Assignment 2 Solution

1. **Demo solution for question 1**

```
#include <ibits/stdc++.h>

using namespace std;

struct State {
    int level, stamina, sleep_time;
    bool operator>(const State& other) const {
        return sleep_time > other.sleep_time;
    }
};

int main() {
    int N, M, A, B;
    cin >> N >> M >> A >> B;

    vector<vector<pair<int, int>>> transitions(N + 1);  // {destination,
stamina_cost}

    // Input all the transitions
    for (int i = 0; i < M; i++) {
        int X, Y, Z;
        cin >> X >> Y >> Z;
        transitions[X].push_back({Y, Z});
        transitions[Y].push_back({X, Z});
    }

    vector<int> sleep(N + 1), stamina(N + 1);

    // Input the stamina and sleep time for each skill level
    for (int i = 1; i <= N; i++) {
        cin >> stamina[i] >> sleep[i];
    }

    // Distance (sleep time) and stamina table
    vector<vector<int>> min_sleep(N + 1, vector<int>(101, INT_MAX)); // N levels,
stamina capped at 100
    min_sleep[A][0] = 0;
    // Priority queue for Dijkstra-like search: {sleep_time, level, stamina}
    priority_queue<State, vector<State>, greater<State>> pq;
    pq.push({A, 0, 0});  // Starting at level A with 0 stamina and 0 sleep time

    while (!pq.empty()) {
        State curr = pq.top();
        pq.pop();

        int level = curr.level;
        int curr_stamina = curr.stamina;
        int curr_sleep = curr.sleep_time;

        // If we reached the ultimate level B, return the result
        if (level == B) {
            cout << curr_sleep << endl;
            return 0;
        }

        // Skip if we already have a better way to this state
        if (curr_sleep > min_sleep[level][curr_stamina]) continue;

        // 1. Try all transitions from the current level
        for (auto [next_level, cost] : transitions[level]) {
            if (curr_stamina >= cost) {
                int new_stamina = curr_stamina - cost;
                if (curr_sleep < min_sleep[next_level][new_stamina]) {
                    min_sleep[next_level][new_stamina] = curr_sleep;
                    pq.push({next_level, new_stamina, curr_sleep});
                }
            }
        }

        // 2. Try eating the devil fruit at the current level
        int new_stamina = stamina[level];
        int sleep_cost = sleep[level];
        if (curr_sleep + sleep_cost < min_sleep[level][new_stamina]) {
            min_sleep[level][new_stamina] = curr_sleep + sleep_cost;
            pq.push({level, new_stamina, curr_sleep + sleep_cost});
        }
    }

    // If no path to the ultimate level B was found
    cout << -1 << endl;
    return 0;
}
```

2. Demo solution for question 1

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>

using namespace std;

int primMST(int a, vector<vector<pair<int, int>>> &graph) {
    vector<bool> visited(a, false);  // To keep track of visited nodes
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;  // Min-heap {weight, vertex}
    int mst_edges = 0;
    int count = 0;  // Number of nodes included in the MST

    // Start from node 0 (you can start from any node)
    pq.push({0, 0});

    while (!pq.empty() && count < a) {
        pair<int, int> current = pq.top();  // Get the smallest edge
        pq.pop();

        int weight = current.first;
        int u = current.second;

        // If the node is already visited, skip it
        if (visited[u]) continue;

        // Mark this node as visited and add it to the MST
        visited[u] = true;
        count++;  // Count the number of nodes added to the MST
        if (weight != 0) {
            mst_edges++;  // Only count edges, not starting node
        }

        // Explore all the neighbors of this node
        for (int i = 0; i < graph[u].size(); i++) {
            int v = graph[u][i].first;
            int w = graph[u][i].second;
            if (!visited[v]) {
                pq.push({w, v});  // Add edges to the priority queue
            }
        }
    }
    return mst_edges;
}

int main() {
    int a, b;
    cin >> a >> b;

    vector<vector<pair<int, int>>> graph(a);

    // Reading the cab routes (edges)
    for (int i = 0; i < b; i++) {
        int u, v;
        cin >> u >> v;
        u--; v--;  // Convert to 0-based index
        graph[u].push_back({v, 1});  // We use weight 1 for every edge (since they
are all equal)
        graph[v].push_back({u, 1});
    }

    // Find the minimum spanning tree using Prim's algorithm
    int result = primMST(a, graph);

    cout << result << endl;

    return 0;
}
```