DAA Lab

Assignment 1

```
#include <iostream>
using namespace std;
void bubble(int a[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
int binary(int a[], int n, int key)
{
    int low = 0;
    int high = n - 1;

    while (low <= high)
    {
        int mid = low + (high - low) / 2;
        if (a[mid] == key)
        {
            return mid;
        }
        else if (a[mid] < key)
        {
            low = mid + 1;
        }
        else
        {
            high = mid - 1;
        }
    }
```

```
    return -1;
}
int main()
{
    int n;
    cout << "Enter no of elements" << endl;
    cin >> n;

    int a[n];
    cout << "Enter elements in array" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    int key;
    cout << "Enter target element " << endl;
    cin >> key;
    bubble(a, n);

    int result = binary(a, n, key);
    if (result == -1)
    {
        cout << "KEY NOT FOUND" << endl;
    }
    else
    {
        cout << "Target" << key << "found at index" << result;
    }

    return 0;
}
```

Assignment 2

```
#include <iostream>
using namespace std;
int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // Choose the last element as pivot
    int i = low - 1;       // Index of smaller element
    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
```

```cpp
            i++;                  // Increment index of smaller element
            swap(arr[i], arr[j]); // Swap if element is smaller than pivot
        }
    }
    swap(arr[i + 1], arr[high]); // Place pivot in the correct position
    return i + 1;                // Return the index of the pivot
}
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high); // Partitioning index
        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main()
{
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    // Dynamically allocate an array
    int *arr = new int[n];
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    // Perform Quick Sort
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    // Free the allocated memory
    delete[] arr;
    return 0;
}
```

Assignment 3

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
struct node
{
    int freq;
    char data;
    const node *child0, *child1;

    node(char d, int f = -1)
    {
        data = d;
        freq = f;
        child0 = NULL;
        child1 = NULL;
    }

    node(const node *c0, const node *c1)
    {
        data = 0;
        freq = c0->freq + c1->freq;
        child0 = c0;
        child1 = c1;
    }

    bool operator<(const node &a) const
    {
        return freq > a.freq;
    }

    void traverse(string code = "") const
    {
        if (child0 != NULL)
        {
            child0->traverse(code + '0');
            child1->traverse(code + '1');
        }
        else
        {
            cout << "Data" << data << "Freq" << freq << "code" << code << endl;
        }
    }
```

```
};
void huffman(string str)
{
    priority_queue<node> qu;
    int frequency[256];
    for (int i = 0; i < 256; i++)
        frequency[i] = 0;
    for (int i = 0; i < str.size(); i++)
    {
        frequency[int(str[i])]++;
    }
    for (int i = 0; i < 256; i++)
    {
        if (frequency[i])
        {
            qu.push(node(i, frequency[i]));
        }
    }
    while (qu.size() > 1)
    {
        node *c0 = new node(qu.top());
        qu.pop();

        node *c1 = new node(qu.top());
        qu.pop();
        qu.push(node(c0, c1));
    }
    cout << "The Huffman Code" << endl;
    qu.top().traverse();
}
int main()
{
    string str;
    cout << "Enter SRING" << endl;
    cin >> str;
    huffman(str);
    return 0;
}
```

Assignment 4

```
#include <iostream>
using namespace std;

struct Item {
    int value;
    int weight;
};

void selectionSort(Item arr[], int N) {
    for (int i = 0; i < N - 1; i++) {
        int maxIndex = i;
        for (int j = i + 1; j < N; j++) {
            double r1 = (double)arr[maxIndex].value / arr[maxIndex].weight;
            double r2 = (double)arr[j].value / arr[j].weight;
            if (r2 > r1) {
                maxIndex = j;
            }
        }
        swap(arr[i], arr[maxIndex]);
    }
}

double fractionalKnapsack(int W, Item arr[], int N) {

    selectionSort(arr, N);

    double totalValue = 0.0;
    for (int i = 0; i < N; i++) {
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
            totalValue += arr[i].value;
        } else {
            totalValue += arr[i].value * ((double)W / arr[i].weight);
            break;
        }
    }

    return totalValue;
}

int main() {
    int W = 50;
    Item arr[] = {{60, 10}, {100, 20}, {120, 30}};
```

```
    int N = sizeof(arr) / sizeof(arr[0]);

    double maxValue = fractionalKnapsack(W, arr, N);
    cout << "Maximum value in Knapsack = " << maxValue << endl;

    return 0;
}
```

Assignment 5

```
#include <iostream>
using namespace std;
int knapsack(int W, int val[], int wt[], int n)
{
    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapsack(W, val, wt, n - 1);

    else
        return max(knapsack(W, val, wt, n - 1), val[n - 1] + knapsack(W - wt[n -
1], val, wt, n - 1));
}
int main()
{
    int n, W;
    cout << "Enter no of items" << endl;
    cin >> n;

    int profit[n], weight[n];
    cout << "Enter profits" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> profit[i];
    }

    cout << "Enter weight" << endl;
    for (int i = 0; i < n; i++)
    {
```

```
        cin >> weight[i];
    }

    cout << "Knapsack cap" << endl;
    cin >> W;

    cout << "Max capacity" << knapsack(W, profit, weight, n);
    return 0;
}
```

Assignment 6

```
#include <iostream>
#include <climits>
using namespace std;
int sum(int prefixsum[], int i, int j)
{
    if (i == 0)
        return prefixsum[j]; // Fixed to return the sum up to j
    return prefixsum[j] - prefixsum[i - 1];
}
int obst(int keys[], int freq[], int n)
{
    int cost[100][100];

    // Initialize prefix sums
    int prefixsum[100];
    prefixsum[0] = freq[0];
    for (int i = 1; i < n; i++)
        prefixsum[i] = prefixsum[i - 1] + freq[i];

    // Initialize cost for single keys
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Calculate cost for lengths from 2 to n
    for (int len = 2; len <= n; len++)
    {
        for (int i = 0; i <= n - len; i++)
        {
            int j = i + len - 1;
```

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

            int freqsum = sum(prefixsum, i, j);

            // Check for each root position
            for (int r = i; r <= j; r++)
            {
                int leftcost = (r > i) ? cost[i][r - 1] : 0;
                int rightcost = (r < j) ? cost[r + 1][j] : 0;
                int totalcost = leftcost + rightcost + freqsum;

                if (totalcost < cost[i][j])
                    cost[i][j] = totalcost;
            }
        }
    }

    return cost[0][n - 1]; // Return the cost of OBST for the entire range
}
int main()
{
    int n;
    cout << "Enter no of keys" << endl;
    cin >> n;
    int keys[100], freq[100];

    cout << "Enter keys" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> keys[i];
    }

    cout << "Enter frequency" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> freq[i];
    }

    cout << "Cost of OBST: " << obst(keys, freq, n) << endl; // Added space for
clarity

    return 0;
}
```

Assignment 7

```
#include <iostream>
using namespace std;

// Function to print the board with queens placed
void print(int board[10][10], int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (board[i][j])
                cout << "Q "; // Queen represented by 'Q'
            else
                cout << ". "; // Empty cell represented by '.'
        }
        cout << "\n";
    }
}

// Function to check if a queen can be placed at board[row][col]
bool safe(int board[10][10], int row, int col, int n)
{
    // Check the row on the left
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check the upper diagonal on the left
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check the lower diagonal on the left
    for (int i = row, j = col; j >= 0 && i < n; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// Recursive function to place queens
bool soln(int board[10][10], int col, int n)
{
```

```
    // Base case: if all queens are placed, return true
    if (col >= n)
        return true;

    // Try placing a queen in each row of the current column
    for (int i = 0; i < n; i++)
    {
        if (safe(board, i, col, n))
        {
            board[i][col] = 1; // Place queen

            // Recur to place rest of the queens
            if (soln(board, col + 1, n))
                return true;

            board[i][col] = 0; // Backtrack if no solution found
        }
    }
    return false;
}

// Function to solve the N-Queens problem
bool solve(int n)
{
    int board[10][10] = {0}; // Initialize the board with all 0s

    if (!soln(board, 0, n))
    {
        cout << "No Solution\n";
        return false;
    }

    print(board, n);
    return true;
}

int main()
{
    int n = 4; // Set n to 4 for the 4-Queens problem
    solve(n);
    return 0;
}
```

Assignment 8

```
#include <iostream>
using namespace std;
void print(int color[], int nodes)
{
    for (int u = 0; u < nodes; u++)
    {
        cout << "Node " << u << " color assigned: " << color[u] << endl;
    }
}
bool isafe(int u, int color[], int c, int graph[][6], int nodes)
{
    for (int v = 0; v < nodes; v++)
    {
        if (graph[u][v] && color[v] == c)
            return false;
    }
    return true;
}
bool graphColoringUtil(int m, int color[], int u, int graph[][6], int nodes)
{
    if (u == nodes)
        return true;
    for (int c = 0; c < m; c++)
    {
        if (isafe(u, color, c, graph, nodes))
        {
            color[u] = c;
            if (graphColoringUtil(m, color, u + 1, graph, nodes))
                return true;
            color[u] = -1;
        }
    }
    return false;
}
void colori(int m, int nodes, int graph[][6])
{
    int color[6];
    for (int i = 0; i < nodes; i++)
        color[i] = -1;

    if (graphColoringUtil(m, color, 0, graph, nodes))
        print(color, nodes);
    else
```

```
        cout << "No solution" << endl;
}
int main()
{
    int nodes, edges, m;
    cout << "Enter number of nodes: ";
    cin >> nodes;
    cout << "Enter number of edges: ";
    cin >> edges;

    int graph[6][6] = {0};
    cout << "Enter edges (node1 node2):" << endl;
    for (int i = 0; i < edges; i++)
    {
        int u, v;
        cin >> u >> v;
        graph[u][v] = 1;
        graph[v][u] = 1;
    }
    cout << "Enter number of colors: ";
    cin >> m;
    colori(m, nodes, graph);
    return 0;
}
```

Assignment 9

```
#include <bits/stdc++.h>
using namespace std;

// Structure to store information about a node in the state space tree
struct Node {
    int level;    // Level of the node in the tree (item index)
    int profit;   // Total profit up to this node
    int weight;   // Total weight up to this node
    float bound;  // Upper bound on the maximum profit achievable from this node
};

// Comparator for sorting nodes based on their bound values (higher bound, higher
priority)
struct CompareBound {
    bool operator()(Node const& n1, Node const& n2) {
        return n1.bound < n2.bound;  // Use for max-heap (greater bound, higher
priority)
```

```
    }
};

// Function to calculate the upper bound on profit for a node
float calculateBound(Node u, int n, int m, int W[], int P[]) {
    if (u.weight >= m) return 0; // If weight exceeds capacity, bound is 0
(invalid)

    float profitBound = u.profit;
    int totalWeight = u.weight;
    int j = u.level + 1;

    // Calculate upper bound using a greedy approach (fractional knapsack)
    while (j < n && totalWeight + W[j] <= m) {
        totalWeight += W[j];
        profitBound += P[j];
        j++;
    }

    // If there's still room in the knapsack, add fractional profit from the next
item
    if (j < n) {
        profitBound += (m - totalWeight) * ((float)P[j] / W[j]);
    }

    return profitBound;
}

// Function to solve the 0/1 Knapsack problem using Branch and Bound
int knapsackBranchAndBound(int n, int W[], int P[], int m) {
    // Create a priority queue to explore nodes (max-heap based on bound)
    priority_queue<Node, vector<Node>, CompareBound> pq;

    // Initial root node (level = -1, profit = 0, weight = 0)
    Node u, v;
    u.level = -1;
    u.profit = 0;
    u.weight = 0;
    u.bound = calculateBound(u, n, m, W, P);

    pq.push(u);

    int maxProfit = 0;

    while (!pq.empty()) {
```

```
        u = pq.top(); // Get the node with the highest bound
        pq.pop();

        // If bound is less than current max profit, prune the node
        if (u.bound <= maxProfit) continue;

        // Explore the next level (next item)
        v.level = u.level + 1;

        // Case 1: Include the current item in the knapsack
        v.weight = u.weight + W[v.level];
        v.profit = u.profit + P[v.level];

        if (v.weight <= m && v.profit > maxProfit) {
            maxProfit = v.profit; // Update the maximum profit
        }

        v.bound = calculateBound(v, n, m, W, P);

        if (v.bound > maxProfit) {
            pq.push(v); // Only add the node if it has potential for better
profit
        }

        // Case 2: Exclude the current item from the knapsack
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = calculateBound(v, n, m, W, P);

        if (v.bound > maxProfit) {
        pq.push(v);
        }
    }

    return maxProfit;
}

int main() {
    int P[10], W[10], n, m;

    cout << "Enter No. of elements: ";
    cin >> n;

    cout << "Enter the capacity of knapsack: ";
    cin >> m;
```

```
    for (int i = 0; i < n; i++) {
        cout << "Enter the Profit and Weight of Object " << i + 1 << ": ";
        cin >> P[i] >> W[i];
    }

    cout << "\nMaximum Profit using Branch and Bound: " <<
knapsackBranchAndBound(n, W, P, m) << endl;

    return 0;
}
```

Assignment 10

```
// Develop a program for Traveling Salesman Problem using Branch and Bound.

#include <iostream>
#include <climits> // for INT_MAX
using namespace std;
#define MAX_V 10          // Maximum number of cities
int graph[MAX_V][MAX_V]; // Adjacency matrix
int n;                    // Number of cities
struct Node
{
    int level;        // Current level (city index)
    int cost;         // Cost of path so far
    int bound;        // Lower bound of the node
    int path[MAX_V]; // Store the path taken
};
// Function to calculate the lower bound for the given node
int calculateBound(Node &u)
{
    int bound = 0;

    // Add cost of edges from the current node
    // Include outgoing edges to the next node
    bool visited[MAX_V] = {false};
    for (int i = 0; i < u.level; i++)
    {
        visited[u.path[i]] = true;
    }
    // Add minimum cost of outgoing edges
```

```
    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            int minEdge = INT_MAX;
            for (int j = 0; j < n; j++)
            {
                if (i != j && !visited[j])
                {
                    minEdge = min(minEdge, graph[i][j]);
                }
            }
            if (minEdge != INT_MAX)
            {
                bound += minEdge;
            }
        }
    }
    // Include the cost of the edges that have been taken
    bound += u.cost;

    return bound;
}
// Branch and Bound TSP
void branchAndBound()
{
    Node minNode; // To keep track of the minimum cost node
    minNode.level = 0;
    minNode.cost = 0;
    minNode.path[0] = 0;        // Start from the first city
    Node queue[MAX_V * MAX_V]; // Static array for nodes
    int front = 0, rear = 0;    // Queue pointers
    queue[rear++] = minNode;
    minNode.bound = calculateBound(minNode);
    int minCost = INT_MAX;
    while (front < rear)
    {
        // Remove the node with the minimum bound from the queue
        Node currentNode = queue[front++];
        // If the level is n - 1, all cities are visited
        if (currentNode.level == n - 1)
        {
            // Cost to return to the starting city
            int totalCost = currentNode.cost +
                            graph[currentNode.path[currentNode.level]][0];
```

```
            minCost = min(minCost, totalCost);
            continue;
        }
        // Explore further
        for (int i = 0; i < n; i++)
        {
            if (currentNode.level < n &&
                graph[currentNode.path[currentNode.level]][i])
            {
                Node newNode;
                newNode.level = currentNode.level + 1;
                newNode.cost = currentNode.cost +
                               graph[currentNode.path[currentNode.level]][i];
                for (int j = 0; j <= currentNode.level; j++)
                {
                    newNode.path[j] = currentNode.path[j];
                }
                newNode.path[newNode.level] = i;
                newNode.bound = calculateBound(newNode);

                // If the bound is less than the current minimum cost
                if (newNode.bound < minCost)
                {
                    queue[rear++] = newNode;
                }
            }
        }
    }
    cout << "Minimum cost to visit all cities: " << minCost << endl;
}
int main()
{
    cout << "Enter the number of cities: ";
    cin >> n;
    cout << "Enter the adjacency matrix (distance between cities):" << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> graph[i][j];
        }
    }
    branchAndBound();
    return 0;
}
```