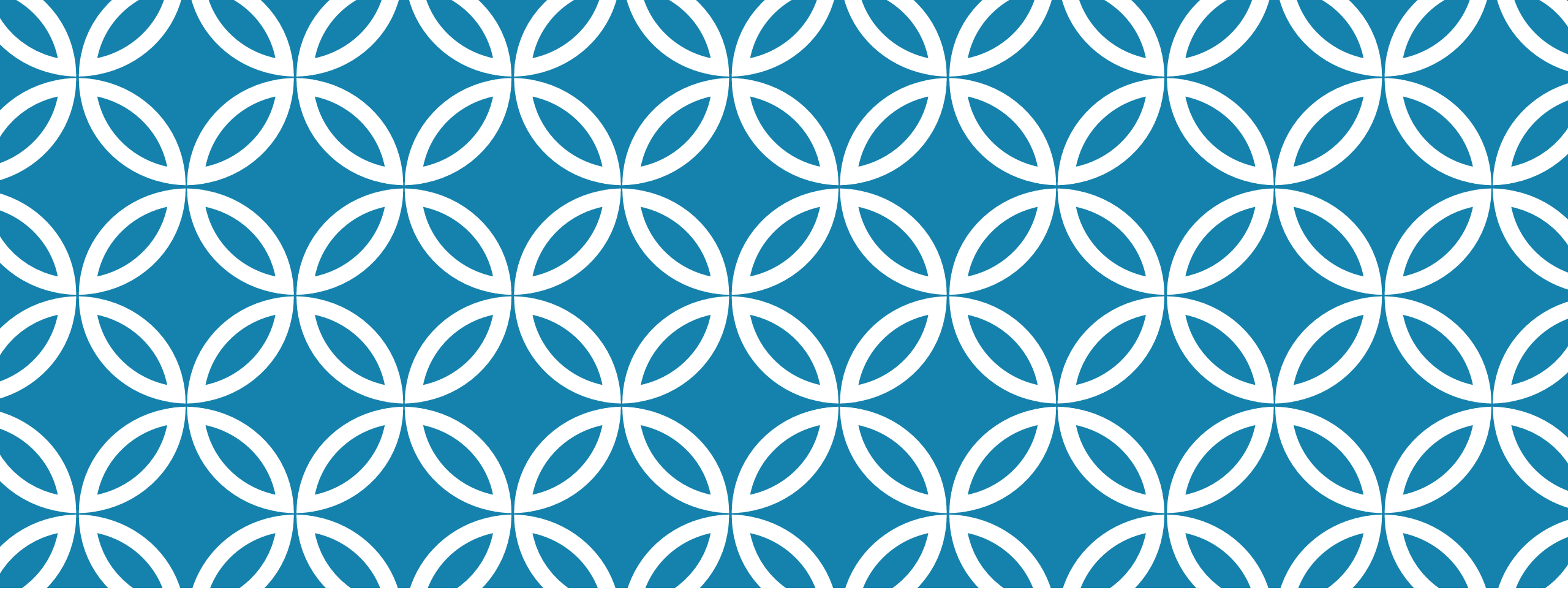


APPENDIX 1: ASSIGNMENT 1, INTRODUCTION TO PYTHON

STRING SEARCH & PATTERN MATCHING

NOTES & REFERENCES

Research Summary of REGEX, Algorithms & Use Cases

Charles Bergin, 10.1.2025

EXECUTIVE SUMMARY

Selected String Search & Pattern Matching Approaches for Short Literary Texts

Key References:

- A Fast string-Matching Algorithm, R. S. Boyer, J. S. Moore (ACM, 1977)
- Analysis of Multiple String Pattern Matching Algorithms A. I. Jony, (IJASCIT, 2014)
- Multiple Skip Multiple Pattern Matching Algorithm (MSMPMA), Z. Qadi, M.J. Aqel, I. El Emary (IJSC, 2007)

APPROACHES

- Regex
  - Regex engines internally use optimized algorithms like Finite State Automata (FSA) or Thompson's NFA to match patterns of varying complexity, from fixed strings to complex rules (e.g., optionality, repetition).
  - Regex processes the entire string using internal algorithms without explicitly managing a "window." Consequently, time complexity depends on the regex pattern and engine design:
    - For simple patterns: O(n)) with linear scans.
    - For complex patterns: With excessive backtracking, worst-case performance can degenerate to O(2n).

- Algorithms
  - Sliding Window Optimization and Heuristics
    - The sliding window optimization involves preprocessing the pattern or string and using a 'window' of elements to incrementally move through the string and reusing information to reduce unnecessary comparisons.
    - The sliding window algorithms achieve efficiency by with Optimization, where hash values are updated as the window slides (e.g., Rabin-Karp), and Heuristics, using precomputed bad character or good suffix tables (e.g., Boyer-Moore).
  - Trie Algorithms
    - A trie approach builds a tree–like data structure to store a set of strings where nodes represent characters and paths represent sets of strings.
    - Tire algorithms are efficient for multi-pattern searching and prefix-based search of large datasets

STRING SEARCH & PATTERN MATCHING

An overview of the types and use cases for different pattern matching and string search approaches.

Type	Description	Use Cases
Exact String Matching	Find exact matches of a pattern in a text	Text searching, keyword matching, pattern recognition
Approximate String Matching	Match strings with allowed differences (fuzzy matching)	Spell checkers, DNA sequence comparison, name matching
Multiple Pattern Matching	Find multiple patterns in a single text	Searching multiple keywords, virus signature detection
Pattern Matching with Wildcards	Search with placeholders (wildcards)	File searches, shell commands, pattern matching in text
Pattern Matching with Regex	Search for complex patterns using regular expressions	Data validation, log parsing, text processing
Substring Search	Search for fixed-length substrings	Text extraction, basic substring search
Longest Common Substring (LCS)	Find the longest substring common to two strings	Bioinformatics, file comparison
Longest Common Subsequence (LCS)	Find the longest sequence of characters common t vo strings	Version control, DNA sequence comparison

STRING SEARCH V. PATTERN SEARCH ALGORITHMS

String search algorithms and pattern search algorithms both deal with identifying specific sequences of characters within a larger text but have distinct purposes and methodologies

Key Differences

Feature	String Search Algorithms	Pattern Search Algorithms
Input	A single, fixed string.	A pattern (may include wildcards, rules).
Complexity	Handles simple searches.	Handles complex and flexible searches.
Speed	Generally faster for exact matches.	May be slower due to pattern complexity.
Tools	Comparisons and heuristics.	Regular expressions, automata, etc.
Examples of Use	Searching "hello" in text.	Finding all words starting with "a".
Flexibility	Less flexible.	Highly flexible.

KEY ALGORITHMS

Aho–Corasick Algorithm

- Introduced in 1975, the Aho–Corasick algorithm is a multi-pattern string matching algorithm designed to efficiently find instances of multiple patterns in a text simultaneously. It constructs a trie structure of the patterns, augmented with failure links to handle mismatches and allows for seamless transitions between patterns during the search phase. This preprocessing results in a time complexity of O(n+m+k), where n is the text length, m is the total length of all patterns, and k is the number of matches. This algorithm is widely used in applications like text searching, intrusion detection, and bioinformatics.

Knuth-Morris-Pratt (KMP) algorithm

- Introduced in 1977, the Knuth-Morris-Pratt (KMP) algorithm is a linear-time string matching algorithm designed to efficiently locate substrings within a text. It avoids redundant comparisons by preprocessing the pattern to create a Longest Proper Prefix which is also a Suffix (LPS) table. This table allows the algorithm to skip unnecessary re-evaluations of already matched characters when a mismatch occurs. During the search phase, the algorithm leverages the LPS table to align the pattern dynamically, reducing the overall number of comparisons. With a time complexity of O(n + m), where n is the length of the text and m is the length of the pattern, KMP is particularly effective for scenarios with repetitive patterns or substrings, though it is less suited for handling complex pattern matching tasks.

Boyer-Moore Algorithm

- Proposed in 1977, the Boyer-Moore algorithm is a string matching algorithm that achieves high efficiency by skipping sections of the text during the search phase. It leverages two heuristics: the bad character rule and the good suffix rule, which determine how far the pattern can be shifted when a mismatch occurs. These rules significantly reduce the number of comparisons, especially in practical scenarios where the pattern is long or the alphabet is large. The algorithm's worst-case time complexity is O(n · m) but it performs close to O(n / m) on average for random text and patterns.

Commentz-Walter Algorithm

- Developed in 1979, the Commentz-Walter algorithm extends the Boyer-Moore algorithm to handle multiple pattern searches efficiently. It combines the concepts of the Boyer-Moore algorithm for individual patterns and the Aho–Corasick algorithm for managing multiple patterns, building a trie structure with precomputed shift tables. This hybrid approach allows for rapid skipping of text sections during mismatches while accommodating multiple patterns. Its time complexity is O(n+m+k), where n is the text length, m is the total length of all patterns, and k is the number of matches, making it suitable for high-performance multi-pattern searching tasks.

Rabin-Karp Algorithm

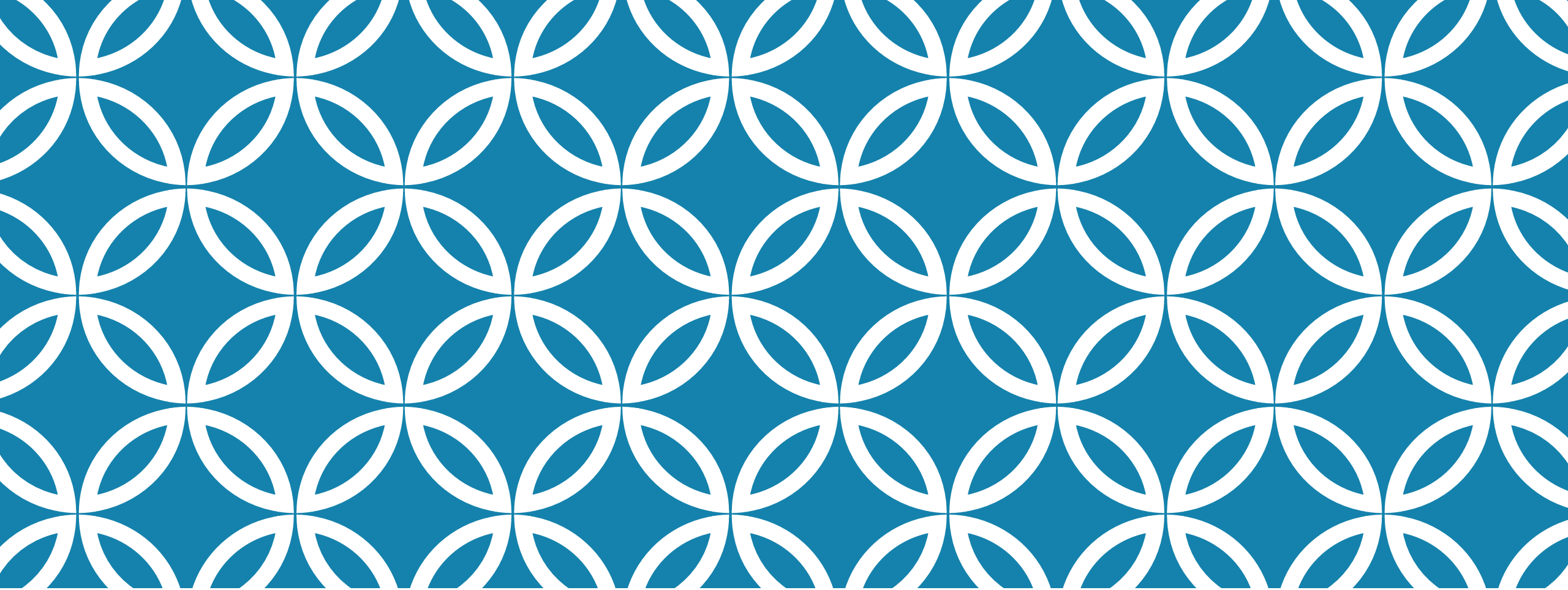
- Developed in 1987, the Rabin-Karp algorithm is a string matching algorithm that utilizes hashing to efficiently search for patterns within a text. It works by calculating a hash value for the pattern and each substring of the text of the same length, allowing for quick identification of potential matches. In case of a hash match, the algorithm verifies the actual characters to confirm the match, ensuring correctness. While its average-case time complexity is O(n+m), where n is the text length and m is the pattern length, its worst-case complexity is O(n·m), particularly if hash collisions occur frequently.

Wu-Manber algorithm

- Introduced in 1994, the Wu-Manber algorithm is an efficient multi-pattern string matching algorithm designed for high-performance searching in large-scale texts. It extends the Boyer-Moore algorithm by leveraging hash tables and shift tables to handle multiple patterns simultaneously. The algorithm preprocesses the patterns to compute a shift table and uses a hashing mechanism to identify candidate matches quickly. Its key innovation lies in minimizing the amount of text scanned during mismatches, making it particularly well-suited for applications like text indexing, network intrusion detection, and digital forensics. The Wu-Manber algorithm achieves a time complexity of O(n) on average, where n is the text length, and excels in scenarios with a large number of patterns

SECTION 1: KEY CONCEPTS

1. String & Pattern Matching
2. Regular Expressions
3. Sliding Window Algorithms



STRING & PATTERN MATCHING

Overview

STRING SEARCH & PATTERN MATCHING

1. Exact String Matching

Definition:

- Find exact matches of a pattern (substring) within a text

Characteristics:

- The pattern is fixed, and the task is to find all occurrences of that pattern in the text.
- The match must be exact (no variations allowed).

Use Cases:

- Searching for keywords in documents.
- Finding exact matches of a pattern in large datasets, such as searching for a term in a log file

2. Approximate String Matching

Definition:

- Fuzzy Matching: finding substrings that are close enough to a pattern, even if they aren't identical. This is often used when there may be errors, typos, or fuzzy data

Characteristics:

- Allows for edit operations such as insertions, deletions, or substitutions.
- Measures how close two strings are, typically using distance metrics.

Use Cases:

- Spell checkers and auto-correction (e.g., finding words with typos).
- DNA sequence matching (where mutations might cause variations in the string).
- Matching names that are spelled differently (e.g., “Jon” vs. “John”)

STRING SEARCH & PATTERN MATCHING

3. Multiple Pattern Matching

Objective:

- Multiple patterns (substrings) within a given text at the same time

Characteristics:

- Requires searching for more than one pattern in the same string or text.
- Efficient algorithms are needed to avoid searching the text repeatedly for each pattern.

Use Cases:

- Searching for multiple keywords or phrases in a large corpus (e.g., searching for multiple words in a document).
- Network packet analysis or virus signature detection (searching for multiple known virus signatures).

4. Pattern Matching with Wildcards

Objective:

- Searching for patterns with wildcards or placeholders that can match any character(s)

Characteristics:

- The pattern may include wildcard characters such as * (any number of characters) or ? (exactly one character).
- It's commonly used in filename matching or regular expressions.

Use Cases:

- File search in a directory (e.g., matching all .txt files).
- Matching user input patterns like email addresses or phone numbers

PATTERN MATCHING WITH REGULAR EXPRESSIONS

Objective

- A more advanced form of string matching that allows complex patterns to be defined and matched

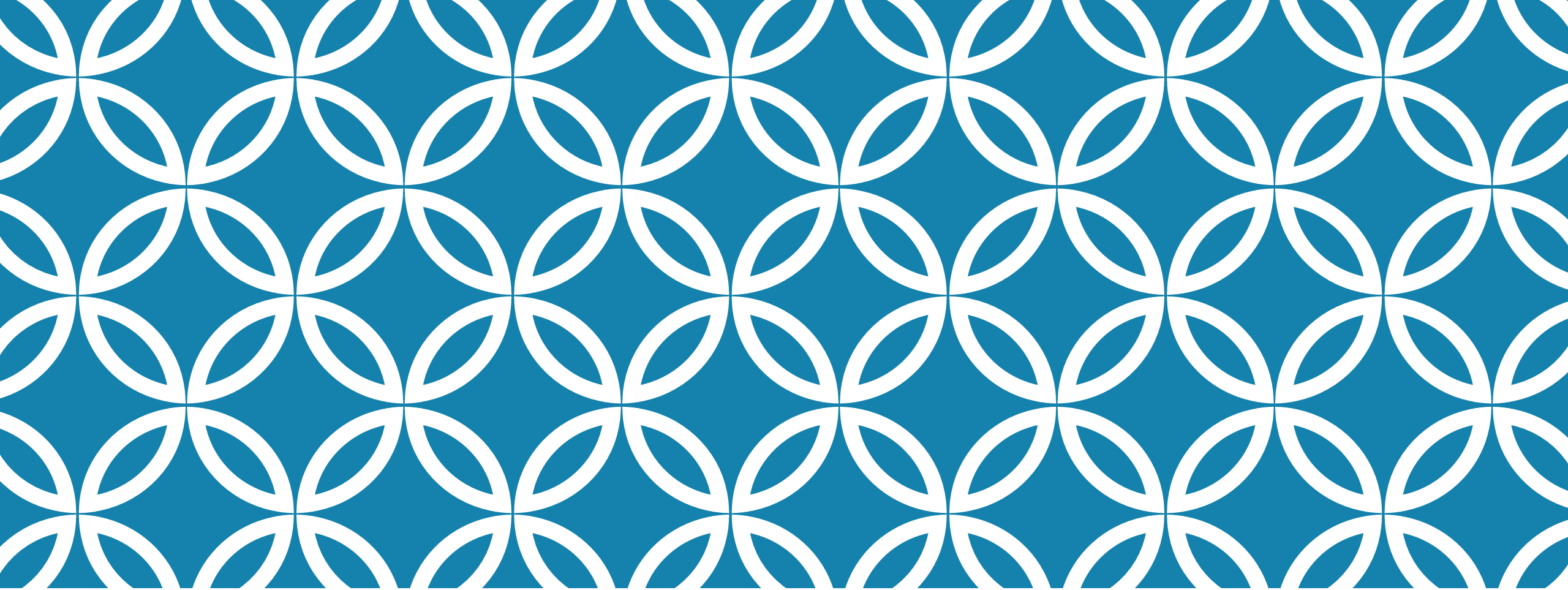
Characteristics:

- Regex patterns can match simple strings or complex structures like numbers, words, or dates.
- Supports wildcards, repetitions, grouping, and more.

Use Cases:

- Data validation (e.g., validating email addresses, phone numbers, or date formats).
- Log parsing and extraction of patterns from unstructured data.
- Text processing (e.g., finding specific patterns, replacing text).

- Algorithms:
  - Finite Automata
    - Used internally by regex engines like PCRE, RE2, etc. Automata-based matching that processes regular expressions efficiently.
  - Thompson's Construction:
    - Used to convert a regular expression to a nondeterministic finite automaton (NFA), then to a deterministic finite automaton (DFA).



REGULAR EXPRESSIONS

Background, Finite State Automata, Practice v. Theory

BACKGROUND TO REGEX

1: Regular Expressions (Regex)

- Definition: A regular expression (regex or regexp) is a sequence of characters that defines a search pattern in text. Used in text searching, find and replace operations, and input validation.
- Origin: Concept formalized by Stephen Cole Kleene in the 1950s for describing regular languages. Became popular through Unix text-processing tools.
- Syntax Variations: Different regex syntaxes exist, with POSIX and Perl being among the most widely used.

2: History and Evolution of Regex

- 1950s: Stephen Cole Kleene introduces regular languages in theoretical computer science.
- 1960s-1970s: Early implementations in Unix tools like QED and grep; regexes become integral to text editors and compilers.
- 1980s: Perl expands regex capabilities; Henry Spencer develops Advanced Regular Expressions used in Tcl and other projects.

3: Modern Applications and Implementations

- Widespread Use: Regexes are widely supported in programming languages (Java, Python, Perl) and text processing tools.
- PCRE Development: Philip Hazel's PCRE (Perl Compatible Regular Expressions) from 1997 enhances regex performance in tools like PHP and Apache.
- Recent Innovations: Hardware implementations (FPGA, GPU) offer faster regex processing compared to traditional CPU methods.

REGEX & FINITE STATE AUTOMATION

A Finite State Automaton (FSA) is a mathematical model of computation used to represent and manipulate regular languages (or patterns). It consists of a finite number of states and transitions between those states based on input symbols. FSAs are often used for tasks such as string matching, lexical analysis, and regular expression evaluation.

A FSA is defined by the following components:

- States (Q): A finite set of states, one of which is the start state, and some of which may be accepting (final) states.
- Alphabet (Σ): A finite set of symbols (input characters) that the automaton processes.
- Transitions (δ): A set of rules that describe how the automaton moves from one state to another, depending on the input symbol. The transition function δ is typically represented as a table or diagram.
- Start State (q_0): The state where the automaton begins.
- Accepting States (F): A set of states where, if the automaton ends up in one of them after processing the entire input, the input is considered accepted by the automaton.

Types of Finite State Automata

- Deterministic Finite Automaton (DFA):
  - In a DFA, for each state and input symbol, there is exactly one transition to a new state.
  - It can be in only one state at a time.
  - No ambiguity: given a current state and an input symbol, the next state is determined unambiguously.
- Non-deterministic Finite Automaton (NFA):
  - In an NFA, for a given state and input symbol, there can be multiple possible next states (or even no state at all).
  - An NFA can be in multiple states simultaneously (conceptually) because of the non-determinism.
  - It is more flexible than a DFA but may require additional work to simulate (e.g., via a DFA conversion).

- Finite State Automata (FSAs) are powerful tools for recognizing patterns and processing regular languages. They are foundational in many areas, including string matching, text processing, and lexical analysis. FSAs are divided into deterministic and non-deterministic types, each with its own strengths and use cases. They work by transitioning between states based on input symbols and accepting or rejecting strings based on the final state reached after processing all the input.

How Finite State Automation works?

How an FSA Works

- An FSA operates by reading an input string (a sequence of symbols) and transitioning between states based on the input symbols. Here's a step-by-step breakdown of how it works:
- Step 1: Start in the initial state
  - The automaton begins in a start state (denoted q_0).
- Step 2: Read the input symbols
  - The automaton reads each symbol from the input string, one at a time.
- Step 3: Transition between states
  - For each input symbol, the automaton follows the transition function δ that tells it which state to move to.
  - The automaton moves from one state to another based on the transition rules defined in the transition function.
- Step 4: Reach a final state
  - After reading all the input symbols, the automaton checks if it is in an accepting state.
    - If it is in one of the accepting (final) states, the input string is accepted by the automaton.
    - If it is not in an accepting state, the input string is rejected.
- Step 5: Accept or reject the string
  - The automaton either accepts or rejects the string based on whether it ends in an accepting state after reading all the symbols.

Key Features of FSAs

- Deterministic vs. Non-Deterministic: In a DFA, for each input symbol, there is only one state transition, whereas in an NFA, multiple transitions (or none) may exist for a given input.
- State Transitions: FSAs rely on transitions between states based on input symbols. In a DFA, each state has exactly one transition for each symbol; in an NFA, there can be multiple transitions or no transition for a given symbol.
- Acceptance Criteria: An input string is accepted by an FSA if, after processing all the input symbols, the automaton ends in an accepting state.
- Language Recognition: FSAs are used to recognize regular languages, which can be described by regular expressions.
- Applications of Finite State Automata
- String Matching: Used in text search engines, lexical analyzers, and pattern matching (e.g., regular expression matching).
- Lexical Analysis: Used in compilers to recognize tokens in the source code (such as keywords, operators, identifiers).
- Text Processing: Efficiently identifies whether a string matches a pattern defined by regular expressions.
- Modeling Systems: FSAs are used to model many systems in fields like control theory, linguistics, and network protocols.

DETERMINISTIC V. NON-DETERMINISTIC FINITE AUTOMATION

Aspect	DFA	NFA
Time Complexity	O(n) (linear time)	O(n * 2^m) (exponential in worst case)
Space Complexity	O(m) (linear in the number of states)	O(m) (linear in the number of states during simulation, but can grow to O(2^m) during simulation)
Conversion Time	N/A	O(2^m) for NFA to DFA conversion
State Growth	Fixed number of states (no exponential growth)	Exponential growth in the number of states during simulation
Efficient For	Large input size, deterministic behavior	Small or medium-sized input strings, complex patterns that can be modeled more compactly

REGEXES

1: Formal Definition of Regular Expressions

- Components:
  - Constants: Denote sets of strings.
  - ∅: Empty set.
  - ε: Set containing only the empty string.
  - Literal Character: Denotes a set with a single character.
- Operations:
  - Concatenation (RS): Combines strings from sets R and S.
  - Alternation (R|S): Union of sets R and S.
  - Kleene Star (R*): Set of all strings formed by concatenating zero or more strings from R.

2: Examples of Regular Expressions

- Simple Examples:
  - `a|b*`: Matches ε, "a", "b", "bb", "bbb", etc.
  - `(a|b)*`: Matches all strings with "a" and "b", including the empty string.
  - `ab*(c|ε)`: Matches "a", "ac", "ab", "abc", etc.
- Complex Example:
  - `(0|(1(01*0)*1))*`: Matches binary numbers that are multiples of 3.

3: Expressive Power and Compactness

- Extended Operators:
  - `a+ = aa*` (one or more occurrences of "a").
  - `a? = (a|ε)` (zero or one occurrence of "a").
- Compactness:
  - Some languages require exponentially larger DFAs compared to their regular expression representation.
  - Example: The language Lk (strings whose kth-from-last letter is "a") can be compactly described by regex but needs 2^k states in a DFA.

4: Equivalence and Algebraic Laws

- Deciding Equivalence:
  - Algorithms exist to determine if two regular expressions describe the same language by converting them to minimal DFAs.
- Algebraic Laws:
  - Equational and Horn clause axioms can be used to establish algebraic laws for regular expressions.
  - despite their simplicity, regular expressions cannot always be systematically rewritten to a normal form.

COMMON PITFALLS IN REGEXES

Catastrophic Backtracking.

- Regular expression seems to take forever, or crashes. Can be resolved by being specific about what should be matched, so the number of matches the engine attempts doesn't rise exponentially.

Making Everything Optional.

- If all the parts of the regex are optional, it matches a zero-length string anywhere. The regex needs to express the facts that different parts are optional depending on which parts are present.

Repeating a Capturing Group vs. Capturing a Repeated Group.

- Repeating a capturing group will capture only the last iteration of the group. Capture a repeated group in order to capture all iterations.

Mixing Unicode and 8-bit Character Codes.

- Using 8-bit character codes like \x80 with a Unicode engine and subject string may give unexpected results.

Source: https://www.regular-expressions.info

THEORETICAL V. PRACTICAL APPLICATION

Formal Regular Expressions:

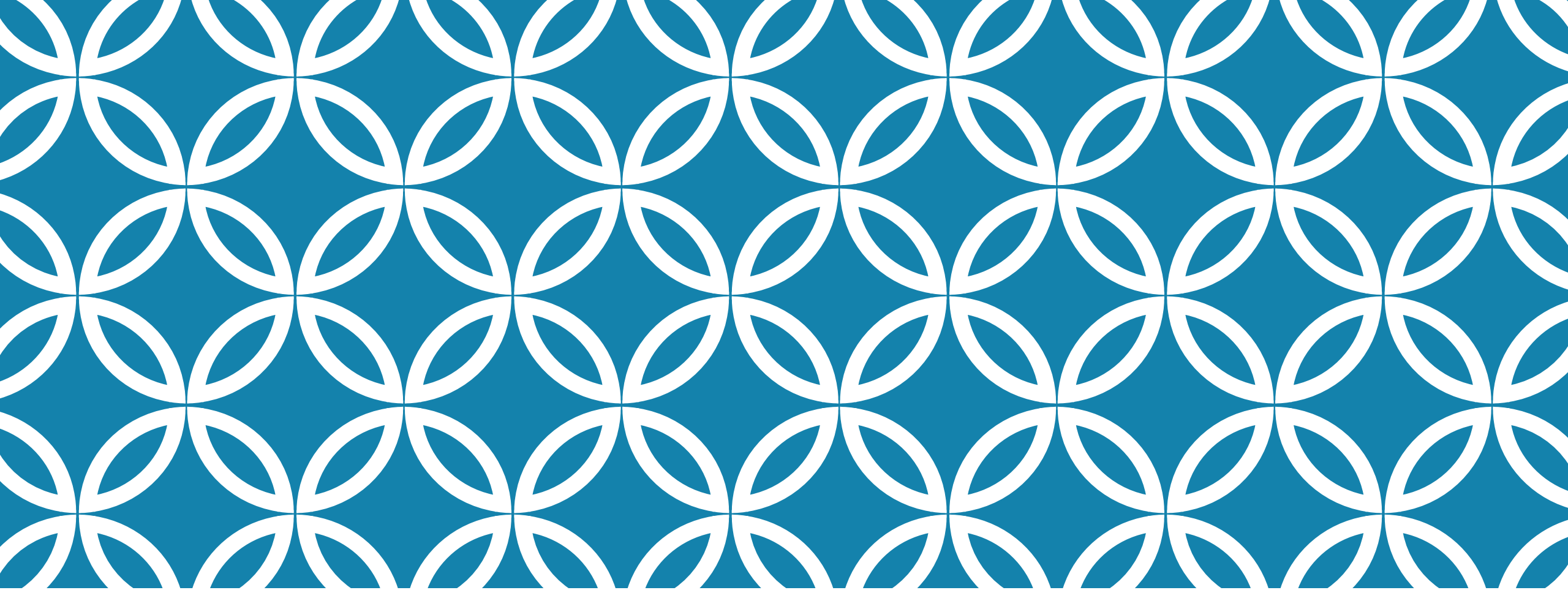
- Rooted in automata theory, regular expressions are a formalism for defining patterns that can be represented by Nondeterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA).
- Thompson's construction algorithm is commonly used to convert a formal regular expression into an equivalent NFA, facilitating efficient pattern matching.
- These patterns are limited to constructs like concatenation, alternation, and the Kleene star, adhering strictly to formal language theory.

Practical Regexes:

- Regexes, as implemented in real-world programming environments, extend the capabilities of formal regular expressions.
- Features like lookahead/lookbehind assertions, backreferences, and lazy quantifiers go beyond the expressive power of NFAs and DFAs, making them unsuitable for strict theoretical models.
- These extensions enable regex engines to handle more complex text-processing tasks but introduce computational trade-offs, such as increased processing complexity in certain cases.

Key Takeaway:

- While formal regular expressions are a powerful tool within the bounds of theoretical computer science, practical regex engines are designed for real-world needs, offering enhanced capabilities that exceed the limitations of formal regular expressions. This distinction is crucial when transitioning between theoretical and applied contexts.



KEY ALGORITHMS

- Sliding Window Optimization & Heuristics
- Trie Algorithms

SLIDING WINDOW APPROACH

The sliding window approach for pattern search typically involves examining a fixed-length portion (or "window") of the input text as it moves across the text. The Big O time complexity for this approach depends on the specific implementation and whether additional optimizations (e.g., hashing) are used.

For practical purposes, algorithms like Rabin-Karp or Boyer-Moore are preferred over naive sliding window methods due to their potential for skipping unnecessary comparisons or leveraging efficient hash computations.

Sliding Window Without Optimizations

- In a naive implementation, the algorithm checks every possible substring of length m (pattern length) within a string of length n, comparing each character in the window to the pattern.
- Time Complexity: For each of the $n-m+1$ windows, m comparisons are made (one for each character in the pattern).
- Total Complexity: $O(n \cdot m)$.

Optimized Sliding Window: O(n) average case with hashing.

Heuristic Approaches: O(n/m) best case when large skips occur.

OPTIMIZATION & HEURISTICS

Sliding Window Approach with Optimzation

In optimized approaches like Rabin-Karp, the sliding window maintains a hash of the current substring. Instead of comparing all characters in the substring with the pattern, we compare their hash values. If the hash matches, a character-by-character comparison is performed as a final check.

Time Complexity:

- Precompute the hash of the pattern and the first window: O(m).

Slide the window across the text:

- Update the hash in O(1) for each of the windows.
- Perform character comparisons only for hash matches (in the worst case, all windows): O(n·m).

Complexity:

- Average: O(n), assuming hash collisions are rare.
- Worst Case: O(n·m), if many hash collisions occur.

Sliding Window Approach with Heuristics

In algorithms like Boyer-Moore, the sliding window is combined with heuristics (e.g., bad character and good suffix rules) to skip portions of the text where matches are impossible.

Complexity:

- Best Case: O(n/m) if many characters are skipped.
- Worst Case: O(n+m) where the pattern is checked against every window (rare).

SLIDING WINDOW EFFICIENCY

- **Basic Sliding Window**: O(n·m) straightforward but inefficient for long patterns.
- **Optimized Sliding Window**: O(n)O(n)O(n) average case with hashing.
- **Heuristic Approaches**: O(n/m)O(n / m)O(n/m) best case when large skips occur.

For practical purposes, algorithms like **Rabin-Karp** or **Boyer-Moore** are preferred over naive sliding window methods due to their potential for skipping unnecessary comparisons or leveraging efficient hash computations.

Approach	Best Case	Worst Case
Basic Sliding Window	$O(n \cdot m)$	$O(n \cdot m)$
Sliding Window with Hashing	$O(n)$	$O(n \cdot m)$
Sliding Window with Heuristics	$O(n/m)$	$O(n + m)$

EXAMPLE 1: RABIN-KARP ALGORITHM

The Rabin-Karp algorithm uses a hashing approach to search for a pattern. It calculates the hash of the pattern and compares it with the hash of each substring of the text. If the hashes match, a comparison is performed.

How it Works:

- Hashing: Rabin-Karp calculates a hash value for the pattern and for substrings of the text. When the hash of a substring matches the hash of the pattern, it performs an exact comparison.
- Rolling Hash: To efficiently calculate the hash of the next substring, Rabin-Karp uses a rolling hash function, which allows updating the hash in constant time as we move through the text.

Time Complexity:

- Best case: O(n) when the hashes match and the comparisons are quick.
- Average case: O(n + m), assuming a good hash function and a reasonable number of hash collisions.
- Worst case: O(n * m), which occurs when every hash matches, and a full string comparison is required (this happens in case of hash collisions).

Space Complexity:

- O(m) for storing the hash values.

Advantages

- Efficient for Short Patterns: Performs well when searching for short patterns in long texts.
- Rolling Hash: Reduces the need to recompute hash values for overlapping substrings.
- Simple to Implement: The logic is straightforward and can be implemented without external libraries.

Limitations

- Hash Collisions: False positives due to hash collisions require verification of matches, which may reduce efficiency.
- Not Optimal for Large Patterns or Multiple Patterns: It becomes slower compared to algorithms like Aho-Corasick or Wu-Manber for these use cases.

EXAMPLE 2: WU-MANBER ALGORITHM

Wu-Manber algorithm is highly useful for finding words or patterns in texts, especially when searching for multiple patterns simultaneously. It is a string-matching algorithm optimized for high-speed searching and is particularly effective in scenarios where:

- Multiple Patterns: Necessary to search for a large set of patterns or words.
- Large Texts: The text to be searched is large, and efficiency is critical.
- Exact Matches: Wu-Manber is designed for exact string matching but can be adapted for approximate matching with small modifications

Advantages

- Efficiency: It reduces unnecessary comparisons, making it faster for large-scale searches.
- Multi-pattern Search: Handles multiple patterns simultaneously, unlike single-pattern algorithms like KMP.
- Scalable: Performs well even when dealing with large texts or a large number of patterns.

Limitations

- Exact Matching: By default, it is for exact matches, so it is not inherently suitable for approximate matching unless modified.
- Setup Overhead: Precomputing the shift and prefix tables can be time-consuming, especially for many patterns

TRIE MATCHING: MULTISTRING & SUFFIX TREES

Trie data structure

- A tree-like data structure used for storing a dynamic set of strings. It is commonly used for efficient retrieval and storage of keys in a large dataset.
- The structure supports operations such as insertion, search, and deletion of keys, making it a valuable tool in fields like computer science and information retrieval.

Suffix Tree

- A compressed trie for all suffixes of the given text.
- To build a suffix tree from given text:
  - 1) Generate all suffixes of given text
  - 2) Consider all suffixes as individual words and build a compressed trie

Problems where Suffix Trees provide optimal time complexity solution.

- 1) Pattern Searching
  2) Finding the longest repeated substring
  3) Finding the longest common substring
  4) Finding the longest palindrome in a string

Source: www.geeksforgeeks.com

TRIE ALGORITHMS

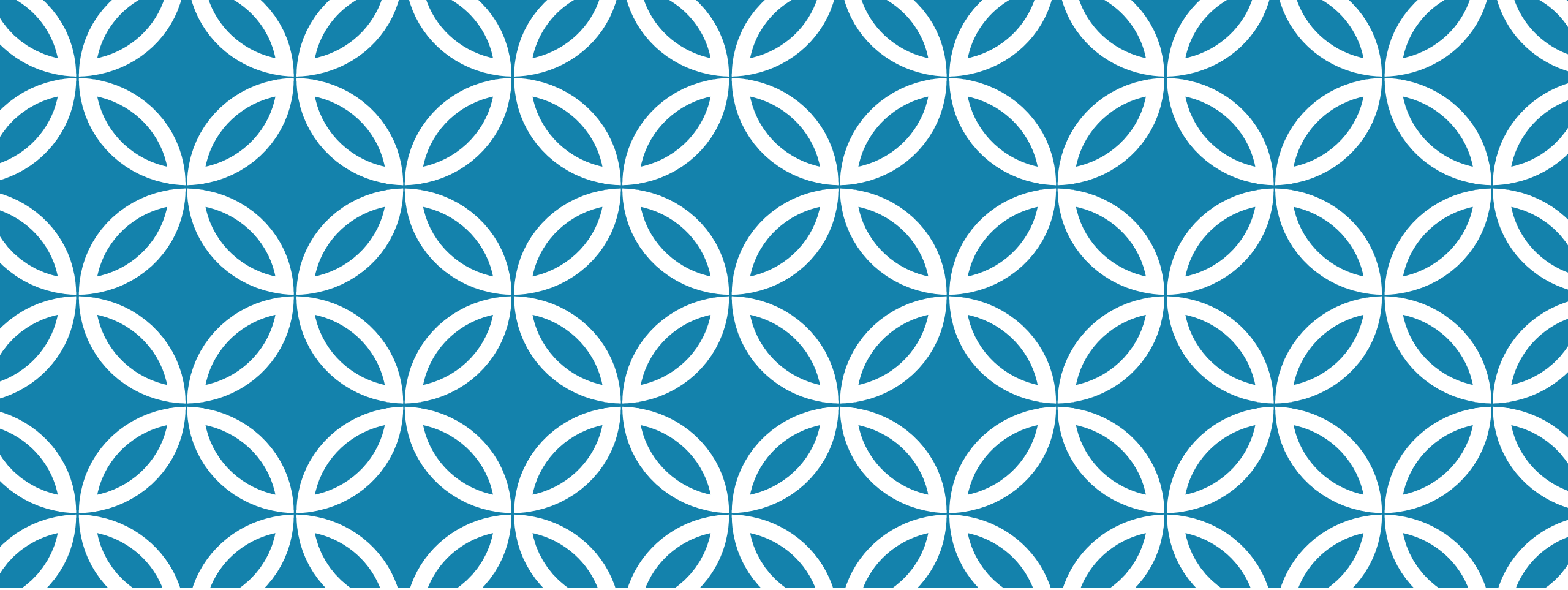
Example 1: Aho–Corasick algorithm

- dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") text. It matches all strings simultaneously.
- The complexity of the algorithm is linear in the length of the within an input strings plus the length of the searched text plus the number of output matches
- Time complexity. Aho-Corasick is considered linear O(m+n+k) where k is the number of matches

Example 2: Commentz-Walter algorithm

- A string searching algorithm that combines two known algorithms in order to attempt to better address the multi-pattern matching problem invented by Beate Commentz-Walter.
- Like the Aho–Corasick string matching algorithm, it can search for multiple patterns at once. It combines ideas from Aho–Corasick with the fast matching of the Boyer–Moore string-search algorithm.
- Developed by adding the shifts within the Boyer–Moore string-search algorithm to the Aho-Corasick, thus moving its complexity from linear to quadratic.
- Commentz-Walter may be considered quadratic O(mn) For a text of length n and maximum pattern length of m, its worst-case running time is O(mn), though the average case is often much better.

SECTION 2.
TECHNICAL CONCEPTS



TIME & SPACE COMPLEXITY OF STRING / PATTERN SEARCH

COMMON TIME COMPLEXITIES

1. **Constant Time (O(1)):**
    1. Runtime does not depend on the input size.
    2. Example: Accessing an element in an array.
2. **Logarithmic Time (O(logn):**
    1. Runtime grows logarithmically with input size.
    2. Example: Binary search in a sorted array.
3. **Linear Time (O(n)):**
    1. Runtime grows directly proportional to input size.
    2. Example: Iterating through an array.
4. **Linearithmic Time (O(nlogn)):**
    1. Runtime grows at nlognn \log nnlogn.
    2. Example: Efficient sorting algorithms like Merge Sort or Quick Sort.
5. **Quadratic Time (O(n2)):**
    1. Runtime grows quadratically with input size.
    2. Example: Nested loops over an array.
6. **Exponential Time (O(2n)):**
    1. Runtime doubles with each additional input.
    2. Example: Recursive solutions to the Traveling Salesman Problem (brute force).
7. **Factorial Time (O(n!)):**
    1. Runtime grows factorially with input size.
    2. Example: Brute force permutations of all elements in a set.

IMPORTANCE OF ASYMPTOTIC COMPLEXITY

Predicting Scalability:

Helps determine how an algorithm performs as input size grows.

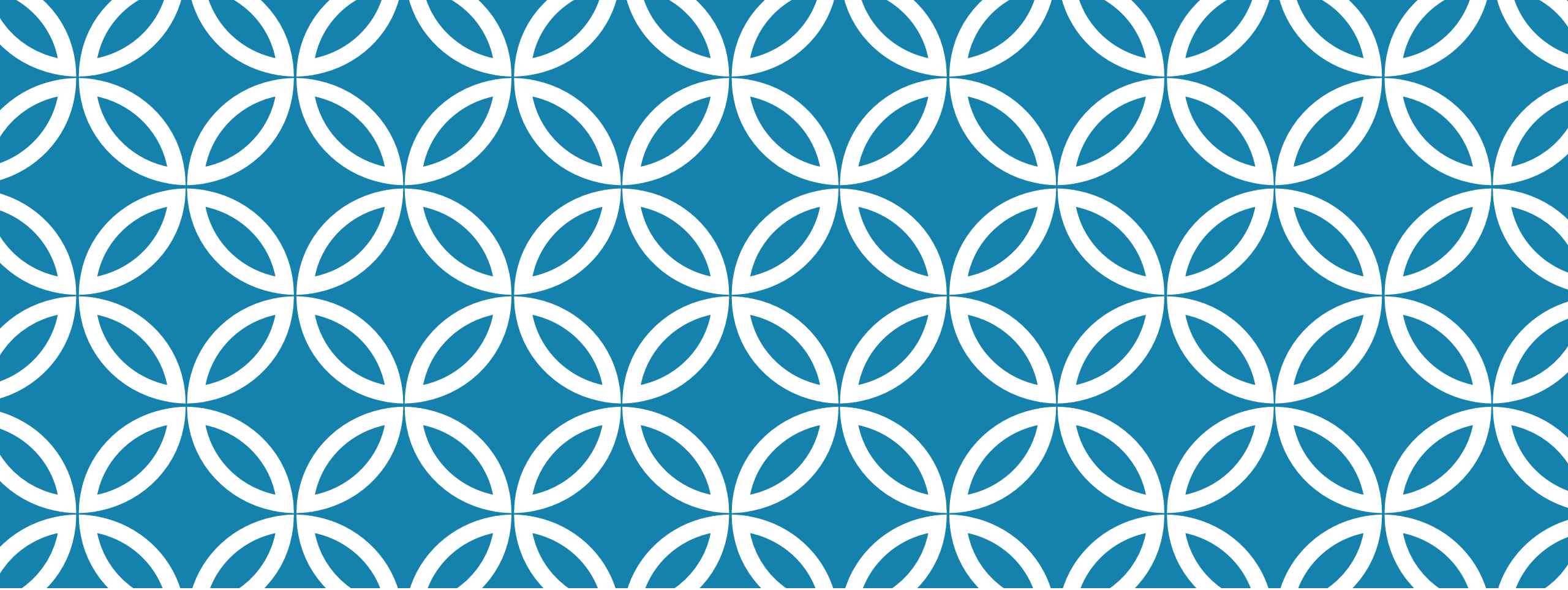
Comparing Algorithms:

Provides a theoretical basis to choose between algorithms for large datasets.

Resource Optimization:

Identifies inefficiencies in terms of time, enabling better design choices.

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)



PARADIGMS & ALGORITHMS

- **Exact String Matching Algorithms:** Optimized for precise matches.
- **Approximate String Matching Algorithms:** Handle errors or variations.
- **Multi-String Matching Algorithms:** Search for multiple patterns simultaneously.

POPULAR SEARCH ALGORITHMS

- Rabin-Karp Algorithm
- KMP Algorithm
- Z algorithm
- Finite Automata
- Boyer Moore – Bad Character Heuristic
- Aho-Corasick Algorithm
- Suffix Array
- Kasai's Algorithm for LCP array from Suffix Array
- Online algorithm for checking palindrome in a stream
- Manacher's Algorithm
- Ukkonen's Suffix Tree Construction – Part 1
- Generalized Suffix Tree

STRING SEARCHING ALGORITHMS

- String searching algorithms are an important class of string algorithms that try to find a place where one or several strings (i.e. patterns) are found within a larger string or text.
- Let Σ be an alphabet (finite set). Formally, both the pattern and searched text are concatenation of elements of Σ. The Σ may be usual human alphabet (A-Z).
- Other applications may use binary alphabet (Σ = {0, 1}) or DNA alphabet (Σ = {A, C, G, T}) in bioinformatics.

Table 1: String matching algorithm summary

Algorithm	Preprocessing time	Complicity matching time
Naïve string search algorithm	0 (no preprocessing)	O((n-m+1) m)
Trie-matching	0 (no preprocessing)	*O (m + #pat · n)*
Rabin-Karp string search algorithm	θ(m)	O((n-m+1) m)
Finite automata	O(m |Σ|)	θ(n)
Knuth-Morris-Pratt algorithm	θ(m)	θ(n)
Boyer-Moore string search algorithm	O(m)	average O(n/m), worst O(n m)

Source: Multiple Skip Multiple Pattern Matching Algorithm (MSMPMA), Qadi, Aqel, El Emary, (IJCS 2007) 2007

STRING SEARCH ALGORITHMS

Naive String Search:

Compares the search string character-by-character with every possible substring in the text.

Knuth-Morris-Pratt (KMP):

Uses a preprocessing step to create a partial match table to skip unnecessary comparisons.

Boyer-Moore:

Searches backwards and uses heuristics (bad character and good suffix rules) to skip sections of the text for faster searching.

Purpose:

- String search algorithms focus on finding exact matches of a specific string (a sequence of characters) within a larger text.

Nature:

- These algorithms treat the search term as a single entity and look for its occurrence(s) without accommodating variations or complex rules.

Common Use Cases:

- Searching for exact phrases in a document.
- Locating keywords in a text file.
- Text editors' "Find and Replace" functionality.

BEST PARADIGMS FOR STRING SEARCH

FOR STRING SEARCHING TASKS, THE FOLLOWING PARADIGMS AND ALGORITHMS ARE GENERALLY MORE EFFICIENT

1. Exact String Matching Algorithms

- Knuth-Morris-Pratt (KMP):
  - A linear-time string matching algorithm that avoids re-examining already matched characters.
- Boyer-Moore:
  - A heuristic-based algorithm that skips over sections of the text based on mismatched characters.
- Rabin-Karp:
  - Uses hashing to find possible matches efficiently, especially useful when searching for multiple patterns.
- Finite State Automata (FSA):
  - A deterministic method used in many text searching problems, particularly in regular expression matching.

2. Approximate String Matching Algorithms

- Levenshtein Distance (Edit Distance):
  - Measures the number of insertions, deletions, or substitutions needed to transform one string into another.
- Hamming Distance:
  - A simpler measure than Levenshtein, suitable for strings of equal length.
- Aho-Corasick Algorithm:
  - A multi-pattern matching algorithm that can efficiently find multiple patterns in text.

PATTERN SEARCH ALGORITHMS

- **Finite Automata for Pattern Matching:**

Constructs a state machine to recognize complex patterns.

- **Regular Expressions (Regex):**

Provides a powerful language for defining patterns with features like character sets, quantifiers, and anchors.

- **Aho-Corasick Algorithm:**

Efficient for searching multiple patterns simultaneously.

Purpose:

- Pattern search algorithms are designed to find occurrences of patterns (which may include wildcards, repetitions, or other rules) within a text. These patterns are often defined using regular expressions.

Nature:

- These algorithms handle more complex queries and are capable of identifying matches even when there are variations in the text (e.g., patterns like "c.t" matching "cat", "cut", or "cot").

Common Use Cases:

- Validating formats (e.g., email addresses, phone numbers).
- Finding substrings with specific characteristics (e.g., starts with "a" and ends with "z").
- Data extraction from semi-structured text (e.g., log files, HTML documents).

MULTI-PATTERN V. SINGLE PATTERN STRING MATCHING SOLUTIONS

- Multi-pattern string matching
  - searches an application file such as DNA sequence or a text file for a set of strings (patterns).
- Substring (single pattern) matching
  - people are familiar with solutions as Boyer-Moore
- Imprecise string matching algorithms
  - use hashing and signature based techniques

Preprocessing

- Single pattern string matching algorithm can be applied to multiple pattern string matching by applying the single pattern algorithm to the search text for each search pattern but does not scale well to larger sets of strings to be matched.
- Multi-pattern string matching algorithms generally preprocess the set of input strings, and then search all of them together over the body of text. Previous work in precise multi-pattern string matching includes Aho-orasick, Commentz Walter, Wu-Manber, and others.

CHARACTERISTICS OF SEARCH

Target Element/Key:

- An element or item within the data collection. This target could be a value, a record, a key, or any other data entity of interest.

Search Space:

- The entire collection of data for the target element. Depending on the data structure used, the search space may vary in size and organization.

Complexity:

- Searching can have different levels of complexity depending on the data structure and the algorithm used. The complexity is often measured in terms of time and space requirements.

Deterministic vs. Non-deterministic:

- Algorithms which follow a clear, systematic approach, like binary search, are deterministic. Others, such as linear search, are non-deterministic, as they may need to examine the entire search space in the worst case.

SEARCH PARADIGMS

Not all search paradigms are suitable for searching for strings in a given text, as the nature of the search problem influences which paradigms are most effective.

Searching for substrings or patterns in text is typically a string matching problem or a pattern matching problem, which may require different approaches depending on the type of problem and constraints (e.g., performance, complexity, and requirements for exact or approximate matches).

Paradigm	Key Strength	Key Weakness
Uninformed Search	Guarantees optimality in structured spaces	Inefficient without heuristics
Informed Search	Efficient with good heuristics	May be suboptimal with poor heuristics
Local Search	Works well for optimization	May get stuck in local optima
Constraint-Based Search	Ensures solutions satisfy rules	Can be computationally expensive
Adversarial Search	Models competitive scenarios	Computationally intensive in large spaces
Probabilistic Search	Handles uncertainty effectively	Requires probabilistic models
Online Search	Adapts to dynamic environments	May lack global optimization
Metaheuristic Search	Finds good-enough solutions in large spaces	No guarantees of optimality

SEARCH PARADIGMS

1. Uninformed Search

- Uninformed search paradigms like Depth-First Search (DFS) or Breadth-First Search (BFS) are generally not well-suited for searching strings in text.
  - These methods are designed for exploring state spaces, such as graphs or trees, and do not leverage the structure of text.
  - They are inefficient for string matching tasks because they do not exploit string patterns or indexing mechanisms.

2. Informed Search

- Informed search paradigms like A* or Greedy Best-First Search rely on heuristics to prioritize certain states. These approaches are generally not directly applicable to text searching, as string searching doesn't naturally have a heuristic to guide the search efficiently.
  - However, A* could be adapted in some advanced cases (e.g., searching for a specific substring in a large text corpus where the "cost" is the number of characters processed), but it's not typically used for direct string matching tasks.

3. Local Search

- Local search methods like Simulated Annealing, Genetic Algorithms, and Tabu Search are used for optimization problems where you're trying to find a solution in a large or complex space, but these approaches are typically not efficient for string matching tasks.
  - String matching doesn't involve optimizing over a large search space in the way that these techniques are designed for. Thus, local search methods would be inefficient and unnecessary.

4. Constraint-Based Search

- Constraint-Based Search approaches, like backtracking or constraint satisfaction problems (CSPs), are not typically used for string matching because they are better suited for problems where there are explicit rules or constraints to be satisfied (e.g., Sudoku or scheduling problems).
  - While certain string matching problems (like matching with wildcards or regular expressions) can be modeled as CSPs, it's still a more indirect approach compared to specialized algorithms for string matching.

SEARCH PARADIGMS

5. Adversarial Search

- Adversarial search (like the Minimax algorithm) is used in game theory and competitive environments (e.g., chess, Go), so it is not relevant for string matching in text. It does not provide any meaningful advantage when it comes to searching for substrings in a text.

6. Probabilistic Search

- Probabilistic search methods like Markov Decision Processes (MDPs) or Particle Filtering are designed for problems involving uncertainty and randomness. These are generally not suitable for exact string matching tasks where you're looking for precise substring matches.
- However, probabilistic methods can be useful in cases of approximate string matching or fuzzy matching, where you want to find substrings that are "close enough" to a given pattern (e.g., typo tolerance, noisy text).

7. Online Search

- Online search methods, where the search process occurs in real-time or dynamically as data is processed, could be applicable to some text search problems in streaming text or when you're dealing with real-time data. However, traditional string matching algorithms don't need to operate in such a manner.
- These methods might be useful for problems like searching in logs or real-time updates.

8. Metaheuristic Search

- Metaheuristic search methods like Ant Colony Optimization (ACO) or Swarm Intelligence are not designed for string searching and would not be an efficient or appropriate approach for standard string matching problems.

DETERMINISTIC V. NON-DETERMINISTIC SEARCH

Deterministic and non-deterministic search are two paradigms in computer science and artificial intelligence for finding solutions in a search space. Their primary difference lies in how they explore the search space and handle uncertainty.

Aspect	Deterministic Search	Non-Deterministic Search
Predictability	Always produces the same result	May produce different results
Randomness	None	Incorporates randomness
Exploration Strategy	Sequential and systematic	Randomized or parallel exploration
Efficiency	Depends on the search strategy	Often faster for large/complex problems
Use Case	Structured problems	Complex or stochastic problems

DETERMINISTIC V. NON-DETERMINISTIC SEARCH

Deterministic Search

- A deterministic search always follows a predefined set of rules to explore the search space. Given the same initial conditions and inputs, it will always produce the same result.

Characteristics:

1. **Predictability:**
    1. The search process is predictable and repeatable. Running the search multiple times with the same input will yield the same path and outcome.
2. **No Randomness:**
    1. It relies on strict logic and algorithms without incorporating randomness or external factors.
3. **Examples:**
    1. **Depth-First Search (DFS):** Explores a branch of the search space completely before backtracking.
    2. **Breadth-First Search (BFS):** Explores all nodes at one depth level before moving to the next.
    3. **Dijkstra's Algorithm:** Finds the shortest path in a weighted graph deterministically.
4. **Use Cases:**
    1. Problems with well-defined rules or environments, such as pathfinding in structured graphs, scheduling, or puzzles like Sudoku.

Non-Deterministic Search

- A non-deterministic search incorporates randomness or allows multiple possible actions at each step, potentially leading to different results for the same input. It is often used to simulate or approximate solutions in complex or uncertain environments.

Characteristics:

1. **Randomness/Multiple Choices:**
    1. It may involve random decisions or assume a system capable of exploring multiple options simultaneously.
2. **Exploration:**
    1. It can explore many branches or paths in parallel (conceptually), which can lead to faster discovery of solutions in some cases.
3. **Examples:**
    1. **Monte Carlo Search:** Uses random sampling to explore the search space.
    2. **Genetic Algorithms:** Use randomness in mutation and selection to evolve solutions.
    3. **Stochastic Search:** Incorporates randomness in decision-making at each step.
4. **Use Cases:**
    1. Problems with vast or poorly defined search spaces, such as optimization problems, game playing (like chess or Go), or real-world scenarios where uncertainty exists.

DFA V. NFA EFFICIENCY

1. Time Complexity

- DFA:
  - Time Complexity: The time complexity of a DFA is O(n), where n is the length of the input string. This is because, at each step, the DFA reads one symbol and moves to the next state. The transition for each symbol is a direct lookup in the transition table, making the process fast and linear.
  - Why Efficient: Every input symbol uniquely determines the next state, so the DFA processes each symbol in constant time.
- NFA:
  - Time Complexity: The time complexity of an NFA is $O(n * 2^m)$, where n is the length of the input string, and m is the number of states. This is because an NFA may need to explore multiple possible paths at each step. In the worst case, the NFA may need to consider all possible subsets of states at each position in the input string (since an NFA can be in multiple states at once).
  - Why Less Efficient: For every input symbol, the NFA may need to explore several transitions (since there can be multiple possible states for each symbol), leading to an exponential increase in computation. If we simulate an NFA directly without converting it into a DFA, it could become very inefficient.
  - Optimization: When using an NFA, one typical optimization is to simulate the NFA with backtracking or use breadth-first search (BFS) or depth-first search (DFS) to explore possible transitions. However, even with optimizations, it can be slower than a DFA for long input strings.

2. Space Complexity

- DFA:
  - Space Complexity: The space complexity of a DFA is O(m), where m is the number of states. This is because, in a DFA, there is a fixed number of states, and the transitions are stored in a table or a similar data structure. Each state requires storing a transition for each possible input symbol.
  - Why Efficient: Since there are no epsilon transitions (no empty symbol transitions), and every state is fixed, the space used by a DFA is linear in terms of the number of states.
- NFA:
  - Space Complexity: The space complexity of an NFA is also O(m), but it can be more complicated due to the fact that the NFA can be in multiple states at once. The space used during the simulation of an NFA may depend on how many states are reachable at any point during the computation. While an NFA itself might use similar space to a DFA, simulating the NFA during processing might require additional space to store the set of active states at each step.
  - Why Less Efficient (in simulation): During simulation, an NFA needs to track all possible states that it could be in after reading each symbol. In the worst case, this could be as many as 2^m states, leading to a space complexity of $O(2^m)$ during simulation, which is much larger than the DFA's linear space.

DFA V. NFA EFFICIENCY

3. Conversion Between NFA and DFA

- An important consideration when comparing DFA vs NFA is that an NFA can be converted into an equivalent DFA using subset construction (or powerset construction). This process involves creating a new DFA state for each subset of NFA states. The space and time complexity of this conversion are as follows:
- Time Complexity for NFA to DFA Conversion:
  - The conversion process takes $O(2^m)$ time, where m is the number of states in the NFA. This is because, in the worst case, the DFA may have to represent all subsets of the NFA states, which results in an exponential growth in the number of states.
- Space Complexity for NFA to DFA Conversion:
  - After conversion, the DFA could have up to 2^m states, which means that the space complexity of the resulting DFA can also be exponential in the number of NFA states.

5. Conclusion: Which is More Efficient?

- DFA is typically more efficient in both time and space when it comes to running the automaton against an input string. It processes each symbol in constant time (O(n)) and uses minimal space (O(m), where m is the number of states). However, the conversion of an NFA to a DFA can result in an exponential increase in the number of states.
- NFA can be more compact and easier to design for certain problems (e.g., regular expressions). But if you're simulating the NFA directly, it can be less efficient due to its exponential time complexity ($O(n * 2^m)$). The simulation of an NFA may require additional space for tracking multiple active states, and the process can be slow for large input sizes.
- Therefore, for large-scale string matching or regular language recognition tasks, DFA is generally preferred due to its better time and space efficiency during execution. NFA is useful for designing compact models but may be slower in practice unless you convert it to a DFA or use specialized optimizations.

CHARACTERISTICS OF A NAIVE STRING SEARCH

Why It's "Naive"?

The naive string search is the most straightforward way to perform string matching. It has the following properties:

- **Time Complexity**: O(n · m) for each search term, where ***n*** is the length of the text and ***m*** is the length of the search term.
- **Space Complexity**: Minimal, as no additional data structures are used beyond the regular expression engine.

In contrast, optimized algorithms like **Knuth-Morris-Pratt (KMP)**, **Boyer-Moore**, or **Rabin-Karp** preprocess the pattern or text to reduce the number of comparisons, achieving better performance for repeated or complex searches.

Why use for simple texts?

The naive string search approach is appropriate where:

1. The focus is on simplicity and flexibility, allowing regex patterns for searching.
2. The text size is likely manageable, and performance is not a critical constraint.
3. The program is designed for general-purpose analysis, not high-performance or large-scale string matching.

CHARACTERISTICS OF A NAIVE STRING SEARCH

Simple Pattern Matching:

- The program uses regular expressions (re.findall) to match a given word or pattern in the entire text. While regex adds some flexibility (e.g., case insensitivity, boundaries), it essentially scans the text sequentially to find matches.

No Optimization for Overlap:

- Naive algorithms do not attempt to optimize for overlapping patterns or partial matches. Each search term is independently matched against the text.

Full Text Scan:

- For each search term, the entire text is scanned, which can be computationally expensive for large inputs. The naive approach does not preprocess the text to create data structures (like a suffix tree or prefix table) for faster repeated searches.

Direct Search Logic:

- There's no sliding window or advanced search optimizations (like skipping unmatched regions). The program checks every occurrence of the search term.

SLIDING WINDOW APPROACH

The sliding window approach for pattern search typically involves examining a fixed-length portion (or "window") of the input text as it moves across the text. The Big O time complexity for this approach depends on the specific implementation and whether additional optimizations (e.g., hashing) are used.

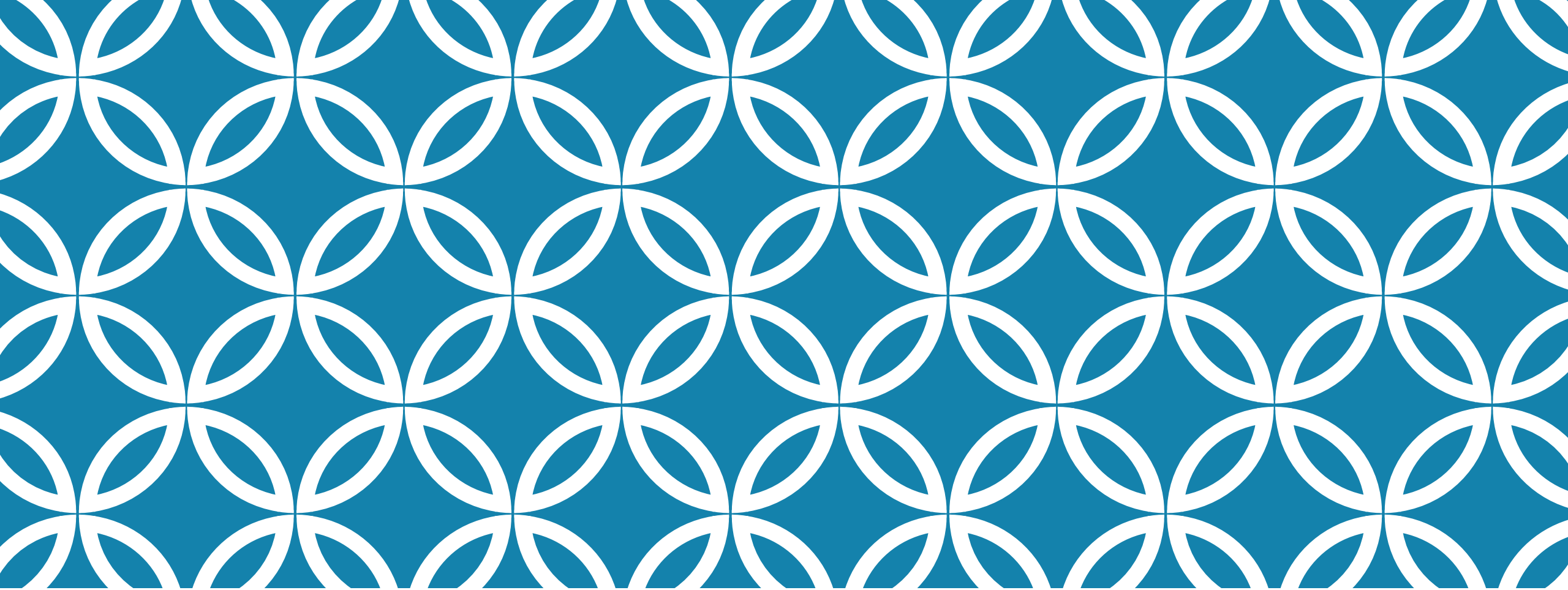
Basic Sliding Window Without Optimizations

- In a naive implementation, the algorithm checks every possible substring of length m (pattern length) within a string of length n, comparing each character in the window to the pattern.
- Time Complexity: For each of the n−m+1 windows, m comparisons are made (one for each character in the pattern).
- Total Complexity: O(n·m).

Use Case:

This complexity applies when you:

- Directly compare the pattern against the substring in the window.
- Perform no preprocessing or optimizations like hashing.



APPENDIX: SEARCH CONCEPTS

Concepts & Code Examples

1. BINARY SEARCH:

Binary search is an efficient algorithm for finding an item from a sorted list of items.

- It works by repeatedly dividing in half the portion of the list that could contain the item until you've narrowed down the possible locations to just one.

Steps:

- Compare the target value to the middle element of the array.
- If the target value is equal to the middle element, the search is complete.
- If the target value is less than the middle element, repeat the search on the left half.
- If the target value is greater, repeat on the right half.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        # Check if target is present at mid
        if arr[mid] == target:
            return mid

        # If target is greater, ignore the left half
        elif arr[mid] < target:
            left = mid + 1

        # If target is smaller, ignore the right half
        else:
            right = mid - 1

    # Target is not present in the array
    return -1

# Example usage
if __name__ == "__main__":
    # A sorted array for demonstration
    arr = [2, 3, 4, 10, 40]
    target = 10

    result = binary_search(arr, target)

    if result != -1:
        print(f"Element is present at index {result}")
    else:
        print("Element is not present in the array")
```

2. LINEAR SEARCH:

Linear search is the simplest search algorithm which checks every element in the list sequentially until the desired element is found or the list ends.

Steps:

- Start from the first element and compare it with the target value.
- Move to the next element and repeat the comparison.
- Continue until the target value is found or the end of the list is reached.

```
# Function to perform linear search
def linear_search(arr, target):
    # Iterate over all elements in the array
    for index, value in enumerate(arr):
        # Check if the current element matches the target
        if value == target:
            return index  # Return the index if found
    return -1  # Return -1 if the target is not found

# Example usage
array = [4, 2, 5, 1, 3]
target_value = 5
result = linear_search(array, target_value)

if result != -1:
    print(f"Element {target_value} found at index {result}.")
else:
    print(f"Element {target_value} not found in the array.")
```

3. DEPTH FIRST SEARCH (DFS):

DFS is an algorithm for traversing or searching tree or graph data structures.

- It starts at the root (or an arbitrary node) and explores as far as possible along each branch before backtracking.

Steps:

- Start at the root (or any arbitrary node) and push it onto a stack.
- Pop a node from the stack, mark it as visited, and push all its adjacent nodes that have not been visited.
- Repeat until the stack is empty.

3. DEPTH FIRST SEARCH (DFS):

DFS using Recursion:

```
# Function to perform DFS on a graph using recursion
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=' ')

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
start_node = 'A'
print("DFS traversal starting from node", start_node, ":")
dfs_recursive(graph, start_node)
```

DFS using Iteration:

```
# Function to perform DFS on a graph using iteration
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()

        if vertex not in visited:
            print(vertex, end=' ')
            visited.add(vertex)

            # Push all unvisited adjacent vertices to the stack
            for neighbor in reversed(graph[vertex]):
                if neighbor not in visited:
                    stack.append(neighbor)

# Example usage
print("\nDFS traversal starting from node", start_node, "using iteration:")
dfs_iterative(graph, start_node)
```

4. BREADTH FIRST SEARCH (BFS):

BFS is an algorithm for traversing or searching tree or graph data structures.

- It starts at the root (or an arbitrary node) and explores all neighbor nodes at the present depth prior to moving on to nodes at the next depth level.

Steps:

- Start at the root (or any arbitrary node) and enqueue it.
- Dequeue a node from the front of the queue, mark it as visited, and enqueue all its adjacent nodes that have not been visited.
- Repeat until the queue is empty.

```
from collections import deque

# Function to perform BFS on a graph
def bfs(graph, start):
    visited = set()  # Set to keep track of visited nodes
    queue = deque([start])  # Queue for BFS, starting with the start
node
    result = []  # List to store the order of traversal

    while queue:
        # Dequeue a vertex from the queue
        vertex = queue.popleft()

        # If the vertex has not been visited, mark it as visited
        if vertex not in visited:
            visited.add(vertex)
            result.append(vertex)

            # Enqueue all adjacent vertices that have not been visited
            for neighbor in graph[vertex]:
                if neighbor not in visited:
                    queue.append(neighbor)

    return result

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
start_node = 'A'
bfs_traversal = bfs(graph, start_node)
print("BFS traversal starting from node", start_node, ":",
bfs_traversal)
```

5. Z ALGORITHM

Description:

- The Z algorithm finds occurrences of a pattern in a string in linear time.
- It creates a Z array where each Z[i] represents the length of the longest substring starting from S[i] which is also a prefix of S.

Steps:

- Create a concatenated string "P$T" where P is the pattern, $ is a special character not present in either P or T, and T is the text.
- Construct the Z array for the concatenated string.
- Whenever Z[i] equals the length of the pattern, it indicates an occurrence of the pattern in the text.

```
def get_z_array(s):
    Z = [0] * len(s)
    L, R, K = 0, 0, 0
    for i in range(1, len(s)):
        if i > R:
            L, R = i, i
            while R < len(s) and s[R] == s[R - L]:
                R += 1
            Z[i] = R - L
            R -= 1
        else:
            K = i - L
            if Z[K] < R - i + 1:
                Z[i] = Z[K]
            else:
                L = i
                while R < len(s) and s[R] == s[R - L]:
                    R += 1
                Z[i] = R - L
                R -= 1
    return Z

def search(pattern, text):
    concat = pattern + "$" + text
    Z = get_z_array(concat)
    result = []
    for i in range(len(Z)):
        if Z[i] == len(pattern):
            result.append(i - len(pattern) - 1)
    return result
```

END SLIDE