Analysis of Algorithms
CSC 402
2022-23

Subject Incharge

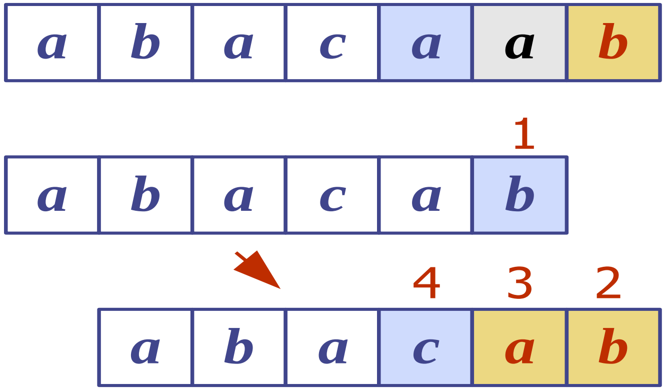
Bidisha Roy

Associate Professor

Room No. 401

email: bidisharoy@sfit.ac.in

STRINGS AND PATTERN MATCHING



STRINGS

- A string is a sequence of characters
- Let P be a string of size m
  - A substring P[i .. j] of P is the substring of P consisting of the characters with ranks between i and j
  - A prefix of P is a substring of the type P[0 .. i]
  - A suffix of P is a substring of the type P[i ..m - 1]
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}

Pattern Matching

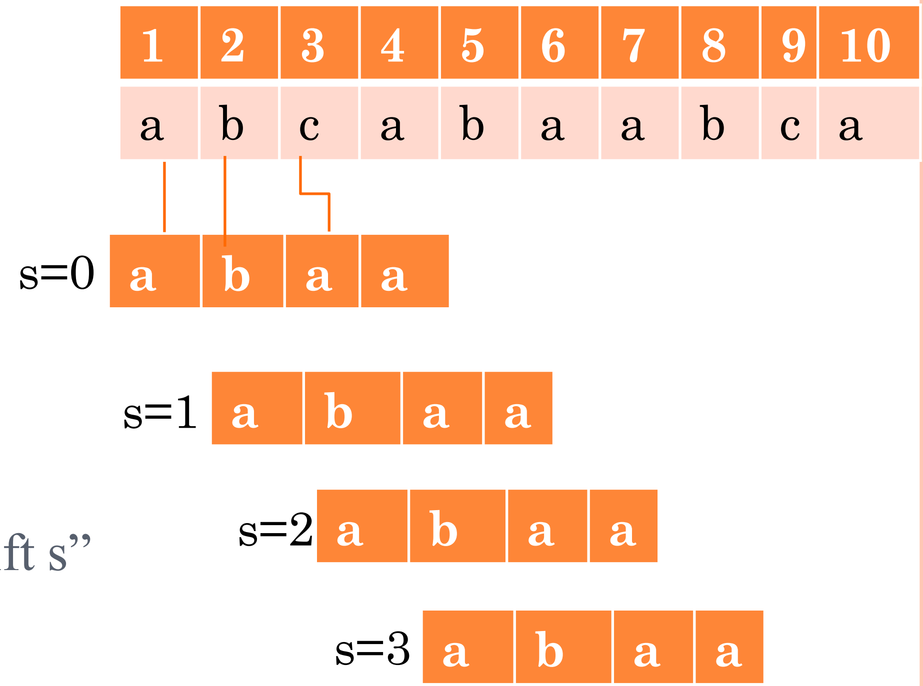
- Given strings ***T*** (text) and ***P*** (pattern), the ***pattern matching problem*** consists of finding a substring of ***T*** equal to ***P***
- Applications:
  - Text editors
  - Search engines
  - Biological research
- Algorithms to be considered
  - Naïve String Matching Algorithms
  - Rabin Karp Algorithm
  - Knuth-Morris-Pratt Algorithm

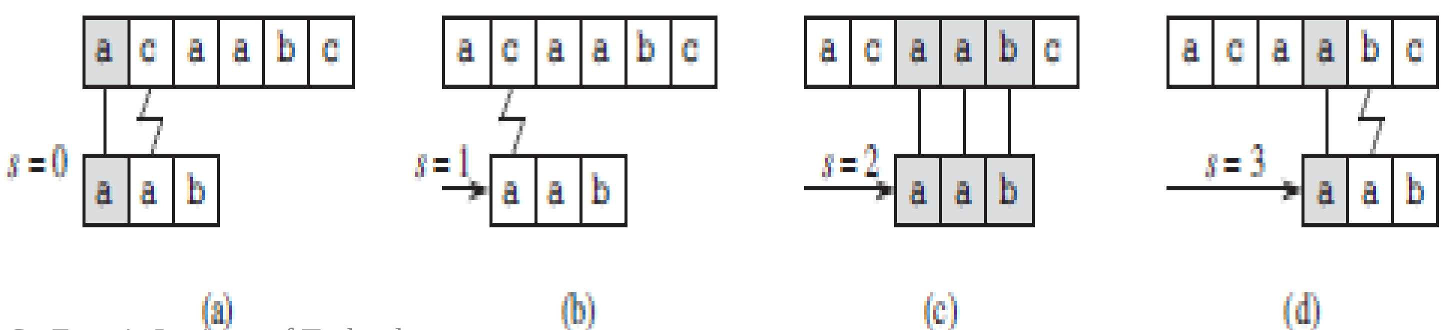
Naïve String Matching Algorithm

- A brute force approach
- Compares first character of Pattern P with Text T
  - If matched, pointers in both string incremented
  - Else, pointer of T is shifted and pointer of P is reset
- Repeated till the end of text or pattern is found

Naïve String Matching Algorithm

- ***NaiveStringMatch(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"





NAÏVE STRING MATCHING ALGORITHM (ALTERNATE IMPLEMENTATION)

- **Algorithm** *NaiveStringMatch*(*T, P*)
  - **Input** text ***T*** of size ***n*** and pattern ***P*** of size ***m***
  - **Output** starting index of a substring of ***T*** equal to ***P*** or indication if no such substring exists
  - **for** $s \leftarrow 0$ **to** $n - m$ ***do***
    - $j \leftarrow 0$
    - **while** $j < m$ ***and*** $T[s + j] = P[j]$ ***do***
      - $j \leftarrow j + 1$
    - **if** (j=m) ***then***
      - **return s** **//Pattern found at shift s** I
  - **return** "There is no substring of T matching P"

....Runs in time O(nm)

Rabin Karp Algorithm

- Performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching
- Idea
  - Compare a string's hash values, rather than the strings themselves
  - The algorithm slides/shifts the pattern one by one matching the hash value of pattern P with the hash value of current substring of T.
  - If the hash value matches then only checks the individual characters of P with substring of T

Rabin Karp Algorithm

- The use of hashing converts a string to a numeric value and comparing numbers is easier than comparing strings
- Requirement
  - Efficient Hash Function
  - Hash value of the next shift should be efficiently calculated/computed from the hash value of the current position

Rabin Karp Algorithm… working

- Characters in both arrays T and P be digits in radix-Σ notation
  - (eg. Σ = (0,1,…,9)
- p be the hash value of the characters in P
- t_s denote the decimal hash value of length m substring in T after shift s
- Choose a prime number *q* such that fits within a computer word to speed computations
- Compute (p mod q)
  - The value of p mod q is what we will be using to find all matches of the pattern P in T.

Rabin Karp Algorithm… working

- Compute (T[s+1, .., s+m] mod q) for s = 0 .. n-m
- Test against P only those sequences in T having the same (mod q) value
- (T[s+1, .., s+m] mod q) can be incrementally computed by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.
- If hash values are same i.e. hash(p) = hash (t_s), actual characters of both strings are compared.
  - If pattern is same, its called a *hit*. Else it's a *spurious hit*

Rabin Karp Algorithm

- The algorithm takes as input:
  - The text T
  - The pattern P
  - The radix d to use (typically $|\Sigma|$)
  - The prime q to use

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

```
1  n = T.length
2  m = P.length
3  h = d^(m-1) mod q
4  p = 0
5  t_0 = 0
6  for i = 1 to m                      // preprocessing
7      p = (dp + P[i]) mod q
8      t_0 = (dt_0 + T[i]) mod q
9  for s = 0 to n - m                  // matching
10     if p == t_s
11         if P[1..m] == T[s+1..s+m]
12             print "Pattern occurs with shift" s
13     if s < n - m
14         t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

Note: if ts<0, ts=ts+q

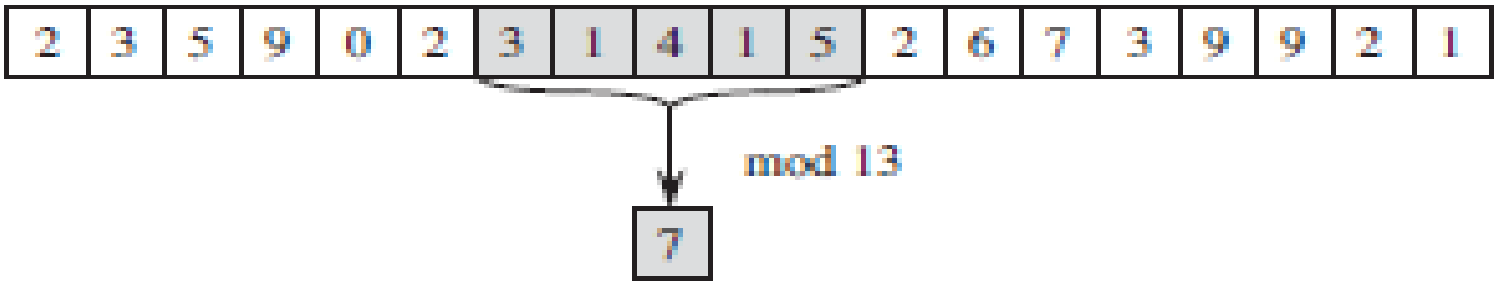
- T="ABDCB", P="DC"
- q=11, n=5, m=2
- d=256 (as there are 256 ASCII characters)
- h=256^(2-1)mod11=3
- After steps 6-8
  - p=7, t_0=8

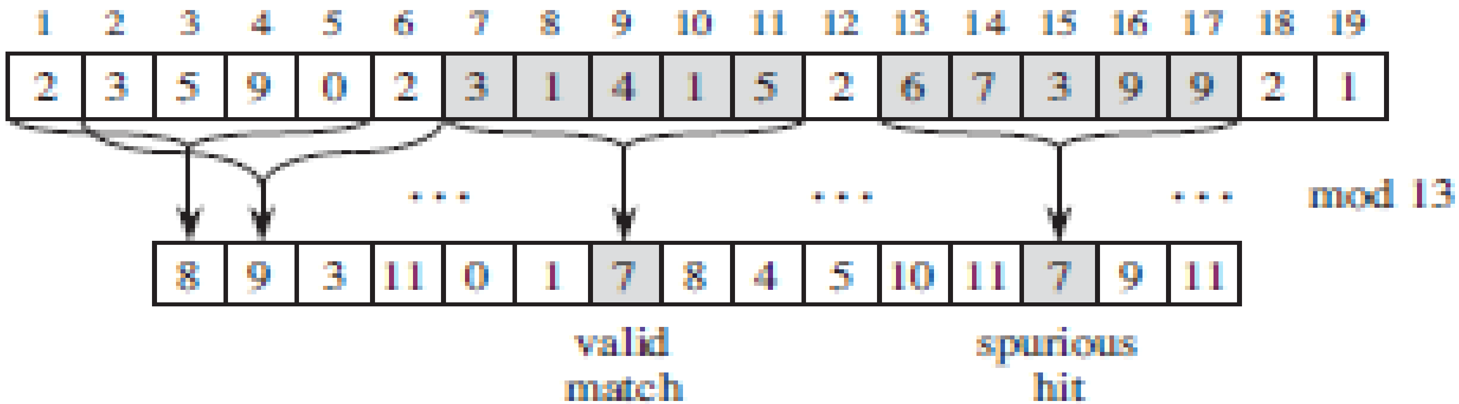
s	substring	if p==t_s	New t_s
0	***AB***DCB	8≠7	t1=2
1	A***BD***CB	7≠2	t2=7
2	AB***DC***B	7=7, Also pattern matches with shift 2	t3=3
3	ABD***CB***	8≠3	3!<3

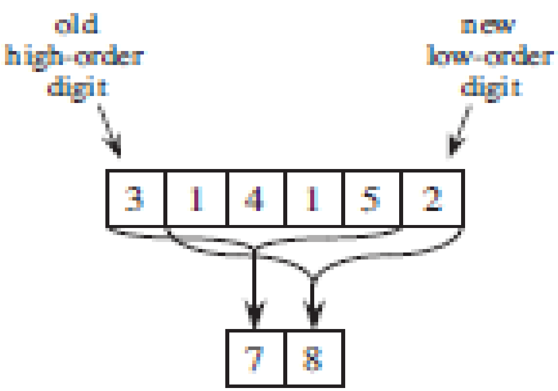
P= 31415 , Q=13, P MOD Q=7 D=10

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T=	2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

Ts=	8	9	3	11	0	1	7	8	4	5	10	11	7	9	11
Shifts=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14







t_7= [10(7-3*3)+2] mod 13

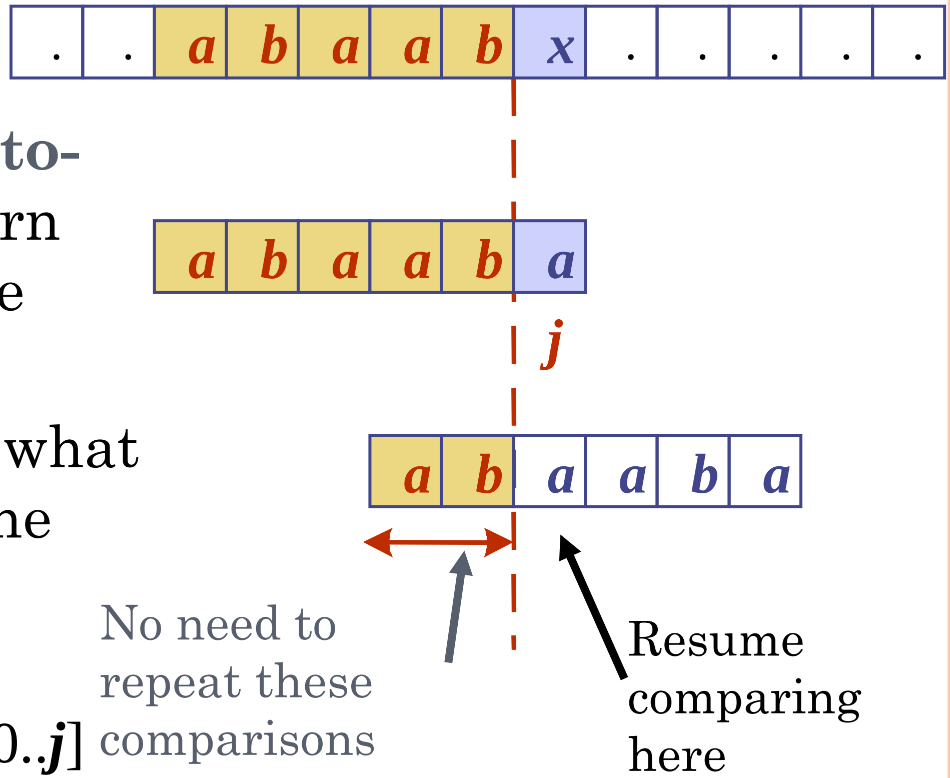
t_7=8

Complexity

- This algorithm uses Θ(m) preprocessing time and its worst case running time is **O((n-m+1)m)**

KNUTH-MORRIS-PRATT ALGORITHM

- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
  - The largest prefix of ***P***[0..***j***] that is a suffix of ***P***[1..***j***]



KMP Algorithm … Prefix Function

- The main idea: to preprocess the Pattern String P so as to compute a *prefix function* π that indicates the proper shift of P so that, to the largest extent possible we can reuse previously performed comparisons.
- The **prefix function** $\pi(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow \pi(j-1)$
- Initially start with $\pi(1) = 0$.

KMP Algorithm … Prefix Function

Prefix-Function(P)

1. m=P.length
2. ***Π[1..m]*** new prefix array
3. ***Π[1]***=0
4. i←0
5. for j← 2 to m
   1. while i>0 and P[i+1]≠P[j]
      1. i← ***π[i]***
   2. If P[i+1]=P[j] then
      1. i ← i +1
   3. Π[j]← i
6. Return ***π***

- m is length of P
- i → longest prefix found in the pattern (also the longest suffix at P[j])
- j→ current index of the pattern for which we have to calculate ***π*** value
- ***Π*** is the **prefix table** array

j	1	2	3	4	5	6
P[***j***]	***a***	***b***	***a***	***c***	***a***	***b***
π(***j***)	0	0	1	0	1	2

KMP Algorithm … Prefix Function

- The prefix function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $\pi(j-1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

KMP Algorithm

Algorithm ***KMPMatch***(*T*, *P*)

1. n← T.length
2. m←P.length
3. ***Π*** **←** Prefix-Function(P)
4. i←0 //number of characters matched
5. for j←1to n
    1. while i>0 and P[i+1]≠T[j] do
        1. i← ***π[i]*** //Skip using prefix table
    2. if P[i+1]=T[j] then
        1. i←i+1 //Next Character matches
    3. if i==m then
        1. "Pattern occurs with shift" j-m
        2. i← ***π[i]***

T = b a c b a b a b a b a c a a b (n=15)
P = a b a b a c a (m=7) (Mismatch in 1st character itself. i=0, i.e. go to next j)

T = b a c b a b a b a b a c a a b
P = a b a b a c a (Mismatch in 2nd character i.e. i=1. Shift P by π(1)=0, 0 places from j)

T = b a c b a b a b a b a c a a b
P = a b a b a c a (For next two iterations shift by 1)

T = b a c b a b a b a b a c a a b
P = a b a b a c a (Mismatch occurs in 6th character, i=5. Shift by π(5) = 3, 3 spaces after last i=0 or go 3 places from j)… contd

	1	2	3	4	5	6	7
P(j)	a	b	a	b	a	c	a
π(j)	0	0	1	2	3	0	1

T = b a c b a b a b a b a c a a b

P = a b a b a c a **(Pattern matched. Index of j is 13. Therefore pattern found after 13-m=13-7=6 shifts. Next shift π(7)=1)**
String ends.

	1	2	3	4	5	6	7
P(j)	a	b	a	b	a	c	a
π(j)	0	0	1	2	3	0	1

$j \rightarrow$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	b	a	a	b	b

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 6th character, j=6, i=5(length of matched characters). Shift will be π(i)=π(5)=1, go back 1 place before j

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 2nd character, j=6, i=1(length of matched characters). Shift will be π(i)=π(1)=0, 0 places from j

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 5th character, j=10, i=4(length of matched characters). Shift will be π(i)=π(4)=0

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 1st character, j=10, i=0. Shift by 1 character

	1	2	3	4	5	6
***P*[]**	***a***	***b***	***a***	***c***	***a***	***b***
π()	0	0	1	0	1	2

1	2	3	4	5	6
a	b	a	c	a	b

All characters match, j=16, i=6.
"Match occurs at 16-6 = 10 shift".
Shift = π(6) =2 places

Alternate Logic

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	b	a	a	b	b

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 6th character, j=5, i=5(length of matched characters). Shift will be i-π(i)=5-π(5)=5-1=4, i.e shift P by 4 places

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 2nd character, j=6, i=1(length of matched characters). Shift will be i-π(i)=1-π(1)=1-0=1, i.e shift P by 1 place

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 5th character, j=10, i=4(length of matched characters). Shift will be i-π(i)=4-π(4)=4-0=4, i.e shift P by 4 places

1	2	3	4	5	6
a	b	a	c	a	b

Mismatch occurs at 1st character, j=10, i=0. Shift by 1 character

	1	2	3	4	5	6
***P*[]**	***a***	***b***	***a***	***c***	***a***	***b***
π()	0	0	1	0	1	2

1	2	3	4	5	6
a	b	a	c	a	b

All characters match, j=16, i=6.
"Match occurs at 16-6 = 10 shift".
Shift = 6- π(6) =2

T = b a c b a b a b a b a c a a b (n=15)
P = a b a b a c a (m=7) (Mismatch in 1st character. Shift by 1)

T = b a c b a b a b a b a c a a b
P =   a b a b a c a (Mismatch in 2nd character i.e. i=1. Shift P by i-π(i)=1-0=1)

T = b a c b a b a b a b a c a a b
P =   a b a b a c a (For next two iterations shift by 1)

T = b a c b a b a b a b a c a a b
P =   a b a b a c a (Mismatch occurs in 6th character, i=5. Shift by 5 - π(5) = 5-3=2)... contd

	1	2	3	4	5	6	7
P(j)	a	b	a	b	a	c	a
π(j)	0	0	1	2	3	0	1

T = b a c b a b a b a b a c a a b

P = a b a b a c a **(Pattern matched. Index of j is 13. Therefore pattern found after 13-m=13-7=6 shifts. Next shift 7-π(7)=7-1=6)**

String ends.

	1	2	3	4	5	6	7
P(j)	a	b	a	b	a	c	a
π(j)	0	0	1	2	3	0	1

KMP Algorithm… Analysis

- The prefix function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the for-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $\pi(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

Other Algorithms

- Boyer Moore String Matching Algorithm
- String Matching with Finite Automata