# LAB Experiments

1. **Implement the activity selection problem to get a clear understanding of greedy approach.**

Code:

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

// Structure for storing activity details
typedef struct {
    int start;
    int finish;
} Activity;

// Comparator function to sort activities based on finish time
int compare(const void *a, const void *b) {
    return ((Activity *)a)->finish - ((Activity *)b)->finish;
}

void activitySelection(Activity arr[], int n) {
    qsort(arr, n, sizeof(Activity), compare);

    printf("Selected activities:\n");
    int i = 0;  // Select the first activity
    printf("Activity %d: [%d, %d]\n", i + 1, arr[i].start, arr[i].finish);

    for (int j = 1; j < n; j++) {
        if (arr[j].start >= arr[i].finish) {
            printf("Activity %d: [%d, %d]\n", j + 1, arr[j].start, arr[j].finish);
            i = j;
        }
```

```
    }
}

int main() {
    Activity activities[] = {{1, 3}, {2, 5}, {4, 7}, {6, 8}, {5, 9}};
    int n = sizeof(activities) / sizeof(activities[0]);

    activitySelection(activities, n);

    return 0;
}
```

**2. Get a detailed insight of dynamic programming approach by the implementation of Matrix Chain Multiplication problem and see the impact of parenthesis positioning on time requirements for matrix multiplication.**

Code:

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

void matrixChainOrder(int p[], int n) {
    int m[n][n];
    int s[n][n];

    // Fill m[][] with 0 (diagonal elements)
    for (int i = 1; i < n; i++) {
        m[i][i] = 0;
    }

    // Calculate minimum multiplication costs for different chain lengths
    for (int l = 2; l < n; l++) {
        for (int i = 1; i < n - l + 1; i++) {
            int j = i + l - 1;
```

```
            m[i][j] = INT_MAX;

            // Find the minimum cost by trying different splits
            for (int k = i; k < j; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }

    printf("Minimum number of multiplications is %d\n", m[1][n - 1]);
}

int main() {
    int arr[] = {1, 2, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    matrixChainOrder(arr, n);
    return 0;
}
```

**3. Compare the performance of Dijkstra and Bellman ford algorithm for the single source shortest path problem.**

Code:

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

#define V 9

int minDistance(int dist[], bool sptSet[]) {
```

```
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v], min_index = v;
        }
    }
    return min_index;
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
```

```
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d \t %d\n", i, dist[i]);
    }
}

int main() {
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 0, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 0 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 0, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 0, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}
```

**4. Through 0/1 Knapsack problem, analyze the greedy and dynamic programming approach for the same dataset.**

Code:

```
#include <stdio.h>

int knapsack(int W, int wt[], int val[], int n) {
    int K[n + 1][W + 1];

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

```
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                K[i][w] = 0;
            } else if (wt[i - 1] <= w) {
                K[i][w] = (val[i - 1] + K[i - 1][w - wt[i - 1]] > K[i - 1][w]) ?
                        val[i - 1] + K[i - 1][w - wt[i - 1]] : K[i - 1][w];
            } else {
                K[i][w] = K[i - 1][w];
            }
        }
    }
    return K[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("Maximum value in Knapsack = %d\n", knapsack(W, wt, val, n));
    return 0;
}
```

**5. Implement the sum of subset and N Queen problem.**

Code:

```
#include <stdio.h>

void printSubset(int subset[], int subsetSize) {
    for (int i = 0; i < subsetSize; i++) {
        printf("%d ", subset[i]);
```

```
    }
    printf("\n");
}

void sumOfSubsets(int arr[], int n, int sum, int index, int subset[], int subsetSize) {
    if (sum == 0) {
        printSubset(subset, subsetSize);
        return;
    }

    for (int i = index; i < n; i++) {
        if (arr[i] <= sum) {
            subset[subsetSize] = arr[i];
            sumOfSubsets(arr, n, sum - arr[i], i + 1, subset, subsetSize + 1);
        }
    }
}

int main() {
    int arr[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(arr) / sizeof(arr[0]);
    int subset[n];

    sumOfSubsets(arr, n, sum, 0, subset, 0);
    return 0;
}
```

**6. Compare the Backtracking and Branch & Bound Approach by the implementation of 0/1 Knapsack problem. Also compare the performance with dynamic programming approach.**

Code:

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

#define N 4

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

bool isSafe(int board[N][N], int row, int col) {
    for (int i = 0; i < col; i++) {
        if (board[row][i]) {
            return false;
        }
    }
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    for (int 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) {
        return true;
    }

    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1)) {
                return true;
            }
            board[i][col] = 0;  // Backtrack
        }
    }
    return false;
}

bool solveNQ() {
    int board[N][N] = {0};
    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist\n");
        return false;
    }
    printSolution(board);
    return true;
}

int main() {
    solveNQ();
```

```
    return 0;
}

int main() {
    // Run individual problems here, for example:
    printf("Activity Selection Problem:\n");
    Activity activities[] = {{1, 3}, {2, 5}, {4, 7}, {6, 8}, {5, 9}};
    int n = sizeof(activities) / sizeof(activities[0]);
    activitySelection(activities, n);

    printf("\nMatrix Chain Multiplication:\n");
    int arr[] = {1, 2, 3, 4};
    matrixChainOrder(arr, sizeof(arr) / sizeof(arr[0]));

    printf("\nDijkstra Algorithm (Shortest Path):\n");
    int graph[V][V] = { /* Graph definition here */ };
    dijkstra(graph, 0);

    printf("\n0/1 Knapsack Problem (Dynamic Programming):\n");
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    printf("Maximum value in Knapsack = %d\n", knapsack(W, wt, val, sizeof(val) / sizeof(val[0])));

    printf("\nSum of Subsets Problem:\n");
    int arr2[] = {3, 34, 4, 12, 5, 2};
    sumOfSubsets(arr2, sizeof(arr2) / sizeof(arr2[0]), 9, 0, (int[]){}, 0);

    printf("\nN-Queens Problem:\n");
    solveNQ();
```

```
    return 0;
}
```