SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 OF THE UGC ACT, 1956)
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

School of Computing

CSE306 - DESIGN & ANALYSIS OF ALGORITHMS LABORATORY

List of Exercises

1. Programs on Comparison Sorts - Merge Sort, Quick Sort and Heap Sort
2. Programs on Linear Sorting techniques - Counting Sort, Radix Sort and Bucket Sort
3. Applications of Divide and Conquer approach - Maximum Subarray problem
4. Applications of Dynamic Programming - Matrix Chain Multiplication
5. Applications of Dynamic Programming - Longest Common Subsequence
6. Applications of Greedy Technique - Text data compression using Huffman Codes
7. Programs on Graphs - Topological Sort of Directed Acyclic Graph
8. Programs on Graphs - Minimum Spanning Tree using Prim’s and Kruskal’s algorithms
9. Programs on Graphs - Single Source Shortest paths using Bellman-Ford algorithm
10. Programs on Graphs - All-Pairs Shortest Paths using Floyd-Warshall algorithm
11. String Matching - with Rabin-Karp algorithm
12. String Matching - Knuth-Morris-Pratt (KMP) algorithm

General Instructions:

For every exercise,

- Input should be generated as random numbers and it should be stored as a separate file.
- The generated file has to give as input for the program.
- Output has to be written as separate file.
- Graph has to be drawn with respect to size of the input and time taken.

Evaluation Procedure:

Lab Performance Assessment

S. No	Metrics	Marks
1	Student's Performance	5
2	Output Verification	2
3	Record	3
4	Total	10

Internal Mark Assessment

S. No	Metrics	Marks
1	Model - I	15
2	Model - II	15
3	Lab Performance Assessment	20
4	Total	50

Ex. No: 01 Comparison Sorting

Objective: To analyze the comparison sorting algorithms by counting the number of swaps performed

Pre-Lab Exercise: Implementation knowledge on insertion sort

1.Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

2.Quick Sort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

3.Heap Sort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Ex. No: 2 Linear Sorting Techniques

Objective: To compare the Non-comparison sorting algorithms with comparison sorting algorithms with respect to the space time trade off

Pre-Lab Exercise: Simple program for time, space trade off

1. Counting Sort

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Note: Each element in the n-element array ‘A’ has nonnegative integer no larger than ‘k’ and ‘B’ is the sorted output array

2. Radix Sort

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Note: Each element in the n-element array ‘A’ has ‘d’ digits, where digit ‘1’ is the lowest-order digit and digit ‘d’ is the highest-order digit.

3. Bucket Sort

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Note: Each element in the n-element array ‘A’ satisfies $0<A[i]<1$

Ex. No: 03 Divide and Conquer approach- Maximum Subarray problem

Objective: To compare the divide conquer approach of maximum subarray problem with its brute force approach

Pre-Lab Exercise: Program to solve maximum subarray problem using brute force approach

1.Solving the Maximum Subarray problem using Divide and Conquer

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Ex. No: 04 Dynamic Programming - Matrix Chain Multiplication

Objective: To compare the dynamic programming approach of matrix chain multiplication problem with its brute force approach

Pre-Lab Exercise: Program to solve matrix chain multiplication problem using brute force approach

1. Matrix chain multiplication problem using dynamic programming

```
Algorithm  Bottom_up_MCM(P)
1.n=p. length-1
2.for i=1 to n
3.    m [i, i] =0
4.for l=1 to n-1
5.    for i=1 to n-l
6.        j=i+l
7.        q=∞
8.        for k=i to j-1
9.            if q>m [i, k] +m [k+1, j] +P_{i-1}P_kP_j
10.               q= m [i, k] +m [k+1, j] +P_{i-1}P_kP_j
11.               s [i, j] =k
12.               m [i, j] =q
13. return(m,s)
```

```
Algorithm OPT_PAR(s,i,j)
    if i==j then
        print Ai
    else
        print "("
    OPT_PAR(s,i,s[i,j])
    OPT_PAR(s,s[i,j]+1,j)
    print ")"
    return
```

Ex. No: 05 Dynamic Programming - Longest Common Subsequence

Objective: To compare the dynamic programming approach of longest Common Subsequence (LCS) problem with its brute force approach

Pre-Lab Exercise: Solve LCS problem using brute force approach

1. problem using dynamic programming

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Ex. No: 06 Applications of Greedy Technique - Text data compression using Huffman Codes

Objective: To compare the variable length encoding with fixed length encoding

Pre-Lab Exercise: Program for fixed length encoding

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

Ex. No: 07 Programs on Graphs - Topological Sort of Directed Acyclic Graph

Objective: Find the linear ordering of all vertices in Directed Acyclic Graph (DAG).

Pre-Lab Exercise: Knowledge on Depth first search algorithm.

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1              // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]        // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // blacken u; it is finished
time = time + 1
u.f = time
```

Ex. No: 08 Programs on Graphs - Minimum Spanning Tree using Prim’s and Kruskal’s algorithms

Objective: Find the Minimum Spanning Tree by using Prim’s and Kruskal’s algorithms.

Pre-Lab Exercise: Knowledge on generic minimum spanning tree.

Prim's Algorithm

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

Ex. No: 09 Programs on Graphs - Single Source Shortest paths using Bellman-Ford algorithm

Objective: Computes shortest path from a single source vertex to all of the other vertices in a weighted digraph.

Pre-Lab Exercise: Graph representation as adjacency list and adjacency matrix.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Ex. No: 10 Programs on Graphs - All-Pairs Shortest Paths using Floyd-Warshall algorithm

Objective: Finding shortest paths between all pairs of vertices in a graph.

Pre-Lab Exercise: A recursive solution to the all-pairs shortest-paths problem.

FLOYD-WARSHALL(W)

$n = W.rows$

$D^{(0)} = W$

for $k = 1$ **to** n

  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix

  for $i = 1$ **to** n

    for $j = 1$ **to** n

      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

return $D^{(n)}$

Ex. No: 11 String Matching - with Rabin-Karp algorithm

Objective: For a given text and patten, find out any valid shifts that are found in the text.

Pre-Lab Exercise: Implement naive string-matching algorithm.

```
RABIN-KARP-MATCHER(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m                        // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (dt_0 + T[i]) mod q
for s = 0 to n - m                    // matching
    if p == t_s
        if P[1..m] == T[s + 1..s + m]
            print "Pattern occurs with shift" s
    if s < n - m
        t_(s+1) = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Ex. No: 12 String Matching - Knuth-Morris-Pratt (KMP) algorithm

Objective: For a given text and patten, find out any valid shifts that are found in the text.

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

Additional Exercises

1. Programs on Strassen’s algorithm -Matrix Multiplication.
2. Applications of Dynamic Programming – Rod Cutting.
3. Applications of Greedy Technique – Activity Selection Problem.
4. Programs on Graphs – Dijikstra’s Algorithm.
5. All-Pairs Shortest Paths – Johnson’s algorithm for sparse graphs.