Design and Analysis of Algorithms (UE18CS251)

Unit IV - Space and Time Tradeoffs

Mr. Channa Bankapur
channabankapur@pes.edu

Space and Time Tradeoffs

- **Input Enhancement**
  - preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward.
  - Eg: Counting methods of sorting, Horspool's algorithm, Boyer-Moore's algorithm.

- **Pre-structuring**
  - use of extra space to facilitate faster and/or more flexible access to the data.
  - Eg: Hashing, B-Trees

- **Dynamic Programming**

Sorting by Counting

1. **Comparison Counting Sorting**
   - For each element of the list, count the total number of elements smaller than this element.
   - These numbers will indicate the positions of the elements in the sorted list.

2. **Distribution Counting Sorting**
   - Suppose the elements of the list to be sorted belong to a finite set (aka domain).
   - Count the frequency of each element of the set in the list to be sorted.
   - Scan the set in order of sorting and print each element of the set according to its frequency, which will be the required sorted list.

Input Enhancement

→ **Sorting by Counting**

→→ **Comparison Counting Sorting**

- For each element of the list, count the total number of elements smaller than the element.
- These numbers indicates the positions (0-based) of the elements in the sorted list.

Eg:	**62**	**31**	**84**	**96**	**19**	**47**
# lesser elements:	**3**	**1**	**4**	**5**	**0**	**2**

	19	**31**	**47**	**62**	**84**	**96**
	0	**1**	**2**	**3**	**4**	**5**

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

Example of sorting by comparison counting

Array $A[0..5]$						
	62	31	84	96	19	47

Initially	*Count* []	0	0	0	0	0	0
After pass $i = 0$	*Count* []	3	0	1	1	0	0
After pass $i = 1$	*Count* []		1	2	2	0	1
After pass $i = 2$	*Count* []			4	3	0	1
After pass $i = 3$	*Count* []				5	0	1
After pass $i = 4$	*Count* []					0	2
Final state	*Count* []	3	1	4	5	0	2

Array $S[0..5]$						
	19	31	47	62	84	96

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}$$

1. Optimal number of key moves.
2. Does it sort when there are duplicate elements?
3. If so, is the sort stable?
4. What happens if the condition **A[i] < A[j]** is replaced with **A[i] <= A[j]** ?
5. Stable, but not in-place. Takes O(n) space.

Q: Write an efficient algorithm to sort a list of n 0s and 1s. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of n 0s and 1s. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of n 0s, 1s and 2s. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of **n** 0s and 1s. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of **n** 0s, 1s and 2s. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of **n** elements where the elements belong to set of natural numbers [0..100], and **101 << n**. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of **n** 0s and 1s. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of **n** 0s, 1s and 2s. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of **n** elements where the elements belong to set of natural numbers [0..100], and **101 << n**. Can it be done faster than O(n log n)?

Q: Write an efficient algorithm to sort a list of **n** elements where the elements belong to set of m elements, and **m << n**. Can it be done faster than O(n log n)?

Input Enhancement
→ Sorting by Counting
→→ Distribution Counting Sorting

13	11	12	13	12	12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

11	12	12	12	13	13

Example of sorting by distribution counting

13	11	12	13	12	12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

11	12	12	12	13	13

	D[0..2]		
A [5] = 12	1	**4**	6
A [4] = 12	1	**3**	6
A [3] = 13	1	2	**6**
A [2] = 12	1	**2**	5
A [1] = 11	**1**	1	5
A [0] = 13	0	1	**5**

S[0..5]					
			12		
		12			
					13
	12				
11					
				13	

Input Enhancement
→ Sorting by Counting
→→ Distribution Counting Sorting

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

//Sorts an array of integers from a limited range by distribution counting
//Input: An array $A[0..n-1]$ of integers between l and u $(l \le u)$
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $j \leftarrow 0$ **to** $u-l$ **do** $D[j] \leftarrow 0$ //initialize frequencies
for $i \leftarrow 0$ **to** $n-1$ **do** $D[A[i]-l] \leftarrow D[A[i]-l]+1$ //compute frequencies
for $j \leftarrow 1$ **to** $u-l$ **do** $D[j] \leftarrow D[j-1]+D[j]$ //reuse for distribution
for $i \leftarrow n-1$ **downto** 0 **do**
    $j \leftarrow A[i]-l$
    $S[D[j]-1] \leftarrow A[i]$
    $D[j] \leftarrow D[j]-1$
return S

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1     //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]            //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

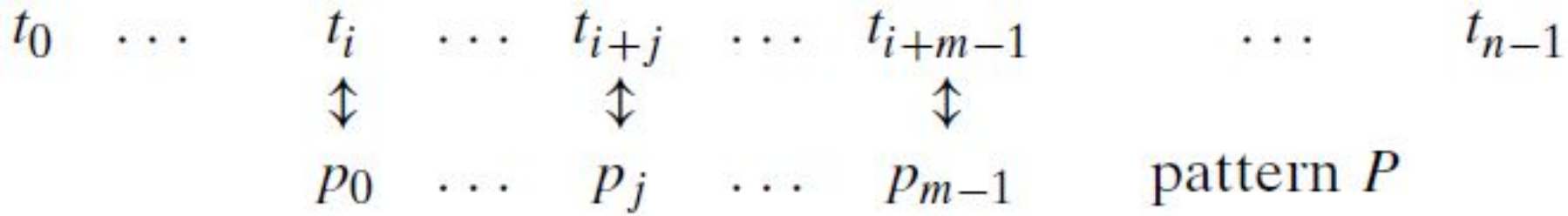
Trace the algorithm for sorting 5-star ratings of 7 people.
(A, 3), (B, 5), (C, 2), (D, 3), (E, 3), (F, 5), (G, 3).
Alphabet: {1, 2, 3, 4, 5} OR **{2, 3, 4, 5} with l=2, u=5**.
Frequencies: 1, 4, 0, 2. ⇒ Distributions: **1, 5, 5, 7**.
Sorted list: (A, 3), (B, 5), (C, 2), (D, 3), (E, 3), (F, 5), (G, 3).

1. Space complexity: O(*u-l*) distributions and O(n) for final sorting.
2. Time complexity: O(n + (*u-l*)).
   It is O(n) when *u-l* ∈ O(n).

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

//Sorts an array of integers from a limited range by distribution counting
//Input: An array $A[0..n-1]$ of integers between l and u $(l \le u)$
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $j \leftarrow 0$ **to** $u-l$ **do** $D[j] \leftarrow 0$ //initialize frequencies
for $i \leftarrow 0$ **to** $n-1$ **do** $D[A[i]-l] \leftarrow D[A[i]-l]+1$ //compute frequencies
for $j \leftarrow 1$ **to** $u-l$ **do** $D[j] \leftarrow D[j-1]+D[j]$ //reuse for distribution
for $i \leftarrow n-1$ **downto** 0 **do**
    $j \leftarrow A[i]-l$
    $S[D[j]-1] \leftarrow A[i]$
    $D[j] \leftarrow D[j]-1$
return S

Input Enhancement in String Matching:

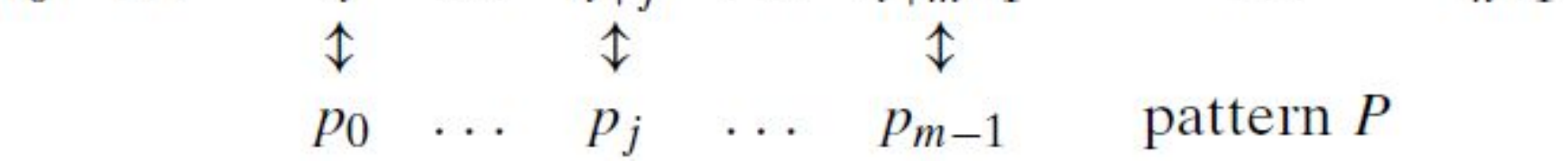


Input Enhancement to improve the average-case/amortized efficiency.

1. Horspool's algorithm
2. Boyer-Moore algorithm
3. Robin-Karp algorithm
4. Finite State Automaton
5. Knuth-Morris-Pratt algorithm

Consider the problem of searching for genes in DNA sequences.
A DNA sequence is represented by a text of the alphabet {A, C, G, T} and the gene segment is the pattern.

Eg: Text with n = 18 and pattern with m = 5.
DNA Sequence: **A C G T T A G C A G C G C A G C G C**
Gene segment: **A G C G C**



ALGORITHM *BruteForceStringMatch*$(T[0..n-1],\ P[0..m-1])$

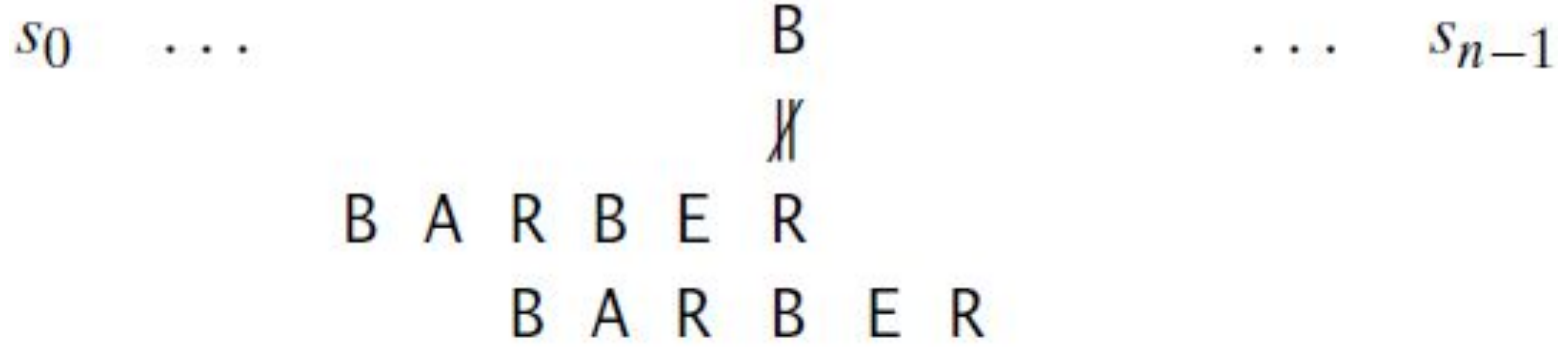
```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

```
NaiveStringMatch(T[0..n-1], P[0..m-1])
   for i ← 0 to n-m
      j ← 0
      while (j < m and T[i+j] = P[j])
         j ← j + 1
      if(j = m) return i
   return -1
```

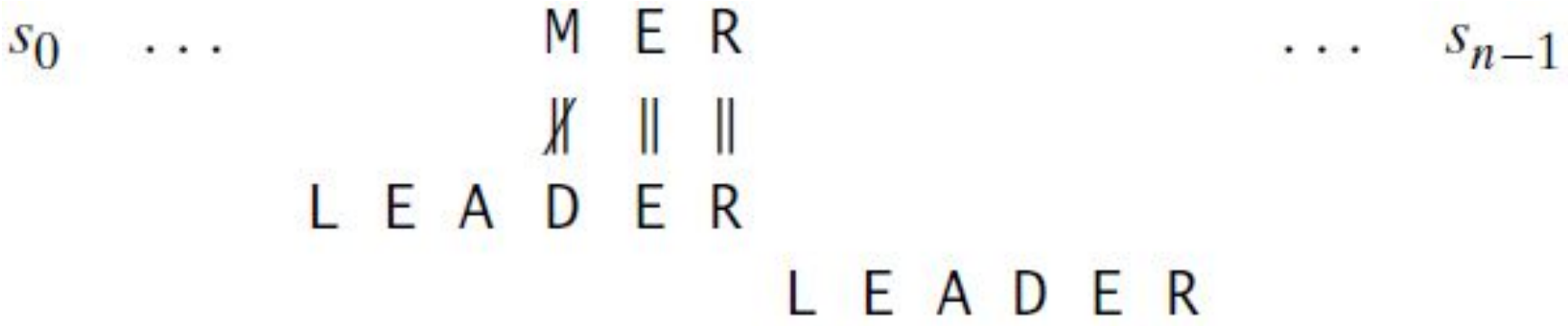
```
NaiveStringMatch2(T[0..n-1], P[0..m-1])
   for i ← m-1 to n-1
      j ← 0
      while (j < m and T[i-j] = P[m-1-j])
         j ← j + 1
      if(j = m) return i-(m-1)
   return -1
```

```
NaiveStringMatch2(T[0..n-1], P[0..m-1])
   for i ← m-1 to n-1
      j ← 0
      while (j < m and T[i-j] = P[m-1-j])
         j ← j + 1
      if(j = m) return i-(m-1)
   return -1
-------------------------------------
NaiveStringMatch3(T[0..n-1], P[0..m-1])
   i ← m-1
   while (i < n)
      j ← 0
      while (j < m and T[i-j] = P[m-1-j])
         j ← j + 1
      if(j = m) return i-(m-1)
      i ← i+1
   return -1
```

Horspool's Algorithm



Horspool's Algorithm



Eg: Pattern **AGCGC**

Shift table:

A	C	G	T
4	2	1	5

Eg: Gene segment in DNA Sequence using Horspool's algo.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
A C G T T A G C A G C G C A G C G C
A G C G C
          A G C G C
            A G C G C
              A G C G C
```

Tbl[T] = 5

Tbl[G] = 1

Tbl[C] = 2

Horspool's Algorithm

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

```
NaiveStringMatch3(T[0..n-1], P[0..m-1])
    i ← m-1
    while (i < n)
        j ← 0
        while (j < m and T[i-j] = P[m-1-j])
            j ← j + 1
        if(j = m) return i-(m-1)
        i ← i+1
    return -1
```

```
HorspoolMatching(T[0..n-1], P[0..m-1])
    STbl[alphabet size] ← ShiftTbl(P[0..m-1])
    i ← m-1
    while (i < n)
        j ← 0
        while (j < m and T[i-j] = P[m-1-j])
            j ← j + 1
        if(j = m) return i-(m-1)
        i ← i + STbl[ T[i] ]
    return -1
```

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])          //generate Table of shifts
i ← m − 1                        //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                        //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

s_0 . . . c . . . s_{n-1}

B A R B E R

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

B A R B E R B A R B E R

B A R B E R B A R B E R

B A R B E R B A R B E R

Consider the problem of searching for genes in DNA sequences using Horspool's algorithm.
A DNA sequence is represented by a text on the alphabet {A, C, G, T}, and the gene or gene segment is the pattern.

Construct the shift table for the following gene segment of your chromosome 10: TCCTATTCTT

Apply Horspool's algorithm to locate the above pattern in the following DNA sequence:

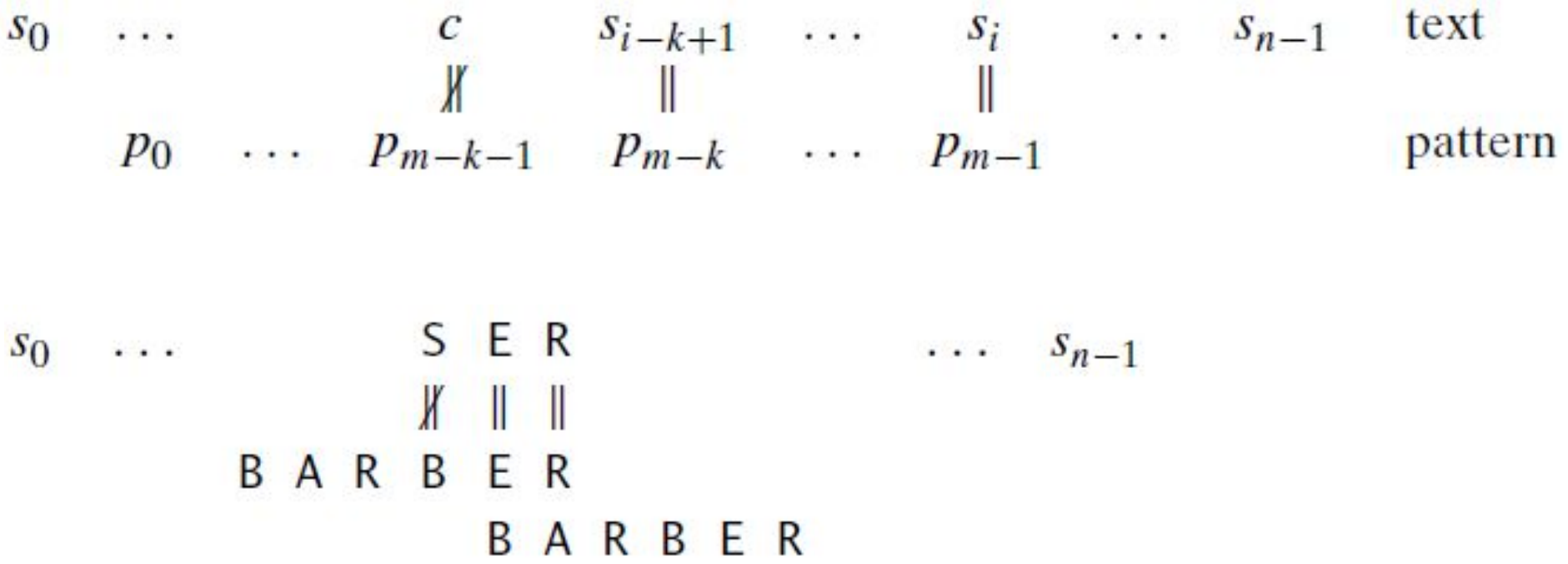
TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT

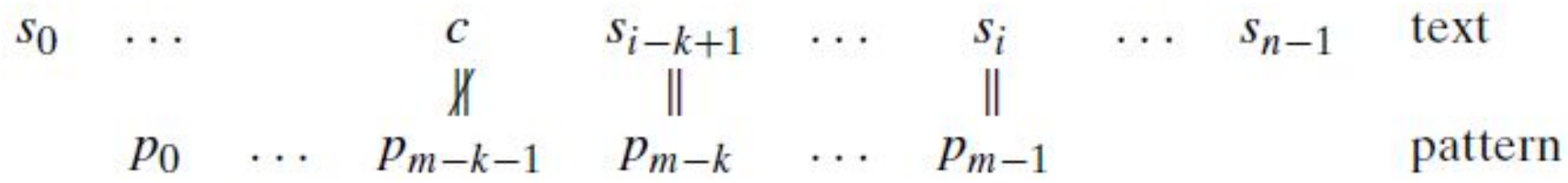
How many character comparisons will be made by Horspool's algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

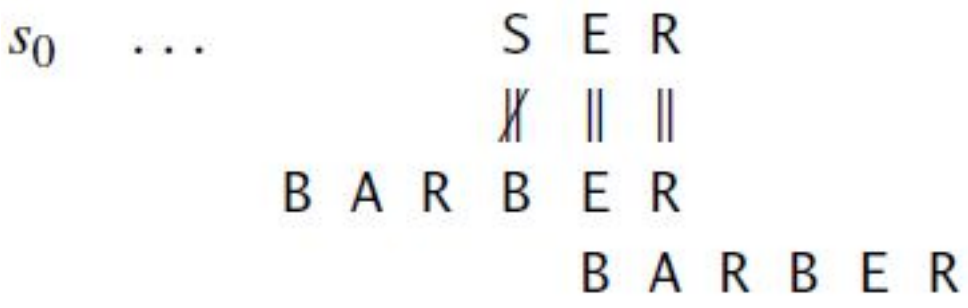
a. 00001 **b.** 10000 **c.** 01010

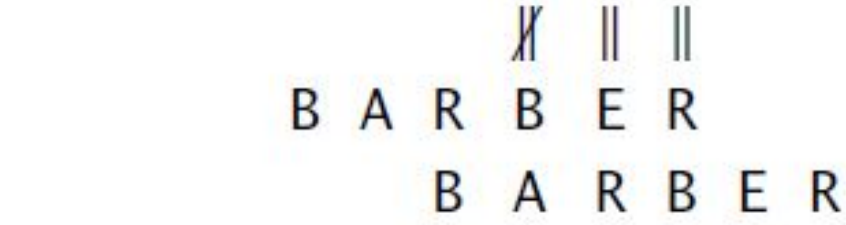
Boyer-Moore Algorithm:

Bad-symbol shift




s_0 ... c s_{i-k+1} ... s_i ... s_{n-1} text
p_0 ... p_{m-k-1} p_{m-k} ... p_{m-1} pattern


s_0 ... S E R ... s_{n-1}
B A R B E R
B A R B E R

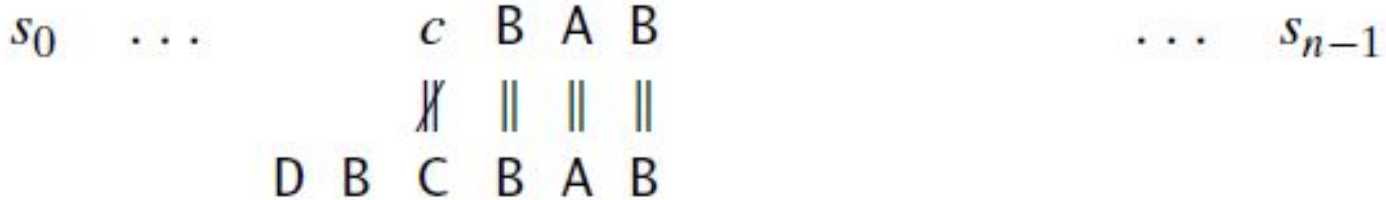

s_0 ... A E R ... s_{n-1}
B A R B E R
B A R B E R

$$d_1 = \max\{t_1(c) - k, 1\}$$

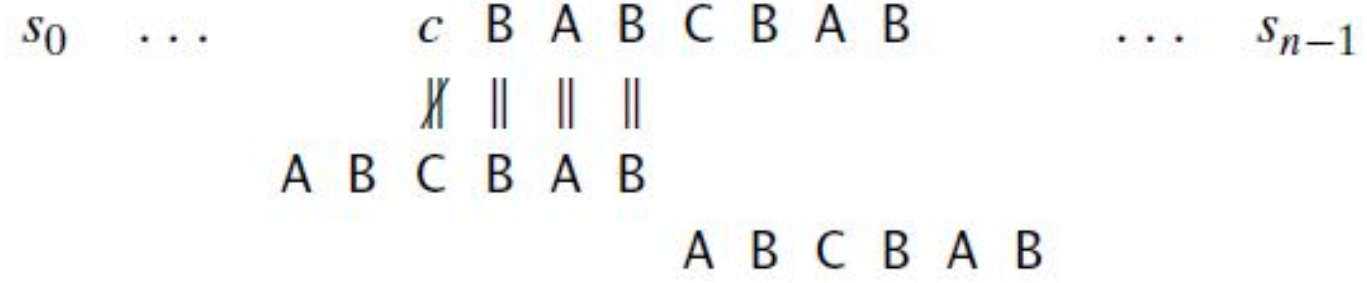
Boyer-Moore Algorithm:

Good-suffix shift

k	pattern	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4



k	pattern	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4
3	$\overline{\text{ABC}}$B$\underline{\text{AB}}$	4
4	$\overline{\text{ABC}}$$\underline{\text{BAB}}$	4
5	$\overline{\text{A}}$$\underline{\text{BCBAB}}$	4



Boyer-Moore Algorithm:

Step 1: For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2: Using the pattern, construct the good-suffix shift table as described earlier.

Step 3: Align the pattern against the beginning of the text.

Step 4: Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text.

…

Boyer-Moore Algorithm:

Step 4: Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after k ≥ 0 character pairs are matched successfully. In the latter case, retrieve the entry t1(c) from the c's column of the bad-symbol table where c is the text's mismatched character. If k > 0, also retrieve the corresponding d_2 entry from the good-suffix table. **Shift the pattern to the right by the number of positions computed by the formula**

$$d = \begin{cases} d_1 & \text{if } k = 0, \quad \text{where } d_1 = \max\{t_1(c) - k, 1\} \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

```
BoyerMooreMatching(T[0..n-1], P[0..m-1])
   STbl[alphabet size] ← ShiftTbl(P[0..m-1])
   GTbl[1..m-1] ← GoodSuffixTbl(P[0..m-1])
   i ← m-1
   while (i < n)
      j ← 0
      while (j < m and T[i-j] = P[m-1-j])
         j ← j + 1
      if(j = m) return i-(m-1)
      d ← max{STbl[T[i-j]] - j, 1}
      if(j>0)
         d2 ← GTbl[j]
         if(d2 > d) d ← d2
      i ← i + d
   return -1
```

BAOBAB

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

k	**pattern**	d_2
1	BAO$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{B}}$AOB$\underline{\text{AB}}$	5
3	$\overline{\text{B}}$AO$\underline{\text{BAB}}$	5
4	$\overline{\text{B}}$A$\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}\underline{\text{AOBAB}}$	5

B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(K) - 0 = 6$

B A O B A B

$d_1 = t_1(\_) - 2 = 4$

$d_2 = 5$

$d = \max\{4, 5\} = 5$

B A O B A B

$d_1 = t_1(\_) - 1 = 5$

$d_2 = 2$

$d = \max\{5, 2\} = 5$

B A O B A B

Eg: Pattern **AGCGC**

Shift table:

A	C	G	T
4	2	1	5

Good-suffix table[1..4]: 5, 2, 5, 5

Gene segment in DNA Sequence using Boyer-Moore's algo.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
A C G T T A G C A G C G C A G C G C
A G C G C
          A G C G C
            A G C G C
              A G C G C
```

STbl[T] = 5

STbl[G] = 1

GTbl[2] = 2

Boyer-Moore algorithm:

Create bad-symbol shift table and good-suffix shift table for the pattern: BAOBABAB

Boyer-Moore algorithm:

Create bad-symbol shift table and good-suffix shift table for the pattern: BAOBABAB

A	**B**	**O**	**Others**
1	2	5	8

k:	**d_2**
1:	4
2:	7
3:	2
4:	7
5,6,7:	7

Space-Time Tradeoffs in Algorithms is not the same as the Space-Time Continuum theory of Physics.

</ Space-Time Tradeoffs >