Coding

Fractional Shares

"""
On our journey to democratize finance for all, we've created the concept of fractional shares. Fractional shares are pieces, or fractions, of whole shares of a company or ETF.

However, exchanges only trade in whole shares. Robinhood is required to manage the fractional portion of each trade.

If Robinhood has 0 shares of AAPL and then a customer wishes to purchase 1.5 (150) AAPL shares, Robinhood will need to request 2 (200) shares from the exchange and hold on to the remaining 0.5 (50) shares.
If another customer requests to purchase 0.4 shares of AAPL, Robinhood can use its inventory (0.5 shares) instead of going out to the exchange and will have 0.1 shares of AAPL remaining.
If the third customer requests 0.5 shares, Robinhood can fill 0.1 shares out of inventory but will need to go to the exchange for an additional share leaving Robinhood's inventory at 0.6 shares.
If a customer requests a dollar based order, we need to convert it to the relevant number of shares and run through the above steps.
Always ensure the firm has a positive quantity in inventory and has under one share after handling an order. There's no need for us to hold onto whole shares!
Steps:

Handle buying fractional shares.
Handle selling fractional shares.
Ensure inventory is less than 1 after each order.
e.g. Customer sells AAPL for 0.75 and then another sells AAPL for 0.50 -- we have 1.25 inventory. We can sell 1 share to the market and keep our inventory small at 0.25.
Ensure inventory is always non-negative after each order.
e.g. Inventory is 0.2 and the customer buys 0.5 shares: ensure we end up with 0.7 shares in inventory.
Always "flatten"! (steps 3+4)
The final 2 digits of every integer is the decimal. e.g. 1000 = 10.00, 20 = 0.20, 100 = 1.

Example scenario:

Input:
// One AAPL buy order for 0.42 shares. AAPL is currently worth $1.
orders: [["AAPL","B","42","100"]]

// Inventory for AAPL is currently 0.99 shares.

inventory: [["AAPL","99"]]

Expected Output:
// The users buys 0.42 shares from inventory, leaving us with 0.57 shares.
[["AAPL","57"]]
Another example scenario:

Input:
// One AAPL buy order for $0.42. AAPL is currently worth $1, so that's 0.42 shares.
orders: [["AAPL","B","$42","100"]]
// Existing AAPL inventory is 0.50 shares.
inventory: [["AAPL","50"]]

Expected Output:
// 0.50 - 0.42 = 0.08 shares leftover.
[["AAPL","8"]]

[execution time limit] 3 seconds (java)

[memory limit] 1 GB

[input] array.array.string orders

A list of orders in the format of [$SYMBOL, $BUY_OR_SELL, $QUANTITY, $CURRENT_PRICE]. Each parameter is a string.

$SYMBOL: Can be "AAPL", "GOOGL", "MEOOOOOW" or anything really.
$BUY_OR_SELL: "B" or "S". B for BUY, S for SELL.
$QUANTITY: Can be a number or a dollar amount (prefixed with $). e.g. "100" for 1 quantity or "$150" for $1.50.
$CURRENT_PRICE: Current price of the symbol with no $ sign. e.g. "1000" for $10.

** All numbers are multiplied by 100 to store two significant digits. e.g. 100 = 1.00, 150 = 1.50, 1025 = 10.25 **

[input] array.array.string inventory

Inventory is a list of the inventory of each symbol. Each element in the list a 2 item list of [$SYMBOL, $QUANTITY] (remember quantity is multiplied by 100!).

An example for AAPL of 0.50 shares and GOOGL of 0.75 shares would be:

[["AAPL","50"],

 ["GOOG","75"]]
[output] array.array.string

The output is the final inventory of each symbol after iterating through each trade. This is expected to be in the same order and format as the inventory parameter.

e.g.

["AAPL","58"],
 ["GOOG","50"]]
"""

Offset Ordering

Suppose we are consuming a list of messages from an ordered stream. Each message is represented by its offset, which denotes its order in the stream.

For example, a 4-message stream may look like:

[offset = 0][offset = 1][offset = 2][offset = 3]

However, while messages may arrive in order, we might not necessarily process them in order.

For example, imagine we are processing messages in a multi-threaded environment, and thus we are at the whim of the scheduler. We may process messages in the order [offset = 3][offset = 0][offset = 1][offset = 2], for instance.

When we're finished with a message (i.e. an offset), we need to tell the stream this, so it knows not to send it again. This is called committing an offset.

To "commit an offset" means that we're done with every message up to that offset. In other words, committing an offset of 2 means "I'm done with messages with offsets 0, 1, and 2". That means we can ONLY commit to 2 if we're ALSO done with 0, 1, and 2.

Problem statement

Given a list of offsets, ordered by when they are processed, return a list of offsets that represent the greediest order of commits. That is, when an offset CAN be committed, we MUST commit it.

We can commit an offset X when EITHER:

- X = 0, since it is the first message of the stream
- All offsets < X are either ready to be committed or are already committed
- If we cannot commit offset X, we represent this as -1.

Example 1:

Input: [2, 0, 1]
Output: [-1, 0, 2]

We iterate through each message from left to right:

1). We try to commit 2, but we CANNOT because all previous offsets (0, 1) have not been committed yet. Thus, we append -1 in our result list, which represents NO commit on this offset. It might help to visualize this state as something like:

```
                (ready to be committed 2)
                  ----------


[offset = 0][offset = 1][offset = 2]
```

2). We try to commit 0, and we CAN because it's the first message of the stream. We commit the offset 0. Thus, we append 0 to our result list.

```
(committed 0)
 xxxxxxxxxx ----------
 [offset = 0][offset = 1][offset = 2]
```

3). We try to commit 1, and we CAN because all messages up to 1 have been committed. We commit the offset 2. Thus, we append 2 to our result list.

```
                (committed 2)


xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx
 [offset = 0][offset = 1][offset = 2]
```

Thus, we output [-1, 0, 2]. Remember, 1 is NOT in the output because the commit of offset 2 encapsulates it.

Example 2

Input: [0, 1, 2]
Output: [0, 1, 2]

We can commit each message as we iterate because each successive offset is the lowest offset we can possibly commit.

Example 3

Input: [2, 1, 0, 5, 4]
Output: [-1, -1, 2, -1, -1]

We do NOT commit 4 and 5. Had we received 3, we would have committed 4 and 5.

Important things to remember

- Assume a clean "state of the world" for every function call, i.e. no offsets have been committed thus far (so we must always start at 0).
- Every offset is >= 0.
- We never have any duplicate offsets.

Margin Calls

/*
Our goal is to build a simplified version of a real Robinhood system that reads a customer's trades from a stream, maintains what they own, and rectifies running out of cash (through a process called a "margin call", which we'll define later). We're looking for clean code, good naming, testing, etc. We're not particularly looking for the most performant solution.

Step 1 (tests 1-4): Parse trades and build a customer portfolio

Your input will be a list of trades, each of which is itself a list of strings in the form [timestamp, symbol, B/S (for buy/sell), quantity, price], e.g.

[["1", "AAPL", "B", "10", "10"], ["3", "GOOG", "B", "20", "5"], ["10", "AAPL", "S", "5", "15"]]

is equivalent to buying 10 shares (i.e. units) of AAPL for 5 each at timestamp 3, and selling 5 shares of AAPL for $15 at timestamp 10.

Input assumptions:

- The input is sorted by timestamp
- All numerical values are nonnegative integers
- Trades will always be valid (i.e. a customer will never sell more of a stock than they own).

From the provided list of trades, our goal is to maintain the customer's resulting portfolio (meaning everything they own), **assuming they begin with $1000**. For instance, in the above example, the customer would end up with $875, 5 shares of AAPL, and 20 shares of GOOG. You should return a list representing this portfolio, formatting each individual position as a list of strings in the form [symbol, quantity], using 'CASH' as the symbol for cash and sorting the remaining stocks alphabetically based on symbol. For instance, the above portfolio would be represented as

[["CASH", "875"], ["AAPL", "5"], ["GOOG", "20"]]

Step 2 (tests 5-7): Margin calls

If the customer ever ends up with a negative amount of cash **after a buy**, they then enter a process known as a **margin call** to correct the situation. In this process, we forcefully sell stocks in the customer's portfolio (sometimes including the shares we just bought) until their cash becomes non-negative again.

We sell shares from the most expensive to least expensive shares (based on each symbol's most-recently-traded price) with ties broken by preferring the alphabetically earliest symbol. Assume we're able to sell any number of shares in a symbol at that symbol's most-recently-traded price.

For example, for this input:

```
[["1", "AAPL", "B", "10", "100"],
["2", "AAPL", "S", "2", "80"],
["3", "GOOG", "B", "15", "20"]]
```

The customer would be left with 8 AAPL shares, 15 GOOG shares, and 80 a share) to cover the deficit. Afterwards, they would have 6 shares of AAPL, 15 shares of GOOG, and a cash balance of $20.

The expected output would be

[["CASH", "20"], ["AAPL", "6"], ["GOOG", "15"]]

Step 3/Extension 1 (tests 8-10): Collateral

Certain stocks have special classifications, and require the customer to also own another "collateral" stock, meaning it cannot be sold during the margin call process. Our goal is to handle a simplified version of this phenomenon.

Formally, we'll consider stocks with symbols ending in "O" to be special, with the remainder of the symbol identifying its collateral stock. For example, AAPLO is special, and its collateral stock is AAPL. **At all times**, the customer must hold at least as many shares of the collateral stock as they do the special stock; e.g. they must own at least as many shares of AAPL as they do of AAPLO.

As a result, the margin call process will now sell the most valuable **non-collateral** share until the balance is positive again. Note that if this sells a special stock, some of the collateral stock may be freed up to be sold.

For example, if the customer purchases 5 shares of AAPL for 75 each, then finally 5 shares of AAPLO for 125, but their shares of AAPL can no longer be used to cover the deficit (since they've become collateral for AAPLO). As a result, 2 shares of GOOG would be sold back (again at 25, 5 AAPL, 5 AAPLO, and 3 GOOG. Thus, with an input of

[["1", "AAPL", "B", "5", "100"], ["2", "GOOG", "B", "5", "75"], ["3", "AAPLO", "B", "5", "50"]]

the corresponding output would be

[["CASH", "25"], ["AAPL", "5"], ["AAPLO", "5"], ["GOOG", "3"]
*/

```
// class User {
//   private:
//     // Skip user info fields
//     // ...
//     vector<vector<string>> records;

//   public:
//     User(vector<vector<string>>& records) : this->records(records);
// };
```

House and Street

// A trade is defined as a fixed-width string containing the 4 properties given below separated by commas:

// Symbol (4 alphabetical characters, left-padded by spaces)
// Type (either "B" or "S" for buy or sell)
// Quantity (4 digits, left-padded by zeros)
// ID (6 alphanumeric characters)
// e.g.

// "AAPL,B,0100,ABC123"

// which represents a trade of a buy of 100 shares of AAPL with ID "ABC123"

// Given two lists of trades - called "house" and "street" trades, write code to filter out groups of matches between trades and return a list of unmatched house and street trades sorted alphabetically. There are many ways to match trades, the first and most important way is an exact match (Tests 1-5):

// An exact match is a house_trade+street_trade pair with identical symbol, type, quantity, and ID
// Note: Trades are distinct but not unique

// For example, given the following input:

```
// // house_trades:
// [
// "AAPL,B,0080,ABC123",
// "AAPL,B,0050,ABC123",
// "GOOG,S,0050,CDC333"
// ]

// // street_trades:
// [
// " FB,B,0100,GBGGGG",
// "AAPL,B,0100,ABC123"
// ]
```

// We would expect the following output:

```
// [
// " FB,B,0100,GBGGGG",
// "AAPL,B,0100,ABC123",
// "GOOG,S,0050,CDC333"
// ]
```

// Because the first (or second) house trade and second street trade form an exact match, leaving behind three unmatched trades.

// Follow-up 1 (Test 6,7,8,9): A "fuzzy" match is a house_trade+street_trade pair with identical symbol, type, and quantity ignoring ID. Prioritize exact matches over fuzzy matches. Prioritize matching the earliest alphabetical house trade with the earliest alphabetical street trade in case of ties.

// Follow-up 2: (Test 10) An offsetting match is a house_trade+house_trade or street_trade+street_trade pair where the symbol and quantity of both trades are the same, but the type is different (one is a buy and one is a sell). Prioritize exact and fuzzy matches over offsetting matches. Prioritize matching the earliest alphabetical buy with the earliest alphabetical sell.

Candlestick Charts

Our goal is to build a simplified version of a real Robinhood system that reads prices from a stream and aggregates those prices into historical datapoints aka candlestick charts. We're looking for clean code, good naming, testing, etc.

Step 1: Parse Prices

Your input will be a comma-separated string of prices and timestamps in the format price:timestamp e.g.

1:0,3:10,2:12,4:19,5:35 is equivalent to

price: 1, timestamp: 0
price: 3, timestamp: 10
price: 2, timestamp: 12
price: 4, timestamp: 19
price: 5, timestamp: 35

You can assume the input is sorted by timestamp and values are non-negative integers.

Step 2: Aggregate Historical Data from Prices

We calculate historical data across fixed time intervals. In this case, we're interested in intervals of 10, so the first interval will be [0, 10). For each interval, you'll build a datapoint with the following values.

Start time
First price
Last price
Max price
Min price

Important: If an interval has no prices, use the previous datapoint's last price for all prices. If there are no prices and no previous datapoints, skip the interval.

You should return a string formatted as {start,first,last,max,min}. For the prices shown above, the expected datapoints are

{0,1,1,1,1}{10,3,4,4,2}{20,4,4,4,4}{30,5,5,5,5}

[execution time limit] 3 seconds (cs)

[input] string prices_to_parse

[output] string

[C#] Syntax Tips

```
// Prints help message to the console
// Returns a string
string helloWorld(string name) {
Console.Write("This prints to the console when you Run Tests");
return "Hello, " + name;
}
```

C#
Mono v6.12.0.122
14151611121391067845

```
}
else
{
if (i > 0 && list[i-1] != null)
{
AggregatePrice aggPrice = new AggregatePrice(list[i-1].lastPrice, i10);
str.Append("{");
str.Append(GetAggregatedPrice(i10, aggPrice));
str.Append("}");
}
```

TESTS
CUSTOM TESTS
RESULTS
Tests passed: 0/3. Compilation error.
Test 1
Input:
prices_to_parse:
"1:0,2:1,3:2,4:3,5:4,6:5,7:6,8:7,9:8,10:9,11:10,12:11,13:12,14:13,15:14,16:15,17:16,18:17,19:18,20:19"

Output:
undefined
Expected Output:
"{0,1,10,10,1}{10,11,20,20,11}"
Console Output:
Empty
Test 2
Test 3

Service DAG

Overview
You are building an application that consists of many different services that can depend on each other. One of these services is the entrypoint which receives user requests and then makes requests to each of its dependencies, which will in turn call each of their dependencies and so on before returning.
Given a directed acyclic graph that contains these dependencies, you are tasked with determining the "load factor" for each of these services to handle this load. The load factor of a service is defined as the number of units of load it will receive if the entrypoint receives a 1 unit of load. Note that we are interested in the worst case capacity. For a given downstream service, its load factor is the number of units of load it is required to handle if all upstream services made simultaneous requests. For example, in the following dependency graph where A is the entrypoint:

Each query to A will generate one query to B which will pass it on to C and from there to D. A will also generate a query to C which will pass it on to D, so the worst case (maximum) load factors for each service is A:1, B:1, C:2, D:2.
(Important: make sure you've fully understood the above example before proceeding!)

Problem Details

service_list: An array of strings of format service_name=dependency1,dependency2.
Dependencies can be blank (e.g. dashboard=) and non-existent dependency references should be ignored (e.g. prices=users,foobar and foobar is not a service defined in the graph). Each service is defined only once in the graph.
entrypoint: An arbitrary service that is guaranteed to exist within the graph
Output: A list of all services depended by (and including) entrypoint as an array of strings with the format service_name*load_factor sorted by service name.
Example

Input:
service_list = ["logging=",

```
"user=logging",
"orders=user,foobar",
"recommendations=user,orders",
"dashboard=user,orders,recommendations"]
entrypoint = "dashboard"
```

Output (note sorted by service name)

```
["dashboard1",
"logging4",
"orders2",
"recommendations1",
"user*4"]
```

[execution time limit] 3 seconds (cs)

[input] array.string service_list

[input] string entrypoint

[output] array.string

[C#] Syntax Tips

```
// Prints help message to the console
// Returns a string
string helloWorld(string name) {
Console.Write("This prints to the console when you Run Tests");
return "Hello, " + name;
}
```

Portfolio Value Optimization

""""

Portfolio Value Optimization
You have some securities available to buy that each has a price Pi.
Your friend predicts for each security the stock price will be Si at some future date.
But based on volatility of each share, you only want to buy up to Ai shares of each security i.
Given M dollars to spend, calculate the maximum value you could potentially
achieve based on the predicted prices Si (and including any cash you have remaining).

Pi = Current Price
Si = Expected Future Price
Ai = Maximum units you are willing to purchase
M = Dollars available to invest
Example 1:
Input:
M = $140 available
N = 4 Securities
P1=15, S1=45, A1=3 (AAPL)
P2=40, S2=50, A2=3 (BYND)
P3=25, S3=35, A3=3 (SNAP)
P4=30, S4=25, A4=4 (TSLA)

Output: $265 (no cash remaining)
3 shares of apple -> 45(15 *3), 135(45 *3)
3 shares of snap -> 75, 105
0.5 share of bynd -> 20, 25
"""

Referral Count

Robinhood is famous for its referral program. It's exciting to see our users spreading the word across their friends and family. One thing that is interesting about the program is the network effect it creates. We would like to build a dashboard to track the status of the program. Specifically, we would like to learn about how people refer others through the chain of referral.

For the purpose of this question, we consider that a person refers all other people down the referral chain. For example, A refers B, C, and D in a referral chain of A -> B -> C -> D. Please build a leaderboard for the top 3 users who have the most referred users along with the referral count.

Referral rules:

A user can only be referred once.
Once the user is on the RH platform, he/she cannot be referred by other users. For example: if A refers B, no other user can refer A or B since both of them are on the RH platform.
Referrals in the input will appear in the order they were made.
Leaderboard rules:

The user must have at least 1 referral count to be on the leaderboard.

The leaderboard contains at most 3 users.
The list should be sorted by the referral count in descending order.
If there are users with the same referral count, break the ties by the alphabetical order of the user name.
Input

rh_users = ["A", "B", "C"]
| | |
v v v
new_users = ["B", "C", "D"]
Output

["A 3", "B 2", "C 1"]
[execution time limit] 3 seconds (java)

[memory limit] 1 GB

[input] array.string rh_users

A list of referring users.

[input] array.string new_users

A list of user that was referred by the users in the referrers array with the same order.

[output] array.string

An array of 3 users on the leaderboard. Each of the element here would have the "[user] [referral count]" format. For example, "A 4".

Reachable Nodes

You are given 2 arrays with paired indices so that index i from array A and index i from array B means that A points to B. This essentially forms a directed graph. The question is, from each node, how many nodes are reachable. We want to find the top 3 nodes that can reach the most other nodes

Eg. A->B
Means A can reach 1 other node. B can reach 0

A -> B -> C

A reaches 2, B reaches 1, C reaches 0

Architecture

Job Scheduler

Requirements for job scheduler listed below. Interviewer copy pasted this into the miro board.
1. Should be able to create jobs
2. Should be able to schedule and run jobs
3. Should be able to report failures and successes
4. Should be reliable and have strong guarantees about its job runs
5. Should be able to view logs and status of running jobs, as well as previously finished jobs
6. Should be able to handle when a job takes longer to run than expected (SLA)

focus on fault tolerance and schema. i got hung up on the last requirement where my schema didn't really support the last req for example if a job is to be run once every 5 minutes and the first one is taking too long, the first job should not impact the next job running at the next scheduled time.

Helpful resource:
https://leetcode.com/discuss/general-discussion/1082786/System-Design:-Designing-a-distributed-Job-Scheduler-or-Many-interesting-concepts-to-learn

https://dropbox.tech/infrastructure/asynchronous-task-scheduling-at-dropbox

Stock Price API

Design an API that can view real-time stock prices and historical prices
- get live price of a ticker
- get historical price with a date range of a ticker given 2 hoses of live data streamed directly from NASDAQ

Limit Order Sell

design robinhood to deal with limit order. That is, users can set a stock price to buy. Only buy stocks when the market price is below this price (the same goes for selling stocks). The system must support concurrency and be able to handle different failure cases.

Trading System

Design a trading system like Robinhood from the perspective of buyers and sellers. A seller should be able to make an offer to sell and cancel the order. A buyer should be able to make an offer to buy and cancel the order

References

https://www.1point3acres.com/bbs/thread-1076537-1-1.html
https://www.1point3acres.com/bbs/thread-901876-1-1.html
https://www.1point3acres.com/bbs/thread-970301-1-1.html