Assignment 3.2 Practice Problem 2 (Split the Bill)

Problem:

In the Splitwise app, people form groups and add the expenses of members of the group. This is especially useful for vacations, where people traveling in a group can maintain an account of their expenses and who paid the bills.

All people in the group are assigned distinct IDs between 1 and N, where N is the size of the group.

In addition to keeping a record of the expenditure, Splitwise also calculates the list of shortest-path transfers (defined later) that will settle up all dues.

Each transaction has the following parameters:

- *transaction_id* - It is a string representing the unique ID by which the transaction is identified.
- *paid_by* - It is a list of lists, where each element of the list is another list having the form [x, y]. Here, x and y denote that person having ID x paid Rs. y.
- *split_as* - It is a list of lists, where each element of the list is another list having the form [x, y]. Here, x and y denote that after all dues are settled, a person having ID x will ultimately contribute Rs. y to the transaction.

For any given transaction, the following condition holds true:-

Total_Amt_Paid = Sum_of_all_splits

In other words, the sum total of all amounts in list paid_by equals the sum total of all amounts in list split_as.

Following is the example of a transaction in a group of size N=64:

- transaction_id : "#f1230"
- paid_by : [[1, 30], [4, 100], [63, 320]]
- split_as : [[1, 120], [2, 20], [3, 40], [4, 40], [37, 100], [51, 40], [53, 90]]

Shortest-Path Transfers: Shortest-path transfers lead to a reduction in the number of transfers.

Specifically, for a group having multiple transactions, the shortest-path transfers will be a list of payments to be made such that:

- Each payment can be represented by a list of the following form:- [payer_id, payee_id, amount]. There is only 1 payer, and 1 payee in each payment, which are distinct from each other. So, payer_id != payee_id, for any payment.
- Each person (out of the N people) can only either be the payer (in all payments involving him), or the payee, but not both.
- The total amount of money that each person should receive/spend, must be equal to the total amount he would receive/spend according to the given list of transactions.

Clearly, there can be several shortest-path transfers for a particular list of transactions.

Specifically, the **lexicographically smallest shortest path** has the following:

- Arrange people who have borrowed money in ascending order of their IDs. Do the same for people who have lent money.
- Now, construct payments so that the least borrower ID has to pay the least lender ID. Continue this process, till all debts have been settled.

Task

Given N members in a group, and lists representing the transactions(expenses), print the payments involved in the lexicographically smallest shortest-path transfers for the group.

Example

Input:

- N = 4
- 5 transactions, that can be represented as follows:
    - transaction_id = "#a1234", paid_by = [[1, 60]], split_as = [[2, 60]].
    - transaction_id = "#a2142", paid_by = [[2, 40]], split_as = [[3, 40]].
    - transaction_id = "#b3310", paid_by = [[3, 30]], split_as = [[4, 30]].
    - transaction_id = "#b2211", paid_by = [[4, 30]], split_as = [[3, 30]].
    - transaction_id = "#f1210", paid_by = [[3, 20]], split_as = [[1, 20]].

Output:

- 2 payments (of the form [payer_id, payee_id, amount]) are to be made, represented by the list:

- [[1, 2, 20], [1, 3, 20]]

Approach:

- The given list of payments satisfies all three necessary conditions. Hence, it is a Shortest-Path Transfer.

Function Description

Complete the function solve. This function takes the following 2 parameters and returns the required answer:

- N: An integer, representing the number of people in the group.
- transaction_list: A list (vector) of transactions. Each transaction is a dictionary, having keys “transaction_id”, “paid_by” and “split_as”. (The contents of each transaction are explained above)

Input format

Note: This is the input format that you must use to provide custom input (available above the Compile and Test button).

- The first line contains two space-separated integers N and M, the number of people in the group, and the number of transactions recorded.
- The next lines describe the M transactions as follows:
  - Each new transaction begins from a new line.
  - The first line of each transaction contains a string, representing the transaction_id of the transaction.
  - The 2nd line of each transaction contains 2 space-separated integers n_payers and n_splits. n_payers denotes the number of people in the paid_by list. n_splits denotes the number of people in the split_as list.
  - The next n_payers lines contain two space-separated integers, the payer and the amount paid.
  - The next n_splits lines contain two space-separated integers, the borrower and the amount borrowed.

Output format

Print the answer in the given format.

- In the first line, print a single integer K, denoting the number of payments involved in the Shortest Path Transfer.
- The next K lines should represent the K payments. Each payment should be printed in a single line as 3 space-separated integers payer_id, payee_id, and amount. Here, payer_id is the ID of the person who needs to pay the amount of money to the person with ID payee_id.

Constraints

- $2 \leq N \leq 2 * 10^5$
- $1 \leq M \leq 5000$
- $1 \leq len(transaction[i][paid_by]) + len(transaction[i][split_as]) \leq 50$
- $1 \leq total\_money\_exchanged\_in\_each\_transaction \leq 10^7$

Solution:

The task addresses the process of finding the **Shortest Path Transfer**, which is essential when trying to deal with Graphs. For this problem, we are not going to be dealing with graphs and will only focus on the Shortest Path problem.

Using tbe sample Input and Output provided:

Sample Input/Output

Input	Output
6 5 #itsmylife 2 3 1 25 3 15 4 10 5 25 6 5 #itsnow 1 4 4 100 1 25 2 25 3 25 4 25 #ornever 2 2 5 30 3 10 1 25 4 15 #iaintgonna 1 3 2 150 1 50 2 50 3 50 #liveforever 2 2 5 13 6 25 4 25 1 13	1 2 75 1 4 13 3 4 12 3 5 18 3 6 20

We can follow the Input and Output Format instructions properly. Since this data will be predetermined and will follow this format, it is efficient to use the *map()* and *split()* functions in order to store the input data.

```
6 5
#itsmylife
2 3
1 25
3 15
4 10
5 25
6 5
```

Since we are trying to access 2 separate integers entered as string inputs at the same time, we can use the map() and split() functions. We can split the input as a list of characters, then use the map function to convert each of them to an integer. Since we have just 2 integers per input line, we just need to assign it to 2 variables. The following code format should be able to handle the amount of input that we will be using:

```
n,m = map(int, input("Enter the group size and transaction count: ").split())
```

This is how we will be taking our input for the entire data set with just a few changes in the variables and input prompts. The entire input code should look something like this:

```
n,m = map(int, input("Enter the group size and transaction count: ").split())

for trans in range(m):
  s = input("Enter transaction ID: ")
  paid_by, split_as = map(int, input("Enter the number of payments and the number of splits: "))

  for pay in range(paid_by):
    pid, pay_amt = map(int, input("Enter the ID and the amount they paid: "))

  for split in range(split_as):
    sid, split_amt = map(int, input("Enter the ID and the amount they split: "))
```

This code completely handles the input side of the problem. However, we still need to write some code that would prepare the data we are accepting in order for it to be used in the solving part. In order to optimize the paying and spliting of the money, we need to track the final balances of each person in our group after the transactions. To do this, we need to create a **balance** list to hold the final balances of each

person after the transactions. We can simply use the **n** variable from the first input to generate this list. The list would contain all zeros, which is the initial balance of every person before the start of the transactions.

```
balance = []
for i in range(n):
  balance.append(0)
```

After this, we need to be able to change the balance of every person when we get an input for them in a transaction. We can do this by subtracting the amount to the corresponding paid_by person, and adding the amount for the split_by person. This is because the values in the balance list will represent how much a person still needs to play. If your balance is positive, it means you need to pay that amount. When balance is negative, it means that you will get paid.

```
for pay in range(paid_by):
    id, amt = map(int, input("Enter the ID and the amount they paid: "))
    balance[id-1] -= amt

  for split in range(split_as):
    id, amt = map(int, input("Enter the ID and the amount they split: "))
    balance[id-1] += amt
```

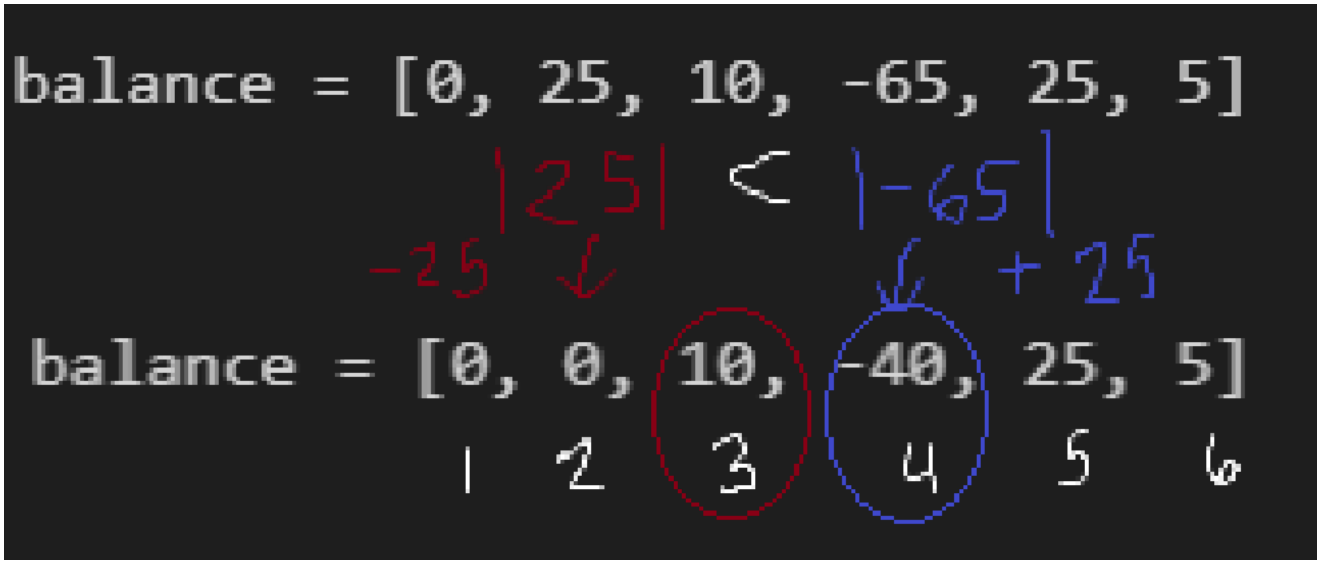
Now we have modified our code to a more complete form, allowing us to take the input data based on the given format and preparing it for processing in the next lines of code.

In solving this problem, what we are trying to do is to **minimize** the amount of transactions needed in order for all balances to be settled amount the group. Given a number of transactions, we need to be able to trace where all the money and debt ends up in order to just directly solve that summation of the balances. Given the following sample balance list:

```
balance = [0, 25, 10, -65, 25, 5]
```

In order to efficiently solve for the Shortest Path Transfer, it should be the lexicographically smallest shortest-path transfers for the entire group. This means we need to sort the balances by their IDs in ascending order. After this, we need to use the balance of the leftmost person that is greater than zero in order to pay the leftmost person with the balance that is less than zero. In order to understand this easier, we'll use the balance list from before:

The red oval shows the person with the smallest ID value that has a balance greater than zero. This means that they have a balance that they need to pay. The blue oval shows the person with the smallest ID value that has a balance less than zero. This means that they have paid for more than they should have and should be paid in return. Our goal now is to move the money around so that all balances will be back to 0 and all debts and deficits are settled. To do this, we just need to add the **positive balance (red, payer)** to the **negative balance (blue, payee)**. This ensures that there would always be a value that becomes zero, either the payer or the payee, depending on which absolute value is less than the other.



Once the lesser balance becomes 0, we can look again for the leftmost values for both positive or negative values. Once we do, we find that Person 3 will be our new payer and Person 4 is still our payee. We just need to repeat the process from earlier until all values become negative. Since we found a set of instructions that can perform this task for us, we can start formulating a pseudocode for our algorithm:

```
for each amount in balance:
  if amount is positive:
    save copy of amount

    while amount_copy greater than 0:
      iterate through balance until negative

      get lesser value between amount_copy and current balance iteration
      subtract lesser value from amount_copy
      add lesser value to balance iteration
```

The last thing we need is to record the transactions that we are making in order to pay off all debts and deficits in the balances. To do this, we need to follow the given output format:

```
1 2 75
```

According to the output format, the first digit to be printed should be the payer's ID, second number should be the payee's ID, then the last number is the total amount transferred during the transaction. Using our example from earlier, our output should be:

```
2 4 25
```

where 2 is the ID of the positive balance and 4 is the ID of the negative balance. We transferred 25 Rs from Person 2 to Person 4 and Person 2's debt was consolidated. This output can be saved every time we modify the balances in the balance list, utilizing an output list to store every transaction. Now that we have a plan, we can start constructing the code for our algorithm.

Python Implementation

Since we already have the code for the input part, we just need to code the algorithm which solves the problem, as well as the output part of our program. Pulling up the pseudocode from earlier:

```
for each amount in balance:
  if amount is positive:
    save copy of amount

    while amount_copy greater than 0:
      iterate through balance until negative

      get lesser value between amount_copy and current balance iteration
      subtract lesser value from amount_copy
```

```
    add lesser value to balance iteration
    save output details
```

From this pseudocode, we can see that we are trying to compare 2 separate values from the balance list. This means that we need to iterate twice. However, since we know that most likely, one and only one value will be zero, we only need to iterate through the second time when the first value is not yet zero. We only need to iterate when we still have a balance to fill and consolidate. This is why we will use a nested while loop instead of a nested for loop. Another thing we will consider is to use a list in order to save all outputs.

```
# This code takes in a transaction data set and returns the lexicographically smallest shortest-path transfers

# Transaction Input
n, m = map(int, input("Enter the group size and the transaction count: ").split()) # take no. of people and no. of transactions
balance = []
for i in range(n):
  balance.append(0) # generate balance list based on n

for trans in range(m): #generate each transaction
  trans_id = input("Enter transaction ID: ") #transaction id
  paid_by, split_as = map(int, input("Enter payment count and split count: ").split()) # take no. of payees, no. of payers

  for payment in range(paid_by): # generate each payment
    id, amt = map(int, input("Enter ID and Amount Paid: ").split()) # take payee id and amount
    balance[id - 1] -= amt # modify corresponding balance

  for split in range(split_as): # generate each split
    id, amt = map(int, input("Enter ID and Amount Split: ").split()) # take pater id and amount
    balance[id - 1] += amt # modify corresponding balance

# Solver Algorithm
transactions = [] # array for new transactions
iterator = 0 # iterator for nested while loop
for i in range(n): # iterate through each amount in balance
  if balance[i] > 0:
    payer_amount = balance[i] # save amount if positive

    while payer_amount > 0 and n > iterator: # iterate while saved positive amount not 0
      if balance[iterator] >= 0:
        iterator += 1 # ignore positive values, take negative value
        continue

      amount = min(payer_amount, abs(balance[iterator])) # compare payer and payee amounts, take smaller value

      payer_amount -= amount # subtract smaller value from payer balance
      balance[iterator] += amount # add smaller value to payee balance

      transactions.append([i+1, iterator+1, amount]) # record payer id, payee id, amount transferred

print("\nNumber of transactions: ", len(transactions)) # print no. of new transactions
for i in range(len(transactions)):
  print(f"Payer ID: {transactions[i][0]}, Payee ID: {transactions[i][1]}, Amount: {transactions[i][2]}") # print all new transactions
```

```
Enter the group size and the transaction count: 6 5
Enter transaction ID: #itsmylife
Enter payment count and split count: 2 3
Enter ID and Amount Paid: 1 25
Enter ID and Amount Paid: 3 15
Enter ID and Amount Split: 4 10
Enter ID and Amount Split: 5 25
Enter ID and Amount Split: 6 5
Enter transaction ID: #itsnow
Enter payment count and split count: 1 4
Enter ID and Amount Paid: 4 100
Enter ID and Amount Split: 1 25
Enter ID and Amount Split: 2 25
Enter ID and Amount Split: 3 25
Enter ID and Amount Split: 4 25
Enter transaction ID: #ornever
Enter payment count and split count: 2 2
Enter ID and Amount Paid: 5 30
Enter ID and Amount Paid: 3 10
Enter ID and Amount Split: 1 25
Enter ID and Amount Split: 4 15
Enter transaction ID: #iaintgonna
Enter payment count and split count: 1 3
Enter ID and Amount Paid: 2 150
Enter ID and Amount Split: 1 50
Enter ID and Amount Split: 2 50
```