Next Computational Problem: String Searching

AUBURN UNIVERSITY

Hugh Kwon

Slides adapted from Dr. Debswapna Bhattacharya's class

Problem Specification

Inputs:

(1)A string of **n** characters from a pre-specified alphabet in which to search, called the "text" T

(2)Another string of **m** characters from the same alphabet that is being searched for, called the "pattern" P, **m≤n**

Outputs: Print the locations of **all** occurrences of P in T, each stated in terms of the "shifts" s, **0≤s≤n−m** – the number of characters that have to be skipped over to get to that occurrence of P; if P does not appear in T then print nothing

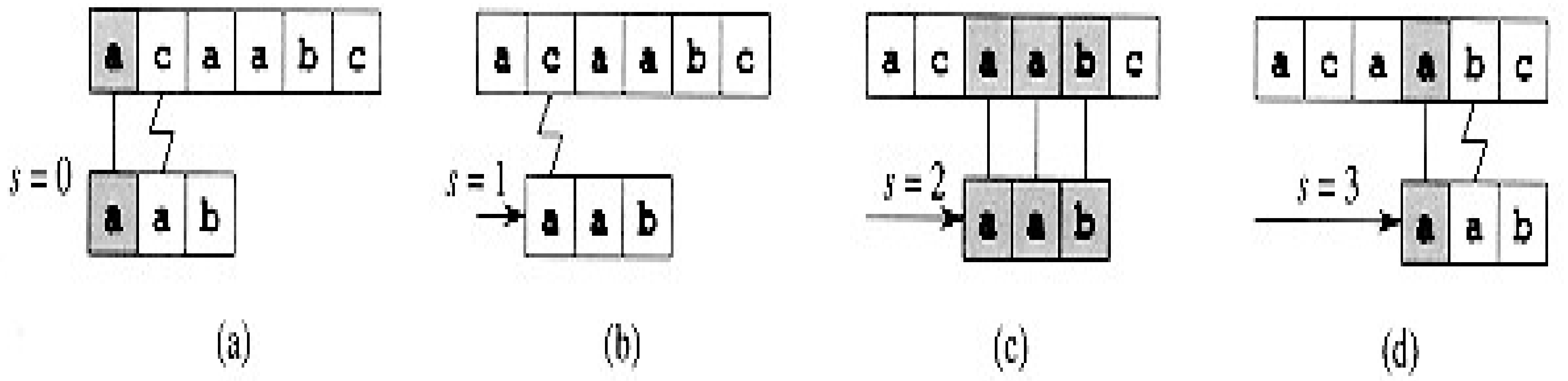
Correctness criterion: For every s printed, P must appear in T at locations s+1...s+m

String Searching/Matching

- Applicable whenever what you are searching for and where you are searching can both me modeled as strings from an alphabet

- E.g.,
  - Searching for specific genes in DNA
  - Searching for words in documents
  - Web searching

What is an obvious or naïve strategy?

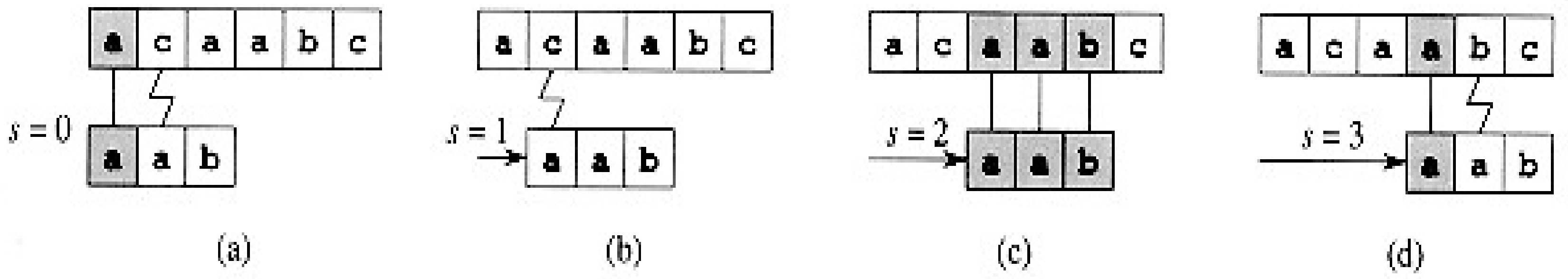


Naïve-String-Matcher (T,P: arrays of char)

1. n=T.length
2. m=P.length
3. for s=0 to n-m
4. if P[1…m]==T[s+1…s+m]
5. print “pattern occurs with shift” s

Complexity:

$O(m(n-m+1))=O(mn-m^2+1)=O(mn)$ when $m<n$



- Key idea:
  - Convert string P into a number #P=456
  - Convert string T into a number #T=34567
  - Look for #P in #T
  - Why is it better than naïve matching?
    - Because numbers can be checked for equality without matching digit by digit!

How to convert a character string to a number?

- Suppose size of alphabet = d
- So there are d possible characters
- Assign a digit 0…(d-1) to each
- Replace string with the digits
- Then you get a number in the base d

- Alphabet has 10 characters a…j
- a=0, b=1,…j=9
- P=bcd, #P=123
- Same transformation applied to the text string T

Computing the "value" of a number

- But there is the problem of computing the value of #P
- #P=345 on a base of 10 => value of #P = $5.10^0+4.10^1+3.10^2$
- To convert an m-character string $a_{m-1}a_{m-2}\ldots a_1a_0$ to a number in base d one has to compute $a_0d^0+a_1d^1+\ldots a_{m-1}d^{m-1}$
- This can be efficiently calculated by Horner's rule: $a_0d^0+a_1d^1+\ldots a_{m-1}d^{m-1} = a_0 + d(a_1 + d(a_2+\ldots d(a_{m-1})))))$ E.g., 345=5+10(4+10(3))
- Computing the value of an m-digit number corresponding to an m-character Pattern can be done with $\Theta(m)$ basic operations

But what about matching with T?

- if $|P|=m$ and $|T|=n$ then $(n-m+1)$ substrings of length m have to be converted too before #P can be compared with each!

- Say #P=456 and #T=34567 then #P needs to be equality checked with 345, 456 and 567

- Complexity $(n-m+1) * \Theta(m)=\Theta(m(n-m+1))$

- naïve matching is $O(m(n-m+1)$

Be Smart

- Say #P=456 and #T=34567 so m=3, n=5

- 456 has to be compared with each of t_0= 345, t_1= 456, t_2= 567

- Once “value” t_0=345 is calculated, “value” of t_1 = $10(345-10^2*3)+6$

- Once “value” t_1 of is calculated, “value” of t_2 = $10(456-10^2*4)+7$

In General

- For $s=0\ldots n-m$ let t_s be the m-character-long substring of text $T[s+1\ldots s+m]$
- Then $t_0=T[1\ldots m]$, $t_1=T[2\ldots.m+1]$ and so on
- t_{s+1} can be calculated directly from t_s in constant time (8 arithmetic operations and 2 array references)!
- $t_{s+1}=d(t_s-d^{m-1}T[s+1])+T[s+m+1]$
- If the base d is 10, $t_{s+1}=10(t_s-10^{m-1}T[s+1])+T[s+m+1]$

Strategy

1. Let m=|P| and n=|T|
2. Precompute d^{m-1} (for repeated use in calculating t_{s+1} from t_s)
3. Calculate #P and #t0
4. Repeat for i=0 to (n-m)
    1. Check if #P==#t_i
    2. If so, one occurrence of P found
    3. Calculate #t_{i+1} from #t_i

- Complexity: $\Theta(m)+\Theta(n-m+1) = \Theta(n-m+1) = O(n)$ when $m<n$

One issue!

- How big can these numbers get?

- Solution: Calculate all numbers modulo q where q is a large prime number
- Problem: $\#t_s$(modulo q) == #P(modulo q) DOES NOT mean that $\#t_s$ == #P
- BUT
- $\#t_s$(modulo q) $\neq$ #P(modulo q) DOES mean that #ts $\neq$ #P

- So spurious hits/matches can occur. Therefore each match needs to be verified character by character.

R-K-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==ts
        if P[1…m]==T[s+1…s+m]
            print "P occurs with shift" s
    if s<m
        t_{s+1}=(d(t_s-T[s+1]h)+T[s+m+1]) mod q
```

Complexity Change

- At most (n-m+1) matches, each requiring m character comparisons to verify
- New Complexity:
- $\Theta(m)+\Theta(n-m+1)+O(m(n-m+1)) = O(m(n-m+1))$: same as naïve matcher in the worst case!

- BUT
- in practice only small number of matches (~ a constant number c) are found so the complexity is actually closer to $\Theta(m)+\Theta(n-m+1)+O(cm)$ where c is a constant = $\Theta(m)+O(n)+O(m) = \Theta(m)+O(n)$ when $m<n$

- Chapter Introduction
  - omit Notation and Terminology (p.986-987)
- 32.1
- 32.2 (omit the discussion on p.994)

Thinking Assignments

- Problems 32.1-1 & 32.1-2
- Problems 32.2-1 & 32.2-2

AUBURN UNIVERSITY