Computer Engineering - Sem IV

NCMPC 41 : Design and Analysis of Algorithms

Module - 3 : String Matching Algorithms (03 Hours)

Instructor : Mrs. Lifna C S

Department of Computer Engineering, VESIT, Mumbai

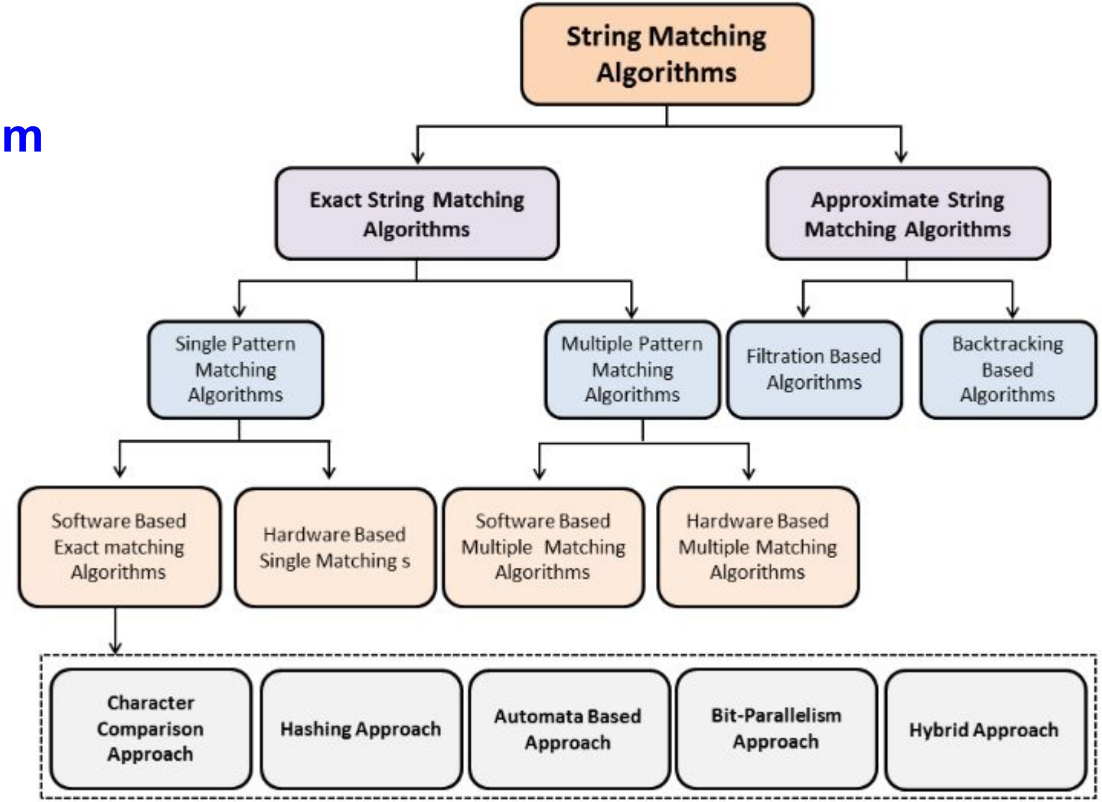
Topics to be covered

String Matching Algorithms

- **Naïve string-matching algorithm**
- Rabin Karp algorithm
- Knuth-Morris-Pratt algorithm

<u>Extra Topics</u>

- Finite State Automata
- Applications of String Matching

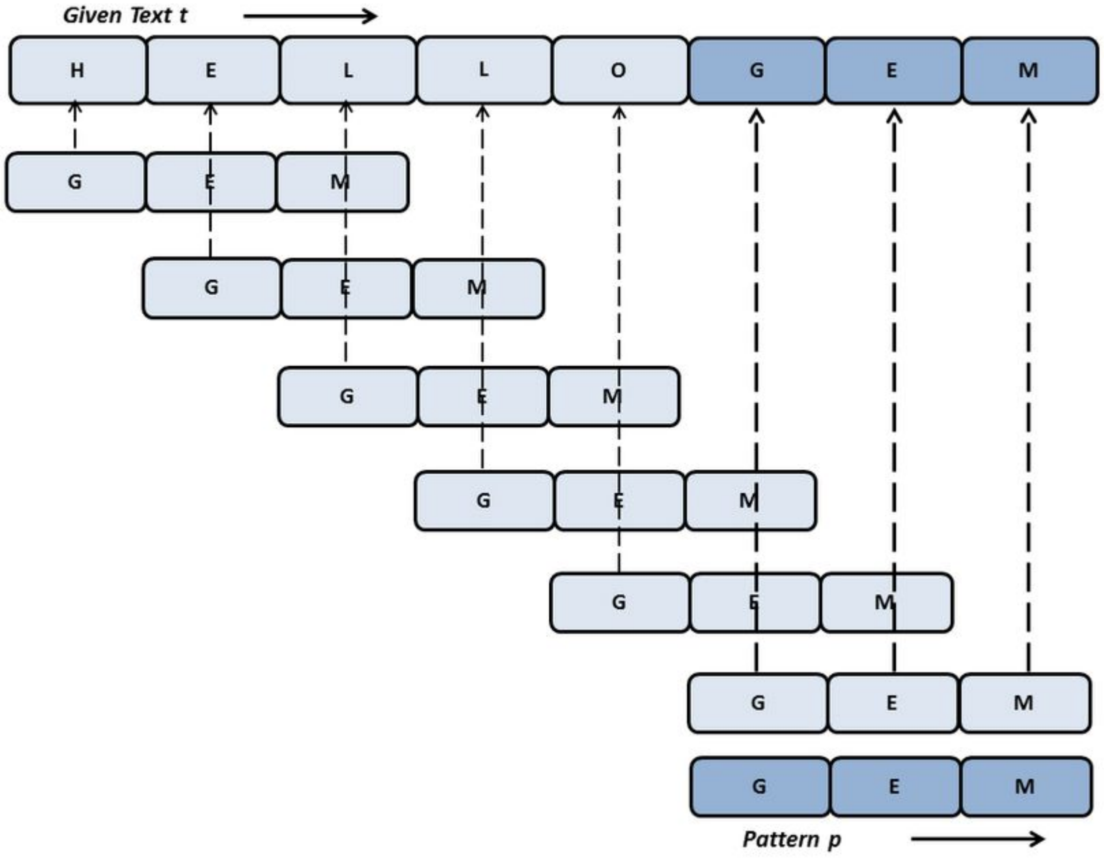


Naïve string-matching algorithm

- Simplest approach for searching a pattern within a given text.
- **Logic** : Checking for the pattern at every possible position in the text, one by one.
- **Algorithm** :

1. Start with the first character of the text.
2. Compare it with the first character of the pattern.
3. If it matches, compare the next character, and continue checking until:
   - The entire pattern matches (successful match).
   - A mismatch occurs (shift to the next position in the text).
4. Repeat the process for all possible positions in the text until the end.

Naïve string-matching algorithm - Example



Naïve string-matching algorithm - Example

Given:

- **Text:** `"ABABABCABABABCABABABC"`
- **Pattern:** `"ABABC"`

Steps:

1. Compare `"ABABC"` with the first 5 characters of the text: `"ABABA"` → mismatch at index 4.
2. Shift by 1 and compare `"ABABC"` with `"BABAB"` → mismatch.
3. Continue shifting until a full match is found.
4. Match found at index **2, 9, and 16**.

Naïve string-matching algorithm - Analysis

- **Best Case**: **O(n)** → When a mismatch occurs early in comparisons.
- **Worst Case**: **O(n m)** → When the pattern and text have many repeated characters, causing unnecessary comparisons.

  Where:

  - n = length of the Text
  - m = length of the Pattern

- **Advantages & Disadvantages**

  ✅ Simple to implement

  ✅ No preprocessing required

  ❌ Slow for large texts and patterns

  ❌ Inefficient for repeated substrings

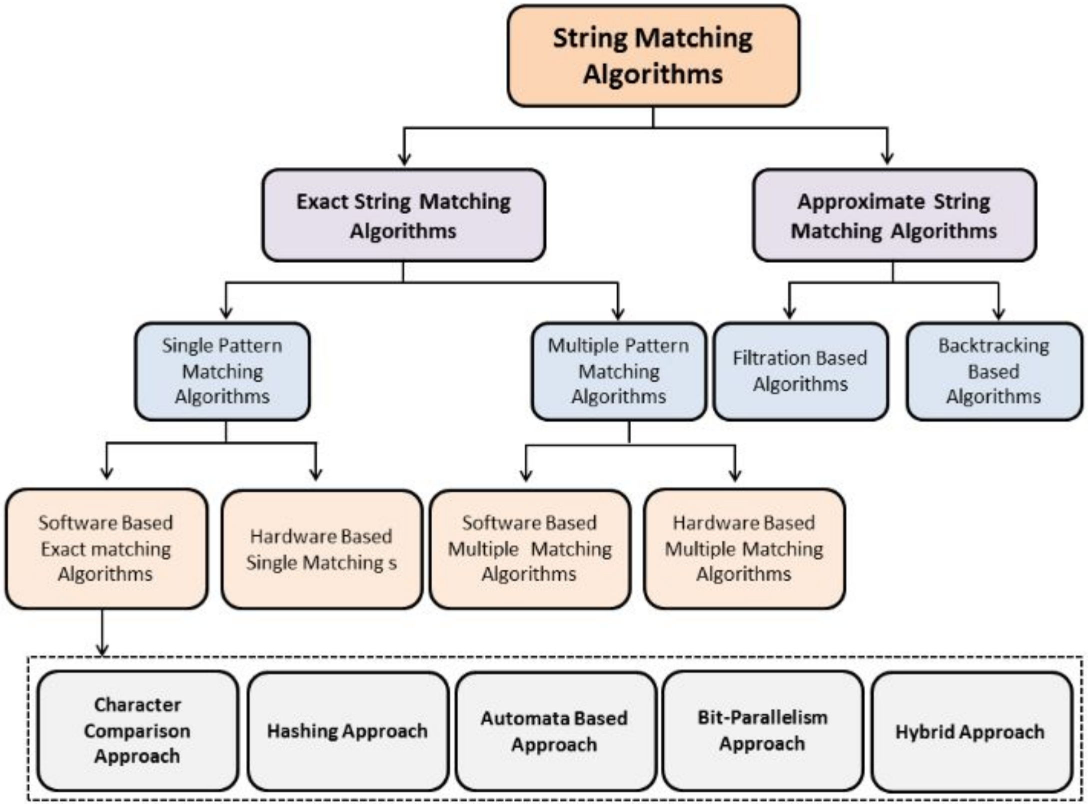
Topics to be covered

String Matching Algorithms

- Naïve string-matching algorithm
- **Rabin Karp algorithm**
- Knuth-Morris-Pratt algorithm

Extra Topics

- Finite State Automata
- Applications of String Matching



Rabin Karp algorithm

- string-searching algorithm that uses **hashing** to find a pattern in a given text efficiently.
- Calculates the hash value of the pattern and compares it with the hash values of substrings in the text.
- If a **match is found** ⇒ perform a character-by-character check to confirm the match.

Steps of the Algorithm

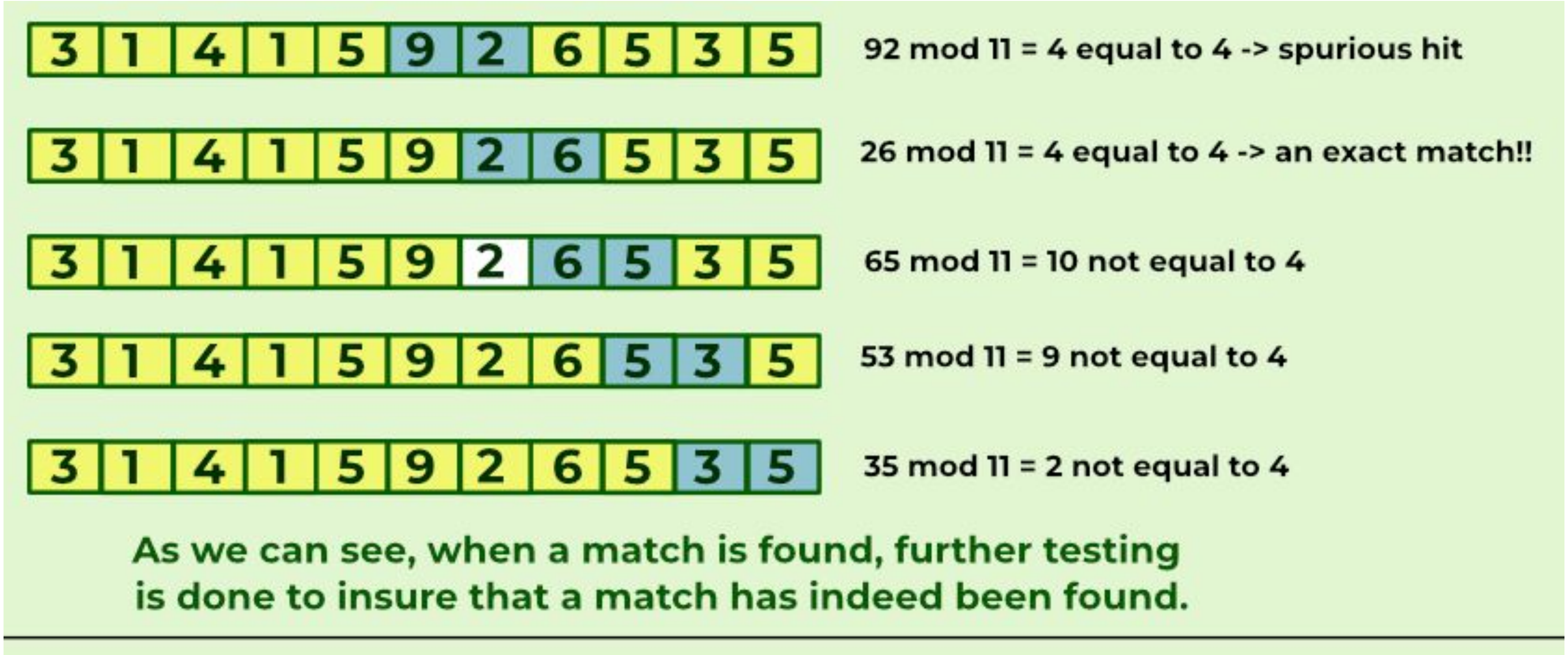
1. Compute the hash value of the pattern.
2. Compute the hash value of the first substring (of the same length as the pattern) in the text.
3. Compare the hash values:
   - If they match, perform a character-by-character comparison to confirm.
   - If they do not match, slide the window one position to the right and compute the new hash value efficiently.
4. Repeat the process until the end of the text.

Rabin Karp algorithm - Example

- Given T = 31415926535 and P = 26
- We choose q = 11
- P mod q = 26 mod 11 = 4

3	1	4	1	5	9	2	6	5	3	5	31 mod 11 = 9 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	14 mod 11 = 3 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	41 mod 11 = 8 not equal to 4
3	1	4	1	5	9	2	6	5	3	5	15 mod 11 = 4 equal to 4 -> spurious hit
3	1	4	1	5	9	2	6	5	3	5	59 mod 11 = 4 equal to 4 -> spurious hit

Rabin Karp algorithm - Example



Rabin Karp algorithm - Example

Step 1: Define Parameters

- **Pattern:** `"cdd"`
- **Text:** `"abccddaefg"`
- **Base (Prime Number):** `101`
- **Modulus (to prevent overflow):** `10^9 + 7` (optional for large numbers)

We use the **hash function**:

$$\text{hash}(s) = \sum (\text{ASCII value of character} \times \text{base}^{\text{position}})$$

Rolling hash update formula:

$$\text{new hash} = (\text{old hash} - \text{outgoing char} \times \text{base}^{m-1}) \times \text{base} + \text{incoming char}$$

where **m = 3** (length of `"cdd"`).

Rabin Karp algorithm - Example

Step 2: Compute Initial Hash of the Pattern (`cdd`)

Pattern: `"cdd"`

Text: `"abccddaefg"`

Using ASCII values:

- `'c'` = **99**
- `'d'` = **100**
- `'d'` = **100**

$$\text{hash}(\text{"}cdd\text{"}) = (99 \times 101^2) + (100 \times 101^1) + (100 \times 101^0)$$

$$= (99 \times 10201) + (100 \times 101) + (100)$$

$$= 1009899 + 10100 + 100 = \mathbf{1020099}$$

Rabin Karp algorithm - Example

Step 3: Compute Initial Hash of First Window (`abc`)

Pattern: `"cdd"`

Text: `"abccddaefg"`

Using ASCII values:

- `'a'` = **97**
- `'b'` = **98**
- `'c'` = **99**

$$\text{hash}(''abc'') = (97 \times 101^2) + (98 \times 101^1) + (99 \times 101^0)$$

$$= (97 \times 10201) + (98 \times 101) + (99)$$

$$= 989497 + 9898 + 99 = \mathbf{999494}$$

Since `999494 ≠ 1020099`, slide the window.

Rabin Karp algorithm - Example

Pattern: `"cdd"`

Text: `"abccddaefg"`

Step 4: Rolling Hash Calculation

Using:

$$\text{new hash} = (\text{old hash} - \text{outgoing char} \times 101^{m-1}) \times 101 + \text{incoming char}$$

Slide Window: `"abc" → "bcc"`

- Outgoing char: `'a'` = **97**
- Incoming char: `'c'` = **99**

$$\text{hash}("bcc") = (999494 - (97 \times 10201)) \times 101 + 99$$

$$= (999494 - 989497) \times 101 + 99$$

$$= 9997 \times 101 + 99 = \mathbf{1000696}$$

No match, continue sliding.

Rabin Karp algorithm - Example

Pattern: `"cdd"`

Text: `"abccddaefg"`

Slide Window: `"bcc" → "ccd"`

- Outgoing char: `'b'` = **98**
- Incoming char: `'d'` = **100**

$$\text{hash}(\text{"}ccd\text{"}) = (1000696 - (98 \times 10201)) \times 101 + 100$$

$$= (1000696 - 999798) \times 101 + 100$$

$$= 898 \times 101 + 100 = \mathbf{99898}$$

No match, continue sliding.

Rabin Karp algorithm - Example

Slide Window: `"ccd" → "cdd"`

- Outgoing char: `'c'` = **99**
- Incoming char: `'d'` = **100**

Pattern: `"cdd"`

Text: `"abccddaefg"`

$$\text{hash}("cdd") = (99898 - (99 \times 10201)) \times 101 + 100$$
$$= (99898 - 1009899) \times 101 + 100$$
$$= 1020099$$

Match found at index 3! Perform a **character-by-character check** to confirm.

Rabin Karp algorithm - Example

Pattern: `"cdd"`

Text: `"abccddaefg"`

Window `"dda"`

- Outgoing char: `'c'` (ASCII 99)
- Incoming char: `'a'` (ASCII 97)

$$\text{new hash} = (1020099 - (99 \times 10201)) \times 101 + 97$$

$$= (1020099 - 1009899) \times 101 + 97$$

$$= (10200) \times 101 + 97 = 1030197$$

No match.

Rabin Karp algorithm - Example

Pattern: `"cdd"`

Text: `"abccddaefg"`

Window `"dae"`

- Outgoing char: `'d'` (ASCII 100)
- Incoming char: `'e'` (ASCII 101)

$$\text{new hash} = (1030197 - (100 \times 10201)) \times 101 + 101$$
$$= (1030197 - 1020100) \times 101 + 101$$
$$= (10097) \times 101 + 101 = 1020298$$

No match.

Rabin Karp algorithm - Example

Pattern: `"cdd"`

Text: `"abccddaefg"`

Window `"aef"`

- Outgoing char: `'d'` (ASCII 100)
- Incoming char: `'f'` (ASCII 102)

$$\text{new hash} = (1020298 - (100 \times 10201)) \times 101 + 102$$
$$= (1020298 - 1020100) \times 101 + 102$$
$$= (198) \times 101 + 102 = 20000$$

No match.

Rabin Karp algorithm - Example

Pattern: `"cdd"`

Text: `"abccddaefg"`

Window `"efg"`

- Outgoing char: `'a'` (ASCII 97)
- Incoming char: `'g'` (ASCII 103)

$$\text{new hash} = (20000 - (97 \times 10201)) \times 101 + 103$$

$$= (20000 - 989497) \times 101 + 103$$

$$= (-969497) \times 101 + 103$$

$$\text{Negative hash, adjusted using modulus (not needed here).}$$

No match.

We found `"cdd"` at **index 3** in `"abccddaefg"`.

Rabin Karp algorithm - Analysis

- Best Case: **O(n+m)** (when no hash collisions occur)
- Worst Case: O(nm) (when hash collisions frequently occur, requiring full character comparison)
- Average Case: **O(n+m)**

  Here, n = is the length of the text,

  m = length of the pattern.

- **Rolling hash function**
  - ensures that computing the next substring's hash is done in constant time, making Rabin-Karp efficient when searching multiple patterns.

Applications of Rabin Karp

- Efficient for multiple pattern searches (e.g., plagiarism detection).
- Best for cases where hash collisions are minimal (e.g., large alphabets with a good hash function).
- Less effective for single pattern matching if hash collisions are frequent.

Rabin Karp algorithm - Problems

	Text	**Pattern**
1.		
2.	ABCDEF	CDE
3.	DAACABCDBA	CAB
4.	AAAABCAEAAABCBDDAAAABC	AABC

Courtesy : [Programiz]()

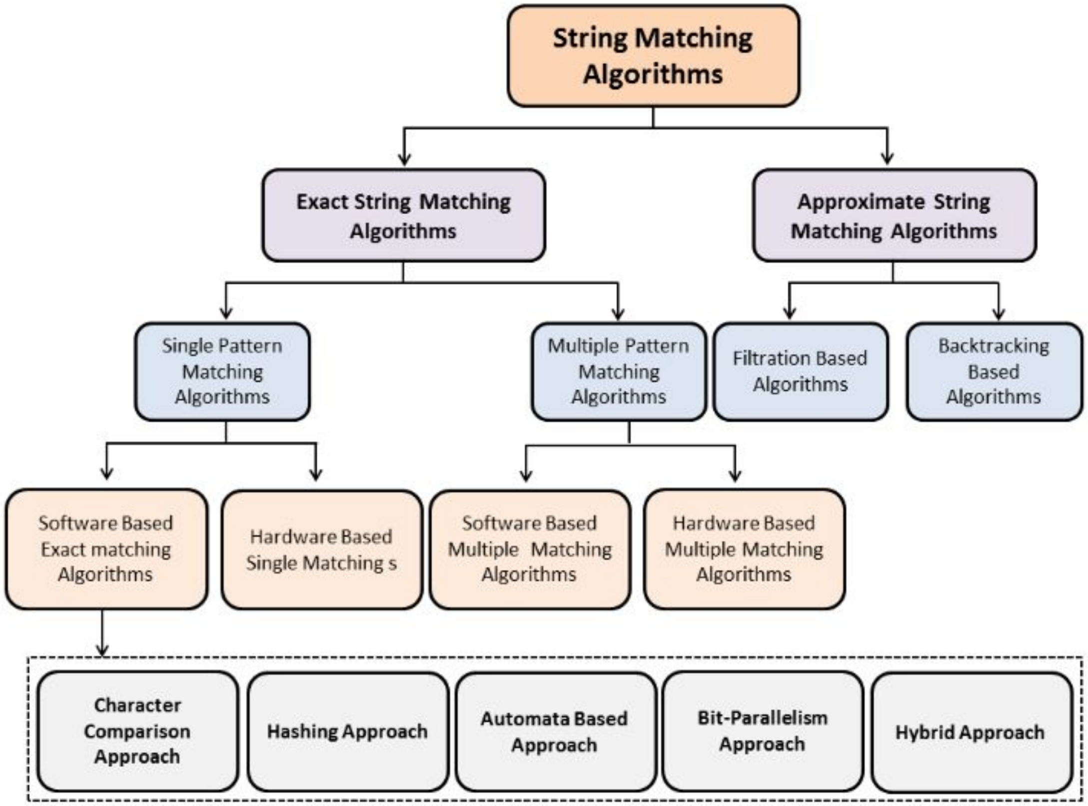
Topics to be covered

String Matching Algorithms

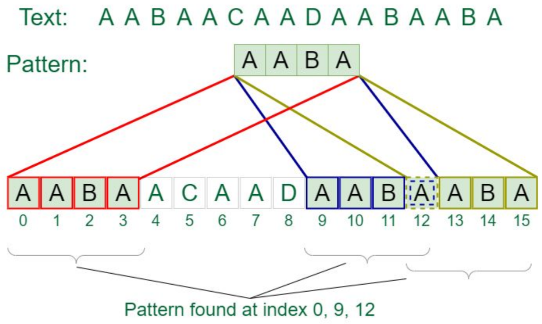
- Naïve string-matching algorithm
- Rabin Karp algorithm
- **Knuth-Morris-Pratt algorithm**

<u>Extra Topics</u>

- Finite State Automata
- Applications of String Matching



Knuth Morris Pratt algorithm

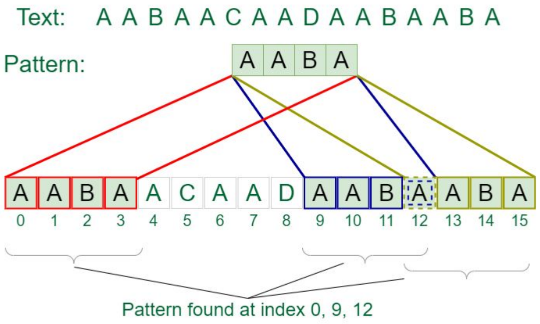


- Efficient string-searching (substring search) algorithm
- Finds occurrences of a pattern P within a text in **O(n + m)** time, where:
    - n = length of the text
    - m = length of the pattern
- KMP avoids unnecessary comparisons by using information gathered in a **preprocessing step**.

Key Advantages of KMP

- Avoids redundant comparisons using **LPS** (Longest Prefix Suffix)
- Works efficiently for large texts and patterns.
- Best suited for cases where multiple searches with the same pattern are needed.

Knuth Morris Pratt algorithm



KMP consists of two main steps:

1. **Preprocessing the pattern (Computing the LPS array)**

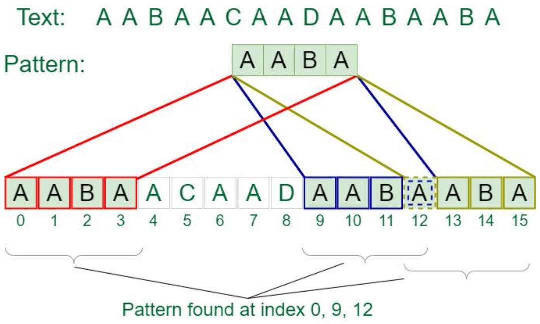
- Helps us skip unnecessary comparisons in the search phase.

- LPS[i] stores, the length of the longest proper prefix of the pattern = Suffix of the pattern up to index i.
- If a mismatch occurs, the LPS array tells us how much the pattern can be shifted while preserving already matched characters.

2. Searching the pattern in the text using LPS

- Compare characters of P with T one by one.
- If a mismatch occurs ⇒ use the LPS array to determine the next position.
- This ensures that we do not compare already matched characters again.

Knuth Morris Pratt algorithm



LPS is the **Longest Proper Prefix** which is also a **Suffix**.

- A proper prefix is a prefix that doesn't include **whole** string.
  - Eg: Given String "abc"
  - prefixes : "", "a", "ab" and "abc"
  - proper prefixes : "", "a" and "ab" only.
  - Suffixes : "", "c", "bc", and "abc".
- Each value, **lps[i]** = length of longest proper prefix of **pat[0..i]** = a suffix of **pat[0..i]**.

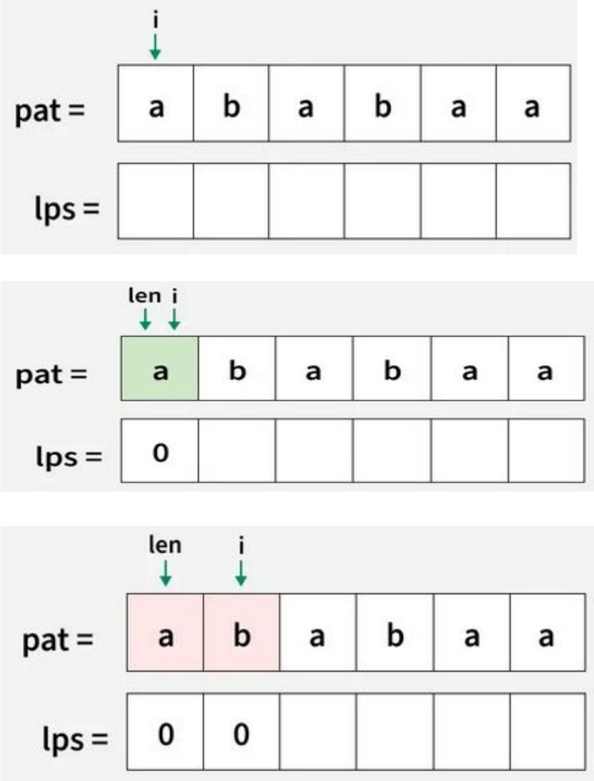
Knuth Morris Pratt algorithm - Algorithm for Constructing the LPS Array

Given a pattern `pat[0...m-1]`, the LPS array is constructed as follows:

- Initialize variables:
  - `lps[0] = 0` (First character has no proper prefix which is also a suffix)
  - `length = 0` (Tracks length of the previous longest prefix suffix)
  - `i = 1` (Start from second character)
- Iterate through the pattern while `i < m`:
  - If `pat[i] == pat[length]`: // **Case - 1 : Match found**
    - Increment `length` (`length++`)
    - Assign `lps[i] = length` // Extend the LPS at the previous index
    - Move to the next index (`i++`)
  - Else (`pat[i] != pat[length]`):
    - If `length = 0` // **Case - 2** : **`There were no matching characters earlier`**
      - Assign `lps[i] = 0` // Also, the current characters are also not matching ⇒ lps[i] = 0
      - Move to the next index (`i++`)
    - Else // **Case - 3** : There may be a smaller prefix that matches the suffix ending at i.
      - Update `length = lps[length - 1]` // Go back to previous prefix suffix

Knuth Morris Pratt algorithm - Example for computing LPS Array

Initialize : len = 0
lps[].len = pat[].len
i = 0



lps[0] = 0 //No proper prefix for single char
i++

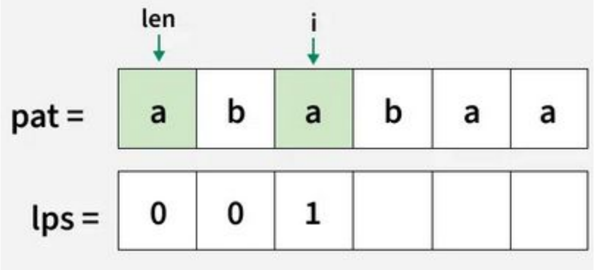
pat[len] != lps[i] && len == 0 // **Case - 2**
lps[i] = 0
i++

len	i
0	0
0	1
0	2

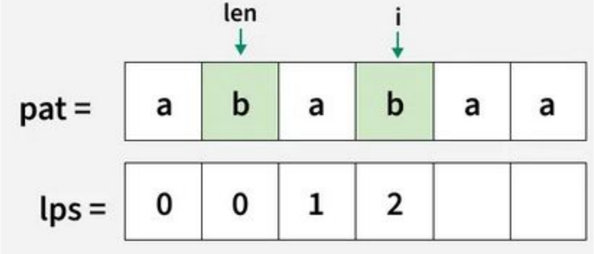
Courtesy : Geeks For Geeks

Knuth Morris Pratt algorithm - <u>Example for computing LPS Array</u>

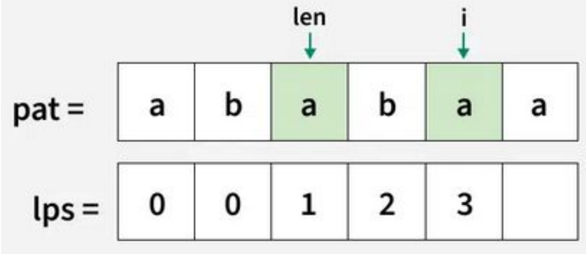
```
pat[len] == lps[i]          // Case - 1
    len++
    lps[i] = len
    i++
```




```
pat[len] == lps[i]          // Case - 1
    len++
    lps[i] = len
    i++
```




```
pat[len] == lps[i]          // Case - 1
    len++
    lps[i] = len
    i++
```



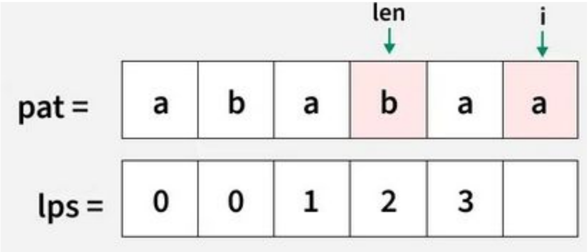
len	i
0	2
1	3
2	4
3	5

Courtesy : Geeks For Geeks

Knuth Morris Pratt algorithm - <u>Example for computing LPS Array</u>

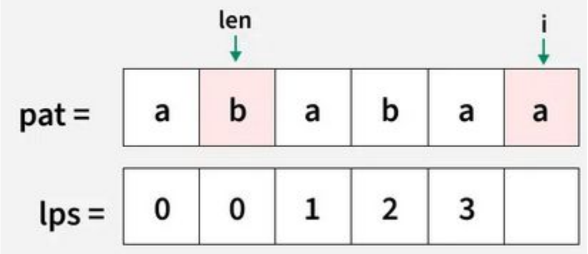
pat[len] != lps[i] && len !=0 // **Case - 3**
len = lps[len-1]

// len = lps[2] = 1

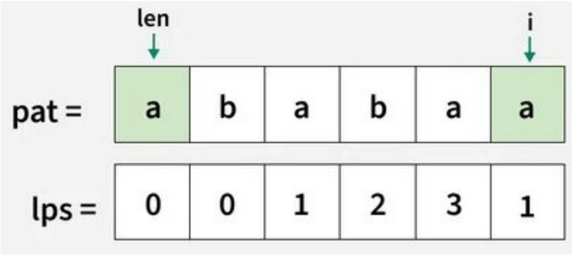


pat[len] != lps[i] && len !=0 // **Case - 3**
len = lps[len-1]

// len = lps[0] = 0



pat[len] == lps[i] // **Case - 1**
len++
lps[i] = len
i++



len	i
3	5
1	5
0	5
1	6

Courtesy : Geeks For Geeks

Knuth Morris Pratt algorithm - Practice

Pattern	LPS []
AAAA	[0, 1, 2, 3]
ABCDE	[0, 0, 0, 0, 0]
AABAACAABAA	[0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]
AAACAAAAAC	[0, 1, 2, 0, 1, 2, 3, 3, 3, 4]
AAABAAA	[0, 1, 2, 0, 1, 2, 3]
ABABCABAB	[0, 0, 1, 2, 0, 1, 2, 3, 2]

Knuth Morris Pratt algorithm - Practice

Text (T): "ABABDABACDABABCABAB"

Pattern (P): "ABABCABAB"

Step 1: Compute the LPS Array

Index i	Pattern Prefix	LPS Value
0	A	0
1	AB	0
2	ABA	1
3	ABAB	2
4	ABABC	0
5	ABABCA	1
6	ABABCAB	2
7	ABABCABA	3
8	ABABCABAB	4

So, **LPS = [0, 0, 1, 2, 0, 1, 2, 3, 4]**

Knuth Morris Pratt algorithm

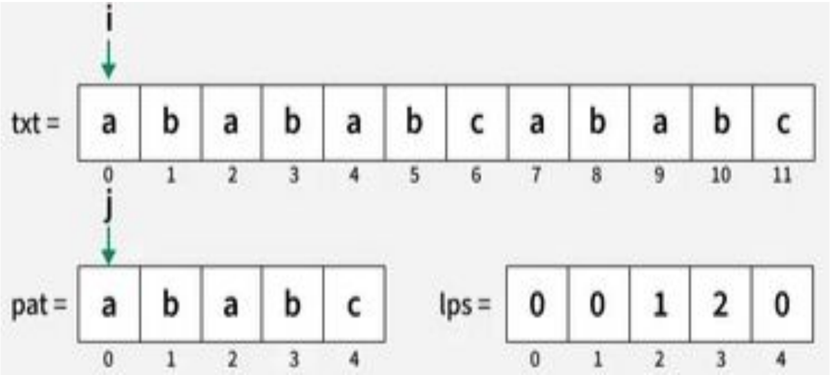
Step 2: Pattern Searching in Text using LPS

We align **P** with **T** and start matching characters:

- If characters match, move both `i` (for text) and `j` (for pattern) forward.
- If a mismatch occurs:
  - If `j = 0`, move `i` forward.
  - Otherwise, use `LPS[j-1]` to shift `j` and continue.

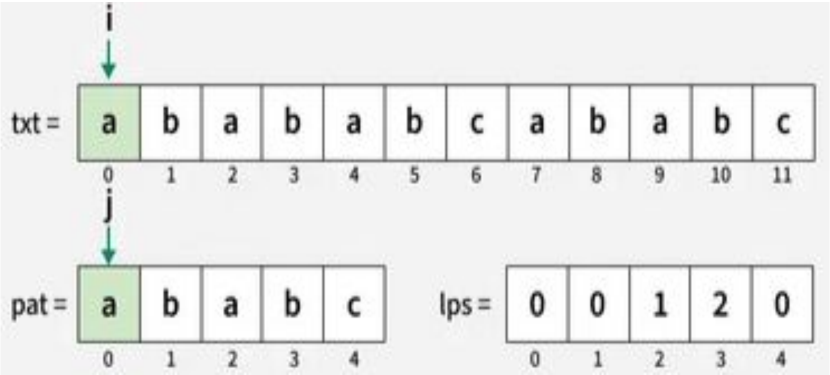
After applying this, the pattern `"ABABCABAB"` is found at **index 10** in the text.

Knuth Morris Pratt algorithm



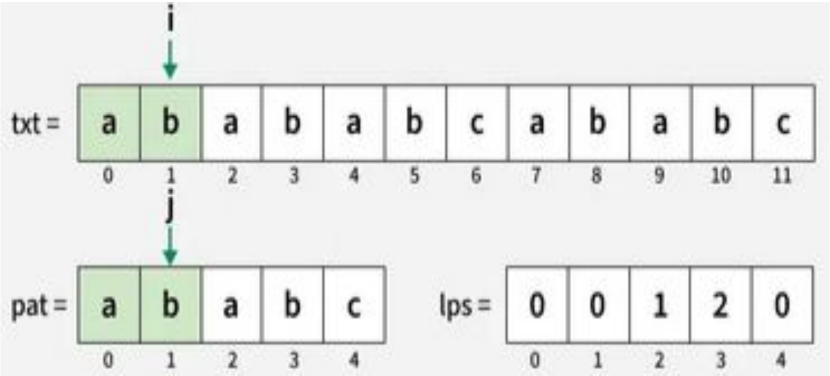
i	j
0	0
1	1

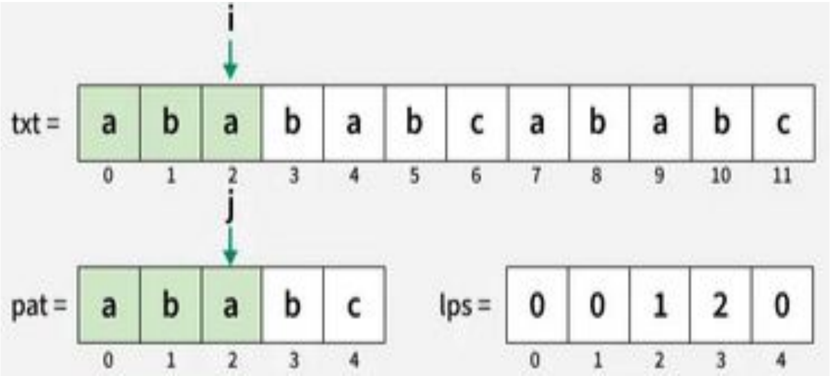
txt[i] == pat[j]
i++
j++



Courtesy : [Geeks For Geeks]

Knuth Morris Pratt algorithm



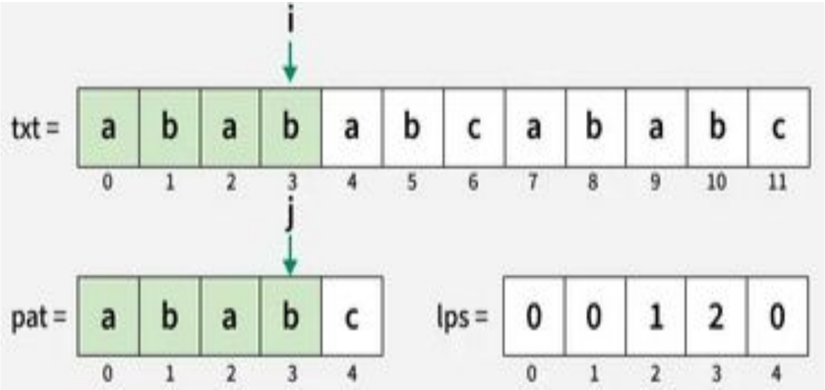


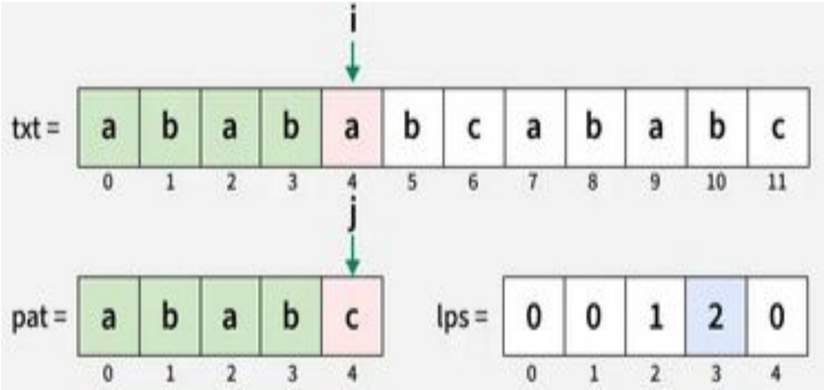
i	j
1	1
2	2
3	3

txt[i] == pat[j]
i++
j++

Courtesy : Geeks For Geeks

Knuth Morris Pratt algorithm

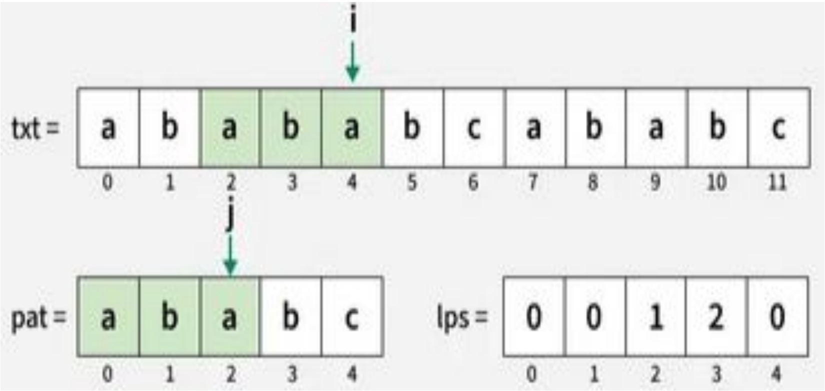


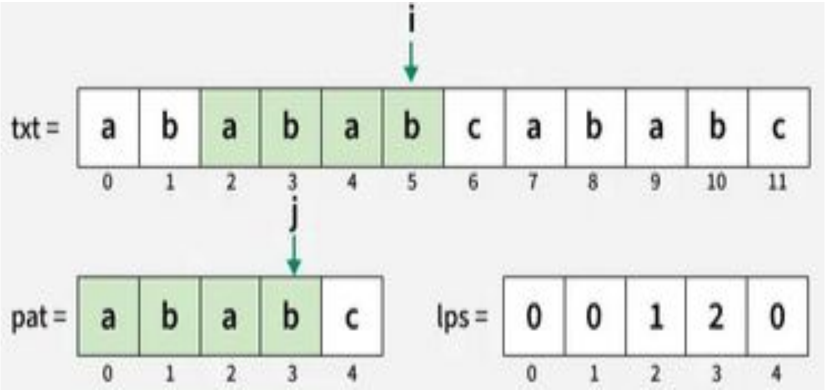


i	j
3	3
4	4
4	2

txt[i] != pat[j]
j = lps[j-1]
  = lps[3]
j = 2

Knuth Morris Pratt algorithm

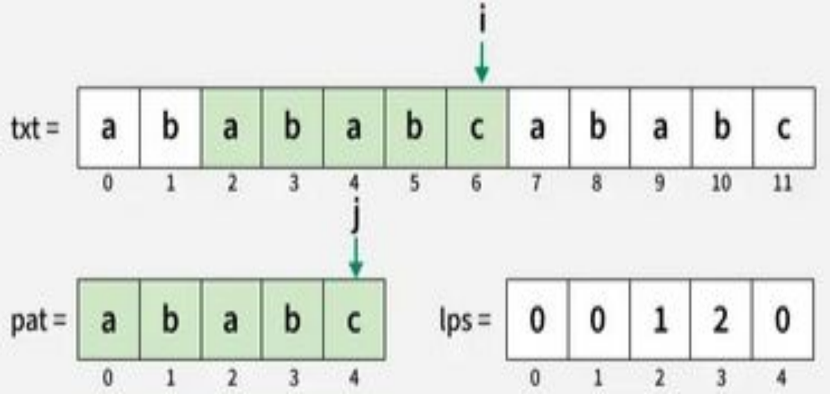


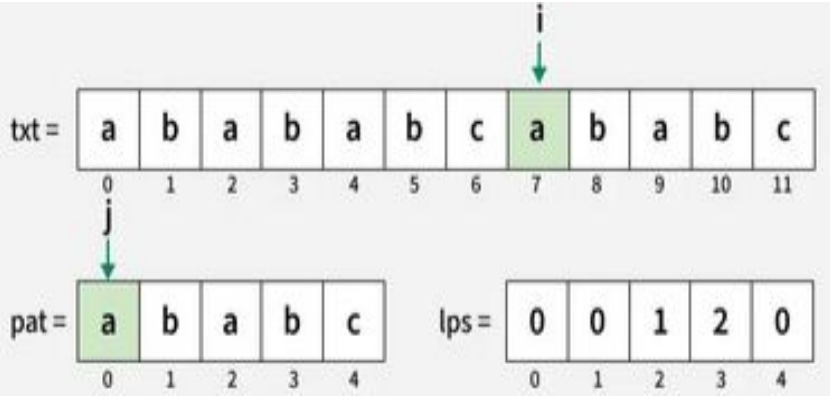


i	**j**
4	2
5	3
6	4

txt[i] == pat[j]
i++
j++

Knuth Morris Pratt algorithm





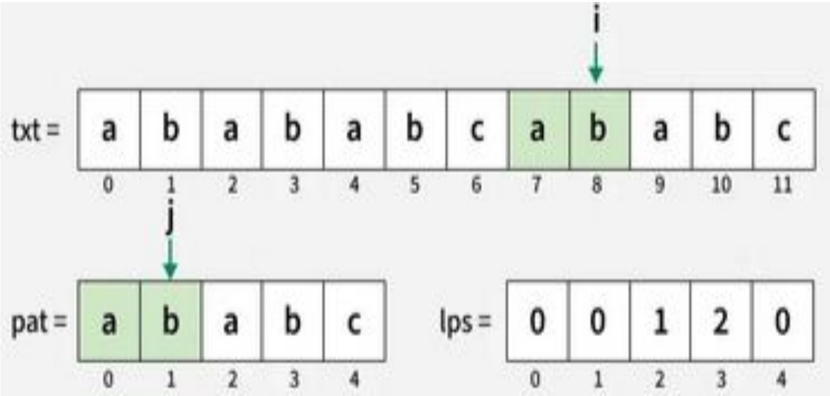
i	j
6	4
7	5
7	0

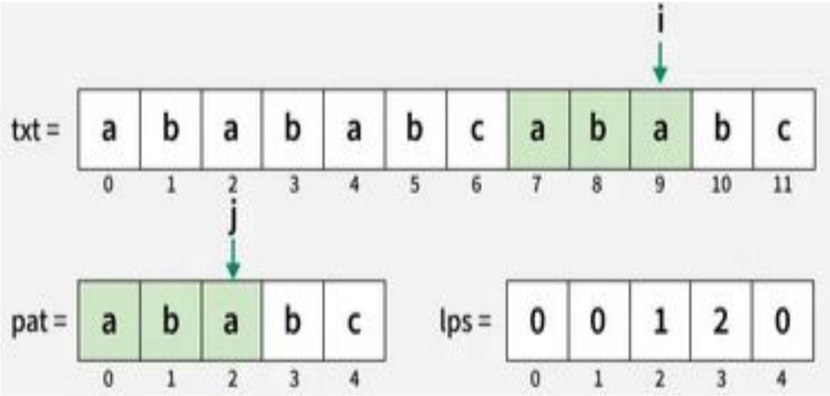
txt[i] == pat[j]
i++
j++

The Complete pattern is matched.

1. Update result[] with starting index
   a. Starting Index (i-j) = (7-5) = 2
   b. Result =[2]
2. Update j = lps[j-1] = lps[5-1] = lps[4] = 0

Knuth Morris Pratt algorithm

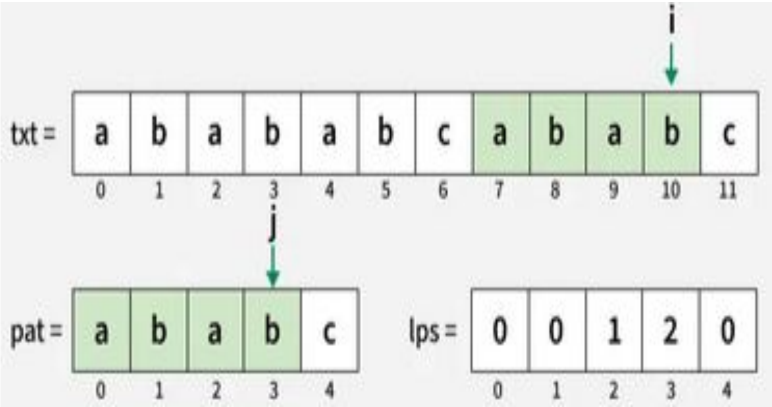


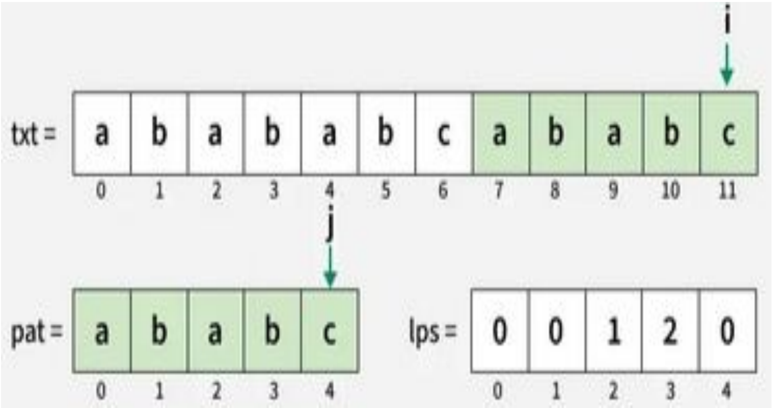


i	j
7	0
8	1
9	2
10	3

txt[i] == pat[j]
i++
j++

Knuth Morris Pratt algorithm





i	j
10	3
11	4
12	5
12	0

txt[i] == pat[j]
i++
j++

The Complete pattern is matched.

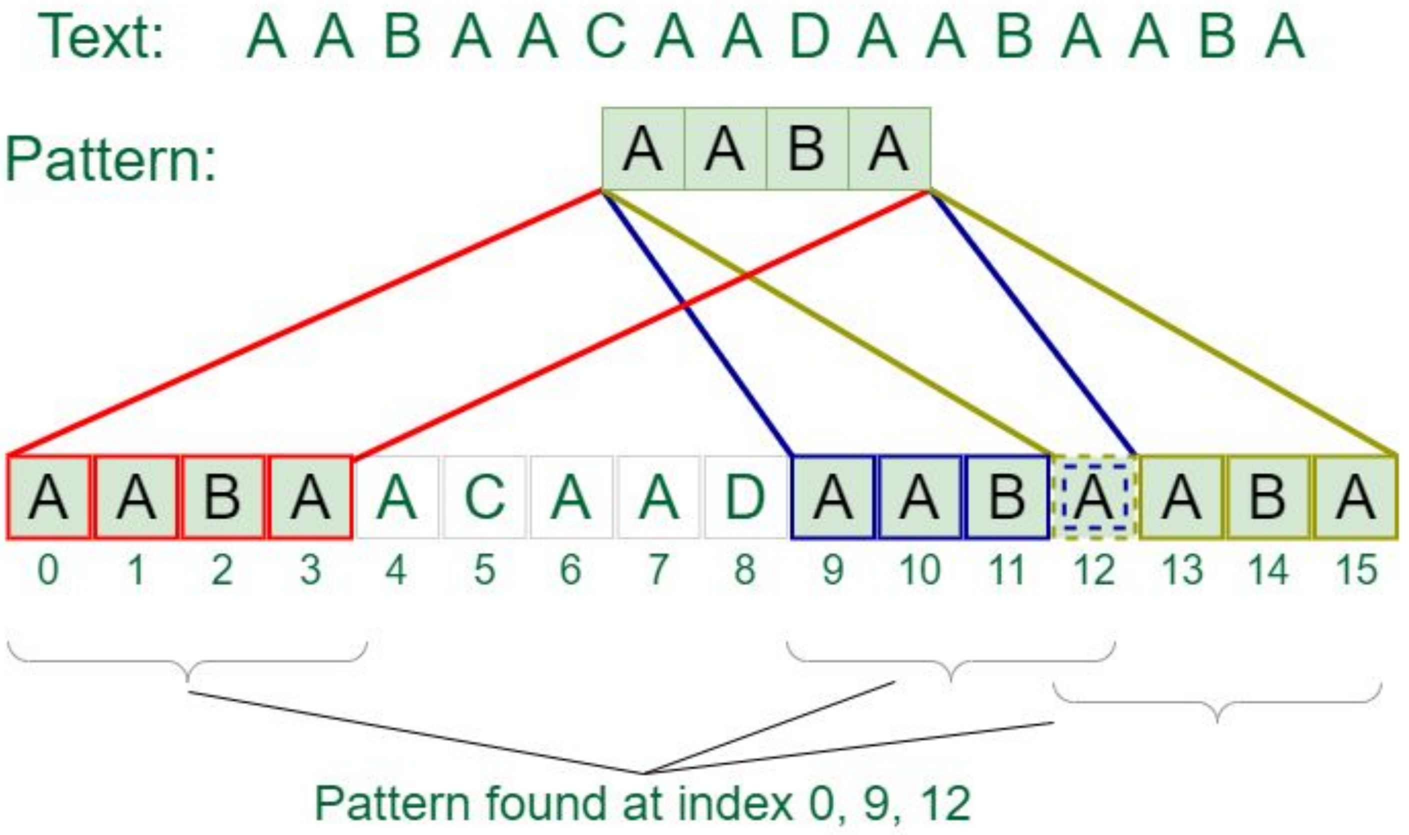
1. Update result[] with starting index
   a. Starting Index (i-j) = (12-5) = 7
   b. Result =[2,7]
2. Update j = lps[j-1] = lps[5-1] = lps[4] = 0

Knuth Morris Pratt algorithm

1. **Preprocessing LPS Array:**
   - Each character of the pattern is processed once.
   - **Time Complexity:** $O(m)$
2. **Searching Phase:**
   - Each character of the text is processed at most once.
   - **Time Complexity:** $O(n)$

Thus, the total time complexity of KMP is $O(n + m)$.

Knuth Morris Pratt algorithm - Example



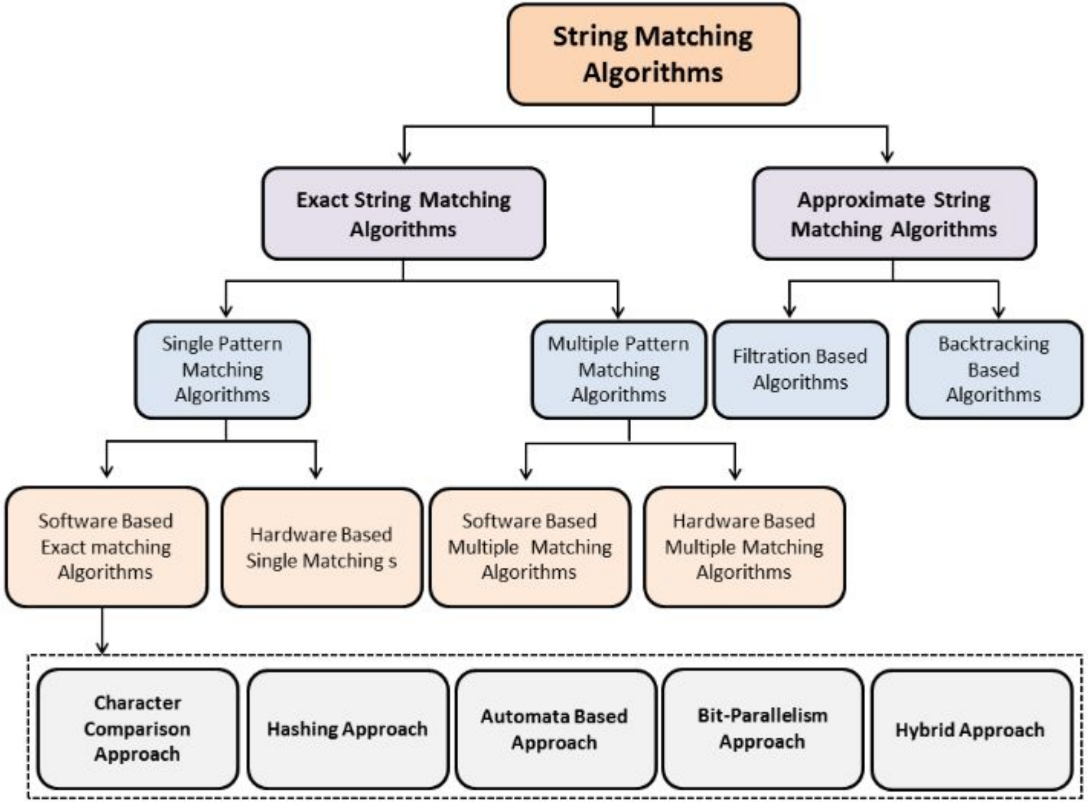
Topics to be covered

String Matching Algorithms

- Naïve string-matching algorithm
- Rabin Karp algorithm
- Knuth-Morris-Pratt algorithm

<u>Extra Topics</u>

- **Finite State Automata**
- Applications of String Matching



Finite State Automata

- efficient **pattern matching**
- Involves two phases:
  1. **Preprocessing Phase**: Construct a Finite State Automaton (FSA) for the given pattern.
  2. **Searching Phase**: Use the FSA to scan the text in a single pass.
- pattern is recognized with **O(m) preprocessing time** and **O(n) search time**.
- Components of finite automata include:
  - <u>States</u>: represent different configurations or situations of the system being modeled
  - <u>Transitions</u> define how the automaton moves from one state to another in response to input symbols.
  - <u>Start State</u>: state from which the automaton begins its operation.
  - <u>Accepting (or Final) States</u>: These states indicate successful or valid outcomes.

Finite State Automata - Definition

- Defined by tuple $M = \{\Sigma, Q, q_0, F, \delta\}$,

  where

  Q = Set of states in finite automata

  Σ = Set of input symbols

  q_0 = Initial state

  F = Set of final states

  δ = Transition function defined as $\delta : Q \times \Sigma \rightarrow Q$

- Working of Finite State Automata
  - Starts with input state q_0
  - Reads the input string character by character and changes the state according to transition function.
  - It accepts the string ⇒ if a finite automaton ends up in one of the final / accepting states.
  - It rejects the string ⇒ if a finite automaton does not end up in final state.

Finite State Automata - Example

- Example :

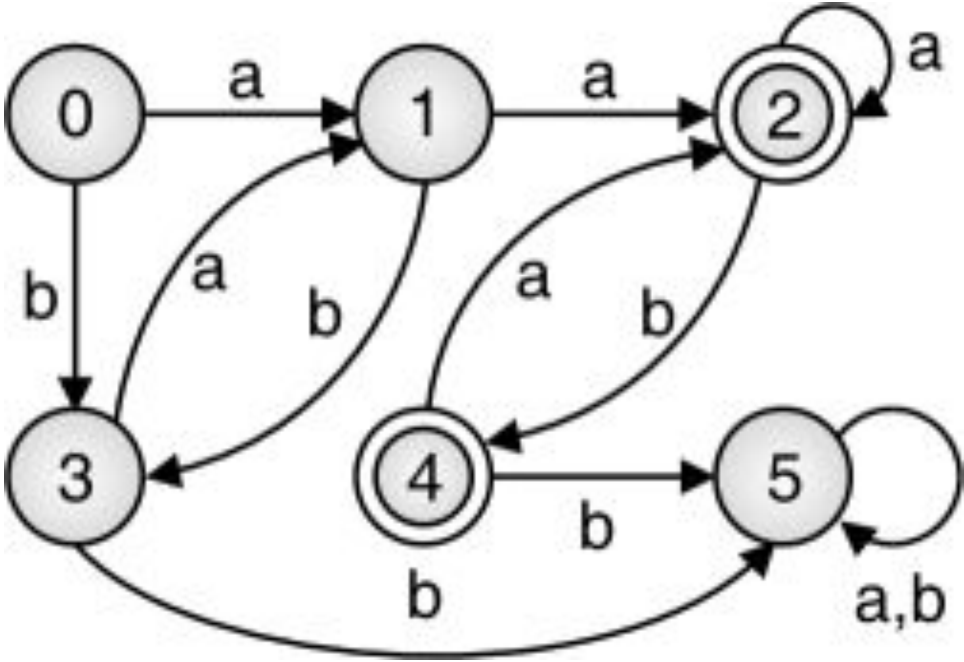
  Q = {0, 1, 2, 3, 4, 5},

  q_0 = 0,

  F = {2, 3},

  Σ = {a, b},

Transition function, δ		
Input State	**a**	**b**
0	1	3
1	2	3
2	2	4
3	1	5
4	2	5
5	5	5



Finite State Automata - Practice

Construct and show simulation of finite automata for matching the pattern **P = abb** for **text T = ababbaababba**

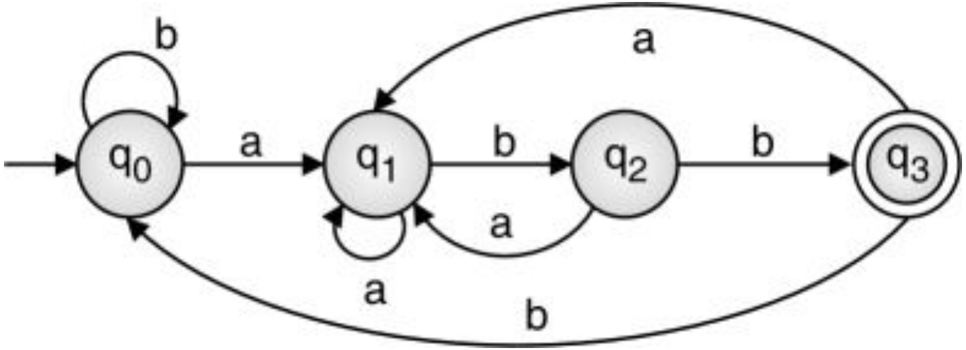
$Q = \{q_0, q_1, q_2, q_3\}$,

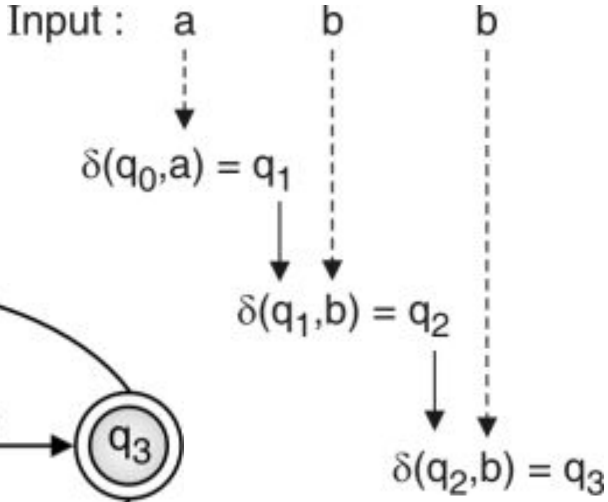
$q_0 = q_0$,

$F = \{q_3\}$,

$\Sigma = \{a, b\}$,

Transition function, δ		
Input State	**a**	**b**
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_1	q_0

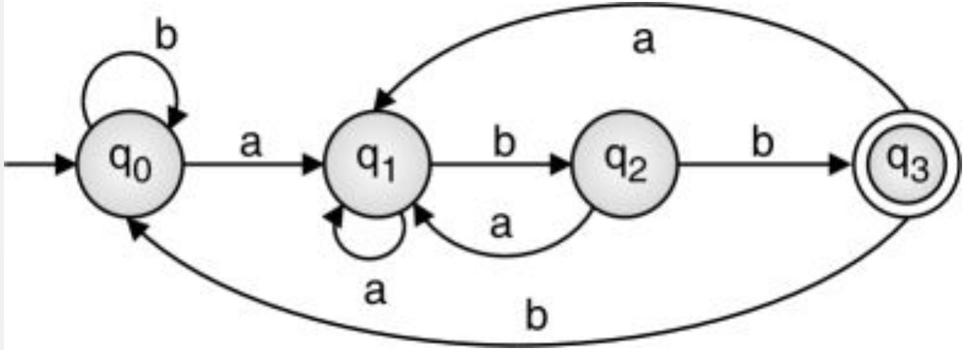




Finite State Automata - Practice

Construct and show simulation of finite automata for matching the pattern **P = abb** for **text T = ababbaababba**

a	b	a	b	b	a	a	b	a	b	b	a	Comment
a	b	b										
	a	b	b									
		a	b	b								Match found
			a	b	b							
				a	b	b						
					a	b	b					
						a	b	b				
							a	b	b			
								a	b	b		Match found
									a	b	b	



Courtesy : Codecrucks.com

Finite State Automata - Complexity Analysis

Phase	Time Complexity
Preprocessing (Building FSA)	$O(m \cdot \Sigma)$
Searching (Scanning text)	$O(n)$

Where:

- m is the length of the pattern.
- n is the length of the text.
- Σ is the alphabet size.

Since searching is **O(n)** and preprocessing is **O(m)**, this makes the FSA-based string matching very efficient for repeated searches on large text databases.

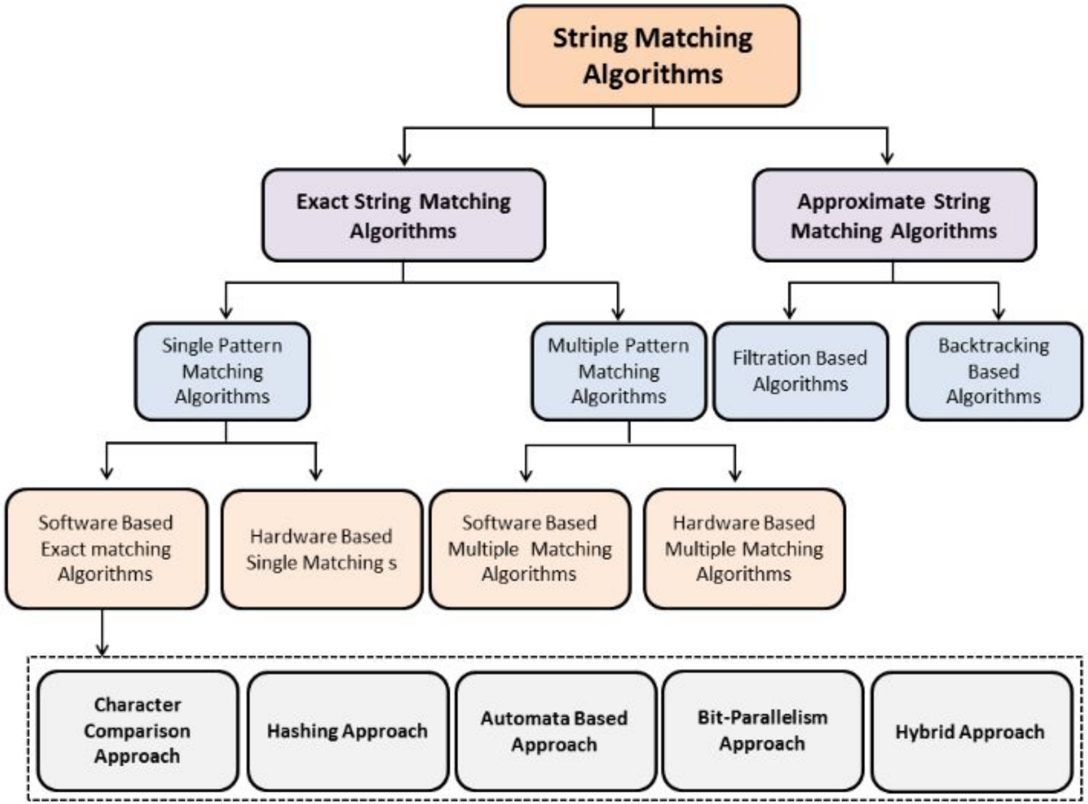
Topics to be covered

String Matching Algorithms

- Naïve string-matching algorithm
- Rabin Karp algorithm
- Knuth-Morris-Pratt algorithm

<u>Extra Topics</u>

- Finite State Automata
- **Applications of String Matchin**



Applications of String Matching Algorithms

1. Text Processing & Search Engines

- **Keyword Search in Documents** (e.g., searching for a word in Microsoft Word, Google Docs).
- **Plagiarism Detection** – Algorithms compare documents to detect copied text.
- **Auto-Suggestions in Search Engines** – Google, Bing, and DuckDuckGo (suggest relevant queries).
- **Spell Checkers & Auto-Correct** – Checks words against a dictionary using approximate string matching.
- **Ex:**
  - The **KMP Algorithm** is used in grep, a command-line tool for searching text.

2. Bioinformatics & DNA Sequence Analysis

- **DNA & Protein Sequence Matching** – identifying genetic patterns, mutations, or evolutionary relationships.
- **Gene Prediction** – Finds specific gene sequences in large DNA databases.
- **Drug Discovery** – Identifies protein structures for new medicines.
- **Ex:**
  - **Aho-Corasick algorithm** is used to match multiple DNA sequences at once.

Applications of String Matching Algorithms

3. Cybersecurity & Intrusion Detection

- **Malware & Virus Detection** – Antivirus software scans files for known virus signatures.
- **Intrusion Detection Systems (IDS)** – Identifies suspicious network patterns to detect cyber attacks.
- **Spam Filtering** – Email services use string matching to detect spam messages.
- **Ex:**
  - **Boyer-Moore Algorithm** is used in intrusion detection for efficient pattern searching.

4. Natural Language Processing (NLP) & Chatbots

- **Named Entity Recognition (NER)** – Identifies entities like names, locations, and organizations in text.
- **Sentiment Analysis** – Detects keywords related to emotions in user reviews.
- **Chatbots & AI Assistants** – Recognizes predefined commands or responses in chat applications.
- **Ex:**
  - **Trie-based Aho-Corasick Algorithm** is used in fast text classification.

Applications of String Matching Algorithms

5. Data Compression & Encoding

- **File Compression (e.g., ZIP, GZIP)** – Lempel-Ziv (LZ) algorithms use pattern matching to reduce file size.
- **Data Deduplication** – Identifies and removes duplicate data in storage systems.
- **Text Prediction & Autocomplete** – Used in mobile keyboards like SwiftKey and Gboard.
- **Ex:**
    - **Lempel-Ziv-Welch (LZW) Compression** is widely used in GIF image compression.

6. E-Commerce & Recommendation Systems

- **Product Search & Filters** – Online stores like Amazon use string matching to recommend products.
- **Price Comparison Websites** – Compare product names and specifications across different sellers.
- **Review Analysis** – Identifies patterns in user reviews for sentiment analysis.
- **Ex:**
    - **Fuzzy Matching Algorithms** help in typo correction in e-commerce search.

Applications of String Matching Algorithms

7. Digital Forensics & Legal Applications

- **Identifying Fake Documents** – Detects manipulated or plagiarized documents.
- **Forensic Text Analysis** – Examines emails, messages, and legal documents for fraud detection.
- **Log File Analysis** – Detects unusual patterns in system logs.
- **Ex:**
  - **KMP Algorithm** is used to efficiently search through large forensic data logs.

8. Database & Big Data Search

- **Indexing & Query Optimization** – String matching helps optimize database searches.
- **Pattern-Based Data Extraction** – Used in log analysis, error detection, and report generation.
- **Big Data Processing (e.g., Hadoop, Elasticsearch)** – Efficiently searches large datasets.
- **Ex:**
  - **Suffix Trees & Suffix Arrays** are used in high-speed database indexing.

Applications of String Matching Algorithms

9. Speech & Handwriting Recognition

- **Speech-to-Text Systems** – Identifies phonetic patterns in speech and converts them to text.
- **Handwriting Recognition** – Used in OCR (Optical Character Recognition) to extract text from images.
- **Automatic Transcription** – Converts spoken words into written text.
- **Ex:**
  - **Levenshtein Distance Algorithm** is used to correct OCR errors.

10. Gaming & Computer Vision

- **Cheat Detection in Online Games** – Compares player actions with known cheating patterns.
- **AI-based Character Recognition** – Detects text in game environments.
- **Captcha Verification** – Ensures human interactions in online forms.
- **Ex:**
  - **Regular Expressions & Trie Structures** are used to match game chat logs.

Compare String Matching Algorithms

Algorithm	Preprocessing Time	Searching Time	Space Complexity	Best Case	Worst Case
Naïve Approach	O(1)	O(nm)	O(1)	O(n)	O(nm)
Knuth-Morris-Pratt (KMP)	O(m)	O(n)	O(m)	O(n)	O(n)
Rabin-Karp	O(m)	O(n) (single match) O(nm) (multiple matches)	O(1)	O(n)	O(nm) (worst case hash collision)
Boyer-Moore	O(m + Σ)	O(n/m)	O(Σ)	O(n/m)	O(nm) (worst case)
Finite State Automaton (FSA)	O(m * Σ)	O(n)	O(m * Σ)	O(n)	O(n)