MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
VN Karazin Kharkiv National University
School of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Master'sThesis

IMPLEMENTATION OF THE KNUTH-MORRIS-PRATT ALGORITHM FOR STRING MATCHING

Finalyear Master'sProgramstudent,
group MCS-64
specialty – Computer Sciences and Information Technologies,
Author: Xu Hui
educationalprogram: "Informatics"
Supervisor: Liudmyla Poliakova
Reviewer: Kyryl Korobchynskyi
Adviser: Anna Zozulia

Kharkiv, 2024

The urgency of the problem. Algorithms for finding a substring in a line are relevant today because we repeatedly work with text, for example, searching for information on the Internet, as well as with MS Word text editors, and in order to find similar words in the text, we use a function that is significantly more efficient in editing and correcting documents and searching for necessary information. Search algorithms are very important, although they differ depending on the type of data processed and their implementation in programs, we take into account the search time, the number of sorting operations used on the data array, and then search using one of the substring search methods. We consider different algorithms for performing searches based on specific tasksandsolvingthem.

the purposethis article is a study of various algorithms for searching for a substring in a line, implementing them in software environments, as well as solving problems relatedtothesearch.

The object of researchthis article contains algorithms for processing string data, and

subject of research- algorithms for findingsubstringsinstrings.

To achieve the formulated goal, the following tasks must be completed:

- consider and analyze algorithms for searching for a substringin a string;
- consider software implementations of substring search algorithmsin a line;
- analyze the execution time of algorithms.

Analysis of recent research and publications. The problem of software implementation of substring search algorithms in a line was dealt with by: Cole, Apostolico, Dancarlo, Croshemur, Kolussi and others, who developed the most effective solutions in terms of the number of character comparisons. The Boyer-Moore algorithm in Cole's work showed that on non-periodic patterns, the algorithm will make no more than three comparisons percompletelinepass.

The algorithm for finding a substring in a string is implemented usingthefollowingalgorithms.

The sequential (direct) search algorithm consists in a symbolic comparison of the string X=x[1]..x[n] with the substring Y=y[1]..y[m], where the length of the string is the function Length(X):= n, and the length of the substring Length(Y):=m, and $0<m\leq n$. At the first stage of the algorithm implementation, the i-th symbol of the array X is compared with the first symbol of the array Y, if it matches, then the second symbol is compared and so on, which start from positions 1,2,..m-n+1 intheword X.

If all characters match, then the string is found. If all characters do not match, that is, no substring is found in the line, then the substring is shifted one position to the right (i=i+1), and the character-by-character comparison is repeated, as in the previous stage. Shifts of the substring are repeated until the condition $i + M > N$ is fulfilled, i.e. a corresponding word is not found, or a complete collection of substring characters with the string does not occur, i.e. a correspondingsubstringisfound.

The direct search algorithm has the following disadvantages:

1. High complexity – $O(N \cdot M)$, in the best case – $O((N – M + 1) \cdot M)$;

2. If the symbol did not match the line, then the search is carried out from the first symbol of the sample, and therefore it is possible to re-examine the previously viewed X symbol. For example: we will find the string good and when searching under the string it will turn out that only three goo symbols matched, and the fourth non-matching algorithm will continue to compare the string and will not leadto a result.

3. Data about the text X obtained when checking a given shift Y is not used in any way when checking subsequent shifts andfinding a substring.

For entered lines of small size, the search works faster, and for multi-megabyte files, the search for a substring takes a long time. The complexity of this algorithm lies in the fact that, to detect the coincidence of characters at the end of the line, it is necessary to make n*m comparisons, i.e. O(N*M).

A fragment of the sequential (direct) search algorithm using the DirectSearch function is shown in Fig. 1. [5, c. 77].

```
Function DirectSearch(X:string;Y:string; var Place:byte):boolean;
Var Res: boolean ;
i,n,m:integer ;
Begin
n:=length(X);
m:=length(Y);
Res:=FALSE;
i:=1;
while (i <= n-m + 1) And Not(Res) doif
Copy(X,i,m) = Y then
begin
Res:=TRUE;
Place:=i;
end
else i:=i+1;
DirectSearch:=Res;
end;
```

Rice. 1. Implementation of the sequential (direct) search algorithm

The next substring search algorithm is the Rabin-Karp algorithm. This algorithm was developed in 1987 by Michael Rabin and Richard Karp. It is based on a simple idea, and is a modification of the linear algorithm using hashing. The idea of the search is that in the word X=x[1]..x[m], where the length of the string is the function Length(X):=n, we search for the substring Y=y[1]..y[n] , where the length of the substring is the function Length(Y):=m. Let's cut out a window of size n and move it along the input line, at the same time we will observe whether the word X in the window matches the given substring. We write some function defined on words of length n. If the values of the function in the line and on the subline are different, then there is no match. Only if the values are the same, you need to check whether the characters match by letter. To speed up the verification of finding the identity of a substring with a string, a hashfunctionisused.

The algorithm uses the fact that if two strings are identical, then theirhashvaluesarealsothesame. To implement a hash function, you need to calculate the hash value of a given substring, and then find a substring with thesamehashvalue.

There are some issues with the hash value. The first problem is that there are many different strings, but in order to have small hash values we must have some strings whose hash values match. This proves that hash values can match, but strings can't. The solution to this problem is a hash function, which ensures that with rather complex input data, it will not be displayed so often, and in the process of results, the average time of searching for a substring willnotbelong.

Another problem is the following line h: = hash (X[i + 1..i + m]).

If we enumerate thehashvaluesforthesubstring X [(i + 1) .. (i + m)], we will need O (m) time, and this happens in each cycle, the algorithm will take O (m · n) time. The answer to this task is that the variable H already contains the hash value for X [i .. (i + m - 1)]. In case we can use it to calculate the next hash value in constant time, then our problem is solved. We will be able to do this using a ring hash. A ring hash is a hash function that is used specifically for this operation. We can use this formula to calculate each subsequent fixed-time hash value: X [(i + 1) .. (i + m)] = X [i .. (i + m - 1)] - X [i] + X [i + m].

This function works, but as a result the expression if X[i..(i + m – 1)] =Y will be executed more often than circular hash functions. Note that if in the case of a bad hash function, the same as a stable function, if X[i..(i + m – 1)] = Y, the probability of executing the function will be n times for each iteration of the loop. Figure 1 shows the Rabin-Karp algorithm. [5, c. 79].

```
 function RabinKarp (string X[1..n], string Y [1..m])
Begin
hsub:=hash(Y[1..m])
h:=hash(X[1..m])
for i from 1 to (n–m+1)
if h==Y
if X[i..(i+m–1)]=sub return
i hs:=hash(s[(i+1)..(i +
m)])return not found
end;
```

Rice. 2. Implementation oftheRabin-Karpalgorithm

The Knuth-Morris-Pratt algorithm was developed in 1977 by J. Morris, V. Pattom and D. With a whip The implementation of the algorithm consists in the fact that we receive the word X = x [1] x [2] ... x [n] at the input, and the algorithm looks from left to right letter by letter, while filling the array of natural numbers l [1] . .l [n], where l[i] = word length l(x[1]...x [i]). In the event that a partial convergence of the substring Y=y[1] y[2]...y[m], where the length of the substring is the Length(Y) function, with the line is detected, then the substring can be shifted several positions to the right and in the following cases it will not

be necessary re-compare thesymbolsthatcoincided.

This algorithm uses pre-processing of the search string, and creates prefix functions based on it. The idea of the function is to find, for each substring X [1.. i] of the row X, the largest substring X [1 .. j] (j <i), present both at the beginning and at the end of the substring (as a prefix and as a suffix). If a given string prefix X of length i is longer than one character, then it is a prefix of a substring of length i-1. Following the function, we check the prefix of the previous substring Y, if the prefix is not suitable, then we check the prefixofitsprefix.

For example: for the string abcoabc, the substring is abc. The meaning of the prefix function is that we rejecttheincorrectoption.

To calculate the prefix-function, the following steps are performed. Let F(X, i)=k, where k is the length of the string prefix. Let's calculate the prefix function for i+1. If our string X[i+1]=X[k+1], then it is obvious that F(X,i+1)=k+1. If the condition is not fulfilled, then we select smaller suffixes. As we see, X[1...F(X,k) will be a suffix of the string X[1...i], and for each j ϵ (k, i) the string X[1...j] will not be a suffix. So, we have an algorithm for calculatingtheprefixfunction:

1. If X[i + 1] = X[k + 1] –> π(X, i + 1) = k + 1.
2. Otherwise, when k = 0 –> π(X, i + 1) = 0.
3. Otherwise, we set k: = π(X, k), GOTO 1.

By calculating the prefix function, we find the largest searched prefix. The authors of the algorithm D. Knuth, D. Morris and V. Pratt proved that the required execution time of the algorithm is O(m + n). This algorithm is much more effective in implementation than the two previous algorithms. In fig. 3 shows the Knuth-Morris-Pratt algorithm using the prefix function [5, c. 85].

```
procedure TForm1.Button1Click(Sender: TObject);
Var F: array of Integer;
k, i, Result: integer;
 X, Y: string;
Begin
 SetLength(F, 1+Length(Y));
 F[1] := 0;
 k := 0;
 for i := 2 to Length(Y) do
 begin
  while (k > 0) and (Y[k+1] <> Y[i]) dok
 := F[k];
  if Y[k+1] = Y[i] then
 Inc(k);
 F[i] := k;
 end;
 k := 0;
 for i := 1 to Length(X) do
  begin while (k > 0) and (Y[k+1] <> X[i]) dok
   := F[k];
  if Y[k+1] = X[i] Then
 Inc(k);
  if k = Length(Y) Then
  begin
   Result := i-length(Y)+1;
```

Rice. 3. Implementation of the Knuth-Morris-Pratt algorithm usingtheprefixfunction

The Boyer-Moore algorithm was developed by R. Boyer and D. Moore in 1977 and is considered the fastest and most efficient in the process of finding a substring in a string than previous algorithms. The algorithm consists of the following steps. In the first step, we implement the shift table for the searched substring Y = y [1] y [2] ... y [m], where the length of the substring is Length(Y). In the next step, the beginning of the line is shifted X = x [1] x [2] . If the last character of the line and the one corresponding to it when searching for a line character do not match, then the substring is shifted relative to the line by one amount obtained from the offset table, and a comparison is made again, starting from the last character of the subline. If the characters are the same, then we compare the penultimate character under the line, etc. If all characters in the substring match the superimposed string characters, this means that the given substring is found and the search is complete. If any character below the string does not match the corresponding character of the string, then we move the substring one character to the right and start the check again with the last character. The entire algorithm works until the occurrence of the searched substring is found, or the end of the lineisnotfound.

The value of the offset in the case of a mismatch of the last character is calculated based on the following: the offset below the line must be minimal, so as not to miss the entry below the line in the string. If the substring does not

contain this character at all, then we move the substring by an amount equal to its length, so that the first character of the line is superimposed on the one following the character being checked. If the given string character occurs in the substring, then we shift the substring so that the string character matches the rightmost occurrence of that character in the substring. The value of the offset of each character of the substring depends on the order of the symbols in the substring, so it is convenient to calculate the offset in advance and save it as a one-dimensional array, where each character of the alphabet corresponds to the offset relative to the last character of the substring. To calculate the table of substring shifts, we must specify the type of the shift table, which we wrote down as follows: Y_table : array [char] of byte. The advantage of this algorithm is that it is necessary to make several preliminary calculations on the substring, so that the comparison of the substring with the initial string is not carried out in all positions - some checks are skipped as those that do not give results. Theworsttime

the operation of the algorithm is O(m + n).

So, Boyer's algorithmand Moore is the most effective in the implementation process and is widely used, and its execution speed increases with the increase of sub-row. In fig. 4 shows the algorithm of Boyer and Moore [5, c. 89].

```
procedure TForm1.Button1Click(Sender: TObject);
 var
result:byte; i, j, k: byte;
   Y_len  : byte;
   X_len : byte;
   Y  table : array [char] of byte;
            X, Y : string;
begin
      Y_len := length(Y);
X_len := length(X);
if Y_len < X_len then
begin
    for i := 0 to 255 do
       Y_table[chr(i)] := Y_len;
   for i := 1 to Y_len-1 do
      Y_table[Y[i]] := Y_len-i;
  i := Y_len; j := i;
   while (j > 0) and (i <= X_len) do
   begin
      j := Y_len; k := i;
      while (j > 0) and (X[k] = Y[j]) do
      begin
         dec(k); dec(j);
      end;
      i := i + Y_table[X[i]];
   end;
       if k > X_len - Y_len then result := 0
   else result := k + 1;
end
 else
   result :=0;
```

Rice. 4. Implementation oftheBoyer-Moorealgorithm

Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm is used to search for a substring (pattern) in a string. It seems that it can be easier: we move along the line and compare the symbols with the sample in sequence. It did not match, move the start of the comparison one step and compare again. And so on until we find a sample or until we reach the end of the term. The function isapproximatelyasfollows:

Knuth-Morris-Prattalgorithmisusedtosearchfor a substring (pattern) in a string. It seems that it can be easier: we move along the line and compare the symbols with the sample in sequence. It did not match, move the start of the comparison one step and compare again. And so on until we find a sample or until we reach the end of the term. The function is approximately as follows:

```
// simple search for a pattern in a string// returns the index of the beginning of a pattern in a string or -1 if not found
intfind(char *samplechar *string) {
        // i-from which place of the term we are looking for
        // j-where we are looking for a sample
        for(inti=0;line[i];++i) {
                for(intj=0;;++j) {
                        if(!sample[j])returnand // sample found
                        if(string[i+j]!=pattern[j])break;
                }
                // until found, continue the outer loop
        }
        // there is no sample
        return-1;}
```

Simplesearchcase 'needle' - sample 'stogistogstogstogstogiglstogstogiglastogiglastog' - search string Complex search caseaabbaabaabaabbaaabaabaabaabaabbaabb The function works and quite quickly if the samples and the string are "good". The good ones are when the internal cycle is interrupted quickly (for 1-3 steps, say, as in a simple case). But if the sample and the string contain frequently repeating nested pieces (as in the complex case above), then the internal loop breaks off near the end of the sample and the search time is estimated as O(<sample length>*<string length>). If the length of the term is 100 thousand, and the length of the sample is 100, then we get O(10 million). Of course, it is really rare to meet a sample with a length of 100, but in Olympiad tasks "searching" is a common thing, so a simple search is often not suitable there. And if the string is a long text, and the sample is a fragment of a word, and it is necessary to find all occurrences of this fragment, and on the fly, as the word is typed (have you noticed how fast browsers do this)? The KMP algorithm finds all occurrences of a pattern in a string and its speed is O(<pattern length>+<pattern length>), so it is out of competition on large texts/patterns or on weak processors (as in low-budget cell phones). And now let's look at the title? Why "small"? Because the highlight of KMP is the prefix function, and it is really small. And why "miracle"? Because he seems to be solving a completely different problem, and this solution, after some miraculous trick, turns into a solution to the problem of finding all the matching samples in time. In order to understand what and how the prefix function does, let's look at a complexstring

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

0 1 0 1 2 3 4 5 2 2 3 4 5 6 7 8 9 3

The line below it is the number (position) of the symbol in the line (for the convenience of describing the algorithm, we count the number as 1), and the bottom line is an array M of prefix lengths, the key to understanding the prefix function. Let's take the symbol with the number 7 (this is a) and for K from 1 to 6, we will consider prefix strings (a substring starting with the first index of the string) and suffixes (a substring whose last symbol in the string is in position 7 (this is "our" symbol a) length K.

K	prefix	suffix	
1	and	a	everything is simple here: the prefix is the first character of the line, and the suffix is our character a
2	aa	ba	
3	aab	aba	
4	**aaaaa**	**aaaaa**	the longest match, and here and below the suffix and prefix began to overlap (as fragments of the search term)
5	aaaaa	baaba	
6	aaaab	abaaba	

Note: for length 4, the suffix (as a sequence of characters) coincides with the prefix, and this is the maximum value of K at which the suffix coincides with the prefix. It is it that is entered in the corresponding position (7) of the array of prefix lengths. It is possible to notice that for K=7 the prefix also coincides with the suffix, since it is one and the same line. But this trivial case does not suit us, because substrings are needed for the algorithm to work. Let S be the initial string,

S(n) is the beginning (prefix) of the string S of length n, S[n] is the symbol in position n of the string S. M[n] is the value in the array, S(M[n]) is the same string, which is the prefix and suffix of the maximum length for position n (for brevity, we denote it by P(n)). The string P(n) is, as it were, "virtual", it is not formed and is not written anywhere. This is just the initial fragment1 of the original string S of length M[n]. And this initial fragment1 coincides (as a sequence of symbols) with fragment2 of length M[n], the last symbol of which is in position n. If M[n]=0, then there is no coincidence. We have: position 7 of the array is filled with the value M[7]=4, the longest line P(7)='aaba' is of length 4 (naturally), we need to go to position 8 and fill M[8]. You can stupidly count all prefixes and suffixes with a length from 1 to 7, compare them and write the maximum length in position 8. But we will go the other way (following the KMP). Suppose that the longest string P(8) of length k is found, which is a prefix and suffix for position 8. The string p7 of the first k-1 characters is a prefix and suffix for position k-1. It is not a fact that for the 7th position it is the longest. However, if it turns out that p7=P7, then P8 is an extension of P7 by one symbol. To check whether it is possible to expand P7 by one position, it is necessary to check whether the symbol added to the suffix matches (this is the symbol S[8]=**a**) with the next prefix character. The next prefix character**a**is in position M[7]+1=5 (think about why so). If it coincided (and in our case it did), then the task is completed — M[8]=M[7]+1, and P(8)=P(7)+symbol in the 8th position S[8]=**a**. We get P(8)='aabaa'. With successful expansion, only one comparison is needed to calculate the next value of the array. By the way, it follows from this that when moving along the array, the values can increase by a maximumof 1.

Forconvenience, I will repeat the complex term so that it is not necessary to scroll up and down.

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

Now the second case — P8 failed to expand, i.e. symbol S[9]=adid not match the string character S at position M[8]+1=6b. The suffix is easily expanded (since a new symbol is simply added to the end), but there are problems with the prefix, because the symbol added to the suffix may not coincide with the next symbol of the prefix. If the prefix P(k) does not fit, it is necessary to find another

such prefix, in short, which has the same suffix and try to expand it. But the prefix is shorter, and with the same suffix it is S[M[M[k]]), because when filling the array M, each element contains the length of the longest prefix with the same suffix. Therefore, if we failed to expand S(M[k]), we try to expand S(M[M[k]]) and so on, until it matches or until we get zero length (then the next symbol of S should simply be compared with 1st time symbol S). The cycle of searching for incoming prefixes ends very quickly, because all the information needed for this is already in the M array. For our string, P(8) is simply a one character extension of P(7), requiring 1 comparison. However, P(8) could not be extended to P(9), since S[9]=**a**, and S[M[8]+1=6] =**b**. If the P8 prefix of length M[8]=5 did not fit, try a prefix of length M[5]=2. It also does not fit: S[2+1] =b. We try a prefix of length M[2]=1 and it can be expanded, because S[1+1] =**a**. Therefore, M[9]=2, one unit longer than the length of the expandable prefix. Filling M[10] requires 2 comparisons (you can check). But to fill in elements from 11 to 17, you will need one comparison. As a result, the calculation of the values of the array takes time of the order of O (the length of the line). So, it is more or less clear with the filling algorithm.

Let's move on to the trick.

Tofindthesampleaaaaaontimeaabaaaabaaaabaaaabaaaablet'sgluethe pattern with thelinelikethisaaaaa@aabaaaabaaaabaaaabaaaabandwewillcall the prefix function for it to fill the array.

a	a	b	a	a	@	a	a	b	a	a	b	a	a	a	a	b	a	a	b	a n d	a	a	b
1	2	3	4	5	6	7	8	9	1 0	1 1	1 2	1 3	1 4	1 5	1 6	1 7	1 8	1 9	2 0	2 1	2 2	2 3	2 4
0	1	0	1	2	0	1	2	3	4	5	3	4	5	2	2	3	4	5	3	4	5	2	3

The symbol '@' plays the role of a separator, it is known to be absent neither in the sample nor in the search string (it is necessary to select exactly such a symbol). Let's look at positions 11, 14, 19, 22 of the array. The values in the array are equal to 5, which means that a suffix of length 5 (a fragment of the search string) matches 5 characters of the prefix. And 5 characters of the prefix is a sample for the search! The search algorithm will be as follows - we paste the sample and the search string with the help of a separator, pass the "glue" to the prefix function and then search the array for elements equal to the length of the sample. , equal to the length of the sample can appear only in positions corresponding to the initial search string. The concatenated string has the length

<sample length>+<string length>, so the calculation time is estimated as O(<sample length>+<string length>), as was said at the beginning of the article. The volume of the necessary buffer prefix function is equal to <sample length>+<string length>, but it is possible to modify the prefix function so that <sample length> buffer is enough (an example of modification isintheappendix).

Prefix function

And now examples of prefix functions. The difference from the description above is that, as is customary in C languages, indices are counted from 0.

An example in C++

The function returns a vector of string prefix lengths, and the string is passed as a string (it is not necessary to calculate the length)

```
vector<size_t> prefix_function (string s) {    size_tn = s.length(); vector<size_t> pi(n); // in the i-th element (its index is i-1) the number of matching characters at the beginning and end for the substring of length i.
                                                // p[0]=0 always, p[1]=1 if it starts with two identical ones
  for(size_ti=1; i<n; ++i) { // we are looking for a prefix-suffix that can be expanded       size_tj = pi[i-1]; // the length of the previous prefix-suffix, possibly zero       while((j > 0) && (s[i] != s[j])) // this cannot be expanded, j = pi[j-1]; // take the length of the smaller prefix-suffix

     if(s[i] == s[j]) ++j; // expand the found (possibly empty) prefix-suffix pi[i] = j; }     returnpi;}
```

Example on C

The function does not return anything. The first parameter is a pointer to the string, the second is a pointer to the fillable array of prefix lengths, the third is the length of the string and the array.

```
// if the compiler doesn't understand size_t, replace with int
void prefix_function (char *s, int *pi, size_t n)
{
   pi[0]=0; // in the i-th element (its index is i-1) the number of matching characters at the beginning and end for the substring of length i.
                    // p[0]=0 always, p[1]=1 if it starts with two identical ones
   for (size_t i=1; i<n; ++i)
   {
     int j = pi[i-1];
     while ((j > 0) && (s[i] != s[j])) // not equal
        j = pi[j-1]; // take the previously calculated value (starting with the maximum possible)
     if (s[i] == s[j]) // equal
        ++j;
     pi[i] = j;
   }
}
```

Thiscodewasadded based on the results of the discussion. The pattern and search term are transmitted in different parameters, i.e. they do not need to be "glued". The array of prefix lengths is populated only for the sample.

```
// an example of a processing function that outputs the index of the beginning of the found sample
intf(size_ti) { printf("%d\n",i);    return1;}// str the search string.// obr the sample we are looking for.// pi an array of prefix lengths for the sample (at least how many characters are in the sample).// int f(size_t i) when the sample is found, this function is called, / / the index of the beginning of the sample found in str is passed to it.// the function returns 0 if the search should be stopped and 1 if the search should be continued.
voidprefix_find (char *streetchar *picturesize_t *piint(*f)(size_t)) { pi[0]=0; // in the i-th element (its index is i-1), the number of coincident // characters at the beginning of the pattern and at the end of the substring of length i. // p[0]=0 always, p[1]=1 if it starts with two identical ones      size_tl; // will be the length of the sample // fill in the array of prefix lengths for the sample     for(l=1; obr[l]; ++l) {       size_tj = pi[l-1];       while((j > 0) && (obr[l] != obr[j])) // not equal j = pi[j-1];              // take the previously calculated value (starting with the maximum possible)
      if(obr[1] == obr[j]) // equal to ++j; pi[l] = j; }     size_tj = 0; // the number of matched symbols, i.e. the index of the compared // symbol in the pattern. In the string, the compared symbol will have the index i     for(size_ti=0; str[i]; ++i) {
this place is the key to the effectiveness of KMP. If the next characters of the string and the pattern did not match, we do not move the pattern by 1 character and do not compare it with the string from the very beginning (as in the primitive search at the beginning of the article). We shift the pattern by a few characters and make a single comparison of str[i] and obr[j]. If it does not match, shift the pattern again until the symbols match or we reach the beginning of the pattern. Then. according to the search term, we do not move back, only forward (albeit with stops).
      while((j > 0) && (str[i] != obr[j])) // The next character in the string did not match the character in the pattern. We move the sample, // and we know for sure that the first j symbols of the sample coincided with the symbols of the string // and we need to compare the j+1st symbol of the sample (its index j) with the i+1st symbol of the string. j = pi[j-1]; // if j=0, then the beginning of the sample has been reached and the cycle should be interrupted
if(str[i] == obr[j]) // there is a match of the next symbol ++j; // increase the length of the matched fragment by 1
if(j==l)         if(!f(i-l+1)) // sample found, call the processing function                  return; // and exit the procedure if it returns 0. }}
```

My story about a small miracle - the KMP algorithm has come to an end. Of course, this article about KMP is far, far from the first and certainly not the last. Here are two articles here, in Khabrakhabr:Substring search. Knuth–Morris-Prattalgorithmand[Substringsearchandrelated questions](). But I hope that someone will find this article useful for themselves, and someone (after all, it might happen) will start using KMP. Even if he did not fully understand the internal mechanism of his work (it is never too late to understand). The KMP algorithm is not the only fast search algorithm, but it is fast enough (for Olympiad-type problems) and at the same time simple. AlgorithmBoyer-Mooreitiscloseto KMP in terms of complexity, and for a certain range of tasks (where the sample does not contain repeating fragments) it is faster. However, for the second round of problems (where the pattern and the search string contain repeating sequences, as in the examples of the article), it is noticeably inferior to KMP. Finally, there are tasks where the choice of one or the other is a matter of taste (about which there is no dispute). In some cases (or even areas) thealgorithmAho - Korasykitcanbemore effective than KMP and Boyer-Moore. But those who really need this algorithm do not need a popularpresentation, IMHO.

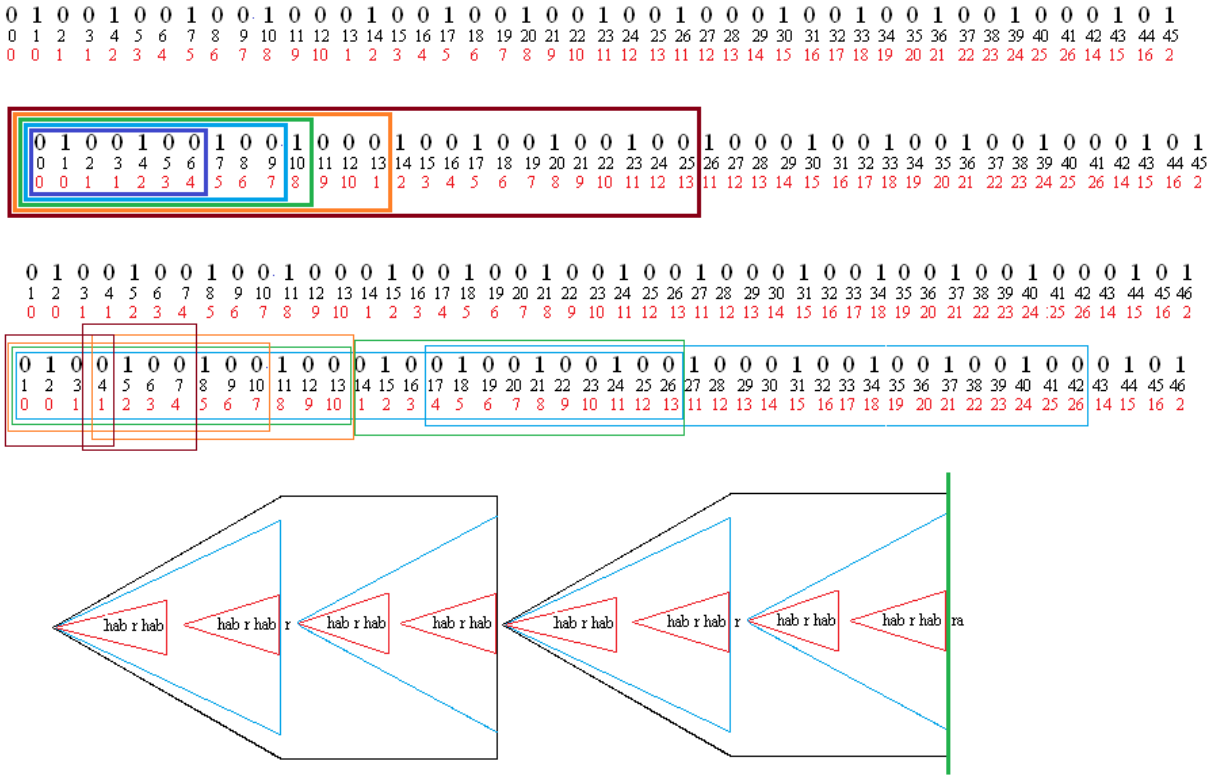
Addition based on the results of the discussion

It is possible to modify the prefix function so that it uses a buffer not O(string length+sample length) but O(sample length). In this case, memory needs to be allocated only for storing the sample prefix lengths, and the search string prefix lengths are not stored in the array. The length for the current position is stored in a variable, and as soon as it is compared with the length of the sample (ie, the sample is found), the prefix function calls the function for processing the found

fragment. The modified function prefix_find is added to the article. In reality, I have never needed to save on allocating memory for an array, but I do not rule out that someone may need it. Mayorov drew my attention to the possibility of saving memory, and he also gave a link in hismessageson the implementation of a variant of the prefix function, which displays the positions of the found fragments in cout. According to the remark, staticlab changed the type of variables to size_t (it will be more correct this way). For those who have read the article to the end, I will provide illustrations showing how prefix-suffix blocks are organized inrathercomplexcases:

Illustrations

The strings of zeros and ones are the same, only in the first two illustrations the indices are 0, and in the following ones - 1. In fact, this is not essential, since the lengths of the substrings are written into the array, and the numbering of symbols in the string does not play a role. The last illustration is a "bird's eye" view of prefixes-suffixesarrays ofindicesandlengths.



Implementation of the algorithm in the Pascallanguage (dialectDelphi5 andabove)

```
Function Pos ( const Needle, HayStack : string ) : Integer;
var
  F: array of Integer;
  k, i: integer;
begin
  SetLength(F, 1+Length(Needle));
  F[1] := 0;
  k := 0;
  for i := 2 to Length(Needle) do
  begin
    while (k > 0) and (Needle[k+1] <> Needle[i]) do
      k := F[k];
    if Needle[k+1] = Needle[i] then
      Inc(k);
    F[i] := k;
  end;
  k := 0;
  for i := 1 to Length(HayStack) do
   begin
    while (k > 0) and (Needle[k+1] <> HayStack[i]) do
      k := F[k];
    if Needle[k+1] = HayStack[i] Then
      Inc(k);
    if k = Length(Needle) Then
    begin
      Result := i-length(Needle)+1;
      Break;
    end;
  end;
end;
```

Implementation of the algorithm in the languageC++

```cpp
#include <iostream>
#include <string>
#include <vector>


using namespace std;

string::size_type KMP(const string& S, int begin, const string& pattern)
{
	vector<int> pf(pattern.length());

	pf[0] = 0;
	for (int k = 0, i = 1; i < pattern.length(); ++i)
	{
		while ((k > 0) && (pattern[i] != pattern[k]))
			k = pf[k - 1];

		if (pattern[i] == pattern[k])
			k++;

		pf[i] = k;
	}

	for (int k = 0, i = begin; i < S.length(); ++i)
	{
		while ((k > 0) && (pattern[k] != S[i]))
			k = pf[k - 1];

		if (pattern[k] == S[i])std::cout << k <<"\t" <<i+1 << "\t";
			k++;

		if (k == pattern.length())
			return (i - pattern.length() + 1);

	}

	return (string::npos);
	std::cout << string::npos;
}

int main()
{
	setlocale(LC_ALL, "ru");

	string pattern = "Raw";
    string ss = "лилRjjruoлилRawилил";
	int begin = 3;
	KMP( ss, begin, pattern);

	return 0;
}
```

Implementation of the algorithm in the languageC

```c
int seek_substring_KMP (char s[], char p[])
{
        int i, j, N, M;
        N = strlen(s);
        M = strlen(p);

        int *d = (int*)malloc(M * sizeof(int));

        d[0] = 0;
        for(i = 1, j = 0; i < M; i++)
        {
                while(j > 0 && p[j] != p[i])
                        j = d[j-1];
                if(p[j] == p[i])
                        j++;
                d[i] = j;
        }
        for(i = 0, j = 0; i < N; i++)
        {
                while(j > 0 && p[j] != s[i])
                        j = d[j - 1];
                if(p[j] == s[i])
            j++;
                if(j == M)
         {
                    free(d);
            return i - j + 1;
         }
        }
    free(d);
        return -1;
}
```

Implementation of the algorithm in the languagec#

```
static int[] GetPrefix(string s)
{
    int[] result = new int[s.Length];
    result[0] = 0;
    int index = 0;

    for (int i = 1; i < s.Length; i++)
    {
        int k = result[i - 1];
        while (s[k] != s[i] && k > 0)
        {
            k = result[k - 1];
        }
        if (s[k] == s[i])
        {
            result[i] = k + 1;
        }
        else
        {
            result[i] = 0;
        }
    }
    return result;
}

static int FindSubstring(string pattern, string text)
{
    int[] pf = GetPrefix(pattern);
    int index = 0;

    for (int i = 0; i < text.Length; i++)
    {
        while (index > 0 && pattern[index] != text[i]) { index = pf[index -
1]; }
        if (pattern[index] == text[i]) index++;
        if (index == pattern.Length)
        {
            return i - index + 1;
        }
    }

    return -1;
}
```

Implementation of the algorithm in the languageJava

```java
public static int[] indexesOf(char[] pattern, char[] text)
{
    int[] pfl = pfl(pattern);
    int[] indexes = new int[text.length];
    int size = 0;
    int k = 0;
    for (int i = 0; i < text.length; ++i)
    {
        while (pattern[k] != text[i] && k > 0)
        {
            k = pfl[k - 1];
        }
        if (pattern[k] == text[i])
        {
            k = k + 1;
            if (k == pattern.length)
            {
                indexes[size] = i + 1 - k;
                size += 1;
                k = pfl[k - 1];
            }
        }
        else
        {
            k = 0;
        }
    }
    return Arrays.copyOfRange(indexes, 0, size);
}

public static int[] pfl(char[] text)
{
    int[] pfl = new int[text.length];
    pfl[0] = 0;

    for (int i = 1; i < text.length; ++i)
    {
        int k = pfl[i - 1];
        while (text[k] != text[i] && k > 0)
        {
            k = pfl[k - 1];
        }
        if (text[k] == text[i])
        {
            pfl[i] = k + 1;
        }
        else
        {
            pfl[i] = 0;
        }
    }

    return pfl;
}
```

Implementation of the algorithm in the languageHaskell

```haskell
import qualified Data.Vector as V
import qualified Data.Vector.Mutable as VM
import qualified Data.ByteString.Char8 as BS

type PrefixTable = V.Vector Int

makePrefixTable :: BS.ByteString -> PrefixTable
makePrefixTable s =
  let n = BS.length s
  in V.create $ do
    p <- VM.new n
    VM.set p 0
    let loop i k | i == n = return ()
                 | k > 0 && (BS.index s i /= BS.index s k) = do
                    nk <- VM.read p (k - 1)
                    loop i nk
                 | otherwise = do
                    let nk = if BS.index s i == BS.index s k
                             then k + 1
                             else k
                    VM.write p i nk
                    loop (i + 1) nk

    loop 1 0
    return p

indexOf :: BS.ByteString -> BS.ByteString -> Int
indexOf needle haystack =
  let pt    = makePrefixTable needle
      hLen  = BS.length haystack
      nLen  = BS.length needle
      loop i k | k == nLen = i - nLen
               | i == hLen = -1
               | k > 0 && (BS.index needle k /= BS.index haystack i) =
                  loop i $ pt V.! (k - 1)
               | otherwise =
                  let nk = if BS.index needle k == BS.index haystack i
                           then k + 1
                           else k
                  in loop (i + 1) nk
  in loop 0 0
```

Go

```go
func computeLPS(pat string) (lps []int) {
    M := len(pat)
    lps = make([]int, M) // lps[0] = 0

    l := 0 // length of the previous longest prefix suffix

    // the loop calculates lps[i] for i = 1 to M-1
    for i := 1; i < M; i++ {
        for {
            if pat[i] == pat[l] {
                l++
                break
            }

            if l == 0 {
                break
            }

            l = lps[l-1]
        }
        lps[i] = l
    }
    return lps
}

func KMPSearch(pat, txt string) {
    M, N := len(pat), len(txt)

    // Preprocess the pattern that will hold the longest prefix suffix values
for pattern
    lps := computeLPS(pat)

    for i, j := 0, 0; i < N; i++ {
        for {
            if pat[j] == txt[i] {
                j++

                if j == M {
                    fmt.Printf("Found pattern at index %d \n", i-j+1)
                    j = lps[j-1]
                }
                break
            }

            if j > 0 {
                j = lps[j-1]
            } else {
                break
            }
        }
    }
}
```

Lua

For Lua version 5.1 andhigher.

```
function generate_fail_table(substring,sublen)
         comparisons = 0
         fail = {0}
         for i=2,sublen do
             temp = fail[i - 1]
             while temp > 0 and string.sub(substring,temp,temp) ~=
string.sub(substring,i - 1,i - 1) do
                  comparisons = comparisons + 1
                  temp = fail[temp]
             end
         fail[i] = temp + 1
         end

         return {fail, comparisons + 1}
end

--The actual kmp algorithm which takes
--the substring and text as arguments
function kmp_algorithm(substring,text)
         --starting positions...
         subloc = 1
         textloc = 1

         --get the length of substring and text..
         sublen = string.len(substring)
         textlen = string.len(text)

         --generate the fail links...
         failure = generate_fail_table(substring,sublen)

         --store failure comparisons
         comparisons = failure[2]

         --store fail table
         fail = failure[1]

         --the main loop..
         while textloc <= textlen and subloc <= sublen do

               if subloc == 0 or string.sub(text,textloc,textloc) ==
string.sub(substring,subloc,subloc) then
                  comparisons = comparisons + 1
                  textloc = textloc + 1
                  subloc = subloc + 1
               else --no match found so follow fail link
                  subloc = fail[subloc]
               end
         end

         if subloc > sublen then
            return {textloc - sublen, comparisons}
         else
            return {0, comparisons}
         end

end
```

Python

```
#from http://code.activestate.com/recipes/117214/
def KnuthMorrisPratt(text, pattern):

    '''Yields all starting positions of copies of the pattern in the text.
Calling conventions are similar to string.find, but its arguments can be
lists or iterators, not just strings, it returns all matches, not just
the first one, and it does not need the whole text in memory at once.
Whenever it yields, it will have read the text exactly up to and including
the match that caused the yield.'''

    # allow indexing into pattern and protect against change during yield
    pattern = list(pattern)

    # build table of shift amounts
    shifts = [1] * (len(pattern) + 1)
    shift = 1
    for pos in range(len(pattern)):
        while shift <= pos and pattern[pos] != pattern[pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift

    # do the actual search
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == len(pattern) or \
              matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == len(pattern):
            yield startPos
```

PHP

```php
<?php declare(strict_types = 1);

function kmp(string $str = '', string $substr = ''): int {

    $n = mb_strlen($str);
    $m = mb_strlen($substr);

    // prefix
    $d[0] = 0;
    for ($i = 1, $j = 0; $i < $m; $i++) {

        while($j > 0 && $substr[$j] != $substr[$i]) {
            $j = $d[$j - 1];
        }

        if($substr[$j] == $substr[$i]) {
            $j++;
        }

        $d[$i] = $j;

    }

    // search
    for($i = 0, $j = 0; $i < $n; $i++) {

        while($j > 0 && $substr[$j] != $str[$i]) {
            $j = $d[$j - 1];
        }

        if($substr[$j] == $str[$i]) {
            $j++;
        }

        if($j == $m) {
            return $i - $j + 1;
        }

    }

    return -1;

}
```

Conclusions. After considering four algorithms for finding a substring in a line, we can conclude that the direct search algorithm is not optimal, as it is characterized by a high complexity of O (N·M) for searching multi-megabyte files. The leading algorithms are Boyer-Moore and Knuth-Morris-Pratt, because they are efficient and perform certain classes of tasks.