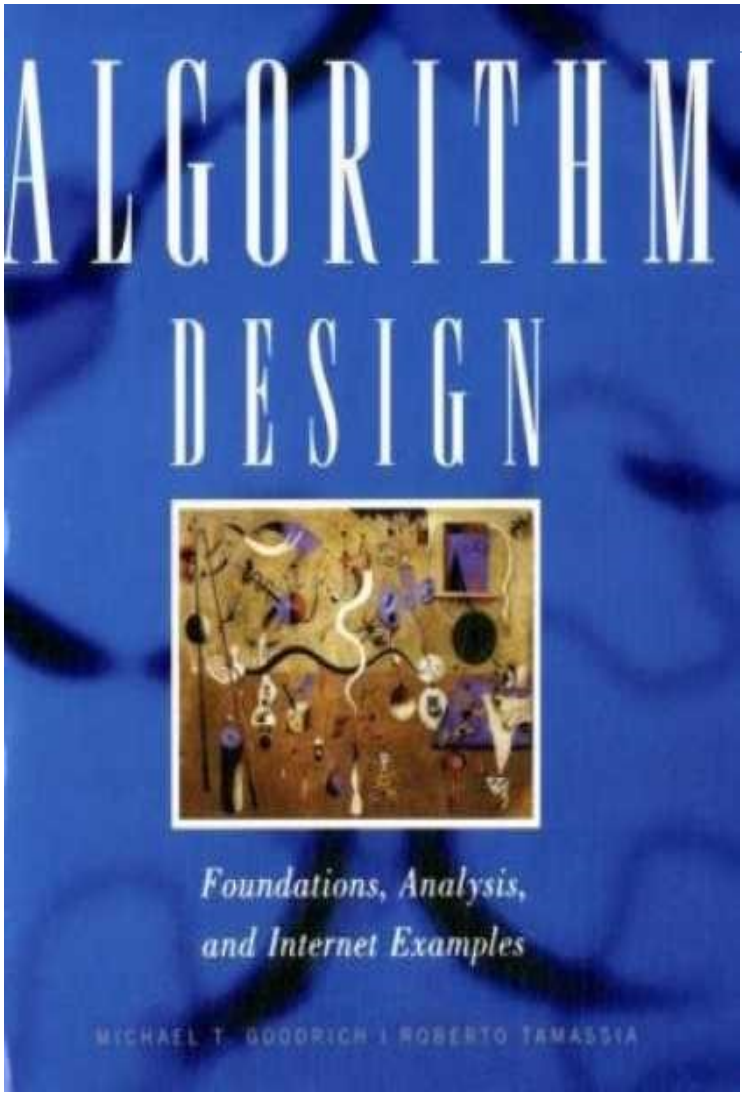
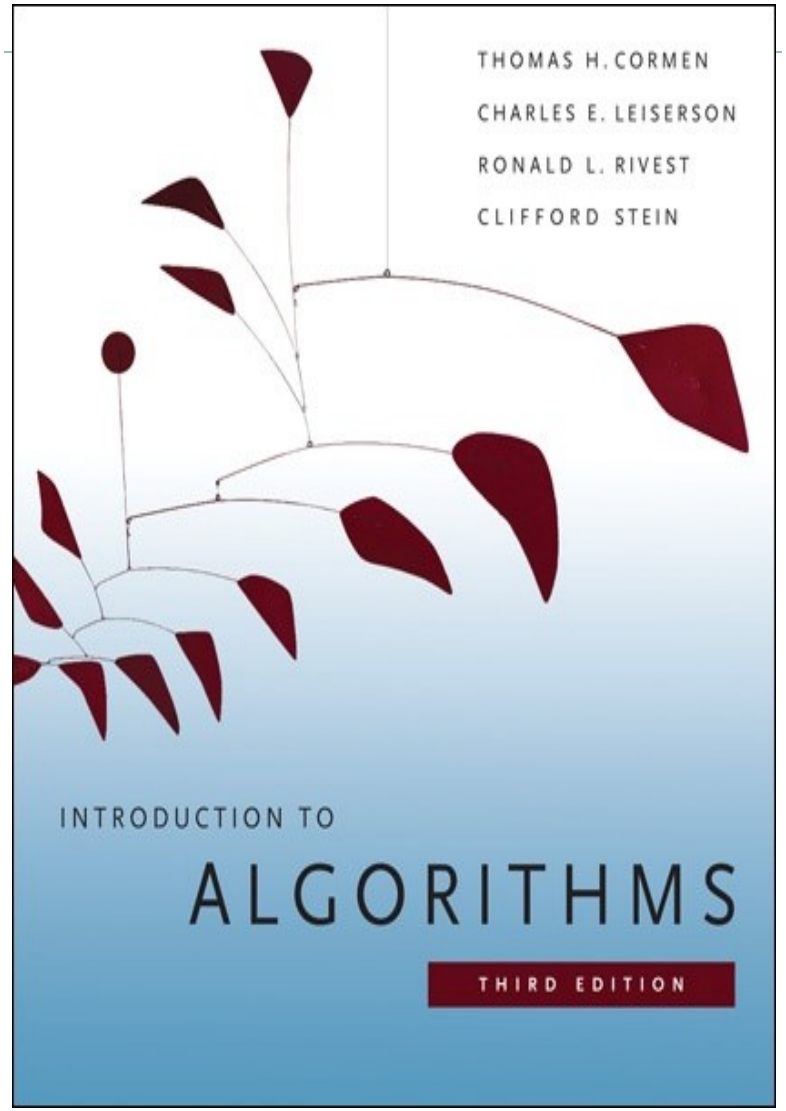
Internet Algorithm
Pattern Matching

Dr. Bibhudatta Sahoo,
National Institute of Technology Rourkela

Reference Books





Introduction

- **What is string matching ?**
  - Finding all occurrences of a *pattern* in a given *text* (or *body of text*)
- Many applications

1. While using editor/word processor/browser
2. Login name & password checking
3. Virus detection
4. Header analysis in data communications
5. DNA sequence analysis
6. Searching engines (like Google and Openfind)
7. Database (GenBank)

Solving real world problem with string matching

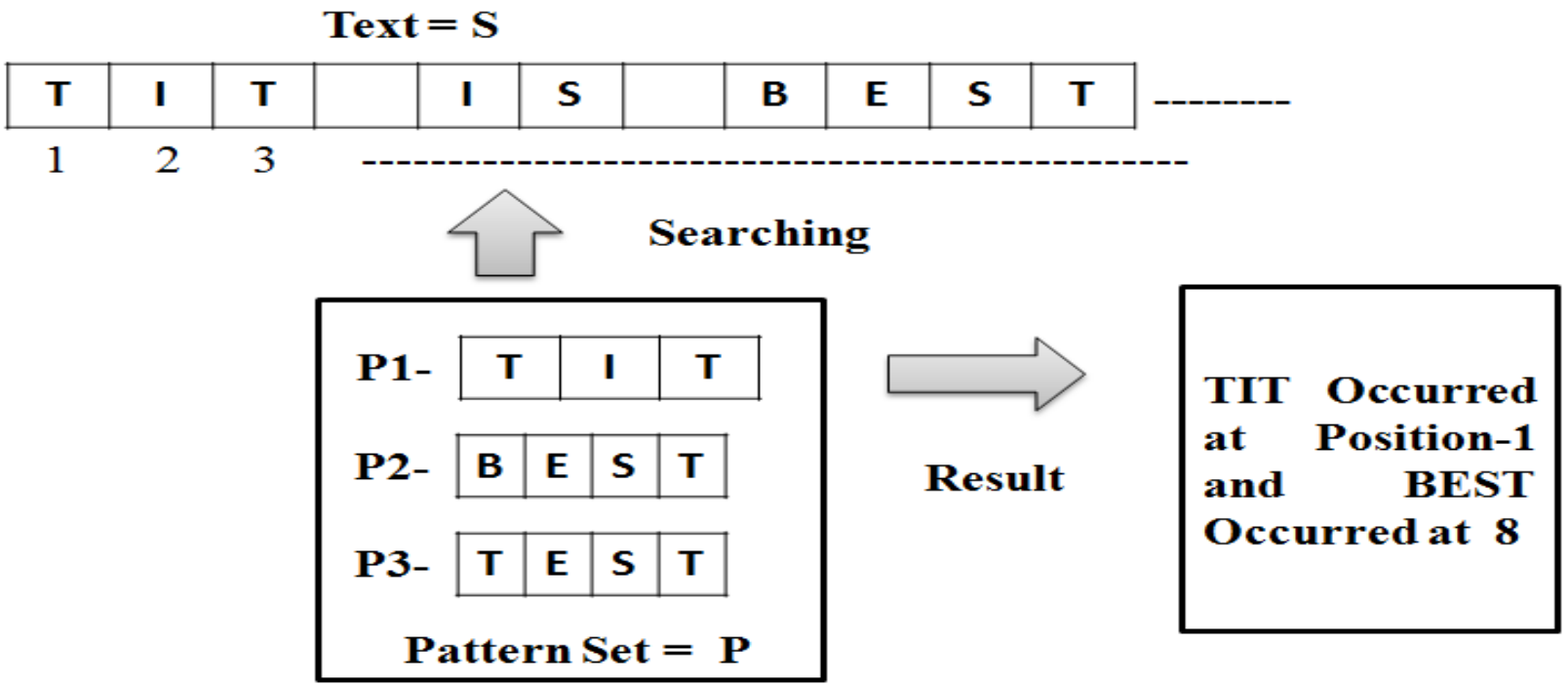
- To search some designated strings in computers is a classical and important problem in computer science.
- We can search or replace some texts in word processors, find some interesting websites from searching engines (like Yahoo and Google) and search for papers from digital libraries.
- The main problem of those operations is string matching.
-

Solving real world problem with string matching

- Many databases (like GenBank) were built to support DNA and protein sequences contributed by researchers in the world.
- Since DNA and protein sequences can be regarded as character strings composed of 4 and 20 different characters respectively, the string matching problem is the core problem for searching these databases.

The problem of String Matching

- Given a string T, the problem of string matching deals with finding whether a pattern P occurs in T and if P does occur then returning position in T where P occurs.



Strings

- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - Digitized image
- 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}

The pattern matching problem

- Let P be a string of size m
  - A substring $P[i \,..\, j]$ of P is the subsequence 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 $P[i \,..\, m-1]$
- 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

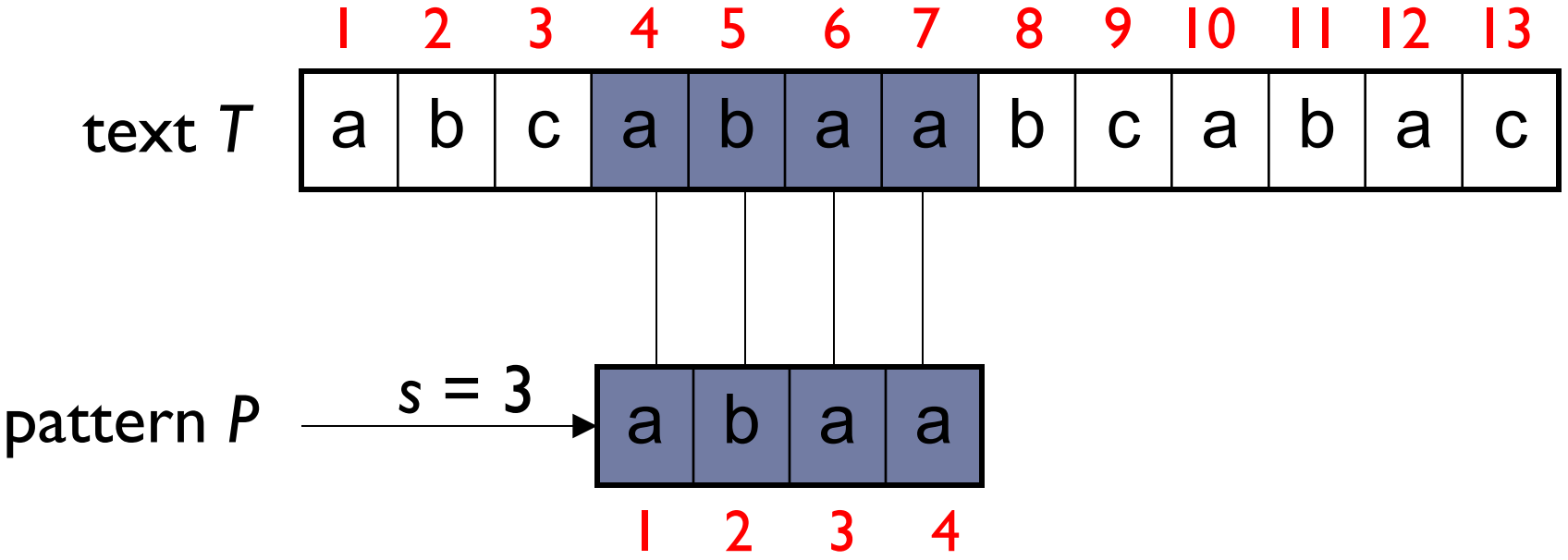
String-Matching Problem

- The *text* is in an array $T[1..n]$ of length n
- The *pattern* is in an array $P[1..m]$ of length m
- Elements of T and P are characters from a *finite alphabet* Σ
  - E.g., $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$
- Usually T and P are called *strings* of characters

String-Matching Problem: Shift

- We say that pattern P *occurs with shift* s in text T if:
  a. $0 \le s \le n-m$ and
  b. $T[(s+1)..(s+m)] = P[1..m]$
- If P occurs with shift s in T, then s is a *valid shift*, otherwise s is an *invalid shift*
- String-matching problem: finding all valid shifts for a given T and P

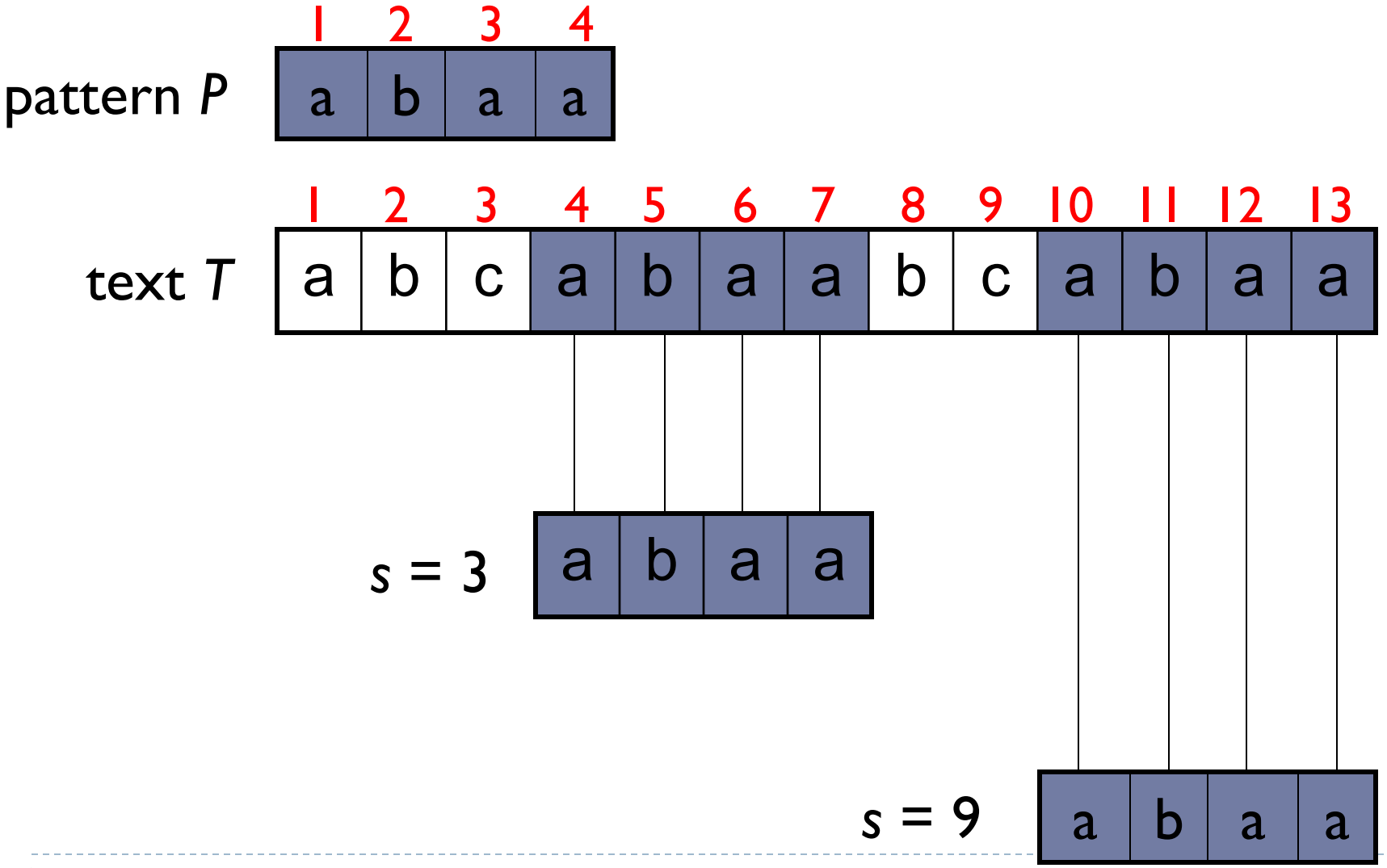
Example 1: SHIFT



shift $s = 3$ is a valid shift

(n=13, m=4 and $0 \leq s \leq n\text{-}m$ holds)

Example 2: SHIFT



String Matching Algorithm

- Algorithms based on character comparison:
  - **Naive Algorithm:** It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
  - **KMP (Knuth Morris Pratt) Algorithm**: The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.
  - **Boyer Moore Algorithm:** This algorithm uses best heurestics of Naive and KMP algorithm and starts matching from the last character of the pattern.
  - **Using the Trie data structure:** It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST.

String Matching Algorithm

- **Deterministic Finite Automaton (DFA) method:**
  - **Automaton Matcher Algorithm:** It starts from the first state of the automata and the first character of the text. At every step, it considers next character of text, and look for the next state in the built finite automata and move to a new state.
- **Algorithms based on Bit (parallelism method):**
  - **Aho-Corasick Algorithm:** It finds all words in O(n + m + z) time where n is the length of text and m be the total number characters in all words and z is total number of occurrences of words in text. This algorithm forms the basis of the original Unix command fgrep.
- **Hashing-string matching algorithms:**
  - **Rabin Karp Algorithm:** It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

Brute-Force Algorithm

String-Matching; a O(mn) approach

- One of the most obvious approach towards the string matching problem would be to compare the first element of the pattern to be searched P, with the first element of the string T in which to locate P.
- If the first element of P matches the first element of T, compare the second element of P with second element of T.
- If match found proceed likewise until entire P is found.
- If a mismatch is found at any position, shift P one position to the right and repeat comparison beginning from first element of P.

Brute force algorithm

Step1: Align pattern at beginning of text

Step2: Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step3: While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

- Time complexity (worst-case): O(mn)

How does the O(mn) approach work

Below is an illustration of how the previously described O(mn) approach works.

String T

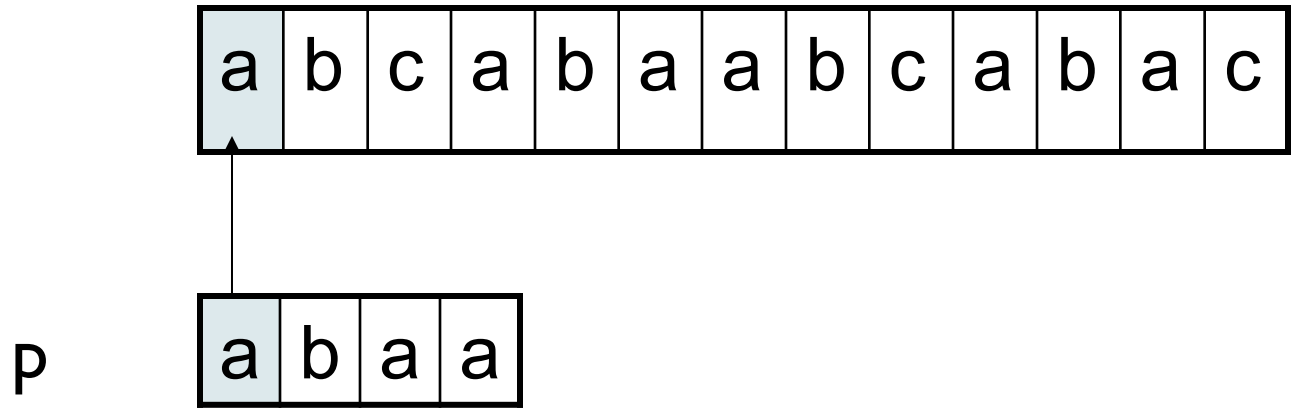
a	b	c	a	b	a	a	b	c	a	b	a	c

Pattern P

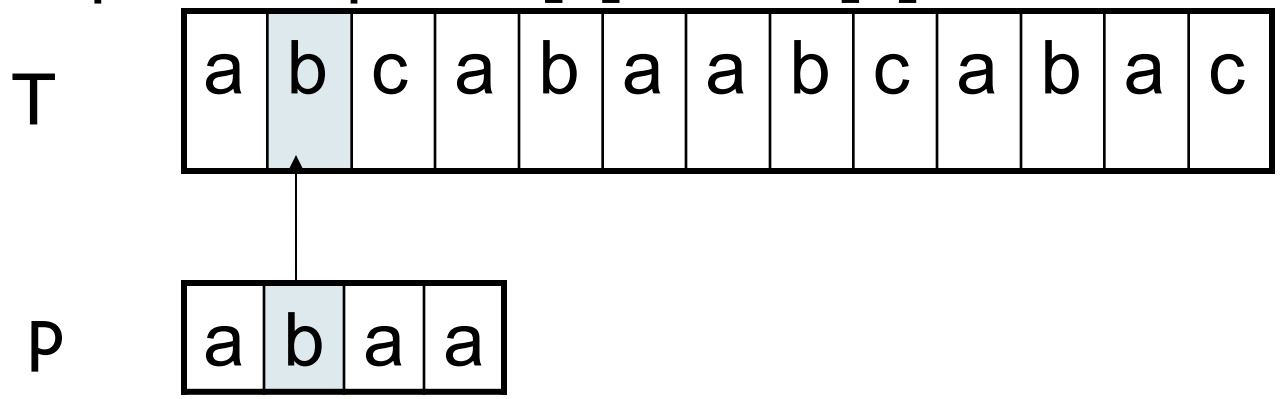
a	b	a	a

The brute-force pattern matching: 1

- Step 1:compare P[1] with T[1]
- T



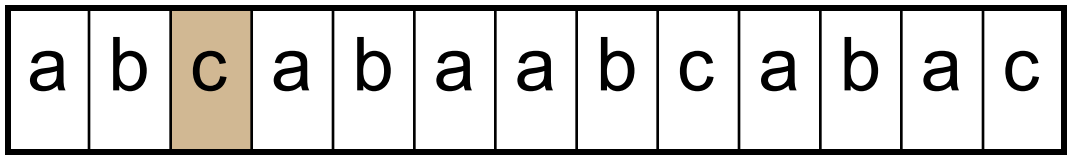
Step 2: compare P[2] with T[2]



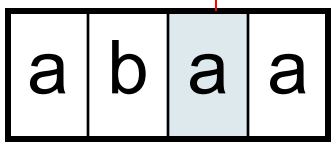
The brute-force pattern matching: 2

Step 3: compare P[3] with T[3]

T



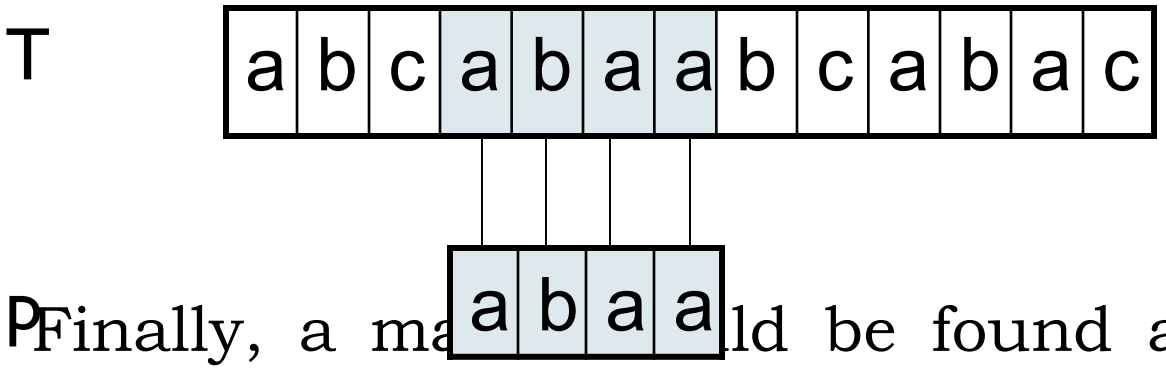
P



Mismatch occurs here..

Since mismatch is detected, shift 'P' one position to the right and repeat matching procedure.

The brute-force pattern matching: 3



- Finally, a match would be found after shifting 'P' three times to the right side.

Brute-Force Algorithm

- The brute-force pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences
  - unlikely in English text

```
Algorithm BruteForceMatch(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 −1
        if no such substring exists
    for i ← 0 to n − m
        { test shift i of the pattern }
        j ← 0
        while j < m ∧ T[i + j] = P[j]
            j ← j + 1
        if j = m
            return i {match at i}
        else
            break while loop {mismatch}
    return -1 {no match anywhere}
```

Brute-Force Algorithm

Algorithm 1: Naive String Searching Algorithm

```
Data: P: The pattern to look for
      T: The text to look in
Result: Returns the number of occurrences of P in T
answer ← 0;
n ← length(T);
k ← length(P);
for i ← 0 to n − k do
    valid ← true;
    for j ← 0 to k − 1 do
        if T[i + j] ≠ P[j] then
            valid ← false;
            break;
        end
    end
    if valid = true then
        answer ← answer + 1;
    end
end
return answer;
```

The Complexity of this algorithm is O(n.k), where n is the length of the text, and k is the length of the pattern.

Example : Brute-Force string matching

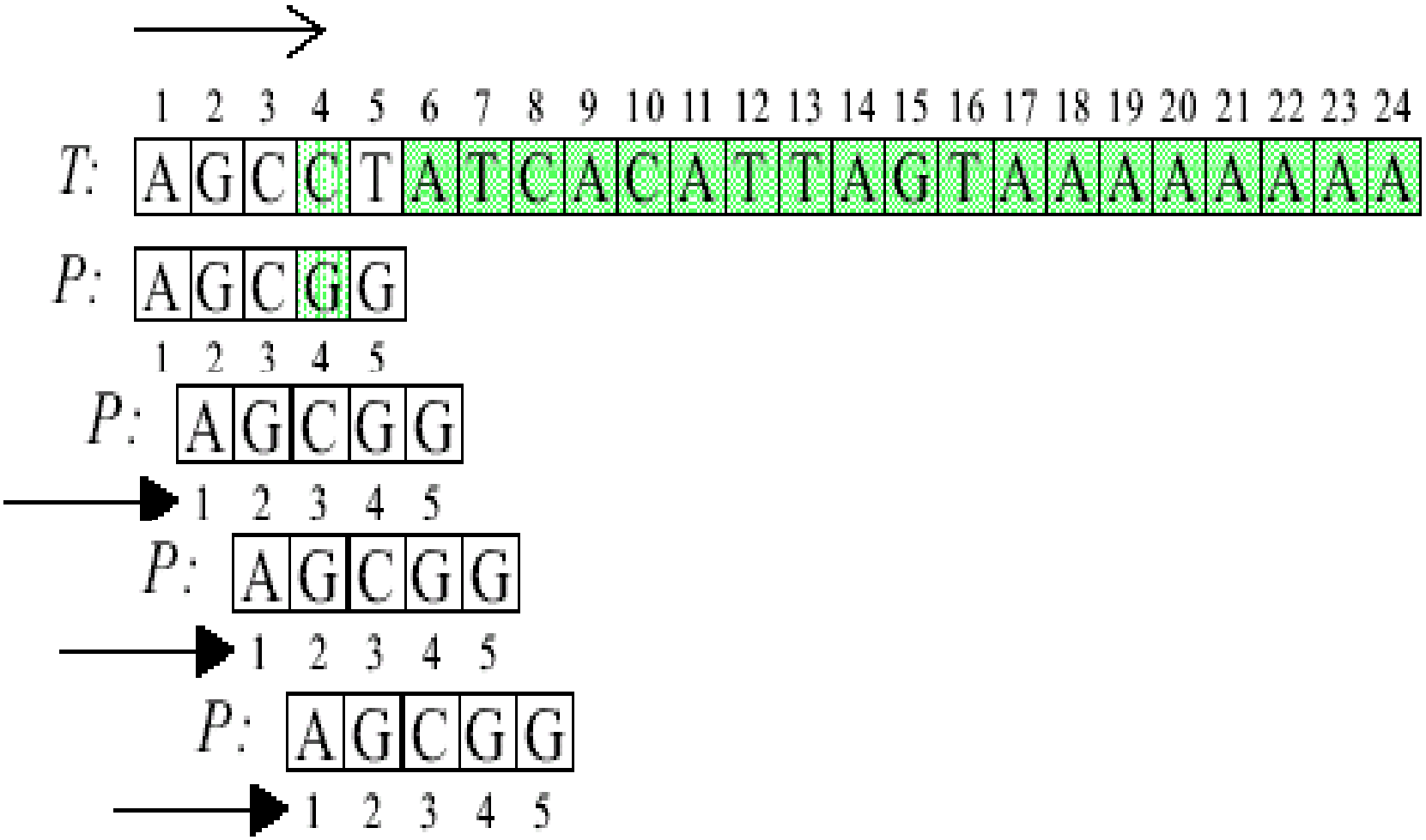
In this example, P ="bab", and T="ababaac "

Location	Comparison	Matching Result
0	**a b a** b a a c **b a b**	Mismatch
1	a **b a b** a a c **b a b**	**Match**
2	a b **a b a** a c **b a b**	Mismatch
3	a b a **b a a** c **b a b**	Mismatch
4	a b a b **a a c** **b a b**	Mismatch

Drawback of brute-force pattern matching

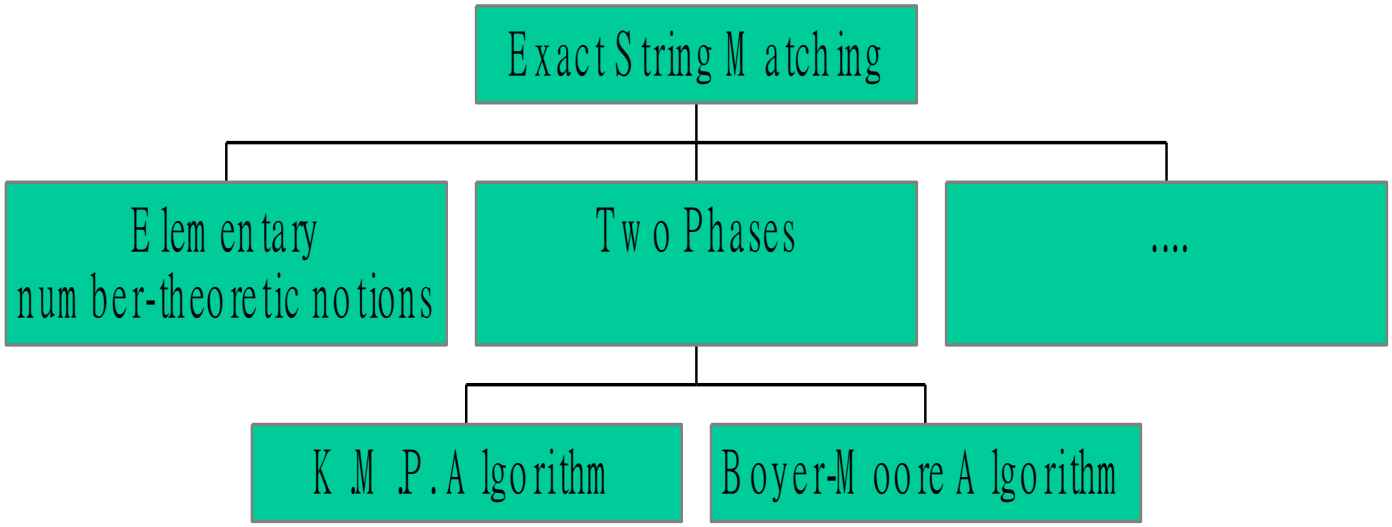
- If 'm' is the length of pattern 'P' and 'n' the length of string 'T', the matching time is of the order O(mn). This is a certainly a very slow running algorithm.
- What makes this approach so slow is the fact that elements of 'T' with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations.
- For example: when mismatch is detected for the first time in comparison of P[3] with T[3], pattern 'P' would be moved one position to the right and matching procedure would resume from here.
- Here the first comparison that would take place would be between P[0]='a' and T[1]='b'.
- It should be noted here that T[1]='b' had been previously involved in a comparison in step 2. this is a repetitive use of T[1] in another comparison.
- It is these repetitive comparisons that lead to the runtime of O(mn).

A Brute-Force Algorithm: worst case



Two phase string matching Algorithm

Two Phase



Phase 1 : generate an array to indicate the moving direction.

Phase 2 : make use of the array to move and match string

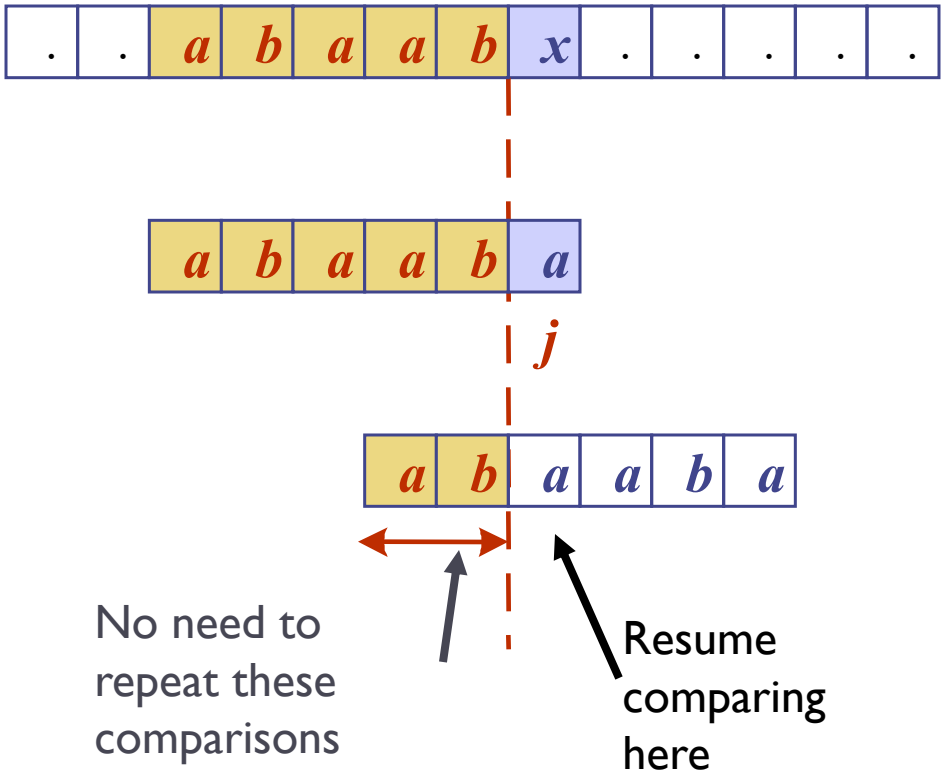
The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt Algorithm

- Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.
- A matching time of O(n) is achieved by avoiding comparisons with elements of T that have previously been involved in comparison with some element of the pattern P to be matched. i.e., backtracking on the string T never occurs

The KMP Algorithm - Motivation

- 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**?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$



Concept of KMP algorithm

- Idea: after some character (such as q) matches of P with T and then a mismatch, the matched q characters allows us to determine immediately that certain shifts are invalid. So directly go to the shift which is potentially valid.
- The matched characters in T are in fact a prefix of P, so just from P, it is OK to determine whether a shift is invalid or not.
- Define a **prefix function** π, which encapsulates the knowledge about how the pattern P matches against shifts of itself.
  - $\pi : \{1,2,\ldots,m\} \rightarrow \{0,1,\ldots,m-1\}$
  - $\pi[q] = \max\{k: k<q \text{ and } P_k \Leftarrow P_q\}$, that is $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q.

Components of KMP algorithm

- The prefix function, Π

  The prefix function,Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself.

  This information can be used to avoid useless shifts of the pattern P. In other words, this enables avoiding backtracking on the string T.

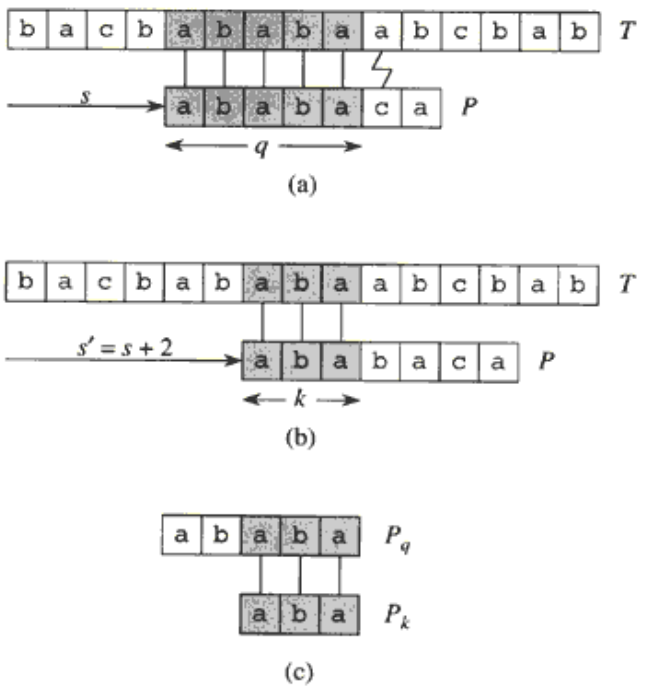
- The KMP Matcher

  With string T, pattern P and prefix function 'Π' as inputs, finds the occurrence of P in ' and returns the number of shifts of P after which occurrence is found.

The KMP Matcher

```
KMP-MATCHER(T, P)
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                                  ▷ Number of characters matched.
 5  for i ← 1 to n                         ▷ Scan the text from left to right.
 6       do while q > 0 and P[q + 1] ≠ T[i]
 7            do q ← π[q]                  ▷ Next character does not match.
 8       if P[q + 1] = T[i]
 9         then q ← q + 1                 ▷ Next character matches.
10       if q = m                         ▷ Is all of P matched?
11         then print "Pattern occurs with shift" i − m
12              q ← π[q]                  ▷ Look for the next match.
```

Prefix function

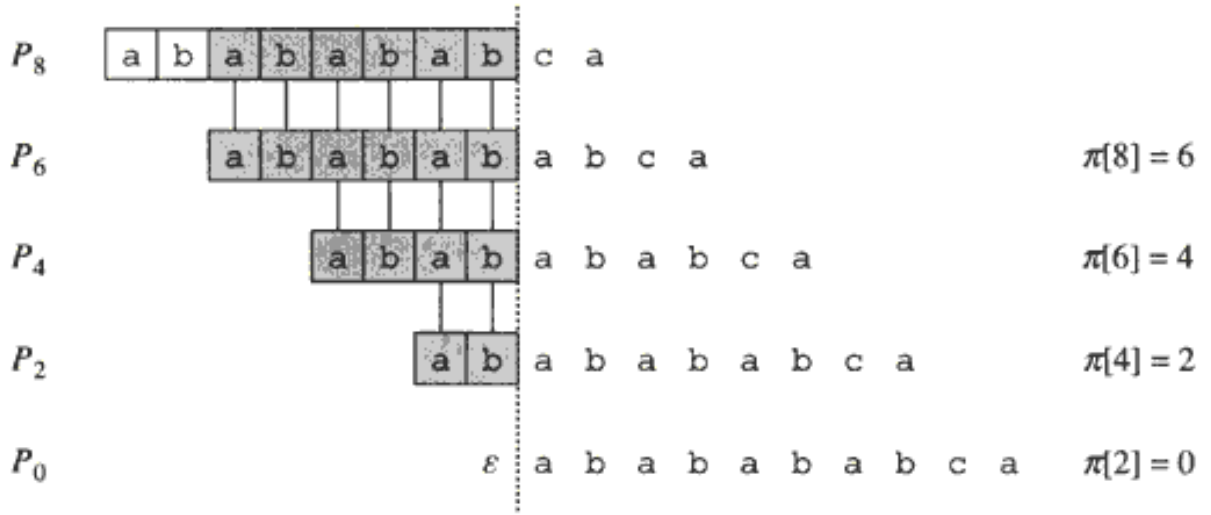


If we precompute prefix function of P (against itself), then whenever a mismatch occurs, the prefix function can determine which shift(s) are invalid and directly ruled out. So move directly to the shift which is potentially valid. However, there is no need to compare these characters again since they are equal.

Figure 32.10 The prefix function π. **(a)** The pattern $P = \texttt{ababaca}$ is aligned with a text T so that the first $q = 5$ characters match. Matching characters, shown shaded, are connected by vertical lines. **(b)** Using only our knowledge of the 5 matched characters, we can deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s + 2$ is consistent with everything we know about the text and therefore is potentially valid. **(c)** The useful information for such deductions can be precomputed by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of P_5 is P_3. This information is precomputed and represented in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$.

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

Figure 32.11 An illustration of Lemma 32.5 for the pattern $P = \texttt{abababacba}$ and $q = 8$. **(a)** The π function for the given pattern. Since $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, and $\pi[2] = 0$, by iterating π we obtain $\pi^*[8] = \{6, 4, 2, 0\}$. **(b)** We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_8; this happens for $k = 6$, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_8. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_8. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : k < q \text{ and } P_k \sqsupset P_q\} = \{6, 4, 2, 0\}$. The lemma claims that $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

The prefix function, Π

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5       do while k > 0 and P[k + 1] ≠ P[q]
 6            do k ← π[k]
 7       if P[k + 1] = P[q]
 8         then k ← k + 1
 9       π[q] ← k
10  return π
```

Example 1: compute Π for the pattern 'p' below:

P	a	b	a	b	a	c	a

Initially: m = length[p] = 7
Π[1] = 0
k = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 1: q = 2, k=0
Π[2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 2: q = 3, k = 0,
Π[3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step 3: q = 4, k = 1
Π[4] = 2

Step 4: q = 5, k =2

Π[5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: q = 6, k = 3

Π[6] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	

Step 6: q = 7, k = 1

Π[7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

After iterating 6 times, the prefix function computation is complete: →

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

The KMP Matcher

The KMP Matcher, with pattern P, string T and prefix function 'Π' as input, finds a match of P IN T.

KMP-Matcher(T,P)

```
1  n ← length[T]
2  m ← length[P]
3  Π ← Compute-Prefix-Function(P)
4  q ← 0                                   //number of characters matched
5  for i ← 1 to n                          //scan T from left to right
6      do while  q > 0 and P[q+1] != T[i]
7             do  q ← Π[q]                 //next character does not match
8         if P[q+1] = T[i]
9           then q ← q + 1                 //next character matches
10        if q = m                         //is all of p matched?
11          then print "Pattern occurs with shift" i – m
12               q ← Π[ q]                 // look for the next match
```

Note: KMP finds every occurrence of a P in T. That is why KMP does not terminate in step 12, rather it searches remainder of T for any more occurrences of P.

The KMP Matcher

```
KMP-MATCHER(T, P)
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                                  ▷ Number of characters matched.
 5  for i ← 1 to n                         ▷ Scan the text from left to right.
 6       do while q > 0 and P[q + 1] ≠ T[i]
 7            do q ← π[q]                  ▷ Next character does not match.
 8       if P[q + 1] = T[i]
 9         then q ← q + 1                 ▷ Next character matches.
10       if q = m                         ▷ Is all of P matched?
11         then print "Pattern occurs with shift" i − m
12              q ← π[q]                  ▷ Look for the next match.
```

Example 2:

T

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P

a	b	a	b	a	c	a

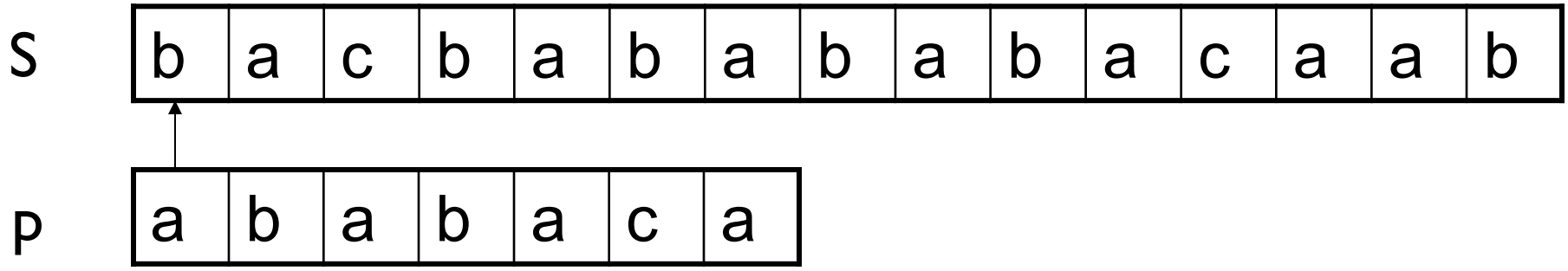
Let us execute the KMP algorithm to find whether 'P' occurs in 'T'.

For 'p' the prefix function, Π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

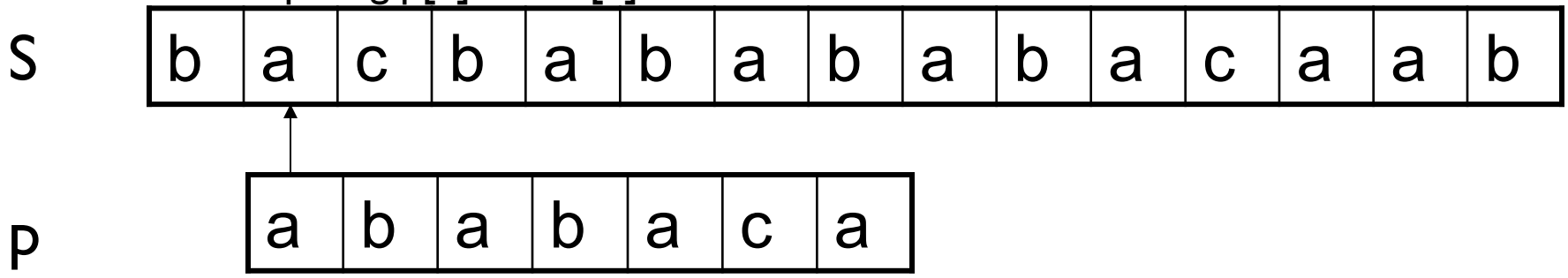
Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1, q = 0
comparing p[1] with S[1]



P[1] does not match with S[1]. 'p' will be shifted one position to the right.

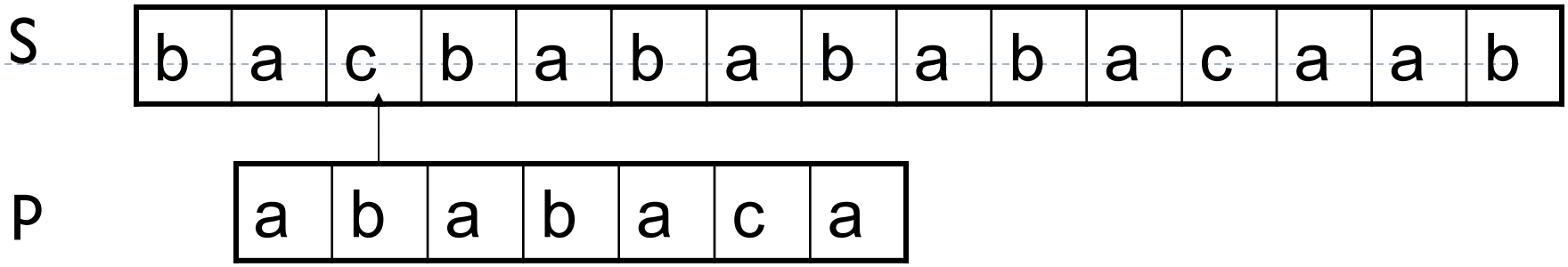
Step 2: i = 2, q = 0
comparing p[1] with S[2]



P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

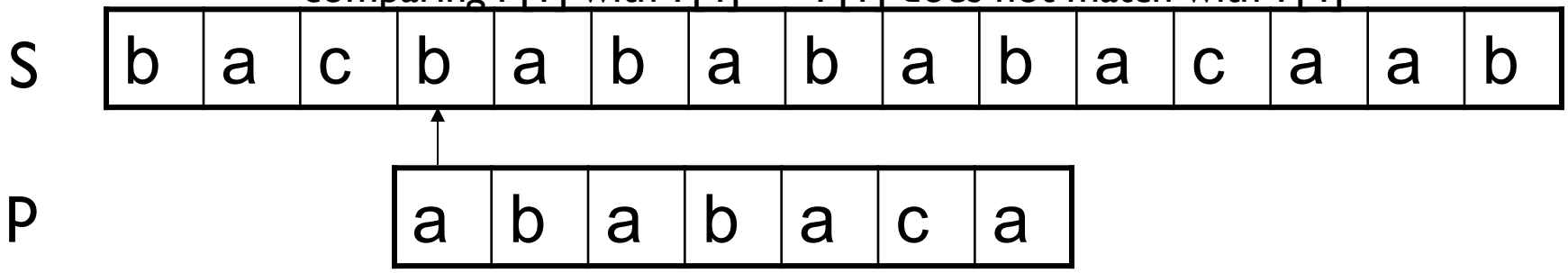
Comparing P[2] with T[3] P[2] does not match with T[3]



Backtracking on P, comparing P[1] and T[3]

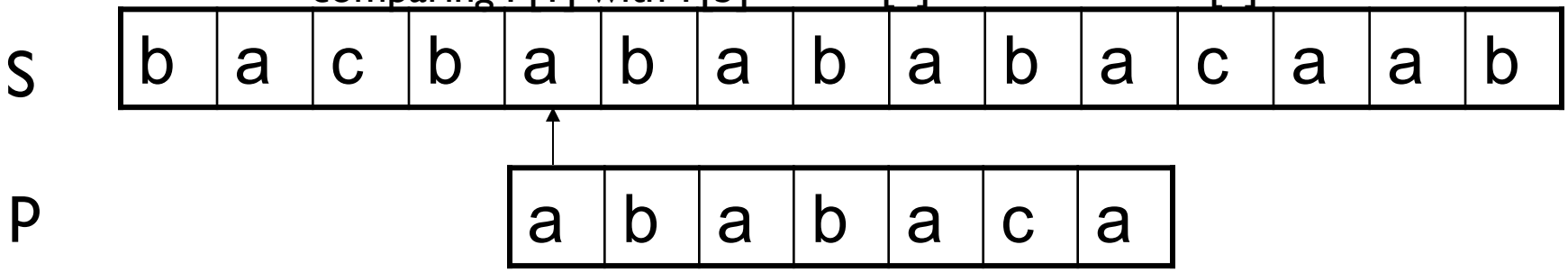
Step 4: i = 4, q = 0

comparing P[1] with T[4] P[1] does not match with T[4]



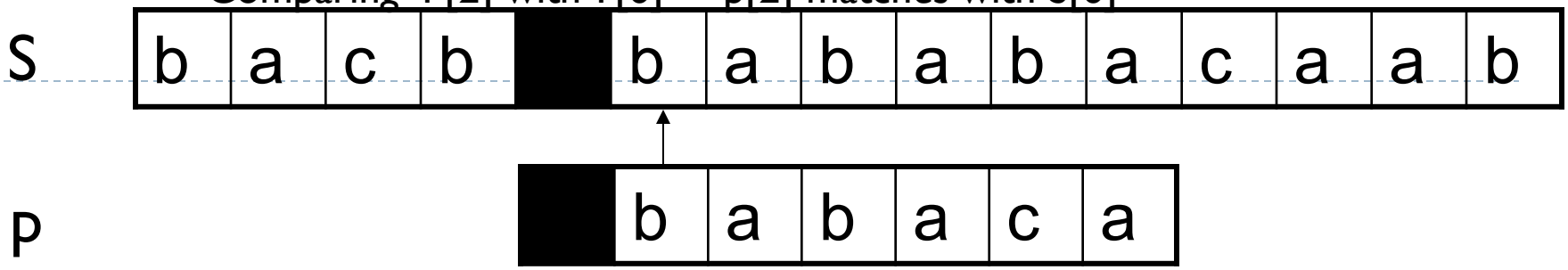
Step 5: i = 5, q = 0

comparing P[1] with T[5] P[1] matches with T[5]



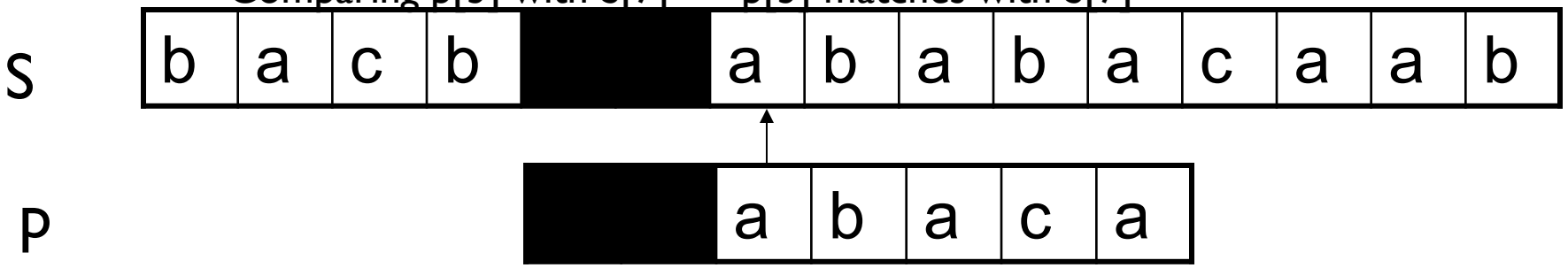
Step 6: i = 6, q = 1

Comparing P[2] with T[6] p[2] matches with S[6]



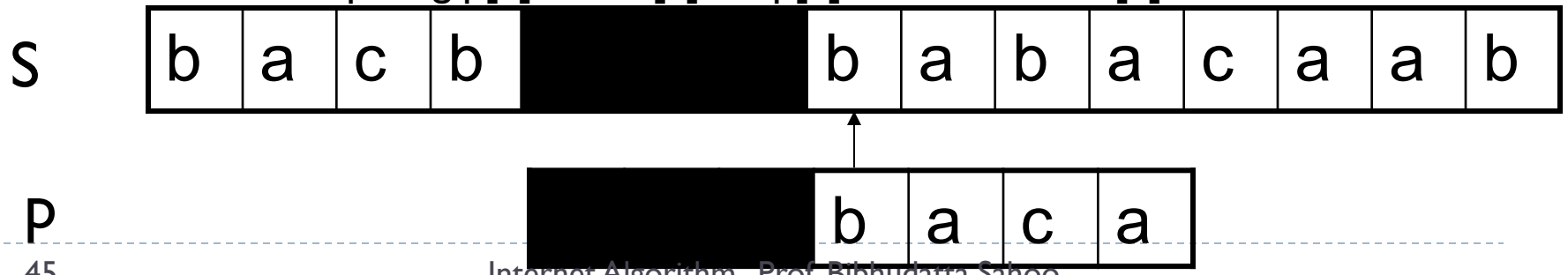
Step 7: i = 7, q = 2

Comparing p[3] with S[7] p[3] matches with S[7]

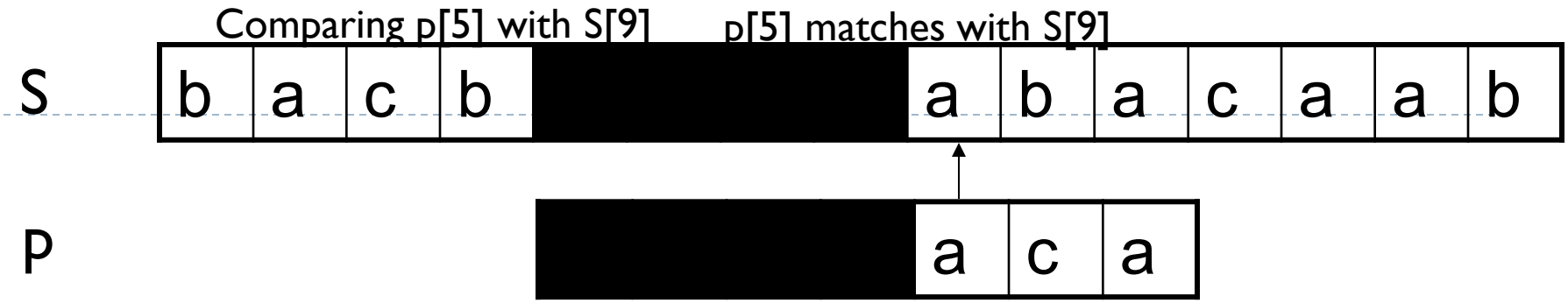


Step 8: i = 8, q = 3

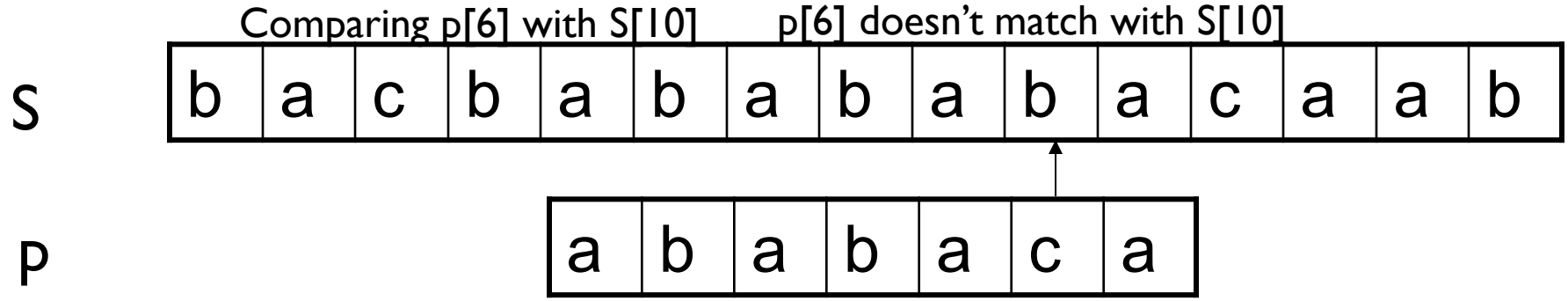
Comparing p[4] with S[8] p[4] matches with S[8]



Step 9: i = 9, q = 4

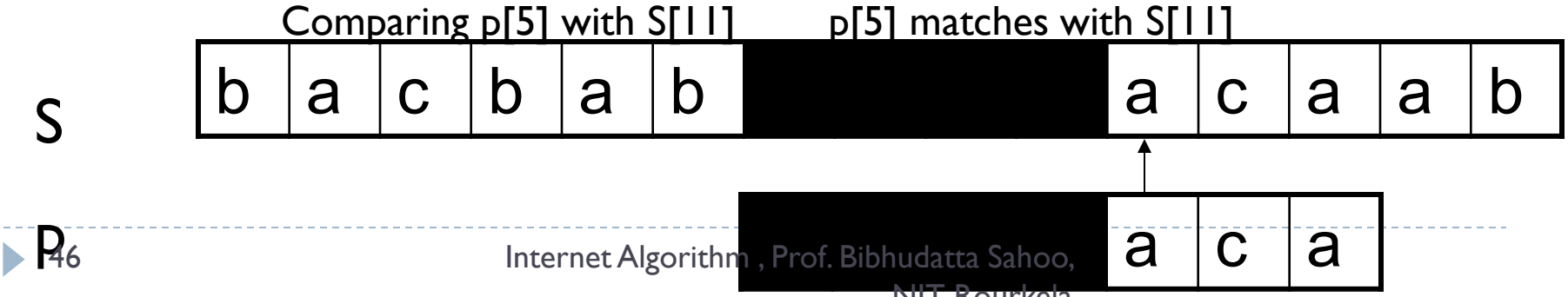


Step 10: i = 10, q = 5



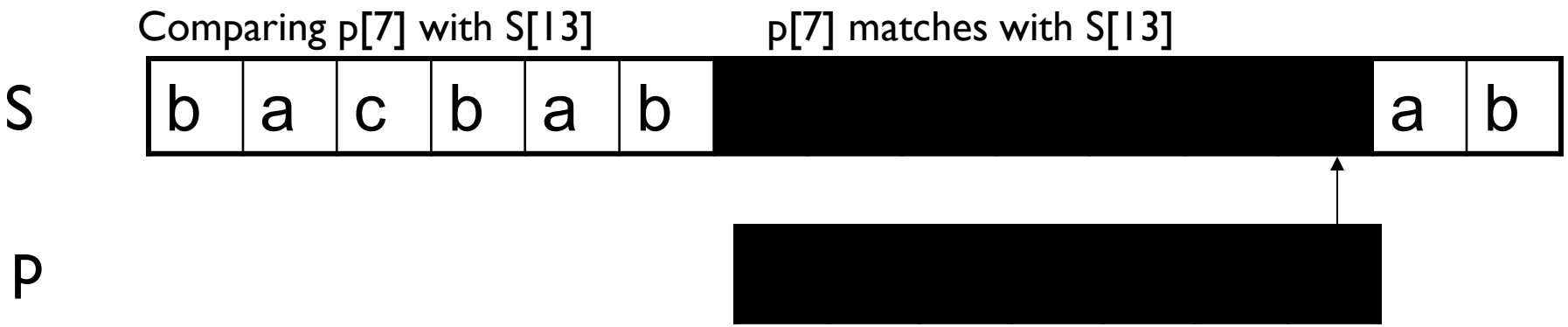
Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4



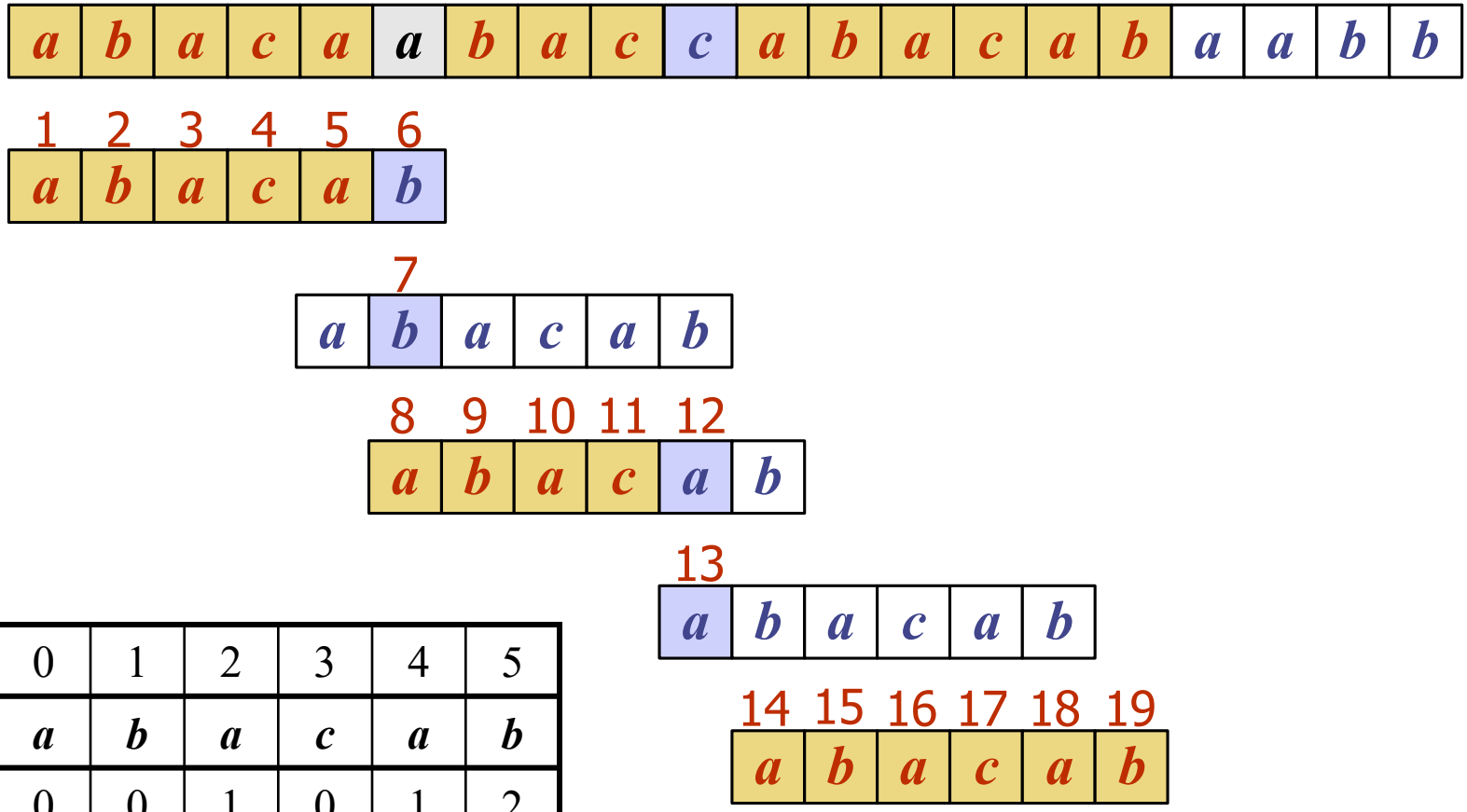
Step 12: i = 12, q = 5

Step 13: i = 13, q = 6



Pattern 'p' has been found to completely occur in string 'S'. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

Example: 3



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

Running - time analysis

```
    Compute-Prefix-Function (Π)
1   m ← length[p]                    //'p' pattern to be matched
2   Π[1] ← 0
3   k ← 0
4       for q ← 2 to m
5           do while k > 0 and p[k+1] != p[q]
6              do k ← Π[k]
7              If p[k+1] = p[q]
8                 then k ← k +1
9              Π[q] ← k
10      return Π
```

In the above pseudocode for computing the prefix function, the **for loop from** step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

```
    KMP Matcher
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0
5 for i ← 1 to n
6    do while  q > 0 and p[q+1] != S[i]
7          do  q ← Π[q]
8    if p[q+1] = S[i]
9       then q ← q + 1
10   if q = m
11     then print "Pattern occurs with shift" i – m
12             q ← Π[ q]
```

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S'. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

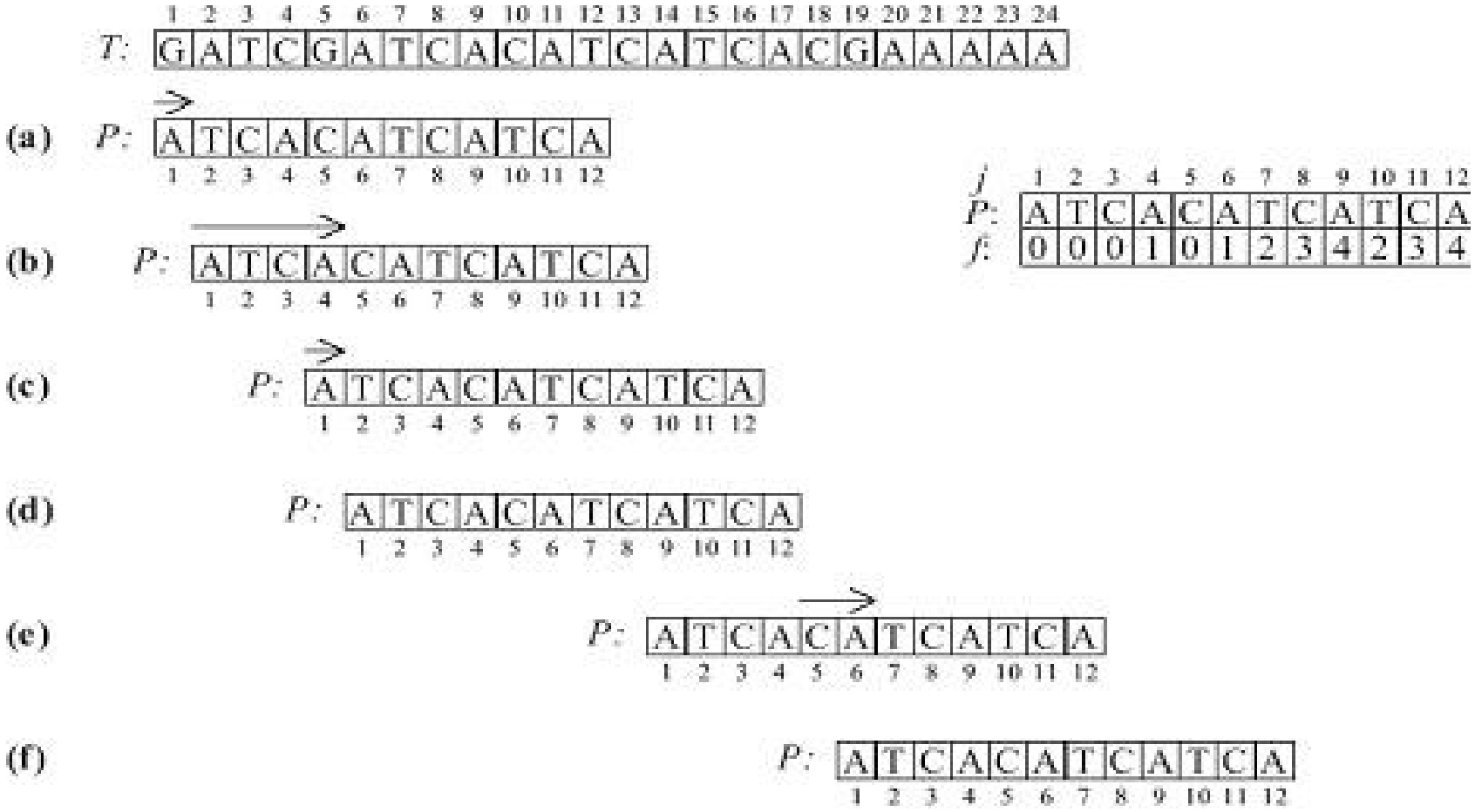
Analysis of KMP algorithm

- The running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$ and KMP-MATCHER $\Theta(m)+ \Theta(n)$.

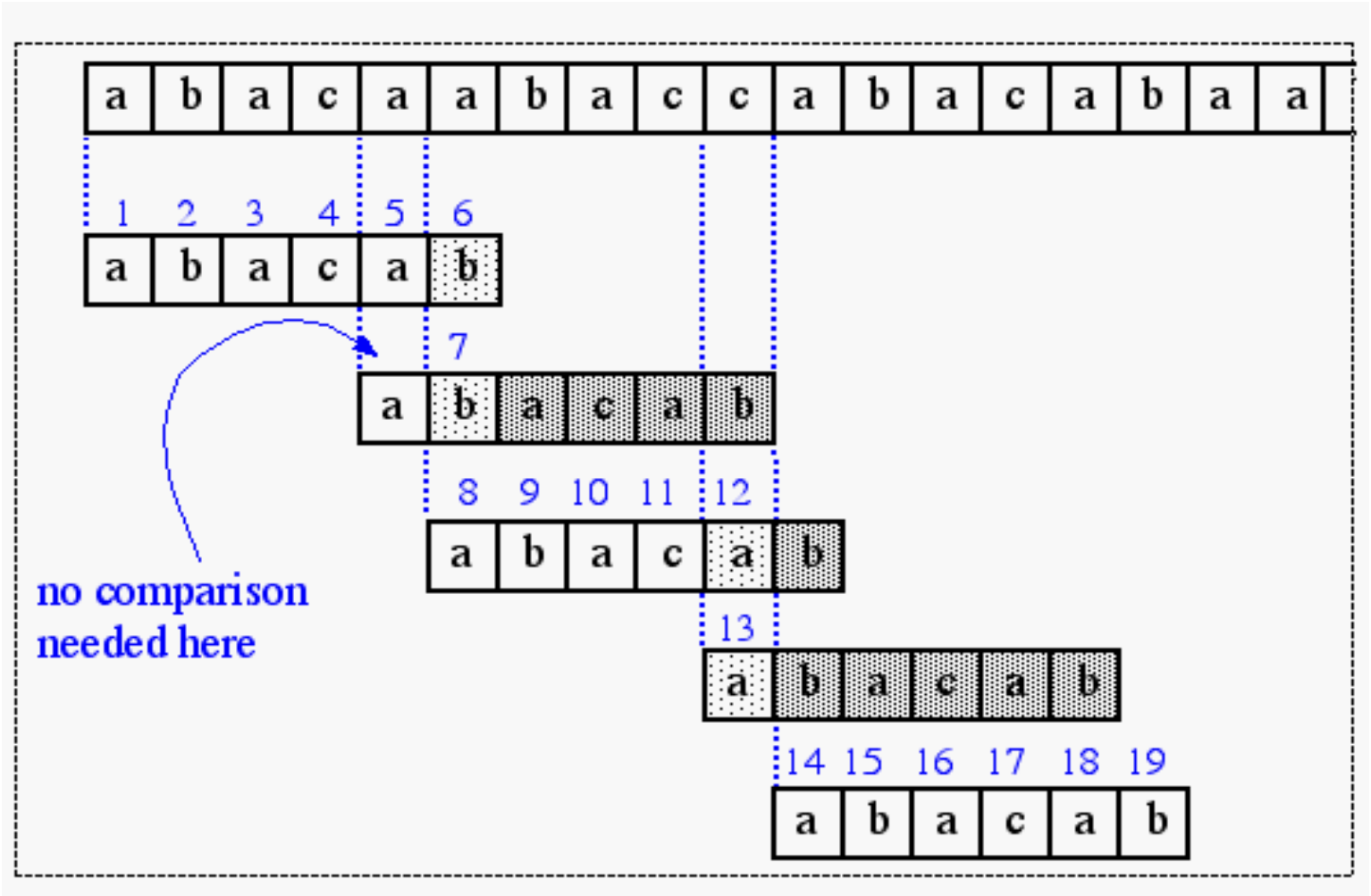
- Using amortized analysis (potential method) (for COMPUTE-PREFIX-FUNCTION):

  - Associate a potential of k with the current state k of the algorithm:
  - Consider codes in Line 5 to 9.
  - Initial potential is 0, line 6 decreases k since $\pi[k]<k$, k never becomes negative.
  - Line 8 increases k at most 1.
  - Amortized cost = actual-cost + potential-increase
  - =(repeat-times-of-Line-5+$O(1)$)+(potential-decrease-at-least the repeat-times-of-Line-5+O(1) in line 8)=$O(1)$.

Example:4


T: G A T C G A T C A C A T C A T C A C G A A A A A
(a) P: A T C A C A T C A T C A
(b) P: A T C A C A T C A T C A
(c) P: A T C A C A T C A T C A
(d) P: A T C A C A T C A T C A
(e) P: A T C A C A T C A T C A
(f) P: A T C A C A T C A T C A
j 1 2 3 4 5 6 7 8 9 10 11 12
P: A T C A C A T C A T C A
f: 0 0 0 1 0 1 2 3 4 2 3 4

Example: 5


a b a c a a b a c c a b a c a b a a
1 2 3 4 5 6
a b a c a b
7
a b a c a b
8 9 10 11 12
a b a c a b
13
a b a c a b
14 15 16 17 18 19
a b a c a b
no comparison
needed here

Example: 6

P = a b a b c

i	1	2	3	4	5
P[i]	a	b	a	b	c
π[i]	0	0	1	2	0

	1	2	3	4	5	6	7	8	9	10
T =	a	b	b	a	b	a	b	a	b	c

Start of 1st loop: q = 0, i = 1 **[a]**
2nd loop: q = 1, i = 2 **[b]**
3rd loop: q = 2, i = 3 **[b]** mismatch detected
4th loop: q = 0, i = 4 **[a]**
5th loop: q = 1, i = 5 **[b]**
6th loop: q = 2, i = 6 **[a]**
7th loop: q = 3, i = 7 **[b]**

8th loop: q = 4, i = 8 **[a]** mismatch detected
9th loop: q = 3, i = 9 **[b]**
10th loop: q = 4, i = 10 **[c]** match detected
Termination: q = 5

Theoretical evaluation

- KMP has a time complexity of O(n+m).
- The following all have time complexities of O(nm):
  - Brute Force, **Horspool** and the **Berry & Ravindran.**
- So theoretically the KMP is the best algorithm for string matching. Is this true in practice?

Exercise

1. Compute the prefix function for the pattern **ababbabbababbababbabb** when the alphabet is = {a, b}.
2. Explain how to determine the occurrences of pattern P in the text T by examining the function for the string PT (the string of length $m + n$ that is the concatenation of P and T).
3. Give a linear-time algorithm to determine if a text T is a cyclic rotation of another string T'. For example, **arc** and car **are** cyclic rotations of each other.
4. Give an efficient algorithm for computing the transition function for the string-matching automaton corresponding to a given pattern P. Your algorithm should run in time $O(m|\Sigma|)$. (*Hint*: Prove that $\delta(q, a) = \delta([q], a)$ *if* $q = m$ *or* $P[q + 1] \neq a$.)

The Boyer-Moore's pattern matching algorithm

Right-to-left scan

Bad character shift rule

Good suffix shift rule

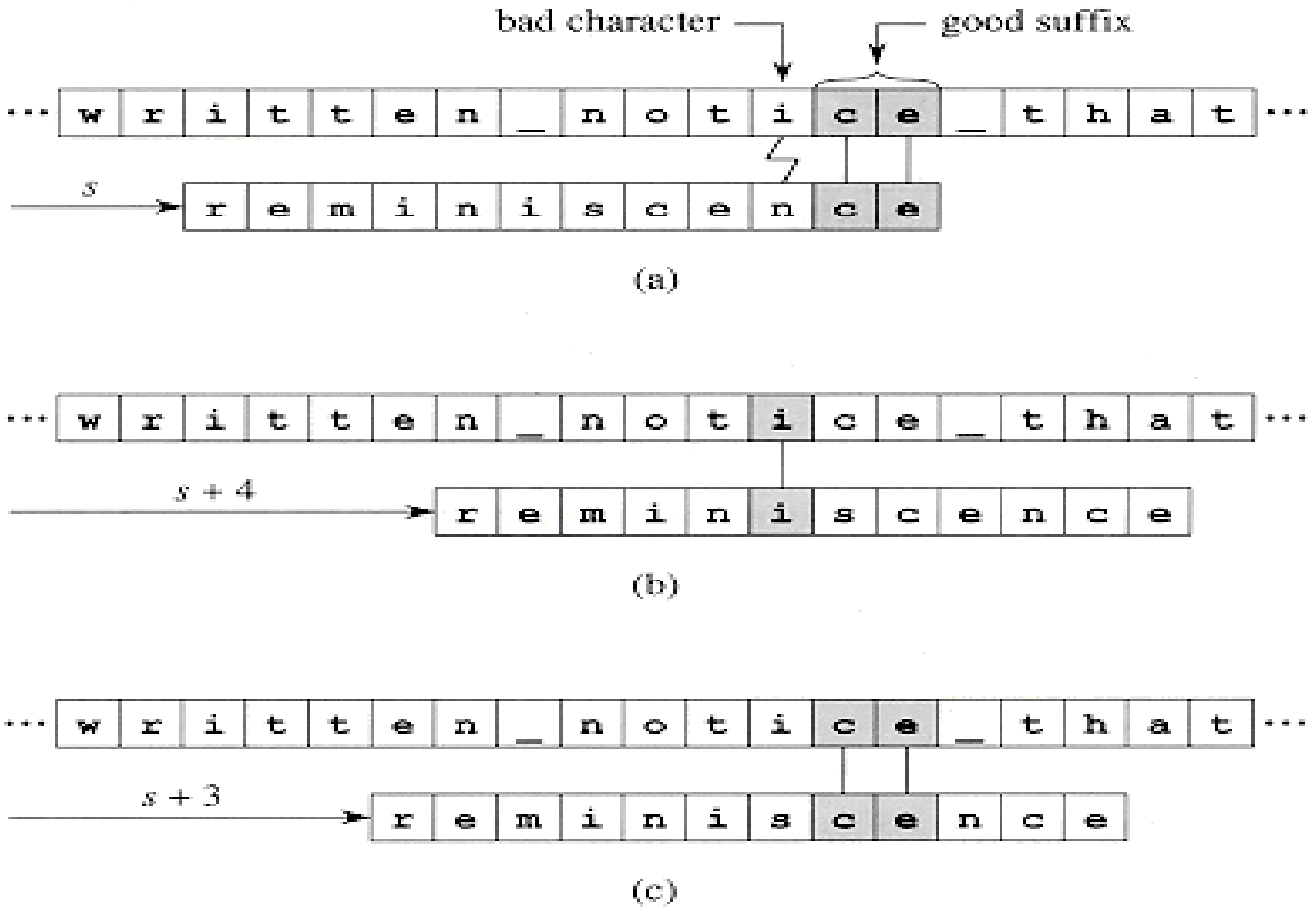
Boyer-Moore Algorithm

- The Boyer-Moore algorithm was proposed by Boyer and Moore in 1977.
- The worst time complexity of this algorithm is no better than that of KMP algorithm, but it is more efficient in practice than the KMP algorithm
- The Boyer-Moore algorithm slides P from left to right; however it compares P and T from right to left
- i.e., P[m] will first compare with T[i]. If they match it then compares P[m-1] with T[i-1], etc.
- Else, It slides P to right, and compare P[m] with T

Concept of Boyer-Moore's pattern matching

- Boyer-Moore's pattern matching algorithm does a couple things differently than the brute force algorithm:
-
- 1) Instead of checking for a string match from the beginning of the string, check for the match from the end. This allows jumps of more than one space when hitting a discrepancy.
-
- 2) Instead of always moving ahead one space after you have eliminated a string choice, jump several spaces if you can.

- If the pattern P is relatively long and the alphabet is reasonably large, then an algorithm due to Robert S. Boyer and J. Strother Moore is likely to be the most efficient string-matching algorithm.


bad character
good suffix
w r i t t e n _ n o t i c e _ t h a t
s
r e m i n i s c e n c e
(a)
w r i t t e n _ n o t i c e _ t h a t
s + 4
r e m i n i s c e n c e
(b)
w r i t t e n _ n o t i c e _ t h a t
s + 3
r e m i n i s c e n c e
(c)

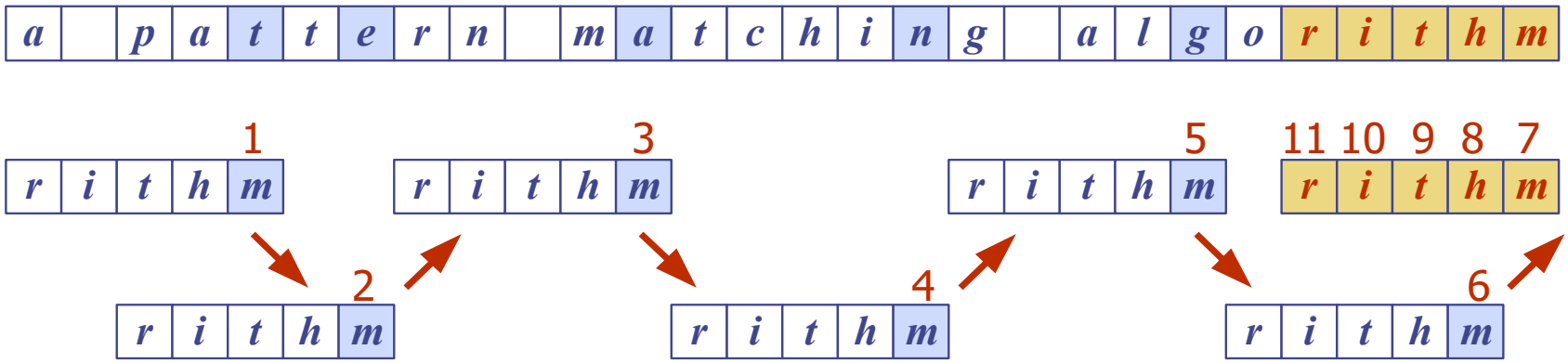
Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics

  Looking-glass heuristic (right-to-left matching): Compare P with a subsequence of T moving backwards

  Character-jump heuristic (bad character shift rule): When a mismatch occurs at $T[i] = c$
  - If P contains c, shift P to align the last occurrence of c in P with $T[i]$
  - Else, shift P to align $P[0]$ with $T[i+1]$

- Example



Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function L mapping Σ to integers, where $L(c)$ is defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- Example:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	-1

- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

Last Function: L(c)

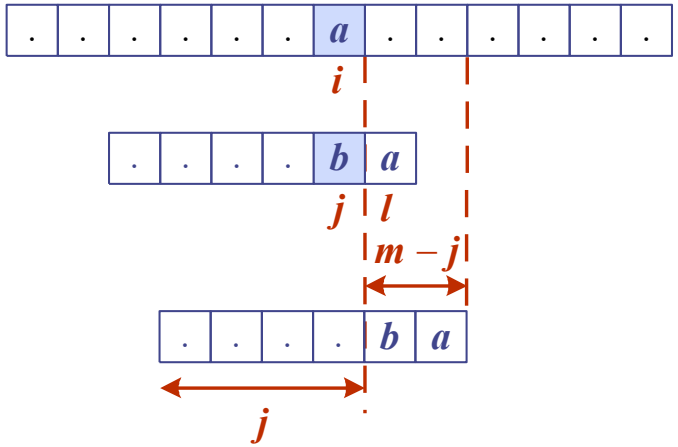
- The function last(c) that takes a character c from the alphabet and specifies how far may shift the pattern P if a character equal to c is found in the text that does not match the pattern

$$last(c) = \begin{cases} \text{index of the last occurrence of c in pattern P} & \text{if c is in P} \\ -1 & otherwise \end{cases}$$

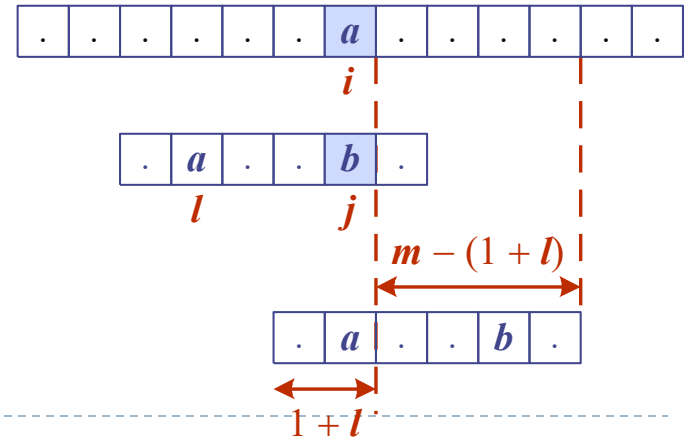
The Boyer-Moore Algorithm

```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ)
    i ← m − 1
    j ← m − 1
    repeat
        if T[i] = P[j]
            if j = 0
                return i { match at i }
            else
                i ← i − 1
                j ← j − 1
        else
            { character-jump }
            l ← L[T[i]]
            i ← i + m − min(j, 1 + l)
            j ← m − 1
    until i > n − 1
    return −1 { no match }
```

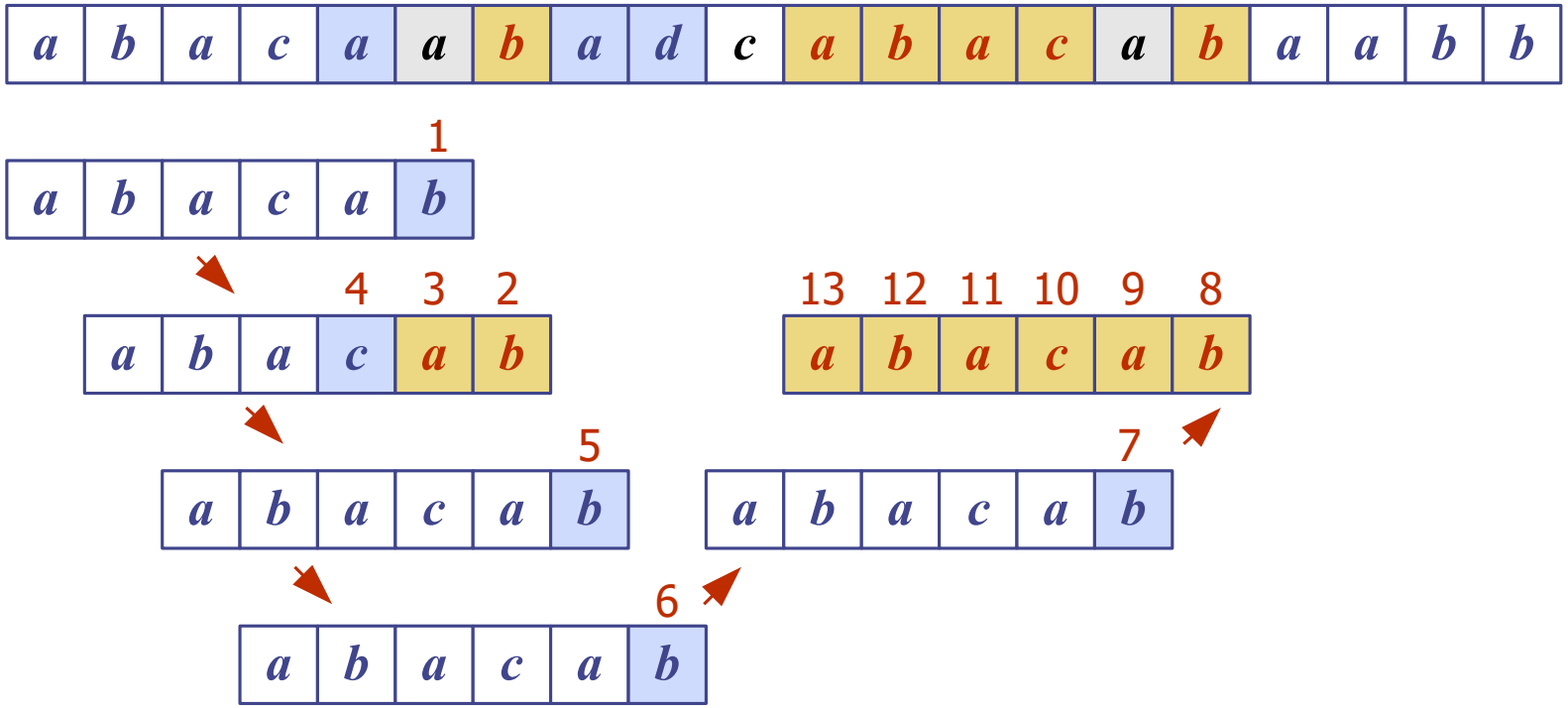
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

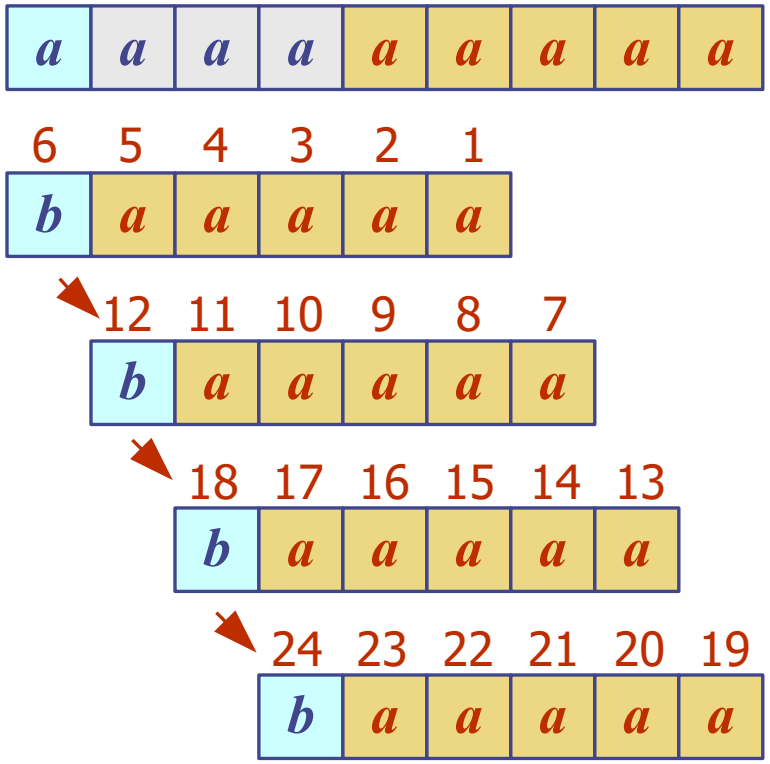


Example


a b a c a a b a d c a b a c a b a a b b
1
a b a c a b
4 3 2
a b a c a b
13 12 11 10 9 8
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text



BOYER_MOORE_MATCHER

- **BOYER_MOORE_MATCHER (T, P)**
- **Input:** Text with n characters and Pattern with m characters
  Output: Index of the first substring of T matching P
- Compute function last
- i ← m-1
- j ← m-1
- Repeat
- If P[j] = T[i] then
- if j=0 then
- return i // we have a match
- else
- i ← i -1
- j ← j -1
- else
- i ← i + m - Min(j, 1 + last[T[i]])
- j ← m -1
- until i > n -1
- Return "no match"

Boyer-Moore's algorithm

- The computation of the last function takes $O(m+|\sum|)$ time and actual search takes $O(mn)$ time. Therefore the worst case running time of Boyer-Moore algorithm is $O(nm + |\sum|)$.
- Implies that the worst-case running time is quadratic, in case of n = m, the same as the naïve algorithm.
- Boyer-Moore algorithm is extremely fast on large alphabet (relative to the length of the pattern).
- The payoff is not as for binary strings or for very short patterns.
- For binary strings Knuth-Morris-Pratt algorithm is recommended.
- For the very shortest patterns, the naïve algorithm may be better.

Rabin–Karp algorithm or Karp–Rabin algorithm

Rabin–Karp algorithm

- the Rabin–Karp algorithm or Karp–Rabin algorithm is a string-searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text.

- The Rabin-Karp string searching algorithm calculates a hash value for the pattern, and for each k-character subsequence of text to be compared.

- If the hash values are unequal, the algorithm will calculate the hash value for next k-character sequence.

-

- If the hash values are equal, the algorithm will compare the pattern and the k-character sequence.

- There is only one comparison per text subsequence, and character matching is only needed when hash values match.

Multiple pattern matching

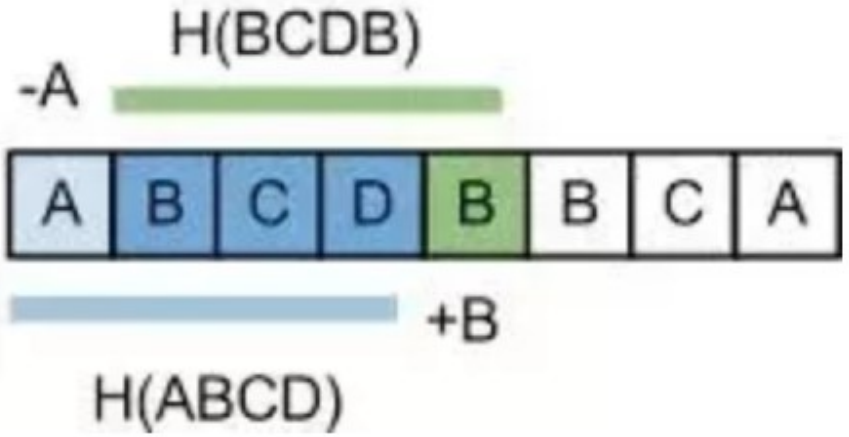
Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

Pattern Found at 0, 9 and 12

What is String-Hashing?

- String hashing is the way to convert a string into an integer known as a hash of that string.
- An ideal hashing is the one in which there are minimum chances of collision (i.e 2 different strings having the same hash).
- The hashing function should be easy to calculate. we should be able to calculate the hash value for a string in linear time.



Hash Function

- The simplest hashing functions used is the sum of ASCII codes of the letters in the string.
- For simplicity, we'll use *int(x)* to denote the ASCII code of x.
- Since we can quickly calculate the hash value of a substring using prefix sums, this hashing function is a valid choice.
- This case, when two distinct strings have the same hash value, is called a collision.
- As a rule, the fewer collisions a hashing function causes, the better it is. Our hashing function has lots of collisions since it doesn't take into account the order and position of letters. Later, we'll discuss better hashing functions.
- Check the following implementation of a hash structure using the discussed hashing function:

Hash Function

Perhaps one of the simplest hashing functions we can use is the sum of ASCII codes of the letters in the string. For simplicity, we'll use $int(x)$ to denote the ASCII code of x. Formally:

$$H(s[0...n-1]) = int(s[0]) + int(s[1]) + ... + int(s[n-1])$$

Since we can quickly calculate the hash value of a substring using prefix sums, this hashing function is a valid choice.

Let's look at a few examples, assuming $s = "aabbab"$:

- $H(s[2...4]) = H("bba") = 98 + 98 + 97 = 293$
- $H(s[3...5]) = H("bab") = 98 + 97 + 98 = 293$

Notice that $H(s[2...4]) = H(s[3...5])$ even though $s[2...4] \neq s[3...5]$. **This case, when two distinct strings have the same hash value, is called a collision.**

Hash Function

Algorithm 2: Hash structure

Data: s: String for which we want to calculate hash values
prefix: Prefix sum of ASCII codes in string s

Function `Init(s)`:
    n ← Length(s);
    prefix[0] ← *int*(s[0])
    for i ← 1 **to** n − 1 **do**
        prefix[i] ← prefix[i − 1] + *int*(s[i]);
    end
end
Function `getHash(L, R)`:
    if L = 0 **then**
        return prefix[R];
    end
    return prefix[R] − prefix[L − 1];
end

Hash Function

The first function $Init$ performs the precalculation on the given string. We iterate over the string s and calculate the prefix sum of the ASCII values of the string's characters.

The second function $getHash$ is used to calculate the hash value of a given range $[L, R]$. To do this, we return the prefix sum at the end of the range, after subtracting the prefix sum of the beginning of the range. Therefore, we get the sum of values in the range $[L, R]$.

The complexity of the precalculation function is $O(n)$, where n is the length of the string. Also, **the complexity of the hashing calculation function is** $O(1)$.

How Rabin-Karp works

- Let characters in both arrays T and P be digits in radix-Σ notation. (Σ = (0,1,...,9)
- Let p be the value of the characters in P
- 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.
- Compute (T[s+1, .., s+k] mod q) for s = 0 .. n-k
- Test against P only those sequences in T having the same (mod q) value
- (T[s+1, .., s+k] mod q) can be incrementally computed by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.

A Rabin-Karp example

- Given T = 31415926535 and P = 26
- We choose q = 11
- The value of **P mod q** = 26 mod 11 = **4**

31 mod 11 = 9 not equal to 4

14 mod 11 = 3 not equal to 4

41 mod 11 = 8 not equal to 4

Rabin-Karp example continued

15 mod 11 = 4 equal to 4 -> spurious hit

59 mod 11 = 4 equal to 4 -> spurious hit

92 mod 11 = 4 equal to 4 -> spurious hit

26 mod 11 = 4 equal to 4 -> an exact match!!

65 mod 11 = 10 not equal to 4

Rabin-Karp example continued …

53 mod 11 = 9 not equal to 4

35 mod 11 = 2 not equal to 4

As we can see, when a match is found, further testing is done to insure that a match has indeed been found.

Rabin-Karp Algorithm

Algorithm 3: Rabin-Karp string search algorithm

```
Data: P: The pattern to look for
      T: The text to look in
      hashT, hashP: Hash structures for strings T, P (Algorithm 2)
Result: Returns the number of occurrences of P in T
answer ← 0;
n ← length(T);
k ← length(P);
for i ← 0 to n − k do
    textHash ← hashT.getHash(i, i + k − 1);
    patternHash ← hashP.getHash(0, k − 1);
    if textHash = patternHash then
        valid ← true;
        for j ← 0 to k − 1 do
            if T[i + j] ≠ P[j] then
                valid ← false;
                break;
            end
        end
        if valid = true then
            answer ← answer + 1;
        end
    end
end
return answer;
```

Rabin-Karp Algorithm

Assume that we're looking for the pattern P in the text T. We're currently at location i in T, and we need to check if $T[i...i+k-1]$ is equal to $P[0...k-1]$, where k is equal to the length of P.

If $H(P[0...m-1]) \neq H(T[i...i+m-1])$ then we can assume that the two strings aren't equal, and skip the matching attempt at this location. Otherwise, **we should attempt to match the strings character by character.**

The worst-case complexity of this algorithm is still $O(n \cdot k)$, where n is the length of the text, and k is the length of the pattern.

However, **if we use a good hashing function, the expected complexity would become $O(n + k \cdot t)$, where t is the number of matches**.

The reason behind this is that a good hashing function would rarely cause collisions. Therefore, we'd rarely need to compare two substrings when there is no match. Furthermore, **if we only want to check if the pattern exists or not, the complexity would become $O(n + k)$**, because we can break after the first occurrence.

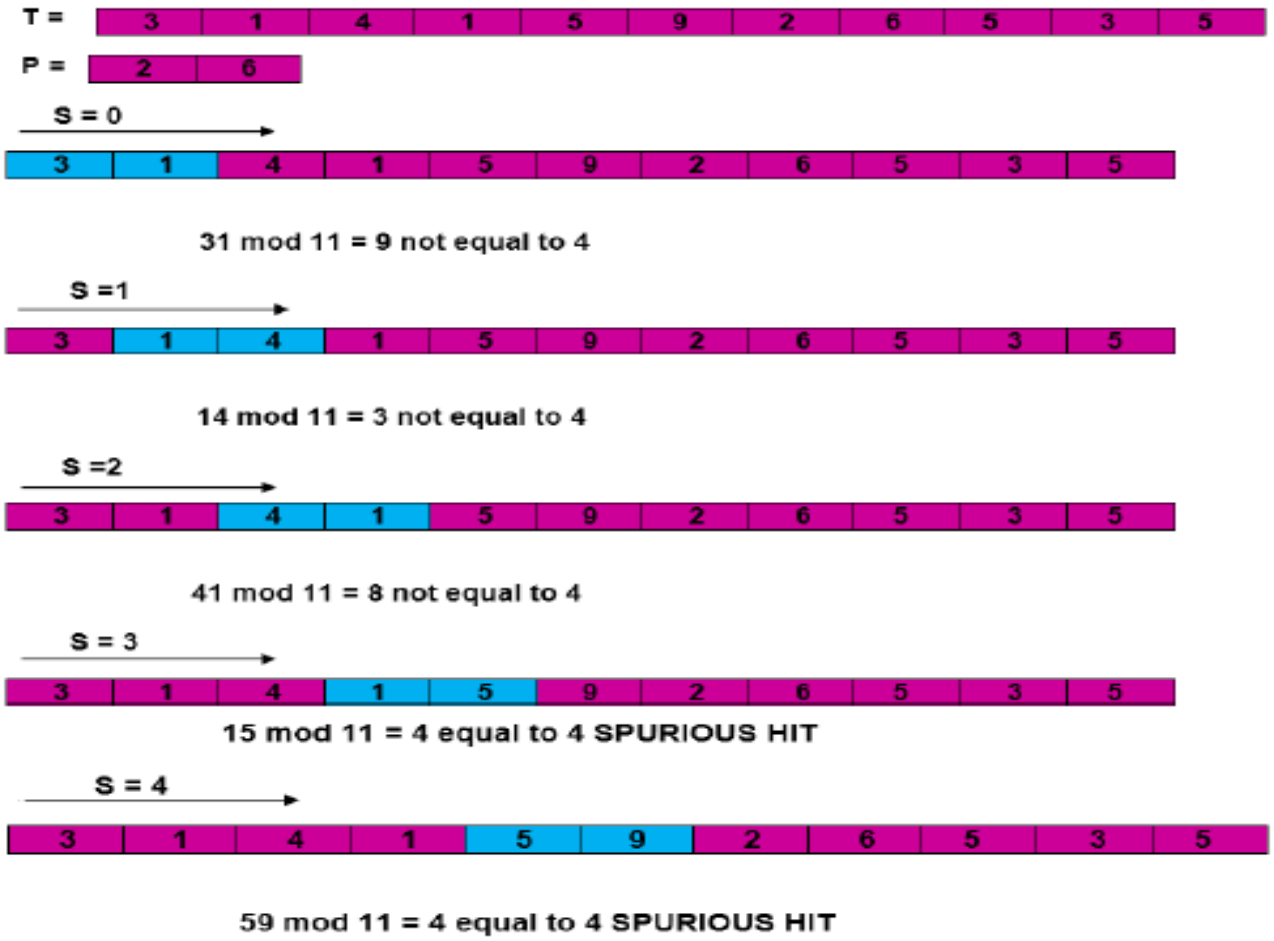
Example: Rabin-Karp Algorihm

- T = 31415926535..... & P = 26

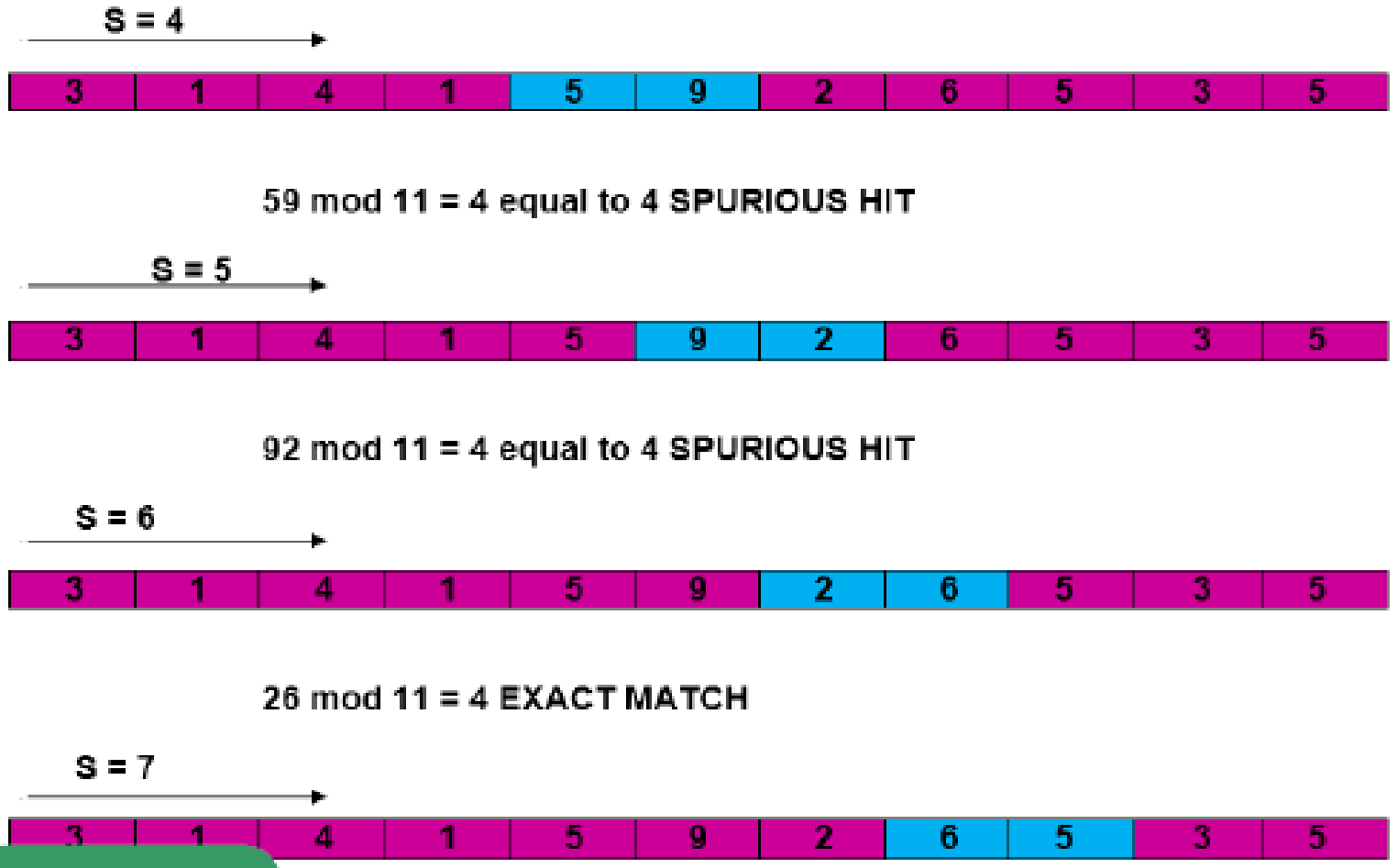
  Here length of text is 11 so q = 11, hence **P mod q** = 26 mod 11 = 4

- Now find the exact match of P mod Q...

- **Solution:**



Example: Rabin-Karp Algorihm



⇧ SCROLL TO TOP

Example: Rabin-Karp Algorihm

65 mod 11 = 10 not equal to 4

53 mod 11 = 9 not equal to 4

35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

Summary

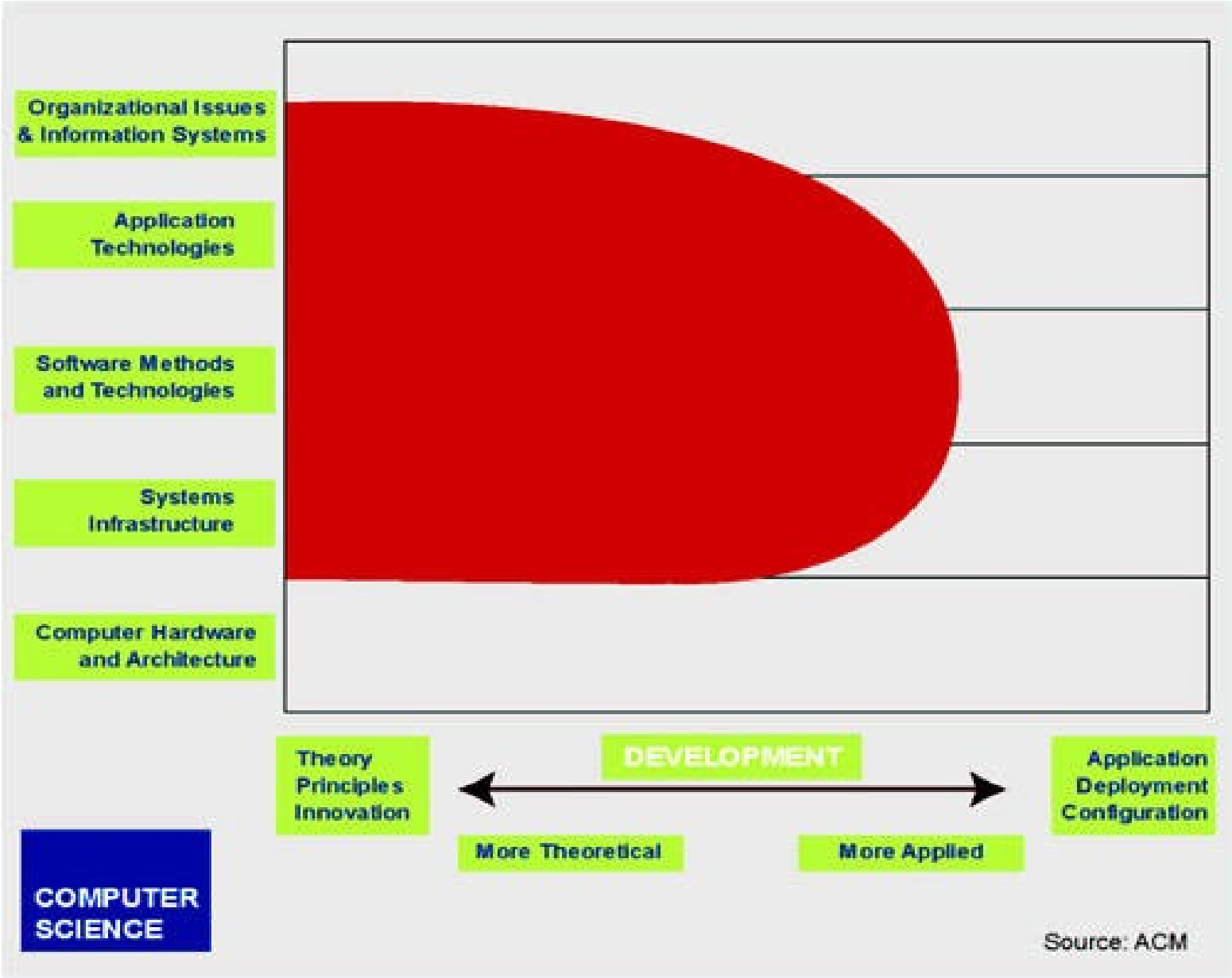
- The running time of the Rabin-Karp algorithm in the worst-case scenario is O(n-k+1)k but it has a good average-case running time.
- If the expected number of valid shifts is small O(1) and the prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time O(n+k) plus the time to required to process spurious hits.

Reference: Multiple pattern matching

- https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/


Thank you for your attention

Computer Science



Exercises

- Describe the Boyer-Moore string search algorithm, stating what the "index" table for a search string will contain. Trace the algorithm for the following example, and state how many character comparisons will be required.

S1= "susan likes the teletubbies"

S2= "teletubbies"

- The Boyer-Moore algorithm, as described in the lecture, will behave oddly for strings with lots of repeated characters. Consider the following case, and suggest a small modification to the algorithm to fix the problem:

S1="000010000"
S2="1000"

Exercises

1. Make up an example search string and document where KMP will do better than Boyer-Moore, and an example where Boyer-Moore will do better than KMP.
2. Give a regular expression that will match “yes” “Yes”, “yeah”, “Yeah”, “yes!”, “Yeah!!!” and similar (i.e., the four main examples followed by any number of exclamation marks).
3. Draw a finite-state machine representing the same pattern.
4. Show how this machine may be represented using two arrays, and trace through the pattern matching algorithm for this pattern and the string “Yes!”

Optional) Approximate String Matching

- String matching algorithms are an important application used in editors and word processors. Spell-checker programs must flag a word and give suggestions for its correct spelling, hence these programs need to match strings approximately. That is, they must be able to check how far apart two strings are from each other. We say that two strings are distance n apart if there is a way to change one into another by a combination of n insertions, deletions or changes of single letters.
- Let $s = s_1 s_2 \ldots s_m$ be a search string of length m and $t = t_1 t_2 \ldots t_m$ be text string of length p. Let $C(i, j)$ be the minimum distance between $s_0 s_1 \ldots s_i$ and a segment of t ending at t_j.
- **Explain why**: $C(0, j) = 0$, $C(i, 0) = i$, and when $i>0$ and $j>0$
- $C(i, j) = min \{C(i-1, j)+1, C(i, j-1)+1, C(i-1, j-1)\}$ if $s_i = t_j$
- $C(i, j) = min \{C(i-1, j)+1, C(i, j-1)+1, C(i-1, j-1)+1\}$ if $s_i \neq t_j$
- Write code for a dynamic programming algorithm based on part (a), that takes a search string and a text string, and finds the j for which $C(i, j)$ is minimum.