Practical no: 1- Experiment Based on common mathematical functions (Selection and Insertion Sort).

A) Selection sort:

Code:

```
include <stdio.h>

void selectionSort(int arr[], int n) { int i, j,
    minIndex, temp;
    for (i = 0; i < n - 1; i++) { minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) minIndex
                = j;}

        temp    =    arr[minIndex];
        arr[minIndex]    =    arr[i];
        arr[i] = temp;
    }}
void printArray(int arr[], int n) { int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    selectionSort(arr, n);
    printArray(arr, n); return
    0;
}
```

Output:

```
11 12 22 25 64


=== Code Execution Successful ===
```

B) Insertion sort:

Code:

```
#include <stdio.h>
void insertionSort(int arr[], int n) { int i,
 key, j;
 for (i = 1; i < n; i++) { key
 = arr[i];
 j = i - 1;
 while (j >= 0 && arr[j] > key) { arr[j +
 1] = arr[j];
 j = j - 1;
 }
 arr[j + 1] = key;
 }
}
void printArray(int arr[], int n) { int i;
 for (i = 0; i < n; i++)
 printf("%d ", arr[i]);
 printf("\n");
```

```
}
int main() {
 int arr[] = {64, 25, 12, 22, 11};
 int n = sizeof(arr) / sizeof(arr[0]);
 printf("Insertion Sort\n");
 printf("Original array:\n");
 printArray(arr, n); insertionSort(arr,
 n); printf("Sorted array:\n");
 printArray(arr, n);
 return 0;
}
```

Output:

Output

```
Insertion Sort
Original array:
64 25 12 22 11
Sorted array:
11 12 22 25 64


=== Code Execution Successful ===
```

Practical no: 2- Experiment Based on divide & conquers approach (Merge Sort, Quick Sort, Binary search).

A) Merge sort:

Code:

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}
```

```
void mergeSort(int arr[], int l, int r) {
if (l < r) {
int m = l + (r - l) / 2;
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
merge(arr, l, m, r);
}
}
void printArray(int arr[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");
}
int main() {
int arr[] = {64, 25, 12, 22, 11};
int n = sizeof(arr) / sizeof(arr[0]);
printf("Original array:\n");
printArray(arr, n);
mergeSort(arr, 0, n - 1);
printf("Sorted array:\n");
printArray(arr, n);
return 0;
}
```

Output:

```
Original array:
64 25 12 22 11
Sorted array:
11 12 22 25 64


=== Code Execution Successful ===
```

B) Quick sort:

Code:

```
#include <stdio.h>
void swap(int* a, int* b) {
int t = *a;
```

```
*a = *b;
*b = t;
}
int partition(int arr[], int low, int high) {
int pivot = arr[high]; // Pivot element
int i = (low - 1);
for (int j = low; j < high; j++) {
if (arr[j] < pivot) {
i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]);
return (i + 1);
}
void quickSort(int arr[], int low, int high) {
if (low < high) {
int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}
void printArray(int arr[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");
}
int main() {
int arr[] = {64, 25, 12, 22, 11};
int n = sizeof(arr) / sizeof(arr[0]);
printf("Quick Sort:\n");
printf("Original array:\n");
printArray(arr, n);
quickSort(arr, 0, n - 1);
printf("Sorted array:\n");
```

```
printArray(arr, n);

return 0;

}
```

Output:

```
Quick Sort:
Original array:
64 25 12 22 11
Sorted array:
11 12 22 25 64


=== Code Execution Successful ===
```

C) Binary search:

Code:

```
#include <stdio.h>
int binarySearch(int arr[], int n, int key) {
int low = 0, high = n - 1, mid;
while (low <= high) {
mid = (low + high) / 2;
if (arr[mid] == key)
return mid;
else if (arr[mid] < key)
low = mid + 1;
else
high = mid - 1;
}
return -1;
}
int main() {
printf("Binary Search:\n");
int arr[] = {11, 12, 22, 25, 64};
int n = sizeof(arr) / sizeof(arr[0]);
int key = 64;
int result = binarySearch(arr, n, key);
if (result != -1)
printf("Element found at index %d\n", result);
else
printf("Element not found\n");
return 0;
}
```

Output:

```
Binary Search:
Element found at index 4


=== Code Execution Successful ===
```

Practical no: 3- Experiment Based on greedy approach (Single source shortest path- Dijkstra Fractional Knapsack problem, Minimum cost spanning trees- Kruskal and Prim's algorithm)

A) Dijkstra's Algorithm:

Code:

```
#include <stdio.h>
#include <limits.h>
#define MAXN 100
#define INF INT_MAX/2
int n; // number of vertices
int graph[MAXN][MAXN];
int distArr[MAXN];
int used[MAXN];
void dijkstra(int src) {
 for (int i = 0; i < n; i++) {
 distArr[i] = INF;
 used[i] = 0;
 }
 distArr[src] = 0;
 for (int i = 0; i < n; i++) {
 int u = -1;
 for (int j = 0; j < n; j++) {
 if (!used[j] && (u == -1 || distArr[j] < distArr[u])) {
 u = j;
 }
 }
 used[u] = 1;
 for (int v = 0; v < n; v++) {
```

```
if (distArr[v] > distArr[u] + graph[u][v]) {
distArr[v] = distArr[u] + graph[u][v];
}
}
}
}
int main() {
// Example
n = 4;
// initialize graph adjacency matrix
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) graph[i][j] = INF;
// edges
graph[0][1] = 1;
graph[0][2] = 4;
graph[1][2] = 2;
graph[1][3] = 5;
graph[2][3] = 1;
dijkstra(0);
printf("Distances from node 0:\n");
for (int i = 0; i < n; i++) {
printf("To %d = %d\n", i, distArr[i]);
}
return 0;
}
```

Output:

```
Distances from node 0:
To 0 = 0
To 1 = 1
To 2 = 3
To 3 = 4


=== Code Execution Successful ===
```

B) Kruskal’s Minimum Spanning Tree:

Code:

```
#include   <stdio.h>
#include  <stdlib.h>
#define MAXV 100
#define MAXE 1000
typedef struct {
 int u, v, w;
} Edge;
Edge edges[MAXE];
int parent[MAXV];
int rankArr[MAXV];
int compareEdges(const void *a, const void *b) {
Edge *ea = (Edge*)a;
 Edge *eb = (Edge*)b;
 return ea->w - eb->w;
}
void makeSet(int n) {
 for (int i = 0; i < n; i++) {
 parent[i] = i;
 rankArr[i] = 0;
 }
}
```

```
int findSet(int x) {
 if (parent[x] != x)
parent[x] = findSet(parent[x]);
 return parent[x];
}
void unionSet(int a, int b) {
 a = findSet(a);
 b = findSet(b);
 if (a != b) {
 if (rankArr[a] < rankArr[b]) {
 parent[a] = b;
 } else if (rankArr[b] < rankArr[a]) {
 parent[b] = a;
 } else {
 parent[b] = a;
 rankArr[a]++;
 }
 }
}
void kruskal(Edge edges[], int V, int E) {
 qsort(edges, E, sizeof(Edge), compareEdges);
 makeSet(V);
 printf("Edges in the MST:\n");
 for (int i = 0; i < E; i++) {
 int u = edges[i].u;
 int v = edges[i].v;
 if (findSet(u) != findSet(v)) {
 printf("%d -- %d == %d\n", u, v, edges[i].w);
 unionSet(u, v);
 }
 }
```

```
}
int main() {
 int V = 4; // number of vertices
 int E = 5; // number of edges
 // define edges (u, v, weight)
 edges[0] = (Edge){0, 1, 1};
 edges[1] = (Edge){0, 2, 3};
 edges[2] = (Edge){1, 2, 1};
 edges[3] = (Edge){1, 3, 4};
 edges[4] = (Edge){2, 3, 1};
 kruskal(edges, V, E);
 return 0;
}
```

Output:

```
Edges in the MST:
0 -- 1 == 1
1 -- 2 == 1
2 -- 3 == 1


=== Code Execution Successful ===
```

Practical no: 4- Experiment using dynamic programming approach (All pair shortest path- Floyd Warshall, 0/1 knapsack)

A) Floyd Warshall Algorithm:

Code:

```
#include <stdio.h>

#define INF 99999

#define V 4

void floydWarshall(int graph[V][V]) {

 int dist[V][V], i, j, k;

 for (i = 0; i < V; i++)

 for (j = 0; j < V; j++)

 dist[i][j] = graph[i][j];

 for (k = 0; k < V; k++)

 for (i = 0; i < V; i++)

 for (j = 0; j < V; j++)

if (dist[i][k] + dist[k][j] < dist[i][j])

 dist[i][j] = dist[i][k] + dist[k][j];

 printf("Shortest distances:\n");

 for (i = 0; i < V; i++) {

 for (j = 0; j < V; j++) {

 if (dist[i][j] == INF)

 printf("INF ");

 else

 printf("%3d ", dist[i][j]);

 }

 printf("\n");

 }
```

```
}
int main() {
 int graph[V][V] = {
 {0, 3, INF, 5},
 {2, 0, INF, 4},
 {INF, 1, 0, INF},
 {INF, INF, 2, 0}
 };
 floydWarshall(graph);
 return 0;
}
```

Output:

```
Shortest distances:
   0    3    7    5
   2    0    6    4
   3    1    0    5
   5    3    2    0


=== Code Execution Successful ===
```

B) 0/1 Knapsack:

Code:

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int knapsack(int W, int wt[], int val[], int n) {
int dp[n+1][W+1];
 for (int i = 0; i <= n; i++)
 for (int w = 0; w <= W; w++) {
 if (i == 0 || w == 0)
 dp[i][w] = 0;
 else if (wt[i-1] <= w)
```

```
 dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w]);
 else
 dp[i][w] = dp[i-1][w];
 }
 return dp[n][W];
}
int main() {
 int val[] = {60, 100, 120};
 int wt[] = {10, 20, 30};
 int W = 50;
 int n = sizeof(val)/sizeof(val[0]);
 printf("Max value = %d\n", knapsack(W, wt, val, n));
 return 0;
}
```

Output:

```
Max value = 220


=== Code Execution Successful ===
```

Practical no: 5- Longest Common Subsequence (Used in TSP & DP).

Code:

```
#include <stdio.h>
#include <string.h>
int max(int a, int b) { return (a > b) ? a : b; }
int lcs(char *X, char *Y, int m, int n) {
 int L[m+1][n+1];
 for (int i = 0; i <= m; i++)
 for (int 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] = 1 + L[i-1][j-1];
 else
L[i][j] = max(L[i-1][j], L[i][j-1]);
 return L[m][n];
}
int main() {
 char X[] = "AGGTAB";
 char Y[] = "GXTXAYB";
 printf("Length of LCS is %d\n", lcs(X, Y, strlen(X), strlen(Y)));
 return 0;
}
```

Output:

```
Length of LCS is 4


=== Code Execution Successful ===
```

Practical no: 6- Graph Algorithms (BFS and DFS).

A) BFS:

Code:

```
#include <stdio.h>
int visited[100], queue[100], front = -1, rear = -1;
void bfs(int adj[10][10], int n, int start) {
visited[start] = 1;
 queue[++rear] = start;
 while (front != rear) {
 int current = queue[++front];
 printf("%d ", current);
 for (int i = 0; i < n; i++) {
 if (adj[current][i] && !visited[i]) {
 queue[++rear] = i;
visited[i] = 1;
 }
 }
 }
}
int main() {
 int n = 4;
 int adj[10][10] = {
 {0, 1, 1, 0},
 {1, 0, 1, 1},
 {1, 1, 0, 0},
 {0, 1, 0, 0}
 };
```

```
 bfs(adj, n, 0);
 return 0;
}
```

Output:

```
0 1 2 3

=== Code Execution Successful ===
```

B) DFS:

Code:

```
#include <stdio.h>
int visited[10];
void dfs(int adj[10][10], int n, int start) {
 visited[start] = 1;
 printf("%d ", start);
 for (int i = 0; i < n; i++)
 if (adj[start][i] && !visited[i])
 dfs(adj, n, i);
}
int main() {
 int n = 4;
 int adj[10][10] = {
 {0, 1, 1, 0},
 {1, 0, 1, 1},
 {1, 1, 0, 0},
 {0, 1, 0, 0}
 };
 dfs(adj, n, 0);
 return 0;
```

}

Output:

```
0 1 2 3

=== Code Execution Successful ===
```

Practical no: 7- Experiment using Backtracking strategy.

Code:

```
#include <stdio.h>
#define N 4
int board[N][N];
int isSafe(int row, int col) {
    for (int i = 0; i < col; i++)
        if (board[row][i]) return 0;
    for (int i=row, j=col; i>=0 && j>=0; i--, j--)
        if (board[i][j]) return 0;
    for (int i=row, j=col; i<N && j>=0; i++, j--)
        if (board[i][j]) return 0;
    return 1;
}
int solve(int col) {
    if (col >= N) return 1;
    for (int i = 0; i < N; i++) {
        if (isSafe(i, col)) {
            board[i][col] = 1;
            if (solve(col + 1)) return 1;
            board[i][col] = 0;
        }
    }
    return 0;
}

void printBoard() {
```

```
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%d ", board[i][j]);
        printf("\n");
    }
}
int main() {
    if (solve(0))
        printBoard();
    else
        printf("Solution does not exist");
    return 0;
}
```

Output:

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0


=== Code Execution Successful ===
```

Practical no: 8- Experiment based on String Matching.

Code:

```
#include <stdio.h>

#include <string.h>

void search(char *txt, char *pat) {

 int n = strlen(txt);

 int m = strlen(pat);

 for (int i = 0; i <= n - m; i++) {

 int j;

 for (j = 0; j < m; j++)

 if (txt[i + j] != pat[j])

 break;

 if (j == m)

 printf("Pattern found at index %d\n", i);

 }

}

int main() {

 char txt[] = "AABAACAADAABAABA";

 char pat[] = "AABA";

 search(txt, pat);

 return 0;

}
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12


=== Code Execution Successful ===
```

Practical no: 9- Implementation of Job Sequencing with Deadlines.

Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
char id;
 int deadline, profit;
} Job;
int cmp(Job a, Job b) {
 return b.profit - a.profit;
}
void jobSequencing(Job jobs[], int n) {
 char result[n];
 int slot[n];
 for (int i = 0; i < n; i++) slot[i] = 0;
 for (int i = 0; i < n; i++) {
 for (int j = jobs[i].deadline - 1; j >= 0; j--) {
 if (!slot[j]) {
 result[j] = jobs[i].id;
 slot[j] = 1;
 break;
 }
 }
 }
 printf("Scheduled Jobs: ");
 for (int i = 0; i < n; i++)
 if (slot[i]) printf("%c ", result[i]);
```

```
}
int main() {
 Job jobs[] = {{'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27}, {'d', 1, 25}, {'e', 3, 15}};
 int n = sizeof(jobs)/sizeof(jobs[0]);
 jobSequencing(jobs, n);
 return 0;
}
```

Output:

```
Scheduled Jobs: b a e

=== Code Execution Successful ===
```

Practical no: 10- Implementation of Bellman Ford Algorithm using Dynamic Programming.

Code:

```
#include <stdio.h>
#define V 5
#define E 8
#define INF 99999
typedef struct {
 int u, v, w;
} Edge;
Edge edges[E] = {
 {0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2},
 {1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3}
};
void bellmanFord(int src) {
 int dist[V];
 for (int i = 0; i < V; i++) dist[i] = INF;
 dist[src] = 0;
 for (int i = 1; i <= V - 1; i++)
 for (int j = 0; j < E; j++) {
 int u = edges[j].u;
 int v = edges[j].v;
 int w = edges[j].w;
 if (dist[u] != INF && dist[u] + w < dist[v])
 dist[v] = dist[u] + w;
 }
 printf("Vertex Distance from Source:\n");
```

```
 for (int i = 0; i < V; i++)
 printf("%d \t %d\n", i, dist[i]);
}
int main() {
 bellmanFord(0);
 return 0;
}
```

Output:

```
Vertex Distance from Source:
0     0
1     -1
2     2
3     -2
4     1


=== Code Execution Successful ===
```