Theory: Rabin-Karp algorithm in Java

2 hours 0 / 5 problems solved

The Rabing-Karp algorithm is one more approach to solving the substring searching problem. The algorithm is similar to the naive approach but utilizes hashing for substring comparison, which allows reducing the total time complexity. In this topic, we will learn how to implement this algorithm in Java.

§1. Implementing the Rabin-Karp algorithm in Java

In our implementation, we will use the polynomial hash function for string hashing. For a string $s = s_0 s_1 ... s_{n-1}$ and constants a and m, it is defined as follows:

$$h_P(s) = \left(s_0 \cdot a^0 + s_1 \cdot a^1 + ... + s_{n-1} \cdot a^{n-1}\right) \ mod\, m.$$

Using this function, the Rabin Karp algorithm can be implemented as follows:

Current topic:

Rabin-Karp algorithm in Java

Topic depends on:

× Rabin-Karp algorithm

× Searching a substring in Java

Table of contents:

↑ Rabin-Karp algorithm in Java

§1. Implementing the Rabin-Karp algorithm in Java

§2. Examples

§3. Summary

Feedback & Comments

```
/* 1 */
public static long charToLong(char ch) {
    return (long)(ch - 'A' + 1);
}

public static List<Integer> RabinKarp(String text, String pattern) {
    /* 2 */
    int a = 53;
    long m = 1_000_000_000 + 9;



    /* 3 */

    long patternHash = 0;

    long currSubstrHash = 0;

    long pow = 1;



    for (int i = 0; i < pattern.length(); i++) {

        patternHash += charToLong(pattern.charAt(i)) * pow;

        patternHash %= m;



currSubstrHash += charToLong(text.charAt(text.length() - pattern.length()
+ i)) * pow;

        currSubstrHash %= m;



        if (i != pattern.length() - 1) {

            pow = pow * a % m;

        }

    }



    /* 4 */

    ArrayList<Integer> occurrences = new ArrayList<>();



    for (int i = text.length(); i >= pattern.length(); i--) {

        if (patternHash == currSubstrHash) {

            boolean patternIsFound = true;



            for (int j = 0; j < pattern.length(); j++) {

                if (text.charAt(i - pattern.length() + j) != pattern.charAt(j)) {

                    patternIsFound = false;

                    break;

                }

            }



            if (patternIsFound) {
```

```
                    occurrences.add(i - pattern.length());

                }

            }



            if (i > pattern.length()) {

                /* 5 */


            currSubstrHash = (currSubstrHash - charToLong(text.charAt(i - 1)) * po
w % m + m) * a % m;

            currSubstrHash = (currSubstrHash + charToLong(text.charAt(i - pattern.
length() - 1))) % m;

            }

        }



        Collections.reverse(occurrences);

        return occurrences;

    }
```

The `RabinKarp` method takes two strings, a pattern and a text, as arguments and returns a list of all occurrences of the pattern in the text. Here we assume that the length of the text is no less than the length of the pattern.

Now let's go through each step of the algorithm in detail.

1. In this implementation, we use a method `charToLong` to associate each symbol with some number. For example, for upper-case letters, it returns the sequence number of a letter in the alphabet, for lower-case letters it returns the sequence number plus 32.
2. Recall that for the Rabin-Karp algorithm we need to choose constants a and m. In this implementation, the constants are equal 53 and $10^9 + 9$ respectively.
3. First, we need to calculate a hash value for the pattern and for the first substring of the text using the formula of the polynomial hash directly. We perform it simultaneously in the `for` loop. Also, we store the current power of a in a variable `pow`. After the last multiplication, it is equal to $a^{|p|-1}$ and the value will be used in further computations.
4. Here we create a list to store occurrences of the pattern. Then, we move along the text from the right to the left calculating and comparing the hash values of the pattern and the current substring. If they are equal, we perform a symbol-by-symbol comparison. If the strings are indeed equal, we add the index of the current substring to the list of all occurrences. At the end of the `for` loop, we update the hash value for the current substring. After the loop is finished, we return the list of all occurrences as a final result.

Note that when calculating a hash value for the next substring (comment 5) we add m to the difference. Since a hash value for the pattern is a non-negative number, hash values for all substrings should be non-negative as well. This addition is done to avoid the processing of negative values.

§2. Examples

Below you can see an example of how to use the described method:

```
List<Integer> occurrences = RabinKarp("ABACABAD", "ABA");
System.out.println(occurrences); // [0, 4]
```

Here is one more example:

```
List<Integer> occurrences = RabinKarp("AAAA", "AA");
System.out.println(occurrences); // [0, 1, 2]
```

§3. Summary

In this topic, we have learned how the Rabin Karp algorithm — a hashing-based approach to the substring searching problem — can be implemented in Java. Note that the hashing technique we have covered here works not only for the Rabin Karp algorithm but can be applied in other problems that require string comparison. Thus, knowing this technique can help you to efficiently solve some other problems connected with strings.

Report a typo

21 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (0) Useful links (0) Show discussion