The Rabin-Karp algorithm : A different approach to exact matching

UNIVERSITY OF MARYLAND

Eliminating spurious comparisons through “fingerprinting”

Rabin-Karp is a form of semi-numerical string matching:

Instead of focusing on comparing characters, think of string as a **sequence of bits or numbers** and use arithmetic operations to search for patterns.

Tends to work best for short patterns, and when there are relatively few occurrences of the pattern in the text.

Characters as digits

- Assume $\sum = \{0,\ldots,9\}$
- Then a string can be thought of as the decimal representation of a number:

427328

- In general, if $|\sum| = d$, a string represents a number in base d.
- Let p = the number represented by query P.
- Let t_s = the number represented by the $|P|$ digits of T that start at position s.

P occurs at position s of $T \Leftrightarrow p = t_s$.

If the pattern is "small", comparison can be fast ($O(1)$)

- Imagine $\log_2(|\sum|) * |P| <= 64$ (typical word size)
- Then, both *p* and t_s can fit in a machine word, and comparison can be done in constant time.
- 2 problems:
  - How do we *encode* the string into a word in constant time?
  - What do we do when $\log_2(|\sum|) * |P| > 64$?

Computing p and t_s

- Consider representing P via the following polynomial:

$$p = P[m] + P[m\text{-}1]10^1 + P[m\text{-}2]10^2 + \ldots + P[1]10^{m-1}$$

- Use Horner's rule to compute O(|P|=m):

$$p = P[m] + 10(P[m\text{-}1] + 10(P[m\text{-}2] + \ldots + 10(P[2] + 10P[1])\ldots)$$

- Example: $427328 = (8+10(2+10(3+10(7+10(2 + 10 \times 4)))))$

- t_0 can be computed the same way in time O(|P|=m).

- t_s can be computed from t_{s-1} **in O(1) time**:

$$t_s = \underbrace{10}_{\text{shift left by 1 digit}}(\underbrace{t_{s-1} - 10^{m-1}T[s\text{-}1]}_{\text{remove high-order digit}}) + \underbrace{T[s+m\text{-}1]}_{\text{add next digit of T as the low-order digit}}$$

Slide adapted from material by Carl Kingsford

Rabin-Karp

Compute p.

Iteratively compute t_s.

Output s when $t_s = p$.

<u>Problem:</u> p and t_s might be huge numbers.

<u>Solution:</u> compute everything modulo some large prime number q.

- If $10q$ is $\leq$ word size, then p mod q and t_s mod q can be computed in a single word.
- If p occurs at t_s, then $p \equiv t_s \pmod{q}$

<u>New problem:</u> If $p \equiv t_s \pmod{q}$, it doesn't necessarily mean there is a match at s.

<u>New solution:</u> if $p \equiv t_s \pmod{q}$, check match explicitly.

Worst-case runtime = O(mn), if every position is a match or false positive.

Slide adapted from material by Carl Kingsford

Rabin-Karp: Example

Slight deviation from above : We will follow the code presented at the end of this lecture, and adopt a 32-bit (signed) fingerprint. Nothing about these details changes the fundamental concept.

T = "try eduroam; it won't work"

P = "eduroam"

d = 256

q = 101

$$\mathbf{p} = 109+256\ (97+256\ (111+\ (256\ (114+256\ (117+256\ (100+256 * 101)))))) \ \%\ 101 = 72$$

m a o r u d e

$$\mathbf{t_0} = 117+256\ (100+256\ (101+\ (256\ (32+256\ (121+256\ (114+256 * 116)))))) \ \%\ 101 = 2$$

u d e ‘ ‘ y r t

Rabin-Karp: Example

Slight deviation from above : We will follow the code presented at the end of this lecture, and adopt a 32-bit (signed) fingerprint. Nothing about these details changes the fundamental concept.

T = "try eduroam; it won't work"

P = "eduroam"

d = 256

q = 101

$$\mathbf{p} = 109+256\,(97+256\,(111+\,(256\,(114+256\,(117+256\,(100+256 * 101)))))) \;\%\; 101 = 72$$

$$\begin{array}{ccccccc} m & a & o & r & u & d & e \end{array}$$

$\mathbf{t_0} = 2$

$256^6 \;\%\; 101$

$$\mathbf{t_1} = (256(2-25*116) + 114) \;\%\; 101 = 71$$

Rabin-Karp: Example

Slight deviation from above : We will follow the code presented at the end of this lecture, and adopt a 32-bit (signed) fingerprint. Nothing about these details changes the fundamental concept.

T = "try eduroam; it won't work"

P = "eduroam"

d = 256

q = 101

$$\mathbf{p} = 109+256\ (97+256\ (111+\ (256\ (114+256\ (117+256\ (100+256 * 101))))))\ \%\ 101 = 72$$

m a o r u d e

$$\mathbf{t_1} = 71$$

$$\mathbf{t_2} = (256(71-25*114) + 111)\ \%\ 101 = 30$$

Rabin-Karp: Example

Slight deviation from above : We will follow the code presented at the end of this lecture, and adopt a 32-bit (signed) fingerprint. Nothing about these details changes the fundamental concept.

T = "try eduroam; it won't work"

P = "eduroam"

d = 256

q = 101

$$\mathbf{p} = \underset{m}{109}+256\,(\underset{a}{97}+256\,(\underset{o}{111}+\,(256\,(\underset{r}{114}+256\,(\underset{u}{117}+256\,(\underset{d}{100}+256 * \underset{e}{101}))))))\ \%\ 101 = 72$$

$$\mathbf{t_2} = 30$$

$$\mathbf{t_3} = (256(30-25*121) + 97)\ \%\ 101 = 68$$

Rabin-Karp: Example

Slight deviation from above : We will follow the code presented at the end of this lecture, and adopt a 32-bit (signed) fingerprint. Nothing about these details changes the fundamental concept.

T = "try eduroam; it won't work"

P = "eduroam"

d = 256

q = 101

$$\mathbf{p} = \underset{m}{109} + 256\,(\underset{a}{97} + 256\,(\underset{o}{111} + (256\,(\underset{r}{114} + 256\,(\underset{u}{117} + 256\,(\underset{d}{100} + 256 * \underset{e}{101})))))) \ \%\ 101 = 72$$

$\mathbf{t_3} = 68$

$\mathbf{t_4} = (256(68-25*32) + 109)\ \%\ 101 = 72$

Rabin-Karp: Example

Slight deviation from above : We will follow the code presented at the end of this lecture, and adopt a 32-bit (signed) fingerprint. Nothing about these details changes the fundamental concept.

T = "try eduroam; it won't work"

P = "eduroam"

d = 256

q = 101

$$\mathbf{p} = \underbrace{109}_{m}+256\,(\underbrace{97}_{a}+256\,(\underbrace{111}_{o}+ (256\,(\underbrace{114}_{r}+256\,(\underbrace{117}_{u}+256\,(\underbrace{100}_{d}+256 * \underbrace{101}_{e})))))) \% \ 101 = 72$$

$\mathbf{t_3} = 68$

$\mathbf{t_4} = (256(68-25*32) + 109) \% \ 101 = 72$

$\mathbf{t_4}$

T = "try $\boxed{\text{eduroam}}$; it won't work"
P = $\boxed{\text{eduroam}}$

Rabin-Karp: Example

Slight deviation from above : We will follow the code presented at the end of this lecture, and adopt a 32-bit (signed) fingerprint. Nothing about these details changes the fundamental concept.

T = "try eduroam; it won't work"

P = "eduroam"

d = 256

q = 101

$$\mathbf{p} = 109+256\,(97+256\,(111+\,(256\,(114+256\,(117+256\,(100+256 * 101)))))) \;\%\; 101 = 72$$

m a o r u d e

$t_4 = 72$	$t_{10} = 11$	$t_{16} = 37$
$t_5 = 8$	$t_{11} = 5$	$t_{17} = 29$
$t_6 = 97$	$t_{12} = 15$	$t_{18} = 98$
$t_7 = 4$	$t_{13} = 69$	$t_{19} = 16$
$t_8 = 53$	$t_{14} = 58$	
$t_9 = 100$	$t_{15} = 84$	

Rabin-Karp Notes

- If your pattern is very small, don't need to use the (mod q) trick, and you can avoid false positive matches.
- You can also pick several different primes $q_1, q_2, \ldots, q_k$ and then require that:

$$p \equiv t_s \pmod{q_1}$$

$$p \equiv t_s \pmod{q_2}$$

$$\vdots$$

$$p \equiv t_s \pmod{q_k}$$

Rabin-Karp Notes

- Think about this with respect to DNA / RNA; how long of a pattern can we search for, without using the mod trick, if we choose the right encoding (assume machine word = 64-bits)?

Rabin-Karp Notes

- Think about this with respect to DNA / RNA; how long of a pattern can we search for, without using the mod trick, if we choose the right encoding (assume machine word = 64-bits)?

  - We can search for a pattern of length <= 32. Consider encoding each nucleotide in 2-bits e.g. A = 00, C = 01, G = 10, T = 11. Then a string of up to 32 nucleotides fits in a single machine word.

- For a good rolling hash for nucleotides, see the ntHash paper (https://academic.oup.com/bioinformatics/article/32/22/3492/2525588)

Bioinformatics, 32(22), 2016, 3492–3494
doi: 10.1093/bioinformatics/btw397
Advance Access Publication Date: 16 July 2016
Applications Note

OXFORD

Sequence analysis

ntHash: recursive nucleotide hashing

Hamid Mohamadi*, Justin Chu, Benjamin P. Vandervalk and Inanc Birol*

Canada's Michael Smith Genome Sciences Centre, British Columbia Cancer Agency, Vancouver, BC V5Z 4S6, Canada

```
/* Following program is a C++ implementation of Rabin Karp
Algorithm given in the CLRS book */
#include <bits/stdc++.h>
using namespace std;

// d is the number of characters in the input alphabet
#define d 256

/* pat -> pattern
       txt -> text
       q -> A prime number
*/
```

```
void search(char pat[], char txt[], int q)
{
        int M = strlen(pat);
        int N = strlen(txt);
        int i, j;
        int p = 0; // hash value for pattern
        int t = 0; // hash value for txt
        int h = 1;
        // The value of h would be "pow(d, M-1)%q"
        for (i = 0; i < M - 1; i++)
                h = (h * d) % q;
        // Calculate the hash value of pattern and first
        // window of text
        for (i = 0; i < M; i++)
        {
                p = (d * p + pat[i]) % q;
                t = (d * t + txt[i]) % q;
        }
        // Slide the pattern over text one by one
        for (i = 0; i <= N - M; i++)
        {
                // Check the hash values of current window of text
                // and pattern. If the hash values match then only
                // check for characters one by one
                if ( p == t )
                {
                        bool flag = true;
                        /* Check for characters one by one */
                        for (j = 0; j < M; j++)
                        {
                                if (txt[i+j] != pat[j])
                                {
                                flag = false;
                                break;
                                }
                                if(flag)
                                cout<<i<<" ";

                        }
                        // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
                        if (j == M)
                                cout<<"Pattern found at index "<< i<<endl;
                }
                // Calculate hash value for next window of text: Remove
                // leading digit, add trailing digit
                if ( i < N-M )
                {
                        t = (d*(t - txt[i]*h) + txt[i+M])%q;
                        // We might get negative value of t, converting it
                        // to positive
                        if (t < 0)
                        t = (t + q);
                }
        }
}
```

Basic implementation of Rabin-Karp following implementation in CLRS (code from https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/**)**

```
/* Driver code */
int main()
{
        char txt[] = "GEEKS FOR GEEKS";
        char pat[] = "GEEK";

        // A prime number
        int q = 101;

        // Function Call
        search(pat, txt, q);
        return 0;
}

// This is code is contributed by rathbhupendra
```