Gaurav's fair share problem & Reduction by Debt Transfer Algorithm

Split shared expenses so the number of transactions are minimum and each member should remember the least number of transactions.

Maximum Transactions = `(n-1)` `WHERE` `n` = `number of members in group`.

Maximum Transactions one person should remember = 2 for any `n` where `n` = `number of members in group`.

Problem

In any group setting, be it with friends, roommates, or colleagues, splitting expenses often results in a tangled web of debts. Remembering who owes whom and how much can be a cognitive burden, leading to confusion and misunderstandings.

Do we really need an algorithm for that?

- Let's assume different scenarios:
  - Group of 2 (`A`, `B`) `WHERE` `A` is doing all the spending.
    - Just divide the expense by 2 and that's how much `B` owes to `A` (Max: One transaction)
  - Group of 2 (`A`, `B`) `WHERE` `A` & `B` both are spending.
    - Just calculate the total expense from each
    - Divide the expense of each by 2 and that's how much they owe to each other. (Max 2 transactions)
    - Subtract from who owes the least from the most. (Max: One transaction)
  - Group of 3 (`A`, `B`, `C`) `WHERE` any of them can be spending.
    - Calculate the total expense from each
    - Divide the expense of each by 3 and that's how much they owe to each other. (Max 6 transactions)
    - Subtract each other's payment and receipt. (Max: Three transaction)
  - Group of 4 or more?
    - This is going to be nasty quickly as complexity rises with number.

But what if in a group of three, the calculation can be done in such a way that there is only 2 transaction at most?

Solution

Reduction by Debt Transfer Algorithm takes each person's contributions within the group and calculates the most efficient way to settle debts. The goal is simple: reduce the number of transactions to minimum so the transactions each individual needs to keep track of will be just two(or one). One for giving and one for receiving.

This algorithm will allow you to split these costs so that there are at most `n-1` transactions where `n` is the number of members in the group.

Key Features

- **Linear Transactions:** Transforming the complex network of debts into a linear series, making it easy to follow and remember.

- **Fairness and Simplicity:** Striking a balance between fairness in splitting expenses and keeping the process straightforward for everyone.
- **Peace of mind:** Each participant only has to remember two transactions, minimizing the mental load and potential for confusion.

Use Cases

- **Shared Living Expenses:** Perfect for roommates sharing rent, utilities, and other common expenses.
- **Group Outings:** Ideal for outings, vacations, or events where participants contribute to shared costs.
- **Project Collaborations:** Streamline financial contributions in team projects, ensuring a fair and clear settlement process.

Proof of Concept

Problem Statement (We will call it `Gaurav's fair share problem`)

- `n` people has spent independently for a group. Split the expenses to settle the payments in such a way that there are no more than `n-1` transactions and ensure that each member has to remember only `2` transactions at most.

Let's assume we have a group of 5 (`A`, `B`, `C`, `D`, `E`) with given expenses:

Head	Cost	By
Some stuff	$560	A
Some other stuff	$278	E
Some good stuff	$1078	B
Some bad stuff	$28	C
Some useful stuff	$238	C
Some really good stuff	$2278	D

So in total:

Head	Cost	By
Total	$560	A
Total	$1078	B
Total	$266	C
Total	$2278	D
Total	$278	E

Let's sort the expenses in ascending order fist

```
Step 1: Sort the raw data based on expense [Ascending]
```

Head	Cost	By
Total	$266	C
Total	$278	E
Total	$560	A
Total	$1078	B
Total	$2278	D

Now, we can split those totals for each member and write them in the form of a matrix, the row being the payer and, the column being the receiver.

(Note that the diagonal values are the amount that is to be paid by oneself. Thus, it has no worth)

`Step 2: Form a per-member split matrix A`

S\R	C	E	A	B	D
C	53.20	53.20	53.20	53.20	53.20
E	55.60	55.60	55.60	55.60	55.60
A	112.00	112.00	112.00	112.00	112.00
B	215.60	215.60	215.60	215.60	215.60
D	455.60	455.60	455.60	455.60	455.60

This basically means

(Format: `Payer` ↬ `Receiver` [`Amount`])

- `C` ↬ `C` [`53.2`]
- `C` ↬ `E` [`53.2`]
- `C` ↬ `A` [`53.2`]
- `C` ↬ `B` [`53.2`]
- `C` ↬ `D` [`53.2`]
- `E` ↬ `C` [`55.6`]
- `E` ↬ `E` [`55.6`]
- `E` ↬ `A` [`55.6`]
- `E` ↬ `B` [`55.6`]
- `E` ↬ `D` [`55.6`]
- `A` ↬ `C` [`112`]
- `A` ↬ `E` [`112`]
- `A` ↬ `A` [`112`]
- `A` ↬ `B` [`112`]
- `A` ↬ `D` [`112`]
- `B` ↬ `C` [`215.6`]
- `B` ↬ `E` [`215.6`]
- `B` ↬ `A` [`215.6`]
- `B` ↬ `B` [`215.6`]
- `B` ↬ `D` [`215.6`]
- `D` ↬ `C` [`455.6`]
- `D` ↬ `E` [`455.6`]
- `D` ↬ `A` [`455.6`]
- `D` ↬ `B` [`455.6`]
- `D` ↬ `D` [`455.6`]

Now, as we can see there are redundant transactions

- Anyone can see that that if `A` has to pay `A` certain amount, it's not a debt. So it can be removed.
- It is obvious that if `A` has to pay `B` `$10` and `B` has to pay `A` `$4` rather than doing 2 transactions, we can just do one, that is `A` pays `B` `$6`.
- Let's do the same here.
- For position `(i, j)`
  - `(i, j) = Max(0, (i, j) - (j, i))`

- This can be done in matrix By `Max(0, A - A')`

`Step 3: Transpose the per-member split matrix A to create new matrix A' Step 4: Create a new matrix B such that B = A-A' Step 5: Create another reduction matrix C such that C = Max(0, B)`

Now we have a reduced matrix

S\R	C	E	A	B	D
C	0.00	0.00	0.00	0.00	0.00
E	2.40	0.00	0.00	0.00	0.00
A	58.80	56.40	0.00	0.00	0.00
B	162.40	160.00	103.60	0.00	0.00
D	402.40	400.00	343.60	240.00	0.00

This means

(Format: `Payer` ↬ `Receiver` [`Amount`])

- `E` ↬ `C` [`2.3999999999999986`]
- `A` ↬ `C` [`58.8`]
- `A` ↬ `E` [`56.4`]
- `B` ↬ `C` [`162.39999999999998`]
- `B` ↬ `E` [`160`]
- `B` ↬ `A` [`103.6`]
- `D` ↬ `C` [`402.40000000000003`]
- `D` ↬ `E` [`400`]
- `D` ↬ `A` [`343.6`]
- `D` ↬ `B` [`240.00000000000003`]

Now! We can see it's reduced a lot, but still not `n-1` right?

If you have a close look, there is nowhere to reduce as we did before. But That's where the magic kicks in!

Since we don't have the option to reduce directly, let's look at a scenario:

- `X` has to pay `Y` and `Z` `$10` and `$26` respectively, and `Y` has to pay `Z` `$ABC`,
  - In a broad sense, this scenario is similar to what we have, there are no direct reductions.
  - But, what if, instead of paying both `Y` and `Z` their respective amount, `X` says:
    - Hey `Y`, since you have to pay something to `Z` anyway,
      - I will not pay `Z`
      - Instead pay you `$10 + $26 = $36`
      - You keep what you owe from me. ie. `$10`
      - Then you pay `Z` `$26` on my behalf and `$ABC` from your own. ie `$10 + $ABC`

Looks like we found a new reduction method. (Let's call it `Reduction by Debt Transfer`)

But, there are not only 3 members, so how can we transfer the debt?

- Think of a queue where people are positioned based on number of people they have to pay to.
- The person who has to pay the most people (Not necessarily the amount) is placed first, and so on.
- So, the person on first can transfer their debt to the person on second by sending the total amount they have to pay.

- The person on the second will keep what the first person has to pay them, Add what they have to pay the person on back.
- And the second person can send that sum to the third and so on.

So, in our case

- The order is `C`, `E`, `A`, `B`, `D`
  - The amount `C`, have to pay `E` is the sum of all debt `C` has.
  - `E` Keeps what `C` owes `E`, Add what `E` owes to `A`. `B`, and `D` and pay that Sum to `A`.
  - This continues till the end and all the debt is settled.

The previous matrix shows how much one has to pay in the column and receive in the row, So, `Sum(row)-Sum(column)` is the transaction they need to make. - mean they need to pay, + means they need to receive.

`Step 6: Create new reduced 1*X transaction matrix D such as D[i] = Sum(Row[i]) - Sum(Col[i])` `Step 7: Every i in matrix Dx will pay i+1 X amount where X = (Di + Di-1)`

Finally

C	E	A	B	D
-626.00	-614.00	-332.00	186.00	1386.00

This means

(Format: `Payer` ↬ `Receiver` [`Amount`])

- `C` ↬ `E` [626]
- `E` ↬ `A` [1240]
- `A` ↬ `B` [1572]
- `B` ↬ `D` [1386]

In a nutshell, The algorithm

- **Step 1:** Sort the raw data based on expense [Ascending]
- **Step 2:** Form a per-member split matrix `A`
- **Step 3:** Transpose the per-member split matrix `A` to create new matrix `A'`
- **Step 4:** Create a new matrix `B` such that `B = A-A'`
- **Step 5:** Create another matrix `C` such that `C = Max(0, B)`
- **Step 6:** Create new reduced 1 row transaction matrix `D` such as `D[i] = Sum(Row[i]) - Sum(Col[i])`
- **Step 7:** Every `Dx` in matrix `D` will pay `Dx+1` `X` amount where `X = (Di + Di-1)`