Pattern Matching algorithms

The problem of string matching

Given a string 'S', the problem of string matching deals with finding whether a pattern 'p' occurs in 's' and if 'p' does occur then returning position in 's' where 'p' occurs.

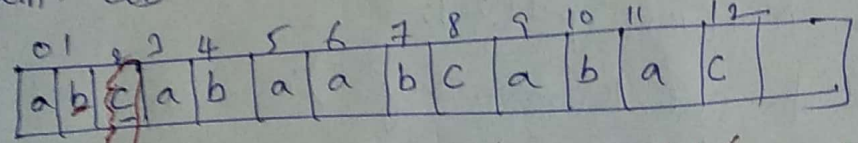
Brute force

Time complexity: $O(mn)$

Here, compare the first element of the pattern to be searched 'p' with the first element of the string 'S'. If the first element matches, compare the second element of 'p' with the second element of S. If match found proceed likewise until entire 'p' is found. If a mismatch is found at any position, shift 'p' to one position to the right and repeat comparisons beginning from first element of 'p'.

[1] Text: abcabaabcabac

[2] Pattern: abaa



Text:

Pattern: a b a a — mismatch, shift 'p' to the right (one position)

a b a a — mismatch

a b a a

a b a a — match found, return the position

Drawback: comparisons are more

// program

```
#include <stdio.h>
#include <string.h>
void search( char *t, char *p)
{ int i,j,flag=0;
  int M = strlen(t);
  int N = strlen(p);
  for(i=0; i<=M-N; i++)
  {
    for(j=0; j<N; j++)
    {
      if (p[j] != t[i+j])
        break;
    }
    if(j==N)
    { printf("\n pattern found at index %d", i);
      flag=1; }
  }
  if(flag==0) printf("\n pattern not found");
}
void main()
{
  int f;
  char text[100], pat[20];
  printf("\n enter the main string:");
  gets(text);
  printf("\n enter the pattern:");
  gets(pat);
  f= search(text, pat);
  if(f==-1)
  printf("\n pattern not found\n");
  else
  printf("\n pattern found at index %d\n", f);
}
```

text: abcabaa
pattern: abaa

M=11, N=4

for(i=0; i<=7; i++)
  for(j=0; j<4; j++)
    p[0] != t[0]
    p[0] = t[1]
    p[0] != t[2]
    p[0] = t[3]

(2) KMP Algorithm (Knuth-Morris-pratt) O(m+n)

→ It is one of the most popular patterns matching algorithms.

→ It was the first linear time complexity algorithm for string matching and used to find a pattern in a Text.

→ This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called 'prefix table' to skip characters comparison while matching. It is also known as LPS Table. (Longest proper prefix which is also suffix)

Steps for creating LPS table (prefix table)

Step (1) Define a one dimensional array with the size equal to the length of the pattern i.e LPS[size]

(2) Define variables i & j. set i=0, j=1 and LPS[0]=0

(3) Compare the characters at pattern[i] and pattern[j].

(4) if both are matched then set LPS[j] = i+1 and increment both i and j by one. goto step (3).

(5) if both are not matched then check the value of variable 'i'. if it is '0' then set LPS[j]=0 and increment 'j' value by one, if it is not '0' then set i= LPS[i-1]. Goto step (3).

(6) Repeat above steps until all the values of LPS[] are filled.

(4) Create LPS table for the following pattern

pattern:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Sol: Define LPS[] array with size '7'

0	1	2	3	4	5	6

Now, i=0, j=1 and lps[0]=0

0	1	2	3	4	5	6
0						

i=0, j=1

Compare pattern[0] with pattern[1] i.e 'A' with 'B' mismatch and i=0, LPS[1]=0 and increment 'j' by one i.e j=2

0	1	2	3	4	5	6
0	0					

i=0, j=2

Compare pattern[0] with pattern[2] i.e 'A' with 'C', mismatch and i=0, LPS[2]=0 and increment j i.e j=3

0	1	2	3	4	5	6
0	0	0				

i=0, j=3

Compare pattern[0] with pattern[3] ie 'A' with 'D' mismatch and i=0, LPS[3]=0 and increment 'j' i.e j=4

0	1	2	3	4	5	6
0	0	0	0			

i=0, j=4

Compare pattern[0] with pattern[4] ie 'A' with 'A', match so, LPS[4]=0+1=1, and increment both i & j i.e i=1, j=5

0	1	2	3	4	5	6
0	0	0	0	1		

i=1, j=5

Compare pattern[1] with pattern[5], 'B'=='B', match so. LPS[5]=1+1=2. increment i & j. i.e i=2, j=6

0	1	2	3	4	5	6
0	0	0	0	1	2	

i=2, j=6

compare pattern[2] with pattern[6] ie 'C' with 'D', mismatch and i!=0. set i=LPS[i-1]=LPS[1]=0

0	1	2	3	4	5	6
0	0	0	0	1	2	

i=0 and j=6

Compare pattern[0] with pattern[6] i.e 'A' with 'D', mismatch and i=0. So LPS[6]=0 and increment 'j'

LPS[]

0	1	2	3	4	5	6
0	0	0	0	1	2	0

Use of Lps Table

a) prefix table [when a mismatch has occured with the first character in the pattern move the pattern one position right]

It is used to decide how many characters are to be skipped for comparison when a mismatch has occured.

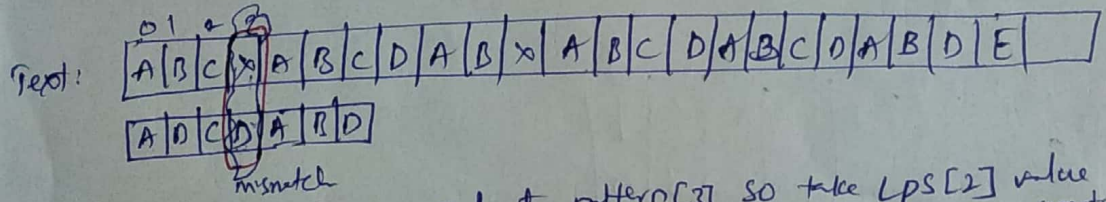
⇒ when a mismatch occurs, check the Lps value of the previous character of the mismatched character in the pattern. if it is '0' then start comparing the first character of the pattern with the ~~next character to~~ the mismatched character in the text. if it is not '0' then start comparing the character which is at an index value equal to the Lps value of the previous character to the mismatched character in the pattern with the mismatched character in the Text.

(8) Consider the following Text and pattern

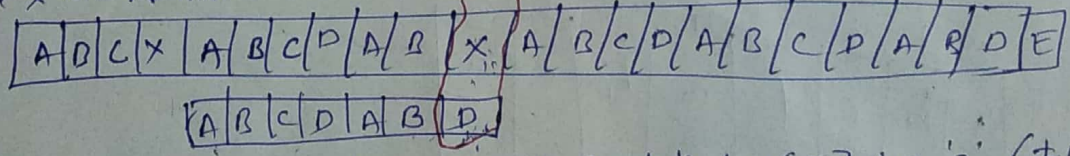
Text: ABC ABCDAB ABCDABCDABDE

pattern: ABCDABD

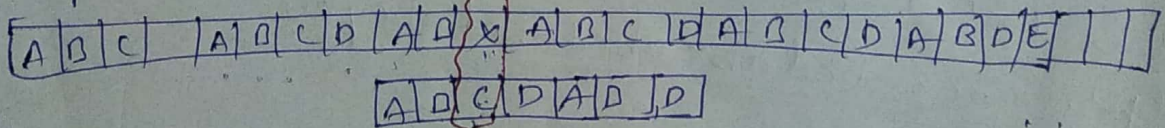
	0	1	2	3	4	5	6
	A	B	C	D	A	B	D
Lps()	0	0	0	0	1	2	0



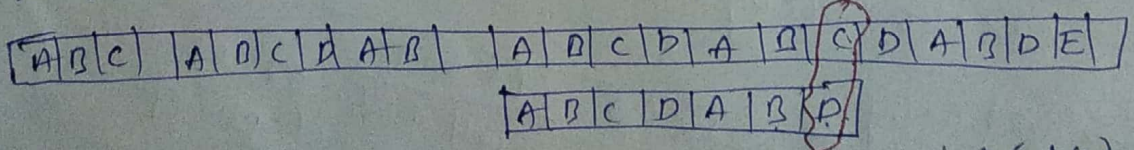
Here mismatch occured at pattern[3]. so take Lps[2] value i.e '0'. we must compare first character in the pattern with the ~~next~~ mismatched character of Text. i.e x with A, mismatch and Lps[0]=0, move right.



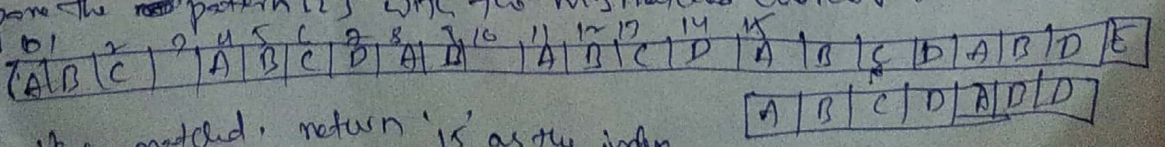
Here mismatch occured at pattern[6]. so take Lps[5] i.e '2'. (≠0) we compare pattern[2] with mismatched character in the text



Here, mismatch occured at pattern[3], so take Lps[2] i.e '0'. compare pattern[0] with the ~~next~~ mismatched character in the Text i.e 'A' with X, mismatch Lps[1]=0. so, Compare pattern[0] with 'X', move right



mismatch occured at pattern[6]. so take Lps[5] i.e '2'. (≠0) Compare the ~~next~~ pattern[2] with the mismatched character in the text



matched. return 'i' as the index

compute failure function(π) prefix table for the following strings

① abcaby ② aabaabaaa ③ abcdabcy

④ abcdabca

① Initialization

i=0, j=0
LPS(0)=0

	0	1	2	3	4	5
	a	b	c	a	b	y
LPS[]	0	0	0	1	2	0

②

	0	1	2	3	4	5	6	7	8
	a	a	b	a	a	b	a	a	a
	0	1	0	1	2	3	4	5	2

③

	0	1	2	3	4	5	6	7
	a	b	c	d	a	b	c	y
	0	0	0	0	1	2	3	0

④

	0	1	2	3	4	5	6	7
	a	b	c	d	a	b	c	a
	0	0	0	0	1	2	3	1

Text: abcx abcd abx abcdabcd abcyab

pattern: abcdabcy

Text: abxabc abcaby abcd

pattern: abcaby

Advantages: optimal & fast O(m+n)

disadvantages: Does not work so well as the size of the alphabets increases. By which more chances of mismatch occurs.

(3) Boyer-Moore pattern Matching Algorithm

It preprocesses the pattern based on two approaches
1. Bad character rule
2. Good Suffix rule

At every step, it moves the pattern by the maximum of the moves suggested by the two rules. So, it uses best of the two rules at every step. Unlike the previous pattern searching algorithms, Boyer-Moore algo starts matching from the last character of the pattern.

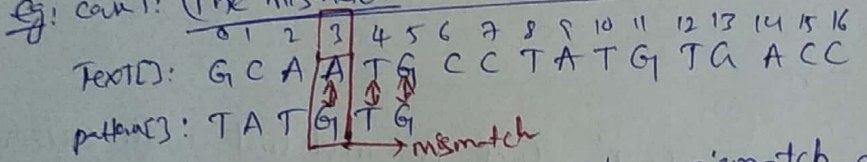
Bad character rule:

The character of the text which doesn't match with the current character of the pattern is called the Bad character. Upon mismatch, we shift the pattern until -

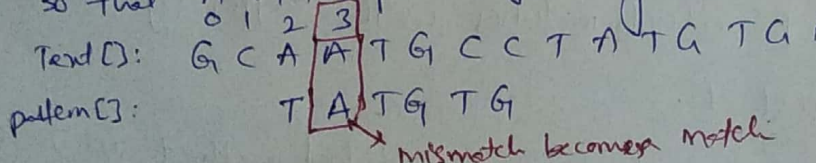
Case 1. The mismatch becomes a match

Case 2. pattern 'p' move past the mismatched character.

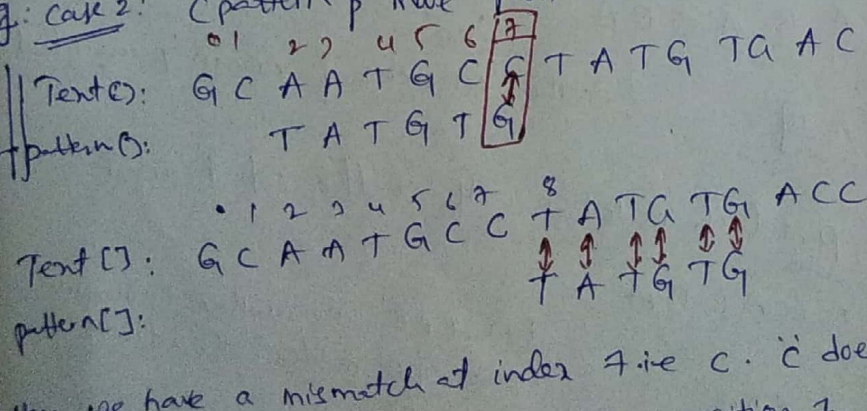
Eg: Case 1: (The mismatch becomes a match)



Start comparison from 5, we got a mismatch at index '3'. Here bad character is 'A'. So, we will search for last occurence of 'A' in the pattern. We got 'A' at index '1'. Now we will shift pattern so that 'A' in the pattern get aligned with 'A' in the text. i.e

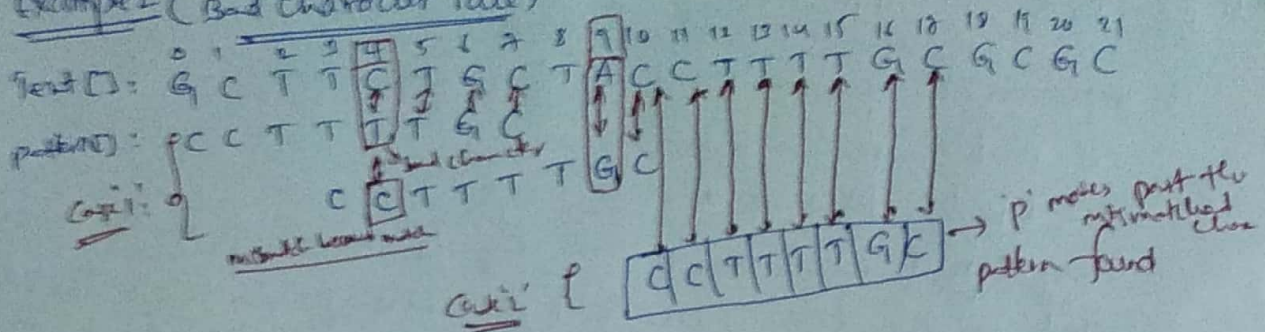


Eg: Case 2: (pattern 'p' move past the mismatched character)



Here, we have a mismatch at index 7 i.e C. 'C' does not exist in the pattern, so shift pattern past to the position 7.

Example (Bad character rule)



Ex 3: A B C A B C D A B A B C D A B C D A B D E

pattern: A B C D A B D

mismatch

Good suffix rule

Let 't' be substring of Text 'T' which is matched with substring of pattern 'p'. Now we shift pattern 'p' until:

Eg:

① Another occurrence of 't' in 'p' matched with 't' in T.

② A prefix of P, which matches with suffix of 't'

③ 'p' moves past 't'.

Case i: Another occurence of 't' in 'p' matched with t in 'T'

T: A B A A B A B A C B A

P: C A B A B

C A B A B

t = 'AB'

Another occurence of 'AB' in 'p' matched with 'T'

Case 2: A prefix of 'p' which matches with suffix of 't' in T

T: A A B A B A B A C B A

P: A B B A B

A B B A B

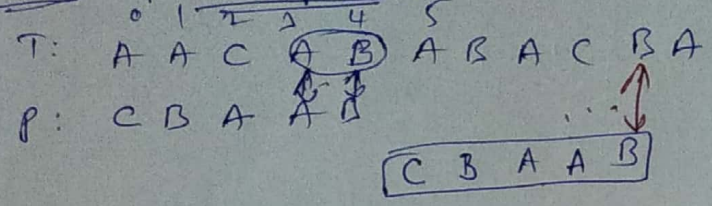
suffix

prefix

t is 'BAB', no another occurence prefix 'AB' matches with suffix of 't'

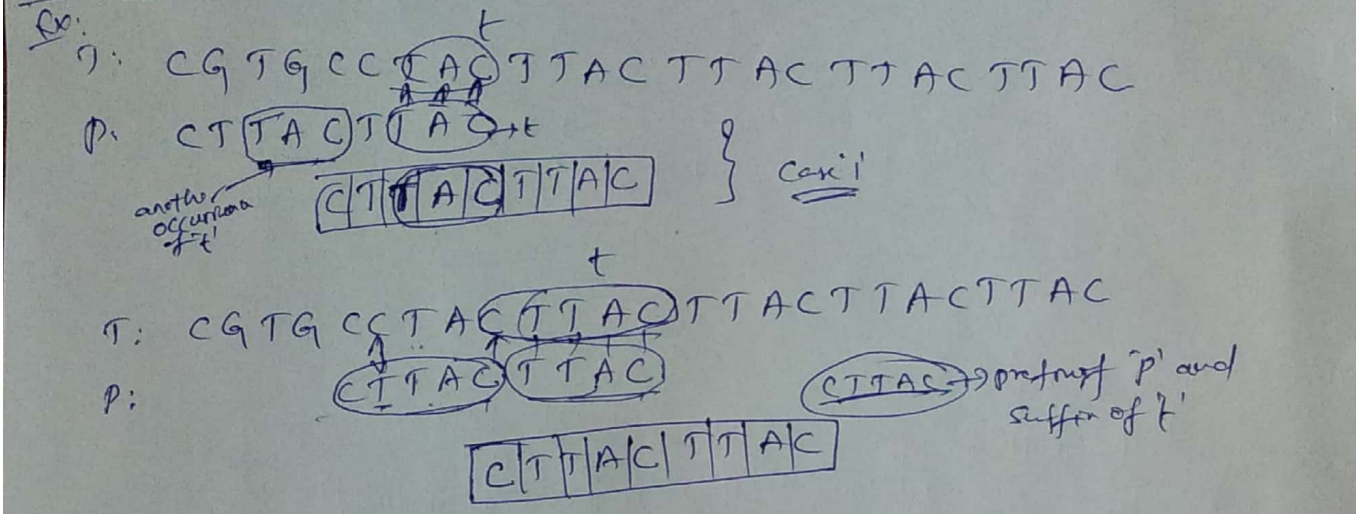
So, match

Case 3: 'p' moves past 't'

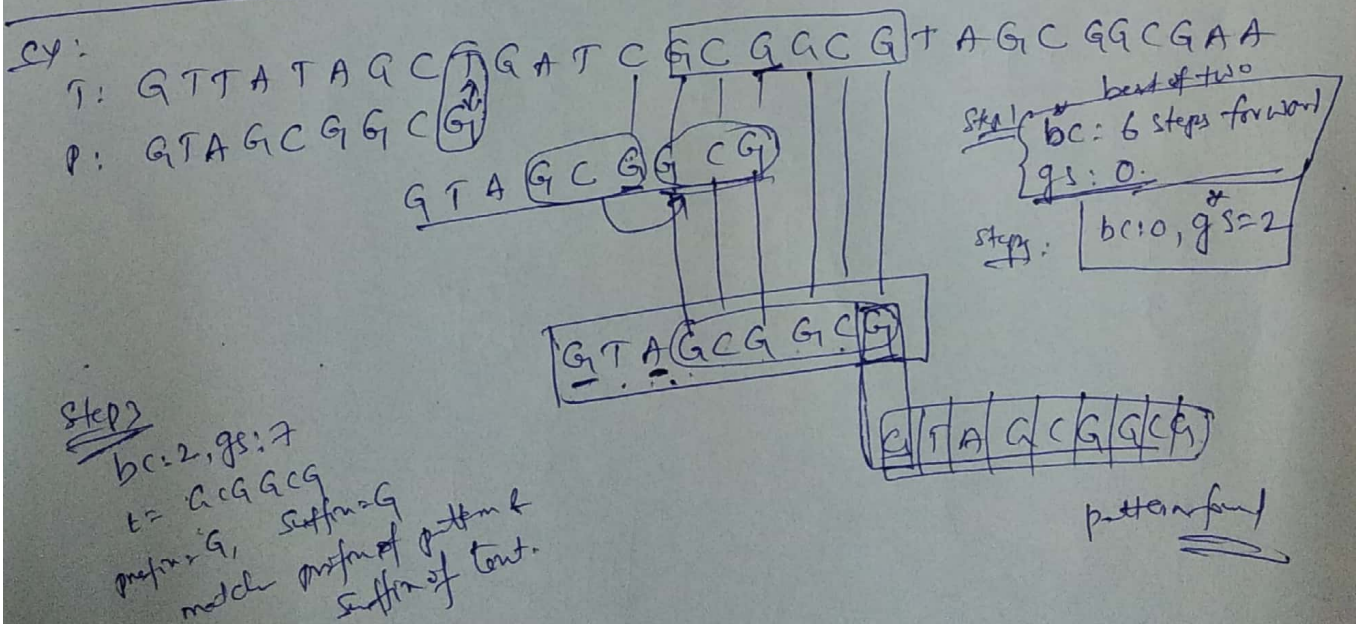


Here, there exist no occurrence of t("AB") in p and also there is no prefix in 'p' which matches with the suffix of t. So, we will shift the 'p' past the t.

Ex:



Ex:



Step2
bc:2, gs:7
t= GCGGCG
prefix: G, Suffix=G
match prefix of pattern & suffix of text.

pattern found