LAB Experiments

- Implement the activity selection problem to get a clear understanding of greedy approach.
- 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.
- Compare the performance of Dijkstra and Bellman ford algorithm for the single source shortest path problem.
- Through 0/1 Knapsack problem, analyze the greedy and dynamic programming approach for the same dataset.
- Implement the sum of subset and N Queen problem.
- Compare the Backtracking and Branch & Bound Approach by the implementation of 0/1 Knapsack problem. Also compare the performance with dynamic programming approach.

ANSWERS

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

void activitySelection(int start[], int finish[], int n) {
    int i, j;
    printf("Selected activities: \n");
    i = 0;
    printf("%d ", i);
    for (j = 1; j < n; j++) {
        if (start[j] >= finish[i]) {
            printf("%d ", j);
            i = j;
        }
    }
}

int main() {
    int n, i;

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

    int start[n], finish[n];

    printf("Enter start times:\n");
    for (i = 0; i < n; i++) {
        printf("Start time of activity %d: ", i+1);
        scanf("%d", &start[i]);
    }

    printf("Enter finish times:\n");
    for (i = 0; i < n; i++) {
        printf("Finish time of activity %d: ", i+1);
        scanf("%d", &finish[i]);
    }

    activitySelection(start, finish, n);

    return 0;
```

```
}
```

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

int mcm(int p[], int n)
{
    int m[n][n];
    int i, j, k, L, q;

    for (i = 1; i < n; i++)
        m[i][i] = 0;

    printf("Initial m matrix:\n");
    for (i = 1; i < n; i++) {
        for (j = 1; j < n; j++) {
            printf("%d ", m[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    for (L = 2; L < n; L++) {
        printf("Processing chain length %d:\n", L);
        for (i = 1; i < n - L + 1; i++) {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++) {
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                printf("  Considering split at k = %d: cost = m[%d][%d] + m[%d][%d] + %d * %d * %d = 
%d + %d + %d = %d\n",
                    k, i, k, k + 1, j, p[i - 1], p[k], p[j], m[i][k], m[k + 1][j], p[i - 1] * p[k] * p[j], q);
                if (q < m[i][j]) {
                    m[i][j] = q;
                    printf("  Updating m[%d][%d] = %d\n", i, j, m[i][j]);
                }
            }
        }
        printf("\n");
        printf("M matrix after processing chain length %d:\n", L);
        for (i = 1; i < n; i++) {
            for (j = 1; j < n; j++) {
                printf("%d ", m[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    }

    return m[1][n - 1];
}

int main()
{
    int n;
    printf("Enter the number of matrices: ");
    scanf("%d", &n);
```

```
    int arr[n + 1];
    printf("Enter the dimensions of the matrices (p0 p1 ... pn): ");
    for (int i = 0; i <= n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Minimum number of multiplications is %d\n", mcm(arr, n + 1));

    return 0;
}
```

3>
```
#include <stdio.h>
#include <limits.h>

#define V 9

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

void dijkstra(int graph[V][V], int src) {
    int dist[V], sptSet[V], prev[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
        prev[i] = -1;
    }
    dist[src] = 0;

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

        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];
                prev[v] = u;
            }
        }
    }

    printf("Dijkstra Algorithm - Shortest distances and paths from source vertex %d:\n", src);
    for (int i = 0; i < V; i++) {
        printf("%d \t %d \t Path: ", i, dist[i]);
        int path[V], pathIndex = 0;
        for (int j = i; j != -1; j = prev[j]) {
            path[pathIndex++] = j;
        }
        for (int j = pathIndex - 1; j >= 0; j--) {
            printf("%d ", path[j]);
        }
        printf("\n");
    }
```

```
}

void bellmanFord(int graph[V][V], int src) {
    int dist[V], prev[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        prev[i] = -1;
    }
    dist[src] = 0;

    for (int i = 1; i < V; i++) {
        for (int u = 0; u < V; u++) {
            for (int v = 0; v < V; v++) {
                if (graph[u][v] != 0 && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                    dist[v] = dist[u] + graph[u][v];
                    prev[v] = u;
                }
            }
        }
    }

    printf("\nBellman-Ford Algorithm - Shortest distances and paths from source vertex %d:\n",
src);
    for (int i = 0; i < V; i++) {
        printf("%d \t %d \t Path: ", i, dist[i]);
        int path[V], pathIndex = 0;
        for (int j = i; j != -1; j = prev[j]) {
            path[pathIndex++] = j;
        }
        for (int j = pathIndex - 1; j >= 0; j--) {
            printf("%d ", path[j]);
        }
        printf("\n");
    }
}

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}
    };

    int source;
    printf("Enter the source vertex (0 to 8): ");
    scanf("%d", &source);

    printf("\nDijkstra's Algorithm Results:\n");
    dijkstra(graph, source);

    printf("\nBellman-Ford Algorithm Results:\n");
    bellmanFord(graph, source);

    return 0;
```

```
}

4>#include <stdio.h>

#define MAX_ITEMS 100

// Greedy Approach: 0/1 Knapsack
float greedyKnapsack(int values[], int weights[], int n, int W) {
    float ratio[n];
    int i, j;
    for (i = 0; i < n; i++) {
        ratio[i] = (float)values[i] / weights[i];
    }

    // Sort items by value/weight ratio in descending order
    for (i = 0; i < n-1; i++) {
        for (j = i+1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
                float tempRatio = ratio[i];
                ratio[i] = ratio[j];
                ratio[j] = tempRatio;

                int tempValue = values[i];
                values[i] = values[j];
                values[j] = tempValue;

                int tempWeight = weights[i];
                weights[i] = weights[j];
                weights[j] = tempWeight;
            }
        }
    }

    int currentWeight = 0;
    float totalValue = 0.0;

    // Pick items based on greedy approach
    for (i = 0; i < n; i++) {
        if (currentWeight + weights[i] <= W) {
            currentWeight += weights[i];
            totalValue += values[i];
        }
    }

    return totalValue;
}

// Dynamic Programming Approach: 0/1 Knapsack (Optimal Solution)
int knapsack(int values[], int weights[], int n, int W) {
    int dp[n+1][W+1];
    int i, w;

    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (weights[i-1] <= w) {
                dp[i][w] = (values[i-1] + dp[i-1][w - weights[i-1]] > dp[i-1][w]) ?
                           values[i-1] + dp[i-1][w - weights[i-1]] : dp[i-1][w];
            } else {
```

```
                dp[i][w] = dp[i-1][w];
            }
        }
    }

    return dp[n][W];
}

int main() {
    int n, W, i;

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

    int values[n], weights[n];

    printf("Enter the values and weights of the items:\n");
    for (i = 0; i < n; i++) {
        printf("Item %d - Value: ", i + 1);
        scanf("%d", &values[i]);
        printf("Item %d - Weight: ", i + 1);
        scanf("%d", &weights[i]);
    }

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);

    printf("\nGreedy Approach: Maximum value = %.2f\n", greedyKnapsack(values, weights, n, W));
    printf("Dynamic Programming Approach: Maximum value = %d\n", knapsack(values, weights,
n, W));

    return 0;
}
```

5>// SUM OF SUBSET

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

bool subsetSum(int arr[], int n, int sum, int subset[], int* subsetSize) {
    if (sum == 0) {
        return true;
    }
    if (n == 0 || sum < 0) {
        return false;
    }

    if (subsetSum(arr, n - 1, sum, subset, subsetSize)) {
        return true;
    }

    if (arr[n - 1] <= sum) {
        subset[(*subsetSize)++] = arr[n - 1];
        if (subsetSum(arr, n - 1, sum - arr[n - 1], subset, subsetSize)) {
            return true;
        }
        (*subsetSize)--;
    }

    return false;
```

```
}

int main() {
    int n, sum;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the sum to check: ");
    scanf("%d", &sum);

    int subset[n];
    int subsetSize = 0;

    if (subsetSum(arr, n, sum, subset, &subsetSize)) {
        printf("Subset with the given sum: ");
        for (int i = 0; i < subsetSize; i++) {
            printf("%d ", subset[i]);
        }
        printf("\n");
    } else {
        printf("No subset with the given sum exists.\n");
    }

    return 0;
}


// N QUEEN PROBLEM

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

#define MAX_N 30

int board[MAX_N];
int leftDiagonal[2 * MAX_N - 1], rightDiagonal[2 * MAX_N - 1], column[MAX_N];

void printSolution(int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i] == j)
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}

bool isSafe(int row, int col, int N) {
    return !column[col] && !leftDiagonal[row - col + N - 1] && !rightDiagonal[row + col];
}

void placeQueen(int row, int col, int N) {
    column[col] = 1;
    leftDiagonal[row - col + N - 1] = 1;
```

```
    rightDiagonal[row + col] = 1;
    board[row] = col;
}

void removeQueen(int row, int col, int N) {
    column[col] = 0;
    leftDiagonal[row - col + N - 1] = 0;
    rightDiagonal[row + col] = 0;
}

bool solveNQ(int row, int N) {
    if (row == N) {
        printSolution(N);
        return true;
    }

    for (int col = 0; col < N; col++) {
        if (isSafe(row, col, N)) {
            placeQueen(row, col, N);
            if (solveNQ(row + 1, N))
                return true;
            removeQueen(row, col, N);
        }
    }

    return false;
}

int main() {
    int N;
    printf("Enter the value of N (size of the board): ");
    scanf("%d", &N);

    if (N > MAX_N) {
        printf("N is too large! Maximum supported N is %d.\n", MAX_N);
        return 0;
    }

    for (int i = 0; i < N; i++) {
        board[i] = -1;
    }

    if (!solveNQ(0, N)) {
        printf("Solution does not exist.\n");
    }

    return 0;
}
```

6>
```
#include <stdio.h>
#include <stdlib.h>

int n, W;
int *weights, *values;

int maxValBacktracking = 0;

void knapsackBacktracking(int i, int currentWeight, int currentValue) {
    if (i == n) {
        if (currentWeight <= W && currentValue > maxValBacktracking) {
```

```
            maxValBacktracking = currentValue;
        }
        return;
    }

    knapsackBacktracking(i + 1, currentWeight, currentValue);
    if (currentWeight + weights[i] <= W) {
        knapsackBacktracking(i + 1, currentWeight + weights[i], currentValue + values[i]);
    }
}

// Branch and Bound approach

typedef struct {
    int level, profit, weight;
} Node;

int bound(Node u) {
    if (u.weight >= W)
        return 0;
    int result = u.profit;
    int j = u.level + 1;
    int totalWeight = u.weight;
    while (j < n && totalWeight + weights[j] <= W) {
        totalWeight += weights[j];
        result += values[j];
        j++;
    }
    if (j < n)
        result += (W - totalWeight) * values[j] / weights[j];
    return result;
}

int knapsackBranchBound() {
    Node u, v;
    u.level = -1;
    u.profit = u.weight = 0;
    int maxProfit = 0;
    Node *queue = (Node*) malloc(n * sizeof(Node));
    int front = 0, rear = 0;

    queue[rear++] = u;

    while (front < rear) {
        u = queue[front++];

        if (u.level == n - 1)
            continue;

        v.level = u.level + 1;
        v.weight = u.weight + weights[v.level];
        v.profit = u.profit + values[v.level];

        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;

        if (bound(v) > maxProfit)
            queue[rear++] = v;

        v.weight = u.weight;
```

```c
        v.profit = u.profit;

        if (bound(v) > maxProfit)
            queue[rear++] = v;
    }

    free(queue);
    return maxProfit;
}

int main() {
    printf("Enter number of items: ");
    scanf("%d", &n);
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);

    weights = (int*) malloc(n * sizeof(int));
    values = (int*) malloc(n * sizeof(int));

    printf("Enter weights and values of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &weights[i], &values[i]);
    }

    // Backtracking approach
    maxValBacktracking = 0;
    knapsackBacktracking(0, 0, 0);
    printf("Backtracking Maximum Value: %d\n", maxValBacktracking);

    // Branch & Bound approach
    int maxValBranchBound = knapsackBranchBound();
    printf("Branch & Bound Maximum Value: %d\n", maxValBranchBound);

    return 0;
}
```