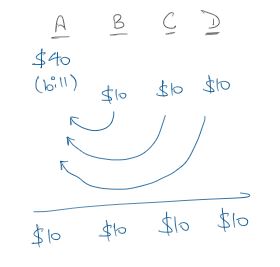
# Algorithms and Future Work

This file covers a few scenarios that we would like our web application to take care of.  
Some of the algorithms mentioned will be used for the demo on Wednesday, whereas the others are for future integration for other groups.

Various methods have been researched and the most appropriate ones have been included in this file. However, there may be other ways to achieve the following test cases in a more effective way. This document provides an initial read into how the application algorithms can be set up.

**Case 1:**

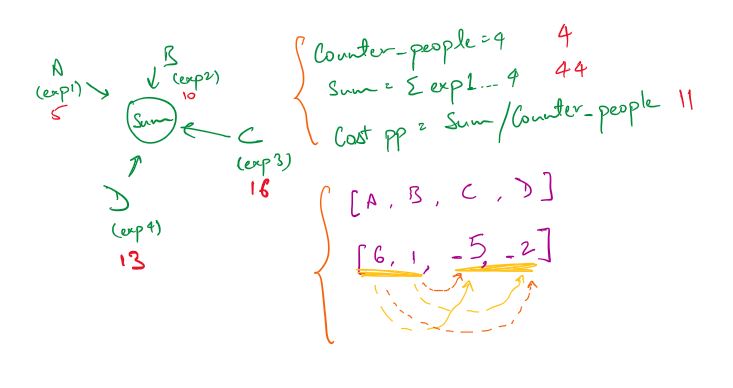
A single bill payment is made by an individual A on behalf of the group. The rest of the members now owe “A” a share of the bill. In our case, for simplicity, the shares will be equally divided.



Therefore, once the payments are made, everyone has had a net expenditure of $10 towards the bill (where 4 people had a total bill of $40).

**Case 2:**

Multiple transactions are made by various people in a group (for e.g. during a trip/vacation). Assuming all the bills must be settled evenly, we now have a scenario where a few people have paid more than they should have and a few people who haven’t paid enough. Therefore, we need to calculate the total amounts owed by the people who underpaid and who they need to pay to bring monetary equilibrium in the group.



**Case 3:**

Multiple transactions are made within a group but only between certain members. Therefore, we now don’t have an equal distribution of the expenditures as some people aren’t involved in certain activities. Below is an example describing this scenario.

Alice and Bill are going to lunch. Bill's card isn't working, so Alice pays $10 for his meal,.  
The next day, Bill and Charles meet each other on a railway station. Charles has no money for the ticket, so Bill buys him one for $5. Later that day, Alice borrows $5 from Charles and $1 from Bill to buy her friend a gift.

The transactions look like this:

Alice -> Bill $10

Bill -> Alice $1

Bill -> Charles $5

Charles -> Alice $5

To reach monetary equilibrium, Bill needs to give Alice $4 (he gave her $1 and Charlie indirectly paid his $5 to Alice already).

While this solution looks neat, it is complicated as the transactions can’t be predetermined and need to achieve the lowest number of payments to achieve equilibrium within the group. The following is a double entry accounting concept that could provide a basis for the solution to our problem.

The transactions could be structured as bookkeeping entries:

Alice Bill Charles Balance

Alice -> Bill $10 10 10- 0 0

Bill -> Alice $1 9 9- 0 0

Bill -> Charles $5 9 4- 5- 0

Charles -> Alice $5 4 4- 0 0

At each transaction, you credit one ledger account and debit another so that the balance is always zero. At the end, you simply work out the minimal number transactions to be applied to each account to return it to zero.

For this simple case, it's a simple $4 transfer from Bill to Alice. What you need to do is to reduce at least one account (but preferably two) to zero for every transaction added. Let's say you had the more complicated:

Alice Bill Charles Balance

Alice -> Bill $10 10 10- 0 0

Bill -> Alice $1 9 9- 0 0

Bill -> Charles $5 9 4- 5- 0

Charles -> Alice $5 4 4- 0 0

Charles -> Bill $1 4 5- 1 0

Then the transactions needed would be:

Bill -> Alice $4 0 1- 1 0

Bill -> Charles $1 0 0 0 0

Unfortunately, there are some states where this simple greedy strategy does not generate the best solution. More information can be found in the links provided in the reference list.

**Miscellaneous Information:**

1) A very similar problem is the Partition problem where you need to achieve zero states. The link to the problem is given in the Reference list.

2) Another good read is David Vavra’s “Mobile Application for Group Expenses and Its Deployment” (link in reference list). The document talks you through the methods used and how the application is deployed. There is a lengthy algorithm pertaining to how the author tackled the problem of splitting a bill towards the end of the document. The description of the algorithm is given below.

The algorithm solves the problem with n-1 transactions, but it's not optimal. From payments, the author computes the balance for each member. Balance is what the member paid minus what they should pay. Mr. Vavra sorts members according to balances increasingly. Then he always takes the poorest and richest and a transaction is made. At least one of them ends up with zero balance and is excluded from further calculations. With this, the number of transactions cannot be worse than n-1. It also minimizes amount of money in transactions. But it's not optimal, because it doesn't detect subgroups which can settle internally.

Finding subgroups which can settle internally is hard. The author describes this in his research paper, but I would recommend avoiding it as it’s very time consuming and can be done without.

3) “Minimize Cash Flow among a given set of friends who have borrowed money from each other” is the most wholesome solution which has the code given at the end of the document (website given in the reference list).

The following is detailed algorithm.

Do the following for every person Pi where i is from 0 to n-1.

**i)** Compute the net amount for every person. The net amount for person ‘i’ can be computed by subtracting sum of all debts from sum of all credits.

**ii)**Find the two persons that are the maximum creditor and maximum debtor. Let the maximum amount to be credited by maximum creditor be maxCredit and maximum amount to be debited from maximum debtor be maxDebit. Let the maximum debtor be Pd and maximum creditor be Pc.

**iii)** Find the minimum of maxDebit and maxCredit. Let minimum of two be x. Debit ‘x’ from Pd and credit this amount to Pc.

**iv)** If x is equal to maxCredit, then remove Pc from set of persons and recur for remaining (n-1) persons.  
  
**v)**If x is equal to maxDebit, then remove Pd from set of persons and recur for remaining (n-1) persons.

See Appendix for full code in C++.

**REFERENCES:**

*[Minimize Cash Flow among a given set of friends who have borrowed money from each other](https://www.citefast.com/?s=APA7" \o "Edit)*[. (2019, December 11). GeeksforGeeks. https://www.geeksforgeeks.org/minimize-cash-flow-among-given-set-friends-borrowed-money/](https://www.citefast.com/?s=APA7" \o "Edit)

*[Algorithm to share/settle expenses among a group](https://www.citefast.com/?s=APA7" \o "Edit)*[. (n.d.). Stack Overflow. https://stackoverflow.com/questions/974922/algorithm-to-share-settle-expenses-among-a-group](https://www.citefast.com/?s=APA7" \o "Edit)

David Vavra. (n.d.). *Mobile Application for Group Expenses and Its Deployment*. Retrieved from http://www.settleup.info/files/master-thesis-david-vavra.pdf

*Partition problem*. (2005, November 29). Wikipedia, the free encyclopedia. Retrieved March 16, 2020, from https://en.wikipedia.org/wiki/Partition\_problem

*[What algorithm to use to determine minimum number of actions required to get the system to "Zero" state?](What algorithm to use to determine minimum number of actions required to get the system to \"Zero\" state? (n.d.). Stack Overflow. https://stackoverflow.com/questions/877728/what-algorithm-to-use-to-determine-minimum-number-of-actions-required-to-get-the/   )*[(n.d.). Stack Overflow. https://stackoverflow.com/questions/877728/what-algorithm-to-use-to-determine-minimum-number-of-actions-required-to-get-the/](What algorithm to use to determine minimum number of actions required to get the system to \"Zero\" state? (n.d.). Stack Overflow. https://stackoverflow.com/questions/877728/what-algorithm-to-use-to-determine-minimum-number-of-actions-required-to-get-the/   )

**APPENDIX:**

// C++ program to fin maximum cash flow among a set of persons

#include<iostream>

using namespace std;

// Number of persons (or vertices in the graph)

#define N 3

// A utility function that returns index of minimum value in arr[]

int getMin(int arr[])

{

int minInd = 0;

for (int i=1; i<N; i++)

if (arr[i] < arr[minInd])

minInd = i;

return minInd;

}

// A utility function that returns index of maximum value in arr[]

int getMax(int arr[])

{

int maxInd = 0;

for (int i=1; i<N; i++)

if (arr[i] > arr[maxInd])

maxInd = i;

return maxInd;

}

// A utility function to return minimum of 2 values

int minOf2(int x, int y)

{

return (x<y)? x: y;

}

// amount[p] indicates the net amount to be credited/debited

// to/from person 'p'

// If amount[p] is positive, then i'th person will amount[i]

// If amount[p] is negative, then i'th person will give -amount[i]

void minCashFlowRec(int amount[])

{

// Find the indexes of minimum and maximum values in amount[]

// amount[mxCredit] indicates the maximum amount to be given

// (or credited) to any person .

// And amount[mxDebit] indicates the maximum amount to be taken

// (or debited) from any person.

// So if there is a positive value in amount[], then there must

// be a negative value

int mxCredit = getMax(amount), mxDebit = getMin(amount);

// If both amounts are 0, then all amounts are settled

if (amount[mxCredit] == 0 && amount[mxDebit] == 0)

return;

// Find the minimum of two amounts

int min = minOf2(-amount[mxDebit], amount[mxCredit]);

amount[mxCredit] -= min;

amount[mxDebit] += min;

// If minimum is the maximum amount to be

cout << "Person " << mxDebit << " pays " << min

<< " to " << "Person " << mxCredit << endl;

// Recur for the amount array. Note that it is guaranteed that

// the recursion would terminate as either amount[mxCredit]

// or amount[mxDebit] becomes 0

minCashFlowRec(amount);

}

// Given a set of persons as graph[] where graph[i][j] indicates

// the amount that person i needs to pay person j, this function

// finds and prints the minimum cash flow to settle all debts.

void minCashFlow(int graph[][N])

{

// Create an array amount[], initialize all value in it as 0.

int amount[N] = {0};

// Calculate the net amount to be paid to person 'p', and

// stores it in amount[p]. The value of amount[p] can be

// calculated by subtracting debts of 'p' from credits of 'p'

for (int p=0; p<N; p++)

for (int i=0; i<N; i++)

amount[p] += (graph[i][p] - graph[p][i]);

minCashFlowRec(amount);

}

// Driver program to test above function

int main()

{

// graph[i][j] indicates the amount that person i needs to

// pay person j

int graph[N][N] = { {0, 1000, 2000},

{0, 0, 5000},

{0, 0, 0},};

// Print the solution

minCashFlow(graph);

return 0;

}