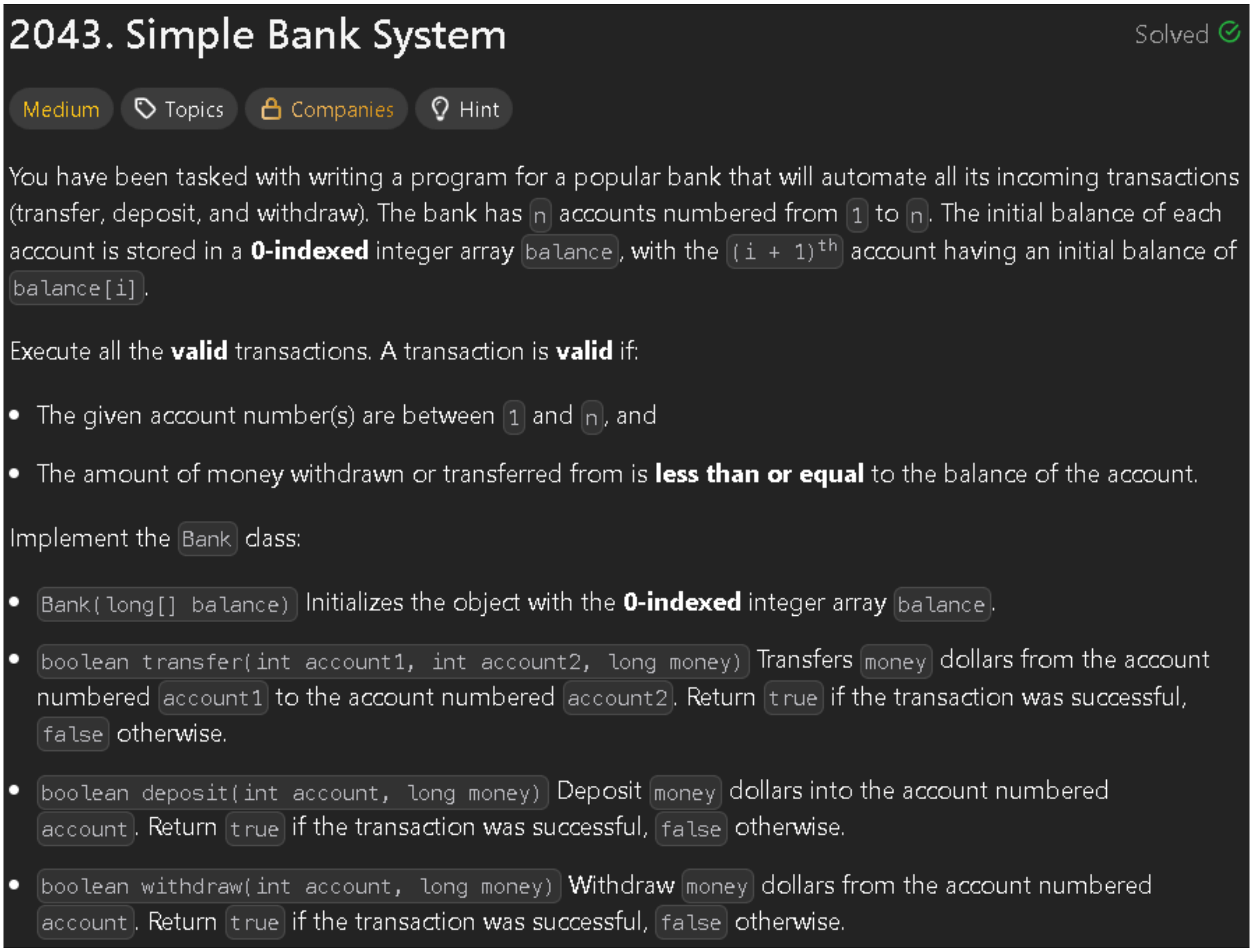


2043. Simple Bank System

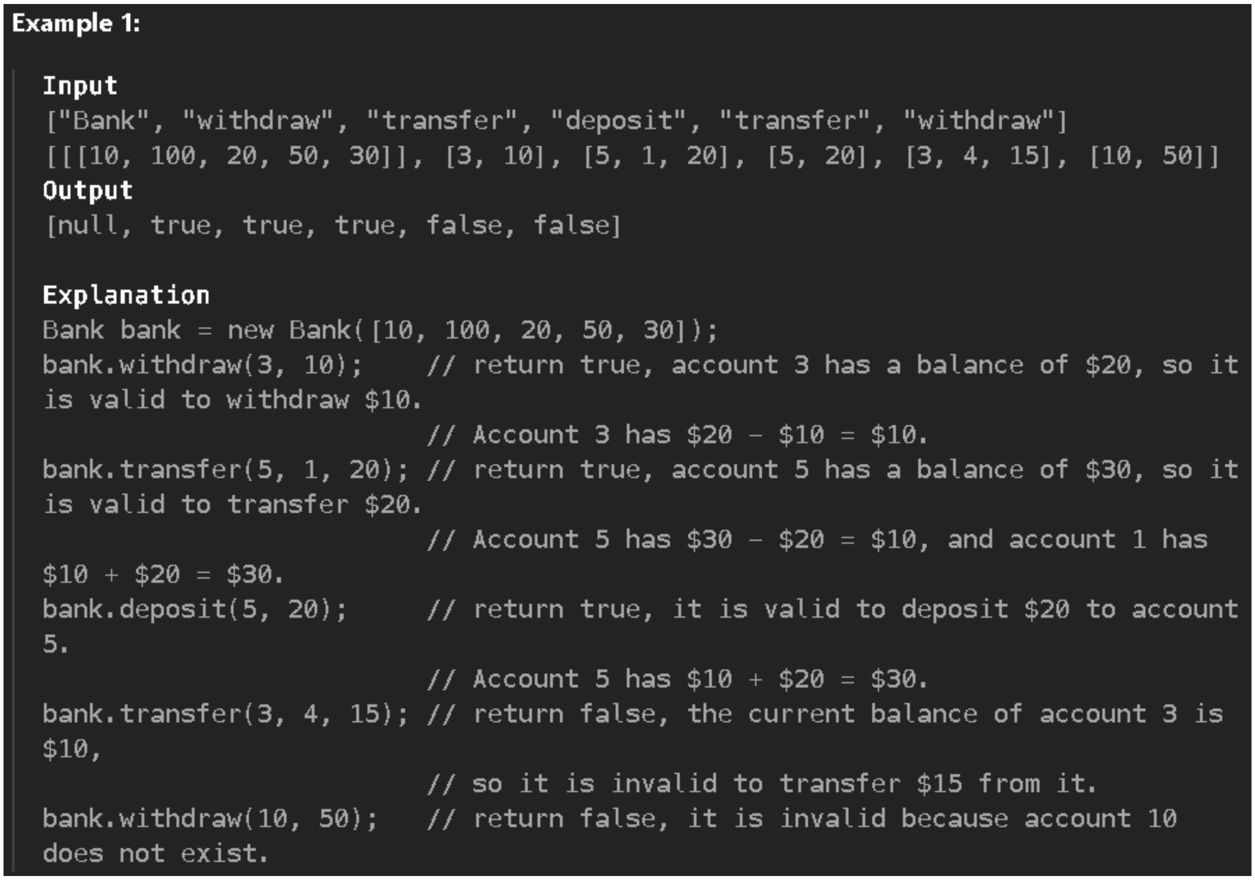
You have been tasked with writing a program for a popular bank that will automate all its incoming transactions (transfer, deposit, and withdraw). The bank has `n` accounts numbered from `1` to `n`. The initial balance of each account is stored in a **0-indexed** integer array `balance`, with the `(i + 1)`th account having an initial balance of `balance[i]`.

Execute all the **valid** transactions. A transaction is **valid** if:

- The given account number(s) are between `1` and `n`, and
- The amount of money withdrawn or transferred from is **less than or equal** to the balance of the account.

Implement the `Bank` class:

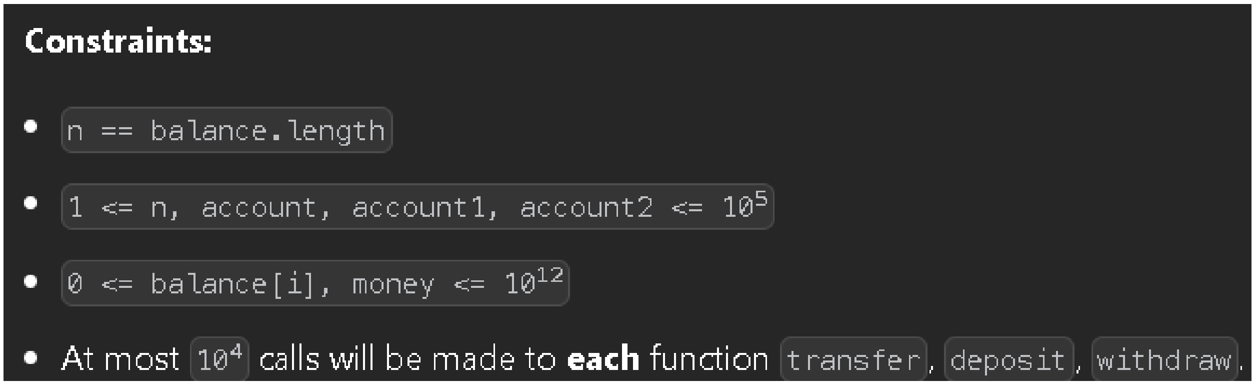
- `Bank(long[] balance)` Initializes the object with the **0-indexed** integer array `balance`.
- `boolean transfer(int account1, int account2, long money)` Transfers `money` dollars from the account numbered `account1` to the account numbered `account2`. Return `true` if the transaction was successful, `false` otherwise.
- `boolean deposit(int account, long money)` Deposit `money` dollars into the account numbered `account`. Return `true` if the transaction was successful, `false` otherwise.
- `boolean withdraw(int account, long money)` Withdraw `money` dollars from the account numbered `account`. Return `true` if the transaction was successful, `false` otherwise.



Example 1:

```
Input
["Bank", "withdraw", "transfer", "deposit", "transfer", "withdraw"]
[[[10, 100, 20, 50, 30]], [3, 10], [5, 1, 20], [5, 20], [3, 4, 15], [10, 50]]
Output
[null, true, true, true, false, false]

Explanation
Bank bank = new Bank([10, 100, 20, 50, 30]);
bank.withdraw(3, 10);    // return true, account 3 has a balance of $20, so it
is valid to withdraw $10.
                         // Account 3 has $20 - $10 = $10.
bank.transfer(5, 1, 20); // return true, account 5 has a balance of $30, so it
is valid to transfer $20.
                         // Account 5 has $30 - $20 = $10, and account 1 has
$10 + $20 = $30.
bank.deposit(5, 20);     // return true, it is valid to deposit $20 to account
5.
                         // Account 5 has $10 + $20 = $30.
bank.transfer(3, 4, 15); // return false, the current balance of account 3 is
$10,
                         // so it is invalid to transfer $15 from it.
bank.withdraw(10, 50);   // return false, it is invalid because account 10
does not exist.
```



Constraints:

- `n == balance.length`
- `1 <= n, account, account1, account2 <=` 10^5
- `0 <= balance[i], money <=` 10^{12}
- At most 10^4 calls will be made to **each** function `transfer`, `deposit`, `withdraw`.

Python:

```
from threading import RLock
from typing import List

class Bank:

    class Account:
        def __init__(self, balance: int):
            self.balance = balance
            self.lock = RLock()
        def deposit(self, amount: int):
            self.lock.acquire()
            try:
                self.balance += amount
            finally:
                self.lock.release()

        def withdraw(self, amount: int):
            self.lock.acquire()
            try:
                if self.balance < amount:
                    return False

                self.balance -= amount
            finally:
                self.lock.release()
            return True

    # Initializes the object with the 0-indexed integer array balance.
    def __init__(self, balance: List[int]):
        self.lock: RLock = RLock()
        self.accounts: List[self.Account] = []
        for b in balance:
            self.accounts.append(self.Account(b))


    # Transfers money dollars from the account numbered account1 to the account numbered
account2.
    # Return true if the transaction was successful, false otherwise.
    def transfer(self, account1: int, account2: int, money: int) -> bool:
        acc1: self.Account = self.get_account(account1)
        acc2: self.Account = self.get_account(account2)

        if not acc1 or not acc2 or money < 0:
            return False

        try:
            acc1.lock.acquire()
            acc2.lock.acquire()
            if acc1.withdraw(money):
                acc2.deposit(money)
            else:
                return False
        finally:
            acc1.lock.release()
            acc2.lock.release()

        return True


    # Deposit money dollars into the account numbered account.
    # Return true if the transaction was successful, false otherwise.
    def deposit(self, account: int, money: int) -> bool:
        if self.check_is_valid_account(account):
            self.get_account(account).deposit(money)
            return True
        return False


    # Withdraw money dollars from the account numbered account.
    # Return true if the transaction was successful, false otherwise.
    def withdraw(self, account: int, money: int) -> bool:
        if self.check_is_valid_account(account):
            return self.get_account(account).withdraw(money)
        return False

    def check_is_valid_account(self, account: int) -> bool:
        return account > 0 and account <= len(self.accounts)

    def get_account(self, account: int) -> Account:
        if not self.check_is_valid_account(account):
            return None

        return self.accounts[account-1]



# Your Bank object will be instantiated and called as such:
# obj = Bank(balance)
# param_1 = obj.transfer(account1,account2,money)
# param_2 = obj.deposit(account,money)
# param_3 = obj.withdraw(account,money)
```

JavaScript:

```
var Bank = function(balance) {
    this.bal = balance;
    this.n = balance.length;
};

Bank.prototype.valid = function(acc) {
    return acc > 0 && acc <= this.n;
};

Bank.prototype.transfer = function(account1, account2, money) {
    if (!this.valid(account1) || !this.valid(account2) || this.bal[account1 - 1] < money)
        return false;
    this.bal[account1 - 1] -= money;
    this.bal[account2 - 1] += money;
    return true;
};

Bank.prototype.deposit = function(account, money) {
    if (!this.valid(account))
        return false;
    this.bal[account - 1] += money;
    return true;
};

Bank.prototype.withdraw = function(account, money) {
    if (!this.valid(account) || this.bal[account - 1] < money)
        return false;
    this.bal[account - 1] -= money;
    return true;
};
```

Java:

```
import java.util.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import static java.util.concurrent.TimeUnit.SECONDS;

class Bank {

    public static void main(String args[]) {
        Bank bank = new Bank(new long[]{10, 100, 20, 50, 30});
        bank.withdraw(3, 10);
        bank.transfer(5, 1, 20);
        bank.deposit(5, 20);
        bank.transfer(3, 4, 15);
        bank.withdraw(10, 50);
        bank.processPendingUpdates(); // Lazy update of top 10 accounts
    }

    private final Account[] accounts;

    public Bank(long[] balance) {
        this.accounts = new Account[balance.length];
        for (int i = 0; i < balance.length; i++) {
            accounts[i] = new Account(balance[i], i + 1);
        }
    }

    public boolean transfer(int fromAccount, int toAccount, long money) {
        if (!validateAccount(fromAccount) || !(validateAccount(toAccount)) || money < 0) return
false;

        Account from = getAccount(fromAccount);
        Account to = getAccount(toAccount);

        try {
            if (from.lock.tryLock(1, SECONDS)) {
                try {
                    if (to.lock.tryLock(1, SECONDS)) {
                        try {
                            if (from.withdraw(money)) {
                                if (to.deposit(money)) {
                                    // Add transaction history for both accounts
                                    from.addTransaction(new
Transaction(Transaction.TransactionType.WITHDRAWAL, money, fromAccount));
                                    to.addTransaction(new
Transaction(Transaction.TransactionType.DEPOSIT, money, toAccount));
                                } else {
                                    // If deposit fails, roll back the withdrawal
                                    from.deposit(money); // Rollback the withdrawal
                                    return false;
                                }
                            } else {
                                return false;
                            }
                        } finally {
                            to.lock.unlock();
                        }
                    }
                } finally {
                    from.lock.unlock();
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        return true;
    }

    public boolean deposit(int account, long money) {
        if (!validateAccount(account)) return false;
        boolean result = getAccount(account).deposit(money);
        if (result) {
            getAccount(account).addTransaction(new
Transaction(Transaction.TransactionType.DEPOSIT, money, account));
        }
        return result;
    }

    public boolean withdraw(int account, long money) {
        if (!validateAccount(account)) return false;
        boolean result = getAccount(account).withdraw(money);
        if (result) {
            getAccount(account).addTransaction(new
Transaction(Transaction.TransactionType.WITHDRAWAL, money, account));
        }
        return result;
    }

    private boolean validateAccount(int account) {
        return account > 0 && account <= accounts.length;
    }

    private Account getAccount(int account) {
        return accounts[account - 1];
    }

    // Nested Transaction class
    public static class Transaction {
        private final TransactionType type;
        private final long amount;
        private final int involvedAccount;
        private final long timestamp;
        public enum TransactionType {
            DEPOSIT,
            WITHDRAWAL,
            MERGE
        }

        public Transaction(TransactionType type, long amount, int involvedAccount) {
            this.type = type;
            this.amount = amount;
            this.involvedAccount = involvedAccount;
            this.timestamp = System.currentTimeMillis();
        }

        public TransactionType getType() {
            return type;
        }

        public long getAmount() {
            return amount;
        }

        public int getInvolvedAccount() {
            return involvedAccount;
        }

        public long getTimestamp() {
            return timestamp;
        }

        @Override
        public String toString() {
            return "Transaction{" +
                    "type=" + type +
                    ", amount=" + amount +
                    ", involvedAccount=" + involvedAccount +
                    ", timestamp=" + timestamp +
                    '}';
        }
    }

    private static class Account {
        private long balance;
        private long totalTransferAmount = 0;  // Track the running total of transfers
    private final Lock lock = new ReentrantLock(true);
    private final List<Transaction> transactionHistory = new ArrayList<>();
    private final int accountId;

    public Account(long balance, int accountId) {
        this.balance = balance;
        this.accountId = accountId;
    }

    public boolean deposit(long amount) {
        try {
            if (lock.tryLock(1, SECONDS)) {
                try {
                    balance += amount;
                    totalTransferAmount += amount;  // Update running total of transfers
                } finally {
                    lock.unlock();
                }
                return true;
            }
        } catch (InterruptedException e) {
            return false;
        }
        return false;
    }

    public boolean withdraw(long amount) {
        try {
            if (lock.tryLock(1, SECONDS)) {
                try {
                    if (balance < amount) return false;
                    balance -= amount;
                    totalTransferAmount += amount;  // Update running total of transfers
                } finally {
                    lock.unlock();
                }
                return true;
            }
        } catch (InterruptedException e) {
            return false;
        }
        return false;
    }
```

```
        public long getTotalTransferAmount() {
            return totalTransferAmount;
        }

        public void addTransaction(Transaction transaction) {
            transactionHistory.add(transaction);
        }

        public void setBalance(long balance) {
            this.balance = balance;
        }

        public long getBalance() {
            return balance;
        }
    }

    // This method processes batch updates and lazy updates the top accounts heap
    public void processPendingUpdates() {
        for (Account account : accounts) {
            updateTopAccounts(account);
        }
        // Optionally, clear dirty flags if using them for batch processing
    }

    // Update the top accounts for heap based on total transfers
    private void updateTopAccounts(Account account) {
        // For example, if you're using a max-heap for the top 10 accounts:
        // This should be adjusted to a more efficient heap management approach
        // based on your specific requirements.
    }
}
```