PROJECT TITLE:

Cash Flow Minimizer: Settling Debts with the Minimum Number of Transactions"

DESCRIPTION:

This project focuses on finding the minimum number of transactions needed to settle debts between a group of people. Each person in the group may owe money to others, and the goal is to figure out how to clear all debts with the least amount of money exchanged.

The system will allow users to input the amount of money each person owes to others. The system will then calculate how to minimize the number of transactions so that each person's debt is cleared. The solution should ensure that no person ends up owing money after the transactions.

PROJECT ANALYSIS:

The cash flow minimization problem involves optimizing debt settlements among multiple parties to:

1. Clear all outstanding debts

2. Minimize the number of transactions

3. Ensure no party ends up with unresolved credits or debits

MATHEMATICAL FOUNDATION:

The problem can be modeled using graph theory:

- Vertices: Represent individuals

- Edges: Represent debts (directed, weighted)

- Net Amount: For each vertex, net = Σ(incoming) - Σ(outgoing)

KEY INSIGHT:

The solution relies on the fact that only net amounts matter for settlement, not individual transactions.

ALGORITHM:

PHASE 1: NET CALCULATION

1. For each person P_i:

   - Calculate total credit (amounts owed to P_i)

   - Calculate total debit (amounts P_i owes)

   - Net amount = Credit - Debit

PHASE 2: TRANSACTION OPTIMIZATION

1. Create two priority queues:
   - Max-heap for creditors (positive net)
   - Min-heap for debtors (negative net)
2. While both heaps are non-empty:
   a. Extract max creditor C and max debtor D
   b. Transaction amount = min(C.net, -D.net)
   c. Create transaction D → C for this amount
   d. Update nets:
      - C.net -= amount
      - D.net += amount
   e. If updated net ≠ 0, reinsert to respective heap

CODE:

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <unordered_map>
using namespace std;
struct Person {
    string name;
    int netAmount;
};
struct Transaction {
    string from;
    string to;
    int amount;
    void print() const {
        cout << from << " pays " << to << " $" << amount << endl;
    }
};
class CashFlowMinimizer {
```

```
private:
    vector<string> people;
    unordered_map<string, int> nameToIndex;
    vector<vector<int>> graph;
public:
    CashFlowMinimizer(const vector<string>& names) : people(names) {
        int n = people.size();
        graph.resize(n, vector<int>(n, 0));

        for (int i = 0; i < n; i++) {
            nameToIndex[people[i]] = i;
        }
    }
    void addDebt(const string& from, const string& to, int amount) {
        int fromIdx = nameToIndex[from];
        int toIdx = nameToIndex[to];
        graph[fromIdx][toIdx] += amount;
    }
    vector<Transaction> minimizeTransactions() {
        int n = people.size();
        vector<int> netAmounts(n, 0);
        // Calculate net amounts
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                netAmounts[i] += (graph[j][i] - graph[i][j]);
            }
     }
        // Create heaps
        auto creditorCmp = [](const Person& a, const Person& b) {
            return a.netAmount < b.netAmount;
        };
        auto debtorCmp = [](const Person& a, const Person& b) {
```

```
            return a.netAmount > b.netAmount;
        };
        priority_queue<Person, vector<Person>,
decltype(creditorCmp)> creditors(creditorCmp);
        priority_queue<Person, vector<Person>, decltype(debtorCmp)>
debtors(debtorCmp);
        for (int i = 0; i < n; i++) {
            if (netAmounts[i] > 0) {
                creditors.push({people[i], netAmounts[i]});
            } else if (netAmounts[i] < 0) {
                debtors.push({people[i], netAmounts[i]});
            }
        }
        vector<Transaction> transactions;
        while (!creditors.empty() && !debtors.empty()) {
            Person creditor = creditors.top();
            Person debtor = debtors.top();
            creditors.pop();
            debtors.pop();
            int amount = min(creditor.netAmount, -debtor.netAmount);
            transactions.push_back({debtor.name, creditor.name,
amount});
            creditor.netAmount -= amount;
            debtor.netAmount += amount;
            if (creditor.netAmount > 0) {
                creditors.push(creditor);
            }
            if (debtor.netAmount < 0) {
                debtors.push(debtor);
            }
        }
        return transactions;
    }
};
```

```
int main() {
    vector<string> people = {"Eli", "Barb", "Mike"};
    // Initialize the cash flow minimizer
    CashFlowMinimizer minimizer(people);
    // Add debts between these three people
    minimizer.addDebt("Eli", "Barb", 1000);    // Eli owes Barb
$1000
    minimizer.addDebt("Eli", "Mike", 2000); // Eli owes Mike $2000
    minimizer.addDebt("Barb", "Mike", 5000);   // Barb owes Mike
$5000
    minimizer.addDebt("Mike", "Eli", 3000); // Mike owes Eli $3000
    // Calculate the minimal transactions
    vector<Transaction> transactions =
minimizer.minimizeTransactions();
    // Print the results
    cout << "Minimum number of transactions required: " <<
transactions.size() << endl;
    cout << "Optimal transactions between Alice, Bob and Charlie:"
<< endl;
    for (const auto& t : transactions) {
        t.print();
    }
    return 0;
}
```

EXAMPLE WALKTHROUGH:

Input:

- Participants: Eli, Barb, Mike

- Debts:

  - Eli→ Barb: $1000

  - Eli → Mike: $2000

  - Barb → Mike: $5000

  - Mike → Eli: $3000

EXECUTION STEPS:

1. Net Calculation:

   - Eli: 3000 - (1000 + 2000) = 0

   - Barb: 1000 - 5000 = -4000

   - Mike: (2000 + 5000) - 3000 = 4000

2. Heap Initialization:

   - Creditors: Mike ($4000)

   - Debtors: Barb (-$4000)

3. Transaction Processing:

   - First transaction: Barb → Mike $4000

   - Heaps become empty after one transaction

Output:

Minimum transactions required: 1

Barb pays Mike $4000

COMPLEXITY ANALYSIS:

1. Time Complexity:

   - Net calculation: $O(n^2)$ for n people

   - Heap operations: $O(n \log n)$ in worst case

   - Total: $O(n^2)$

2. Space Complexity:

   - $O(n^2)$ for storing the debt graph

   - $O(n)$ for net amounts and heaps

PRACTICAL APPLICATIONS:

1. Roommate Expense Splitting

2. Business Partner Settlements

3. Group Travel Expense Reconciliation

4. Cryptocurrency Payment Channels

5. Banking Settlement Systems

This comprehensive solution provides an optimal, efficient method for debt minimization with clear mathematical foundations and practical implementation.