Algorithms

14. String-Pattern Searching

Handwritten by pankaj kumar

String - Pattern Searching Algorithm

Date

Page No. 191.

① Naive Pattern Searching

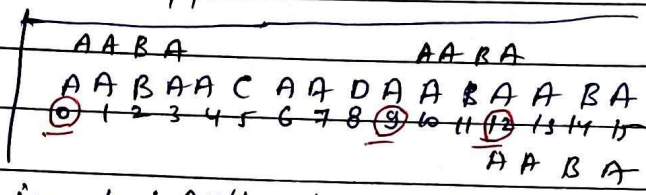
Given a text txt[0..n-1] and a pattern pat[0..m-1]. write a function search (char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. you may assume that n>m.

Ex:- ① txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"

o/p:- 10 (pattern found at index 10)

② txt[] = "AABAACAADAABAABA", pat[] = "AABA"

o/p:- 0, 9, 12



Note:- pattern searching is an important problem in computer science. when we do search for a string in notepad/word file or browser or db.

- Naive Pattern Searching:- The idea is to slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
- i.e check for the match of the first character of the pattern in the string, if it matches then check for the subsequent characters of the pattern with the respective characters of the string. if a mismatch found then moved forward in the string.

* code:-

```
void search(char* pat, char* txt){
    int m = strlen(pat);
    int n = strlen(txt);
    for(int i=0; i<=n-m; i++){
        int j;
        for(j=0; j<m; j++)
            if(txt[i+j] != pat[j]) break;
        if(j==m) cout<<i<<endl;
    }
}
```

TC: O(n*m)

O(n) ⊛ Best Case TC: The best case occur, when first character not present at all in txt.

O(m*(n-m+1)) ⊛ Worst Case: when all character of text and pattern are same. or only last character is different

① AAAA = pat[]
AAAAAAAAAA = txt[]

② AAAB
AAAAAAAAAAAAAAAAB

(2) Rabin-karp Algorithm for pattern Searching :-

- Like the Naive Algorithm, Rabin-karp algorithm also slides the pattern one by one. But unlike the naive algorithm, Rabin karp algo. matches the hash value of the pattern with the hash value of current substring of text, and if the hash value match then only it starts matching individual characters. so Rabin karp algo. needs to calculate hash values for following string.

① Pattern itself. ② All the substring of the text of length m, i.e equal to length of pattern string.

- Since we need to efficiently calculate hash values for all the substring of size m of the text, we must have a hash function which has the following property

⇒ *** Hash at the next shift must be efficiently computable from the current hash value & next character in text or we can say that, hash (txt[s+1..s+m]) must efficiently computable from hash (txt[s..s+m-1]) and txt[s+m] i.e., hash(txt[s+1..s+m]) = rehash(txt[s+m], hash(txt[s..s+m-1])) and rehash must be O(1) operation.

⇒ The hash function suggested by Rabin-karp calculates an integer value. the integer value for a string is numeric value of a string. For ex, if all possible characters are from 1 to 10, the numeric value of "122" will be "122. the no. of possible character is higher than 10 (256) and pattern length can be large. so numeric value cannot be practically stored as an integer, therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value.

$$hash(txt[s+1..s+m]) = (d(hash(txt[s..s+m-1]) - txt[s]*h) + txt[s+m]) \bmod q.$$

here, d = no. of character in alphabet
q = A prime number
$h = d^{(m-1)}$ or $d^{\wedge(m-1)}$

ex:- "1223"
hash("122") = 122
d = 10
q = x
$h = 10^2 = 100$
hash("1223") = 10*(122 - 1*(10^(3-1))) + 3 = 223

* code

```
#include <bits/stdc++.h>
using namespace std;
#define d 256
void search (char pat[], char txt[], int q) {
    int m = strlen(pat);
    int n = strlen(txt);
    int i, j, p=0, t=0, h=1;          // p → hash value for pattern, t → hash value for txt
    for (i=0; i<m-1; i++)
        h = (h*d)%q;                  // The value of h would be
                                      // "pow(d, m-1)%q"
    for (i=0; i<m; i++) {             // Calculate the has value of
        p = (d*p + pat[i])%q;         // pattern and first window
        t = (d*t + txt[i])%q;         // of text
    }
    for (i=0; i<=N-m; i++) {          // Slide the pattern over text one
                                      // by one.
        if (p==t) {                   // if has value of current window & pattern
                                      // match then only check for
            for (j=0; j<m; j++)       // next character one by one
                if (txt[i+j] != pat[j])
                    break;
            if (j==m)                 // the pattern found
                cout << i << " ";
        }
        if (i<N-m) {                  // calculate hash value for next window
                                      // of text remove leading digit, add trailing
            t = (d*(t-txt[i]*h) + txt[i+m])%q;   // digit
            if (t<0) t = (t+q);       // we might get -ve value of t,
                                      // converting it to positive.
        }
    }
}
int main() {
    char txt[] = "GEEKS FOR GEEKS";
    char pat[] = "GEEK";
    int q = 101;
    search(pat, txt, q);
    return 0;
}
```

* average time complexity is $O(n+m)$

* but worst case is $O(nm)$ when all character of pattern & text are same as the hash value of all substring of txt with hash value of pat[].
  txt[] = "AAAAAAAAAA"
  pat[] = "AAA".

(3) kmp algorithm for pattern searching.

- the TC of Naive pattern searching and Rabin-karp for searching patterns. The worst case complexity of both of the algorithm is $O(n*m)$.
- The TC of kmp in worst case is $O(n)$.

✱ kmp (knuth morris pratt) pattern searching.

→ The kmp matching algo. uses degenerating property (pattern having the same sub-pattern appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). the basic idea behind kmp's algo is: whenever we detect a mismatch (after some matches). we already know some of the character in the text of the next window. we take advantage of this information to avoid matching characters that we know will anyway match.

✱ Matching Overview

- txt = "AAAAABAAABA", pat = "AAAA"

we compare first window of txt with pat

txt = "AAAAABAAABA", pat = "AAAA" [we find a match]

- In the next step we compare next window of text with pat

txt = "AAAA(A)BAAABA"
pat = "AAA(A)" [pattern shifted one position]

Note:- here we don't need to compare whole string. where kmp does optimization over Naive. here, we only compare fourth (A) of pattern with four character of current window of text to decide whether current window matches or not. since we know first three characters will anyway match, we skipped matching first three characters.

✱ Need of preprocessing

Here, a question arises how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array lps[] that tells us the count of characteres to be skipped.

- lps indicates longest proper prefix which is also suffix.

For ABC:- prefix is "", "A", "AB" & "ABC" proper prefix is "A" & "AB", suffix are: "", "C", "BC", & "ABC".

* preprocessing overview :-

- construct an auxiliary lps[] of size m (same as size of pattern) which is used to skip characters while matching
- we search for lps in sub-patterns. more clearly we focus on sub-strings of patterns that are either prefix and suffix
- For each sub-pattern pat[0..i] where i=0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

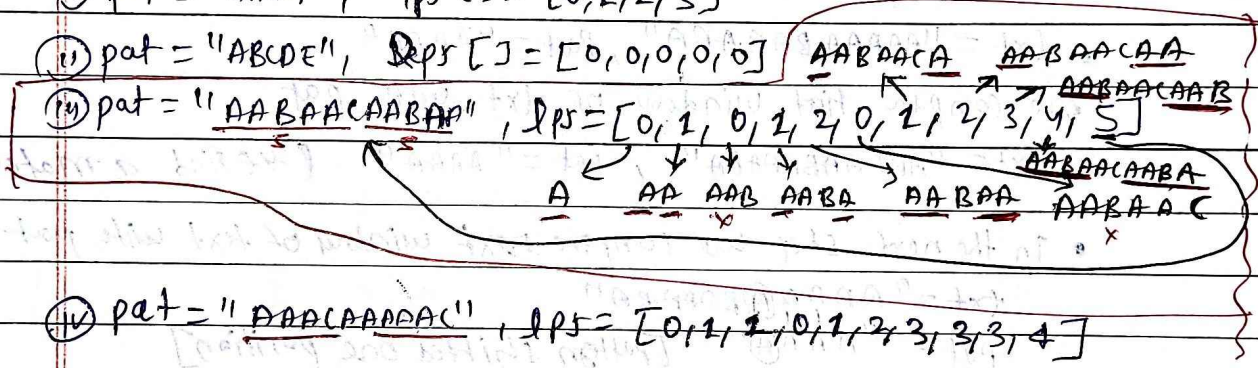
Note:- lps[i] could also be defined as longest prefix which is also proper suffix.

ex:- lps[] construction:

(i) pat = "AAAA", lps[] = [0,1,2,3]

(ii) pat = "ABCDE", lps[] = [0,0,0,0,0]

(iii) pat = "AABAACAABAA", lps = [0,1,0,1,2,0,1,2,3,4,5]



(iv) pat = "AAACAAAAAC", lps = [0,1,2,0,1,2,3,3,3,4]

* Searching algorithm / how to use lps[]

- we start comparison of pat[j] with j=0 with character of current window of text
- we keep matching characters txt[i] & pat[j] and keep incrementing i & j while pat[j] & txt[i] keep matching.
- when we see a mismatch.
  - we know that characters pat[0..j-1] match with txt[i-j..i-1]
  - since, we know lps[j-1] is count of character of pat[0..j-1] that are both proper prefix and suffix.
  - from above two points, we can conclude that we do not need to match lps[j-1] character with txt[i-j..i-1] because we know that these character will anyway match.

ex:- pat = "AAAA" , lps = [0,1,2,3]

txt = "AAAABAAABA"

in this situation we don't need to match pat[0] with txt[i], we will match pat[lps[j-1]] with txt[i] i.e match pat[3] with txt[i]

, here if,
txt = AAAAABAABA
pat = AAAA
(last character not match)

txt = AAAABAABA
pat = AABA

* Code :-

```
#include <bits/stdc++.h>
using namespace std;
void computeLPSArray(char* pat, int m, int* lps);
void kmpSearch(char* pat, char* txt){
    int m = strlen(pat);
    int n = strlen(txt);
    int lps[m];
    computeLPSArray(pat, m, lps);
    int i=0, j=0;
    while (i<n){
        if(pat[j]==txt[i]){
            j++; i++;
        }
        if(j==m){
            cout<<"Found pattern at index "<<i-j<<"\n";
            j=lps[j-1];
        }
        else if(i<n && pat[j]!=txt[i]){
            if(j!=0)
                j=lps[j-1];
            else
                i=i+1;
        }
    }
}
void computeLPSArray(char* pat, int m, int* lps){
    int len=0;
    lps[0]=0;
```

```
int i=1;
while (i<m){
    if (pat[i]==pat[len]){
        len++;
        lps[i]=len;
        i++;
    }
    else{
        if (len!=0){
            len=lps[len-1];
        }
        else{
            lps[i]=0;
            i++;
        }
    }
}
int main(){
    char txt[]="ABABDABACDABABCABAB";
    char pat[]="ABABCABAB";
    KMPSearch(pat,txt);
    return 0;
}
```

(*) Creating LPS Array (Longest prefix suffix) O(m)

(4.) Z-algorithm :-

- This algorithm finds all occurrences of a pattern in text in linear time. Let length of text be 'n' and of pattern be m, then total time taken is O(m+n) with linear space.
- Since KMP & Z have both same time complexity as linear but Z-algorithm is simpler to understand.
- In this algo, we construct a z array.

(*) Z-array :-

- Z-array is of same length of string. An element Z[i] of Z array stores length of the longest substring starting from str[i] which is also a prefix of str[0...n-1]. The first entry of Z array is meaningless as complete string is always prefix of itself.

Date

Page No. 199.

ex:- Index 0 1 2 3 4 5 6 7 8 9 10 11

① Text a a b c a a b x a a a z

Z-values x 1 0 0 3 1 0 0 2 2 1 0

② str = "aaaaaa", Z[] = {x, 5, 4, 3, 2, 1}

③ str = "aabaacd", Z[] = {x, 1, 0, 2, 1, 0, 0}

④ str = "abababab", Z[] = {x, 0, 6, 0, 4, 0, 2, 0}

* How Z-array used to search pattern in Linear time?

- The idea is to concatnate pattern and text, and create a string "P$T" where P is pattern & $ is special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. in Z array if Z value at any point is equal to pattern length, then pattern is present at that point.

ex:-

pattern P = "aab", Text T = "baabaa"

The concatenated string is = "aab$baabaa"

Z-array for concatenated string is {x, 1, 0, 0, 0, 3, 1, 0, 2, 1}.

here, the value of 3 in Z array indicates present of pattern

* To construct Z array

ex:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	a	b	x	a	a	b	x	c	a	a	b	x	a	a	b	x	a	y

Z-array → x 1 0 0 4 1 0 0

For index 5 we don't need to compare since the pattern is same which is at index 1 (abx) and (abx) bcz 0123 == 4567

∴ it is obvious 123 == 562

similarly 6 & 7 will be equal to Z[2] & Z[3] i.e 0, 0

x 1 0 0 4 1 0 0 3 1 0 2 1 0 0 4 1 0 0

same as idx 1

before using 4!

of index 13 by comparing with index Z[4] which is 4, we need to calculate if (13+4) should be lies in the range length of (3) since (13+4) is 17 and it is not in the range so again we have to compare from S[17] and S[4] and change range from 13 to 16.

Code:-

```
#include <bits/stdc++.h>
using namespace std;
void getzarr(string str, int z[]);
void search(string text, string pattern){
    string concat = pattern + "$" + text;
    int l = concat.length();
    int z[l];
    getzarr(concat, z); //construct zarray
    for(int i=0; i<l; ++i){
        if(z[i] == pattern.length()) // if z[i] equal to pattern length
            cout<<"pattern found at index "<<i-pattern.length-1
                <<endl;
    }
}

void getzarr(string str, int z[]){ //Z-array construction
    int n = str.length();          // code
    int L=0, R=0, k;
    // [L,R] make a window which matches with prefix of string (concatnate)
    for(int i=1; i<n; ++i)
        if(i>R){ // i>R i.e nothing matches so we will
            L=R=i; // calculate z[i] using naive way.
            while(R<n && str[R-L]==str[R]){ // if R-L=0
                R++;                        // in starting, so it will start
            }                               // checking from 0th index.
            z[i] = R-L;
            R--;
        }
        else{
            k = i-L; // k corresponds to no. which matches in [L,R] interval
            if(z[k]<R-i+1) // if z[k] is less than remaing interval
                z[i] = z[k] // then z[i] will be equal to z[k].
            else{
                L=i  // if out of interval then start from R and
                while(R<n && str[R-L]==str[R]) // check manually.
                    R++;
                z[i] = R-L;
                R--;
            }
        }
}
```

O(n+1)

(5) Longest Palindromic Substring

Given a string s, return the longest palindromic substring in s.

ex:- ① I/P: s="babad" O/P: "bab" or "aba"
② s="cbbd" O/P: "bb"
③ s="abc" ans="a" or "b" or "c"

* Two Pointer Approach :- (code)

① → a b c d e f e d c g

while(i>=0) ← ... i k → while(k<n) → do this for every i ∈ (0,n)

② → a b c d e f f e d c g

while(i>=0) ← ... i k → while(k<n) → do this for every i ∈ (0,n)

skip similar characters

```
string longestPalindrome(string s){
    if (s.empty()) return "";
    if (s.size()==1) return s;
    int start=0, len=1;
    for(int i=0; i<s.size();){
        if (s.size()-i<=len/2) break;        // max length palindrome is already calculated
        int j=i, k=i;
        while (k<size()-1 && s[k]==s[k+1]) ++k;   // skip duplicate characters
        i=k+1;
        while(k<size()-1 && s[j-1]==s[k+1])
        { ++k; --j; }                        // Expand palindrome
        int temp=k-j+1;
        if (temp>len){
            start=j;
            len=temp;
        }
    }
    return s.substr(start, len);
}
```

TC: 1+2+3+4+5 ... + n/2 + n/2 - 1 + ... + 3 + 2 + 1 = n*(n/2)/2 = O(n^2)