

ANALYSIS AND DESIGN OF ALGORITHMS

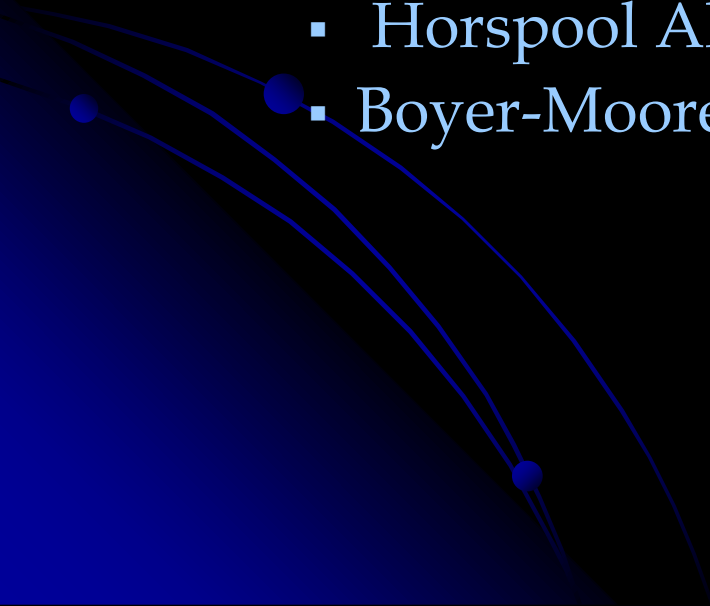
UNIT-III

CHAPTER 7:

SPACE AND TIME TRADEOFFS



OUTLINE

- ❖ Space and Time Tradeoffs
 - Sorting by Counting
 - Input Enhancement in String Matching
 - Horspool Algorithm
 - Boyer-Moore Algorithm
- 

Space and Time Tradeoffs

- The two techniques based on pre-processing of data thereby increasing the speed of the algorithm are:
 - **Input enhancement**
 - **Pre-structuring**
- **Input Enhancement Technique**: In this technique it is required to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterwards.

We discuss the following algorithms based on it:

- Counting methods for sorting.
 - Boyer-Moore algorithm for string matching and its simplified version suggested by Horspool.
-
- **Pre-structuring**: This technique that exploits space-for-time tradeoffs simply uses extra space to facilitate faster and/or more flexible access to the data.
 - Hashing.

Sorting by Counting

- For each element of a list to be sorted, *count the total number of elements smaller than this element* and record the results in a table.
- These numbers will *indicate the positions of the elements* in sorted list:
Example: If the count is 3 for some element, it should be in the 4th position in the sorted array.
- Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.
- This algorithm is called **Comparison counting sort**.

A: Elements

Count

20	35	10	18	40	15
3	4	0	2	5	1

S: sorted Elements

0	1	2	3	4	5
10	15	18	20	35	40

Sorting by Counting

ALGORITHM *ComparisonCountingSort*($A[0 \dots n-1]$)
// Sorts an array by comparison counting
// Input: An array $A[0 \dots n-1]$ of orderable elements
// Output: Array $S[0 \dots n-1]$ of A 's elements sorted in
// nondecreasing order
for $i \leftarrow 0$ to $n-1$ do Count[i] $\leftarrow 0$
for $i \leftarrow 0$ to $n-2$ do
 for $j \leftarrow i+1$ to $n-1$ do
 if $A[i] < A[j]$
 Count[j] \leftarrow Count[j] + 1
 else Count[i] \leftarrow Count[i] + 1
for $i \leftarrow 0$ to $n-1$ do $S[\text{Count}[i]] \leftarrow A[i]$
return S

Tracing of Comparison counting Sort

Let us illustrate the working of this algorithm by taking the elements 20, 35, 10, 18, 40, 15. The outermost loop varies from 0 to $n-2$ and thus total number of passes required will be $n-1$. In this example, since there are 6 elements, we require 5 passes. The figure below provides the number of elements less than the corresponding item at each pass.

Array A[0...5]

0	1	2	3	4	5
20	35	10	18	40	15

Initially

Count[]

0	0	0	0	0	0
---	---	---	---	---	---

After pass $i = 0$

Count[]

3	1	0	0	1	0
---	---	---	---	---	---

After pass $i = 1$

Count[]

	4	0	0	2	0
--	---	---	---	---	---

After pass $i = 2$

Count[]

		0	1	3	1
--	--	---	---	---	---

After pass $i = 3$

Count[]

			2	4	1
--	--	--	---	---	---

After pass $i = 4$

Count[]

				5	1
--	--	--	--	---	---

Final state

Count[]

3	4	0	2	5	1
---	---	---	---	---	---

Array S[0...5] Sorted

0	1	2	3	4	5
10	15	18	20	35	40

Analysis of Comparison count sort

- The input size metric for this algorithm is n .
- The basic operation is comparison statement “if $A[i] < A[j]$ ” in the innermost for loop.
- The number of comparisons can be obtained as shown below:

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

Since the algorithm makes the same number of key comparisons as selection sort ($\Theta(n^2)$) and in addition uses a linear amount of extra space, it can hardly be recommended for practical use.

Sorting by Distribution counting

- If element values are integers between some lower bound l and upper bound u , we can compute the frequency of each of those values and store them in array $F[0 \dots u-l]$. Then the first $F[0]$ positions in the sorted list must be filled with l , the next $F[1]$ positions with $l+1$, and so on.
- Consider a situation of sorting a list of items with some other information associated with their keys without overwriting the list's elements.
 - Store the frequency of occurrence of each element in an array.
 - Then we can copy elements into new array $S[0 \dots n-1]$ to hold sorted list as follows:
 - Array A elements whose values are equal to the lowest value l are copied into the first $F[0]$ elements of S , i.e., positions 0 through $F[0]-1$, the elements of value $l+1$ are copied to positions from $F[0]$ to $(F[0]+F[1])-1$, and so on.
- Since such accumulated sums of frequencies are called a distribution in statistics, the method itself is known as **distribution counting**.

Sorting by Distribution counting

Example: Consider sorting the array

13	11	12	13	12	12
----	----	----	----	----	----

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting.

The frequency and distribution arrays are as follows:

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

Sorting by Distribution counting

- Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n-1$, the distribution values must be reduced by 1 to get corresponding element positions.

Input array

13	11	12	13	12	12
----	----	----	----	----	----

- It is convenient to process the input array from right to left.
- For example, the last element is 12 in above list, and, since its distribution value is 4, we place this 12 in position $4-1 = 3$ of the sorted array S.
- Then we decrease the 12's distribution value by 1 and proceed to next element (from the right) in the given array.

Sorting by Distribution counting

Input array

13	11	12	13	12	12
----	----	----	----	----	----

$D[0 \dots 2]$

0	1	2
---	---	---

$S[0 \dots 5]$

0	1	2	3	4	5
---	---	---	---	---	---

Index value
→

$A[5] = 12$

1	4	6
1	3	6
1	2	6
1	2	5
1	1	5
0	1	5

$A[4] = 12$

$A[3] = 13$

$A[2] = 12$

$A[1] = 11$

$A[0] = 13$

			12		
		12			
					13
	12				
11					
				13	

Figure: Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

Sorting by Distribution counting

ALGORITHM *DistributionCounting*($A[0 \dots n-1]$, L , U)

//Sorts an array of integers from a limited range by distribution counting

//Input: An array $A[0 \dots n-1]$ of integers between l and u ($l \leq u$)

//Output: Array $S[0 \dots n-1]$ 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

13	11	12	13	12	12
----	----	----	----	----	----

Analysis of sorting by distribution counting

- The input size metric for this algorithm is n .
- The statements within the for loop can be considered as basic operation.
- The number of times basic operation is executed can be obtained as shown below:

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

So, time complexity of sorting by distribution counting is $\Theta(n)$.

This is a better time-efficiency class than that of the most efficient sorting algorithms – mergesort, quicksort, and heapsort $\Theta(n \log_2 n)$ - we have encountered.

Sorting by Distribution counting

- **Problem1:** Trace the distribution counting sort algorithm for the following list:

2, 1, 3, 2, 3, 2, 1, 2

- **Problem2:** Assuming that the set of possible list values is {a, b, c, d}, sort the following list in alphabetical order by the distribution counting algorithm:

b, c, d, c, b, a, a, b.

Input Enhancement in String Matching

- The pattern matching algorithm using Brute-force method had the worst-case efficiency of $\Theta(mn)$ where m is the length of the pattern and n is the length of the text. In the average-case, its efficiency turns out to be in $\Theta(n)$.
- Several better algorithms have been discovered. Most of them exploit the input enhancement idea: *preprocess the pattern to get some information about it, store this information in a table*, and then use this information during an actual search for the pattern in a given text.
- The various algorithms which uses the *input enhancement technique* for string matching are:
 - Knuth-Morris-Pratt algorithm
 - Boyer-Moore algorithm
 - Horspool's algorithm which is a simplified version of Boyer - Moore algorithm.

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

- preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs
- Determines the size of shift by looking at the character c of the text that was aligned against the last character of the pattern.

How far to shift?

In general, the following four possibilities can occur :

Consider , as an example, searching for the pattern BAOBAB in some text:

Case 1: Look at first (rightmost) character in text that was compared:

The character c is not in the pattern

$s_0 \dots \dots S \dots \dots s_{n-1}$ (S not in pattern)

BAOBAB

BAOBAB

If there are no c 's in the pattern – eg., c is letter S in above example,
we can safely *shift the pattern by its entire length*.

How far to shift?

Case 2: The character c is in the pattern (but not the rightmost)

$s_0 \dots \mathbf{O} \dots s_{n-1}$ (O occurs once in pattern)
BAOBAB

BAOBAB

$s_0 \dots \mathbf{A} \dots s_{n-1}$ (A occurs twice in pattern)
BAOBAB

BAOBAB

If there are occurrences of character c in the pattern but it is not the last one there – e.g., c is letters O and A in above examples respectively – *the shift should align the rightmost occurrence of c in the pattern with the c in the text.*

Case 3: If c happens to be the last character in the pattern but there are no c 's among its other $m-1$ characters, the shift should be similar to that of Case 1: the pattern should be shifted by the entire pattern's length m .

$s_0 \dots \text{MER} \dots s_{n-1}$
LEADER
LEADER

Case 4: Finally, if c happens to be the last character in the pattern and there are other c 's among its first $m-1$ characters, the shift should be similar to that of Case 2: the rightmost occurrence of c among the first $m-1$ characters in the pattern should be aligned with the text's c .

$s_0 \dots \text{OR} \dots s_{n-1}$
REORDER
REORDER

- We can precompute shift sizes and store them in a table.
- The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters.

HORSPOL'S ALGORITHM

- The table's entries will indicate the shift sizes computed by the formula:

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

Example: For the pattern BAOBAB, all the table's entries will be equal to 6, except for the entries A, B, O, which will be 1, 2 and 3 respectively.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Figure: Shift Table contents for above example.

ALGORITHM FOR COMPUTING THE SHIFT TABLE ENTRIES

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

// Fills the shift table used by Horspool's algorithm

// Input: Pattern $P[0 \dots m-1]$ and an alphabet of possible characters

// Output: $\text{Table}[0 \dots \text{size}-1]$ indexed by the alphabet's characters and

// filled with shift sizes computed

initialize all the elements of Table with m

for $j \leftarrow 0$ **to** $m-2$ **do** $\text{Table}[P[j]] \leftarrow m - 1 - j$

return Table

- Initialize all the entries to the pattern's length m .
- Scan the pattern left to right repeating the following step $m-1$ times:
for the j^{th} character of pattern ($0 \leq j \leq m-2$), overwrite its entry in the table with $m-1-j$, which is the character's distance to the right end of the pattern. Since algorithm scans pattern from left to right, the last overwrite will happen for a character's rightmost occurrence.

HORSPool'S ALGORITHM

- Step 1** For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table.
- Step 2** Align the pattern against the beginning of text.
- Step 3** Repeat the following 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 text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c 's column of the shift table where c is the text's character currently aligned against the last character of the pattern and shift the pattern by $t(c)$ characters to the right along the text.

HORSPPOOL'S ALGORITHM

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

// Implements Horspool's algorithm for string matching

// Input: Pattern $P[0\dots m-1]$ and $T[0\dots n-1]$

// Output: The index of the left end of the first matching substring or

// -1 if there are no matches

ShiftTable($P[0\dots m-1]$) // generate Table of shifts

$i \leftarrow m-1$ // position of the pattern's right end

while $i \leq n-1$ **do**

$k \leftarrow 0$ // number of matched characters

while $k \leq m-1$ **and** $P[m-1-k] = T[i-k]$ **do**

$k \leftarrow k + 1$

if $k = m$

return $i - m + 1$

else $i \leftarrow i + \text{Table}[T[i]]$

return -1

HORSPPOOL'S ALGORITHM

Example: For the pattern BARBER, all the table's entries will be equal to 6, except for the entries E, B, R, and A, which will be 1, 2, 3, and 4 respectively.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
4	2	6	6	1	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6

The actual search in a particular text proceeds as follows:

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

HORSPPOOL'S ALGORITHM

Problems:

Trace the Horspool's algorithm for the following :

a) Text: **NOBODY_NOTICED_HIM**

Pattern: **NOT**

b) Text: **BAABABABCCA**

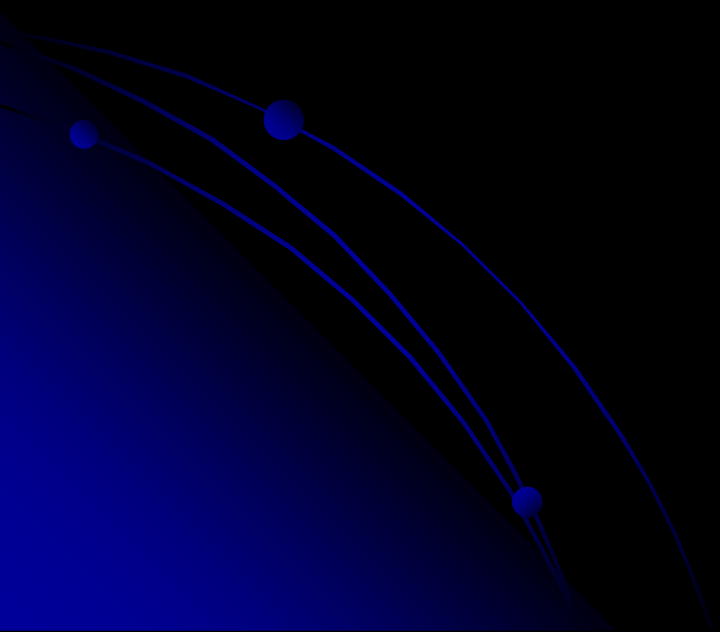
Pattern: **ABABC**

c) Text: **ANALYSIS_OF_ALGORITHMS**

Pattern: **ALGO**

HORSPPOOL'S ALGORITHM

- The worst-case efficiency of Horspool's algorithm is in $\Theta(nm)$.
- But for random texts, the efficiency is in $\Theta(n)$, and though in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm.



Boyer-Moore Algorithm

- Step 1 Construct the bad-symbol shift table for the given pattern and the alphabet used in both pattern and text.
- Step 2 Using the pattern, construct the good-suffix shift table.
- 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:
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 $k \geq 0$ character pairs are matched successfully.
If no characters match, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and shift the pattern to the right by $t_1(c)$.

If $0 < k < m$ characters are matched, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and entry $d_2(k)$ from the good-suffix table and shift the pattern to the right by

$$d = \begin{cases} d_1 & \text{if } k = 0 \\ \max \{d_1, d_2\} & \text{if } k > 0 \end{cases}$$

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

Boyer-Moore Algorithm Examples

Apply Boyer-Moore algorithm to search for the pattern BAOBAB in the text BESS_KNEW_ABOUT_BAOBABS

The bad-symbol table is as follows:

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

The good-suffix table is as follows:

k	<i>pattern</i>	d_2
1	BAO <u>B</u> A <u>B</u>	2
2	<u>B</u> A <u>O</u> B <u>A</u> B	5
3	<u>B</u> A <u>O</u> B <u>A</u> B	5
4	<u>B</u> A <u>O</u> B <u>A</u> B	5
5	<u>B</u> A <u>O</u> B <u>A</u> B	5

Boyer-Moore Algorithm Examples

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



Boyer-Moore Algorithm Examples

Apply Boyer-Moore algorithm to search for the following:

1. Pattern: EXAMPLE

Text: HERE IS A SIMPLE EXAMPLE

2. Pattern: AT_THAT

Text: WHICH_FINALLY_HALTS.__ AT_THAT.

The worst-case efficiency of the algorithm to search for the first occurrence of the pattern is known to be linear.

End of Chapter 7

