AOA Practical Exam

1. Implementation of Selection sort and its analysis.

```c
#include <stdio.h>

void selection(int arr[], int n) {
    int i, j, small;

    for (i = 0; i < n - 1; i++) {
        small = i;

        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[small])
                small = j;

        int temp = arr[small];
        arr[small] = arr[i];
        arr[i] = temp;
    }
}

void printArr(int a[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}

int main() {
    printf("Number of elements in array:");
    int n;
    scanf("%d", &n);

    int a[n];
    printf("Enter array elements:");
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting array elements are - \n");
    printArr(a, n);

    selection(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
```

```
    return 0;
}
```

Output:

Number of elements in array:6

Enter array elements : 12 31 25 8 32 17

Before sorting array elements are -

12 31 25 8 32 17

After sorting array elements are -

8 12 17 25 31 32

Time Complexity:

Best Case : $O(n^2)$

Average Case : $O(n^2)$

Worst Case : $O(n^2)$

Space Complexity : O(1)

Stable : Yes

2. Implementation of Insertion sort and its analysis.

```c
#include <stdio.h>

/* function to sort an array with insertion sort */
void insert(int a[], int n) {
    int i, j, temp;
    for (i = 1; i < n; i++)
    {
        temp = a[i];
        j = i - 1;

        /* Move the elements greater than temp to one position ahead from
their current position*/
        while (j >= 0 && temp <= a[j]) {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = temp;
    }
}

void printArr(int a[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}

int main() {
    printf("Number of elements in array:");
    int n;
    scanf("%d", &n);

    int a[n];
    printf("Enter array elements:");
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting array elements are - \n");
    printArr(a, n);

    insert(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
```

```
    return 0;
}
```

Output :

Number of elements in array:6

Enter array elements:12 31 25 8 32 17

Before sorting array elements are -

12 31 25 8 32 17

After sorting array elements are -

8 12 17 25 31 32

Time Complexity :

Best Case : O(n)

Average Case : $O(n^2)$

Worst Case : $O(n^2)$

Space Complexity : O(1)

Stable : Yes

3. Implementation of Merge sort and its analysis.

```c
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

```
    /* Copy the remaining elements of L[], if there are any */
    while (i < n1)  {
        arr[k] = L[i];
        i++;
        k++;
    }



    /* Copy the remaining elements of R[], if there are any */
    while (j < n2)  {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
```

```c
/* Driver code */
int main()
{
    printf("Number of elements in array:");
    int n;
    scanf("%d", &n);

    int arr[n];
    printf("Enter array elements:");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("Given array is \n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("\nSorted array is \n");
    printArray(arr, n);
    return 0;
}
```

Output :

Number of elements in array:6

Enter array elements:12 31 25 8 32 17

Given array is

12 31 25 8 32 17

Sorted array is

8 12 17 25 31 32

Time Complexity :

Best Case : O(n*log n)

Average Case : O(n*log n)

Worst Case : O(n*log n)

Space Complexity : O(n)

Stable : Yes

4. Implementation of Quick sort and its analysis.

```c
#include <stdio.h>
/* function that consider last element as pivot,
place the pivot at its exact position, and place
smaller elements to left of pivot and greater
elements to right of pivot.  */
int partition(int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        // If current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i + 1];
    a[i + 1] = a[end];
    a[end] = t;
    return (i + 1);
}

/* function to implement quick sort */
/* a[] = array to be sorted, start = Starting index, end = Ending index */
void quick(int a[], int start, int end)
{
    if (start < end)
    {
        int p = partition(a, start, end); // p is the partitioning index
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}
```

```c
void printArr(int a[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}

int main() {
    printf("Number of elements in array:");
    int n;
    scanf("%d", &n);

    int a[n];
    printf("Enter array elements:");
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting array elements are - \n");
    printArr(a, n);
    quick(a, 0, n - 1);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Output :

Number of elements in array: 6

Enter array elements: 12 31 25 8 32 17

Before sorting array elements are -

12 31 25 8 32 17

After sorting array elements are -

8 12 17 25 31 32

Time Complexity :

Best Case : O(n*logn)

Average Case : O(n*logn)

Worst Case : $O(n^2)$

Space Complexity : O(n*logn)

Stable : No

5. Implementation of Fractional Knapsack Problem.

```
#include <stdio.h>

void simple_fill(int n, int W, int c[], int v[])
{
    int cur_w;
    float tot_v = 0;
    int i, maxi;
    int used[10];

    for (i = 0; i < n; ++i)
        used[i] = 0; /* I have not used the ith object yet */

    cur_w = W;
    while (cur_w > 0) { /* while there's still room*/
        /* Find the best object */
        maxi = -1;
        for (i = 0; i < n; ++i)
            if ((used[i] == 0) &&
                ((maxi == -1) || ((float)v[i]/c[i] >
(float)v[maxi]/c[maxi])))
                maxi = i;

        used[maxi] = 1; /* mark the maxi-th object as used */
        cur_w -= c[maxi]; /* with the object in the bag, I can carry less */
        tot_v += v[maxi];
        if (cur_w >= 0)
            printf("Added object %d (%d$, %dKg) completely in the bag. Space
left: %d.\n", maxi + 1, v[maxi], c[maxi], cur_w);
        else {
            printf("Added %d%% (%d$, %dKg) of object %d in the bag.\n",
(int)((1 + (float)cur_w/c[maxi]) * 100), v[maxi], c[maxi], maxi + 1);
            tot_v -= v[maxi];
            tot_v += (1 + (float)cur_w/c[maxi]) * v[maxi];
        }
    }
    printf("Filled the bag with objects worth %.2f$.\n", tot_v);
}

int main()
{
    int n, W;
    printf("Number of objects: ");
    scanf("%d", &n);
```

```
    printf("Enter the cost of each object: ");
    int c[n];
    for (int i = 0; i < n; i++)
        scanf("%d", &c[i]);

    printf("Enter the value of each object: ");
    int v[n];
    for (int i = 0; i < n; i++)
        scanf("%d", &v[i]);

    printf("Enter the maximum weight of the bag: ");
    scanf("%d", &W);

    simple_fill(n, W, c, v);
    return 0;
}
```

Output :

Number of objects: 5

Enter the cost of each object: 12 1 2 1 4

Enter the value of each object: 4 2 2 1 10

Enter the maximum weight of the bag: 15

Added object 5 (10$, 4Kg) completely in the bag. Space left: 11.

Added object 2 (2$, 1Kg) completely in the bag. Space left: 10.

Added object 3 (2$, 2Kg) completely in the bag. Space left: 8.

Added object 4 (1$, 1Kg) completely in the bag. Space left: 7.

Added 58% (4$, 12Kg) of object 1 in the bag.

Filled the bag with objects worth 17.33$.

Time Complexity :

Best Case : O(n*logn)

Average Case : O(n*logn)

Worst Case : O(n*logn)

Space Complexity : O(n)

Stable : No

6. Implementation of Prim's Algorithm for finding Minimum Cost Spanning Tree.

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

void primMST(int V, int graph[][V])
{
    int parent[V];  // Array to store the parent of each vertex in the MST
    int key[V];     // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
    {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    // Always include first vertex in MST.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the set of vertices not yet
included in MST
        int u, min_key = INT_MAX;
        for (int v = 0; v < V; v++)
        {
            if (mstSet[v] == false && key[v] < min_key)
            {
                u = v;
                min_key = key[v];
            }
        }

        // Add the picked vertex to the MST Set
        mstSet[u] = true;
```

```
        // Update key value and parent index of the adjacent vertices of the
picked vertex.
        for (int v = 0; v < V; v++)
        {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the edges of the Minimum Spanning Tree
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
    {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}

int main()
{
    // Taking input graph as adjacency matrix
    printf("Enter the number of vertices in the graph: ");
    int V;
    scanf("%d", &V);

    printf("Enter the adjacency matrix of the graph:\n");
    int graph[V][V];
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            scanf("%d", &graph[i][j]);
        }
    }

    primMST(V, graph); // Call the Prim's Algorithm function

    return 0;
}
```

Output :

Enter the number of vertices in the graph: 5

Enter the adjacency matrix of the graph:

0 9 75 0 0 9 0 95 19 42 75 95 0 51 66 0 19 51 0 31 0 42 66 31 0

Edge Weight

0 - 1 9

3 - 2 51

1 - 3 19

3 - 4 31

Time Complexity :

Best Case : O(E*logV)

Average Case : O(E*logV)

Worst Case : $O(V^2)$

Space Complexity : O(V)

Stable : Yes

7. Implementation of Kruskal's Algorithm for finding Minimum Cost Spanning Tree.

```c
#include <stdio.h>
#include <stdlib.h>

// Comparator function to use in sorting
int comparator(const void *p1, const void *p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;

    return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n) {
    for (int i = 0; i < n; i++)
    {
        parent[i] = i;
        rank[i] = 0;
    }
}

// Function to find the parent of a node
int findParent(int parent[], int component) {
    if (parent[component] == component)
        return component;

    return parent[component] = findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n) {
    // Finding the parents
    u = findParent(parent, u);
    v = findParent(parent, v);

    if (rank[u] < rank[v])
    {
        parent[u] = v;
    }
    else if (rank[u] > rank[v])
    {
        parent[v] = u;
    }
```

```
    else
    {
        parent[v] = u;

        // Since the rank increases if
        // the ranks of two sets are same
        rank[u]++;
    }
}

// Function to find the MST
void kruskalAlgo(int n, int edge[n][3]) {
    // First we sort the edge array in ascending order
    // so that we can access minimum distances/cost
    qsort(edge, n, sizeof(edge[0]), comparator);

    int parent[n];
    int rank[n];

    // Function to initialize parent[] and rank[]
    makeSet(parent, rank, n);

    // To store the minimun cost
    int minCost = 0;

    printf("Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++)
    {
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];

        // If the parents are different that
        // means they are in different sets so
        // union them
        if (v1 != v2)
        {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0],
                   edge[i][1], wt);
        }
    }
    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}
```

```
// Driver code
int main()
{
    int edge[5][3] = {{0, 1, 10},
                      {0, 2, 6},
                      {0, 3, 5},
                      {1, 3, 15},
                      {2, 3, 4}};

    kruskalAlgo(5, edge);

    return 0;
}
```

Output :

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost Spanning Tree: 19

Time Complexity :

Best Case : O(E * logE)

Average Case : O(E * logE)

Worst Case : O(E * logE)

Space Complexity : O(E)

Stable : Yes

8. Implementation of single source shortest path using Dynamic Programming (Bellman Ford Algorithm).

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define INF INT_MAX

typedef struct Edge {
    int source, destination, weight;
} Edge;

void BellmanFord(int vertices, int edges, int source, Edge edge[], int
distance[]) {
    // Initialize distance from source to all vertices as infinity except
source
    for (int i = 0; i < vertices; ++i) {
        distance[i] = INF;
    }
    distance[source] = 0;

    // Relax all edges |vertices| - 1 times
    for (int i = 1; i < vertices; ++i) {
        for (int j = 0; j < edges; ++j) {
            int u = edge[j].source;
            int v = edge[j].destination;
            int weight = edge[j].weight;
            if (distance[u] != INF && distance[u] + weight < distance[v]) {
                distance[v] = distance[u] + weight;
            }
        }
    }

    // Check for negative weight cycles
    for (int i = 0; i < edges; ++i) {
        int u = edge[i].source;
        int v = edge[i].destination;
        int weight = edge[i].weight;
        if (distance[u] != INF && distance[u] + weight < distance[v]) {
            printf("Graph contains negative weight cycle\n");
            return;
        }
    }
```

```
    // Print the distances
    printf("Vertex    Distance from Source\n");
    for (int i = 0; i < vertices; ++i) {
        printf("%d\t  %d\n", i, distance[i]);
    }
}

int main() {
    int vertices, edges, source;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    printf("Enter the number of edges: ");
    scanf("%d", &edges);

    printf("Enter the source vertex: ");
    scanf("%d", &source);

    Edge edge[edges];
    for (int i = 0; i < edges; ++i) {
        printf("Enter edge %d's source destination and weights: ", i+1);
        scanf("%d %d %d", &edge[i].source, &edge[i].destination,
&edge[i].weight);
    }

    int distance[vertices];
    BellmanFord(vertices, edges, source, edge, distance);

    return 0;
}
```

Output :

Enter the number of vertices: 4

Enter the number of edges: 5

Enter the source vertex: 0

Enter edge 1's source destination and weights: 0 1 5

Enter edge 2's source destination and weights: 0 1 6

Enter edge 3's source destination and weights: 1 2 7

Enter edge 4's source destination and weights: 1 4 -5

Enter edge 5's source destination and weights: 1 3 6

Vertex	Distance from Source
0	0
1	5
2	12
3	11

Time Complexity :

Best Case : O(E)

Average Case : O(VE)

Worst Case : O(VE)

Space Complexity : O(V)

Stable : Yes

9. Implementation of Longest Common Subsequence Algorithm.

```c
#include <stdio.h>
#include <string.h>

#define MAX_LENGTH 100

// Utility function to get max of 2 integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns length of LCS for X[0..m-1], Y[0..n-1] and stores the LCS in lcs[]
int lcs(char *X, char *Y, int m, int n, char lcs[])
{
    int L[m+1][n+1];
    int i, j;

    // Build L[m+1][n+1] in bottom-up fashion
    for (i = 0; i <= m; i++) {
        for (j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i-1] == Y[j-1]) {
                L[i][j] = L[i-1][j-1] + 1;
                lcs[L[i][j]-1] = X[i-1];
            }
            else
                L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
    }

    // L[m][n] contains length of LCS for X[0..m-1] and Y[0..n-1]
    return L[m][n];
}

// Driver code
int main() {
    char S1[MAX_LENGTH], S2[MAX_LENGTH], lcs_seq[MAX_LENGTH];
    printf("Enter String 1: ");
    fgets(S1, MAX_LENGTH, stdin);
    printf("Enter String 2: ");
    fgets(S2, MAX_LENGTH, stdin);

    int m = strlen(S1)-1; // Exclude newline character
    int n = strlen(S2)-1; // Exclude newline character
```

```
    int lcs_length = lcs(S1, S2, m, n, lcs_seq);
    lcs_seq[lcs_length] = '\0'; // Append null character to make it a string

    printf("Length of LCS is %d\n", lcs_length);
    printf("LCS is %s\n", lcs_seq);

    return 0;
}
```

Output :

Enter String 1: abaaba

Enter String 2: babbab

Length of LCS is 4

LCS is baba

Time Complexity :

Best Case : O(nm)

Average Case : O(nm)

Worst Case : O(nm)

Space Complexity : O(nm)

Stable : Yes

10. Implementation of 8 Queen Problems.

```c
#include <stdio.h>
#include <stdbool.h>
#define N 8
int count = 1;

void printSolution(int board[N][N]) {
    printf("Solution %d: \n", count);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
    printf("\n");
    count = count + 1;
}

bool isSafe(int board[N][N], int row, int col) {
    int i, j;
    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;
    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;
    return true;
}

bool solveNQUtil(int board[N][N], int col) {
    if (col == N) {
        printSolution(board);
        return true;
    }
    bool res = false;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            res = solveNQUtil(board, col + 1) || res;
```

```
            board[i][col] = 0; // backtrack
        }
    }
    return res;
}

int main() {
    int board[N][N] = { { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 } };
    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
    }
    return 0;
}
```


Output :

Solution 92:

0 0 1 0 0 0 0 0

0 0 0 0 0 1 0 0

0 0 0 1 0 0 0 0

0 1 0 0 0 0 0 0

0 0 0 0 0 0 0 1

0 0 0 0 1 0 0 0

0 0 0 0 0 0 1 0

1 0 0 0 0 0 0 0

Time Complexity :

Best Case : O(1)

Average Case : O(n!)

Worst Case : O(n!)

Space Complexity : $O(n^2)$

Stable : Yes

11. Implementation of KMP algorithm.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Function to implement the KMP algorithm
void KMP(const char *text, const char *pattern, int m, int n)
{
    // base case 1: pattern is NULL or empty
    if (*pattern == '\0' || n == 0)
    {
        printf("The pattern occurs with shift 0");
    }

    // base case 2: text is NULL, or text's length is less than that of
pattern's
    if (*text == '\0' || n > m)
    {
        printf("Pattern not found");
    }

    // next[i] stores the index of the next best partial match
    int next[n + 1];

    for (int i = 0; i < n + 1; i++)
    {
        next[i] = 0;
    }

    for (int i = 1; i < n; i++)
    {
        int j = next[i];

        while (j > 0 && pattern[j] != pattern[i])
        {
            j = next[j];
        }

        if (j > 0 || pattern[j] == pattern[i])
        {
            next[i + 1] = j + 1;
        }
    }
```

```c
    for (int i = 0, j = 0; i < m; i++)
    {
        if (*(text + i) == *(pattern + j))
        {
            if (++j == n)
            {
                printf("The pattern occurs at index %d\n", i - j + 1);
            }
        }
        else if (j > 0)
        {
            j = next[j];
            i--; // since `i` will be incremented in the next iteration
        }
    }
}

// Program to implement the KMP algorithm in C
int main(void)
{
    char text[100], pattern[100];
    printf("Text: ");
    gets(text);
    printf("Pattern: ");
    gets(pattern);

    int n = strlen(text);
    int m = strlen(pattern);

    KMP(text, pattern, n, m);

    return 0;
}
```

Output :

Text: ABCABAABCABAC

Pattern: CAB

The pattern occurs at index 2

The pattern occurs at index 8

Time Complexity :

Best Case : O(n+m)

Average Case : O(n+m)

Worst Case : O(n+m)

Space Complexity : O(m)

Stable : Yes