RABIN-KARP FINGERPRINTING ALGORITHM

A Comprehensive Study of Efficient String Matching

Presented by :

Hari Shankar Gharai A125006

Krushna Kalyan Mohanty A125008

Under the guidance of :

Dr. Ajaya Kumar Dash

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,

IIIT BHUBANESWAR

Contents:

Introduction to String Matching:

What is String Matching?

String matching is the problem of finding occurrences of a pattern string within a larger text string. This fundamental problem appears in numerous applications across computer science.

Real-World Applications

Text Editors:	Search functionality (Ctrl + F)
Bioinformatics:	DNA sequence analysis
Information Retrieval:	Search engines
Plagiarism Detection:	Document comparison
Network Security:	Intrusion detection systems

Problem Definition:

Formal Statement:

Given:

Text T of length n: T[0..n-1]

Pattern P of length m: P[0..m-1]
where $m \leq n$

Find:

All valid shifts s ($0 \leq s \leq n-m$)
where P matches T[s..s+m-1]

Example:

T = "AABAACAADAABAABA" P = "AABA"

Valid shifts: 0, 9, 12

Complexity Goals

- Minimize the number of character comparisons.
- Achieve better than O(nm) worst-case time.
- Maintain reasonable space complexity.

Naive Approach:

Brute Force Algorithm

```
NaiveSearch(T, P):
    n = length of text T
    m = length of pattern P
    for s = 0 to (n - m):
        j = 0
        while (j < m) AND (T[s + j] == P[j]):
            j = j + 1
        if j == m :
            return s
    return -1
```

Complexity Analysis:

Case	Time Complexity	Example
Best Case	O(n)	First character always mismatches
Average Case	O(n + m)	Random text and pattern
Worst Case	O(nm)	T="AAAA...", P="AAA...B"

Rabin-Karp Algorithm Overview

Historical Context

Developed by **Michael O. Rabin** and **Richard M. Karp** in 1987, the algorithm uses hashing to achieve average-case linear time complexity for string matching.

Key Innovation: Fingerprinting

Core Idea: Use hash values as “fingerprints”

Instead of comparing strings character-by-character:

- Compute hash value h(P) for pattern
- Compute hash value h(T[s..s+m-1]) for each text window
- Compare hash values (O(1) operation)
- Verify match only when hash values match

Hash Function Fundamentals:

Polynomial Rolling Hash

For a string $S = s_0 s_1 s_2 \ldots s_{m-1}$, the hash value is:

$$h(S) = (s_0 \cdot d^{(m-1)} + s_1 \cdot d^{(m-2)} + \ldots + s_{m-1} \cdot d^0) \bmod q$$

where:
d = base (size of alphabet, typically 256)
q = large prime number (e.g., 101, 997)
s_i = numeric value of character at position i

Example:

Pattern P = "ABC" with d = 10, q = 13
A=1, B=2, C=3

$h(\text{"ABC"}) = (1\times10^2 + 2\times10^1 + 3\times10^0) \bmod 13$
$= (100 + 20 + 3) \bmod 13$
$= 123 \bmod 13 = \mathbf{6}$

Rolling Hash Mechanism:

Efficiency Trick:

Instead of recomputing the entire hash for each window, we **update** the previous hash value.

Recurrence Formula:

$h(T[s+1..s+m]) = (d \times (h(T[s..s+m-1]) - T[s] \times d^{(m-1)}) + T[s+m]) \bmod q$

Step-by-Step Transformation

Window 1: "ABC" → hash = h_1
Window 2: "BCD"
Rolling update:
1. Remove 'A': $h_1 - (A \times d^{(m-1)})$
2. Shift left: $(h_1 - A \times d^{(m-1)}) \times d$
3. Add 'D': previous + D
4. Apply mod q: h_2 = result mod q

Complexity Benefit

- Initial hash: O(m)
- Each rolling update: **O(1)**
- Total for all n-m+1 windows: O(n)

Algorithm Design & Structure:

Main Algorithm Flow:

Preprocess: Compute h(P)

↓

Compute h(T[0..m-1])

↓

For each position s in text

↓

Compare h(T[s..s+m-1]) with h(P)

↓

If hash matches:
Verify character-by-character

↓

Update hash using rolling formula

Pseudocode:

```
RabinKarp(T, P, d, q):
    n = length(T) , m = length(P)
    h_p = 0, h_t = 0 ,h=1
for i = 1 to m-1:          // Compute h = d^(m-1) mod q
        h = (h × d) mod q
for i = 0 to m-1:          // Compute initial hash values for pattern and
text
        h_p = (d × h_p + P[i]) mod q
        h_t = (d × h_t + T[i]) mod q
    for s = 0 to n - m:        // Slide pattern over the text
        if h_p == h_t:         // If hash values match, verify characters
            if P[0..m-1] == T[s..s+m-1]:
                print "Pattern found at index", s
        if s < n - m:        // Compute hash for next window (if exists)
            h_t = (d × (h_t - T[s] × h) + T[s + m]) mod q
if h_t < 0:           // Ensure the hash value is positive
                h_t = h_t + q
```

Example(Step by Step) :

Problem Setup: Text T = “31415926535”, Pattern P = “26”, d = 10, q = 11

Execution Trace:

Step	Window	Hash Calculation	Hash Value	Match?
0	P="26"	$(2\times10 + 6) \bmod 11$	4	Pattern
1	"31"	$(3\times10 + 1) \bmod 11$	9	No
2	"14"	$10\times(9-3\times10)+4 \bmod 11$	2	No
3	"41"	$10\times(2-1\times10)+1 \bmod 11$	3	No
4	"15"	$10\times(3-4\times10)+5 \bmod 11$	5	No
5	"59"→"26"	...	4	**Yes!**

Mathematical Foundation:

Why Modulo Operation ? The Overflow Problem

What happens Without Modulo

Example: Pattern "HELLO" (length 5), d=256 (ASCII)

hash = $H \times 256^4 + E \times 256^3 + L \times 256^2 + L \times 256^1 + O \times 256^0$

$= 72 \times 4{,}294{,}967{,}296 + 69 \times 16{,}777{,}216 + 76 \times 65{,}536 + 76 \times 256 + 79$

= **309,237,645,391** (11-digit number!)

The Problems

Issue	Without Modulo	With Modulo ($q=10^9+7$)
Max Value (m=10)	$\approx 10^{25}$ (25 digits!)	$\leq 10^9$ (9 digits)
Memory	Need BigInteger	Standard 32-bit int
Speed	Slow (multi-word arithmetic)	Fast (single CPU instruction)
Overflow	❌ CRASHES	✅ Controlled

Key Insight: Modulo keeps numbers small & manageable, allowing O(1) arithmetic, preventing overflow

Mathematical Foundation:

Why Use Prime Modulus?

- **Better distribution:** Prime q reduces clustering of hash values
- **Lower collision rate:** Fewer spurious hits

Hash Collision Probability

For random strings and large prime q:

P(collision) ≈ 1/q

Example: If $q = 10^9 + 7$, collision probability $\approx 10^{(-9)}$

Mathematical Foundation:

Why Modulo Cause Collision ?

Different inputs...
"AB" → hash = 28
"CD" → hash = 41
"XY" → hash = 15

All different numbers!

Same remainder!
28 mod 13 = 2
41 mod 13 = 2
15 mod 13 = 2

COLLISION!

Collision Probability

For random strings with prime modulus q:

P(collision) ≈ 1/q

q = 101 → ~1% chance
q = 1,000,000,007 → ~0.0000001% chance

Mathematical Foundation:

Spurious Hits: Example Setup: Text: “31415926535”, Pattern: “26”
$d = 10$, $q = 11$

Step-by-Step Execution

Pattern hash:

h(“26”) = $(2\times10 + 6) \bmod 11 = 26 \bmod 11 = 4$

Window 1: “31”

h(“31”) = $31 \bmod 11 = 9 \neq 4 \rightarrow$ Skip

Window 2: “14”

h(“14”) = $14 \bmod 11 = 3 \neq 4 \rightarrow$ Skip

Window 3: “41”

h(“41”) = $41 \bmod 11 = 8 \neq 4 \rightarrow$ Skip

Window 4: “15”

h(“15”) = $15 \bmod 11 = 4 = 4 \rightarrow$ **HASH MATCH!**

Verify: “15” vs “26” → **DIFFERENT!**

This is a SPURIOUS HIT

Window 5: “59”

h(“59”) = $59 \bmod 11 = 4 = 4 \rightarrow$ **HASH MATCH!**

Verify: “59” vs “26” → **DIFFERENT!**

Another SPURIOUS HIT

Window 6: “92”

h(“92”) = $92 \bmod 11 = 4 = 4 \rightarrow$ **HASH MATCH!**

Verify: “92” vs “26” → **DIFFERENT!**

Yet another SPURIOUS HIT

Window 7: “26”

h(“26”) = $26 \bmod 11 = 4 = 4 \rightarrow$ **HASH MATCH!**

Verify: “26” vs “26” → **MATCH FOUND**

Mathematical Foundation:

Why Do Spurious Hits Occur?

The Pigeonhole Principle
If you have **more items than containers**, some containers must hold multiple items

Possible Strings:
For 2 digits, base 10:
"00", "01", "02", ..., "99"
= **100 possibilities**

Hash Values (mod 11):
Can only be:
0, 1, 2, 3, 4, 5,
6, 7, 8, 9, 10
= **11 buckets**

Collision is Guaranteed!
100 strings must fit into 11 buckets
By pigeonhole principle: at least one bucket has ≥9 strings!

This means at least 9 different strings will have the same hash value.

Strings that Hash to 4 (mod 11)
"15" → 15 mod 11 = 4
"26" → 26 mod 11 = 4
"37" → 37 mod 11 = 4
"48" → 48 mod 11 = 4
"59" → 59 mod 11 = 4
"70" → 70 mod 11 = 4
.....
All these collide!

Mathematical Foundation:

Choosing Parameters to Minimize Spurious Hits

Impact of q (Modulo) on Collisions:

q value	Collision Probability	Expected Spurious Hits (searching 1M positions)	Verdict
11	~9%	~90,909	❌ Terrible
101	~1%	~9,901	⚠️ Poor
997	~0.1%	~1,003	⚠️ Okay
1,000,003	~0.0001%	~1	✅ Good
1,000,000,007	~0.0000001%	~0.001	✅ Excellent

Why Use Prime Numbers?

Better distribution: Prime q spreads hash values more uniformly
Mathematical properties: Avoids patterns in collisions
Standard practice: Common choices: 10^9+7, 10^9+9, $2^{31}-1$

Practical Recommendation

For most applications: **q = 1,000,000,007**
Large enough to minimize collisions
Small enough to fit in 32-bit integer
($2^{31}-1 = 2,147,483,647$)

Performance Analysis:

Best Case:
$O(n + m)$

Scenario: No hash collisions, or first character always differs
Example: T = “AAAAA”, P = “B”
Behavior: Quick hash comparison, no verification needed

Average Case:
$O(n + m)$

Scenario: Random text with good hash function
Expected matches: Small constant
Expected collisions: $\approx (n-m)/q \approx 0$ for large q
Behavior: Linear scanning with rare verifications

Worst Case:
$O(nm)$

Scenario: Hash collides at every position
Example: T = “AAAA...A”, P = “AAA...A” with poor q
Behaviour: Degenerates to naive algorithm
Mitigation: Choose large prime q, use multiple hashes

Performance Analysis:

Time Complexity:

Phase	Operation	Time Complexity
Preprocessing	Compute h = d^(m-1) mod q	O(m)
Initial Hash	Compute h(P) and h(T[0..m-1])	O(m)
Sliding Window	n-m+1 hash updates	O(n-m)
Verification	Character comparison on match	O(m) per match

Overall Time Complexity

Expected Case: O(n + m)
Assuming constant expected matches and low collision rate

Multiple pattern matching: O(n + km) for k pattern.

Worst Case: O(nm)
When hash collides at every position (requires verification each time)

With proper choice of q, expected case dominates

Performance Analysis:

Space Complexity:

Component	Space	Description
Input Storage	O(n + m)	Text and pattern strings
Hash Values	O(1)	h_p, h_t, h (constant variables)
Loop Variables	O(1)	Indices and counters
Total	**O(n + m)**	Linear space

Space-Efficient Characteristics

- **No auxiliary data structures:** Unlike KMP's failure function or Boyer-Moore's skip tables
- **In-place processing:** Only stores current hash values
- **Scalability:** Memory usage grows linearly with input size

Comparison with Other Algorithms

All major string matching algorithms use O(n + m) space,
Rabin-Karp has simpler memory management

Comparative Algorithm Analysis:

Algorithm	Preprocessing	Best Case	Average Case	Worst Case
Naive	O(1)	O(n)	O(n+m)	O(nm)
Rabin-Karp	O(m)	O(n+m)	O(n+m)	O(nm)
KMP	O(m)	O(n)	O(n)	O(n+m)
Boyer-Moore	O(m+σ)	O(n/m)	O(n)	O(nm)

σ = alphabet size

When to Choose Rabin-Karp

- Multiple pattern matching (k patterns)
- 2D or higher dimensional pattern matching(Ex : Image processing)

Applications:

1. Multiple Pattern Matching

Search for k different patterns simultaneously
Compute hash for each pattern: O(km)
Compare text window hash against all pattern hashes: O(k) per window
Total time: O(n + km) - much better than k separate searches

2. 2D Pattern Matching

Find 2D pattern in 2D text (image processing)
Apply rolling hash in both dimensions
Used in image recognition and computer vision

3. DNA Sequence Analysis

Find specific gene sequences in genome
Base size d = 4 (A, C, G, T)[4 Nucleotides]
Handle approximate matching with modifications

Practical Use Cases

Software Development

- Version control systems
- Code duplicate detection

Data Mining

- Pattern discovery in data streams
- Duplicate record detection

Bioinformatics

- Genome assembly
- Sequence alignment processing
- Protein structure analysis

Industry Standard:

- Rabin-Karp variants used in rsync,
- Google's internal tools,
- Bioinformatics packages

Advantages and Limitations:

Advantages

- **Simplicity:** Easy to understand and implement
- **Flexibility:** Naturally extends to multiple patterns
- **Versatility:** Works in multiple dimensions
- **Space efficient:** $O(1)$ auxiliary space
- **Average case:** Excellent $O(n+m)$ performance
- **Parallelizable:** Multiple patterns can be checked in parallel

Limitations

- **Worst case:** $O(nm)$ time complexity
- **Spurious hits:** Requires verification step
- **Hash quality:** Performance depends on good hash function
- **Overflow concerns:** Need careful modular arithmetic
- **Not optimal:** KMP guarantees better worst case $O(n+m)$.
- **Parameter tuning:** Choosing d and q matters

Implementation Considerations:

1. Choosing Parameters

Base (d): Typically: $d = 256$ (ASCII) or $d = 10$ (decimal) Should match alphabet size.

Prime (q): Large prime to minimize collisions .
Small enough to avoid overflow.

2. Handling Overflow

- Use modular arithmetic at every step.
- For subtraction: add q if result is negative.
- Consider using 64-bit integers for large alphabets.

3. Optimization Techniques

- **Precompute powers:** Store d^i mod q values.
- **Multiple hashes:** Use 2+ hash functions to reduce false positives.
- **Early termination:** Stop on first match if only one needed.
- **Bloom filters:** Combine with Bloom filter for pre-screening.

Conclusion:

In Conclusion we can say that practical algorithms often involve trade-offs. By accepting rare, verifiable errors (spurious hits), we gain simplicity, versatility, and excellent average-case performance. The algorithm's elegance lies not in avoiding all edge cases, but in handling them efficiently.

References:

[1] Karp, R. M., & Rabin, M. O. (1987). *"Efficient randomized pattern-matching algorithms."* IBM Journal of Research and Development, 31(2), 249-260.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *"Introduction to Algorithms"* (4th ed.). MIT Press. Chapter 32: String Matching.

[3] Knuth, D. E. (1997). *"The Art of Computer Programming, Volume 3: Sorting and Searching"* (2nd ed.). Addison-Wesley.

[4] Sedgewick, R., & Wayne, K. (2011). *"Algorithms"* (4th ed.). Addison-Wesley. Section 5.3: Substring Search.

[5] Gusfield, D. (1997). *"Algorithms on Strings, Trees, and Sequences."* Cambridge University Press.

[6] Navarro, G., & Raffinot, M. (2002). *"Flexible Pattern Matching in Strings."* Cambridge University Press.

Thank You!