Analysis and Design of Algorithms - Laboratory Programs Summary

Program Overview Table

Here's a brief explanation of each formula:

Selection Sort: Finds minimum element by comparing with current minimum index
Topological Sort: Reduces in-degree and adds vertices with zero in-degree to processing stack
Merge Sort: Classic divide-and-conquer recurrence relation
Quick Sort: Partitioning logic that separates elements around pivot
0/1 Knapsack: Dynamic programming recurrence for optimal substructure
Floyd's Algorithm: Relaxation formula considering all intermediate vertices
Warshall's Algorithm: Transitive closure condition using logical AND
Prim's Algorithm: Greedy selection of minimum weight edge from visited to unvisited vertices
Kruskal's Algorithm: Union-Find condition to avoid cycles in MST
Dijkstra's Algorithm: Relaxation condition for shortest path updates
N-Queens: Conflict detection for same row/column and diagonal attacks

Key observations:

Sorting algorithms (Selection, Merge, Quick) work on integer arrays with varying time complexities
Graph algorithms (Topological, Floyd's, Warshall's, Prim's, Kruskal's, Dijkstra's) use adjacency/cost matrices
Dynamic Programming (Knapsack) optimizes resource allocation problems
Backtracking (N-Queens) explores all possible solutions systematically

S.No	Program/Algorithm Name	Time Complexity	Input Type	Output	Main Formula/Condition
1	Selection Sort	$O(n^2)$	Array of integers	Sorted array + execution time	`if (array[j] < array[min]) min = j`
2	Topological Sort	O(V + E)	Adjacency matrix	Topological sequence	`indegree[v]--; if(indegree[v]==0) add to stack`
3	Merge Sort	O(n log n)	Array of integers	Sorted array + execution time	`T(n) = 2T(n/2) + O(n)`
4	Quick Sort	O(n log n) avg, $O(n^2)$ worst	Array of integers	Sorted array + execution time	`partition: while(a[i] <= a[key]) && while(a[j] > a[key])`
5	0/1 Knapsack (DP)	O(nW)	Items with profit & weight + capacity	Selected items + max profit	`dp[i][w] = max(dp[i-1][w], p[i] + dp[i-1][w-weight[i]])`
6a	Floyd's Algorithm	$O(V^3)$	Weighted adjacency matrix	Shortest distances between all pairs	`dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`
6b	Warshall's Algorithm	$O(V^3)$	Adjacency matrix	Path matrix (transitive closure)	`if(path[i][k]==1 && path[k][j]==1) then path[i][j]=1`
7	Prim's Algorithm (MST)	$O(V^2)$	Cost adjacency matrix	Minimum spanning tree + cost	`min_edge = min(cost[i][j]) where visited[i]!=0 && visited[j]==0`
8	Kruskal's Algorithm (MST)	O(E log E)	Cost adjacency matrix	Minimum spanning tree + cost	`if(find(u) != find(v)) then union(u,v)`
9	Dijkstra's Algorithm	$O(V^2)$	Cost matrix + source vertex	Shortest paths from source	`dist[w] = min(dist[w], dist[u] + cost[u][w])`
10	N-Queens Problem	O(N!)	Number of queens (N)	All possible solutions	`conflict = (a[i]==a[pos]) OR (abs(a[i]-a[pos])==abs(i-pos))`

Algorithm Descriptions (5 lines each)

1. Selection Sort Algorithm

1. Find the minimum element in the unsorted portion of the array
2. Swap it with the first element of the unsorted portion
3. Move the boundary of the unsorted portion one position to the right
4. Repeat steps 1-3 until the entire array is sorted
5. The algorithm maintains two portions: sorted (left) and unsorted (right)

2. Topological Sort Algorithm

1. Calculate the in-degree of all vertices in the directed graph
2. Initialize a stack with all vertices having in-degree 0
3. Pop a vertex from stack, add it to the topological order
4. Reduce in-degree of all adjacent vertices by 1
5. If any adjacent vertex's in-degree becomes 0, push it to stack; repeat until stack is empty

3. Merge Sort Algorithm

1. Divide the array into two halves recursively until single elements remain
2. Conquer by merging the divided subarrays in sorted order
3. Compare elements from both subarrays and place smaller element first
4. Continue merging until all elements are combined in sorted order
5. The algorithm follows divide-and-conquer paradigm with guaranteed O(n log n) performance

4. Quick Sort Algorithm

1. Choose a pivot element (usually the first element)
2. Partition the array such that elements smaller than pivot are on left, larger on right
3. Place the pivot in its correct sorted position
4. Recursively apply quick sort to the left and right subarrays
5. The algorithm is in-place but performance depends on pivot selection

5. 0/1 Knapsack (Dynamic Programming) Algorithm

1. Create a 2D table where dp[i][w] represents maximum profit using first i items with weight limit w
2. For each item, decide whether to include it or not based on maximum profit
3. If item weight > current capacity, exclude the item
4. Otherwise, take maximum of (include item + remaining capacity profit) or (exclude item profit)
5. Backtrack through the table to find which items were selected for optimal solution

6a. Floyd's Algorithm (All-Pairs Shortest Path)

1. Initialize distance matrix with direct edge weights (infinity for no direct path)
2. For each vertex k, consider it as an intermediate vertex
3. For each pair of vertices (i,j), check if path through k is shorter
4. Update distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
5. After considering all vertices as intermediates, matrix contains shortest paths between all pairs

6b. Warshall's Algorithm (Transitive Closure)

1. Initialize path matrix same as adjacency matrix
2. For each vertex k, consider it as an intermediate vertex
3. For each pair of vertices (i,j), check if there's a path from i to j through k
4. If path[i][k] = 1 AND path[k][j] = 1, then set path[i][j] = 1
5. Final matrix shows reachability between all pairs of vertices

7. Prim's Algorithm (Minimum Spanning Tree)

1. Start with any vertex and mark it as visited
2. Find the minimum weight edge connecting visited to unvisited vertex
3. Add this edge to MST and mark the new vertex as visited
4. Repeat step 2-3 until all vertices are included in MST
5. The algorithm grows the MST one vertex at a time using greedy approach

8. Kruskal's Algorithm (Minimum Spanning Tree)

1. Sort all edges in ascending order of their weights
2. Initialize each vertex as a separate set (using Union-Find data structure)
3. For each edge in sorted order, check if it connects vertices from different sets
4. If yes, add edge to MST and union the two sets
5. Continue until MST has (V-1) edges, ensuring no cycles are formed

9. Dijkstra's Algorithm (Single-Source Shortest Path)

1. Initialize distance to source as 0 and all other vertices as infinity
2. Mark all vertices as unvisited and source as current vertex
3. Find unvisited vertex with minimum distance and mark it as visited
4. Update distances of all unvisited neighbors if shorter path is found
5. Repeat steps 3-4 until all vertices are visited or shortest path to destination is found

10. N-Queens Problem (Backtracking)

1. Place queens one by one in different columns starting from leftmost column
2. For each queen, check if placement conflicts with previously placed queens
3. Check for conflicts in same row, column, and both diagonals
4. If conflict exists, backtrack and try next position for current queen
5. If all queens are placed successfully, record the solution; continue to find all solutions