Theory: Rabin-Karp algorithm

27 minutes 0 / 5 problems solved Skip this topic Start practicing

247 users solved this topic. Latest completion was 2 days ago.

For a pattern p and a text t, the simplest substring searching algorithm performs a symbol by symbol comparison of the pattern with each substring of the text, which results in $O(|p| \cdot |t|)$ running time. If you work with large texts and performance is crucial, this might be too slow. To find a substring faster, we need to use more efficient approaches than the naive direct comparison.

One of the algorithms that help search a substring faster is the **Rabin-Karp algorithm**. It uses string hashing for a faster comparison thus significantly reducing the total running time compared with the naive approach.

Current topic:

Rabin-Karp algorithm

Topic depends on:

✓ Searching a substring

✓ String hashing

Topic is required for:

Rabin-Karp algorithm in Java

Table of contents:

§1. Algorithm description

The main idea of the Rabin-Karp algorithm is to use **string hashing** and compare strings' **hash values** instead of a direct comparison. To compute the substrings' hash values fast, the algorithm uses the polynomial hash functions and their rolling hash property. In more detail, the algorithm can be formulated as follows:

1. Calculate the hash value for a pattern.
2. Moving along the text (from right to left), calculate the hash value for the current substring of the text using the rolling hash property.
3. If hash values for the pattern and the current substring are not equal, move to the next substring. Otherwise, perform a symbol-by-symbol comparison of the strings. If the strings are indeed equal, add the current index to a list of occurrences.
4. Repeat steps 2-3 until all substrings of the text are processed. Then, return a list of all occurrences.

§2. An example

Consider how the algorithm works for a pattern $ACDC$ and a text $BACDCCBA$. For simplicity, we will use the following constants for the polynomial hash function: $a = 3$ and $m = 11$. First, we need to calculate the hash for the pattern:

$$h_P(ACDC) = \left(1 \cdot 3^0 + 3 \cdot 3^1 + 4 \cdot 3^2 + 3 \cdot 3^3\right)\ mod\ 11 = 127\ mod\ 11 = 6.$$

The next step is to calculate the hash value for the first suffix of the text:

$$h_P(CCBA) = \left(3 \cdot 3^0 + 3 \cdot 3^1 + 2 \cdot 3^2 + 1 \cdot 3^3\right)\ mod\ 11 = 57\ mod\ 11 = 2.$$

Since $h_P(CCBA) \neq h_P(ACDC)$, we know that the strings are not equal as well. The next step is to calculate the hash value for the next substring of the text. Using the rolling hash property, you can do that as follows:

$$h_P(DCCB) = \left(\left(h_P(CCBA) - 1 \cdot 3^3\right) \cdot 3 + 4\right)\ mod\ 11 = \\ ((2 - 27) \cdot 3 + 4)\ mod\ 11 = 6.$$

The hash values of the pattern and the current substring are equal, so we need to check that the strings are indeed equal. Since $DCCB \neq ACDC$, the attempt is not successful, so we move to the next substring:

$$h_P(CDCC) = \left(\left(h_P(DCCB) - 2 \cdot 3^3\right) \cdot 3 + 3\right)\ mod\ 11 = \\ ((6 - 54) \cdot 3 + 3)\ mod\ 11 = 2.$$

We can see that this hash value is not equal to the hash value of the pattern, thus the strings are not equal. The next substring to consider is $ACDC$:

$$h_P(ACDC) = \left(\left(h_P(CDCC) - 3 \cdot 3^3\right) \cdot 3 + 1\right)\ mod\ 11 = \\ ((2 - 81) \cdot 3 + 1)\ mod\ 11 = 6.$$

The hash values as well as substring match, hence we have found an occurrence. The last substring to process is $BACD$:

$$h_P(BACD) = \left(\left(h_P(ACDC) - 3 \cdot 3^3 \right) \cdot 3 + 2 \right) \ mod\ 11 = \\ ((6 - 81) \cdot 3 + 2) \ mod\ 11 = 8.$$

Since $h_P(BACD) \neq h_P(ACDC)$, the strings are not equal. All the substrings of the text are processed and the algorithm is finished.

§3. Complexity analysis

Let's estimate the running time of the algorithm for a pattern p and a text t (assuming that $|t| \geq |p|$).

The first step of the algorithm is to calculate the hash value for the pattern and then for the first suffix of the text. This requires $2 \cdot |p|$ operations. After that, we calculate the hash value for the remaining $|t| - |p|$ substrings of the text using the rolling hash property. Each of these calculations can be done in $O(1)$. When the hash values of the pattern and the current substring are equal, we perform a symbol-by-symbol comparison which requires $|p|$ operations in the worst case. So, the overall running time of the algorithm can be estimated as

$$2 \cdot |p| + (|t| - |p| + occ \cdot |p|) = O(|t| + occ \cdot |p|),$$

where occ is the number of times the pattern occurs in the text.

Note that in some cases, the algorithm may work in $O(|t| \cdot |p|)$. For example, for $p = \text{AAA}$ and $t = \text{AAAAAAAA}$, the algorithm will perform a symbol-by-symbol comparison of the pattern with each substring of the text, which results in the worst-case running time.

Also, the same situation may happen if there are a lot of collisions between the pattern and substrings of the text. However, for the polynomial hash function, the probability of a collision is $\approx \frac{1}{m}$, which is low for a big m.

To sum up, although in some cases the Rabin-Karp algorithm *may* work in $O(|t| \cdot |p|)$, in reality, such cases are quite rare, which makes this algorithm a very good and efficient alternative to the naive substring searching approach.

Report a typo

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

Start practicing

Comments (9) Hints (0) Useful links (0) Show discussion