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

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

2.Quick Sort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

3.Heap Sort

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

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

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       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)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      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)
1  for i = 1 to d
2      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)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊n A[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  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)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  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)
1.      if i==j then
2.              print Ai
3.      else
4.              print "("
5.      OPT_PAR(s,i,s[i,j])
6.      OPT_PAR(s,s[i,j]+1,j)
7.      print ")"
8.      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)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  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)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  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)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

```
DFS(G)
1  for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6       if u.color == WHITE
7            DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1              // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]        // explore edge (u, v)
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK              // blacken u; it is finished
 9  time = time + 1
10  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)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  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)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  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)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4       let D^(k) = (d_ij^(k)) be a new n × n matrix
5       for i = 1 to n
6            for j = 1 to n
7                 d_ij^(k) = min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
8  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)
 1  n = T.length
 2  m = P.length
 3  h = d^(m−1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                    // preprocessing
 7       p = (dp + P[i]) mod q
 8       t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n − m                // matching
10       if p == t_s
11            if P[1..m] == T[s + 1..s + m]
12                 print "Pattern occurs with shift" s
13       if s < n − m
14            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)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6       while k > 0 and P[k + 1] ≠ P[q]
 7            k = π[k]
 8       if P[k + 1] == P[q]
 9            k = k + 1
10       π[q] = k
11  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.