Unit V: String Searching Algorithms: Introduction, Preliminaries, The Naive Algorithm, The Knutt-Morris-Pratt Algorithm, The Boyer-Moore Algorithm, The Shift-Or Algorithm, The Karp Rabin Algorithm.

INTRODUCTION

- String searching, also known as pattern matching, is a fundamental operation that involves finding the presence and position of a substring (called a pattern) within a larger string of text.
- The purpose of string searching is to determine whether the given pattern exists in the text and, if so, where it occurs. In its simplest form, the algorithm compares the pattern with every possible substring of the text
- String searching is an important component of many problems, including text editing, data retrieval, and symbol manipulation.
- The string searching or matching problem consists of finding all occurrences (or the first occurrence) of a pattern in a text, where the pattern and the text are strings over some alphabet.
- It is well known that to search for a pattern of length m in a text of length n (where $n > m$) the search time is $0(n)$ in the worst case (for fixed m). Moreover, in the worst case, at least $n - m + 1$ characters must be inspected.

PRELIMINARIES

We use the following notation:

- n: the length of the text
- m: the length of the pattern (string)
- c: the size of the alphabet *
- $\overline{C}_n$ the expected number of comparisons performed by an algorithm while searching the pattern in a text of length n

The probability of finding a match between a random text of length m and a random pattern of length m is

$$Prob\{match\} = \frac{1}{c^m}$$

The expected number of matches of a random pattern of length m in a random text of length n is

$$E[matches] = \begin{cases} \frac{n-m+1}{c^m}, & \text{if } n \geq m; \\ 0 & \text{otherwise} \end{cases}$$

THE NAIVE ALGORITHM

- The naive, or brute force, algorithm is the simplest string matching method.
- The idea consists of trying to match any substring of length m in the text with the pattern.
- Whenever a mismatch is detected in the comparison process, the input text is shifted one position, and the comparison process is initialized and restarted. The expected number

of comparisons when searching an input text string of n characters for a pattern of m characters is

$$N_c = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)(n-m+1) + O(1)$$

- where Nc is the expected number of comparisons and c is the size of the alphabet for the text.
- For search of any large streams the number of comparisons can be estimated by the number of characters being searched. For smaller items the length of the text pattern (m) can have an effect on the number of comparisons.
- Whenever a character mismatch occurs after matching of several characters, the comparison begins by going back in from the character which follows the last beginning character.

```
T : c a b a b a b c d

P : a...

P :   a b a b c

P :     a...

P :       a b a b c
```

- Naïve algorithm checks **every position** of the text for a possible match.
- It does **not skip** any characters.
- Match found when **all characters** of the pattern align with the text substring.
- In this example, **pattern "ababc"** is found at **position 4**.

The Knuth-Pratt-Morris algorithm made a major improvement in brute force algorithms in that even in the worst case it does not depend upon the length of the search term and does not require comparisons for every character in the input stream.

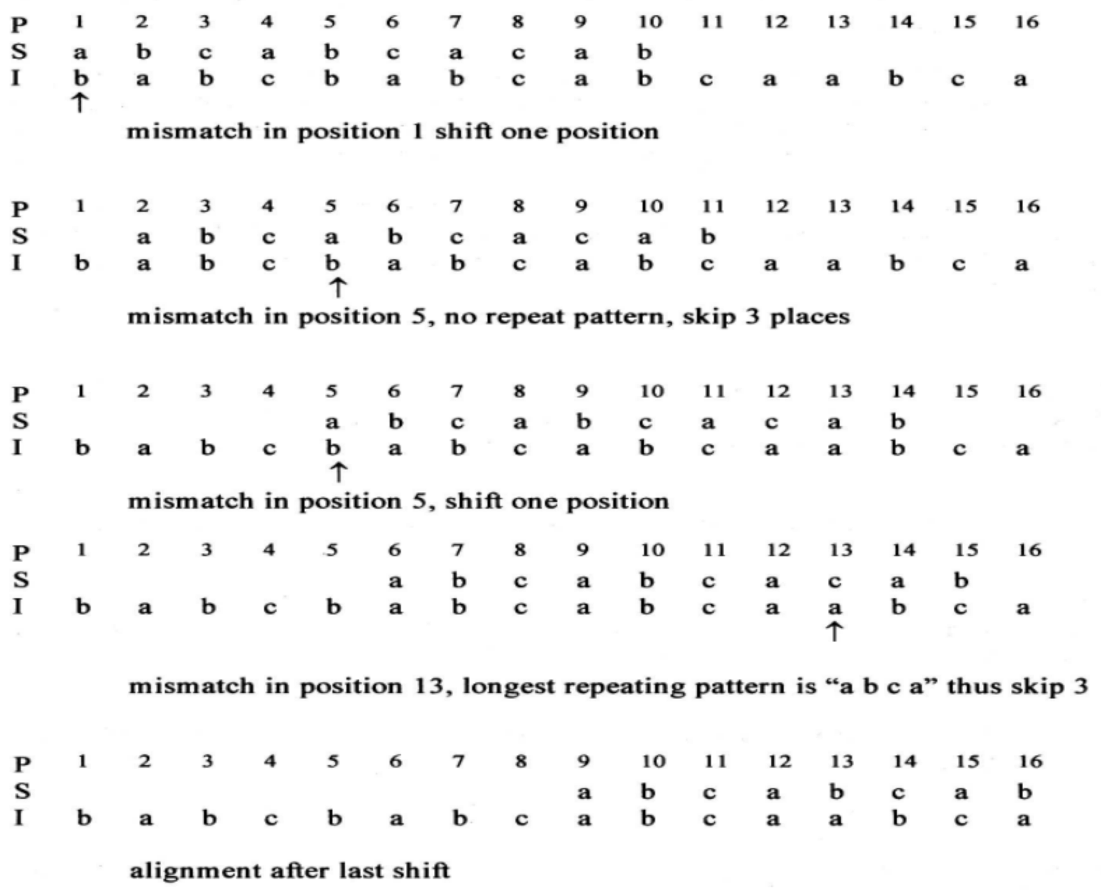
<u>THE KNUTT- MORRIS-PRATT ALGORITHM</u>

The KMP algorithm avoids this inefficiency of **Naive algorithm** by preprocessing the pattern using an auxiliary array called LPS (Longest Prefix Suffix). This array stores the length of the longest proper prefix which is also a suffix for every prefix of the pattern. The basic concept behind the algorithm is that whenever a mismatch is detected, the previous matched characters define the number of characters that can be skipped in the input stream prior to starting the comparison process again.

Example:

```
Position          1 2 3 4 5 6 7 8
Input Stream    = a b d a d  e f g
Search Pattern  = a b d f
```

- The algorithm allows the comparison to jump at least tile 3 positions associated with the recognized “a b d”.



```
P   1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
S   a  b  c  a  b  c  a  c  a  b
I   b  a  b  c  b  a  b  c  a  b  c  a  a  b  c  a
    ↑
    mismatch in position 1 shift one position

P   1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
S      a  b  c  a  b  c  a  c  a  b
I   b  a  b  c  b  a  b  c  a  b  c  a  a  b  c  a
                ↑
    mismatch in position 5, no repeat pattern, skip 3 places

P   1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
S               a  b  c  a  b  c  a  c  a  b
I   b  a  b  c  b  a  b  c  a  b  c  a  a  b  c  a
                ↑
    mismatch in position 5, shift one position

P   1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
S                  a  b  c  a  b  c  a  c  a  b
I   b  a  b  c  b  a  b  c  a  b  c  a  a  b  c  a
                                       ↑
    mismatch in position 13, longest repeating pattern is “a b c a” thus skip 3

P   1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
S                           a  b  c  a  b  c  a  b
I   b  a  b  c  b  a  b  c  a  b  c  a  a  b  c  a

    alignment after last shift
```

Figure 9.4 Example of Knuth-Morris-Pratt Algorithm

The KMP algorithm works in two main steps:

1. **Preprocessing Step – Build the LPS Array**

- First, we process the pattern to create an array called LPS (Longest Prefix Suffix).
- This array tells us: "If a mismatch happens at this point, how far back in the pattern can we jump without missing any potential matches?"
- It helps us avoid starting from the beginning of the pattern again after a mismatch.
- This step is done only once, before we start searching in the text.

LPS (Longest Prefix Suffix) Array

- The LPS array stores, for every position in the pattern, the length of the longest proper prefix which is also a suffix of the substring ending at that position.

Example: Pattern "abcdabca"

At index 0: lps[0] = 0
At index 1: lps[1] = 0
At index 2:lps[2] = 0
At index 3: lps[3] = 0 (no repetition in "abcd")
At index 4: lps[4] = 1 ("a" repeats)
At index 5: lps[5] = 2 ("ab" repeats)
At index 6: lps[6] = 3 ("abc" repeats)
At index 7: lps[7] = 1 (mismatch, fall back to "a")
Final LPS: [0, 0, 0, 0, 1, 2, 3, 1]

2. Matching Step – Search the Pattern in the Text

- Now, we start comparing the pattern with the text, one character at a time.
- If the characters match: Move forward in both the text and the pattern.
- If the characters don't match:
    - ➢ If we're not at the start of the pattern, we use the LPS value at the previous index (i.e., lps [j - 1]) to move the pattern pointer j back to that position. This means: jump to the longest prefix that is also a suffix — no need to recheck those characters.
    - ➢ If we're at the start (i.e., j == 0), simply move the text pointer i forward to try the next character.
- If we reach the end of the pattern (i.e., all characters matched), we found a match! Record the starting index and continue searching.

EXAMPLE:

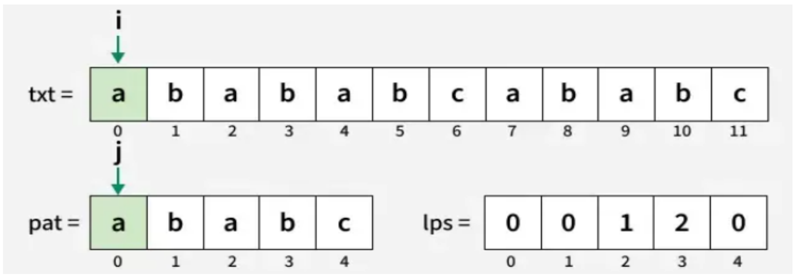
text = "abababcababc", pattern = "ababc"

LPS for Pattern = ababc

At index 0: lps[0] = 0
At index 1: lps[1] = 0
At index 2: lps[2] = 1
At index 3: lps[3] = 2
At index 3: lps[4] = 0

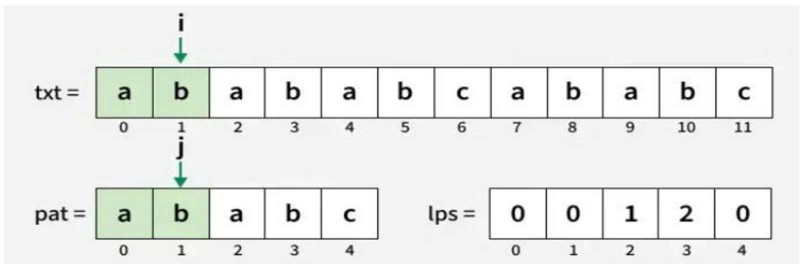
Final LPS: [0, 0, 1, 2, 0]

Step 1: Compare position 0 in text ('a') and position 0 in pattern ('a').



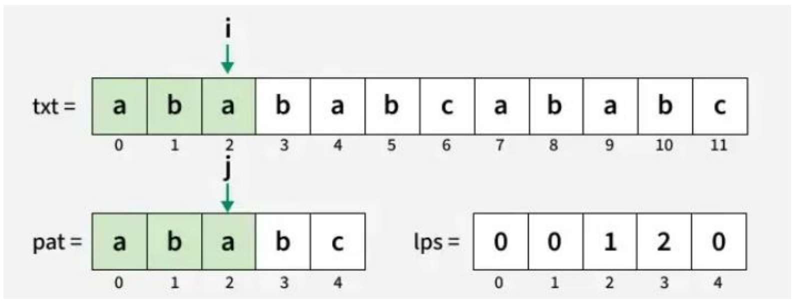
They are equal. Increment i and j → i = 1, j = 1.

Step 2: Compare position 1 in text (‘b’) and position 1 in pattern (‘b’).



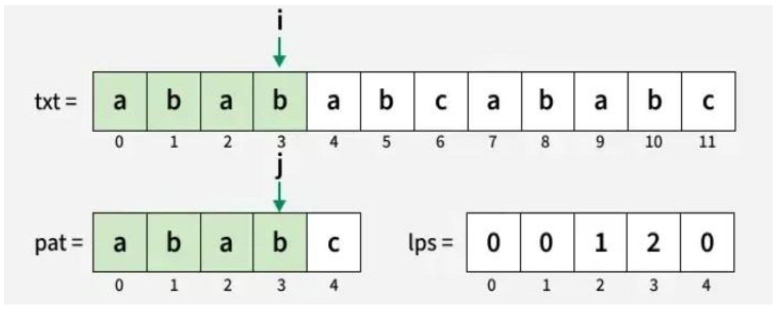
They match. Increment i and j → i = 2, j = 2.

Step 3: Compare position 2 in text (‘a’) and position 2 in pattern (‘a’).



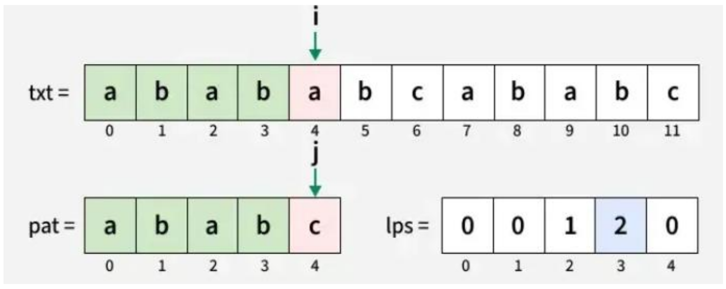
Match. Increment i and j → i = 3, j = 3.

Step 4: Compare position 3 in text (‘b’) and position 3 in pattern (‘b’).



Match. Increment i and j → i = 4, j = 4.

Step 5: Compare position 4 in text (‘a’) and position 4 in pattern (‘c’).

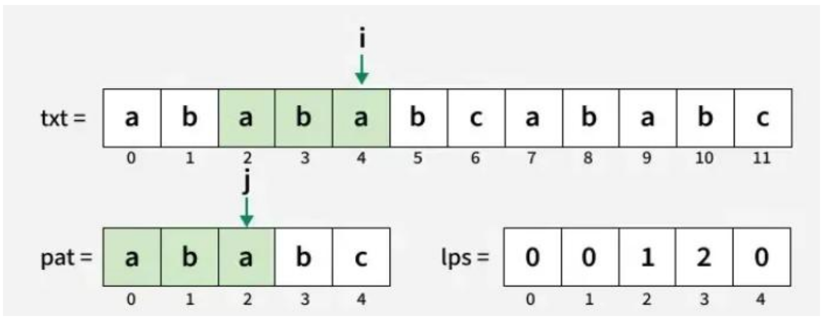


Mismatch.

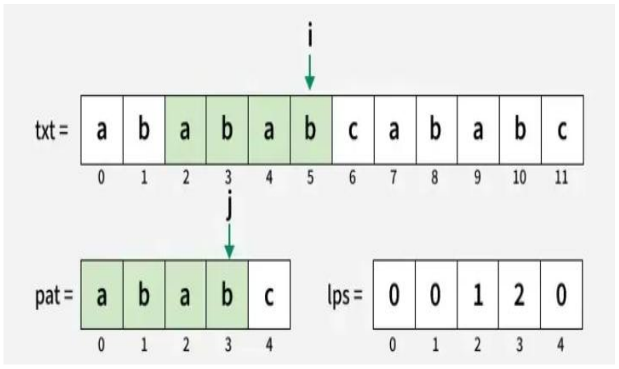
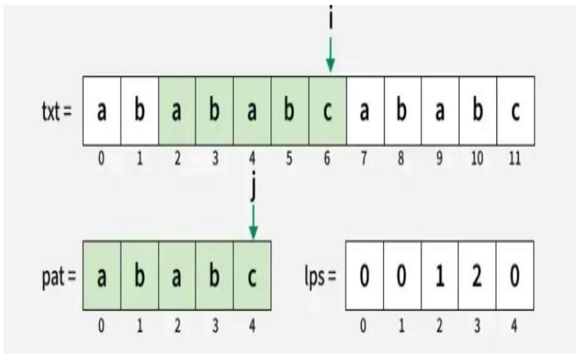
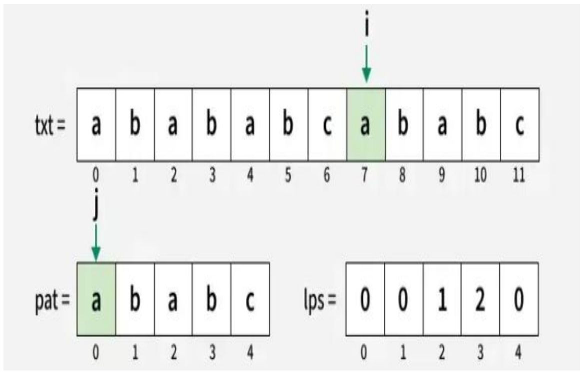
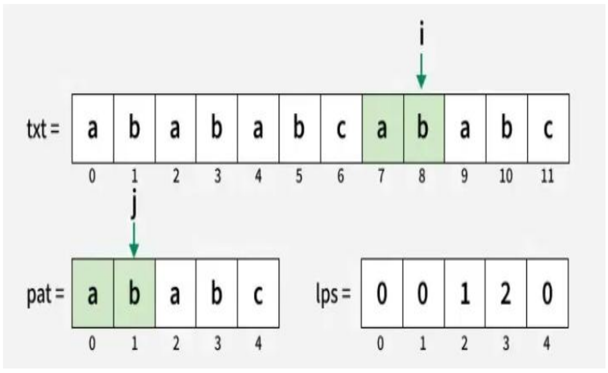
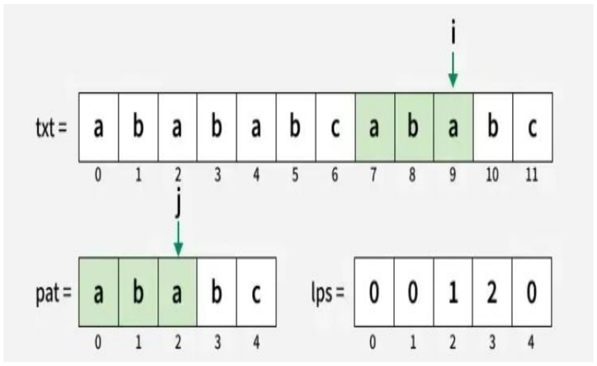
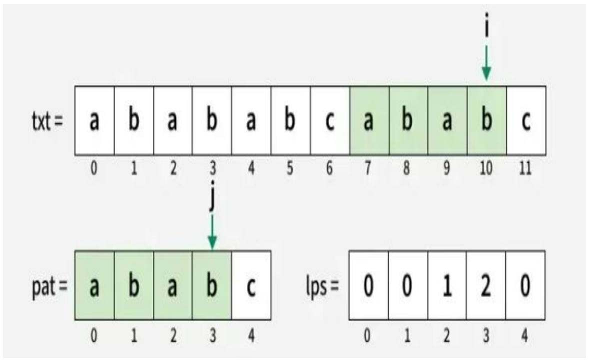
Since j ≠ 0, we don’t increment i.

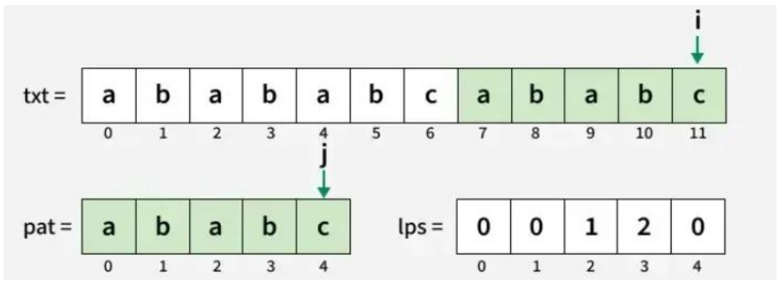
Use LPS: j = lps[j−1] = lps[3] = 2.

we *use the LPS array* to skip unnecessary re-comparisons.



Compare remaining positions to find all the occurrences of the Pattern.



Final Output:

- **LPS Array:** [0, 0, 1, 2, 0]
- **Pattern found at indices:** 2 and 7
- **Total Matches:** 2

BOYER MOORE ALGORITHM

- The Boyer Moore algorithm starts matching from the last character of the pattern.
- At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics.
- The shift can be computed using two heuristics:
  1. Bad Character Heuristic
  2. Good Suffix Heuristic

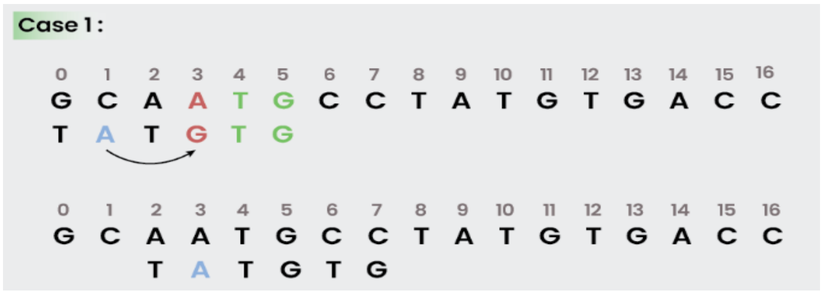
1. Bad Character Heuristic

The idea of bad character heuristic is simple. 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 -

1. The mismatch becomes a match.
2. Pattern P moves past the mismatched character.

<u>Case 1</u> – Mismatch become match

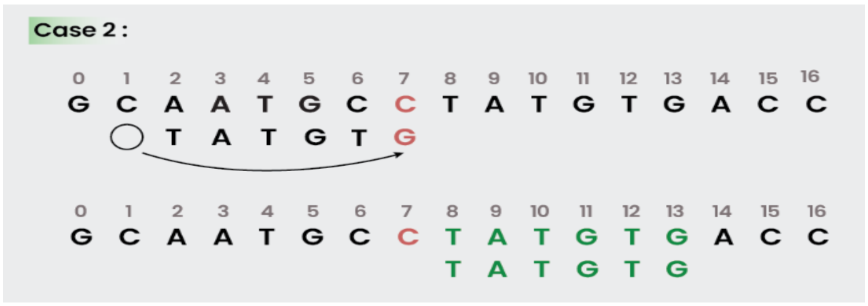
We will look up the position of the last occurrence of the mismatched character in the pattern, and if the mismatched character exists in the pattern, then we'll shift the pattern such that it becomes aligned to the mismatched character in the text T.



- In the above example, we got a mismatch at position **3**.
- Here our mismatching character is "A". Now we will search for last occurrence of "A" in pattern. We got "A" at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that "A" in pattern get aligned with "A" in text.

Case 2 – Pattern move past the mismatch character

- We'll look up the position of last occurrence of mismatching character in pattern and if character does not exist, we will shift pattern past the mismatching character.



- The mismatching character "C" does not exist in pattern before position 7 so we'll shift pattern past to the position 7.

2. Good Suffix Heuristic for Pattern Searching

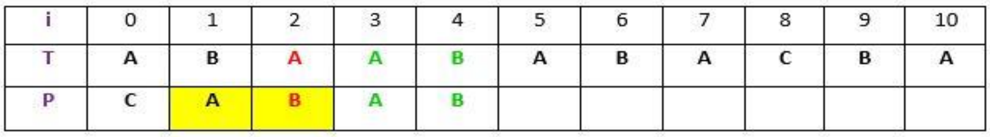
- The **Strong Good Suffix Heuristic** is an important optimization in the **Boyer-Moore** algorithm for string pattern matching. It helps to skip unnecessary comparisons and efficiently shift the pattern when a mismatch occurs.
- Let t be a substring of the text T that matches a substring of the pattern P. When a mismatch occurs, the pattern is shifted based on the following rules:

1. **Another occurrence of t in P**
2. **Prefix of P matches the suffix of t**
3. **Move the pattern past t**

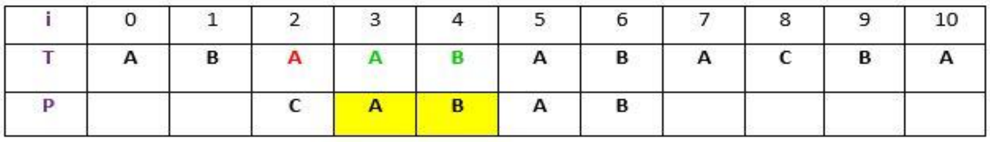
Case 1: Another occurrence of t in P matched with t in T

- If there are other occurrences of the substring t in the pattern P, the pattern is shifted so that one of these occurrences aligns with the substring t in the text T. This allows the pattern to continue matching efficiently.

For example-



i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						



i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

In the above example, we have a substring t of text T that matches with pattern P (in green) before the mismatch occurs at index 2. Now, we search for occurrences of t ("AB") within P.

We find an occurrence starting at position 1 (highlighted with a yellow background). Therefore, we right shift the pattern by 2 positions to align the occurrence of t in P with t in T.

Case 2: A prefix of P, which matches with suffix of t in T

- It is not always guaranteed that we will find an occurrence of t in P. In some cases, there may be no exact occurrence at all. In such situations, we can instead look for a **suffix of t** that matches with a **prefix of P**. If a match is found, we can shift the pattern P to align the matched suffix of t with the prefix of P, allowing us to continue the search effectively. This method helps when direct matches for t are not found, and it ensures that the search can still progress efficiently.

For example –

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Figure – Case 2

In above example, we have got t (“BAB”) matched with P (in green) at index 2-4 before mismatch. But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t. We have found prefix “AB” (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t “AB” starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.

Case 3: P moves past t

If the above two cases are not satisfied, we will shift the pattern past the t.

For example -

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Figure – Case 3

If above example, 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, in that case, we can never find any perfect match before index 4, so we will shift the P past the t i.e., to index 5.

SHIFT-OR ALGORITHM

The Shift-Or algorithm (also known as the Bitap algorithm or Shift-And algorithm) is an efficient string searching algorithm used to find occurrences of a pattern within a text. The Shift-Or algorithm is a fast-string matching technique that represents the pattern as a set of bitmasks, where each bit corresponds to a character position in the pattern. It processes the text one character at a time, updating a bitmask that represents how well the current part of the text matches the pattern. The algorithm maintains a bit vector called state (D) that tracks the current matching progress. A match is detected when the most significant bit becomes 0.

EXAMPLE:

Text: *aabaacaadaaba*

Pattern: *aaba*

Bitmask Table for the pattern – aaba

Step 1: Construct Bitmask Table

- Each character of the pattern is represented by a bitmask.
- Bit positions correspond to the pattern positions (from right to left).

Character	**a**	**a**	**b**	**a**	**Bitmask**			
	b1	**b2**	**b3**	**b4**	**b4**	**b3**	**b2**	**b1**
a	0	0	1	0	0	1	0	0
b	1	1	0	1	1	0	1	1
*	1	1	1	1	1	1	1	1

Step 2: Finite Automata Representation

The automaton simulates matching pattern characters

Finite Automata for the pattern:

$$\begin{tikzcd}
{} \arrow[r] & q0 \arrow[r, "a"] & q1 \arrow[r, "a"] & q2 \arrow[r, "b"] & q3 \arrow[r, "a"] & q4
\end{tikzcd}$$

Step 3: Initialize State (D)

Start with D = 1111 — meaning no characters have been matched yet.

Each bit in D shows which prefixes of the pattern may have matched so far.

Step 4: Process Each Character in Text

For each text character T_j:

- Left shift the state: D << 1
- Take the bitmask for the current character: $B[T_j]$

- Combine using OR operation: (D << 1) | B[T_j]

  Where:

  T_j: Current character in the Text

  B [T_j]: Bitmask for character T_j

  D: Current state of matching

  TEXT:

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

State (D)	T_j	B[T_j]	D<<1	(D<<1) | B[T_j]	Matched
1 1 1 1	a	0 1 0 0	1 1 1 0	1 1 1 0	Yes
1 1 1 0	a	0 1 0 0	1 1 0 0	1 1 0 0	Yes
1 1 0 0	b	1 0 1 1	1 0 0 0	1 0 1 1	Yes
1 0 1 1	a	0 1 0 0	0 1 1 0	0 1 1 0	Yes

Pattern **"aaba" found at position 1**

Step 5: Searching for the Next Occurrence

Now the next search starts from **position 2** (text index 2).
We again initialize **D = 1111** and continue the same process.

State (D)	T_j	B[T_j]	D<<1	(D<<1) | B[T_j]	Matched
1 1 1 1	a	0 1 0 0	1 1 1 0	1 1 1 0	Yes
1 1 1 0	b	1 0 1 1	1 1 0 0	1 1 1 1	No

- No full match for the pattern starting at text positions 2...9.
- Second occurrence found **starting at position 10**

Conclusion:

- First occurrence: positions **1–4**
- Second Occurrence: positions **10-13**

KARP RABIN ALGORITHM

- The main idea behind Rabin-Karp is to convert strings into numeric hashes so we can compare numbers instead of characters. This makes the process much faster.
- To do this efficiently, we use a **rolling hash**, which allows us to compute the hash of the next substring in constant time by updating the previous hash.
- So instead of rechecking every character, we just slide the **window** and adjust the hash.
- We first compute the hash of the pattern, then compare it with the hashes of all substrings of the text.
- If the hashes match, we do a quick character-by-character check to confirm (to avoid errors due to hash collisions).

- A string is converted into a numeric hash using a polynomial rolling hash. For a string s of length n, the hash is computed as:

  $hash(s) = (s[0] * p^{(n-1)} + s[1] * p^{(n-2)} + ... + s[n-1] * p^{0}) \bmod q$

Where:

s[i] *is the numeric value of the* i^{th} *character ('a' = 1, 'b' = 2, ..., 'z' = 26)*

p *is any small integer*

q *is a large prime number to avoid overflow and reduce hash collisions*

To calculate hash value of first window, hash(s) formula is used.

Hash of next window is calculated using rolling hash function.

Rolling Hash Function:

hash(s+1) = (((h(s)-s[i] mod q) * p) mod q + s[i+m]) mod q

Where:

h(s) = old hash value

s[i] =character value leaving the window

s[i+m] = new character value entering the window

- Compare the rolling hash value of every window with pattern hash value.
- If they are not equal move to next window and repeat the procedure
- If they are equal compare each character in the window with each character in the pattern, if same they are matched else not matched.

EXAMPLE:

Text: *a b c d e f g h i j k*

Pattern: *d e f g*

Step:1 - Find the hashing value of the pattern

$hash(s) = (s[0] * p^{(n-1)} + s[1] * p^{(n-2)} + ... + s[n-1] * p^{0}) \bmod q$

hash(pattern) = hash (d e f g)

$= (4*10^{3} + 5*10^{2} + 6*10^{1} + 7*10^{0}) \bmod 113$

= (4000 + 500 + 60 + 7) mod 113

=47

Srep:2 - Hash of a window of text of length 4, as the *window size* is **equal to the length of the pattern**.

a	b	c	d	e	f	g	h	i	j	k
1	2	3	4	5	6	7	8	9	10	11

Calculating the Initial Window Hash

- To calculate the hash value for the **first window of the text ("abcd")**, we use **the same formula as we used for the pattern.**

  hash (abcd) = $(1*10^3 + 2*10^2 + 3*10^1 + 4*10^0)$ mod 113
  = (1000 + 200 + 30 + 4) mod 113
  = 1234 mod 113
  hash (abcd) = 104
- Now, compare the hash value of the **initial text window (104)** with the **pattern hash (47)**.
- Since the hash values are **not equal**, there is **no match** in this window.
- **Move to the next window** in the text and compute the new hash using the rolling hash formula.

Step:3 - We apply the rolling formula to calculate the values of remaining windows from position 2. The leading digit removed is a = 1, the new digit added is e = 5.

Rolling Hash Function:

*hash(s+1) = (((h(s)-s[i] mod q) * p) mod q + s[i+m]) mod q*

hash (bcde) = (((h(abcd) − $1*10^3$ mod 113) * 10) mod 113 + 5) mod 113

= (((104 – 1000 mod 113) * 10) mod 113 + 5) mod 113

= (((104 – 96) * 10) mod 113 + 5) mod 113

= (80 mod 113 + 5) mod 113

= 85 mod 113

= 85

- Now, compare the hash value of the **initial text window (85)** with the **pattern hash (47)**.
- Since the hash values are **not equal**, there is **no match** in this window.
- **Move to the next window** in the text and compute the new hash value.

Step:4 - The leading digit removed is b = 2, the new digit added is f = 6.

hash (cdef) = (((h(bcde) − $2*10^3$ mod 113) * 10) mod 113 + 6) mod 113

= (((85 – 2000 mod 113) * 10) mod 113 + 6) mod 113

= (((85 – 79) * 10) mod 113 + 6) mod 113

= ((60 mod 113) + 6) mod 113

= 66 mod 113

= 66

- Now, compare the hash value of the **initial text window (66)** with the **pattern hash (47)**.

- Since the hash values are **not equal**, there is **no match** in this window.
- **Move to the next window** in the text and compute the new hash value.

Step:4 - The leading digit removed is c = 3, the new digit added is g = 7.

hash (defg) = (((h(cdef) − $3*10^3$ mod 113) * 10) mod 113 + 7) mod 113

= (((66 – 3000 mod 113) * 10) mod 113 + 7) mod 113

= (((66 – 62) * 10) mod 113 + 7) mod 113

= ((40 mod 113) + 7) mod 113

= 47 mod 113

= 47

- Now, compare the hash value of the **initial text window (47)** with the **pattern hash (47)**.
- Since the hash values are **equal,** the **pattern is found at that position 4** in the text

Now continue the procedure to check for other occurrences.

Conclusion:

The pattern occurred at position 4 in the text