A Thesis/Project/Dissertation Report

on

SHARE X -MOBILE APP

Submitted in partial fulfillment of the
requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

GALGOTIAS UNIVERSITY
(Established under Galgotias University Uttar Pradesh Act No. 14 of 2011)

Under The Supervision of
Dr. Pallavi Jain
Assistant Professor

Submitted By

Ashutosh Singh (19SCSE1010649)
Soumya Bhambani (19SCSE1050009)

.

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING /
DEPARTMENT OF COMPUTERAPPLICATION
GALGOTIAS UNIVERSITY, GREATER NOIDA
INDIA
DECEMBER, 2021

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
GALGOTIAS UNIVERSITY, GREATER NOIDA

CANDIDATE'S DECLARATION

I/We hereby certify that the work which is being presented in the thesis/project/dissertation, entitled **"SHARE X"** in partial fulfillment of the requirements for the award of the B.Tech Engineering submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of month, Year to Month and Year, under the supervision of Dr. Pallavi Jain (Assistant Professor), Department of Computer Science and Engineering/Computer Application and Information and Science, of School of Computing Science and Engineering , Galgotias University, Greater Noida

The matter presented in the thesis/project/dissertation has not been submitted by me/us for the award of any other degree of this or any other places.

Ashutosh Singh

Soumya Bhambani

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Pallavi Jain

Assistant Professor

CERTIFICATE

The Final Thesis/Project/ Dissertation Viva-Voce examination of Ashutosh Singh (19SCSE1010649) and Soumya Bhambani (19SCSE1050009) has been held on _____ and his/her work is recommended for the award of B. Tech Engineering.

Signature of Examiner(s) **Signature of Supervisor(s)**

Signature of Project Coordinator **Signature of Dean**

Date: December, 2021

Place: Greater Noida

Abstract

This Share x project is an app that is basically a logical and mathematical implementation over the financial expenses made by a group of people only with the help of raw inputs of each individual in that group.
App provide a general platform for a group of people to store the records of their expenses or financial contributions made by every individual in a group.
And by the end of the event the app will automatically calculate all the expenses and divide equally among the group. Well talking about the most interesting and tricky part that is it will also display the number of shares remaining by an individual to contribute in that event, and this will also decide that how much and to whom that individual has to give that money after the completion of the event.
This app reduces the time factor for every user of it. It will help the youth for planning and execution of a particular event in any financial situation.

Table of Contents

List of Table

List of Figures

Acronyms

B.Tech.	Bachelor of Technology
M.Tech.	Master of Technology
BCA	Bachelor of Computer Applications
MCA	Master of Computer Applications
B.Sc. (CS)	Bachelor of Science in Computer Science
M.Sc. (CS)	Master of Science in Computer Science
SCSE	School of Computing Science and Engineering

CHAPTER-1

Introduction

Share X will be a free app that allows consumers to split costs with friends. If a party needs to share the cost of a particular bill, Share X ensures that anyone who pays is reimbursed for the fair amount and minimum amount of services. Separately, users can send an email notification when credit is required, and the app allows users to send an IOU to someone in the group.

The app is marketed to track unregulated debts, including rent, dining and travel expenses, bills, and more. Share X users add notes to the app about who owes them, who owes them, and why. This service eliminates the need to keep receipts because the user can add any cost to the app when the cost is received.

You can use Share X using its mobile application on your phone. It works like a digital IOU, and easily calculates what each person in the group owes. You can also pay using third-party apps like Venmo with your Share X app.

1.2 FORMULATION OF PROBLEM

As the youth is getting creative day by day generation by generation, they trend to explore every single side of life for adventure, knowledge, to gain perspective and for fun basis.

So being in this era of life you could never afford to move with slow pace, neither this generation cope with slowness, having this aspect in mind our app will provide a platform for the people who seeks to have adventurous life and leave small tension behind like managing their budgets, transactions and partnership with their friend or family members.

This app namely SHARE X deals with multiple people's money having an event common in them.

For example, suppose a group of friends planned a tour for vacation, and they decided to invest in that tour with different ratio of money but some of them are not able to deposit money at the time the group needed, and that money issues managed by the remaining friends. And after the tour this app will provide information about the money invested by each of them in the group and how much money is overdue to whom. This app also helps in keeping records of transactions made by each group member and where for more accuracy.

Because of the endless monthly expenses and the lack of a track record of who was paying what and when (depending solely on memory), I began to realize that such a system led to inexplicable spending that was difficult to follow. Also, sometimes no one remembered whether he had included the costs or not! We had to browse through the endless list to see if we had it. Because even though I remember the bill I paid, Share X did not allow me to search for a category or time range, let alone Search at all.

1.2.1 TOOL AND TECHNOLOGY USED

App generally require following tools-

1. A programming language.

2. App studio for better interactive platform for users.
   A user-friendly app always a first choice of users.

3. General mathematical calculators for operations.
   For managing the records and perform calculations over data inputted by users.

4. A feedback platform for users.
   Feedback helps making the app better day by day.

Added new features are

- Adding new cab expenses
- Food ordering expenses
- Linked bank account
- Direct payment record to members

Basic concept used for simplifying

Simplify debt (a.k.a. “debt simplification”) is a Share X feature that redesigns debt between groups of people. It does not change the total amount that anyone owes, but it makes it easier to repay people by reducing the total amount of payments.

CHAPTER-2

PROJECT DESIGN

Basic objective while designing the app.

Tasks:
1. Set your currency type.
2. Create a weekend program group.
3. Pay for a movie and lunch. Keep your track.
4. How much has the team spent in total?
5. Find out how much you owe.
6. Looks like you have to pay a lot of people. Is there a way to combine payments in a group?
7. Send a reminder to your debtor friend.
8. Get organized
9. Delete group.

Analysis

Based on the above activities, I have updated user responses and comments.

1. Add credit
Most users do not change the date of the bill, so the bill is recorded on the date it was added.
Once the description of the bill is entered, the icon next to it changes when the app recognizes the words, for example: Grocery, Gift, Dinner, etc.
If the app does not know the meaning of the bill, the section icon says ‘General’, usually, which the user ignores.
Users can change the icon if they want to but, in most cases, they do not.
Summary: The user did not take into account the date or section of the thumbnail when adding the bill, which would have helped to set up bills in the future.

2. Debt planning and search
Debts are not categorized, as most users thought they would be (due to category icons displayed while adding credits). They are listed on a monthly basis. Users are not able to search for or edit credits based on time and / or category. And since our user did not set a date or category, the bill is kept with details that the user may not remember later, making the search even more difficult.
Summary: Share X does not allow you to filter credits based on a given category or time, which allows the user to check the endless list of credits.

3. Group details
When users open a Group Details page, the Group Cost list is displayed first. Clicking the 'Balance' button, displays information about group debts. When you tap 'View group' within the ellipses, it shows group and individual expenses.

Performance tests have given me a very important understanding that the Group's debts and expenses are hidden. The cost list is very focused. Users may prefer to see how much they spend, rather than looking at an endless list of bills. For anyone, managing their own expenses, group debts and expenses is very important.
Summary: User is not provided with important information, such as personal expenses, on the first visit.

4. Make Debts Easier
'Debt Lifting' is a very useful feature for Share X. It automatically covers all debts to reduce multiple payments between team members. However, it is within 'group settings' and not everyone knows this concept. So, unless the user is aware of this option, it remains unused.
Summary: Most users are not aware of this feature.

5. Sending reminders
The reminder is sent as an email to the registered email ID. During a user survey, it was revealed that not everyone checks their email. In some cases, the registered ID is an old email account, created long ago to receive social media notifications. So, there is a good chance the Reminders will not be read.
Summary: Reminders sent as emails remain unread.

6. Deleting the Group
When asked to delete a Group, users have long pressed on behalf of the Group waiting for the 'Remove' action. This is because of how users connect the task to be done with the action; remembering the 'Remove' behavior of other apps such as WhatsApp or Gmail.

Ideation

I reviewed the observations and identified pain points that hindered the effectiveness of the app - Adding a Bill, Credit Planning and Search and Grouping Information.
Asambe!
1. Group details
'View total groups and 'Balance' are no longer hidden. They are on the big screen as 'My Expenses' and 'team balances' respectively.
Making debt easier in advance.
The 'fix' action is no longer a separate button. It is shown only when the user owes money to someone (under Group balance).
The buttons, 'Adjust' and 'Balance', have been replaced by the 'Cost list' which leads the user to the cost list in that group.

2. Adding a bill

Adding costs is like writing notes. Everyone will have a different approach. Therefore, the chances of the same type of bill with different meanings are higher.
For example, I have incurred a cost called 'Grocery'. Next, my roommate put the costs down and named it ‘Milk, oil, etc.’, i.e., ‘Igrosa’.
Since debts can be arranged in any way, such a difference in naming can come in the form of smooth cost tracking.
I am of the opinion that if users do not have the motivation to do the job properly, they will not do it. So, if the app does not allow you to filter or search for credits, users will not make much effort while cutting costs (as they may think it is useless).
My way:
We often remember meeting together. One word can create context to make relevant pieces of information work. This is the concept of '# hashtag' that is easily integrated into our daily interactions.
Hashtag: A word or phrase preceded by the hash symbol (#), which is used on social networking websites and applications, to identify messages on a specific topic.
How will it help when you add a bill?
When you add a bill, a lot of thought is added while writing the definition of the bill.
Each word will be converted into a hashtag and costs will be marked on these terms.

Therefore, when searching for these costs, even if the user only remembers ‘cleaning’, they can search #cleaning and find all the costs marked in this name. Adding a #home home to search can filter results continuously.
It works exactly like twitter and Instagram but the difference here is that the meaning is translated into hashtags. The user does not need to select a list of categories while adding a bill. Everyone who knows the context of the bill - for example, their roommates will have the same view of the bill - will be able to easily search the bill. These hashtags are organized i.e., the user can add or remove tags generated by the application.
Splitwise has a few pre-defined categories like groceries, dinner, gift, movie, electricity, etc. issued when it sees the meaning. Thus, hashtags are ultimately a combination of user description and application categories.
Since we have a habit of typing hashtags everywhere, I find this method very accurate and close to the way we discuss making the act of remembering hashtags easier.

3. Search for credit
Tapping the 'Cost List' button on the Group information screen will lead the user to the list.
The user can search for specific credits by typing specific descriptive words in the search bar.
'Filters', in the upper right corner, help reduce search.
All tagged costs, typed in the search area, are full on screen.

4. Sending reminders
The user makes the default setting in the Account settings, choosing which reminder should be sent as email or SMS. A reminder message is a template provided by the app.

5. Friend information details screen
What's next? - Concept: Complete
This feature will allow for additional credit within the application. The user will be able to select the credits they want to add.
This option will only be available on the Cost list screen.
This is WIP, which is why it requires a lot of thought. This may translate into a completely new ‘reporting’ category.

CHAPTER-3

FUNCTIONALITY/ WORKING OF PROJECT

Create a cost-sharing request.

A cost-sharing application is where you can increase your costs and differentiate between different people. The app maintains a balance between people such as who owes money.

For example

He lives with 3 other friends.

You: User1 (id: u1)

Flat mates: User 2 (u2), User 3 (u3), User4 (u4)

This month's electricity bill was Rs. 1000.

Now you can just go to the app and add to pay 1000,

choose all 4 people and choose to divide equally.

Input: 1 1000 4 1 2 3 EQUALS

For this purchase, everyone owes 250 to User1.

The app should update the balances on each profile accordingly. User2 owes User 1: 250 (0 + 250)

User3 owes User 1: 250 (0 + 250)

User4 owes User 1: 250 (0 + 250)

Now, It's a BBD Auction on Flipkart and there's something on your card.

You are buying User2 and User3 for a few items as requested.

The total number of individuals is different.

Input: 1 1250 2 2 EXACT 370 880

For this purchase, User2 owes 370 to User1 and User3 owes 880 to User1.

The app should update the balances on each profile accordingly.

User2 owes User 1: 620 (250 + 370)

User3 owes User 1: 1130 (250 + 880)

User4 owes User 1: 250 (250 + 0)

Now, you go out with your partner and go with your brother / sister.

User4 pays and everyone shares equally. Debt 2 people.

Input: 4 1200 4 1 1 2 4 PERCENT 40 20 20 20

For this transaction, User1 owes 480 User4, User2 owes 240 User4 and User3 owes 240 User4.

The app should update the balances on each profile accordingly.

User1 owes User 4: 230 (250-480)

User2 owes User 1: 620 (620 + 0)

User2 owes User 4: 240 (0 + 240)

User3 owes User 1: 1130 (1130 + 0)

User3 owes User 4: 240 (0 + 240)

Requirements

User: Each user must have a user ID, name, email, mobile number.

Cost: It may be EQUAL, EXACT or PERCENT

Users can add any value, select any type of cost and share it with any available users.

The percentage and value given can be up to two decimal places.

In the case of percentages, you need to make sure that the total percentage of shares is 100 or not.

In the real situation, you need to make sure that the total amount of shares equals the total amount or not.

The application should be able to reflect the cost of one user and the balances of everyone.

If you are asked to display the balances, the app should show the balance of the user and all users where there is a non-zero balance.

The value should be shortened to two decimal places. Specifies that user1 paid 100 and the value is divided equally among 3 people. Give 33.34 to the first person and 33.33 to the others.

Input

You can create fewer users on your main route. No need to take it for granted.

There will be 3 types of inputs:

Cost per format: COSTS <user-id-of-person-who-paid> <no-of-users> <space-separated-list-of-users> <EQUAL / EXACT / PERCENT> <split values-space -who- ke-unequal>

Show all balances: SHOW

Show one user balance: SHOW <user-id>

For example: say, Anna, Bob, and Charlie live in an apartment. Anna owes Bob $ 20, and Bob owes Charlie $ 20. Instead of making two separate payments, Share X would tell Anna to pay Charlie $ 20 directly, thus reducing the total amount of payments made. This ensures that people get their money back quickly and efficiently.
Now let's try to extract an algorithm that may work under the hood of this Share X feature with the help of the example shown below.
Consider a group of seven people — Alice, Bob, Charlie, David, Emma, Fred, and Gabe. They went out to visit together and at the end of the trip they realized they had the following debts;
Gabe owes Bob $ 30.
Gabe owes David $ 10.
Fred owes Bob $ 10.
Fred owes Charlie $ 30.
Fred owes David $ 10.
Fred owes Ema $ 10.
Bob owes Charlie $ 40.
Charlie owes David $ 20.
David owes Ema $ 50.
To better understand, let's represent the information above in the form of a target graph, as shown below.

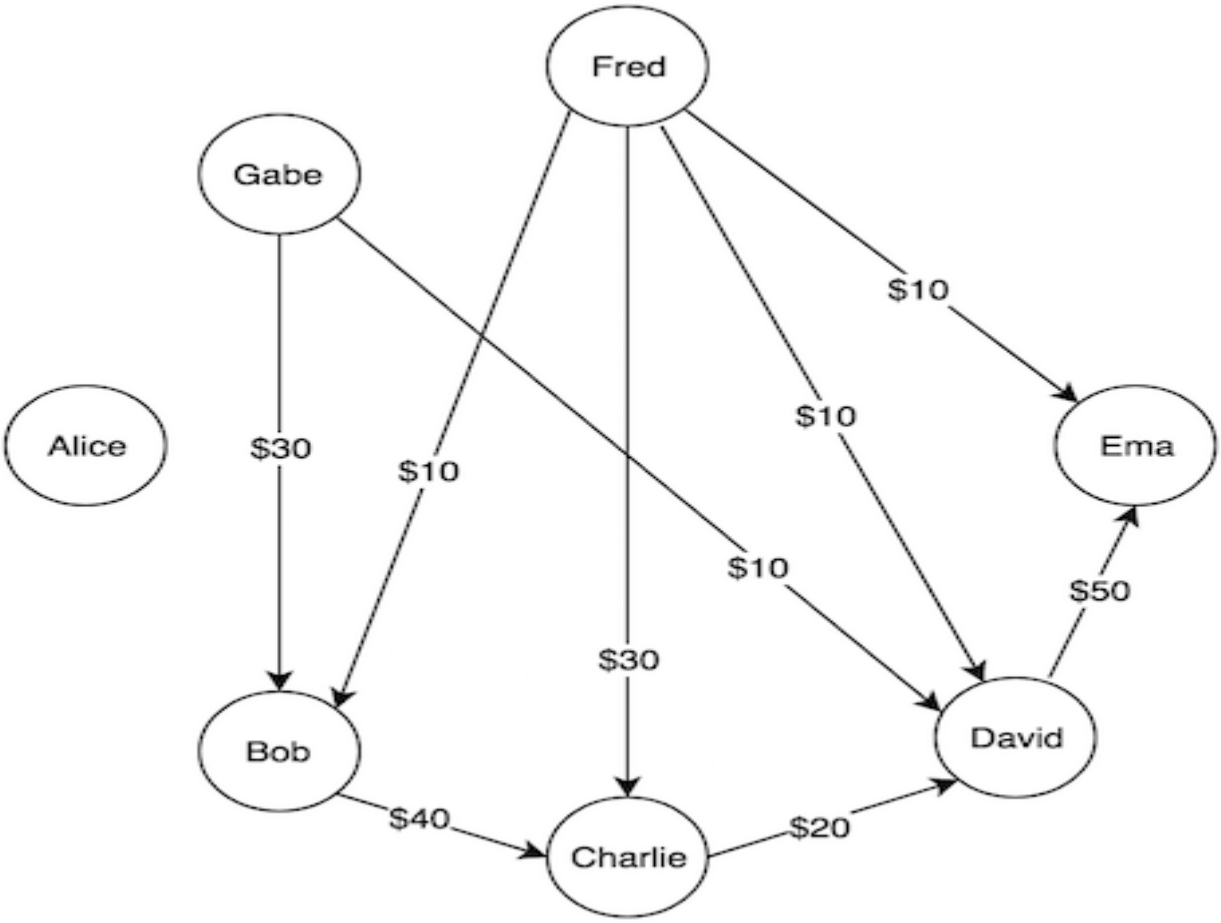

Figure 1. Representing Debts in the form of a Directed Graph

Now, since the purpose of the 'Make Debt Ease' feature is to reduce the total number of payments made to a group (as Share X suggests here), let's try to create an algorithm behind it. The first step is to get a complete change in the finances of each person in the group, which can be determined using the following formula,
Total Currency Change = (Cash Flow - Cash Flow)
For example, Charlie has a Net change of $ 50, calculated as shown below.
Total Currency Change (Charlie) = (Total Amount (Charlie) - Total Amount (Charlie))
= (($ 40 + $ 30) - ($ 20)) = $ 50.
So, Total Currency Exchange is $ 0 for Alice, $ 0 for Bob, $ 50 for Charlie, $ 10 for David, $ 60 for Emma, $ 60 for Fred and $ 40 in Gabe.
2. We now classify them into two types of people, namely donors (those with extra money, net Change in Money Money) and Recipients (those who need extra cash, with bad value. Total Currency Exchange). In the example above, Charlie and Ema are the hosts, while David, Fred, and Gabe are the hosts. Alice and Bob have their debts settled and that is why they are not Giver or Recipients.

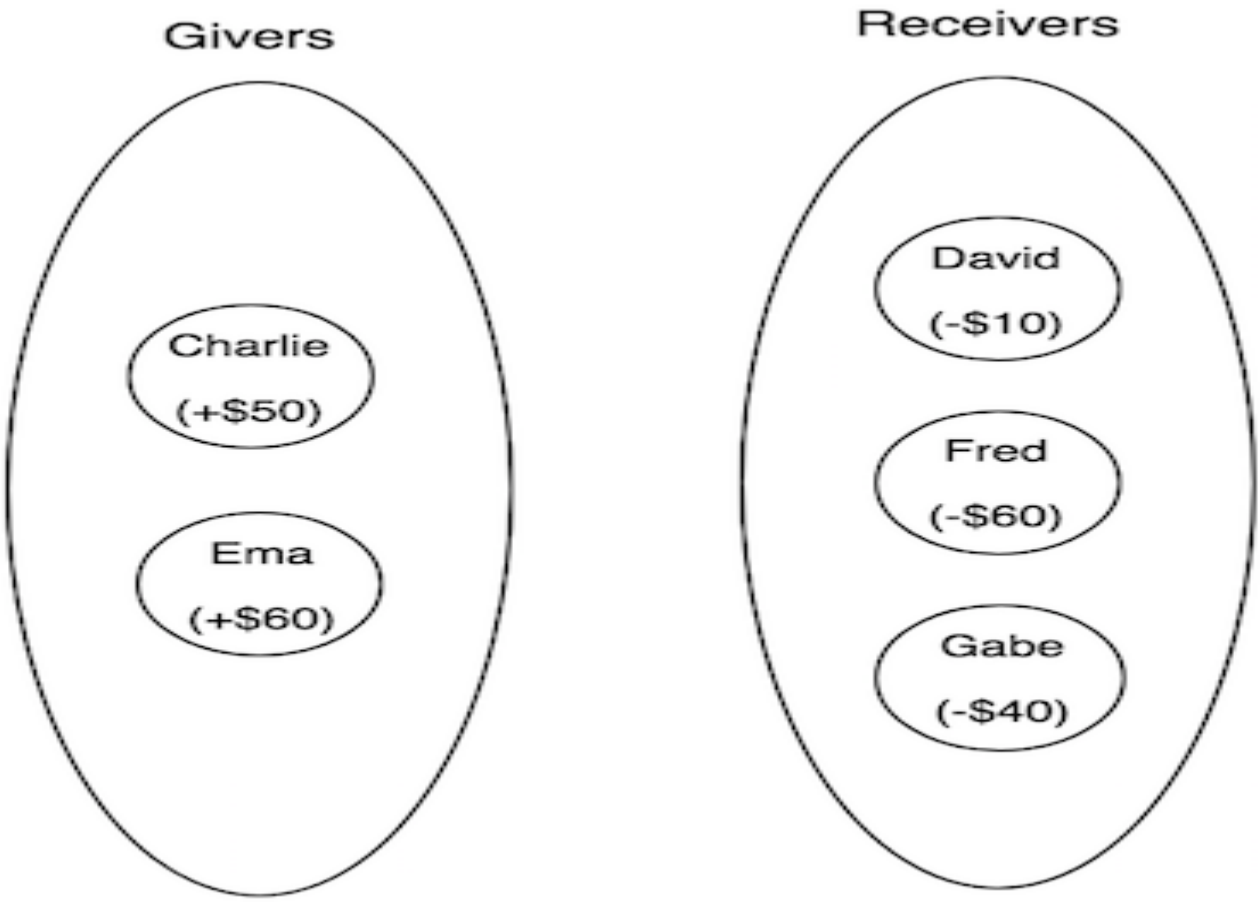

Figure 2. Clustering Givers and Receivers into two different groups

Why is this problem NP-Complete?

In order to reduce the number of payments that have to be made to settle debts, Donors should only transfer money to Recipients and Recipients should receive money from Donors only. Also, it can be noted that the total amount owed to Givers is always equal to the amount of money the Recipients will receive.
Since our goal is to reduce the total payments that need to be made, we must ensure that each Provider transfers money to a very small number of Receivables (As otherwise you will end up doing additional work, which will increase the total amount. Of transactions made). This means that in any given credit, we should always check the status of each donor who only does one job (for a particular recipient) to pay his / her debts (because then the total amount of services will be equal to the number of Contributors and would therefore be the appropriate solution). Therefore, in the appropriate case, each Provider will transfer the money to one recipient. This, in turn, means that every Recipient must receive all the money from the Donor or not receive any amount from him completely.
For example, let the G1, G2, G3 and G4 be the amount owed to four Providers respectively. Also, R1 and R2 represent the amount that the recipients will receive. Now, for any particular

Recipient, he or she can receive the entire G1 value from the original Provider or not accept the full amount. Similarly, he can accept the entire G2 total from the Second Giver or not receive any amount, and so on. This, in turn, means that we want a Subset of Givers that covers exactly the given amount of the Receiver and we need to do this for all Recipients. This is nothing but the sum of the Problems of the Subsides, except that here we may be given more than one amount (depending on the number of Recipients).
Now, we've been discussing it so far for a fair trial. It could also be the case that the best solution itself requires a subset of Givers to do more than one operation. If so, we need to do more than what we planned to do with the right case, i.e., examine all possible ways to separate the value from the Givers. This shows that the problem of debt relief is at least as complex as the sum of the problems of the subsets and therefore we are NP-Complete. Also, it means that not only is it a polynomial-time solution to this problem but it will also require a clear number of steps to reduce the total amount of payments. Now, this brings us to a very important question,
Can Share X use an exponential algorithm to solve the NP-Complete Problem in real time?
While I was pondering these thoughts, a colleague with whom I shared this information came back to me with a Share X page that mentioned 3 rules followed by a feature, listed below,
1. Everyone owes the same amount in the end,
2. No one owes a debt to a former debtor, either
3. No one owes more money than he did before making it easier.
Here, the first and third laws are something that should be obeyed automatically. But what is most interesting to us is the Second Commandment, which says 'No one owes a debt to a former debtor'.
The problem therefore lies in changing the transfer value of existing items without introducing new ones. This is an algorithmic translation of the following,
With the provided Credit Guaranteed graph (as shown in Figure 1), change (if necessary) the weights at the existing edges without introducing new ones.
Now the question is how do we do it. This can be solved if we divide the problem into two parts as shown below,

Will the existing edge (i.e., work) become part of the graph after simplifying debt?
If it is on the graph after simplifying the debt, what will be the weight (i.e., the value) of it?
The answer to the second question is to increase the debt (i.e., weight) on the edge, in order to eliminate the debts that flow in other directions between the same vertices. For example, let's look at Figure 1 again. Here, if Fred transfers $ 20 to Ema, then Fred will not pay David anything and David will only pay $ 40 to Ema, thus reducing the amount of money that will be made from 3 to 2 .
Now, in order to address the first question, namely to know that the edge will be part of the graph after simplifying the debts, we will try all the edges one by one.
How do we increase the weight on the existing edge?
Once we have selected the edge, we can increase its weight (i.e., credit) by using the Maximum-Flow Algorithm, which helps one determine the maximum flow between the source and the sink in a specific target graph.
Therefore, the problem-solving algorithm as described below,
Feed bills in the form of a targeted graph (not represented by G) in the algorithm.
Select one edge that can be visited, say (u, v) on the target graph G.

Now that you as a source and v as a sink, use the maxflow algorithm (which is probably Dinic's maxflow algorithm, as it is one of the appropriate resources) to determine the maximum possible flow of money from you to v.
Also, include the Residual Graph (we don't represent it in G'), which shows the possible additional flow of debt on the input graph after subtracting the debt flow between u and v.
If the maximum flow between u and v (let's represent the maximum flow) is greater than zero, then add the edge (u, v) as heavy as the maximum flow to the Remaining Graph.
Now go back to Step 1 and download Residual Graph G '.
Once all edges have been visited, the Residual Graph G 'obtained from the last duplication will be the one with the least number of edges (i.e., functions).

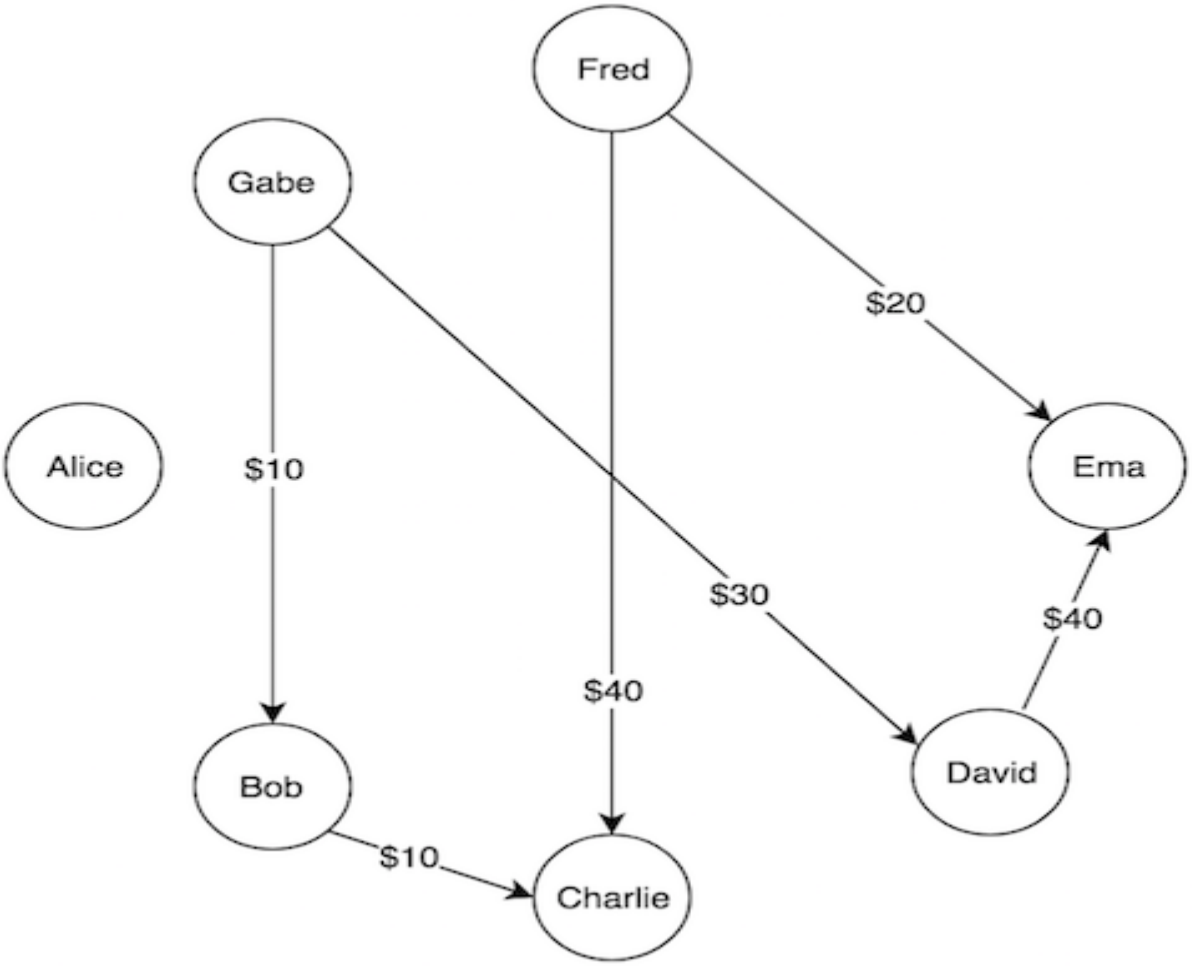

Fig-3 Simplified debts graph returned by the algorithm

It can therefore be seen that the algorithm reduces the number of tasks required to resolve all debts from 9 (shown in Figure 1) to 6 (shown in Figure 3) in the event that we are our role model.

The algorithm has a complex of O (V^2E^2), where V is the number of points on the graph and E is the number of edges on the graph. Here, the O (V^2E) complexity of the use of the High Flow Dinics and the additional O (E) is due to the repeating algorithm across all the ends of the graph. Also, there is another use of Maximum flow Problem with O (EV) complex, if we need to improve complexity. Now, once that is done, the complexity of the aforementioned algorithm will drop to O (E^2V), which will grow very well in the context of using the real world of Share X.

A SAMPLE CODING FOR THE PRESCRIBED APP IS GIVEN BELOW-

User.java

```
package com.workattech.splitwise.models;


public class User {
    private String id;
    private String name;
    private String email;
    private String phone;

    public User(String id, String name, String email, String phone) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.phone = phone;
    }

    public String getId() {
        return id;
    }
```

```
public void setId(String id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getPhone() {
    return phone;
}

public void setPhone(String phone) {
    this.phone = phone;
```

```
    }
}
```

Split.java

```
package com.workattech.splitwise.models.split;

import com.workattech.splitwise.models.User;

public abstract class Split {
    private User user;
    double amount;

    public Split(User user) {
        this.user = user;
    }

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public double getAmount() {
        return amount;
    }
```

```
    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

EqualSplit.java

```
package com.workattech.splitwise.models.split;

import com.workattech.splitwise.models.User;

public class EqualSplit extends Split {

    public EqualSplit(User user) {
        super(user);
    }
}
```

ExactSplit.java

```
package com.workattech.splitwise.models.split;

import com.workattech.splitwise.models.User;

public class ExactSplit extends Split {

    public ExactSplit(User user, double amount) {
        super(user);
```

```
        this.amount = amount;
    }
}
```

PercentSplit.java

```
package com.workattech.splitwise.models.split;

import com.workattech.splitwise.models.User;

public class PercentSplit extends Split {
    double percent;

    public PercentSplit(User user, double percent) {
        super(user);
        this.percent = percent;
    }

    public double getPercent() {
        return percent;
    }

    public void setPercent(double percent) {
        this.percent = percent;
    }
}
```

ExpenseMetadata.java

```
package com.workattech.splitwise.models.expense;

public class ExpenseMetadata {
    private String name;
    private String imgUrl;
    private String notes;

    public ExpenseMetadata(String name, String imgUrl, String notes) {
        this.name = name;
        this.imgUrl = imgUrl;
        this.notes = notes;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getImgUrl() {
        return imgUrl;
    }

    public void setImgUrl(String imgUrl) {
        this.imgUrl = imgUrl;
```

```
    }

    public String getNotes() {
        return notes;
    }

    public void setNotes(String notes) {
        this.notes = notes;
    }
}
```

Expense.java

```
package com.workattech.splitwise.models.expense;

import com.workattech.splitwise.models.User;
import com.workattech.splitwise.models.split.Split;

import java.util.List;

public abstract class Expense {
    private String id;
    private double amount;
    private User paidBy;
    private List<Split> splits;
    private ExpenseMetadata metadata;

    public Expense(double amount, User paidBy, List<Split> splits, ExpenseMetadata metadata) {
```

```
    this.amount = amount;
    this.paidBy = paidBy;
    this.splits = splits;
    this.metadata = metadata;
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public double getAmount() {
    return amount;
}

public void setAmount(double amount) {
    this.amount = amount;
}

public User getPaidBy() {
    return paidBy;
}

public void setPaidBy(User paidBy) {
```

```
        this.paidBy = paidBy;
    }

    public List<Split> getSplits() {
        return splits;
    }

    public void setSplits(List<Split> splits) {
        this.splits = splits;
    }

    public ExpenseMetadata getMetadata() {
        return metadata;
    }

    public void setMetadata(ExpenseMetadata metadata) {
        this.metadata = metadata;
    }

    public abstract boolean validate();
}
```

EqualExpense.java

```
package com.workattech.splitwise.models.expense;

import com.workattech.splitwise.models.User;
import com.workattech.splitwise.models.split.EqualSplit;
```

```
import com.workattech.splitwise.models.split.Split;


import java.util.List;


public class EqualExpense extends Expense {
    public EqualExpense(double amount, User paidBy, List<Split> splits, ExpenseMetadata expenseMetadata) {
        super(amount, paidBy, splits, expenseMetadata);
    }

    @Override
    public boolean validate() {
        for (Split split : getSplits()) {
            if (!(split instanceof EqualSplit)) {
                return false;
            }
        }

        return true;
    }
}
```

ExactExpense.java

```
package com.workattech.splitwise.models.expense;


import com.workattech.splitwise.models.User;
import com.workattech.splitwise.models.split.ExactSplit;
```

```
import com.workattech.splitwise.models.split.Split;

import java.util.List;

public class ExactExpense extends Expense {
    public ExactExpense(double amount, User paidBy, List<Split> splits, ExpenseMetadata expenseMetadata) {
        super(amount, paidBy, splits, expenseMetadata);
    }

    @Override
    public boolean validate() {
        for (Split split : getSplits()) {
            if (!(split instanceof ExactSplit)) {
                return false;
            }
        }

        double totalAmount = getAmount();
        double sumSplitAmount = 0;
        for (Split split : getSplits()) {
            ExactSplit exactSplit = (ExactSplit) split;
            sumSplitAmount += exactSplit.getAmount();
        }

        if (totalAmount != sumSplitAmount) {
            return false;
```

```
        }

        return true;
    }
}
```

PercentExpense.java

```
package com.workattech.splitwise.models.expense;

import com.workattech.splitwise.models.User;
import com.workattech.splitwise.models.split.PercentSplit;
import com.workattech.splitwise.models.split.Split;

import java.util.List;

public class PercentExpense extends Expense {
    public PercentExpense(double amount, User paidBy, List<Split> splits, ExpenseMetadata expenseMetadata) {
        super(amount, paidBy, splits, expenseMetadata);
    }

    @Override
    public boolean validate() {
        for (Split split : getSplits()) {
            if (!(split instanceof PercentSplit)) {
                return false;
            }
```

```
        }

        double totalPercent = 100;
        double sumSplitPercent = 0;
        for (Split split : getSplits()) {
            PercentSplit exactSplit = (PercentSplit) split;
            sumSplitPercent += exactSplit.getPercent();
        }

        if (totalPercent != sumSplitPercent) {
            return false;
        }

        return true;
    }
}
```

ExpenseType.java

```
package com.workattech.splitwise.models.expense;


public enum ExpenseType {
    EQUAL,
    EXACT,
    PERCENT
}
```

Now that I have created the models, let me create the services which will be used to talk to the models and to each other.

Here, I'm creating an ExpenseService which will be used to talk to 'Expense'. As of now, it is only being used to create an Expense object depending on the ExpenseType.

ExpenseService.java

```
package com.workattech.splitwise;


import com.workattech.splitwise.models.User;
import com.workattech.splitwise.models.expense.*;
import com.workattech.splitwise.models.split.PercentSplit;
import com.workattech.splitwise.models.split.Split;


import java.util.List;


public class ExpenseService {
    public static Expense createExpense(ExpenseType expenseType, double amount, User paidBy, List<Split> splits, ExpenseMetadata expenseMetadata) {
        switch (expenseType) {
            case EXACT:
                return new ExactExpense(amount, paidBy, splits, expenseMetadata);
            case PERCENT:
                for (Split split : splits) {
                    PercentSplit percentSplit = (PercentSplit) split;
                    split.setAmount((amount*percentSplit.getPercent())/100.0);
                }
                return new PercentExpense(amount, paidBy, splits, expenseMetadata);
            case EQUAL:
                int totalSplits = splits.size();
                double splitAmount = ((double) Math.round(amount*100/totalSplits))/100.0;
```

```
            for (Split split : splits) {
                split.setAmount(splitAmount);
            }
            splits.get(0).setAmount(splitAmount + (amount - splitAmount*totalSplits));
            return new EqualExpense(amount, paidBy, splits, expenseMetadata);
        default:
            return null;
    }
  }
}
```

Now, I'll create the orchestrator 'Expense Manager' which the 'Driver' class will talk to. Expense Manager will store the user and expense data.

ExpenseManager.java

```
package com.workattech.splitwise;

import com.workattech.splitwise.models.User;
import com.workattech.splitwise.models.expense.Expense;
import com.workattech.splitwise.models.expense.ExpenseMetadata;
import com.workattech.splitwise.models.expense.ExpenseType;
import com.workattech.splitwise.models.split.Split;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
public class ExpenseManager {
    List<Expense> expenses;
    Map<String, User> userMap;
    Map<String, Map<String, Double> > balanceSheet;

    public ExpenseManager() {
        expenses = new ArrayList<Expense>();
        userMap = new HashMap<String, User>();
        balanceSheet = new HashMap<String, Map<String, Double>>();
    }

    public void addUser(User user) {
        userMap.put(user.getId(), user);
        balanceSheet.put(user.getId(), new HashMap<String, Double>());
    }

    public void addExpense(ExpenseType expenseType, double amount, String paidBy,
List<Split> splits, ExpenseMetadata expenseMetadata) {
        Expense expense = ExpenseService.createExpense(expenseType, amount,
userMap.get(paidBy), splits, expenseMetadata);
        expenses.add(expense);
        for (Split split : expense.getSplits()) {
            String paidTo = split.getUser().getId();
            Map<String, Double> balances = balanceSheet.get(paidBy);
            if (!balances.containsKey(paidTo)) {
                balances.put(paidTo, 0.0);
            }
```

```
        balances.put(paidTo, balances.get(paidTo) + split.getAmount());

        balances = balanceSheet.get(paidTo);
        if (!balances.containsKey(paidBy)) {
            balances.put(paidBy, 0.0);
        }
        balances.put(paidBy, balances.get(paidBy) - split.getAmount());
    }
}

public void showBalance(String userId) {
    boolean isEmpty = true;
    for (Map.Entry<String, Double> userBalance : balanceSheet.get(userId).entrySet()) {
        if (userBalance.getValue() != 0) {
            isEmpty = false;
            printBalance(userId, userBalance.getKey(), userBalance.getValue());
        }
    }

    if (isEmpty) {
        System.out.println("No balances");
    }
}

public void showBalances() {
    boolean isEmpty = true;
    for (Map.Entry<String, Map<String, Double>> allBalances : balanceSheet.entrySet()) {
```

```
            for (Map.Entry<String, Double> userBalance : allBalances.getValue().entrySet()) {
                if (userBalance.getValue() > 0) {
                    isEmpty = false;
                    printBalance(allBalances.getKey(), userBalance.getKey(), userBalance.getValue());
                }
            }
        }

        if (isEmpty) {
            System.out.println("No balances");
        }
    }

    private void printBalance(String user1, String user2, double amount) {
        String user1Name = userMap.get(user1).getName();
        String user2Name = userMap.get(user2).getName();
        if (amount < 0) {
            System.out.println(user1Name + " owes " + user2Name + ": " + Math.abs(amount));
        } else if (amount > 0) {
            System.out.println(user2Name + " owes " + user1Name + ": " + Math.abs(amount));
        }
    }
}
```

Note: I could have used enums instead of different Expense classes for a simpler design but avoided as I wanted to keep the solution language agnostic. If you're solving this in Java, you may try solving it using enums. I could also have used interfaces at certain places but I avoided it here for simplicity.

Now, that our core design is done, let's create the Driver class where we would take user input to add expenses to the ExpenseManager and to show balances as well.

Driver.java

```
package com.workattech.splitwise;


import com.workattech.splitwise.models.User;
import com.workattech.splitwise.models.expense.ExpenseType;
import com.workattech.splitwise.models.split.EqualSplit;
import com.workattech.splitwise.models.split.ExactSplit;
import com.workattech.splitwise.models.split.PercentSplit;
import com.workattech.splitwise.models.split.Split;


import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;


public class Driver {
    public static void main(String[] args) {
        ExpenseManager expenseManager = new ExpenseManager();


        expenseManager.addUser(new User("u1", "User1", "gaurav@workat.tech",
"9876543210"));
        expenseManager.addUser(new User("u2", "User2", "sagar@workat.tech", "9876543210"));
        expenseManager.addUser(new User("u3", "User3", "hi@workat.tech", "9876543210"));
        expenseManager.addUser(new User("u4", "User4", "mock-interviews@workat.tech",
"9876543210"));
```

```
        Scanner scanner = new Scanner(System.in);
        while (true) {
            String command = scanner.nextLine();
            String[] commands = command.split(" ");
            String commandType = commands[0];

            switch (commandType) {
                case "SHOW":
                    if (commands.length == 1) {
                        expenseManager.showBalances();
                    } else {
                        expenseManager.showBalance(commands[1]);
                    }
                    break;
                case "EXPENSE":
                    String paidBy = commands[1];
                    Double amount = Double.parseDouble(commands[2]);
                    int noOfUsers = Integer.parseInt(commands[3]);
                    String expenseType = commands[4 + noOfUsers];
                    List<Split> splits = new ArrayList<>();
                    switch (expenseType) {
                        case "EQUAL":
                            for (int i = 0; i < noOfUsers; i++) {
                                splits.add(new EqualSplit(expenseManager.userMap.get(commands[4 +
i])));
                            }
```

```
                    expenseManager.addExpense(ExpenseType.EQUAL, amount, paidBy, splits,
null);
                    break;
                case "EXACT":
                    for (int i = 0; i < noOfUsers; i++) {
                        splits.add(new ExactSplit(expenseManager.userMap.get(commands[4 + i]),
Double.parseDouble(commands[5 + noOfUsers + i])));
                    }
                    expenseManager.addExpense(ExpenseType.EXACT, amount, paidBy, splits,
null);
                    break;
                case "PERCENT":
                    for (int i = 0; i < noOfUsers; i++) {
                        splits.add(new PercentSplit(expenseManager.userMap.get(commands[4 +
i]), Double.parseDouble(commands[5 + noOfUsers + i])));
                    }
                    expenseManager.addExpense(ExpenseType.PERCENT, amount, paidBy,
splits, null);
                    break;
            }
            break;
        }
    }
}
```

CHAPTER-4
RESULT AND DISCUSSION

Share X provides this Self-Serve API to facilitate integration with third-party applications, as well as open functionality for hobbies and power users to interact with their Share X account and build plugins or other tools.

If you would like to integrate your marketing application with Share X, we strongly encourage you to contact developer so that our developer team can assist you in discussing your application, provide confidential APIs and Enterprise support, and provide a valid commercial license for integration. The Self-Serve API listed here may be suitable for internal image processing and other testing tasks.

If you are developing a non-commercial plugin application or personal project, we recommend that you use the Self-Serve API listed under the API Terms of Use. Please note that our Self-Serve API has a maintenance level and access restrictions, which may change at any time and may not be well suited to commercial projects. If this is a problem for your application status, please contact us to discuss your needs.

All Self-Service API users are subject to the API Terms of Use below.

CHAPTER-5
CONCLUSION AND FUTURE SCOPE

CONCLUSION

Share X is a well-designed and well-used app, which solves a major problem of keeping track of expenses among friends. The paid subscription of the app also adds a few other features such as, search costs, performance, etc. Making a few changes can make the in-depth information about an already rich user of the app less restrictive.

FUTURE SCOPE

- Can be used on major organising events.
- Role of bank accounts can be increased in payments records sections.
- Variety of new sections for expenses could be added.

CHAPTER-6
REFERENCES

- http://ixd.prattsi.org/2020/02/design-critique-splitwise-android-app/
- https://workat.tech/machine-coding/editorial/how-to-design-splitwise-machine-coding-ayvnfo1tfst6
- https://medium.com/@mithunmk93/algorithm-behind-splitwises-debt-simplification-feature-8ac485e97688
- https://uxdesign.cc/splitwise-a-ux-case-study-dc2581971226