

Strings and Pattern Matching

- Brute Force, Rabin-Karp, Knuth-Morris-Pratt
- Regular Expressions



String Searching

- The previous slide is not a great example of what is meant by “String Searching.” Nor is it meant to ridicule people without eyes....
- The object of **string searching** is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).
- As with most algorithms, the main considerations for string searching are speed and efficiency.
- There are a number of string searching algorithms in existence today, but the three we shall review are **Brute Force**, **Rabin-Karp**, and **Knuth-Morris-Pratt**.

Brute Force



- The **Brute Force** algorithm compares the pattern to the text one character at a time, until unmatching characters are found

<i>T</i> WO ROADS DIVERGED IN A YELLOW WOOD <i>ROADS</i>
T <i>W</i> O ROADS DIVERGED IN A YELLOW WOOD <i>ROADS</i>
TW <i>O</i> ROADS DIVERGED IN A YELLOW WOOD <i>ROADS</i>
TWO ROADS DIVERGED IN A YELLOW WOOD <i>ROADS</i>
TWO ROADS DIVERGED IN A YELLOW WOOD ROADS

Compared characters are *italicized*.

Correct matches are in **boldface** type.

- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.

Brute Force Pseudo-Code

- Here's the pseudo-code
- ```
do if (text letter == pattern letter)
 compare next letter of pattern to next
 letter of text
else move pattern down text by one letter
while (entire pattern found or end of text)
```

The diagram shows the process of finding the pattern "cat" in the text "cool cat Rolo went over the fence". It consists of six rows, each representing a step in the search. The text is displayed in a monospaced font. The pattern "cat" is shown in red. The first row shows the pattern starting at the first position of the text, with the 'c' of "cat" aligned under the 'c' of "cool". The subsequent rows show the pattern shifted one position to the right in each step. The fifth row shows the pattern shifted to the position of the second "cat" in the text. The sixth row shows the pattern shifted to the position of the third "cat" in the text, which is highlighted with a black box, indicating a successful match.

|                                   |     |
|-----------------------------------|-----|
| cool cat Rolo went over the fence | cat |
| cool cat Rolo went over the fence | cat |
| cool cat Rolo went over the fence | cat |
| cool cat Rolo went over the fence | cat |
| cool_cat Rolo went over the fence | cat |
| cool cat Rolo went over the fence | cat |

# Brute Force-Complexity

- Given a pattern M characters in length, and a text N characters in length...
- Worst case: compares pattern to each substring of text of length M. For example, M=5.
- This kind of case can occur for image data.

```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAH 5 comparisons made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAH 5 comparisons made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAH 5 comparisons made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAH 5 comparisons made
5) AAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAH 5 comparisons made
...
N) AAAAAAAAAAAAAAAAAAAAAAAABAAAH
 AAAAH 5 comparisons made
```

Total number of comparisons:  $M(N-M+1)$

Worst case time complexity:  $O(MN)$

# Brute Force-Complexity(cont.)

- Given a pattern M characters in length, and a text N characters in length...
- Best case if pattern found: Finds pattern in first M positions of text. For example, M=5.

1) **AAAAA**AAAAAAAAAAAAAAAAAAAAAAAAAH  
**AAAAA**      **5 comparisons made**

Total number of comparisons: M  
Best case time complexity:  $O(M)$

# Brute Force-Complexity(cont.)

- Given a pattern M characters in length, and a text N characters in length...
- Best case if pattern not found: Always mismatch on first character. For example, M=5.

```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
 OOOOH 1 comparison made
2) AA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
 OOOOH 1 comparison made
3) AAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
 OOOOH 1 comparison made
4) AAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
 OOOOH 1 comparison made
5) AAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
 OOOOH 1 comparison made
...
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAH
 1 comparison made OOOOH
```

Total number of comparisons: N

Best case time complexity:  $O(N)$

# Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will do a **Brute Force** comparison between the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.
- Perhaps an example will clarify some things...



# Rabin-Karp Example

- Hash value of “AAAAA” is 37
- Hash value of “AAAAH” is 100

```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAAH
 37≠100 1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAAH
 37≠100 1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAAH
 37≠100 1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
 AAAAAH
 37≠100 1 comparison made
...
N) AAAAAAAAAAAAAAAAAAAAAAAABAAAAH
 BAAAAH
 100=100
 5 comparisons made
```

# Rabin-Karp Algorithm

*pattern is M characters long*

**hash\_p**=hash value of pattern

**hash\_t**=hash value of first M letters in body of text

**do**

**if** (**hash\_p** == **hash\_t**)

    brute force comparison of pattern

    and selected section of text

**hash\_t**= hash value of next section of text, one character over

**while** (end of text      **or**

    brute force comparison == true)

# Rabin-Karp

- Common Rabin-Karp questions:
  - “What is the hash function used to calculate values for character sequences?”
  - “Isn’t it time consuming to hash every one of the M-character sequences in the text body?”
  - “Is this going to be on the final?”
- To answer some of these questions, we’ll have to get mathematical.

# Rabin-Karp Math

- Consider an M-character sequence as an M-digit number in base b, where b is the number of letters in the alphabet. The text subsequence  $t[i .. i+M-1]$  is mapped to the number

$$x(i) = t[i] \cdot b^{M-1} + t[i+1] \cdot b^{M-2} + \dots + t[i+M-1]$$

- Furthermore, given  $x(i)$  we can compute  $x(i+1)$  for the next subsequence  $t[i+1 .. i+M]$  in constant time, as follows:

$$x(i+1) = t[i+1] \cdot b^{M-1} + t[i+2] \cdot b^{M-2} + \dots + t[i+M]$$

$$x(i+1) = x(i) \cdot b$$

Shift left one digit

$$- t[i] \cdot b^M$$

Subtract leftmost digit

$$+ t[i+M]$$

Add new rightmost digit

- In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character.

# Rabin-Karp Math Example

- Let's say that our alphabet consists of 10 letters.
- our alphabet = a, b, c, d, e, f, g, h, i, j
- Let's say that “a” corresponds to 1, “b” corresponds to 2 and so on.

The hash value for string “cah” would be ...

$$3*100 + 1*10 + 8*1 = 318$$

# Rabin-Karp Mods

- If  $M$  is large, then the resulting value ( $\sim b^M$ ) will be enormous. For this reason, we hash the value by taking it **mod** a **prime number  $q$** .
- The **mod** function ( $\%$  in Java) is particularly useful in this case due to several of its inherent properties:

$$[(x \bmod q) + (y \bmod q)] \bmod q = (x+y) \bmod q$$

$$(x \bmod q) \bmod q = x \bmod q$$

- For these reasons:

$$h(i) = ((t[i] \cdot b^{M-1} \bmod q) + (t[i+1] \cdot b^{M-2} \bmod q) + \dots + (t[i+M-1] \bmod q)) \bmod q$$

$$h(i+1) = (h(i) \cdot b \bmod q$$

Shift left one digit

$$- t[i] \cdot b^M \bmod q$$

Subtract leftmost digit

$$+ t[i+M] \bmod q )$$

Add new rightmost digit

$$\bmod q$$

# Rabin-Karp Complexity

- If a sufficiently large prime number is used for the *hash function*, the hashed values of two different patterns will usually be distinct.
- If this is the case, searching takes  $O(N)$  time, where  $N$  is the number of characters in the larger body of text.
- It is always possible to construct a scenario with a worst case complexity of  $O(MN)$ . This, however, is likely to happen only if the prime number used for hashing is small.

# The Knuth-Morris-Pratt Algorithm

- The **Knuth-Morris-Pratt (KMP)** string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.
- A **failure function (f)** is computed that indicates how much of the last comparison can be reused if it fails.
- Specifically,  $f$  is defined to be the longest prefix of the pattern  $P[0,...,j]$  that is also a suffix of  $P[1,...,j]$ 
  - Note:** **not** a suffix of  $P[0,...,j]$
- Example:-value of the
- KMP failure function:

|        |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|
| $j$    | 0 | 1 | 2 | 3 | 4 | 5 |
| $P[j]$ | a | b | a | b | a | c |
| $f(j)$ | 0 | 0 | 1 | 2 | 3 | 0 |
- This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.
  - if the comparison fails at (4), we know the a,b in positions 2,3 is identical to positions 0,1



# The KMP Algorithm (contd.)

- the KMP string matching algorithm: Pseudo-Code

Algorithm KMPMatch(T,P)

**Input:** Strings T (text) with n characters and P (pattern) with m characters.

**Output:** Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

# Algorithm

```
f ← KMPSFailureFunction(P) {build failure function}
i ← 0
j ← 0
while i < n do
 if P[j] = T[i] then
 if j = m - 1 then
 return i - m - 1 {a match}
 i ← i + 1
 j ← j + 1
 else if j > 0 then {no match, but we have advanced}
 j ← f(j-1) {j indexes just after matching prefix in P}
 else
 i ← i + 1
return "There is no substring of T matching P"
```

# The KMP Algorithm (contd.)

- The KMP failure function: Pseudo-Code

Algorithm **KMPMatch**(T,P)

**Input:** String P (pattern) with m characters

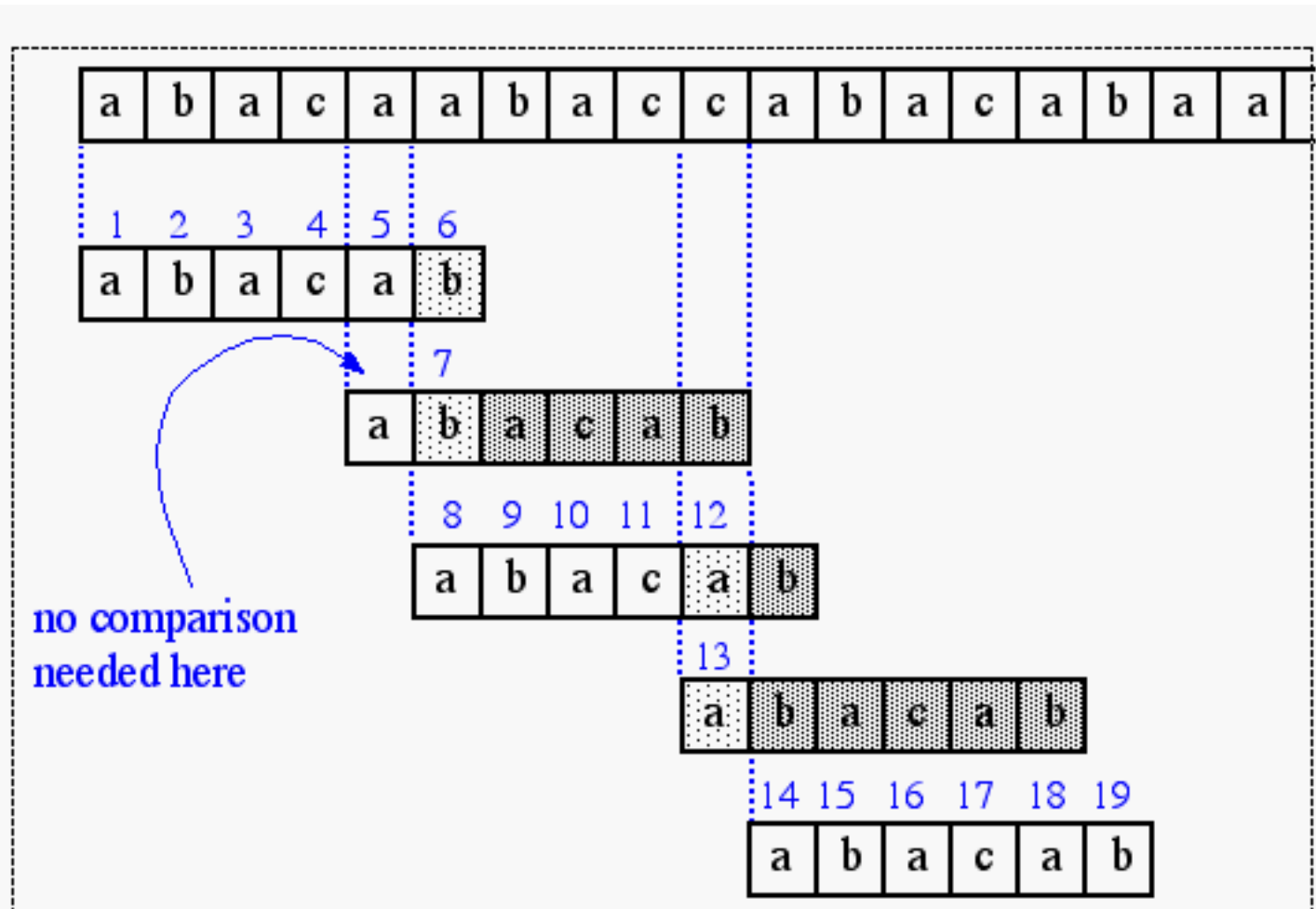
**Output:** The failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of P[1,...,j]

# Algorithm

```
f ← KMPSuccessFunction(P) {build failure function}
i ← 0
j ← 0
while i ≤ m-1 do
 if P[j] = T[i] then
 if j = m - 1 then
 { we have matched j+1 characters}
 f(i) ← j + 1
 i ← i + 1
 j ← j + 1
 else if j > 0 then
 j ← f(j-1) {j indexes just after matching prefix in P}
 else {there is no match}
 f(i) ← 0
 i ← i + 1
```

# The KMP Algorithm (contd.)

- A graphical representation of the KMP string searching algorithm



# The KMP Algorithm (contd.)

- Time Complexity Analysis
- define  $k = i - j$
- In every iteration through the while loop, one of three things happens.
  - 1) if  $T[i] = P[j]$ , then  $i$  increases by 1, as does  $j$   $k$  remains the same.
  - 2) if  $T[i] \neq P[j]$  and  $j > 0$ , then  $i$  does not change and  $k$  increases by at least 1, since  $k$  changes from  $i - j$  to  $i - f(j-1)$
  - 3) if  $T[i] \neq P[j]$  and  $j = 0$ , then  $i$  increases by 1 and  $k$  increases by 1 since  $j$  remains the same.
- Thus, each time through the loop, either  $i$  or  $k$  increases by at least 1, so the greatest possible number of loops is  $2n$
- This of course assumes that  $f$  has already been computed.
- However,  $f$  is computed in much the same manner as `KMPMatch` so the time complexity argument is analogous. `KMPFailureFunction` is  $O(m)$
- Total Time Complexity:  $O(n + m)$

# Regular Expressions

- notation for describing a set of strings, possibly of infinite size
- $\epsilon$  denotes the empty string
- $ab + c$  denotes the set  $\{ab, c\}$
- $a^*$  denotes the set  $\{\epsilon, a, aa, aaa, \dots\}$
- Examples
  - $(a+b)^*$  all the strings from the alphabet  $\{a,b\}$
  - $b^*(ab^*a)^*b^*$  strings with an even number of a's
  - $(a+b)^*\text{sun}(a+b)^*$  strings containing the pattern “sun”
  - $(a+b)(a+b)(a+b)a$  4-letter strings ending in a