# Searching for Patterns

# Set 1 (Naive Pattern Searching)

iven 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.

Examples:

Input: txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output: Pattern found at index 10

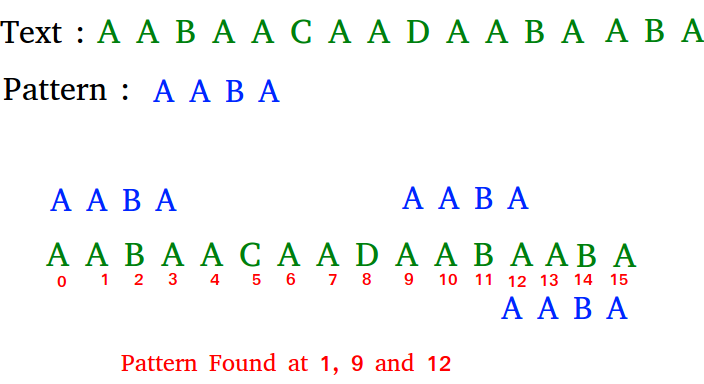
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

**Naive Pattern Searching:**  
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.

// C program for Naive Pattern Searching algorithm

#include<stdio.h>

#include<string.h>

void search(char \*pat, char \*txt)

{

int M = strlen(pat);

int N = strlen(txt);

/\* A loop to slide pat[] one by one \*/

for (int i = 0; i <= N - M; i++)

{

int j;

/\* For current index i, check for pattern match \*/

for (j = 0; j < M; j++)

if (txt[i+j] != pat[j])

break;

if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]

printf("Pattern found at index %d n", i);

}

}

/\* Driver program to test above function \*/

int main()

{

char txt[] = "AABAACAADAABAAABAA";

char pat[] = "AABA";

search(pat, txt);

return 0;

}

Output:

Pattern found at index 0

Pattern found at index 9

Pattern found at index 13

**What is the best case?**  
The best case occurs when the first character of the pattern is not present in text at all.

txt[] = "AABCCAADDEE"

pat[] = "FAA"

The number of comparisons in best case is O(n).

**What is the worst case ?**  
The worst case of Naive Pattern Searching occurs in following scenarios.  
1) When all characters of the text and pattern are same.

txt[] = "AAAAAAAAAAAAAAAAAA"

pat[] = "AAAAA".

2) Worst case also occurs when only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAAB"

pat[] = "AAAAB"

Number of comparisons in worst case is O(m\*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the worst case to O(n). We will be covering KMP in the next post. Also, we will be writing more posts to cover all pattern searching algorithms and data structures.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**KMP (Knuth Morris Pratt) Pattern Searching**  
The [Naive pattern searching algorithm](http://www.geeksforgeeks.org/?p=11871) doesn’t work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

txt[] = "AAAAAAAAAAAAAAAAAB"

pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"

pat[] = "ABABAC" (not a worst case, but a bad case for Naive)

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns 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 algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

**Matching Overview**

txt = "AAAAABAAABA"

pat = "AAAA"

We compare first window of **txt** with **pat**

txt = "**AAAA**ABAAABA"

pat = "**AAAA**" [Initial position]

We find a match. This is same as [Naive String Matching](http://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/).

In the next step, we compare next window of **txt** with **pat**.

txt = "**AAAAA**BAAABA"

pat = "**AAAA**" [Pattern shifted one position]

This is where KMP does optimization over Naive. In this

second window, we only compare fourth A of pattern

with fourth 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?**

An important question arises from above explanation,

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 count of characters to be skipped.

**Preprocessing Overview:**

* KMP algorithm does preproceses pat[] and constructs an auxiliary **lps[]** of size m (same as size of pattern) which is used to skip characters while matching.
* **name lps indicates longest proper prefix which is also suffix.**. A proper prefix is prefix with whole string **not** allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC” and “ABC”.
* 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. We need to use proper at one place to make sure that the whole substring is not considered.

Examples of lps[] construction:

For the pattern “AAAA”,

lps[] is [0, 1, 2, 3]

For the pattern “ABCDE”,

lps[] is [0, 0, 0, 0, 0]

For the pattern “AABAACAABAA”,

lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern “AAACAAAAAC”,

lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern “AAABAAA”,

lps[] is [0, 1, 2, 0, 1, 2, 3]

**Searching Algorithm:**  
Unlike [Naive algorithm](http://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/), where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match character that we know will anyway match.

How to use lps[] to decide next positions (or to know number of characters to be skipped)?

* + We start comparison of pat[j] with j = 0 with characters of current window of text.
  + We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
  + When we see a **mismatch**
    - We know that characters pat[0..j-1] match with txt[i-j+1…i-1] (Note that j starts with 0 and increment it only when there is a match).
    - We also know (from above definition) that lps[j-1] is count of characters 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 these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider above example to understand this.

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

lps[] = {0, 1, 2, 3}

i = 0, j = 0

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

txt[i] and pat[j[ match, do i++, j++

i = 1, j = 1

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

txt[i] and pat[j[ match, do i++, j++

i = 2, j = 2

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

pat[i] and pat[j[ match, do i++, j++

i = 3, j = 3

txt[] = "**AAAA**ABAAABA"

pat[] = "**AAAA**"

txt[i] and pat[j[ match, do i++, j++

i = 4, j = 4

Since j == M, print **pattern found** and resset j,

j = lps[j-1] = lps[3] = 3

Here unlike Naive algorithm, we do not match first three

characters of this window. Value of lps[j-1] (in above

step) gave us index of next character to match.

i = 4, j = 3

txt[] = "A**AAAA**BAAABA"

pat[] = "**AAAA**"

txt[i] and pat[j[ match, do i++, j++

i = 5, j = 4

Since j == M, print **pattern found** and reset j,

j = lps[j-1] = lps[3] = 3

Again unlike Naive algorithm, we do not match first three

characters of this window. Value of lps[j-1] (in above

step) gave us index of next character to match.

i = 5, j = 3

txt[] = "AA**AAAB**AAABA"

pat[] = "**AAAA**"

txt[i] and pat[j] do NOT match and j > 0, change only j

j = lps[j-1] = lps[2] = 2

i = 5, j = 2

txt[] = "AAA**AABA**AABA"

pat[] = "**AAAA**"

txt[i] and pat[j] do NOT match and j > 0, change only j

j = lps[j-1] = lps[1] = 1

i = 5, j = 1

txt[] = "AAAA**ABAA**ABA"

pat[] = "**AAAA**"

txt[i] and pat[j] do NOT match and j > 0, change only j

j = lps[j-1] = lps[0] = 0

i = 5, j = 0

txt[] = "AAAAA**BAAA**BA"

pat[] = "**AAAA**"

txt[i] and pat[j] do NOT match and j is 0, we do i++.

i = 6, j = 0

txt[] = "AAAAAB**AAABA**"

pat[] = "**AAAA**"

txt[i] and pat[j] match, do i++ and j++

i = 7, j = 1

txt[] = "AAAAAB**AAAB**A"

pat[] = "**AAAA**"

txt[i] and pat[j] match, do i++ and j++

We continue this way...

|  |
| --- |
| // C++ program for implementation of KMP pattern searching  // algorithm  #include<bits/stdc++.h>    void computeLPSArray(char \*pat, int M, int \*lps);    // Prints occurrences of txt[] in pat[]  void KMPSearch(char \*pat, char \*txt)  {      int M = strlen(pat);      int N = strlen(txt);        // create lps[] that will hold the longest prefix suffix      // values for pattern      int lps[M];        // Preprocess the pattern (calculate lps[] array)      computeLPSArray(pat, M, lps);        int i = 0;  // index for txt[]      int j  = 0;  // index for pat[]      while (i < N)      {          if (pat[j] == txt[i])          {              j++;              i++;          }            if (j == M)          {              printf("Found pattern at index %d n", i-j);              j = lps[j-1];          }            // mismatch after j matches          else if (i < N && pat[j] != txt[i])          {              // Do not match lps[0..lps[j-1]] characters,              // they will match anyway              if (j != 0)                  j = lps[j-1];              else                  i = i+1;          }      }  }    // Fills lps[] for given patttern pat[0..M-1]  void computeLPSArray(char \*pat, int M, int \*lps)  {      // length of the previous longest prefix suffix      int len = 0;        lps[0] = 0; // lps[0] is always 0        // the loop calculates lps[i] for i = 1 to M-1      int i = 1;      while (i < M)      {          if (pat[i] == pat[len])          {              len++;              lps[i] = len;              i++;          }          else // (pat[i] != pat[len])          {              // This is tricky. Consider the example.              // AAACAAAA and i = 7. The idea is similar              // to search step.              if (len != 0)              {                  len = lps[len-1];                    // Also, note that we do not increment                  // i here              }              else // if (len == 0)              {                  lps[i] = 0;                  i++;              }          }      }  }    // Driver program to test above function  int main()  {      char \*txt = "ABABDABACDABABCABAB";      char \*pat = "ABABCABAB";      KMPSearch(pat, txt);      return 0;  } |

Output:

Found pattern at index 10

**Preprocessing Algorithm:**  
In the preprocessing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]. See computeLPSArray () in the below code for details.

**Illustration of preprocessing (or construction of lps[])**

pat[] = "**AAACAAAA**"

len = 0, i = 0.

**lps[0] is always 0**, we move

to i = 1

len = 0, i = 1.

Since pat[len] and pat[i] match, do len++,

store it in lps[i] and do i++.

len = 1, **lps[1] = 1**, i = 2

len = 1, i = 2.

Since pat[len] and pat[i] match, do len++,

store it in lps[i] and do i++.

len = 2, **lps[2] = 2**, i = 3

len = 2, i = 3.

Since pat[len] and pat[i] do not match, and len > 0,

set len = lps[len-1] = lps[1] = 1

len = 1, i = 3.

Since pat[len] and pat[i] do not match and len > 0,

len = lps[len-1] = lps[0] = 0

len = 0, i = 3.

Since pat[len] and pat[i] do not match and len = 0,

Set **lps[3] = 0** and i = 4.

len = 0, i = 4.

Since pat[len] and pat[i] match, do len++,

store it in lps[i] and do i++.

len = 1, **lps[4] = 1**, i = 5

len = 1, i = 5.

Since pat[len] and pat[i] match, do len++,

store it in lps[i] and do i++.

len = 2, **lps[5] = 2**, i = 6

len = 2, i = 6.

Since pat[len] and pat[i] match, do len++,

store it in lps[i] and do i++.

len = 3, **lps[6] = 3**, i = 7

len = 3, i = 7.

Since pat[len] and pat[i] do not match and len > 0,

set len = lps[len-1] = lps[2] = 2

len = 2, i = 7.

Since pat[len] and pat[i] match, do len++,

store it in lps[i] and do i++.

len = 3, **lps[7] = 3**, i = 8

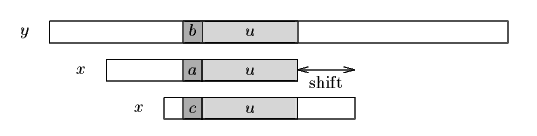
We stop here as we have constructed the whole lps[].

Description

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the «search» and «substitute» commands.

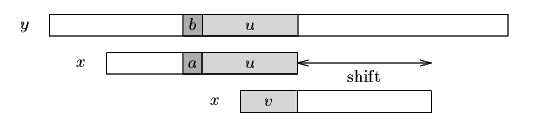
The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the ***good-suffix shift*** (also called matching shift and the ***bad-character shift*** (also called the occurrence shift).

Assume that a mismatch occurs between the character *x*[*i*]=*a* of the pattern and the character *y*[*i*+*j*]=*b* of the text during an attempt at position *j*.  
Then, *x*[*i*+1 .. *m*-1]=*y*[*i*+*j*+1 .. *j*+*m*-1]=u and *x*[*i*] neq y[*i*+*j*]. The good-suffix shift consists in aligning the segment *y*[*i*+*j*+1 .. *j*+*m*-1]=*x*[*i*+1 .. *m*-1] with its rightmost occurrence in *x* that is preceded by a character different from *x*[*i*] (*see figure 13.1*).



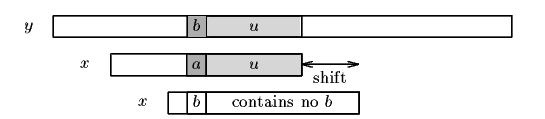
**Figure 13.1**. The good-suffix shift, *u* re-occurs preceded by a character *c* different from *a*.

If there exists no such segment, the shift consists in aligning the longest suffix *v* of *y*[*i*+*j*+1 .. *j*+*m*-1] with a matching prefix of *x* (*see figure 13.2*).



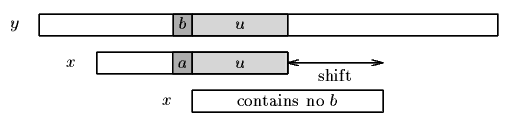
**Figure 13.2**. The good-suffix shift, only a suffix of *u* re-occurs in *x*.

The bad-character shift consists in aligning the text character *y*[*i*+*j*] with its rightmost occurrence in *x*[0 .. *m*-2]. (*see figure 13.3*)



**Figure 13.3**. The bad-character shift, *a* occurs in *x*.

If *y*[*i*+*j*] does not occur in the pattern *x*, no occurrence of *x* in *y* can include *y*[*i*+*j*], and the left end of the window is aligned with the character immediately after *y*[*i*+*j*], namely *y*[*i*+*j*+1] (*see figure 13.4*).



**Figure 13.4**. The bad-character shift, *b* does not occur in *x*.

Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the the good-suffix shift and bad-character shift. More formally the two shift functions are defined as follows.

The good-suffix shift function is stored in a table *bmGs* of size *m*+1.

Let us define two conditions:

|  |  |
| --- | --- |
| http://www-igm.univ-mlv.fr/%7Elecroq/string/images/hand.gif | *Cs*(*i*, *s*): for each *k* such that *i* < *k* < *m*, *s* geq *k* or *x*[*k*-*s*]=*x*[*k*] and |

|  |  |
| --- | --- |
| http://www-igm.univ-mlv.fr/%7Elecroq/string/images/hand.gif | *Co*(*i*, *s*): if *s* <*i* then *x*[*i*-*s*] neq *x*[*i*] |

Then, for 0 leq *i* < *m*: *bmGs*[*i*+1]=min{s>0 : *Cs*(*i*, *s*) and *Co*(*i*, *s*) hold}  
and we define *bmGs*[0] as the length of the period of *x*. The computation of the table *bmGs* use a table *suff* defined as follows: for 1 leq *i* < *m*, *suff*[*i*]=max{*k* : x[*i*-*k*+1 .. *i*]=x[*m*-*k* .. *m*-1]}

The bad-character shift function is stored in a table *bmBc* of size sigma. For *c* in Sigma: *bmBc*[*c*] = min{*i* : 1 leq *i* <*m*-1 and *x*[*m*-1-*i*]=*c*} if *c* occurs in *x*, *m* otherwise.

Tables *bmBc* and *bmGs* can be precomputed in time ***O***(*m*+sigma) before the searching phase and require an extra-space in ***O***(*m*+sigma). The searching phase time complexity is quadratic but at most 3*n* text character comparisons are performed when searching for a non periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast. When searching for a*m*-1b in b*n* the algorithm makes only ***O***(*n* / *m*) comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.