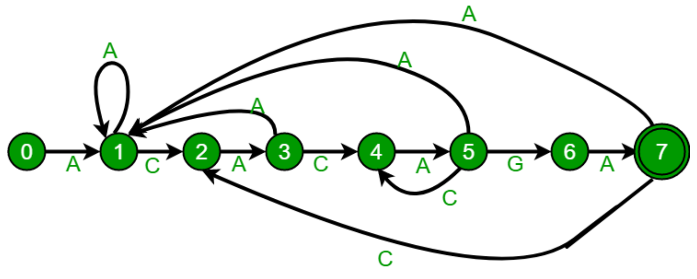

0 D 1 E 2 A 3 D 4 E 5 L 6 E 7 P 8 H
we can start comparing from index 2, coz it already matches till index 1.
0 D 1 E 2 A 3 D 4 E 5 Y 6 E

```
bool KMP(string text, string pattern){
    calculatePrefixArray(pattern);
        int i = 0; // pointer to traverse on text
        int j = 0; // pointer to traverse on pattern

        while(i < text.size()){
                if(j==pattern.size()) return true;

                if(text[i] == pattern[j]){
                        i++;
                        j++;
                }else{
                        if(j != 0){
                                /*
                                if i had a matching sequence lately
                                do not go to the beginning
                                go to the beginning of the proper prefix
                                and do not increment i
                                */
                                j = prefix_array[j-1];
                        }else{
                                // increment i
                                i++;
                        }
                }
        }

        return j == pattern.size();
}
```

The time complexity for KMP is `O(m+n)` and memory complexity is `O(n)`.

2.5 String Matching Automata

In string matching automata, we precalculate a pattern and build a 2D array that represents a Finite Automata. We will search for the pattern in the text according to this array. It will tell us which state to go next, according to the current state we are in. Calculation of the state table is the tricky part of this algorithm. This algorithm is useful while dealing with the relatively small sized pattern texts and alphabets, and large size of input texts.



	character			
state	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The size of the state array will be `size of the alphabet * (size of the pattern + 1)`. We will be on `state(0)` at the beginning.

Let's say we are at the state `t` and checking for the letter `l` of the alphabet. Now, what we should do is consider the string `pat[0...t-1] + l`. Let's call this new string `newPattern`. Our next state will be the length of the longest proper prefix of the string `pat + '' + newPattern`. The character '' is placed in the middle of two strings to distinguish them (make sure '' is not in the given alphabet or choose one outside from the alphabet).

```
int state_table[N+1][M];
// N=length of the input, M=size of the alphabet
void create_state_table(string pattern, string alphabet){
        string pat = "";
        for(int i=0; i<pattern.size()+1; i++){
                for(int j=0; j<alphabet.size(); j++){
                        string tmp = pat;
                        pat += alphabet[j];
                        //cerr << pat << endl;
                        int new_state=0, pattern_ptr=0;

                        prefix_array.clear();
                        calculatePrefixArray(pattern + '#' + pat);

                        new_state = prefix_array[pattern.size() + i + 1];

                        state_table[i][j] = new_state;
                        pat = tmp;
                }
                if(i < pattern.size()) pat += pattern[i];
        }
}
```

Time complexity for building the state table is `O(n*m)` (n is the size of the pattern and m is the size of the alphabet).

While searching for the pattern in the text, we simply follow our state table. If we reach the last state, it means we have found the pattern.

```
bool finite_automata(string pattern, string text, string alphabet){
        create_state_table(pattern, alphabet);
        int current_state = 0;
        int last_state = pattern.size();

        for(int i=0; i<text.size(); i++){
                current_state = state_table[current_state][s[i]];
                if(current_state == last_state) return true;
        }
        return false;
}
}
```

2.6 Rabin-Karp Algorithm

This algorithm is the only string matching algorithm among the ones we are discussing that does not use prefix function.

By the definition of hash, a string's hash value can be considered as unique. So if two hash values of two strings are equal, these two strings are same. By using this, we can tell if two substrings are matching or not. There is very very small probabilty(Birthday Paradox[8]) of that different strings have same hash value, but we can assume it is impossible.

```
bool is_same(string& a, string& b){
    return hash(a) == hash(b);
}
```

How can we use this in string matching? Let's say we are looking for pattern p in text t. Length of these strings are n and m, respectively. We will traverse in each substring of p of length n in $O(m)$ and check whether they are equal or not. In naive solution, this goes to complexity of $O(n*m)$. But thanks to the properties of hashing we discussed above, checking two string complexity is reduced to $O(1)$, so total complexity is with hashing average $O(m+n)$ but worst case is $O(n*m)$[9].

If we know hash(str[i..j]) we can easily calculate hash(str[i + 1..j+ 1]) with using properties of rolling hash.

remember rolling hash: $hash(s) = (s[0]*p^{(k)} + s[1]*p^{(k-1)} + ... + s[k]*p^0) \pmod b$

$hash(str[i....i+k]) = (s[i]*p^{(k)} + s[i+1]*p^{(k-1)} + ... + s[i+k]*p^0) \pmod b$
$hash(str[i+1....i+k+1]) = (s[i+1]*p^{(k)} + s[i+1]*p^{(k-1)} + ... + s[i+k+1]*p^0) \pmod b$

We can see that:
$hash(str[i+1....i+k+1]) = (hash(str[i....i+k]) - (s[i+1]*p^{(k)}))*p + s[i+k+1]*p^0 \pmod b$

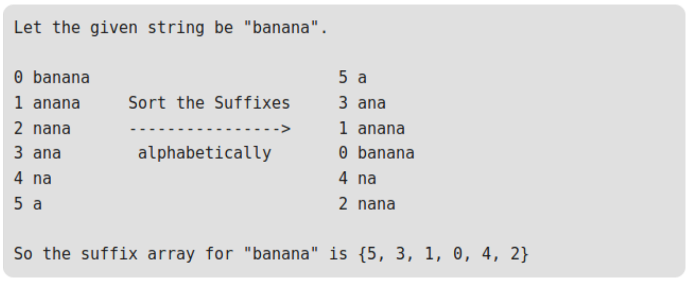
```cpp
bool rabin_karp(string& text, string& pattern){

        int m = text.size();
        int n = pattern.size();
        int p = 271; //same p using in hash_func
        long long int mod = 1e9 + 7 //same mod using in hash_func
        long long int hash_substring, hash_pattern;
        hash_substring = hash_func(string(text.begin(), text.begin() + n));
        hash_pattern = hash_func(pattern);

        if(hash_substring == hash_pattern)
                return true;

        for(int i = n; i<m; i++){
                hash_substring =
                    ((hash_substring - (text[i - n] * p^(n-1)))*p + text[i] * p^0) % mod;
                if(hash_substring == hash_pattern)
                        return true;
        }

        return false;
}
```

2.7 Suffix Array

Suffix Array is a very efficient data structure that is being widely used in compression, bioinformatics and in areas that string manipulation is used.

The problem we will be interested in is pattern finding.

Suffix Array is a sorted array consisting of all suffixes of a string.



```
Let the given string be "banana".

0 banana                          5 a
1 anana     Sort the Suffixes     3 ana
2 nana      ---------------->     1 anana
3 ana        alphabetically       0 banana
4 na                              4 na
5 a                               2 nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

2.7.1 Naive Approach

Naive Approach has a complexity of $O(n^2 * log(n))$ and n is the length of the string. In naive approach, we simply traverse through all suffixes of the string and store them in an array. While sorting this array, we get $O(n^2 * log(n))$ since comparing two strings is $O(n)$.

2.7.2 $O(n * log(n) * log(n))$ Algorithm

One can observe that the suffixes we are trying to sort belong to the same string. By using this fact, we will first sort suffixes by their first character, and then we will sort suffixes of length two, and then of length four, and the length of the suffixes we will compare will grow exponentially. We will be performing sorting operation in $O(log(n))$ step, therefore we have a complexity of $O(n*log(n)*log(n))$.

```
#define MAXN 65536
#define MAXLG 17
char A[MAXN];
struct entry
{
    int nr[2];
    int p;
} L[MAXN];
int P[MAXLG][MAXN];
int N,i;
int stp, cnt;
int cmp(struct entry a, struct entry b){
    return a.nr[0]==b.nr[0] ?(a.nr[1]<b.nr[1] ?1: 0): (a.nr[0]<b.nr[0] ?1: 0);
}
int main()
{
    gets(A);
    for(N=strlen(A), i = 0; i < N; i++)
    P[0][i] = A[i] - 'a';
    for(stp=1, cnt = 1; cnt < N; stp++, cnt *= 2)
    {
        for(i=0; i < N; i++){
            L[i].nr[0]=P[stp- 1][i];
            L[i].nr[1]=i +cnt <N? P[stp -1][i+ cnt]:-1;
            L[i].p= i;
        }
        sort(L, L+N, cmp);
        for(i=0; i < N; i++)
            P[stp][L[i].p] =i> 0 && L[i].nr[0]==L[i-1].nr[0] && L[i].nr[1] == L[i- 1].nr[1] ?
    }
    return 0;
}
```

2.7.3 Pattern Finding

Since our suffix array is sorted, we can perform search operation efficiently by using binary search.

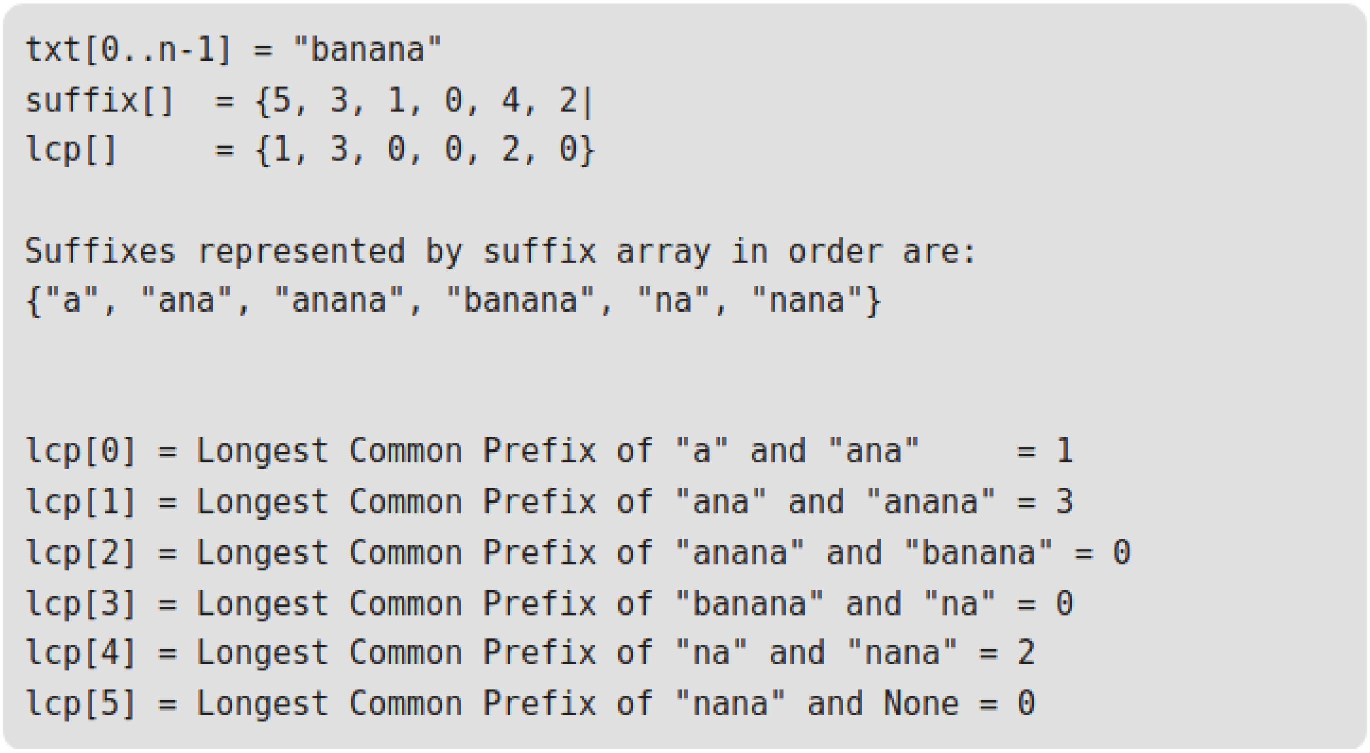
```cpp
    // A suffix array based search function to search a given pattern
    // 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat);  // get length of pattern, needed for strncmp()

    // Do simple binary search for the pat in txt using the
    // built suffix array
    int l = 0, r = n-1;  // Initilize left and right indexes
    while (l <= r)
    {
        // See if 'pat' is prefix of middle suffix in suffix array
        int mid = l + (r - l)/2;
        int res = strncmp(pat, txt+suffArr[mid], m);

        // If match found at the middle, print it and return
        if (res == 0)
        {
            cout << "Pattern found at index " << suffArr[mid];
            return;
        }

        // Move to left half if pattern is alphabtically less than
        // the mid suffix
        if (res < 0) r = mid - 1;

        // Otherwise move to right half
        else l = mid + 1;
    }

    // We reach here if return statement in loop is not executed
    cout << "Pattern not found";
}
```

Note that in the above implementation, `suffArr[i]` corresponds to *i′th* suffix of a string. For example let that string be ”*banana*” and *i* be one.

`suffArr[i] = 3 string[3..n] = "ana"`

2.8 LCP Array

LCP means longest common prefix. `LCP[i]` holds the length of the longest common prefix of `suffArr[i]` and `suffArr[i+1]`.



```
txt[0..n-1] = "banana"
suffix[]  = {5, 3, 1, 0, 4, 2|
lcp[]     = {1, 3, 0, 0, 2, 0}

Suffixes represented by suffix array in order are:
{"a", "ana", "anana", "banana", "na", "nana"}


lcp[0] = Longest Common Prefix of "a" and "ana"     = 1
lcp[1] = Longest Common Prefix of "ana" and "anana" = 3
lcp[2] = Longest Common Prefix of "anana" and "banana" = 0
lcp[3] = Longest Common Prefix of "banana" and "na" = 0
lcp[4] = Longest Common Prefix of "na" and "nana" = 2
lcp[5] = Longest Common Prefix of "nana" and None = 0
```

There are many ways to construct a LCP Array but we are going to discuss the one that uses precalculated suffix array (Kasai's Algorithm).

```
int lcp(int x, int y){
    // lcp's of substring s[x..n] and s[y..n]
    int res = 0;
    for(int i=logn; i>=0; i--){
        if(P[i][x] == P[i][y]){
            res += pow(2, i);
        }
        x += pow(2, i);
        y += pow(2, i);
        // move to the next characters
    }
    return res;
}
```

The algorithm above has a complexity of $O(log(n))$ and should be called for each suffix so its total complexity is $O(n * log(n))$.

3 References

References

[1] https://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/

[2] https://cp-algorithms.com/string/prefix-function.html

[3] https://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/

[4] https://cp-algorithms.com/string/string-hashing.html

[5] https://www.geeksforgeeks.org/kasais-algorithm-for-construction-of-lcp-array-from-suffix-array/

[6] https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/

[7] https://cp-algorithms.com/string/rabin-karp.html

[8] https://en.wikipedia.org/wiki/Birthday$_paradoxhttps : //en.wikipedia.org/wiki/Rabin-Karp_algorithm$