Ch: 8 String matching

↳ String matching algorithm are normally used in text processing. (Done in Compilation of Program).

↳ String Matching means finding one or more generally all the occurrences of a string in the text.

↳ These occurrence are called as pattern. Hence, Sometime string matching algorithm are also called as pattern matching algorithms.

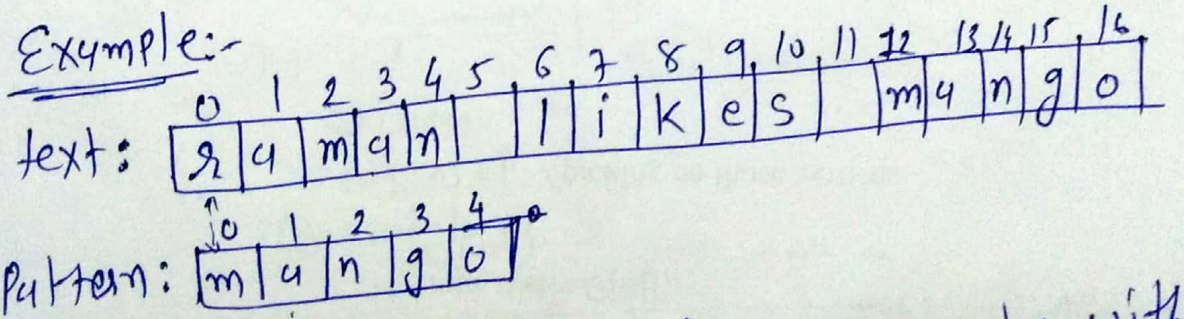
→ In this section we will discuss various string matching algorithms such as -

1. The Naive method
2. Rabin-Karp method
3. Finite Automata for string matching
4. KMP-Algorithm (Knuth Morris Pratt).

(1) The Naive method

↳ This is the simplest method which works using Brute force approach. (Straight forward approach).

Example:-



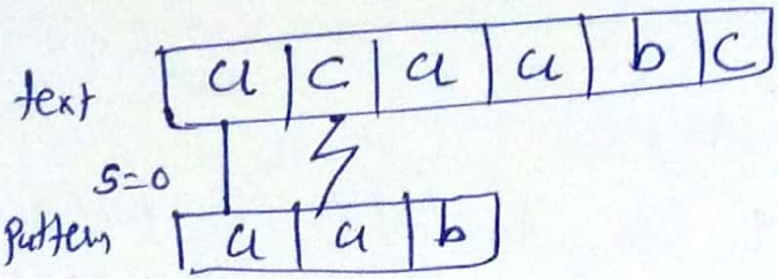
↳ Hence, Return index 12, because a match with the pattern is found from that location in text.

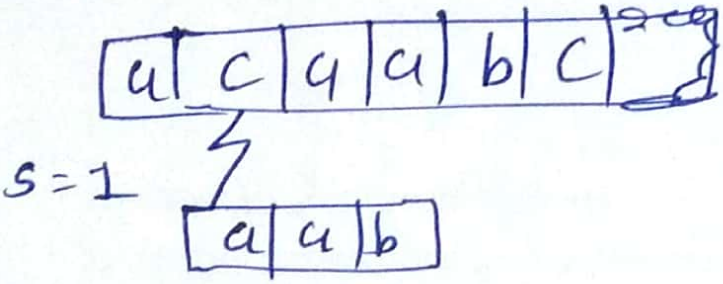
→ $\Theta(mn)$

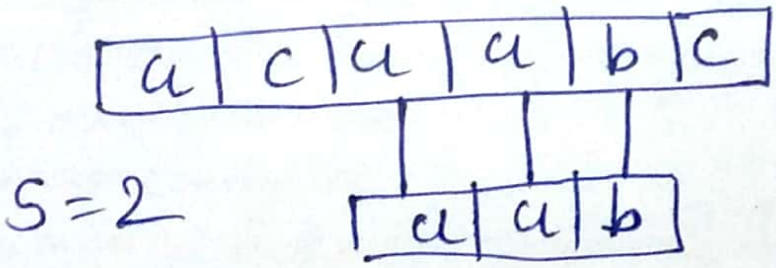
(1) The Naive String Matching algorithm

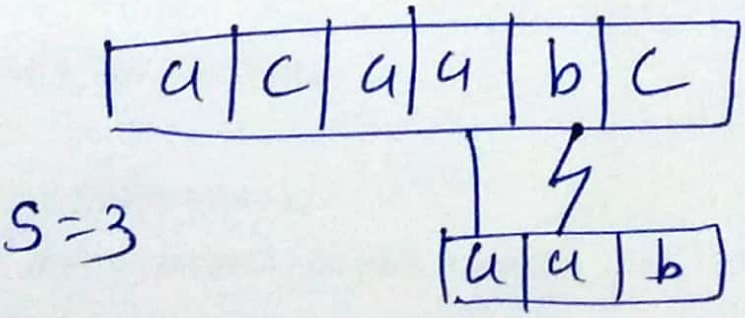
```
Naive-string-Matcher(T, P)
1. n = T.length
2. m = P.length
3. for s = 0 to n-m
4.       if P[1...m] == T[s+1 ... s+m]
5.             Print "Pattern occurs with shift" s.
```

Example of Naive method









↳ one occurrence of the pattern at shift $s=2$.

(2) The Rabin Karp Algorithm

↳ This algorithm assumes each character to be a digit in radix-d notation.

↳ $\Theta((n-m+1)m)$

Example Using the Rabin-Karp algorithm for text $T = 3141592653589793$ find for Pattern $P = 26$ with modulo $q = 11$.

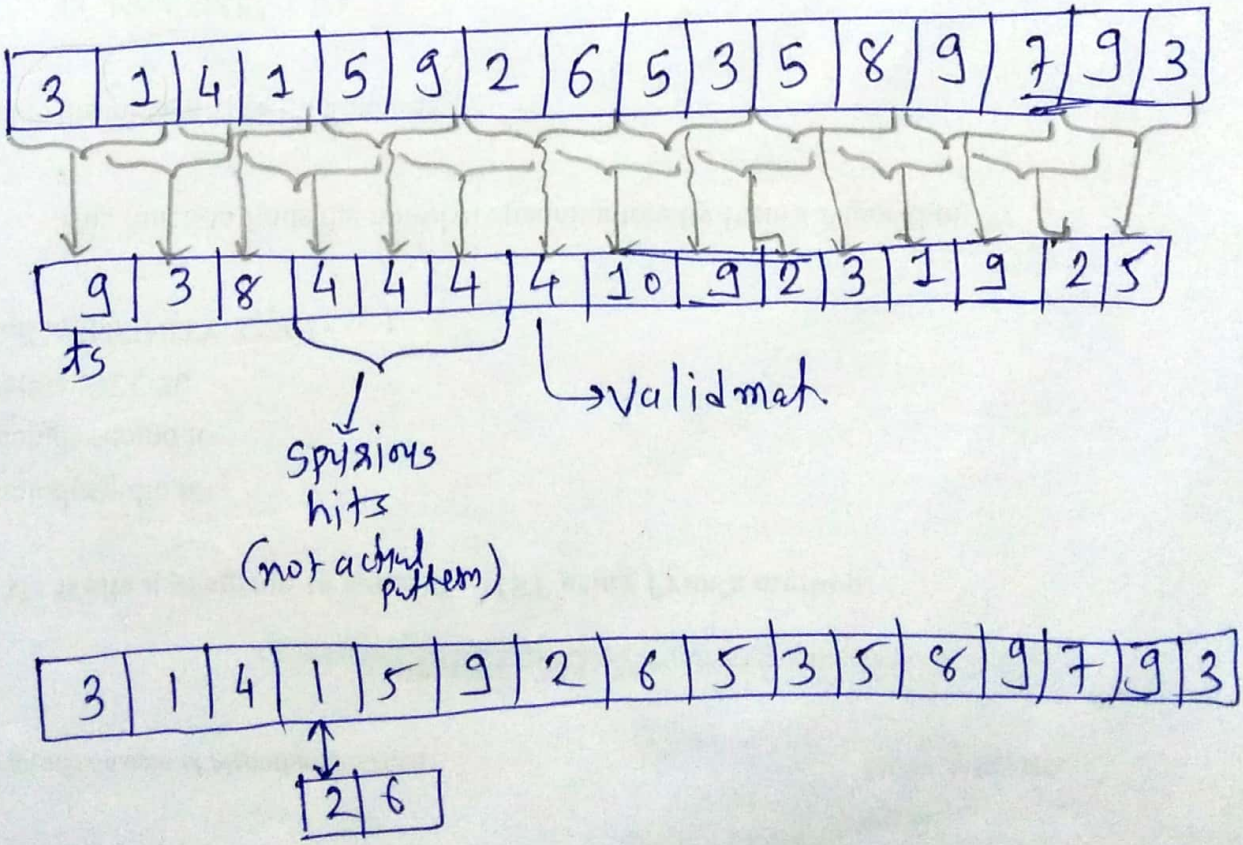
↳

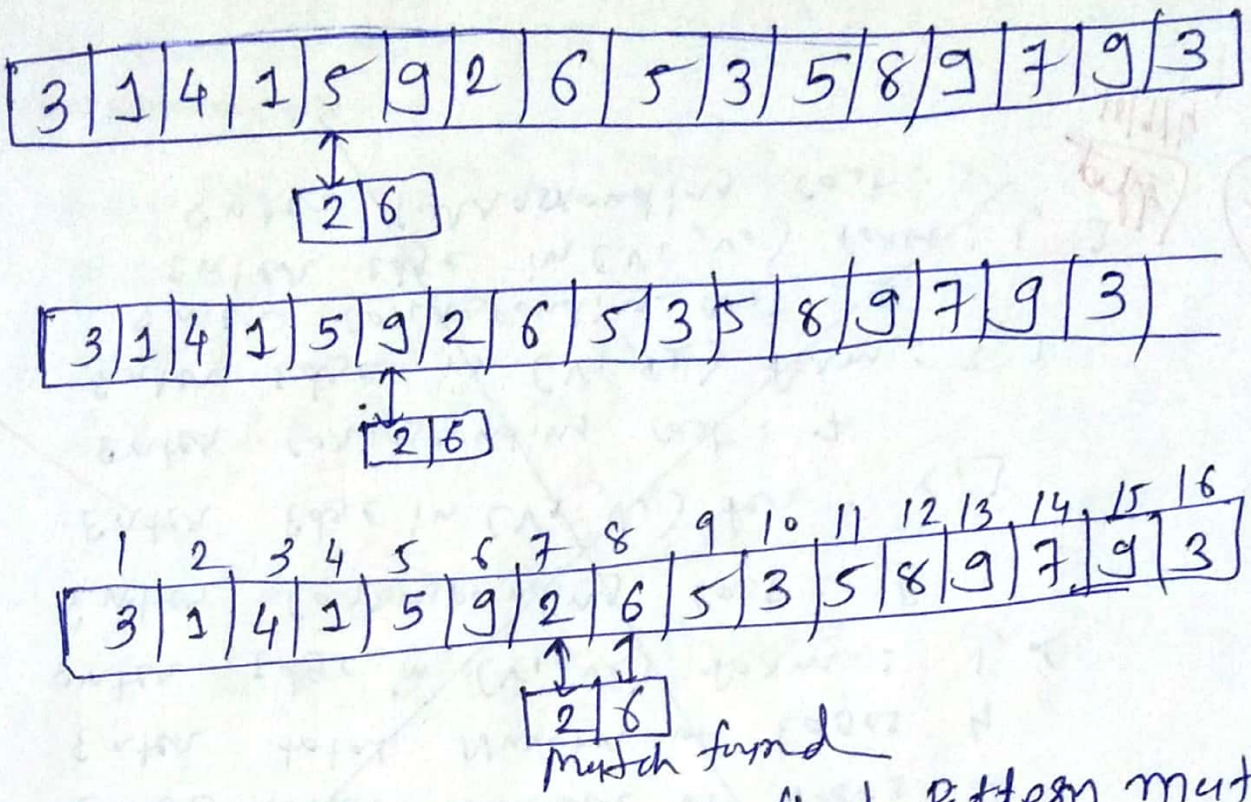
Text = | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 7 | 9 | 3 |

Pattern = | 2 | 6 |

Here $26 \bmod 11 = \underline{4} \leftarrow p$

↳ We will now obtain mod 11 value for each text.





↳ At 7th position we find Pattern matching.

Ex: Text = 2359023141526739921

Pattern = 31415.

$q = 13$

(2) The Rabin Karp Algorithm

Rabin-Karp-Matcher (T, P, d, q) (text, pattern, base of num.)

```
1.  n = T.length
2   m = P.length
3   h = d^(m-1) mod q        // hash value
4   p = 0
5   to = 0
6   for i=1 to m
7       p = (dp + P[i]) mod q      } modulo stored
8       to = (dto + T[i]) mod q    } in p & to
9   for s=0 to n-m
10.     if p == ts        // modulus value match!
11.         if P[1...m] == T[s+1...s+m]
12.             Print "Pattern occurs with shift" s
13.     if s < n-m
14.         ts+1 = (d(ts - T[s+1]h) + T[s+m+1]) mod q
```

↳ Running time of Algorithm $((n-m+1)m)$

(*) String Matching with Finite Automata

↳ Finite Automata is a very effective tool which is used in string matching algorithms.

↳ Definition of finite Automata :-

A finite automata is a five tuples.

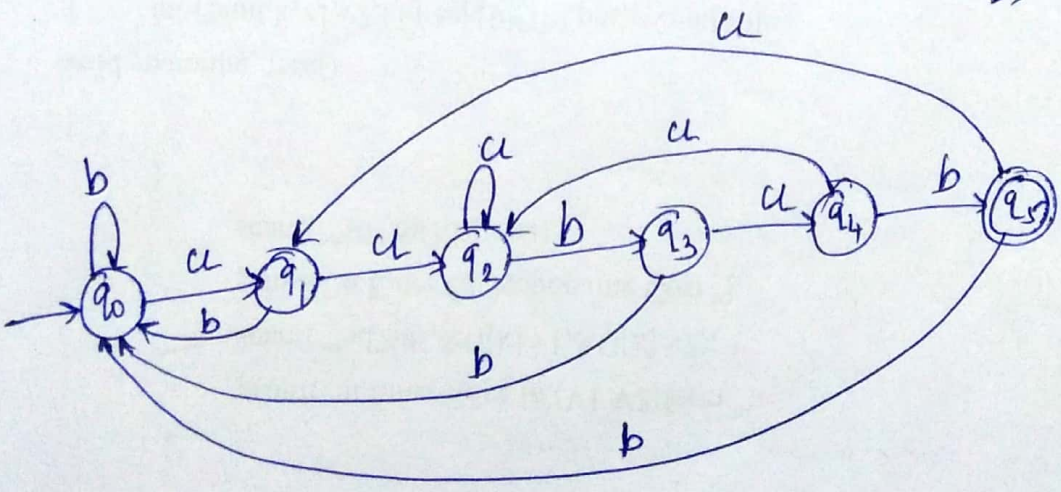
$M = (Q, \Sigma, q_0, A, \delta)$

Where
- Q is a set of states.
- q_0 is a start state
- A is a final or Accepting state
- Σ is a finite Input set
- δ is a function from $Q \times \Sigma \to Q$.

Example :- Text = aaabababaabaababaab.
Pattern = aababab.

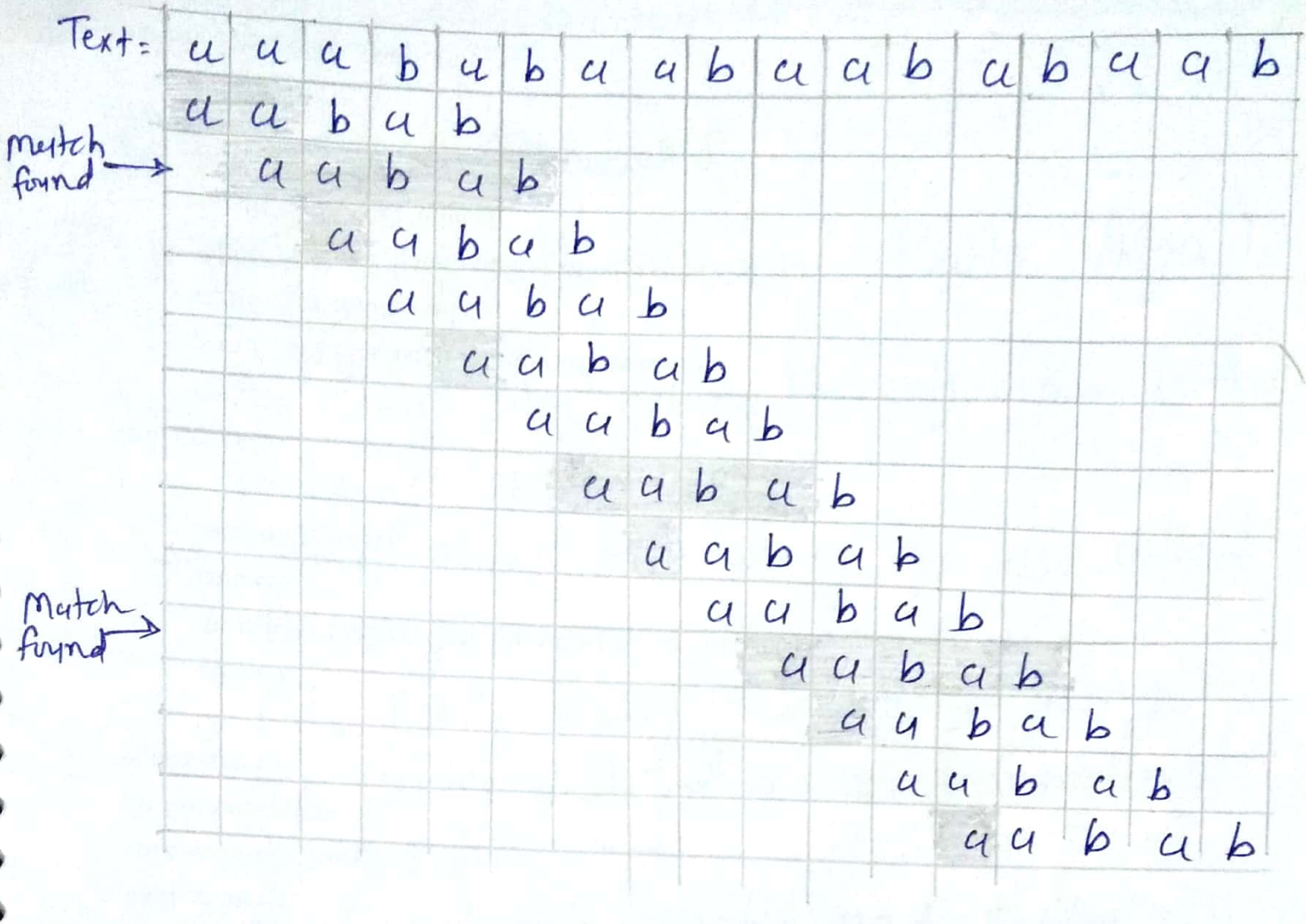
↳ We can Construct a finite Automata for Pattern = aabab.

→ Transition table



F.A.

	a	b
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_3
q_3	q_4	q_0
q_4	q_2	q_5
q_5	q_1	q_0



Finite Automaton. matcher (T, δ, m)

1. n = T.length
2. q = 0
3. for i = 1 to n
4. q = δ(q, T[i])
5. if q == m ← last position of pattern (accepting state)
6. print "Pattern occurs with shift" i-m

(*) KMP (Knuth Morris Pratt Algo.)

↳ The Basic idea behind this algorithm is to build a prefix array.

↳ Some times this array is also called π array.

↳ This prefix array is built using the prefix and suffix information of Pattern.

↳ The overlapping prefix and suffix is used in KMP algorithm.

Example : text = budbubybabuduab
Pattern = ababadu

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
text :-	b	a	d	b	a	b	a	b	a	b	a	d	a	a	b
Pattern :-	a	b	a	b	a	d	a								

↳ first find out prefix table.

→ Consider string "a"

prefix = ε
suffix = ε

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

↳ Consider string "ab"

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

Psefix = ∧, a
Suffix = ∧, b

↳ Consider string "aba"

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

Psefix = ∧, a, ab
Suffix = ∧, a, ba

↳ Consider string "abab"

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

Psefix = ∧, a, ab, aba
Suffix = ∧, b, ab, bab.

↳ Consider string "ababa"

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

Psefix = ∧, a, ab, aba, abab
Suffix = ∧, a, ba, aba, baba

↳ Consider string: "ababad"

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

Psefix = ∧, a, ab, aba, abab, ababa
Suffix = ∧, d, ad, bad, abad, babad

↳ Consider string "ababada"

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

Psefix = ∧, a, ab, aba, abab, ababa, ababad
Suffix = ∧, a, da, ada, bada, abada, babada

We will Compute the Psefix table for the Pattern "ababada".

	1	2	3	4	5	6	7
	0	1	2	3	4	5	6
P[i]	a	b	a	b	a	d	a
π[i]	0	0	1	2	3	0	1

→ Compare text[0] with Pattern[0]

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Text.	b	a	d	b	a	b	a	b	a	b	a	d	a	a	b

Pattern	a	b	a	b	a	d	a
	0	1	2	3	4	5	6

↳ Here b & a, as it is not matching.

↳ Compare text[1] with Pattern[0] {Psefix_table[0]=0

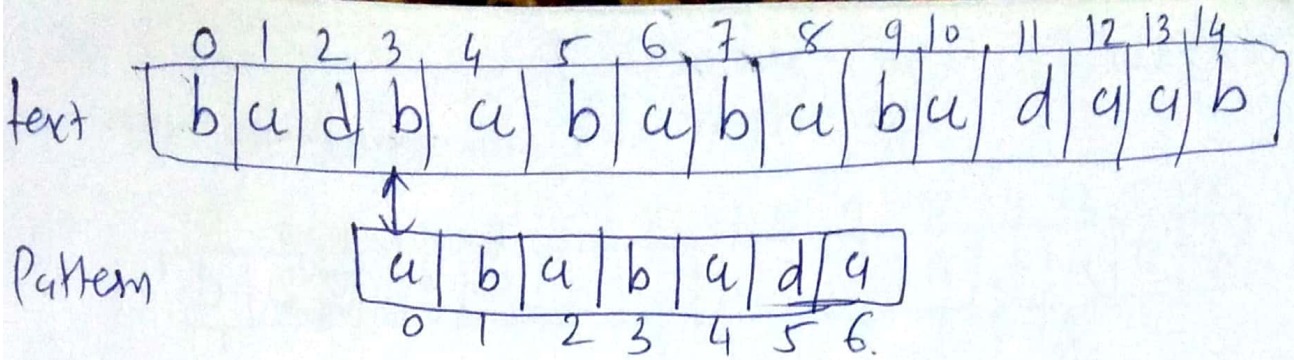
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
text	b	a	d	b	a	b	a	b	a	b	a	d	a	a	b

Pattern	a	b	a	b	a	d	a
	0	1	2	3	4	5	6

→ text[1] is matching with Pattern[0], We will now compare text (2) with Pattern[1]

↳ But Text[2] is not match with Pattern[1].

So, Back track on Pattern and Compare Pattern[0] with text(2) because we consult Psefix_table[0] = 0, So, Consider Pattern[0] is Compared with Text(2).

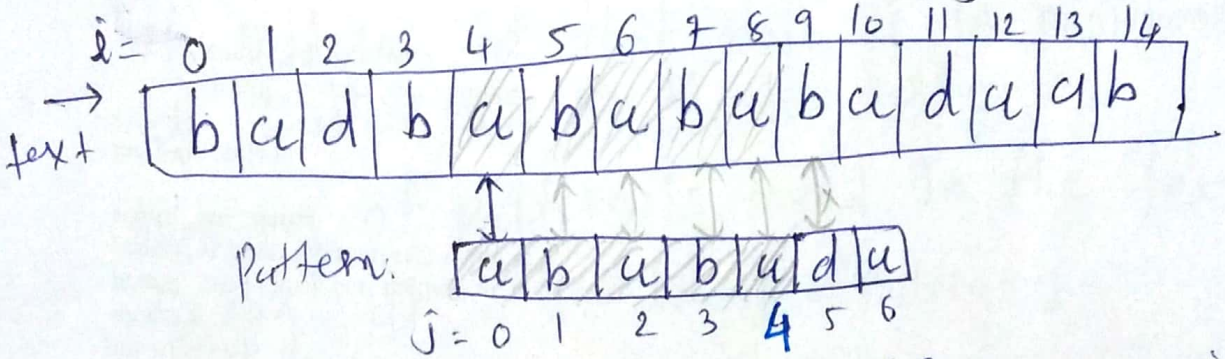


↳ Here text[3] is not matching with Pattern[0].

↳ We will then ask Psefix table[0] for the location of Pattern.

Psefix_table[0] = 0.

So, we will Compare pattern[0] with text[4].



- text[4] matches with Pattern[0], increment i & j
- text[5] matches with Pattern[1], increment i & j
- text[6] matches with Pattern[2], increment i & j
- text[7] matches with Pattern[3], increment i & j
- text[8] matches with Pattern[4], increment i & j.

But text[9] is not matching with Pattern[5].

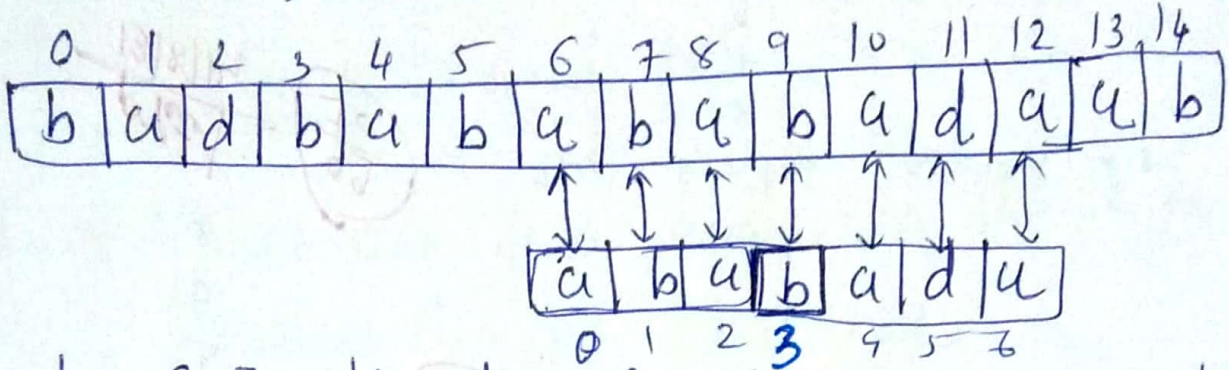
↳ Hence we must backtrack on Pattern, that means j will be positioning on location 4. (Pattern 5 to decrease 4)

→ Consider Psefit_table[4] = 3, → first three characters matched (k=3)

that indicates, Compare pattern[3] with current i position, means text[9]

text

→ Hence we will Compare text[9] with Pattern[3], which is matching.



- text[10] with Pattern[4], matching ∴ increment i and j
- text[11] with Pattern[5], matching ∴ increment i and j
- text[12] with Pattern[6], matching ∴ increment i and j.

↳ Thus, we have reached on the last character of Pattern, at that time i is positioned at location 12 in the text array.

matching of pattern is found in the text array at = i - length of Pattern + 1

$= 12 - 7 + 1$

$= 6$

↳ Thus required Pattern matches at location 6.

Ex: find Prefix table for "abadab"

a	b	a	d	a	b
0	0	1	0	1	2

⊛ The Knuth-Morris-Pratt Algorithm

```
KMP_Matcher (T, P)
1.  n = T.length
2.  m = P.length
3.  Π = 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 = Π[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 = Π[q]         // look for the next match
```

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 $P[k+1] \neq P[q]$
7. $k = \pi[k]$
8. if $P[k+1] == P[q]$
9. $k = k + 1$
10. $\pi[q] = k$
11. return π