<https://leetcode.ca/2017-03-09-465-Optimal-Account-Balancing/>

You are given an array of transactions transactions where transactions[i] = [fromi, toi, amounti] indicates that the person with ID = fromi gave amounti to the person with ID = toi.

Return *the minimum number of transactions required to settle the debt*.

**Example 1:**

**Input:** transactions = [[0,1,10],[2,0,5]]

**Output:** 2

**Explanation:**

Person #0 gave person #1 10.

Person #2 gave person #0 5.

Two transactions are needed.

One way to settle the debt is person #1 pays person #0 and #2 5 each.

**Example 2:**

**Input:** transactions = [[0,1,10],[1,0,1],[1,2,5],[2,0,5]]

**Output:** 1

**Explanation:**

Person #0 gave person #1 10.

Person #1 gave person #0 1.

Person #1 gave person #2 5.

Person #2 gave person #0 5.

Therefore, person #1 only need to give person #0 4, and all debt is settled.

**Constraints:**

1 <= transactions.length <= 8

transactions[i].length == 3

0 <= fromi, toi < 12

fromi != toi

1 <= amounti <= 100

**Attempt 1: 2025-04-30**

**Solution 1: Backtracking + Math (180 min)**

**Refer to Deepseek**

**Problem Understanding**

This problem requires us to determine the minimum number of transactions needed to settle all debts among a group of people, where each transaction involves transferring money from one person to another.

**Approach**

We'll use a backtracking approach to explore all possible ways to settle the debts:

Calculate Net Balances: First compute each person's net balance (total owed minus total owes).

Filter Non-Zero Balances: Only consider people with non-zero balances.

Backtracking Search: Try to match positive balances with negative balances to settle debts.

**Solution Code**

class Solution {

public int minTransfers(int[][] transactions) {

// Step 1: Calculate net balance for each person

Map<Integer, Integer> balanceMap = new HashMap<>();

for (int[] t : transactions) {

int from = t[0], to = t[1], amount = t[2];

balanceMap.put(from, balanceMap.getOrDefault(from, 0) - amount);

balanceMap.put(to, balanceMap.getOrDefault(to, 0) + amount);

}

// Step 2: Collect non-zero balances

List<Integer> debts = new ArrayList<>();

for (int balance : balanceMap.values()) {

if (balance != 0) {

debts.add(balance);

}

}

// Step 3: Backtrack to find minimum transactions

return backtrack(debts, 0);

}

private int backtrack(List<Integer> debts, int start) {

// Skip settled debts

while (start < debts.size() && debts.get(start) == 0) {

start++;

}

if (start == debts.size()) {

return 0;

}

int minTransactions = Integer.MAX\_VALUE;

for (int i = start + 1; i < debts.size(); i++) {

// Only consider opposite signs

if (debts.get(start) \* debts.get(i) < 0) {

// Settle debt between start and i

debts.set(i, debts.get(i) + debts.get(start));

minTransactions = Math.min(minTransactions, 1 + backtrack(debts, start + 1));

// Backtrack

debts.set(i, debts.get(i) - debts.get(start));

}

}

return minTransactions;

}

}

**Explanation**

Net Balance Calculation: We first compute how much each person owes or is owed after all transactions.

Debt List Creation: We create a list containing only non-zero balances (people who actually need to settle debts).

Backtracking:

We try to match each positive balance with negative balances.

For each match, we update the balances and recursively process the remaining debts.

We keep track of the minimum number of transactions needed.

Optimization: The algorithm skips already settled debts and only considers opposite sign balances for matching.

**Complexity Analysis**

Time Complexity: O(N!), where N is the number of non-zero balances. This is because in the worst case we might need to explore all permutations of debt settlements.

Space Complexity: O(N) for the recursion stack and storing the debt list.

This approach efficiently explores all possible ways to settle debts while minimizing the number of transactions through backtracking.

**i see another solution as below from page:** <https://algo.monster/liteproblems/465,> **how this solution works ? and which is better ?**

class Solution {

public int minTransfers(int[][] transactions) {

// The array 'balance' will hold the net amount for up to 12 individuals

// Negative values mean the person needs to pay that amount, positive values mean the person should receive that amount

int[] balance = new int[12];

// Calculate the balance for each person involved in the transactions

for (int[] transaction : transactions) {

balance[transaction[0]] -= transaction[2]; // person paying out

balance[transaction[1]] += transaction[2]; // person receiving payment

}

// Create a list to store non-zero balances (amounts that need to be settled)

List<Integer> nonZeroBalances = new ArrayList<>();

for (int b : balance) {

if (b != 0) {

nonZeroBalances.add(b);

}

}

// Prepare to find the minimum number of transactions to settle all debts

int numAccounts = nonZeroBalances.size();

int[] minTransfers = new int[1 << numAccounts]; // 1<<numAccounts is 2^numAccounts

Arrays.fill(minTransfers, Integer.MAX\_VALUE / 2); // Initialize with a large value

minTransfers[0] = 0; // No transfers needed when there is no debt

// Loop through all possible subsets of debts

for (int i = 1; i < (1 << numAccounts); ++i) {

int sum = 0;

// Calculate the sum of balances in the current subset

for (int j = 0; j < numAccounts; ++j) {

if ((i >> j & 1) == 1) { // If the j-th person is in the current subset (i)

sum += nonZeroBalances.get(j);

}

}

// If the sum is zero, then the current subset can be settled among themselves

if (sum == 0) {

// Set initial transfers for this subset as the number of involved accounts minus 1 transfer

minTransfers[i] = Integer.bitCount(i) - 1;

// Try to split the subset into two parts and minimize their transfers

for (int j = (i - 1) & i; j > 0; j = (j - 1) & i) {

// Update the minimum transfers for the current subset

minTransfers[i] = Math.min(minTransfers[i], minTransfers[j] + minTransfers[i ^ j]);

}

}

}

// Return the number of transactions for the set including all non-zero balances

return minTransfers[(1 << numAccounts) - 1];

}

}

The solution you found from algo.monster uses a \*\*bitmask dynamic programming\*\* approach to solve the Optimal Account Balancing problem. Let me explain how it works and compare it with the backtracking solution.

**How the Bitmask DP Solution Works**

Balance Calculation:

Computes net balances for each person (same as backtracking solution)

Non-Zero Balances:

Creates a list of only non-zero balances (same as backtracking)

Bitmask DP Setup:

Uses a bitmask to represent subsets of people with debts

minTransfers[mask] stores minimum transactions for subset represented by mask

DP Transitions:

For each subset (bitmask), checks if sum of balances is zero (settleable)

If settleable, calculates minimum transactions as:

Either (number of people in subset) - 1 (complete graph)

Or the sum of two smaller subsets' transactions

Final Result:

Returns minTransfers[full\_mask] which covers all people

**Comparison with Backtracking Solution**

**Bitmask DP Solution**

Pros:

More efficient for small number of people (≤12)

Systematic exploration of all subsets

Avoids redundant calculations through DP memoization

Cons:

Space complexity O(2^N) limits to small N (~12)

More complex implementation

Harder to understand and modify

**Backtracking Solution**

Pros:

Simpler to implement and understand

Works for any number of people

More intuitive recursive approach

Cons:

Worst-case time complexity O(N!)

May explore redundant paths

**Which is Better?**

For Small N (≤12):

The bitmask DP solution is better (faster, more efficient)

Constraints suggest N≤12 is expected (since 2^12 = 4096 is manageable)

For Larger N:

Backtracking is the only viable option

Can be optimized with memoization or pruning

Readability:

Backtracking is more intuitive for most developers

Bitmask DP is more "clever" but harder to maintain

**Recommendation**

Use the bitmask DP solution for:

Coding competitions (where N is small)

When you need optimal performance for small N

Use the backtracking solution for:

General programming interviews

When code readability is important

When N might be large

The bitmask solution is likely better for LeetCode's test cases since they're designed to work within the constraints, but both are valid approaches.