Q1.) Write a program to convert NFA to DFA

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <map>

using namespace std;

class NFAState {
public:
    map<char, set<int>> transitions; // Transitions on input symbols
    set<int> epsilonTransitions;     // Epsilon transitions

    void addTransition(char input, int nextState) {
        transitions[input].insert(nextState);
    }

    void addEpsilonTransition(int nextState) {
        epsilonTransitions.insert(nextState);
    }
};

class DFAState {
public:
    set<int> nfaStates;
};

class NFA {
    vector<NFAState> states;

public:
    NFA(int numStates) : states(numStates) {}

    void addTransition(int state, char input, int nextState) {
        states[state].addTransition(input, nextState);
    }

    void addEpsilonTransition(int state, int nextState) {
        states[state].addEpsilonTransition(nextState);
    }

    set<int> epsilonClosure(const set<int>& startStates) const {
        set<int> closure = startStates;
```

```
        vector<int> stack(startStates.begin(), startStates.end());

        while (!stack.empty()) {
            int current = stack.back();
            stack.pop_back();

            for (int nextState : states[current].epsilonTransitions) {
                if (closure.insert(nextState).second) {
                    stack.push_back(nextState);
                }
            }
        }
        return closure;
    }

    set<int> transition(const set<int>& states, char symbol) const {
        set<int> result;
        for (int state : states) {
            auto it = this->states[state].transitions.find(symbol);
            if (it != this->states[state].transitions.end()) {
                result.insert(it->second.begin(), it->second.end());
            }
        }
        return epsilonClosure(result);
    }

    const vector<NFAState>& getStates() const {
        return states;
    }
};

class DFA {
    vector<DFAState> states;

public:
    void addState(const set<int>& nfaStates) {
        states.push_back({nfaStates});
    }

    size_t size() const {
        return states.size();
    }

    const DFAState& operator[](size_t index) const {
```

```
        return states[index];
    }

    DFAState& operator[](size_t index) {
        return states[index];
    }

    vector<DFAState> getStates() const {
        return states;
    }
};

DFA convertNFAtoDFA(const NFA& nfa, int startState, const set<char>& alphabet) {
    DFA dfa;
    map<set<int>, int> stateMapping;

    set<int> startStates;
    startStates.insert(startState);
    set<int> startClosure = nfa.epsilonClosure(startStates);
    stateMapping[startClosure] = 0;
    dfa.addState(startClosure);

    for (size_t i = 0; i < dfa.size(); ++i) {
        const set<int>& currentState = dfa[i].nfaStates;

        for (char c : alphabet) {
            set<int> newState = nfa.transition(currentState, c);

            if (!newState.empty() && stateMapping.find(newState) == stateMapping.end())
{
                stateMapping[newState] = dfa.size();
                dfa.addState(newState);
            }
        }
    }

    return dfa;
}

void printDfaTable(const DFA& dfa, const set<char>& alphabet, const NFA& nfa) {
    cout << "DFA Transition Table:" << endl;
    cout << "DFA State\t|\t";
    for (char c : alphabet) {
        cout << c << "\t";
```

```
    }
    cout << endl;

    for (size_t i = 0; i < dfa.size(); i++) {
        cout << "DFA State " << i << "\t|\t";
        for (char c : alphabet) {
            set<int> result = nfa.transition(dfa[i].nfaStates, c);
            if (!result.empty()) {
                //Find the DFA state corresponding to the NFA state set
                for (size_t j = 0; j < dfa.size(); j++) {
                    if (dfa[j].nfaStates == result) {
                        cout << j << "\t";
                        break;
                    }
                }
            } else {
                cout << "-\t";
            }
        }
        cout << endl;
    }
}

int main() {
    int numStates;
    cout << "Enter the number of states in the NFA: ";
    cin >> numStates;

    NFA nfa(numStates);
    set<char> alphabet;
    int numAlphabets;
    cout << "Enter the number of alphabets (excluding epsilon): ";
    cin >> numAlphabets;

    cout << "Enter the alphabets (one character each): ";
    for (int i = 0; i < numAlphabets; ++i) {
        char symbol;
        cin >> symbol;
        alphabet.insert(symbol);
    }

    int numTransitions;
    cout << "Enter the number of transitions: ";
    cin >> numTransitions;
```

```
    cout << "Enter transitions in the format: [state] [input] [next state]. Use 'e' for
epsilon transitions.\n";

    for (int i = 0; i < numTransitions; ++i) {
        int state, nextState;
        char input;
        cin >> state >> input >> nextState;
        if (input == 'e') {
            nfa.addEpsilonTransition(state, nextState);
        } else {
            nfa.addTransition(state, input, nextState);
        }
    }

    int startState;
    cout << "Enter the start state: ";
    cin >> startState;

    // Converting NFA to DFA
    DFA dfa = convertNFAtoDFA(nfa, startState, alphabet);

    // Print DFA Table
    printDfaTable(dfa, alphabet, nfa);

    return 0;
}
```

Output)

```
PS C:\Users\Mukul  Dev\OneDrive\Desktop\Mukul\DTU\Year 3\Sem 6\Compiler Design\Lab> cd "c:\Users
ign\Lab\" ; if ($?) { g++ nfa2dfa.cpp -o nfa2dfa } ; if ($?) { .\nfa2dfa }
Enter the number of states in the NFA: 3
Enter the number of alphabets (excluding epsilon): 2
Enter the alphabets (one character each): a
b
Enter the number of transitions: 4
Enter transitions in the format: [state] [input] [next state]. Use 'e' for epsilon transitions.
0 a 0
0 a 1
0 b 0
1 b 2
Enter the start state: 0
DFA Transition Table:
DFA State       |       a       b
DFA State 0     |       1       0
DFA State 1     |       1       2
DFA State 2     |       1       0
```