Rabin–Karp String Matching (Java)

Student: Nurdan Zhanabayev

Algorithm Overview

My code implements **Rabin–Karp string matching** using **polynomial rolling hash**.

Workflow

1. **Guard Clause:** Handles empty strings, null input, or cases where the pattern is longer than the text.
2. **Preprocessing:**
    a. Compute the hash of the pattern using polynomial rolling hash:

$$hash(P) = \sum_{i=0}^{m-1} (P[i] - 'a' + 1) * P^i \bmod M$$

    b. Compute prefix hashes and powers of P for the text.
3. **Sliding Window Search:**
    a. For each substring of length m in the text, compute the rolling hash using prefix hashes.
    b. Compare it with the pattern hash.
    c. If hashes match, perform direct substring comparison to avoid collisions.
4. **Counting Operations:** The hashComparisons counter tracks how many hash comparisons were performed.

Key Design Choices

- **Large prime modulus** M = 1_000_000_009 reduces hash collisions.
- **Base** P = 31 provides good hash distribution for lowercase English letters.
- Guard clause ensures **robustness** for edge cases (empty text/pattern or pattern longer than text).

Description

This project implements the **Rabin–Karp string matching algorithm** using a **Polynomial Rolling Hash**.

It efficiently finds all occurrences of a pattern within a given text by comparing hash values instead of performing direct string comparisons.

Project Structure

File	Description
RabinKarp.java	Core algorithm implementation using rolling hash
Main.java	Demonstrates algorithm on short, medium, and long test strings
RabinKarpTest.java	Automated JUnit test suite covering edge and normal cases
report.md	Report summarizing design, testing, and complexity analysis
pom.xml	Maven build configuration

Build & Run

Prerequisites:

- Java 23
- Maven 3.8+

Build and run:

mvn compile

mvn exec:java -Dexec.mainClass="com.example.rabinkarp.Main"

Sample Execution Output

Test Case	Text Length	Pattern	Matches (Indices)	Time (ms)	Hash Comparisons
Short	10	abc	[0, 4, 7]	0.73175	8
Medium	18	aaab	[14]	0.01466	15
Long	32	aba	[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]	0.03375	30

Observations

- The number of hash comparisons grows **linearly** with text length.
- Execution time remains very low even for medium and long strings, demonstrating **efficiency**.
- The algorithm correctly identifies **overlapping matches** and handles **multiple occurrences**.

Complexity Analysis

Aspect	Analysis
Time Complexity (average)	O(n + m), where n = text length, m = pattern length. Hash calculation and rolling are linear.
Worst-case Time Complexity	O(n * m) in case of repeated hash collisions requiring substring comparison. Rare with a large prime M.
Space Complexity	O(n) for storing prefixHash and pow arrays. Constant space for pattern hash and counter.

Explanation:

- Preprocessing hashes: O(n)
- Pattern comparison on hash match: O(k * m), where k = number of hash matches
- For typical input with minimal collisions, the algorithm performs effectively in **linear time**.

JUnit Test Coverage

Category	Description	Example Test
Basic match	Standard substring search	testShortStringBasicMatch
Single match	One occurrence in medium text	testMediumStringSingleMatch
No match	Pattern not present	testNoMatch
Full text match	Pattern equals text	testPatternEqualsText

Overlapping matches	Repeated characters	testSingleCharacterRepeated, testOverlappingMatches
Edge cases	Empty text/pattern, pattern longer than text	testEmptyText, testEmptyPattern, testPatternLongerThanText
Case sensitivity	Detect lowercase vs uppercase	testCaseSensitivity
Performance	Long string efficiency	testLongStringPerformance
Boundary positions	Beginning/end matches	testPatternAtBeginning, testPatternAtEnd
Multiple matches	Multiple non-overlapping occurrences	testMultipleDistinctMatches
False positive protection	Random string	testRandomStringNoFalsePositives

Deep Insights from Results

1. **Efficiency**

The algorithm demonstrates **linear time performance** in practice. Execution times are extremely low:

- Less than 1 ms for short and medium strings
- ~0.034 ms for longer 32-character inputs

The **rolling hash approach** efficiently avoids repeated substring comparisons.

2. **Accuracy**

- **Overlapping patterns:** Correctly identifies repeated sequences, e.g., "aaaaa" with "aaa"
- **Multiple occurrences:** Finds all non-overlapping instances of the pattern
- **Boundary matches:** Accurately detects patterns at the start or end of the text

3. **Robust Edge Case Handling**

- **Empty text or pattern:** Returns no matches without errors
- **Pattern longer than text:** Correctly returns no matches
- **Case sensitivity:** Only exact character matches are considered

4. **Scalability**

Performance testing with large inputs, such as 10,000 repeated "abc" sequences, shows the algorithm **scales linearly** with text length.

5. **Predictable Operation Count**

The hashComparisons counter grows approximately **linearly** with text length, aligning with **O(n + m)** average time complexity expectations.

Example Hash Parameters

Parameter	Symbol	Value	Purpose
Base	P	31	Hash base (multiplier)
Modulus	M	1,000,000,009	Large prime to prevent overflow

Hash Formula	—	hash = Σ (s[i] - 'a' + 1) * P^i mod M	Polynomial rolling hash

Concluding Remarks

- The **Rabin–Karp algorithm** with polynomial rolling hash is highly **efficient and reliable** for substring search.
- **Prefix hashes**, **modular arithmetic**, and **guard clauses** ensure **correctness and robustness**, handling edge cases gracefully.
- Comprehensive **JUnit testing** validates overlapping matches, multiple occurrences, boundary matches, and large-scale performance.
- The **hash comparison counter** (hashComparisons) provides insight into predictable, near-linear scaling.

Overall: This implementation is **correct, efficient, scalable, and thoroughly tested**, demonstrating both **theoretical efficiency** and **practical reliability**.