Executive Summary
Rabin-Karp String Matching Algorithm

Henry Chen, Amogh Ghadge, Yuyan Wang

April 21, 2025

1 Introduction / Problem Statement

String matching is a fundamental problem in computer science, with broad applications in text processing, search engines, plagiarism detection, and so on. The core challenge is to efficiently find all occurrences of a pattern (or multiple patterns) within a larger text.

One key limitation in many traditional approaches is their potential for slow performance, especially in the face of multiple patterns or large texts. Rabin-Karp's strength lies in its efficiency in finding multiple occurrences of a pattern, and its potential for faster performance on large texts.

The Rabin-Karp algorithm addresses this problem by introducing a hashing-based approach to string matching. Unlike the naive method—which checks for matches by comparing the pattern to every possible substring of the text character by character, resulting in a runtime of $O(nm)$ for a text of length n and a pattern of length m—Rabin-Karp improves average-case performance by converting strings into hash values. This allows for constant-time comparison of substrings, reducing unnecessary character comparisons and achieving an average-case runtime of $O(n + m)$ when using a good hash function.

However, there are key challenges that must be addressed. The primary issue is the risk of spurious hits—cases where the hash of a substring matches the hash of the pattern, but the actual strings are not equal. These false positives occur due to hash collisions and require an additional step of exact string comparison to verify the match, which can affect performance if matches are frequent. The worst-case time complexity of the Rabin-Karp algorithm is $O(nm)$, where n is the length of the text and m is the length of the pattern, and occurs when the pattern matches every equal length substring of the text.

2 Overall Intuitive Approach / Solution

At a high level, the key idea of the Rabin-Karp algorithm is to represent strings—both the pattern and substrings of the text—as numerical hash values. This way, the algorithm can

quickly filter out most of the non-matching substrings by simply comparing their hash values with the hash of the pattern.

To do this, Rabin-Karp first computes a hash value for the pattern string. Then, it slides a window of the same length over the text, one character at a time, and computes the hash value of the current substring in the window. If the hash values do not match, the substring is definitely not a match, and the algorithm moves on. If the hash values do match, there may be a match, so the algorithm performs a direct string comparison to confirm because collisions could occur.

This process allows the algorithm to avoid many unnecessary character comparisons. Additionally, with the use of the rolling hash technique, the algorithm can efficiently update the hash value as the window moves, rather than recalculating it from scratch.

3 Implementation

The Rabin-Karp algorithm relies on two main ideas: computing a hash value for strings and efficiently updating that hash as the search window slides across the text. The core implementation revolves around representing strings as numerical values and comparing those values to identify potential matches.

a. Hashing Strings

To compare strings numerically, each character is treated as a digit in a number using a chosen base d (e.g., 256 for ASCII characters). A string of length m is then hashed using a polynomial hash function of the form:

$$hash(S) = (u_{S[0]} \cdot d^{m-1} + u_{S[1]} d^{m-2} + ... + u_{S[m-1]} \cdot d^0) \bmod q$$

Where

S: is the substring with length m (length of pattern)

$u_{S[i]}$: is the Unicode of character at $S[i]$

d: is the base

q: is a large prime number to integer overflow and collisions

b. Rolling Hash

Instead of recalculating the hash from scratch for every substring, the algorithm uses a rolling hash to update the hash value in constant time as the window slides forward by one character:

$$hash = (d \cdot (hash - u[\text{character to be removed}] \cdot h) + u[\text{character to be added}]) \bmod q$$

Where

d: is the base

$u[i]$: is the Unicode of character i

$h = d^{m-1} \bmod q$ is the highest order coefficient

q: is a large prime number

c. Rabin-Karp Algorithm Python Code Implementation (Python)

```
# find all indices in the text that match the pattern
def rabin_karp(text, pattern):
    indices, base, q, target_hash, curr_hash, l = [], 26, 10007, 0, 0, 0

    # Compute the hash of the pattern
    for i in range(len(pattern)):
        # Subtract ord("a") to normalize the ASCII value
        target_hash = (target_hash * base + (ord(pattern[i]) - ord("a") + 1)) % q

    # Sliding window
    for r in range(len(text)):
        curr_hash = (curr_hash * base + (ord(text[r]) - ord("a") + 1)) % q

        # Rolling hash update
        if r - l + 1 > len(pattern):
            curr_hash = (curr_hash - (ord(text[l]) - ord("a") + 1) * pow(base, len(pattern),
                q)) % q
            l += 1

        # Check if the current window matches the pattern
        if r - l + 1 == len(pattern) and curr_hash == target_hash:
            if text[l:r+1] == pattern:
                indices.append(l)

    return indices
```

The base here is 26 which represents all lowercase letters. However, it is easy to extend the base to all characters and the normalization step can be removed from the code if it is unnecessary.

4 Summary of Programming Challenge

Our programming challenge asks students to compute the number of palindromic substrings in a given string of lower case letters using hashing for efficient substring comparison. The learning objective is for students to understand how to extend and adapt classical string matching algorithms (like Rabin-Karp) to non-traditional problems. The high-level twist in this assignment is that instead of looking for a pattern within another string, we use Rabin-Karp hashing to compare substrings from the original and reversed strings. This means that students will need to apply the hash techniques both by adding the characters to the most significant position and the least significant position at the same time to their respective hash, allowing for constant-time palindrome detection over all substrings.
Unlike typical applications of Rabin-Karp that involve matching one pattern against a text, this challenge forces students to maintain two different growing hashes and add to them from

the most significant side and least significant side. This ensures that the student understands the original rolling hash algorithm and how to adjust it accordingly for different use cases.

5 Key Ideas for Solving Programming Challenge

The main idea for solving the challenge is to utilize growing hashes for substrings of both the original and reversed strings to detect palindromes in constant time. The critical observation is that a substring $s[i...j]$ is a palindrome if it matches its reverse, which corresponds to $s[n-j...n-i]$.

To compare substrings without repeatedly reversing or slicing them, you need to implement a variation of the Rabin-Karp rolling hash. For each starting index i in the string, you iteratively build the hash for s[i...j] and simultaneously construct the hash of its reversed counterpart using the pre-reversed string r. For the original hash, each new character is appended to the end, which involves multiplying the current hash by a base and adding the new character's value. For the reverse hash, the new character represents a higher-order term and is added using $base^{(j-i)}$.

Whenever the two hash values match, you need perform a direct comparison of the substring against its reverse to confirm it is a true palindrome and avoid false positives due to hash collisions. This combined approach of rolling hashes and manual verification enables a balance between performance and correctness. The final answer accumulates the count of all valid palindromic substrings found this way.

Key ideas include:

- Identify that a palindromes will have the same hash
- Use hash for O(1) substring comparison
- Update the original hash by adding characters to the least significant side, and the reversed hash by adding to the most significant side.
- Use direct comparison to confirm matches and avoid false positives

These ideas are interesting but not overly complex. The most difficult realization is that palindromes will have the same hash, however, all other key ideas fall into place once this realization is made. The complexity of the solution is $O(n^2)$, because we are checking all possible substrings. Compared to a naive $O(n^3)$ solution (which compares each substring character by character), this solution dramatically reduces inner-loop cost using hash comparisons in $O(1)$ time.

6 Conclusion

Pattern recognition in strings is a fundamental problem in computer science. Traditional approaches, such as the naive method of comparing every substring character by character, are often too slow, especially for large texts.

The Rabin-Karp algorithm improves on this by using a rolling hash function to convert substrings and patterns into numerical values, enabling constant-time comparisons. This significantly reduces unnecessary character checks and improves efficiency.

In this project, we extended the Rabin-Karp approach to solve a new problem: counting all palindromic substrings in a string. Unlike standard pattern matching, this required checking whether each substring is equal to its reverse. This requires maintaining two rolling hashes, one for the original string and one for its reverse, updating one from the least significant side and the other from the most significant. When the hashes matched, perform a final string comparison to confirm the palindrome.

This adaptation highlights the flexibility of Rabin-Karp. By understanding the core idea behind rolling hashes and rethinking how they can be applied, we were able to develop an elegant and efficient solution, reducing runtime from cubic to quadratic.