Competitive Programming and Contests

String algorithms: Rabin-Karp and Knuth-Morris-Pratt

Federico Ramacciotti

`https://github.com/0xfederama/cpc-exam-2022`

01/01/2022

Academic Year 2021/2022

Contents

1 Introduction **2**

2 Rabin-Karp algorithm **3**
2.1 The algorithm 3
2.2 Example 4
2.3 Time analysis 4

3 Knuth-Morris-Pratt algorithm **6**
3.1 The algorithm 6
3.2 Example 7
3.3 Time analysis 8

4 Problems **9**
4.1 SPOJ - A needle in the haystack 9
4.1.1 Time analysis 9
4.2 TopCoder - Running Letters 10
4.2.1 Time analysis 11
4.3 Codechef - Pizza Tossing 12
4.3.1 Time analysis 13
4.4 SPOJ - Extend to palindrome 14
4.4.1 Time analysis 15
4.5 SPOJ - Period 16
4.5.1 Time analysis 17
4.6 CodeForces - Password 18
4.6.1 Time analysis 18
4.7 SPOJ - Findsr 19
4.7.1 Time analysis 19

Chapter 1

Introduction

Exact string matching is a trivial problem that has many use cases in fields like text, image, signal and speech processing. It can be solved using several different algorithms based on different approaches. In this paper I am going to present one of the simplest algorithms and one of the fastest algorithms addressing this problem.

Statement
Given a text T and a pattern P, determine whether the pattern appears in the text.

Naive solution
The naive solution consists of simply aligning the pattern with the left end of the text and comparing them. If a match is found, report it; otherwise, shift the text and restart the process until the pattern is not longer than the remaining part of the text.
This solution is simple to understand and to implement but too slow to use it. In fact, in the worst-case scenario, the algorithm performs exactly $O(m(n-m+1))$ comparisons (with $n = |T|$ and $m = |P|$).

Other solutions
Other algorithms usually pre-process the pattern in order to make assumptions and shorten the matching time, as we will see later in the paper. The first algorithm, Rabin-Karp, has a matching time of $O(m(n-m+1))$ with better performance in practical usage and the second algorithm, Knuth-Morris-Pratt, has a matching time of $O(n)$. Both of these algorithms have a pre-processing time of $O(m)$.

Chapter 2

Rabin-Karp algorithm

The Rabin-Karp algorithm uses the **hash function** to compare two strings: if the hash function is well-designed, two strings are equal if (and only if) their hash values are equal. The hash function used in the algorithm can be a mathematical module using a prime number q. The algorithm computes, at each position of the text, the hash value of a string starting at that position with the same length as the pattern. If this hash value equals the hash value of the pattern, it compares the strings to check if the result is not a false positive, since the reverse of the hash function does not give a 100% correct correct match.

2.1 The algorithm

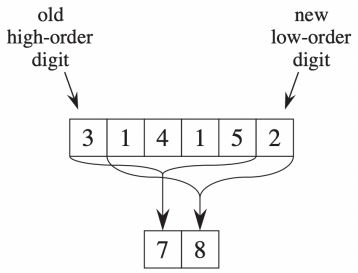
```
RABIN-KARP-MATCHER(T, P, d, q)
1     n = T.length
2     m = P.length
3     h = d^(m-1) mod q
4     p = 0
5     t_0 = 0
6     for i = 1 to m           // pre-processing
7         p = (dp + P[i]) mod q
8         t_0 = (dt_0 + T[i]) mod q
9     for s = 0 to n - m       // matching
10        if p == t_s
11            if P[1..m] == T[s + 1..s + m]
12                print "Pattern occurs with shift" s
13        if s < n - m
14            t_s+1 = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Listing 1: Rabin-Karp pseudocode[1]

The procedure works as follows, interpreting all characters as radix-d digits. Line 3 initializes h to the high-order digit in an m-digit window. Lines 4-8 calculate p as $P[1..m]$ mod q and t_0 as $T[1..m]$ mod q. Every time line 10 is executed the for loop maintains the invariant $t_s = T[s+1..s+m]$ mod q. If line 10 is true, checks for a false positive evaluating $P[1..m] = T[s+1..s+m]$. If $s < n - m$ is true, in order to ensure that the loop invariant holds, it executes line 14, computing t_{s+1} mod q in constant time.

2.2 Example

In order to see why the Rabin-Karp algorithm is, in practical use, better than the naive solution, let's see an example. Suppose we have an alphabet of numbers, $\Sigma = \{0, 1, ..., 9\}$, so that we can consider a string like `25` as the decimal number 25. Suppose also that our pattern that we are searching for is `13658` and we are working mod 13, so our pattern's hash value is 13658 mod 13 = 8. In the text, we are now comparing the pattern to the string `31415`, viewed as the decimal number 31415. The high-order digit is 3 and 31415 mod 13 = 7. After comparing this hash value to our pattern's hash value (and checking that it's not a false positive), we shift the string to the next character, adding the new low-order digit `2` to form the string `14152`. Now, in order to find the new hash value, we don't need to recompute the entire module operation, but we can just do as the following:



$$14152 \equiv (31415 - 3*10000)*10+2 \ (mod\ 13) \ \equiv (7 - 3*3)*10+2 \ (mod\ 13) \ \equiv 8 \ (mod\ 13)$$

Now the new hash value is 8 and we see that it is the same as our pattern's hash. To check whether this is a false positive, we need to manually compare our pattern string `13658` to `14152`; as we can clearly see, the strings are different (we found a spurious hit), so we have to continue with the next digit until we find a match in the strings.

2.3 Time analysis

Analyzing the pseudocode we can see that the pre-processing phase takes $O(m)$ time, while the matching process takes $O(m(n-m+1))$ in the worst-case scenario. In a practical scenario there are only a few valid shifts, perhaps a constant c of them. With this assumption, the expected time for the matching process is $O(cm+(n-m+1)) = O(n+m)$, plus the time needed to check for false positives. Given $O(n/q)$ the number of false positives, $O(n)$ places where line 10 of the

algorithm fails and $O(m)$ time to check for each success, the expected time for the matching process is

$$O(n) + O(m(v + n/q))$$

with v as the number of valid shifts. If we choose $q \geq m$ and if the number of valid shifts is small ($O(1)$), we can assume that the algorithm takes only $O(m+n)$ time. Since $m \leq n$, the matching time is $O(n)$.

Chapter 3

Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm manages to reduce the time needed both for matching and for pre-processing to linear time. The algorithm uses an **auxiliary function** π, precomputed from the pattern in time $O(m)$ and stored in an array $\pi[1..m]$. So, for any state $q = 0, 1..., m$ and any character $a \in \Sigma$, the value $\pi[q]$ contains the information needed to compute a transition function on a that does not depend on a. Using the array π, we can immediately know if a certain shift is invalid, in order to avoid it and to save time.

3.1 The algorithm

```
KMP-MATCHER(T, P)
1     n = T.length
2     m = P.length
3     pi = COMPUTE-PREFIX-FUNCTION(P)
4     q = 0                    // number of characters matched
5     for i = 1 to n           // scan the text from left to right
6         while q > 0 and P[q + 1] != T[i]
7             q = pi[q]        // next character does not match
8         if P[q + 1] == T[i]
9             q = q + 1        // next character matches
10        if q == m            // is all of P matched?
11            print "Pattern occurs with shift" i - m
12            q = pi[q]        // look for the next match
```

Listing 2: Matching function[1]

```
COMPUTE-PREFIX-FUNCTION(P)
1     m = P.length
2     let pi[1..m] be a new array
3     pi[1] = 0
4     k = 0
5     for q = 2 to m
6         while k > 0 and P[k + 1] != P[q]
7             k = pi[k]
8         if P[k + 1] == P[q]
9             k = k + 1
10        pi[q] = k
11    return pi
```

Listing 3: Pre-processing function[1]

The first thing the algorithm does is executing the prefix function. This function takes as input only the pattern and analyzes it in order to **find the longest proper prefix that is also a suffix** of the given string, storing its length. Once the π array is created, line 5 of the first function starts the matching loop: it compares the text to the pattern and, as soon as a partial match of length x is found, the algorithm skips ahead an amount of $x - \pi[x-1]$ characters.

3.2 Example

Supposing that our pattern is `ababaca`, let's find the π array. First, we need to initialize the array with $\pi[1] = 0$. Then, the function divides the pattern in substrings, each with incremental length (the first is `a`, then `ab`, then `aba`, ...). In this example, we have $i = 5$ and `ababa`; the prefixes of this substring are `a`, `ab`, `aba`, `abab`, while the suffixes are `a`, `ba`, `aba`, `baba`. So, the matching prefix/suffixes are `a` and `aba`, and, since the longest one is 3 character in length, $\pi[5] = 3$. Now, the next step is comparing the pattern with the text, aligning them and comparing the characters. As mentioned earlier, if only x character match, the algorithm will skip the next $x - \pi[x-1]$ characters, reducing the time needed to compare them.

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

3.3 Time analysis

The time complexity for the pre-processing function is $O(m)$ and it's quite tricky to find. The outer loop of the algorithm is executed exactly $|m|-1$ times, while the inner loop is executed no more times than the outer loop, since it uses a variables that decreases over time. So, the total time complexity of the pre-processing function is $O(2 * |m| - 1) = O(m)$.

The matching function only scans the text one time, resulting in $O(n)$ time. With that being said, the total time complexity of the algorithm is $O(n+m)$ and, like Rabin-Karp, given $m \leq n$, it's $O(n)$.

Chapter 4

Problems

4.1 SPOJ - A needle in the haystack

`https://www.spoj.com/problems/NHAY/`
This is the classic Knuth-Morris-Pratt use case: find a needle in an haystack, that means to find all occurrences of a pattern in a given text. I used the standard prefix function and the standard KMP matcher, in order to print the index of the first element of every occurrence of the pattern in the text. For example, for the pattern `foobarfoo` and the text `barfoobarfoobarfoobarfoobarfoo`, the output is 3, 9, 15, 21.

```
vector<int> pi = compute_prefix_kmp(pattern);
int q = 0;
for (int i=0; i < len_text; i++) {
    while (q > 0 && pattern[q] != text[i]) {
        q = pi[q];
    }
    if (pattern[q] == text[i]) {
        q++;
    }
    if (q == len_pattern) {
        q = pi[q-1];
    }
}
```

Listing 4: KMP matching function implementation

4.1.1 Time analysis

The time needed to compute the algorithm is, as analyzed in Chapter 3.3, $O(m+n) = O(n)$, with n length of the text and m length of the pattern.

4.2 TopCoder - Running Letters

https://www.topcoder.com/thrive/articles/SRM%20401
In this problem we have an electronic sign that displays a running message displayed over and over. Observing it, we notice some letters of the message and we write them. The purpose of the problem is to find the minimal possible length of the message, that is easily done by subtracting the period of the string from its length.

The input is a string enclosed between two quotation marks, composed by couples of N and S. To form the complete input string, we need to concatenate S for N times, and concatenate the result with the next couples. In order to decode the input, I tokenized the string into a vector of strings and then recomposed it multiplying the substring with a user-defined operator on strings, as you can see below.

```
string operator * (string a, unsigned int b) {
    string output = "";
    while (b--) output += a;
    return output;
}
```

Listing 5: Operator `*` that multiplies the string `a` for `b` times

```
int compute_prefix_kmp(string a) {
    int len = a.length();
    int pi[len];
    pi[0] = 0;
    for (int i=1; i<len; i++) {
        int j = pi[i-1];
        while (j > 0 && a[i]!=a[j]) {
            j = pi[j-1];
        }
        if (a[i] == a[j]) {
            j++;
        }
        pi[i] = j;
    }
    return len - pi[len - 1];
}
```

Listing 6: Code used to find the solution of the problem

To solve the problem I used the prefix function of KMP in order to find the period of the string. The solution is, as stated before, the length of the string minus the period.

4.2.1 Time analysis

The time complexity of this algorithm is a little bit tricky. Computing the prefix function takes $O(n)$ time[1], with n length of the input string. Other that this process, the algorithm performs string repetition on the substrings of the input string. Assuming the input is something like `"x s1 y s2 z s3"`, we have to repeat $s_1 * x$, $s_2 * y$, $s_3 * z$. The best case scenario is when $x = y = z = 1$, that takes $O(n)$ time. Anyway, since x, y and z are integers, the complexity is $O(n * k)$[2], that following the big O notation is still $O(n)$. Assuming all this, the total time complexity of the algorithm is $O(n)$.

[1] See 3.3

[2] Strings are considered like lists of characters, and repeating them takes $O(n * k)$ time

4.3 Codechef - Pizza Tossing

`https://www.codechef.com/problems/PTOSS`
This problem is similar to a coin flip problem: a pizza chef tosses his pizzas and records the side on which it lands. Then, he tries to find a pre-defined lucky sequence. The object of this problem, given the chef's lucky sequence and his toss history, is to find the expected number of times the chef has to toss the pizza in order to match his lucky sequence.

For each test case, the input contains int n, string $s1$, int m and string $s2$; the first n characters of $s1$ are the chef's lucky sequence, while the first m characters of $s2$ are the chef's toss history. In order to find both the lucky sequence and the toss history, the input must be formatted as follows: every character of s is transformed in a 5-bit binary integer from 0 to 31 ('a'-'z', 'A'-'F'); then, every 0 and 1 are replaced respectively by F and B and the string is truncated to the n-th (or m-th) digit. For example, if we have $n = 8$ and $s1 = An$, we transform An in 1101001101, then in `BBFBFFBBFB` and finally we take the first n characters, resulting in `BBFBBFFBB`.

```
string str_to_fb(string s) {
    vector<bool> binary_list;
    binary_list.reserve(5 * s.length());
    for (int i=0; i < s.length(); i++) {
            int v = value(s[i]);
            vector<bool> binary = int_to_binary(v);
            binary_list.insert(binary_list.end(),
             ↪  binary.begin(), binary.end());
    }
    string fb(5 * s.length(), ' ');
    for (int i=0; i < 5*s.length(); i++) {
            fb[i] = binary_list[i]==true ? 'B' : 'F';
    }
    return fb;
}
```

Listing 7: Function to convert the s strings to strings with B and F

After reading the 4 values and transforming the strings, the next thing I did was computing a prefix function on the lucky sequence, in order to find the amount of common prefixes/suffixes at each index. Then, I used the array `tmp1` (with a support array `tmp2`) to store the number of expected tosses for each letter of the lucky sequence. After that, using the integer variable `curr`, I found the index of the expected tosses for the chef's current toss history in common with the lucky sequence. Finally, the solution is `tmp1[n] - tmp1[curr]`, that is the number of

total expected tosses minus the number of expected tosses already done by the chef.

```
int curr = 0;
for (int i=0; i<m && curr!=n; i++) {
    while (curr > 0 && toss_history[i] != lucky_seq[curr]) {
        curr = pi[curr - 1];
    }
    if (toss_history[i] == lucky_seq[curr]) {
        curr++;
    }
}
long sol = (mod - (tmp1[n] - tmp1[curr] + mod) % mod) % mod;
```

Listing 8: Code used to find the solution of a test

4.3.1 Time analysis

The time complexity of this algorithm is $O(n)$. The first thing to do, that is decoding the input in strings with FB, takes $O(n)$ time, since it just computes the binary value and substitutes 0s and 1s. The prefix function then takes $O(n)$[3]. Building the `tmp1` and `tmp2` arrays takes $O(n)$ time, due simply to the for loop. To find the index of the chef's toss history, the next loop takes, with a similar way as the prefix function loop, only $O(n)$ time. Finally, the total time complexity is the sum of the previous complexities and, given the Big O notation, it reduces to $O(n)$

[3]See 3.3

4.4 SPOJ - Extend to palindrome

`https://www.spoj.com/problems/EPALIN/`
The task of this problem is to find palindromes of some words. Given as input an unknown amount of words, for each one find the shortest possible palindrome. For example, the shortest palindrome for `abcdeed` is `abcdeedcba`.

In order to find it, I concatenated the input string with its reverse and a character `*` that I'm sure does not appear in the text. In this way, using Knuth-Morris-Pratt's prefix function to create the *pi* array, I can easily see if some characters are already palindrome in the string. For example, `abcdeed` becomes `deedcba*abcdeed` and we can see that 4 characters are already palindrome. Using the *pi* array, with $pi[word.length() * 2]$ we will easily have the index of the last character that's not palindrome. In the example above (`abcdeed`), the *pi* array is `000100000001234` and $pi[14] = 4$. Now, the fourth character in the concatenated string `deedcba*abcdeed` is `c`, so the solution is the original word concatenated with `cba`.

```
string word = input;
// Reverse and join the strings
string reversed = word;
reverse(reversed.begin(), reversed.end());
string word_double = reversed + "*" + word;

// Build the pi array with prefix function
vector<int> pi = compute_prefix_kmp(word_double);
```

Listing 9: Initialization code

```
int ok = pi[word.length()*2];
string sol = word;
for (int i = ok; i < word.length(); i++) {
    sol += word_double[i];
}
```

Listing 10: For loop used to create the solution

4.4.1 Time analysis

The time complexity of this algorithm is $O(n)$. In fact, reversing a string only takes $O(n)$[4], computing the prefix still takes $O(n)$[5] and building the solution takes less than $O(n)$, depending on the string. With the Big O notation the time complexity reduces to $O(n)$.

[4] `https://www.geeksforgeeks.org/stdreverse-in-c/`
[5] See 3.3

4.5 SPOJ - Period

https://www.spoj.com/problems/PERIOD/
This last problems aims to know, given a string, if every prefix is a periodic substring. Precisely, for each index i of the string S we want to know the largest $K > 1$ such that the prefix A of the string S with length i can be written as $A * K$.

I solved this problem adding an `if` statement at the end of the prefix function, as shown in the code listing below. In that part of the code, I simply check if I found a period using $(i+1)\%(i+1-j) == 0$, with j calculated before in the function. If true, as the exercise asks, I print the index $i+1$ and the period length $(i+1)/(i+1-j)$.

```
void compute_prefix_kmp(string a) {
    int len = a.length();
    vector<int> pi(a.length());
    pi[0] = 0;
    for (int i=1; i < a.length(); i++) {
        int j = pi[i-1];
        while (j>0 && a[i]!=a[j]) {
            j = pi[j-1];
        }
        if (a[i] == a[j]) {
            j++;
        }
        pi[i] = j;
        if (j > 0 && (i+1)%(i+1-j)==0) {
            cout << i+1 << " " << (i+1)/(i+1-j) << endl;
        }
    }
}
```

Listing 11: Modified prefix function

The same thing could have also been done outside the prefix function, scanning again the *pi* array and computing j before finding the period. I did it directly in the prefix function in order to avoid the useless time spent to completely re-scan the array.

4.5.1 Time analysis

The time complexity of this algorithm is simply $O(n)$, derived from the Knuth-Morris-Pratt compute prefix function[6], since the solution to the problem does not do anything else besides reading the input and printing the answer directly in the prefix function.

[6]See 3.3

4.6 CodeForces - Password

`https://codeforces.com/problemset/problem/126/B`
Asterix, Obelix, Suffix and Prefix have found a temple, closed with a password-protected door. Asterix found a string s carved on a rock close to the doors, and thinks that the password is a substring of that string. Prefix thinks the substring should be at the beginning of s, Suffix thinks it should be at the end of s and Obelix thinks it should be inside s. Asterix, to please them all, chooses the substring that is both a prefix, a suffix and inside s. For example, the password for `fixprefixsuffix` is `fix`, while for `abcda` there is no suitable password. We have to help Asterix find it.

To solve this problem I started computing the prefix function of KMP on the input string. Then I check if the last element of the pi array is different from 0, otherwise we can easily tell that the password doesn't exist since there is no common prefix and suffix. After that, I initialize two variables, `max` and `found`, respectively to the last element of pi and to $pi[max - 1]$. Then I search all the pi array to find the same common prefix/suffix inside the string. If I find it, I print the substring from 0 to `found`, otherwise it's all `Just a legend`.

```
if (pi[input.length()-1] == 0) {
    cout << "Just a legend" << endl;
    return 0;
}

int max = pi[input.length()-1];
int found = pi[max-1];
for (int i=1; i < input.length()-1; i++) {
    if (pi[i]==max) {
        found = max;
    }
}

if (found == 0) cout << "Just a legend" << endl;
else cout << input.substr(0, found) << endl;
```

Listing 12: Core code of the algorithm

4.6.1 Time analysis

The time needed to execute this algorithm on a string long n characters is easily $O(n)$, due to the prefix function. In fact, the only other loop executes for $n - 2$ times, so the total is $O(n)$.

4.7 SPOJ - Findsr

`https://www.spoj.com/problems/FINDSR/`
The purpose of this problem is, given a string s, to find the maximum N such that the N-th root of s exists. The N-th root of a string s is a substring t such that $t^N = s$, where $t^3 = ttt$. For instance, if s is `abcabcabcabc` and $N = 2$, t is `abcabc`. This problem is easily solved applying Knuth-Morris-Pratt's prefix function on the string, with which we find the period. N is simply length of the string divided by the period, if it exists. Otherwise, N is 1.

```
while(cin >> input && input != "*") {
    vector<int> pi = compute_prefix_kmp(input);

    int len = input.length();
    int max = pi[len-1];
    int period_k = 1;
    if (len % (len-max) == 0) {
        period_k = len/(len-max);
    }
    cout << period_k << endl;
}
```

Listing 13: Core loop of the algorithm

4.7.1 Time analysis

The only thing this algorithm does is executing the prefix function, which is $O(n)$. Other than that, it finds the solution in $O(1)$. So, the total time complexity of this problem is simply $O(n)$.

Bibliography

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, 3rd edition*, pp. 985-1011. McGraw-Hill Education, 2009.

[2] TheLlama, 2018. *Introduction to string searching algorithms.*
`https://www.topcoder.com/thrive/articles/Introduction%20to%20String%20Searching%20Algorithms`

[3] D. Gusfield, University of California. *Algorithms on strings, trees and sequences.*
`https://github.com/rossanoventurini/CompetitiveProgramming/blob/master/notes/StringMatching.pdf`