

Design & Analysis of Algorithms

Saman Riaz (PhD)
Associate Professor
RSCI, Lahore

Lecture # 09

STRING MATCHING

Substring Search

Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

↑
match

Substring search applications

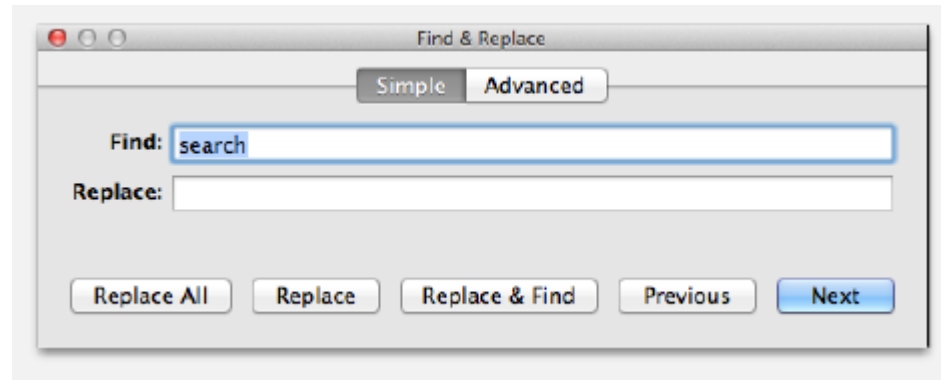
Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

match



Substring Search

Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

match

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.

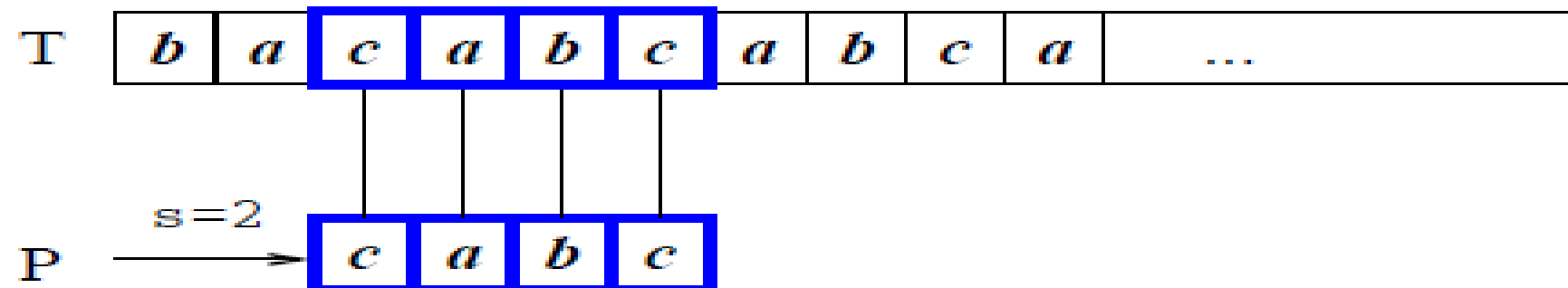


<http://citp.princeton.edu/memory>

Given a **text** array $T[1 \dots n]$ and a **pattern** array $P[1 \dots m]$ such that the elements of T and P are characters taken from alphabet Σ . e.g., $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \dots, z\}$.

The **String Matching Problem** is to find *all* the occurrence of P in T .

A pattern P occurs with **shift** s in T , if $P[1 \dots m] = T[s + 1 \dots s + m]$. The String Matching Problem is to find all values of s . Obviously, we must have $0 \leq s \leq n - m$.



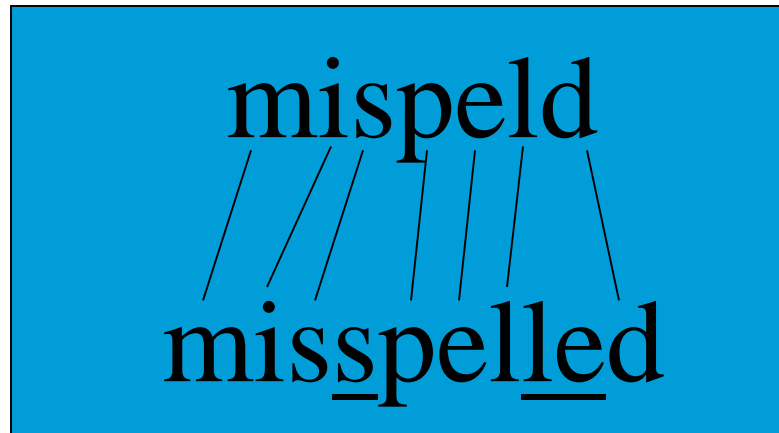
Exact String Matching Problem

- ***P* occurs with shift *s* in *T* or *P* occurs beginning at position *s+1* in text *T* if**
$$0 \leq s \leq n-m \text{ and } T[s+1..s+m] = P[1..m]$$

i.e., $T[s+j] = P[j]$, for $1 \leq j \leq m$
- If ***P* occurs with shift *s***, *s* is a valid shift; otherwise we call *s* an invalid shift.
- **String matching problem:** finding all valid shifts with which a given pattern *P* occurs in a given text *T*

Approximate String Matching

- Given a text to search (T) and a pattern to look for (P).
- Find all of the occurrences of P that exist in T, allowing a defined number of errors to be present in the matches



Notation & terminology

- Σ^* = set of all finite-length strings from alphabet Σ
- ε = **empty string** (zero length string)
- $|x|$ = length of string x
- xy = concatenation of strings x and y , length is $|x|+|y|$
- w is a **prefix** of x , $w \preceq x$, if $x=wy$, where $y \in \Sigma^*$
- w is a **suffix** of x , $w \preceq^r x$, if $x=yw$, where $y \in \Sigma^*$

Notation & terminology (cont'd)

- ε is both a suffix and a prefix of every string
 - $ab \sqsubseteq abcca$
 - $cca \sqsupseteq abcca$
- We denote k-character prefix $P[1..k]$ of the pattern $P[1..m]$ by P_k .
- So $P_0 = \varepsilon$ and $P_m = P = P[1..m]$
- Similarly, k-character prefix of text T as T_k
- String matching prob.: find all shifts in the range $0 \leq s \leq n-m$ such that $P \sqsupseteq T_{s+m}$

Overlapping Suffices

- x , y and z are strings and $x \sqsupseteq z$ and $y \sqsupseteq z$ if:
 - $|x| \leq |y|$ then $x \sqsupseteq y$
 - $|x| \geq |y|$ then $y \sqsupseteq x$
 - $|x| = |y|$ then $x = y$

String Comparison

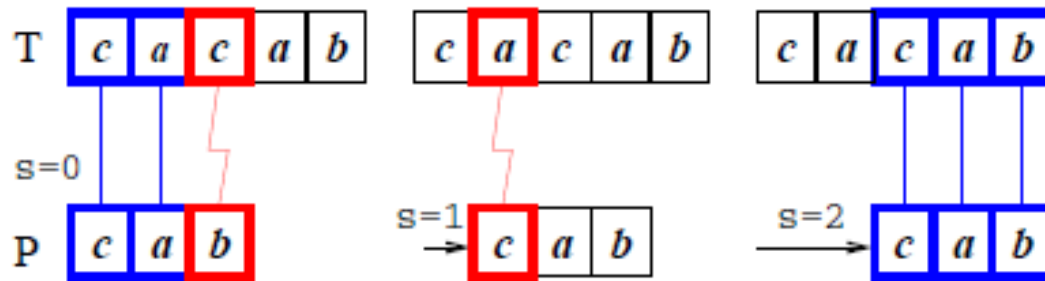
- The test “ $x=y$ ” is assumed to take time $\Theta(t+1)$ where t is the length of the longest string z such that $z \sqsubseteq x$ and $z \sqsubseteq y$.
- $t=0$, when $z = \varepsilon$

String matching algorithms

- <http://www-igm.univ-mlv.fr/~lecroq/string/index.html>
 - There are about 35 algorithms for exact string matching, some of them are:
 - Brute Force algorithm (Naïve Algorithm)
 - Boyer-Moore algorithm

Naïve string matching algorithm

Initially, P is aligned with T at the first index position. P is then compared with T from **left-to-right**. If a mismatch occurs, "slide" P to *right* by 1 position, and start the comparison again.



- Takes $\Theta((n-m+1)m)$ time.
- If $m=n/2$, it becomes $\Theta(n^2)$



Naïve string matching Algo. (pseudocode)

The naive algorithm 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 .

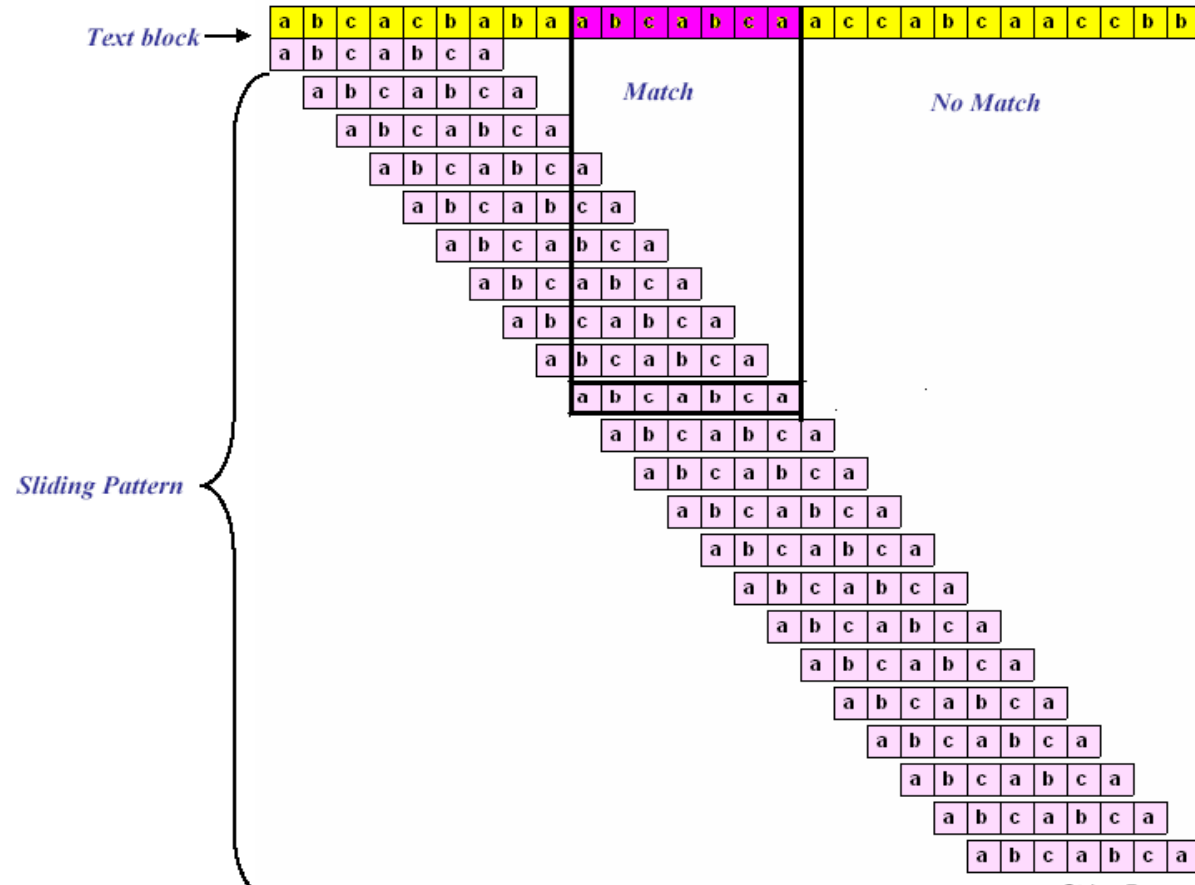
NAIVE-STRING-MATCHER(T, P)

```
1   $n \leftarrow \text{length}[T]$ 
2   $m \leftarrow \text{length}[P]$ 
3  for  $s \leftarrow 0$  to  $n - m$ 
4      do if  $P[1..m] = T[s + 1..s + m]$ 
5          then print "Pattern occurs with shift"  $s$ 
```

Naïve string matching Algo.

```
MATCH( $P, T$ )  
 $n \leftarrow \text{length}[T]$    
 $m \leftarrow \text{length}[P]$    
for  $i \leftarrow 1$  to  $n - m + 1$  do  
  for  $j \leftarrow 1$  to  $m$  do  
    if  $T[i + j] = P[j]$   
      then if  $j = m$   
        then return  $i$   
      else break inner loop  
return  $-1$ 
```


Naïve String Matching Algorithm



Analysis

- The code consists of two nested loops

```
for  $i = 1$  to  $n - m + 1$  do  
    for  $j = 1$  to  $m$  do
```

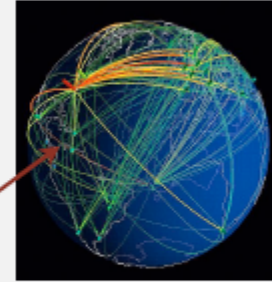
- The outer loop iterates $n - m + 1$ times. The inner loop iterates m times. Thus, in worst case, altogether $m(n - m + 1)$ iterations are performed.
- Thus, running time for the algorithm is $O(m(n - m + 1))$, which is simplified as $O(mn)$

Backup

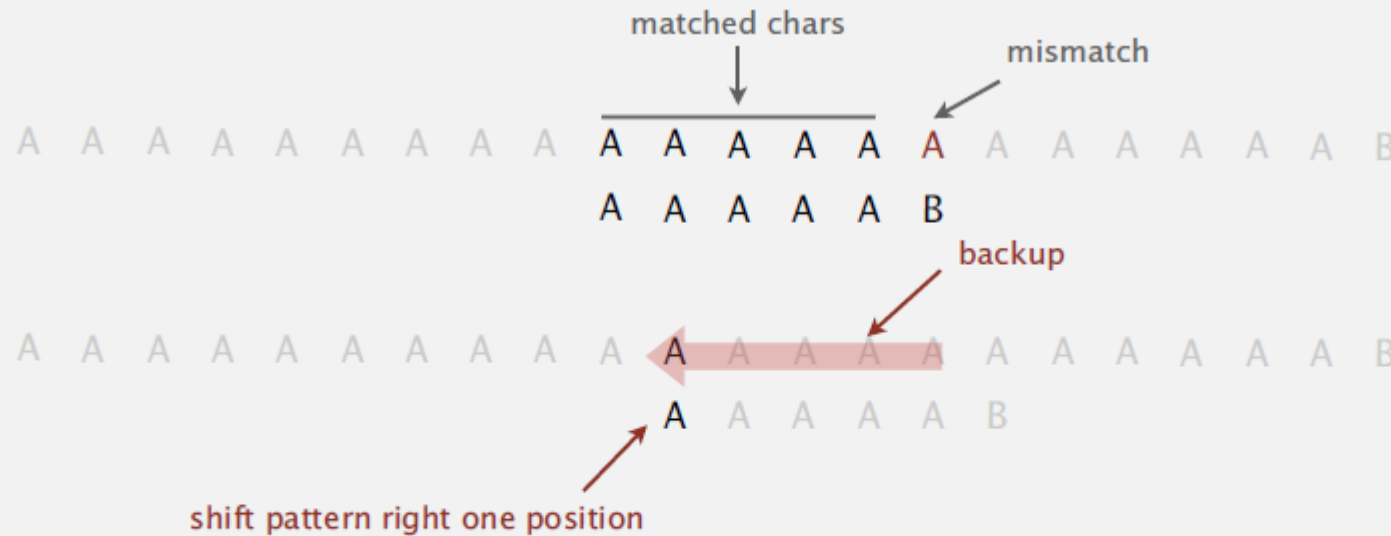
In many applications, we want to avoid **backup** in text stream.

- Treat input as stream of data.
- Abstract model: standard input.

"ATTACK AT DAWN"
substring search
machine
found



Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of last M characters.

Approach 2. Stay tuned.

BOYER-MOORE ALGORITHM

Boyer-Moore algorithm

- ❖ It was developed by **Robert S. Boyern** and **J Strother Moore** in 1977.
- ❖ The Boyer-Moore algorithm is consider the most efficient string-matching algorithm.
 - **For Example:** text editors and commands substitutions.

❖ WHY WE USE THIS ALGORITHM??

- Problem for Brute Force Search:
 - We keep considering too many comparisons, So the time complexity increase $O(mn)$. That's why Boyer-Moore Algorithm. It works the fastest when the alphabet is moderately sized and the pattern is relatively long.

Boyer-Moore algorithm

- ❖ The algorithm scans the characters of the pattern from **right to left** i.e **beginning with the rightmost character**.
- ❖ If the text symbol that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by m positions behind this text symbol.
- ❖ **Example:**
“Hello to the world.” is a string and if we want to search “world” for that string that’s a Pattern.

- ❖ Boyer Moore is a combination of following two approaches.
 - 1) Bad Character Approach
 - 2) Good Suffix Approach
- ❖ Both of the above Approach can also be used independently to search a pattern in a text.
- ❖ But here we will discuss about Bad-Match Approach.
- ❖ Boyer Moore algorithm does preprocessing.
- ❖ **Why Preprocessing??**
 - To shift the pattern by more than one character.

Bad Character approach

- ❖ The character of the text which doesn't match with the current character of pattern is called the **Bad Character**.
- ❖ Upon mismatch we shift the pattern until –
 - 1) The mismatch become a match.
 - ❖ If the mismatch occur then we see the Bad-Match table for shifting the pattern.
 - 2) Pattern P move past the mismatch character.
 - ❖ If the mismatch occur and the mismatch character not available in the Bad-Match Table then we shift the whole pattern accordingly.

Good suffix Approach

- ❖ Just like bad character heuristic, a preprocessing table is generated for good suffix Approach.
- ❖ Let t be substring of text T which is matched with substring of pattern P .
Now we shift pattern until :
 - 1) Another occurrence of t in P matched with t in T .
 - 2) A prefix of P , which matches with suffix of t
 - 3) P moves past t .

Here we use Bad-Match Approach for Searching

Boyer-Moore algorithm

Steps to find the pattern :

Step 1: Construct the bad-symbol shift table.

Step 2: Align the pattern against the beginning of the text.

Step 3: Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text.

1- Construct Bad Match Table:

Formula for constructing Bad Match Table:

- **Formula:**

Values = Length of pattern-index-1

Example:

Text: "WELCOMETOTEAMMAST"

Pattern: 'TEAMMAST'

Pattern T E A M M A S T Length = 8
 Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values						

Bad Match Table


 Pattern **T E A M M A S T** Length = 8
 Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values	7					

Values = $\text{Max}(1, \text{Length of string} - \text{index} - 1)$

$$T = \text{max}(1, 8 - 0 - 1) = 7$$



Pattern **T E A M M A S T** Length = 8
 Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values	7	6				

Values = $\max(1, \text{Length of string} - \text{index} - 1)$

$$E = \max(1, 8 - 1 - 1) = 6$$


Pattern **T E A M M A S T** Length = 8
Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values	7	6	5			

Values = $\max(1, \text{Length of string} - \text{index} - 1)$

$$A = \max(1, 8 - 2 - 1) = 5$$

↓

Pattern **T E A M M A S T** Length = 8

Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values	7	6	5	4		

Values = $\max(1, \text{Length of string} - \text{index} - 1)$

$$M = \max(1, 8 - 3 - 1) = 4$$

↓

Pattern **T E A M M A S T** Length = 8

Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values	7	6	5	3		

Values = $\max(1, \text{Length of string} - \text{index} - 1)$

M = $\max(1, 8 - 4 - 1) = 3$


 Pattern **T E A M M A S T** Length = 8
 Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values	7	6	2	3		

Values = $\max(1, \text{Length of string} - \text{index} - 1)$
 A = $\max(1, 8 - 5 - 1) = 2$

Pattern **T E A M M A S T** Length = 8
 Index # 0 1 2 3 4 5 6 7



Letter	T	E	A	M	S	*
Values	7	6	2	3	1	

Values = Length of string - index - 1

$$S = \max(1, 8 - 6 - 1) = 1$$

Pattern **T E A M M A S T**
 Index # 0 1 2 3 4 5 6 7



Letter	T	E	A	M	S	*
Values	1	6	2	3	1	

Values = $\max(1, \text{Length of string} - \text{index} - 1)$

$$T = \max(1, 8 - 7 - 1) = 1$$

Pattern **T E A M M A S T** Length = 8
Index # 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Values	8	6	2	3	1	8



Any other letter is presented by '*' is taken equal value to length of string i.e 8 here.

2- Align the Pattern

Text:

W E L C O M E T O T E A M M A S T

Pattern:

T E A M M A S T

Move 6 spaces toward right

Bad-Match Table

Letter	Values
T	8
E	6
A	2
M	3
S	1
*	8



Matching.....



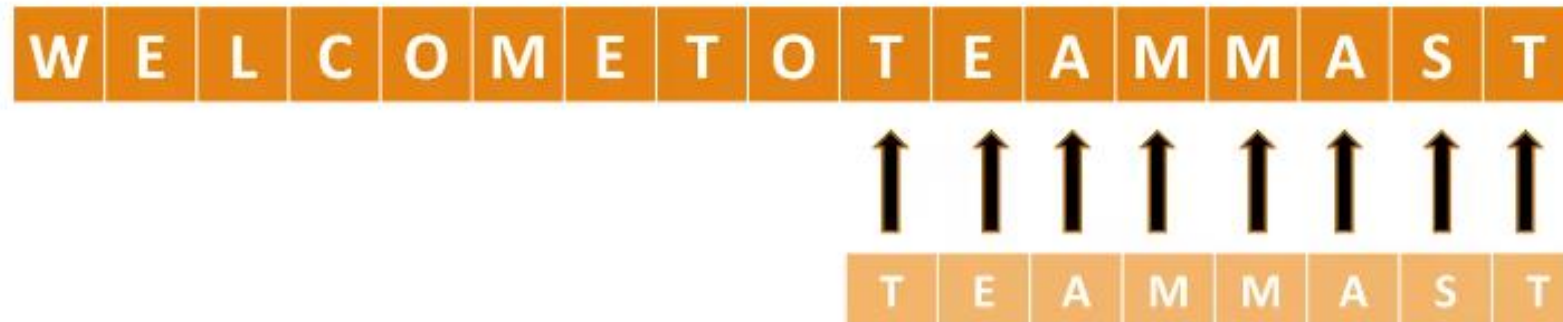
Now move 3 spaces toward right.

Bad-Match Table

Letter	Values
T	8
E	6
A	2
M	3
S	1
*	8



Matching.....



Hence the pattern match.....

Bad-Match Table

Letter	Values
T	8
E	6
A	2
M	3
S	1
*	8

Time complexity

- ❖ The preprocessing phase in $O(m+\Sigma)$ time and space complexity and searching phase in $O(mn)$ time complexity.
- ❖ It was proved that this algorithm has $O(m)$ comparisons when P is not in T . However, this algorithm has $O(mn)$ comparisons when P is in T .

The Boyer-Moore Algorithm

Algorithm **BoyerMooreMatch**(T, P, Σ)

$L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma)$

$i \leftarrow m - 1$

$j \leftarrow m - 1$

repeat

if $T[i] = P[j]$

if $j = 0$

return i { match at i }

else

$i \leftarrow i - 1$

$j \leftarrow j - 1$

else

{ character-jump }

$l \leftarrow L[T[i]]$

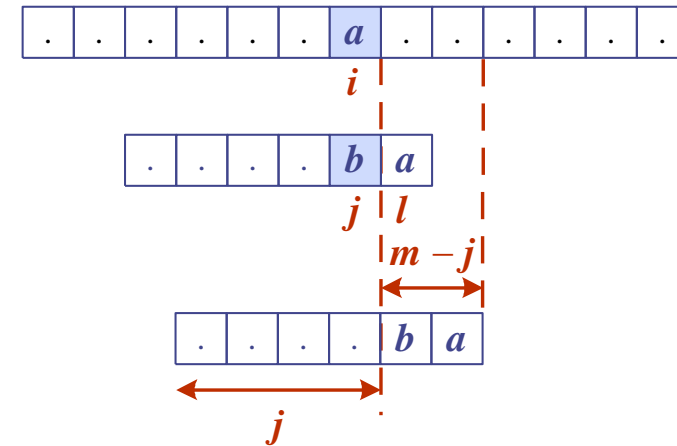
$i \leftarrow i + m - \min(j, 1 + l)$

$j \leftarrow m - 1$

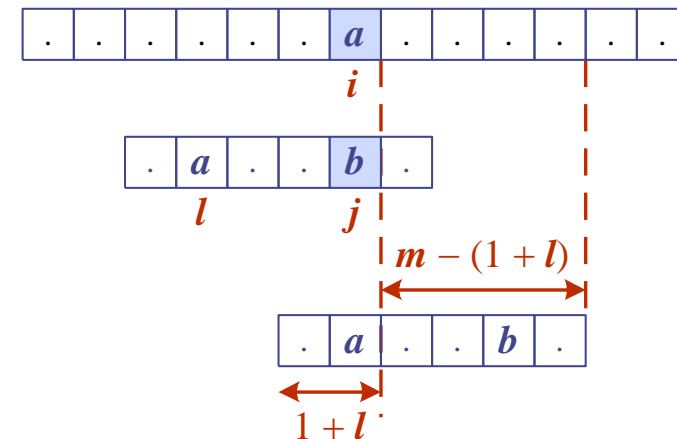
until $i > n - 1$

return -1 { no match }

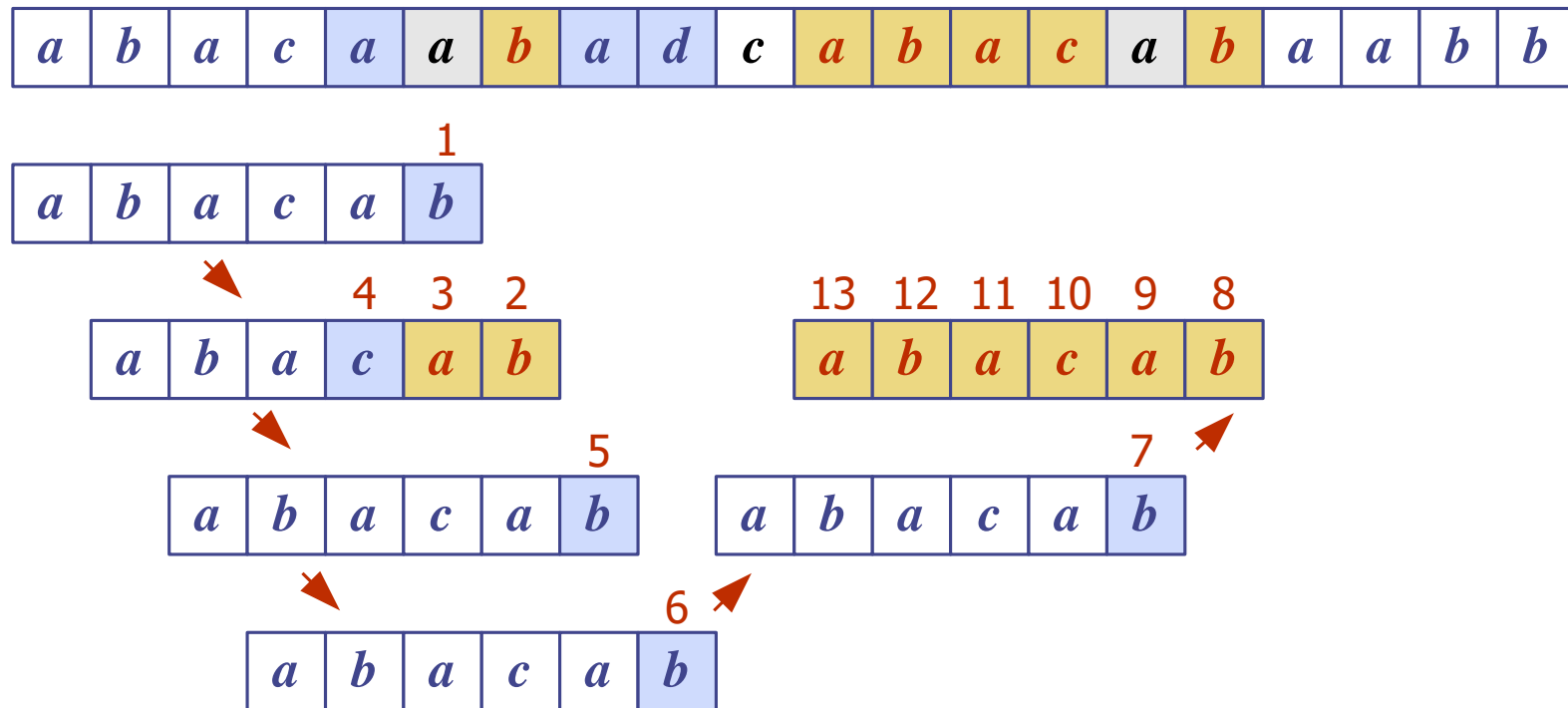
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

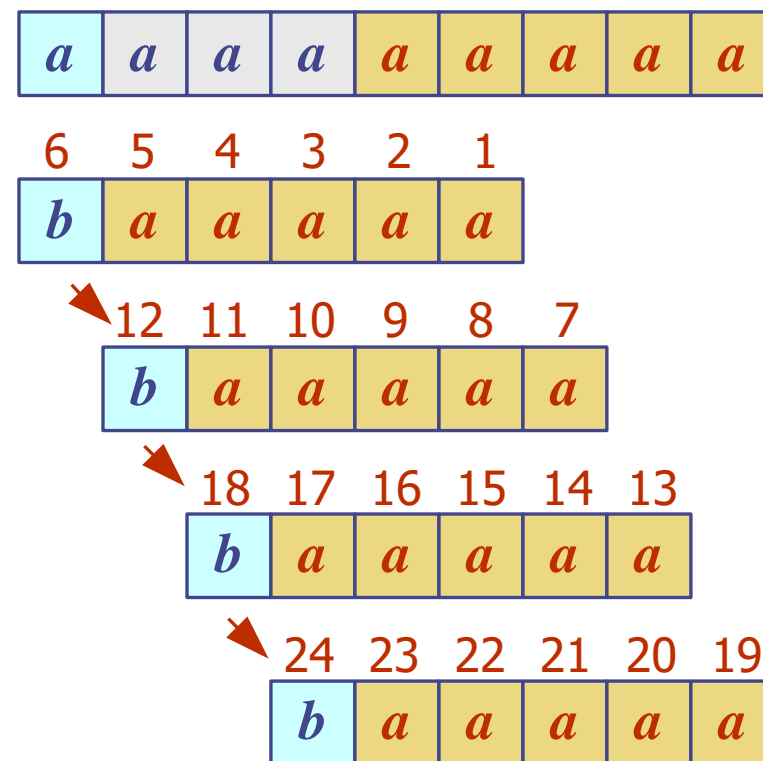


Example



Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
 - $T = aaa \dots a$
 - $P = baaa$
- 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



KNUTH-MORRIS-PRATT ALGORITHM

The problem of String Matching

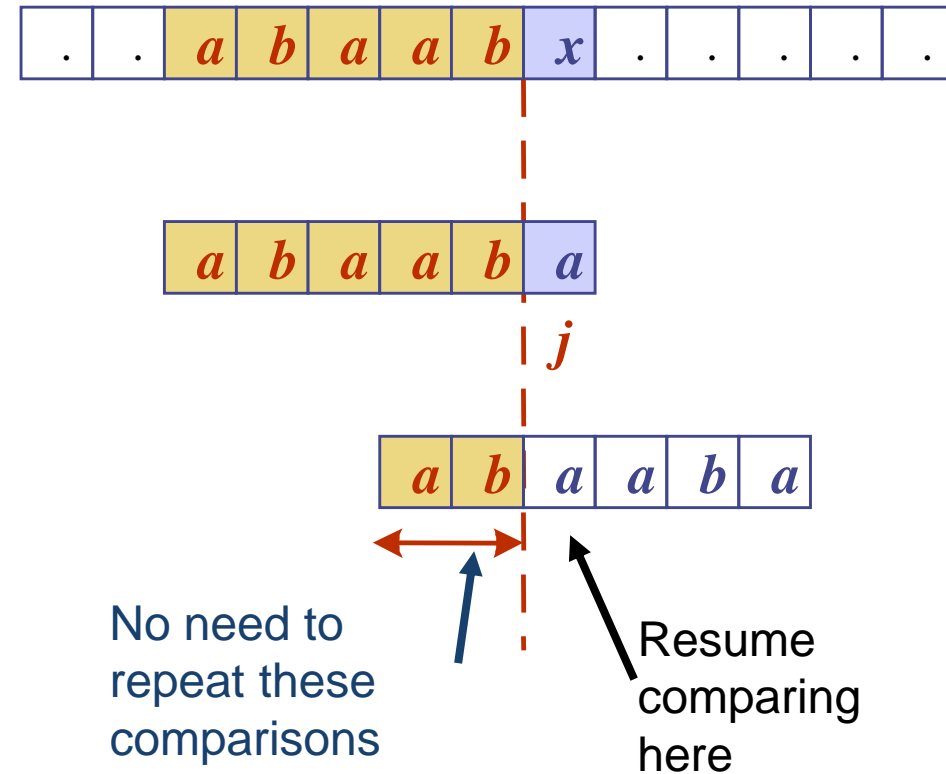
Given a string 'S', the problem of string matching deals with finding whether a pattern 'p' occurs in 'S' and if 'p' does occur then returning position in 'S' where 'p' occurs.

.... 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 'S' in which to locate 'p'. If the first element of 'p' matches the first element of 'S', compare the second element of 'p' with second element of 'S'. 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'.

The KMP Algorithm - Motivation

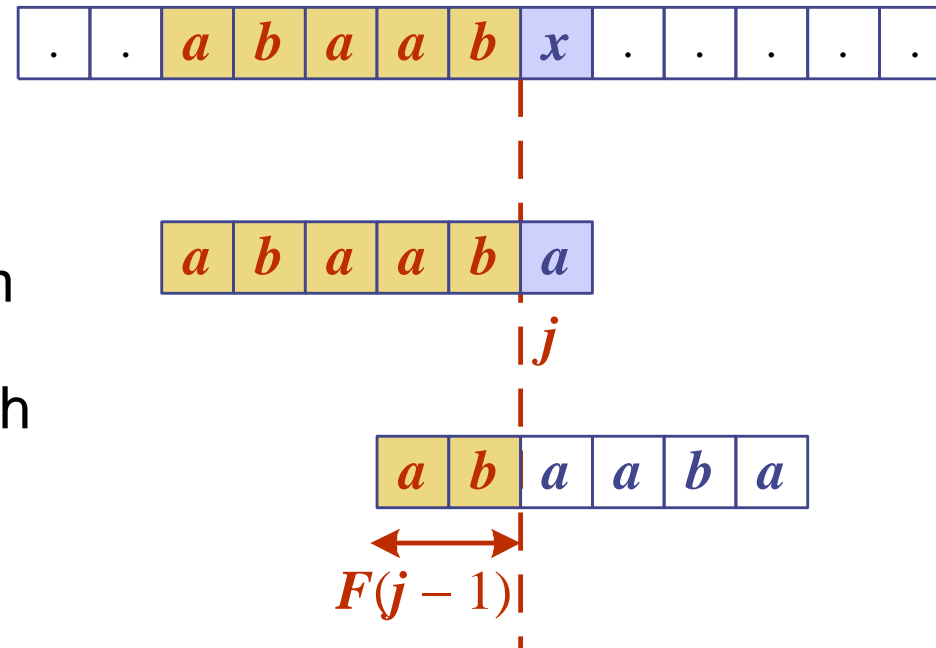
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix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



KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ 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 $j \leftarrow F(j - 1)$

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

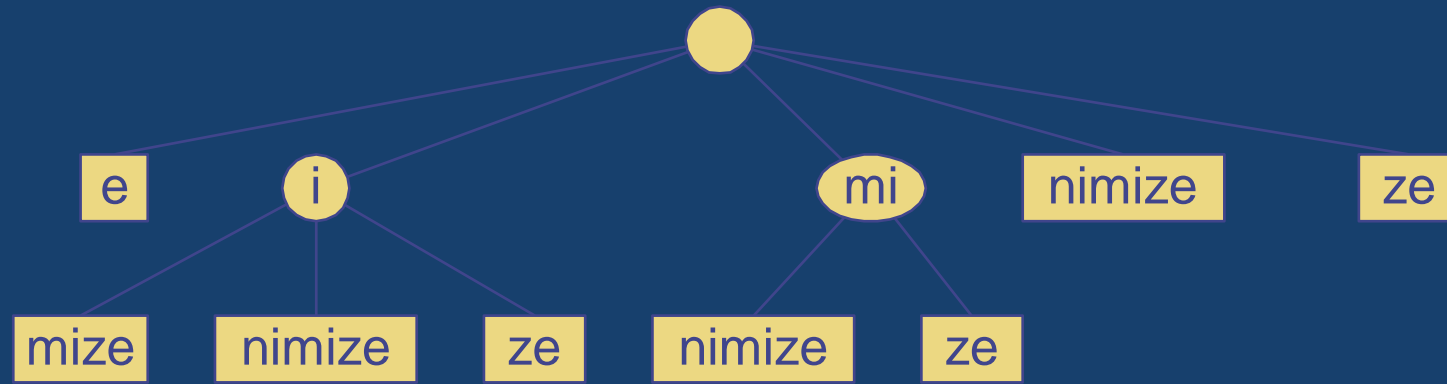


The KMP Algorithm

- The failure 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 $F(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

Algorithm *KMPMatch*(T, P)

```
 $F \leftarrow \text{failureFunction}(P)$   
 $i \leftarrow 0$   
 $j \leftarrow 0$   
while  $i < n$   
    if  $T[i] = P[j]$   
        if  $j = m - 1$   
            return  $i - j$  { match }  
        else  
             $i \leftarrow i + 1$   
             $j \leftarrow j + 1$   
    else  
        if  $j > 0$   
             $j \leftarrow F[j - 1]$   
        else  
             $i \leftarrow i + 1$   
return  $-1$  { no match }
```



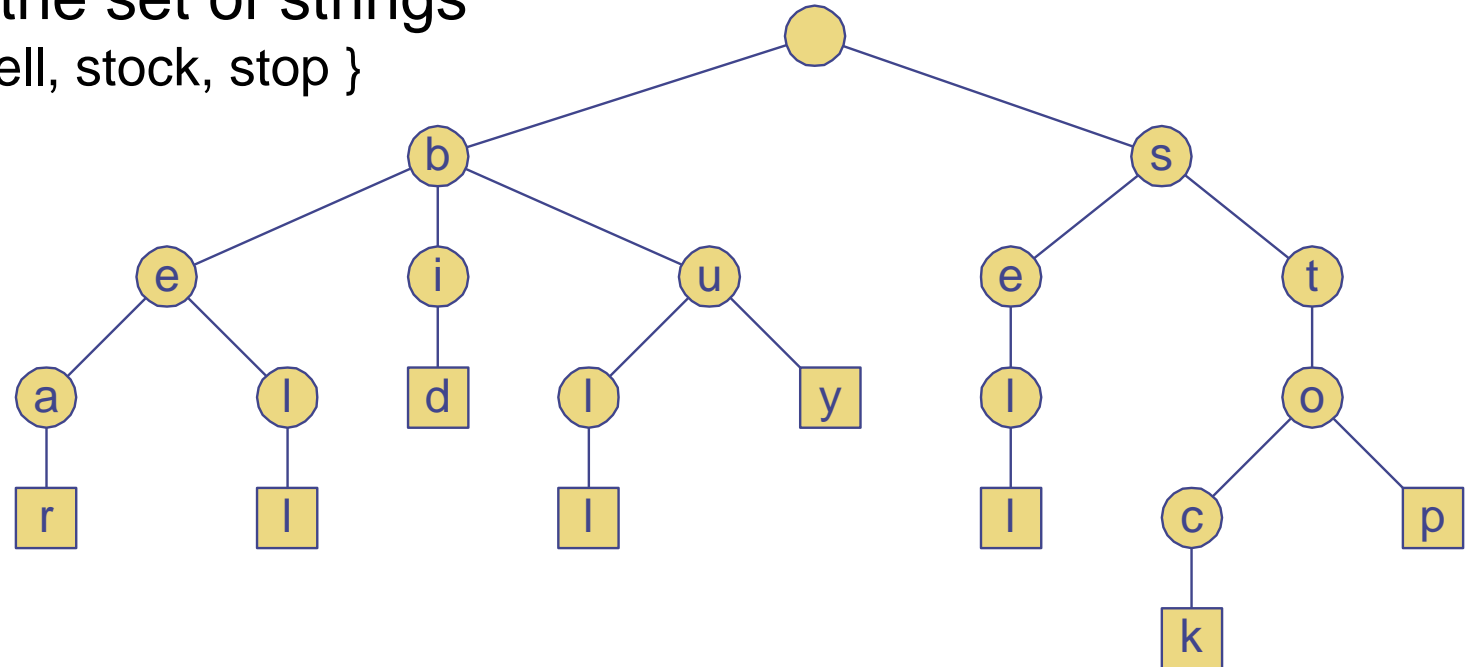
TRIES

Preprocessing Strings

- Preprocessing the pattern speeds up pattern matching queries
 - After preprocessing the pattern, KMP's algorithm performs pattern matching in time proportional to the text size
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
- A trie is a compact data structure for representing a set of strings, such as all the words in a text
 - A trie supports pattern matching queries in time proportional to the pattern size

Standard Trie (1)

- The standard trie for a set of strings S is an ordered tree such that:
 - Each node but the root is labeled with a character
 - The children of a node are alphabetically ordered
 - The paths from the external nodes to the root yield the strings of S
- Example: standard trie for the set of strings
 $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$



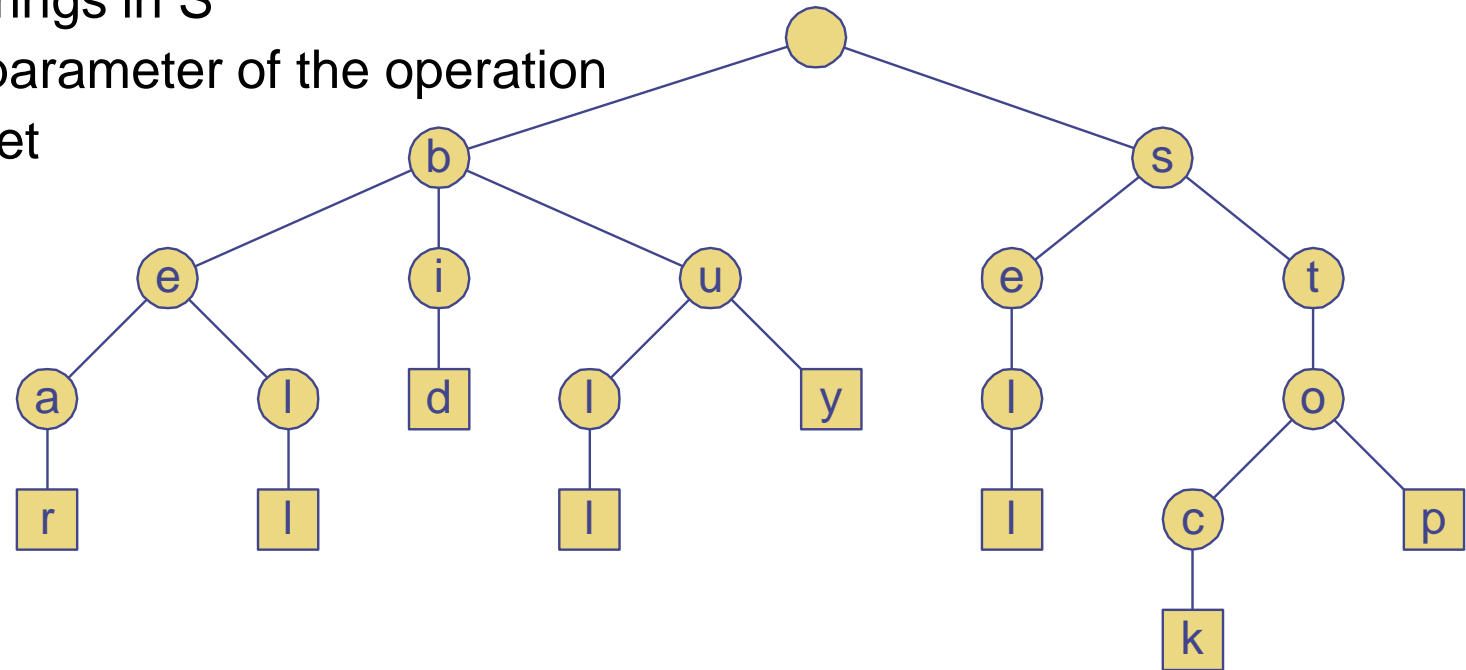
Standard Trie (2)

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:

n total size of the strings in S

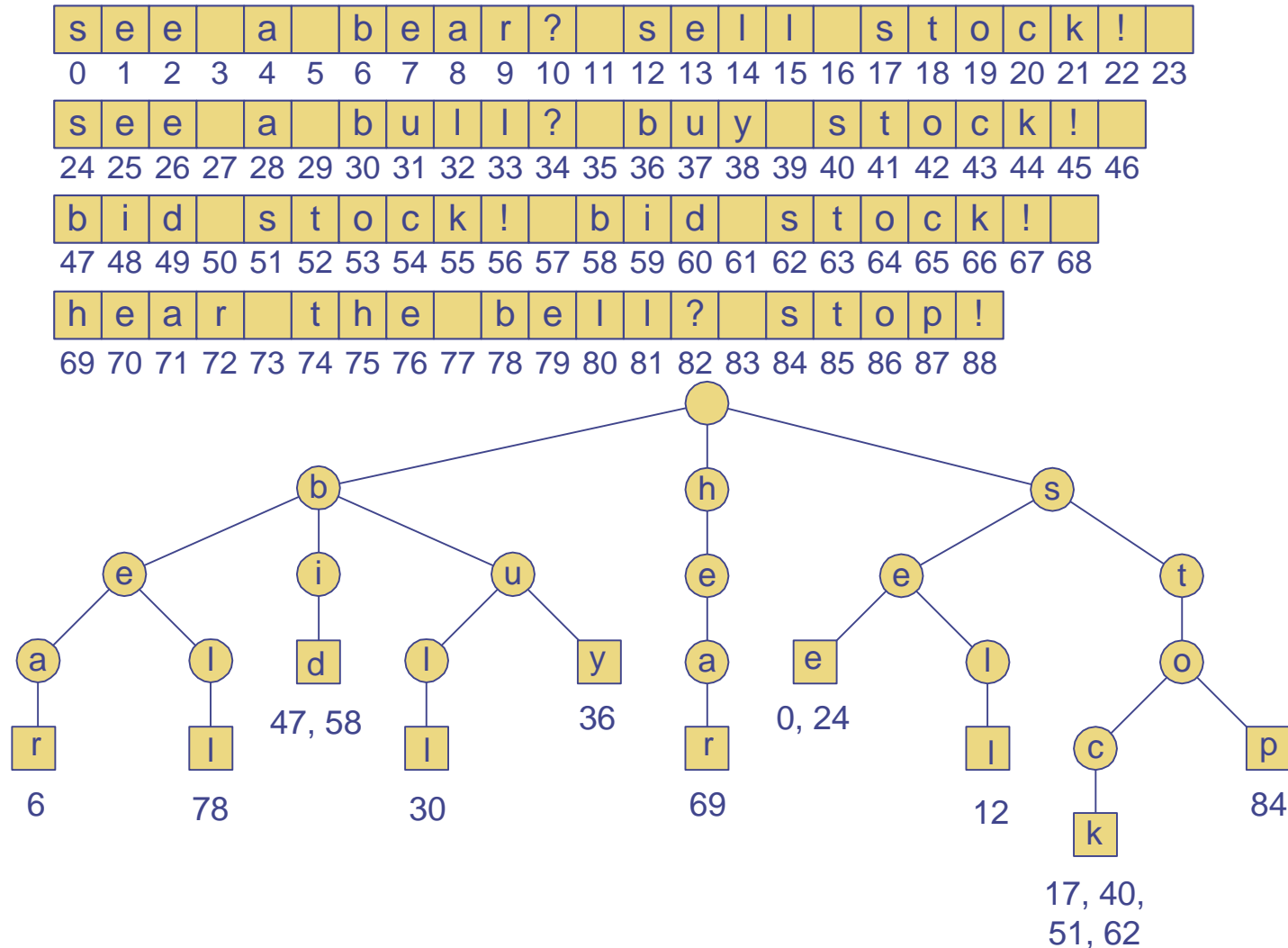
m size of the string parameter of the operation

d size of the alphabet



Word Matching with a Trie

- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text



Compressed Trie

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes

