RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Go, change the world

Academic year 2024-2025 (Even Sem)

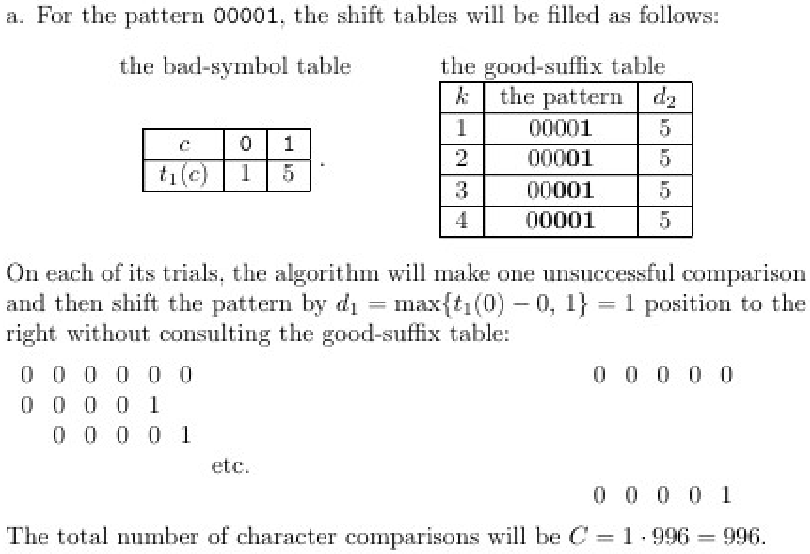
DEPARTMENT OF **COMPUTER SCIENCE & ENGINEERING**

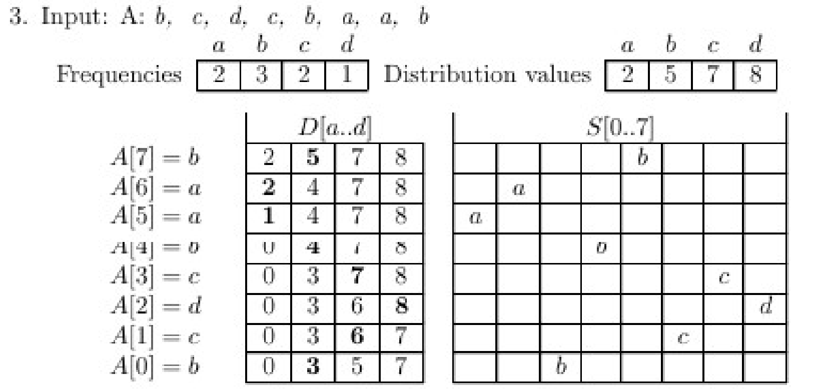
Date: May 2025	**CIE II**	Maximum Marks: 50 + 10
Course Code: CD343AI	Sem: 4th UG	Duration: 120 Minutes

Design and Analysis of Algorithms

(Common to CS/CD/CY/IS/AIML)

Note: - Answer all the questions.

<table>
<tr><th>S.No</th><th>Part A</th><th>M</th><th>BT</th><th>CO</th></tr>
<tr><td>1.1</td><td>__________ and __________ are examples of sorting algorithms that can achieve linear time complexity under certain conditions.
<b>Insertion Sort</b> and <b>Distribution Counting Sort</b></td><td>02</td><td>1</td><td>1</td></tr>
<tr><td>1.2</td><td>Consider the pattern 00001 and a binary text consisting of 1000 zeros (000...0). Using the Boyer-Moore algorithm, analyze and calculate the number of character comparisons made during the pattern search.

a. For the pattern 00001, the shift tables will be filled as follows:
the bad-symbol table
<table><tr><td>c</td><td>0</td><td>1</td></tr><tr><td>$t_1(c)$</td><td>1</td><td>5</td></tr></table>.
the good-suffix table
<table><tr><th>k</th><th>the pattern</th><th>d_2</th></tr><tr><td>1</td><td>0000<b>1</b></td><td>5</td></tr><tr><td>2</td><td>000<b>01</b></td><td>5</td></tr><tr><td>3</td><td>00<b>001</b></td><td>5</td></tr><tr><td>4</td><td>0<b>0001</b></td><td>5</td></tr></table>
On each of its trials, the algorithm will make one unsuccessful comparison and then shift the pattern by $d_1 = \max\{t_1(0) - 0, 1\} = 1$ position to the right without consulting the good-suffix table:
0 0 0 0 0 0   0 0 0 0 0
0 0 0 0 1
   0 0 0 0 1
           etc.
                          0 0 0 0 1
The total number of character comparisons will be $C = 1 \cdot 996 = 996$.</td><td>02</td><td>3</td><td>2</td></tr>
<tr><td>1.3</td><td>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.</td><td>02</td><td>3</td><td>3</td></tr>
</table>

<table>
<tr>
<td></td>
<td>
3. Input: A: b, c, d, c, b, a, a, b
<table>
<tr><td></td><td><i>a</i></td><td><i>b</i></td><td><i>c</i></td><td><i>d</i></td><td></td><td><i>a</i></td><td><i>b</i></td><td><i>c</i></td><td><i>d</i></td></tr>
<tr><td>Frequencies</td><td>2</td><td>3</td><td>2</td><td>1</td><td>Distribution values</td><td>2</td><td>5</td><td>7</td><td>8</td></tr>
</table>
<table>
<tr><td></td><td colspan="4">$D[a..d]$</td><td colspan="8">$S[0..7]$</td></tr>
<tr><td>$A[7] = b$</td><td>2</td><td><b>5</b></td><td>7</td><td>8</td><td></td><td></td><td></td><td></td><td><i>b</i></td><td></td><td></td><td></td></tr>
<tr><td>$A[6] = a$</td><td><b>2</b></td><td>4</td><td>7</td><td>8</td><td></td><td><i>a</i></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>$A[5] = a$</td><td><b>1</b></td><td>4</td><td>7</td><td>8</td><td><i>a</i></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>$A[4] = b$</td><td>0</td><td><b>4</b></td><td>7</td><td>8</td><td></td><td></td><td></td><td><i>b</i></td><td></td><td></td><td></td><td></td></tr>
<tr><td>$A[3] = c$</td><td>0</td><td>3</td><td><b>7</b></td><td>8</td><td></td><td></td><td></td><td></td><td></td><td></td><td><i>c</i></td><td></td></tr>
<tr><td>$A[2] = d$</td><td>0</td><td>3</td><td>6</td><td><b>8</b></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td><i>d</i></td></tr>
<tr><td>$A[1] = c$</td><td>0</td><td>3</td><td><b>6</b></td><td>7</td><td></td><td></td><td></td><td></td><td></td><td><i>c</i></td><td></td><td></td></tr>
<tr><td>$A[0] = b$</td><td>0</td><td><b>3</b></td><td>5</td><td>7</td><td></td><td></td><td><i>b</i></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>How can you determine the number of paths of length two between any two vertices in a graph using its adjacency matrix? Provide an example to illustrate this process.

Multiply adjacency matrix by itself ----- 1M
Example ----- 1M</td>
<td>02</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>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 a chromosome
TCCTATTCTT

<b>A-5, C-2, G-10, T-1</b></td>
<td>01</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1.6</td>
<td>Justify why is Binary search not considered a good example of a pre-sorting technique.
The brute-force solution/sequential search takes n comparisons in the worst case.
Binary search, which requires pre-sorting takes (n log n) comparisons in the worst case</td>
<td>01</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td><b>Part B</b></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>You are given the results of a completed round-robin tournament in which n teams played each other exactly once. Each game ended with a victory for one of the teams, and these results are represented in the form of a directed graph (digraph a) — where an edge from team U to team V indicates that team U defeated team V.
Design and write an algorithm using Depth First Search(DFS) to list the teams in a sequence such that no team in the list has lost to the team immediately after it. Analyze the time efficiency of your algorithm. Then, apply your algorithm to digraph (a) and show the step-by-step solution.</td>
<td>10</td>
<td>3</td>
<td>4</td>
</tr>
</table>

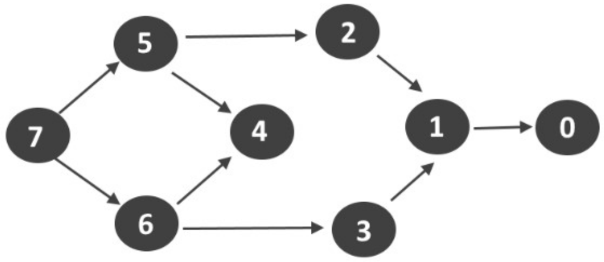


Fig. a

Algorithm(4M)

```
TOPOSORT(G):
   Input: A directed acyclic graph G = (V, E)
   Output: A list representing topological ordering of vertices

   1. Create an empty stack `S` to store the topological order
   2. Mark all vertices in V as unvisited
   3. For each vertex v in V:
         If v is not visited:
            DFS(v)
   4. Return the contents of stack S (reversed)

Procedure DFS(v):
   1. Mark v as visited
   2. For each neighbor u of v:
         If u is not visited:
            DFS(u)
   3. Push v onto stack S
```

Time complexit :O(V+E) for adjacency list O(V^2) for adjacency matrix **(2M)**

step-by-step solution (4M)

7 (1, 8)
5(2, 5,)
2 (3, 3)
1 (4, 2)
0 (5, 1)
4 (6, 4)
6(7, 7)
3(8, 6)

Reverse the pop order will give the final team sequence: 7, 6, 3, 5, 4, 2, 1, 0

<table>
<tr><td>2 a</td><td>

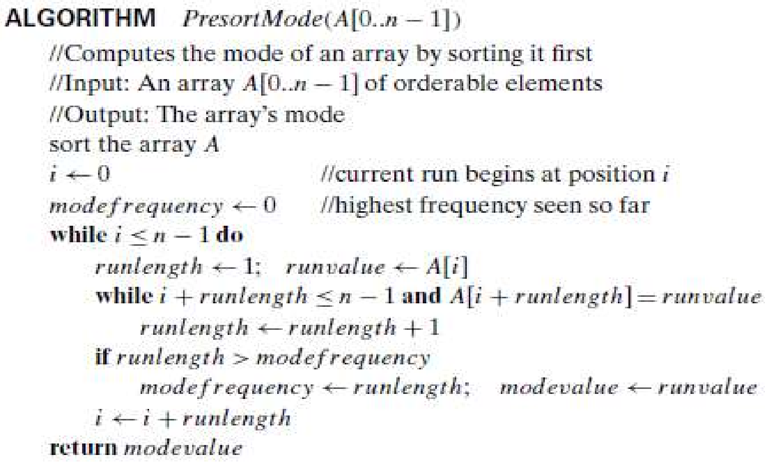
Design a decrease-by-one algorithm for generating the power set of a set of n elements. (The power set of a set S is the set of all the subsets of S, including the empty set and S itself.)

```
Algorithm PowerSet(S)
Input: A set S
Output: All subsets of S (the power set)

1. If S is empty:
2.     Return { {} }   // Only the empty set
3. Remove one element x from S
4. SubsetsWithoutX = PowerSet(S)   // Recursively find subsets of smaller set
5. SubsetsWithX = add x to each subset in SubsetsWithoutX
6. Return SubsetsWithoutX ∪ SubsetsWithX
```

</td><td>5</td><td>3</td><td>4</td></tr>
<tr><td>2 b</td><td>

Design a presorting-based algorithm for computing the mode in a given list of numbers and determine its efficiency



```
ALGORITHM  PresortMode(A[0..n − 1])
    //Computes the mode of an array by sorting it first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: The array's mode
    sort the array A
    i ← 0                      //current run begins at position i
    modefrequency ← 0          //highest frequency seen so far
    while i ≤ n − 1 do
        runlength ← 1;   runvalue ← A[i]
        while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
            runlength ← runlength + 1
        if runlength > modefrequency
            modefrequency ← runlength;   modevalue ← runvalue
        i ← i + runlength
    return modevalue
```

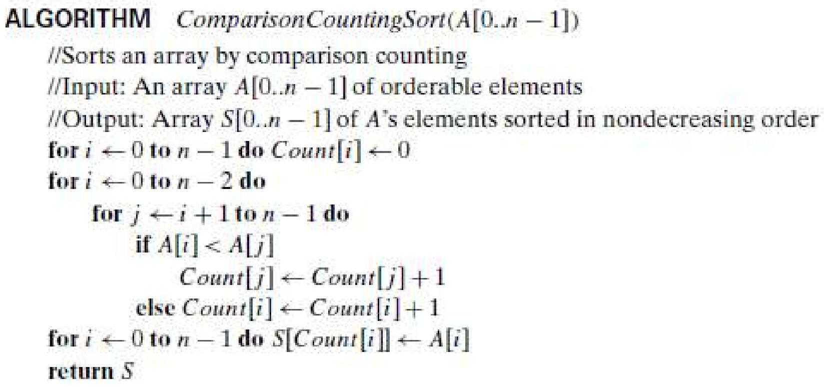
</td><td>5</td><td>3</td><td>3</td></tr>
<tr><td>3 a</td><td>

Write the algorithm for Sorting by Comparison Counting and apply it to the following list of elements: 62, 31, 84, 96, 19, 47. Show the complete trace of the sorting process, including the count array and the resulting sorted list.

Show the complete trace of the sorting process, including the count array and the resulting sorted list.

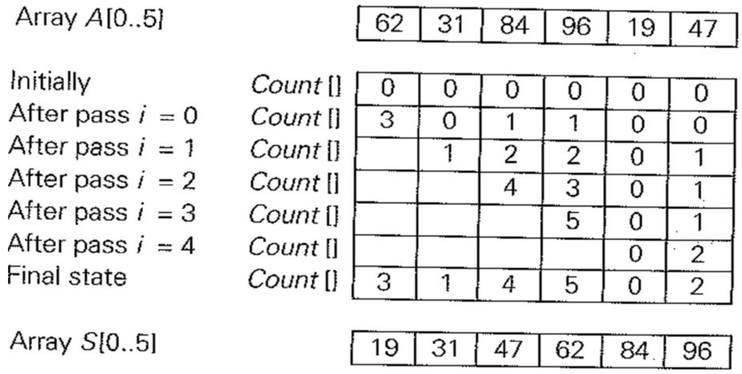
Write the Algorithm for sorting by comparison counting. ----------- 3M

</td><td>05</td><td>3</td><td>4</td></tr>
</table>



```
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 in nondecreasing order
    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
```

Apply the same for the list : 62, 31, 84, 96, 19, 47 and Show the complete trace. ------------ 3M



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

3 b	Consider the problem of finding the smallest and largest elements in an array of n numbers. Design a presorting-based algorithm for solving this problem and determine its efficiency class. Input: An array A[0...n−1] of n numbers Output: The smallest and largest elements in the array 1. Sort the array A using an efficient comparison-based sorting algorithm (e.g., Merge Sort, Heap Sort, or Quick Sort) 2. Return A[0] as the smallest element 3. Return A[n−1] as the largest element • Sorting takes **O(n log n)** time using efficient algorithms (like Merge Sort or Heap Sort). • Accessing the first and last elements takes **O(1)** time.	05	3	4

4 Write the Heap Sort algorithm along with the heapify procedure used to maintain the max-heap property. Construct a Max Heap using bottom-up heap construction for the following list of elements 2, 9, 7, 6, 5, 8. Show the complete trace of the heap construction pictorially at each step. Briefly discuss the time complexity of Heap Sort algorithm.

Heap sort algorithm – 5M

Algorithm Heap Sort

1. Build a Max-Heap from the input array using the bottom-up heapify approach.
2. Repeat the following until the heap size is greater than 1:
   a. Swap the first element (maximum) with the last element.
   b. Reduce the heap size by 1.
   c. Heapify the root element to restore the max-heap property.

Algorithm Heapify

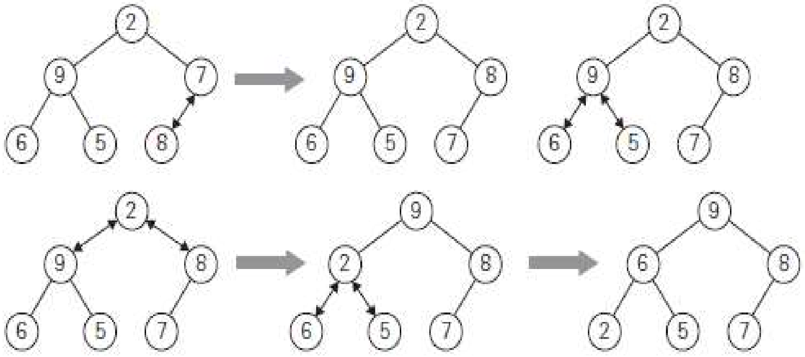
Input: Array A[1..n] (1-based indexing)

Output: A is transformed into a max-heap

```
for i = floor(n/2) downto 1:
    siftDown(A, i, n)

Function siftDown(A, i, n):
    k = i
    v = A[k]
    heap = false
    while not heap and 2*k ≤ n:
        j = 2*k
        if j < n and A[j] < A[j+1]:
            j = j + 1
        if v ≥ A[j]:
            heap = true
        else:
            A[k] = A[j]
            k = j
    A[k] = v
```

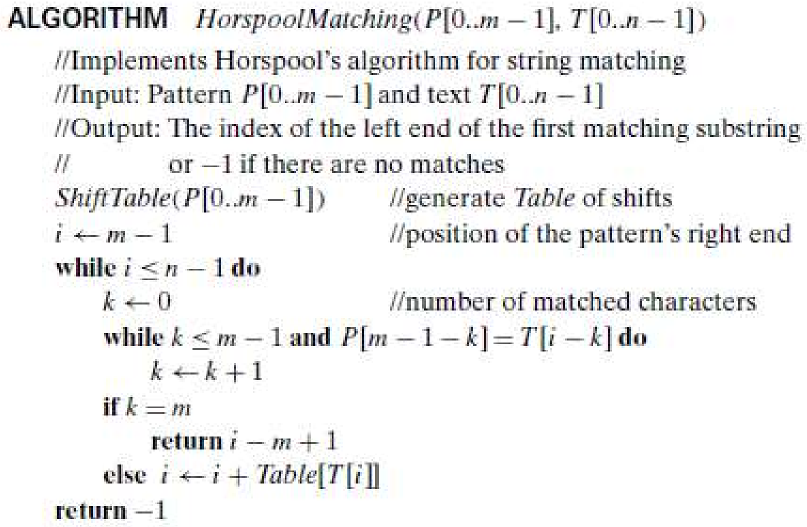
Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. ----3M



The heap construction stage of the algorithm is in O(n) and sort is O(n log n) we get O(n) + O(n log n) = O(n log n). **(2M)**

5	Write the pseudocode for Horspool's algorithm to search for the pattern in the text. Summarize the efficiency of the algorithm in comparison with brute force approach and its Boyer-Moore algorithm. Demonstrate Brute force, Horspool and its Boyer-Moore approach for following example. Text: ababcababcabc Pattern: abc	10	3	3

Write the pseudocode for Horspool's algorithm to search for the pattern in the text. ------- 3M

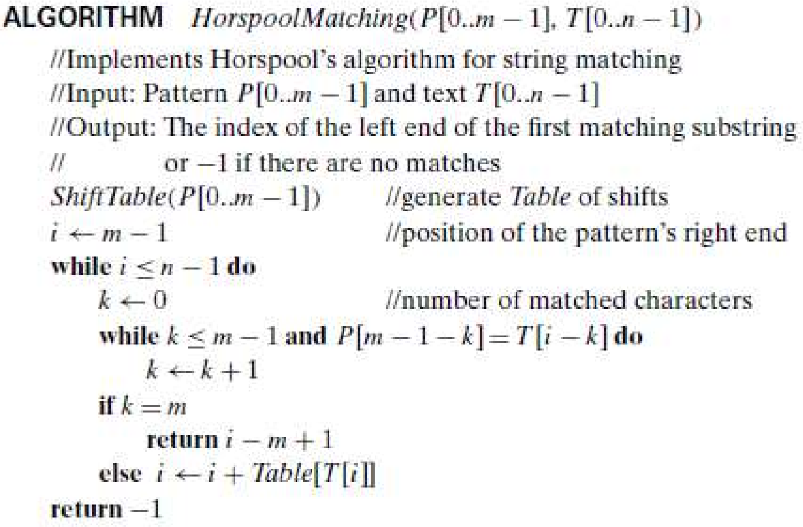


```
ALGORITHM  HorspoolMatching(P[0..m − 1], T[0..n − 1])
    //Implements Horspool's algorithm for string matching
    //Input: Pattern P[0..m − 1] and text T[0..n − 1]
    //Output: The index of the left end of the first matching substring
    //          or −1 if there are no matches
    ShiftTable(P[0..m − 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
```

Summarize the efficiency of the algorithm in comparison with brute force approach and its predecessor algorithm. ------- 2M

- A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in *O(nm)*.
- But for random texts, it is in Θ*(n)*, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm.

Write the pseudocode for Horspool's algorithm to search for the pattern in the text. ------- 3M



```
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
    $i \leftarrow m-1$                 //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 $i \leftarrow i + Table[T[i]]$
    return $-1$
```

Summarize the efficiency of the algorithm in comparison with brute force approach and its predecessor algorithm. ------- 2M

- A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in *O(nm)*.
- But for random texts, it is in Θ*(n)*, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm.
- Horspool is **much faster than brute force** in practice and **easier to implement** than Boyer-Moore, though slightly less efficient than Boyer-Moore in some cases. It strikes a good **balance between simplicity and performance**.

solution

BT-Blooms Taxonomy, CO-Course Outcomes, M-Marks

Marks Distribution	Particulars		CO1	CO2	CO3	CO4	CO5	L1	L2	L3	L4	L5	L6
	Quiz/ Test	Max Marks	02	12	21	25		2	10	48			

Course Outcomes: After completing the course, the students will be able to: -	
CO 1	Apply knowledge of computing and mathematics to algorithm analysis and design.
CO 2	Analyze a problem and identify the computing requirements appropriate for a solution.
CO 3	Apply algorithmic principles and computer science theory to the modeling for evaluation of computer-based solutions in a way that demonstrates comprehension of the trade-offs involved in design choices.
CO 4	Investigate and use optimal design techniques, development principles, skills and tools in the construction of software solutions of varying complexity.
CO 5	Demonstrate critical, innovative thinking, and display competence in solving engineering problems.
CO 6	Exhibit effective communication and engage in continuing professional development through experiential learning.