String Matching Algorithms

Tanjina Helaly

String Matching

- Also known as
  - Substring matching
  - Pattern matching
- Find pattern of length M in a text of length N.

Text

A A C A B A B A A B A B A A B

A B A A A B A A

Pattern

Applications

- Particular patterns in DNA sequences.
- Text editors
- Search engines
  - web crawling: finding strings inside other strings
- Spam detection
  - Look for pattern e.g. profit & bankaccount, lose weight
- Screen scraping
- Plagiarism Detection

Formal definition

- We formalize the string-matching problem as follows.
  - We assume that
  - **Text** T[1…n] -> an array of length n.
  - **Pattern** P[1….m] -> an array of length m <= n.
  - The elements of P and T are characters drawn from a finite alphabet ∑.
    - For example, we may have ∑ = {0,1, …} or ∑={a, b, …} or ∑={ASCII values}
  - Given strings ***T*** (text) and ***P*** (pattern), the pattern matching problem consists of finding a substring of ***T*** equal to ***P.***

What is Shift

- The pattern P ***occurs with shift*** s in text T
  - if 0 <= s <= n-m and T [s+1… s+m]= P[1…m]
  - (that is, if T[s+j]=P[j], for 1 <= j <= m).
- If P occurs with shift s in T , then we call s a ***valid shift***;
- The ***string-matching problem*** is the problem of finding **all valid shifts** with which a given pattern P occurs in a given text T .

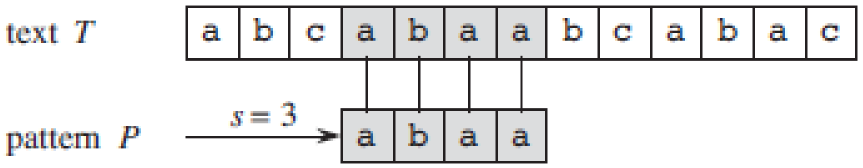


Figure 32.1 An example of the string-matching problem, where we want to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text, at shift $s = 3$, which we call a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.

Pattern matching algorithms

- Brute-force algorithm
- Rabin-Karp algorithm
- Boyer-Moore algorithm
- Knuth-Morris-Pratt algorithm

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figure 32.2 The string-matching algorithms in this chapter and their preprocessing and matching times.

The naive string-matching algorithm

- Brute force
- finds all valid shifts using a loop that checks the condition P[1…m] = T[s+1 … s+m] for each of the n-m+1 possible values of s.

```
NAIVE−STRING−MATCHER(T,P)
n = T:length
m = P:length
for s = 0 to n − m
    if P[1...m] == T[s+1 ... s+m]
        print "Pattern occurs with shift" s
```

The naive string-matching algorithm - Complexity

- Complexity -> O((n-m+1)m)
  - For each of the n-m+1 possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift.
  - Why need to compare all m even if there is a mismatch before m?
    - Has some room to improve
  - Worst case. See the example below.
    - T = aaaaaaaa
    - P = aaaa

SCOPE OF IMPROVEMENT

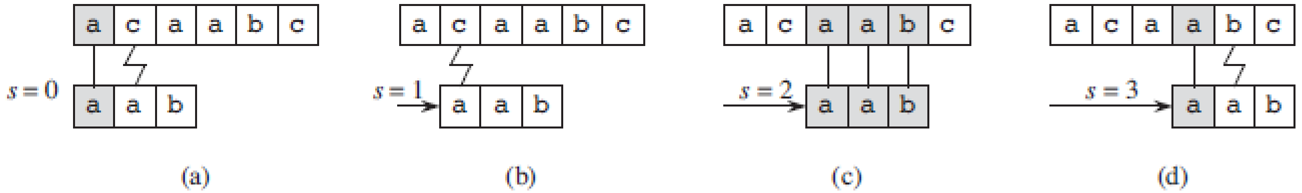


Figure 32.4 The operation of the naive string matcher for the pattern P = aab and the text T = acaabc. We can imagine the pattern P as a template that we slide next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift $s = 2$, shown in part (c).

THE NAIVE STRING-MATCHING ALGORITHM - WEAKNESS

- The naive string-matcher is inefficient because
  - it entirely ignores information gained about the text for one value of s when it considers other values of s.
  - Such information can be quite valuable, however.

- For example, if P = aaab and we find that s = 0 is valid, then none of the shifts 1, 2, or 3 are valid, since T[4] = b.

The Rabin-Karp algorithm

The Rabin-Karp algorithm

- Rabin and Karp proposed a string-matching algorithm that
  - performs well in practice and
  - also generalizes to other algorithms for related problems, such as
    - two-dimensional pattern matching.
  - The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$.
  - however, its average-case running time is better.

The Rabin-Karp algorithm

- Assume we have to find for a match of the following patter in the text below.
  - Text – BALLTHEBALL length -> N = 11
  - Pattern – BALL length -> M = 4 (M<=N)

- Assume hash(BALL) = x
- Now
  - Take a window of size M in the Text from beginning
  - Calculate the hash
  - If the hash of the window and the pattern is same.
    - Compare each character of the window and the pattern.
  - Slide the window 1 position right.
  - Repeat until the window contains <M character.

THE RABIN-KARP ALGORITHM

- The solution of working modulo q is not perfect, however:

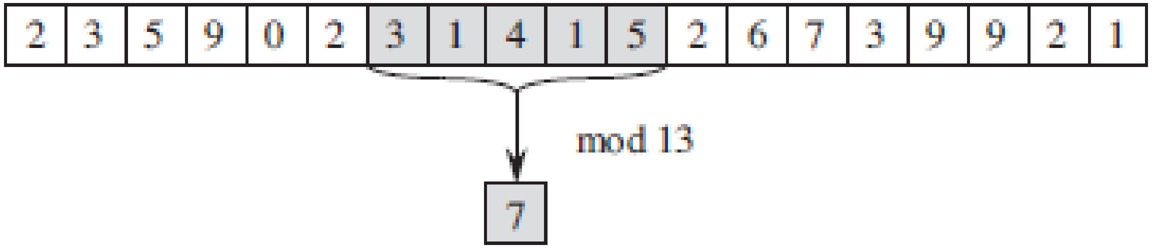
$hash(T_s) \equiv hash(P)$ *does not imply* $Ts = p$

but

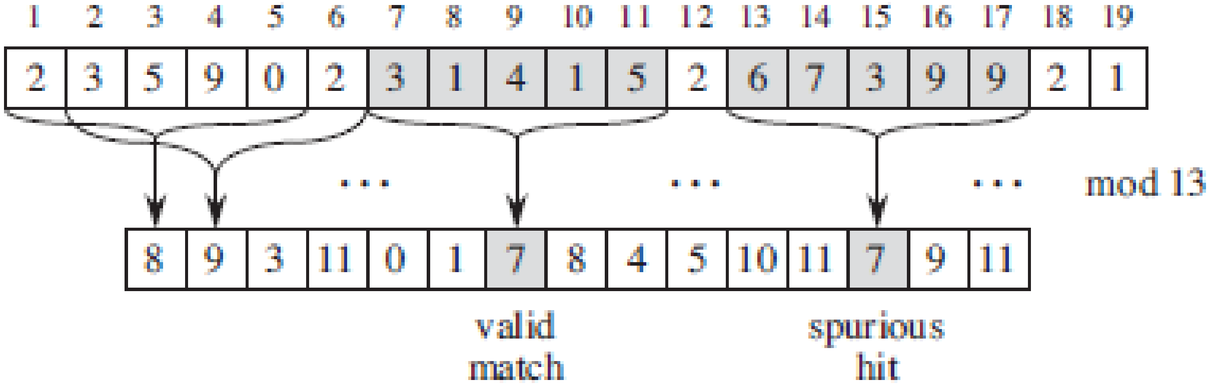
$hash(T_s) \not\equiv hash(P)$ *imply* $Ts \neq p$

- We can thus use the test $hash(T_s) \equiv hash(P)$ as a fast heuristic test to rule out invalid shifts s.
- Any shift s for which $hash(T_s) \equiv hash(P)$ must be tested further to see whether s is really valid or we just have a ***spurious hit***.

The Rabin-Karp algorithm



(a)



Can we improve?

- Use the hash value to current window calculate the hash value of next window.
  - Will reduce the time complexity of hash function from $\Theta(m)$ to $\Theta(1)$

Pattern Calculation

- Pattern – "31415"
- Value = $3 \times 10^4 + 1 \times 10^3 + 4 \times 10^2 + 1 \times 10^1 + 5 \times 10^0$
- Or $(((3 \times 10+1) \times 10+4) \times 10+1) \times 10+5$

- **Algorithm**: (for calculating the value of current window)
- p=0, P="31415", d=10, m=5

for i from 1 to m

*p = (d*p + P[i])*

Hash Calculation

- Some Mod properties
  - (A + B) mod C = ((A mod C) + (B mod C)) mod C
  - (A x B) mod C = ((A mod C) x (B mod C)) mod C

- So, Algorithm for **Mod(Hash)** calculation without any previous hash

for i from 1 to m

*p = (d*p + P[i]) mod q*

- Complexity = O(m)

Hash Calculation – Improved

2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

- T[s+1 .. s+m] = T_s = 31415

- T[s+2 .. s+m+1] = T_{s+1} =10(31415-10000x3) + 2 = 14152
  = d(T[s+1..s+m] – d^{m-1} x T[s+1]) + T[s+m+1]

Similar expression for hash calculation

- t_{s+1} = (d(t_s-T[s+1]h) + T[s+m+1]) mod q [Complexity = O(1)]
  where h = d^{m-1} mod q
  t_s = hash value of T[s+1 .. s+m] (at shift s)

- So, we are using the hash value to current window (t_s) calculate the hash value of next window (t_{s+1}).

Improvement

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

- Where
  - t_s = hash value at shift s
  - t_{s+1} = hash value at shift s+1 (next window)
  - d = radix. (number of character in the alphabet system)
    - For numerical d =10
    - For binary d =2
    - For ASCII d = 256
  - q = a prime number
  - h = d^{m-1} mod q
  - T[s+1] = leading character of current window
  - T[s+m+1] = trailing character that will be brought in in the next window.

The Rabin-Karp algorithm

RABIN-KARP-MATCHER(T, P, d, q)
n = T:*length*
m = P:*length*
h = d^{m-1} mod q
p = 0
t_0 = 0
for i = 1 **to** m // preprocessing
    p = (dp + P[i]) mod q
    t_0= (dt_0+T[i]) mod q
for s = 0 **to** n - m // matching
    if p == t_s
        if P [1 ... m] == T[s+1 ... s+m]
            print "Pattern occurs with shift" s
    if s < n- m
        t_{s+1} = (d(t_s- T[s+1]h) + T[s+m+1]) mod q

The Rabin-Karp algorithm-Time Complexity

RABIN-KARP-MATCHER(T, P, d, q)

```
n = T:length
m = P:length
h = d^(m-1) mod q                              O(m) or O(log m)
p = 0
t_0 = 0
for i = 1 to m // preprocessing                O(m)
    p = (dp + P[i]) mod q
    t_0 = (dt_0 + T[i]) mod q
for s = 0 to n - m // matching                 O(n-m+1)
    if p == t_s
        if P [1 ... m] == T[s+1 ... s+m]       O(m)*O(n-m+1)
            print "Pattern occurs with shift" s
    if s < n- m
        t_(s+1) = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

Time Complexity

- Best Case (No matching) : $\Theta(n)$ [m+n-m+1]

- Worst Case : $\Theta(mn)$ [m + (n-m+1)m]
  - Text = “aaaaaaaa”
  - Pattern = “aaaa”;

- Average Case: $\Theta(m+n)$ [realistic]

EXAMPLE

- T = BALLTHEBALL, P = BALL, q = 29 (prime number)

- Calculate hash, p
  - Repeat for each character and calculate p=dp+P[i]
  - p=(dp+P[1]) mod q = 256*0+66 mod 29 = 8
  - p=(dp+P[2]) mod q = 256*8+65 mod 29 = 25
  - p=(dp+P[3]) mod q = 256*25+76 mod 29 = 9
  - p=(dp+P[4]) mod q = 256*9+76 mod 29 = 2

- Calculate hash, t_0
  - Repeat for each character and calculate $t_0 = dt_0 + T[i]$
  - $t_0 = dt_0 + T[1])$ mod q = 256*0+66 mod 29 = 8
  - $t_0 = dt_0 + T[2])$ mod q = 256*8+65 mod 29 = 25
  - $t_0 = dt_0 + T[3])$ mod q = 256*25+76 mod 29 = 9
  - $t_0 = dt_0 + T[4])$ mod q = 256*9+76 mod 29 = 2

EXAMPLE

- P = 2, t_0 =2, m=4, n=11,
- q=29, h=256^3 mod 29 = 20

Window (red text)	t_s	2 hashes equals	Text matches ?
BALLTHEBALL	2	Yes	Yes
BALLTHEBALL	(256*(**2**-66*20) +84) mod 29 = 4	No	NA
BALLTHEBALL	(256*(**4**-65*20) +72) mod 29 = 27	No	NA
BALLTHEBALL	(256*(**27**-76*20) +69) mod 29 = 23	No	NA
BALLTHEBALL	(256*(**23**-76*20) +66) mod 29 = 11	No	NA
BALLTHEBALL	(256*(**11**-84*20) +65) mod 29 = 0	No	NA
BALLTHEBALL	(256*(**0**-72*20) +76) mod 29 = 26	No	NA
BALLTHEBALL	(256*(**26**-69*20) +76) mod 29 = 2	Yes	Yes

- Advantages:
  - Can be extended to 2d pattern.
  - Can be extended to find multiple patterns.

- Disadvantages:
  - Arithmetic operation is slower than character comparison.

- Try at home:
  - Extend Rabin-Karp to find any one of **P** possible patterns in a text of size **N**.

References

- Chapter 32 (32.1, 32.2) - Cormen