Q1. The Naive algorithm:

a. Illustrate the Naïve algorithm given the following:

Text = “abcabcabcabc”

Pattern = “abcabc”

b. How many character comparisons are made?

c. How can the worst-case number of comparisons be O(nm)? Give an example where this occurs.

Length of text, n = 12

Length of pattern, m = 6

s = n – m + 1

= 12-6+1

= 7

We check starting at each position from 0 to 6 since s starts from 0.

0	1	2	3	4	5	6	7	8	9	10	11
a	b	c	a	b	c	a	b	c	a	b	c

s = 0

a	b	c	a	b	c	a	b	c	a	b	c
a	b	c	a	b	c						

6 comparisons

s = 1

a	b	c	a	b	c	a	b	c	a	b	c
	a	b	c	a	b	c					

1 comparison

s = 2

a	b	c	a	b	c	a	b	c	a	b	c
		a	b	c	a	b	c				

1 comparison

s = 3

a	b	c	a	b	c	a	b	c	a	b	c
			a	b	c	a	b	c			

6 comparisons

s = 4

a	b	c	a	b	c	a	b	c	a	b	c
				a	b	c	a	b	c		

1 comparison

s = 5

a	b	c	a	b	c	a	b	c	a	b	c
					a	b	c	a	b	c	

1 comparison

s = 6

a	b	c	a	b	c	a	b	c	a	b	c
						a	b	c	a	b	c

6 comparisons

Total Comparisons:

- Matches at s = 0, 3, 6 → 3 × 6 = 18
- Mismatches at s = 1, 2, 4, 5 → 4 × 1 = 4

Total = 18 + 4 = 22 comparisons

Worst Case:

- All characters in both string and pattern are the same.
- All characters in both string and pattern are the same except for the last character.

Text = aaaaa
Pattern = aab

s = 0

a	a	a	a	a
a	a	b		

3 comparison

s = 1

a	a	a	a	a
	a	a	b	

3 comparison

s = 2

a	a	a	a	a
		a	a	b

3 comparison

Length of text, n = 5
Length of pattern, m = 3

The naive algorithm would perform (m) comparisons for each of the (n - m + 1) possible shifts

= O (m*(n-m+1))
= O (mn)

Q2.The Rabin-Karp algorithm:

a. Illustrate the Rabin-Karp algorithm given the following:

Text: "ABCCDEABCDEF"

Pattern: "ABC”

a)

A	B	C	C	D	E	A	B	C	D	E	F	← Text, T[11]

A	B	C	← Pattern, P[2]

Using the hash function:

$$\boxed{H = c_1 \times b^{n-1} + c_2 \times b^{n-2} + \ldots + c_n \times b^0 \mod p}$$

c = value set for the characters (ASCII or A=1, B=1, ...)
b = base prime number, 31
n = length of string
p = large prime number, 101

A	=	1
B	=	2
C	=	3
D	=	4
E	=	5
F	=	6

Calculating hash for the pattern

A	B	C
1	2	3

$$\text{hash} = (1 \cdot 31^2) + (2 \cdot 31^1) + (3 \cdot 31^0) \mod 101$$

$$= 961 + 62 + 3 \mod 101$$

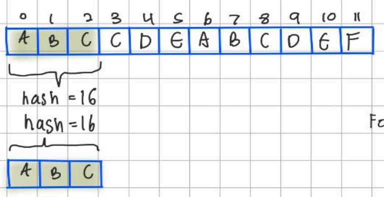
$$= 1026 \mod 101$$

$$\underset{(\text{pattern})}{\text{hash}} = 16$$

Calculate hash for the first 3 characters of text

$$\underset{(\text{text})}{H} = (1 \cdot 31^2) + (2 \cdot 31^1) + (3 \cdot 31^0) \mod 101$$

$$= 16$$



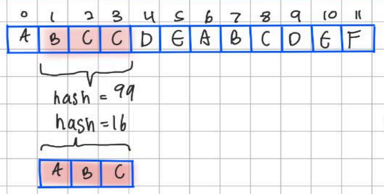
Found match at; 0

$$\boxed{\underset{(\text{text})}{H_{new}} = [b \times (H_{old} - C_{old} \times b^{n-1}) + C_{new}] \mod p}$$

$$= 31 \times (16 - 1 \times 31^2) + 3) \mod 101$$

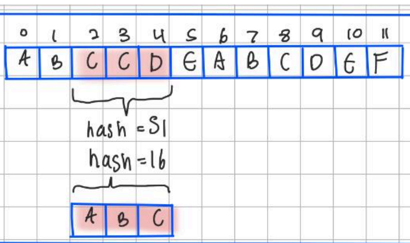
$$= -29292 \mod 101$$

$$H_{new} = 99$$



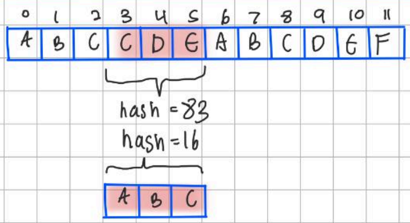
Found match at; 0

H_{new} (text) $= 31 \times (99 - 2 \times 31^2) + 4 \bmod 101$
$= -56509 \bmod 101$
$= 51$



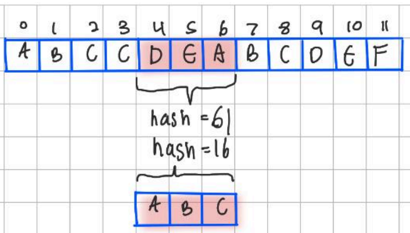
Found match at: 0

H_{new} (text) $= 31 \times (51 - 3 \times 31^2) + 5 \bmod 101$
$= -87787 \bmod 101$
$= 83$



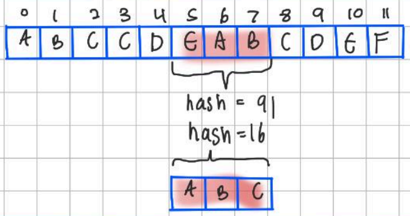
Found match at: 0

$H_{new} = [31 \times (83 - 3 \times 31^2) + 1] \bmod 101$
$= -86799 \bmod 101$
$= 61$



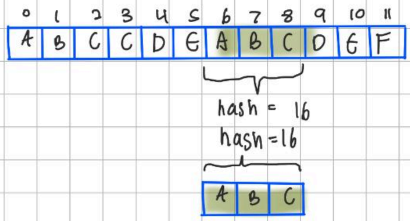
Found match at: 0

H_{new} (text) $= 31 \times (61 - 4 \times 31^2) + 2 \bmod 101$
$= -117271 \bmod 101$
$= 91$



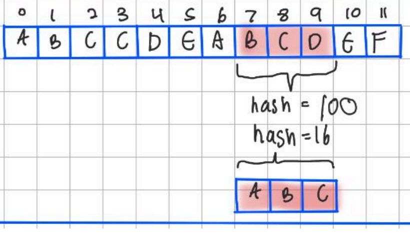
Found match at: 0

H_{new} (text) $= 31 \times (91 - 5 \times 31^2) + 3 \bmod 101$
$= -146131 \bmod 101$
$= 16$

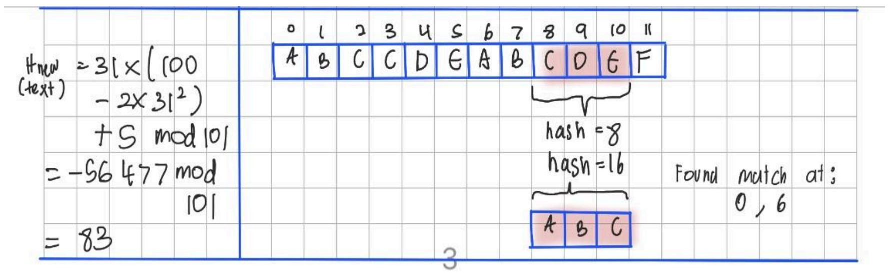


Found match at: 0, 6

H_{new} (text) $= 31 \times (16 - 1 \times 31^2) + 4 \bmod 101$
$= -29291 \bmod 101$
$= 100$



Found match at: 0, 6



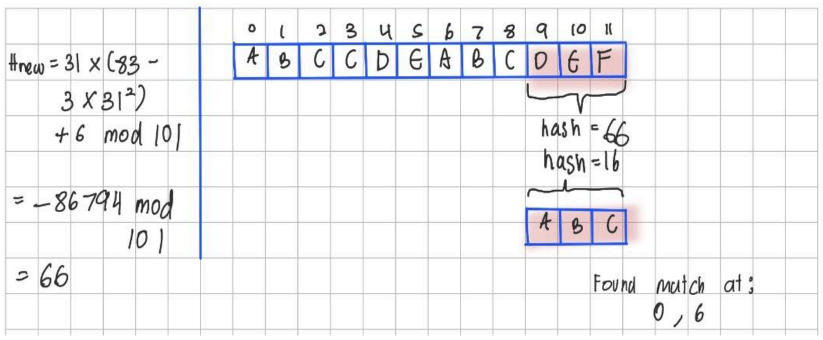
Hnew (text) $= 31 \times (100 - 2 \times 31^2) + 5 \bmod 101$
$= -56477 \bmod 101$
$= 83$

0	1	2	3	4	5	6	7	8	9	10	11
A	B	C	C	D	E	A	B	C	D	E	F

hash = 8
hash = 16

A	B	C

Found match at: 0, 6



Hnew $= 31 \times (83 - 3 \times 31^2) + 6 \bmod 101$
$= -86794 \bmod 101$
$= 66$

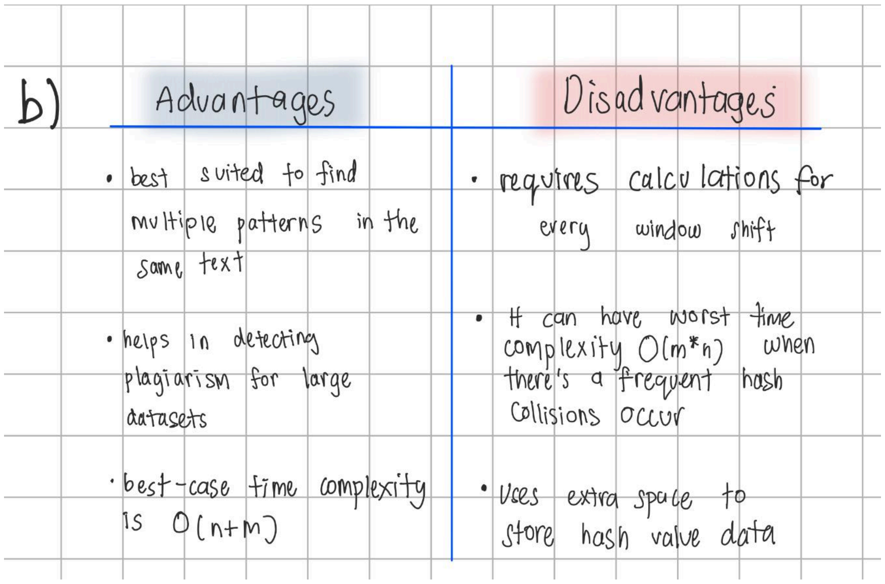
0	1	2	3	4	5	6	7	8	9	10	11
A	B	C	C	D	E	A	B	C	D	E	F

hash = 66
hash = 16

A	B	C

Found match at: 0, 6

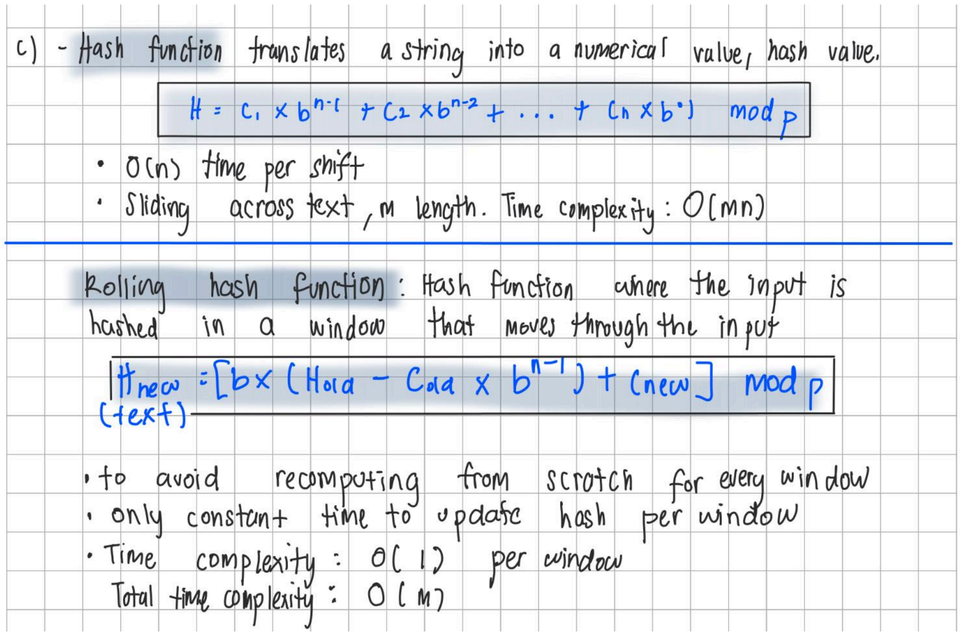
b. Highlight the advantages and disadvantages of the Rabin-Karp algorithm.



b)

Advantages	Disadvantages
• best suited to find multiple patterns in the same text	• requires calculations for every window shift
• helps in detecting plagiarism for large datasets	• It can have worst time complexity O(m*n) when there's a frequent hash collisions occur
• best-case time complexity is O(n+m)	• Uses extra space to store hash value data

c. Describe the hash function used in the Rabin-Karp algorithm and its significance and efficiency.



c) - Hash function translates a string into a numerical value, hash value.

$$\boxed{H = c_1 \times b^{n-1} + c_2 \times b^{n-2} + \dots + c_n \times b^0 \bmod p}$$

- $O(n)$ time per shift
- Sliding across text, m length. Time complexity : $O(mn)$

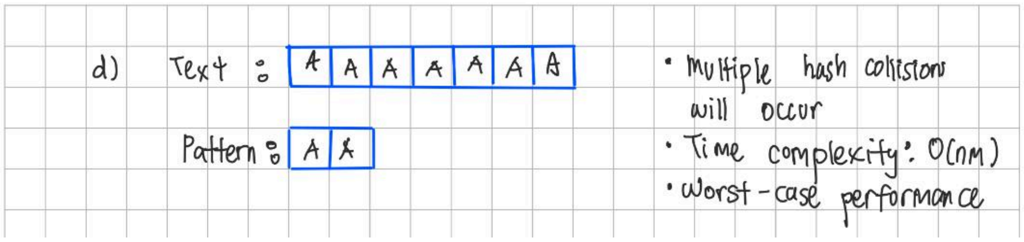
Rolling hash function : Hash function where the input is hashed in a window that moves through the input

$$\boxed{H_{new} = [b \times (H_{old} - C_{old} \times b^{n-1}) + C_{new}] \bmod p}$$

(text)

- to avoid recomputing from scratch for every window
- only constant time to update hash per window
- Time complexity : $O(1)$ per window
  Total time complexity : $O(m)$

d. Describe a scenario where Rabin-Karp performs poorly, even though it's theoretically efficient.



d) Text : | A | A | A | A | A | A | A |

Pattern : | A | A |

- multiple hash collisions will occur
- Time complexity : $O(nm)$
- worst-case performance

Q3. The Knuth-Morris-Pratt (KMP) algorithm:

a. Illustrate the KMP algorithm given the following:

Text: "ABABABACABABABACABAB"

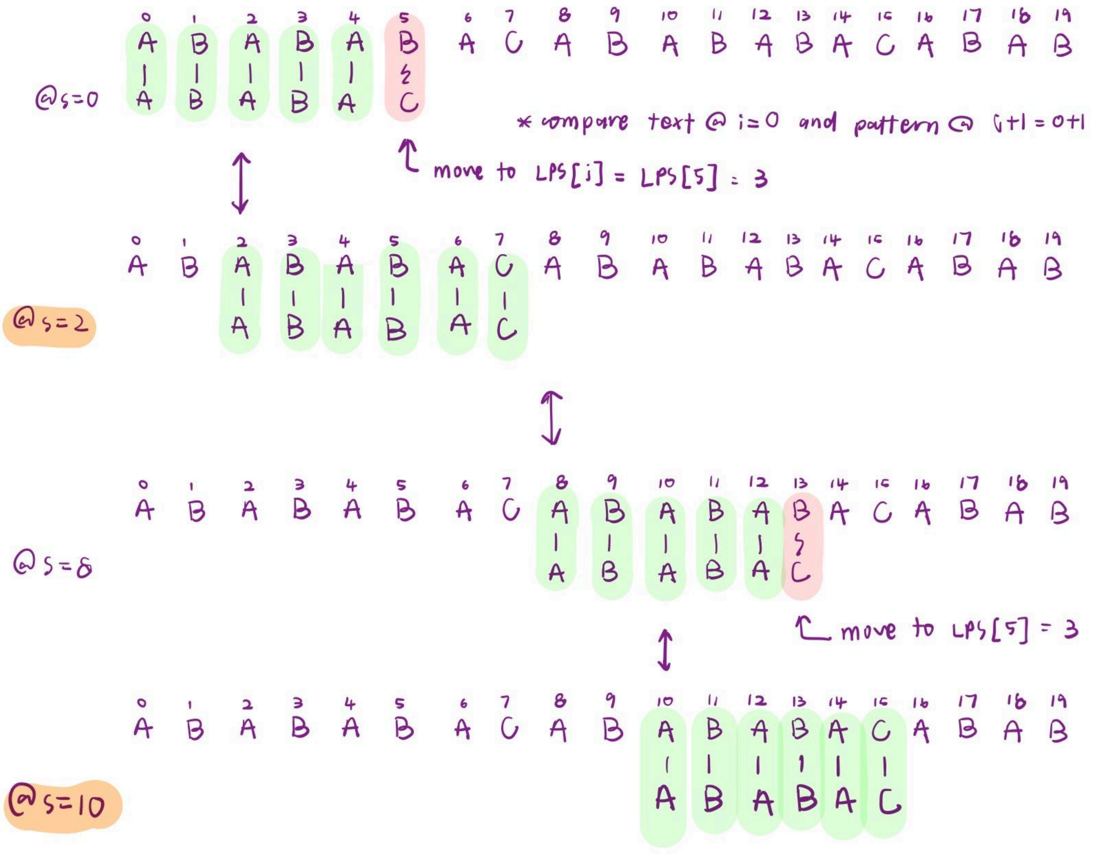
Pattern: "ABABAC"

① Build the Prefix Table (LPS Table)

LPS table : 0

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

② compare the text and the pattern



@s=0
* compare text @ i=0 and pattern @ i+1 = 0+1
move to LPS[i] = LPS[5] = 3
@s=2
@s=8
move to LPS[5] = 3
@s=10

∴ Pattern found at s=2 and s=10

b. Explain the advantages as well as the limitations of the algorithm.

Advantages:

- Efficient: runs in 0(n+m) time
- No backtracking done in the text
- Best used for large texts and patterns

Limitations:

- More complex than Naive or Rabin-Karp
- Needs extra memory for the LPS table
- Only works with exact pattern matching

c. Describe the role of the prefix function in the KMP algorithm and how it contributes to its efficiency.

- The prefix function is crucial for optimizing pattern matching efficiency by identifying repeated patterns within the pattern itself
- Allows algorithm to:
    - Reuse past matches to avoid rechecking characters
    - Avoid backtracking by shifting the pattern based on the LPS table
    - Skip unnecessary comparisons
- Thus, reducing overall time complexity from O(nm) (from Naive Algorithm) to O(n+m)

d. A text has length n = 20,000, and a pattern has length m = 200. Explain why KMP is better than the Naive algorithm in this context

- Naive algorithm

  O ((n-m+1)*m) = O ((20000-200+1)*(200)) = 3960200 comparisons

- KMP algorithm

  O (n+m) = O (20000+200) = 20200 comparisons

- KMP is much faster when the pattern is long or the text is huge, especially with partial matches.

Q4. The Finite Automata algorithm:

a. Illustrate the Finite Automata algorithm given the following:
Text:	"ABABABACABABABACABAB"
Pattern: "ABABAC"

STEP 1: Build an automata using the pattern given

$$\begin{tikzcd}
\bigcirc 0 \arrow[r, "A"] & \bigcirc 1 \arrow[r, "B"] & \bigcirc 2 \arrow[r, "A"] & \bigcirc 3 \arrow[r, "B"] & \bigcirc 4 \arrow[r, "A"] & \bigcirc 5 \arrow[r, "C"] & \circledcirc 6
\end{tikzcd}$$

STEP 2: Build the transition table

state/input	A	B	C
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	1	0	0

STEP 3: Apply in the text

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[i]:	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**C**	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**C**	**A**	**B**	**A**	**B**
0	1	2	3	4	5	4	5	**6**	1	2	3	4	5	4	5	**6**	1	2	3	4

b. Highlight the advantages, limitations, and constraints associated with finite automata in certain scenarios. Provide some real-world examples.

- **Advantages**
    - ➔ Flexible
    - ➔ Simple, making them easy to develop even for unskilled programmers
    - ➔ It is easy to go from a significant abstract to a code execution. Each state and transition corresponds directly to conditional logic
    - ➔ The reachability of a state can be easily determined.

- **Limitations and Constraints**
    - ➔ **Limited Expressive Power:** can recognize only regular languages.
    - ➔ **Inability to Count:** unsuitable for tasks that require counting or balancing symbols.
    - ➔ **Limited Memory:** have a fixed memory capacity (states), so they're not well-suited for tasks that involve extensive memory or complex history. It remembers things by changing states. The memory is present in the form of states Q only and according to the automata principle.
    - ➔ **Lack of Contextual Understanding:** cannot inherently maintain or understand context, which is crucial for many real-world applications.

- **Real-world examples**

1. Vending machines
    - ➔ **States**: Represent how much money has been inserted
    - ➔ **Transitions**: Triggered by inserting coins or making a selection.
    - ➔ **Input symbols**: Value of coins, item selections.
    - ➔ **Accepting state**: When enough money is inserted and a valid selection is made.

2. Traffic lights
    - ➔ **States**: Red, Green, Yellow.
    - ➔ **Transitions**: Triggered by timers
    - ➔ **Input symbols**: Timer ticks
    - ➔ **Cycle (continuous process)**: Transitions between states follow strict timing logic.

3. Elevator system
    - ➔ **States**: Current floor, door open/closed, moving up/down.
    - ➔ **Transitions**: Triggered by button presses or arrival at floors.
    - ➔ **Input symbols**: Floor requests, door sensors.
    - ➔ **Accepting states**: Reaching the requested floor and opening doors.

4. Designing spell checkers and text editors
    - ➔ **States**: Partial word matches as characters are typed.
    - ➔ **Transitions**: Each character typed moves to a new state in the dictionary automaton.
    - ➔ **Input symbols**: Typed characters.
    - ➔ **Accepting state**: When a complete, valid word is matched.

Q5. The Trie algorithm:

a. Illustrate the Trie algorithm given the following:

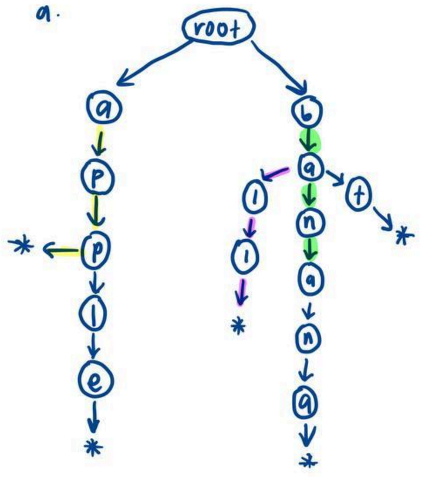
Set of words in a Trie data structure:

"apple", "app", "banana", "bat", "ball"

Search: "app", "ban", "all"

b. Discuss the advantages and potential drawbacks or limitations of using Tries in certain contexts.

a.



① "app"

→ Found

② "ban"

→ Not found

→ Prefix, but not a complete word

③ "all"

→ Not found

→ does not start from the root

(b) Advantages

1. Fast search time (O(m))
- Searching a word in a Trie takes time proportional to the length of the word (m), not the number of words stored in the Trie.

2. Avoids Hash Collisions
- Tries don't use hash functions, so there's no risk of two different words having same address.
- Hash collisions slow down performance and require extra memory to resolve.

3. Efficient Prefix Matching
- Tries naturally support prefix-based queries, like finding all words starting with "app".
- Good for
  - Predictive typing
  - Autocomplete system
  - Spell-checking

Limitations

1. High Memory Usage
→ Each node potentially has pointers to all characteristics in the alphabet.

2. Complexity in implementations
→ Involves pointers and memory allocation per characters.

3. Inefficient for small datasets

c. Provide the pseudocode and calculate the running time complexity

```
# function TriesInsert(Trie T, string word):
     node = T.root
     n = word.length
     for i = 1 to n:
         char = word[i]
         if char not in node.children:
            node.children[char] = new TrieNode()
        node = node.children[char]
    node.is_end = True  // Mark end of the word
```

```
# function TriesSearch(Trie T, string word):
    node = T.root
    n = word.length
    for i = 1 to n:
        char = word[i]
        if char not in node.children:
            return False
        node = node.children[char]
    return node.is_end  // True if word exists
```

```
# function TriesDelete(Trie T, string word):
    node = T.root
    n = word.length
    if n == 0:
        return False  // Empty word cannot be deleted
    for i = 1 to n:
        char = word[i]
        if char not in node.children:
            return False  // Word not found
        node = node.children[char]
    if not node.is_end:
        return False  // Word exists only as a prefix
    node.is_end = False  // Unmark the word
    return True
```

Time complexity = O(dn) where d=alphabet size, m=word length

Case Study 1: The Social Media

The social media platform allows users to post text-based content, including status updates, comments, and messages. With millions of active users and vast textual data generated daily, the existing text search functionality was becoming slow and inefficient.

1. Describe the challenges when implementing string matching for a social media platform.
   - **Scale of data:** Millions of users create massive amounts of text every second
   - **Variety and unstructured nature of text:** Users write in different formats, styles, slang, abbreviations, emojis, hashtags
   - **Real-time requirements:** Users expect instant results when searching or typing — autocomplete, suggestions
   - **Need for relevance and ranking:** Matching isn't enough — users want relevant results (e.g., most recent, most liked, most related).
   - **Security and privacy:** Matching algorithms must respect privacy settings (e.g., private messages shouldn't appear in public search)

2. Suggest a possible implementation and justify the selection.

   Implementation of Full-Text Search Using Elasticsearch

   Elasticsearch is a highly scalable, open-source full-text search engine built on top of Apache Lucene. It enables fast, accurate, and flexible text search over large datasets, which is essential for social media platforms.

   Key Features of Elasticsearch

- **Inverted Indexing:** Enables fast retrieval of documents containing specific terms.
- **Fuzzy Search:** Supports typo tolerance and approximate string matching.
- **Scalability:** Easily handles large datasets through horizontal scaling across clusters.
- **Near Real-Time Indexing:** New posts and comments become searchable within seconds.
- **Custom Analyzers:** Supports tokenization and stemming for multiple languages.

Case Study 2: DNA Sequence Analyser

A researcher needs to search for patterns like ATGCATG in a long DNA string (up to 2 million characters). Sometimes multiple patterns must be matched simultaneously.

1. Which string matching algorithm would be most efficient and why?
2. What are the trade-offs between using KMP, Rabin-Karp, and Trie-based algorithms?

Case Study 2.

Case Study 2: DNA Sequence Analyser

A researcher needs to search for patterns like ATGCATG in a long DNA string (up to 2 million characters). Sometimes multiple patterns must be matched simultaneously.

1. Which string matching algorithm would be most efficient and why?
2. What are the trade-offs between using KMP, Rabin-Karp, and Trie-based algorithms?

1. KMP

→ Ideal for searching a single pattern in a very long text like DNA sequence.

→ Avoids unnecessary comparisons by using prefix table that helps it skip characters when mismatches occur.

$T(n) = O(n+m)$

X Naive

→ Too slow for long DNA

X Rabin-Karp

→ Hash collisions make it unreliable

→ Not ideal for biological sequences w low alphabet diversity.

X Finite Automaton

→ requires large transition table

X Trie

→ Can't search within continuous text

2.

a) KMP

✓
- Efficent for single pattern match.

X
- Not efficent for multiple patterns, need to run for each patter seperately.

2) Rabin-Karp

✓
- Can be very fast on average when using multiple patterns.

X
- Hash collisions, which make it unsuitable for many patterns.

3) Trie-based

✓
- Multiple pattern matching

X
- Not optimal for large sequences

Case Study 3: Spam Filter in Messaging App

A chat application must detect whether predefined keywords (e.g., buy now, win, free, urgent) appear in real-time messages.

1. What characteristics of this problem make it suitable (or unsuitable) for different algorithms (e.g, finite automaton, Rabin-Karp, Trie, KMP)?

Characteristics of the Spam Filter:

- **Multi-Pattern Matching**
  - Need to detect multiple predefined keywords.
- **Real-Time Processing**
  - Messages must be scanned instantly (low latency).
- **Dynamic Keyword Updates**
  - Keywords may be added/removed frequently(new spam terms).
- **Variable-Length Keywords**
  - Keywords can be single words (*"free"*) or phrases (*"buy now"*).
- **Prefix/Suffix Overlaps**
  - Some keywords may share prefixes (*"free" and "freedom"*).

Characteristics/ Algorithm	**Multi-Pattern Matching**	**Real-Time Processing**	**Dynamic Keyword Updates**	**Variable-Length Keywords**	**Prefix/Suffix Overlaps**
Naive	Checks every keyword separately	Slow – Linear scan for every pattern	Updates instantly since there is no preprocessing	Handles all lengths	Repeats work on shared prefixes
Finite-automaton	Super fast multi-pattern search	One-pass message scanning	Can't update keywords without full rebuild	Handles all lengths	Shares prefixes across keywords
Rabin -Karp	Uses rolling hashes to compare patterns(Slo	Hash collisions can degrade performance	Needs hash recomputation	Better for fixed lengths	Doesn't track prefix/suffix overlaps

	wer for many keywords)				
KMP	Single-pattern only	Fast for single words but scales poorly with many keywords	Slow -It must recompute its LPS cheat sheet every time	Handle all lengths	Only optimizes repetitions in one pattern (no Cross-Pattern Optimization)
Tries	Built for multiple patterns	Finds matches in one message pass / one traversal	Updates instantly without rebuilding	Handle all lengths	Shares prefixes across keyword

2. Propose a solution that ensures high performance and low latency.
    - **Tries Algorithm**
1. **Multi-Pattern Efficiency**
    - Tries check all keywords at once in a single pass, eliminating redundant scans.
2. **One-Pass Scanning**
    - The message is scanned just once, from start to finish.
3. **Instant Updates**
    - No need to rebuild the entire system
    - Updates happen instantly without slowing down filtering.
4. **Handles Any Keyword Length**
    - Short words (like "win") and long phrases (like "limited-time offer") are processed just as efficiently.
5. **Shared Prefix Optimization**
    - Common word starts (e.g., "free" and "freedom") are stored once, saving memory and reducing search time.

3. Describe how you would update the filtering system dynamically as new keywords are added or removed.
    - Optimize the matching algorithm by using Aho-Corasick, which matches multiple keywords in a single pass
    - Add caches to store frequently detected keywords
    - Use 2 types of pattern matching, one algorithm is for the precompiled keywords, 2nd algorithm is for the updated structure of new keywords.