Advanced Data Structures and Algorithms

Topics: String Operations and Brute Force Pattern Matching

Presented by: RISHAB BISWAS(A125017) & SIDDHARTH KUMAR(A125021)

M.Tech Academic Presentation

Introduction to String Processing

- String processing refers to algorithms and techniques for manipulating and analyzing strings of text.
- Common operations include searching, matching, concatenation, and substring extraction.
- Applications: text editors, DNA sequence analysis, search engines, and data compression.

Common String Operations

- Concatenation: Joining two or more strings together.
- Substring extraction: Extracting parts of a string.
- String comparison: Lexicographical comparison.
- String reversal: Reversing order of characters.
- Palindrome checking: Checking if a string reads same backward and forward.

Importance of String Processing

- Used in text mining and information retrieval.
- Helps in natural language processing (NLP) tasks.
- Essential for compiler design (tokenization, parsing).
- Basis for pattern matching algorithms and data compression.

Introduction to Pattern Matching

- Pattern matching involves finding occurrences of a substring (pattern) within a larger string (text).
- Two main approaches: Brute Force and Efficient Algorithms (like KMP, Rabin-Karp, Boyer-Moore).
- Brute Force is the simplest approach — checks all possible positions.

Brute Force Pattern Matching

- The algorithm checks for the pattern starting from every position in the text.
- If mismatch occurs, it shifts the pattern by one position and compares again.
- Time Complexity: $O(n \times m)$, where n = length of text and m = length of pattern.
- Space Complexity: $O(1)$.

Steps of Brute Force Pattern Matching

- Start from the first index of the text.
- Compare pattern characters with the text sequentially.
- If all characters match, record the position.
- If mismatch, move one position forward in text and repeat.
- Continue until the end of text.

Brute Force Pattern Matching(Pseudocode)

- Algorithm BruteForceMatch(T, P)
- # T → Text of length n
- # P → Pattern of length m

- 1. n ← length(T)
- 2. m ← length(P)

- 3. for i ← 0 to n - m do
- 4. j ← 0
- 5. while j < m and T[i + j] = P[j] do
- 6. j ← j + 1
- 7. if j = m then
- 8. print("Pattern found at index", i)
- 9. end for

Example of Brute Force Pattern Matching

- Text: A B C D A B D A B C A B D A B C
- Pattern: A B D
- Matches found at positions: 5 and 10 (1-based index).

Advantages and Disadvantages

- Advantages:
- Simple to implement and understand.
- Works for small text and pattern sizes.
- Disadvantages:
- Inefficient for large strings.
- Repeated comparisons lead to high time complexity.

Comparison with Other Pattern Matching Algorithms

- KMP Algorithm: Uses preprocessing to avoid redundant comparisons (O(n + m)).
- Rabin-Karp Algorithm: Uses hashing for pattern matching.
- Boyer-Moore Algorithm: Compares from right to left, skips sections of text efficiently.
- Brute Force: Basic method; foundation for understanding other algorithms.

Applications of String Processing & Pattern Matching

- Search engines and text processing tools.
- Bioinformatics (DNA sequence matching).
- Plagiarism detection and data validation.
- Spell checkers and auto-correct systems.
- Intrusion detection systems in cybersecurity.

Conclusion

- String processing and pattern matching are fundamental in data structures and algorithms.
- Brute force provides the foundation to understand optimized algorithms.
- Efficient pattern matching is vital for large-scale data handling and text analytics.

Time Complexity Analysis of Brute Force Pattern Matching

- Let n = length of text, m = length of pattern.
- Algorithm checks for the pattern starting from every position in the text.
- For each position, up to m character comparisons may occur.
- → Overall complexity depends on number of matches and mismatches.

Best, Average, and Worst Case Analysis

- • Best Case: Immediate mismatch at each position → O(n)
-   Example: Text = 'AAAAAA', Pattern = 'B'
- • Average Case: Partial matches before mismatch → O(n × m)
- • Worst Case: Long prefix matches at each shift → O(n × m)
-   Example: Text = 'AAAAAA', Pattern = 'AAAAB'

Example of Time Complexity (Step-by-Step)

- Text: ABABABABAB | Pattern: ABAB
- → i = 0 → Match (4 comparisons)
- → i = 1 → Mismatch after 3 comparisons
- → i = 2 → Match (4 comparisons)
- → Average comparisons ≈ 3–4 per shift → $O(n \times m)$
- Space Complexity: $O(1)$

Summary Table of Time Complexity

- Case | Description | Time Complexity
- --
- Best | Immediate mismatch | O(n)
- Average | Partial match at some positions | O(n × m)
- Worst | Full match before mismatch | O(n × m)
- Space Complexity = O(1)

Thankyou!!!

(Questions???)