2241. Design an ATM Machine

Medium Greedy Design Array Leetcode Link

Problem Description

This problem presents a simulation of an ATM machine that operates with five denominations of banknotes: 20, 50, 100, 200, and 500 dollars. The ATM starts out empty and supports two primary operations: depositing and withdrawing money. When depositing, the user increases the count of each denomination in a specific order. Withdrawing involves the user requesting a certain amount of money, and the ATM provides this amount using the largest denomination banknotes first. The ATM tries to give the user as little banknotes as possible, ensuring the total amount of banknotes provided sums up to the amount requested. If the ATM cannot provide the exact amount with the banknotes available (without splitting a banknote), the withdraw request is rejected and should return `[-1]` without any banknotes being dispensed.

Intuition

To solve this problem, we maintain an array that keeps track of the count of each denomination available in the ATM. On depositing banknotes, we simply increase the count for each denomination. On withdrawing, we iterate through the denominations from the largest to the smallest. We determine the maximum number of banknotes of the current denomination we can use without exceeding the requested amount. This is achieved by dividing the requested amount by the value of the current denomination. We then subtract the value of these banknotes from the requested amount and proceed to the next smallest denomination. This process is repeated until the amount is fulfilled or we realize that the withdrawal cannot be made with the available notes (i.e., there is still some amount left to be withdrawn after using all possible banknotes). In this case, we return `[-1]`. If the withdraw request is successful, we update the ATM's banknote count and return the banknotes distributed.

Solution Approach

The solution approach for the ATM class involves creating basic operations to simulate the deposit and withdrawal process of an ATM. The data structure used here is a simple list to keep track of the count of available banknotes of each denomination.

- **Initialization (`__init__`):**
  - We initialize the banknote counters as a list of 5 zeros (`self.cnt = [0] * 5`), each corresponding to one of the five denominations available.
  - We also create a list of denominations (`self.d = [20, 50, 100, 200, 500]`) corresponding to the actual values of the banknotes for easy reference.
- **Deposit (`deposit`):**
  - This method takes a list of integers that represent the counts of banknotes to be added for each denomination.
  - The method updates the counts by iterating through each denomination and adding the respective count from the input list (`banknotesCount`) to the ATM's count (using `self.cnt[i] += v`).
- **Withdrawal (`withdraw`):**
  - The withdraw method attempts to dispense the requested `amount` using the largest banknotes first.
  - We create a temporary list `ans` to store the number of banknotes to be given for each denomination.
  - From the largest to the smallest denomination, we calculate the maximum number of banknotes of the current denomination that can be used (`min(amount // self.d[i], self.cnt[i])`) without surpassing the requested amount.
  - We then subtract the total value of these banknotes from the `amount`, thus reducing the remaining amount to be dispensed.
  - If, after iterating through all denominations, the `amount` is greater than zero, this means that the remaining amount cannot be dispensed using available banknotes, so we return `[-1]`.
  - If the `amount` is zero, the withdrawal is successful, and we then update the ATM's banknote counters by subtracting the banknotes to be dispensed (`self.cnt[i] -= v`).
  - Finally, we return the array `ans`, showing the number of each banknote that will be dispensed.

This algorithm follows a greedy approach, always trying to use the largest denomination banknotes first. This is a common pattern when the goal is to minimize the number of items or entities (in this case, the number of banknotes) to reach a certain sum or total. The greedy approach works here because there are no restrictions on combinations that could prevent the use of the largest banknotes first.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Assume we have an instance of the ATM class named `atm`, and we follow these operations:

1. The atm starts off empty, so each denomination has a count of 0.
2. We deposit an array of banknotes: `banknotesCount = [0, 0, 1, 0, 2]`. This means 0 units of $20, 0 units of$50, 1 unit of $100, 0 units of$200, and 2 units of $500 banknotes are added to the ATM.
   - The ATM counts are updated, and now we have `self.cnt = [0, 0, 1, 0, 2]`.
3. Suppose a user requests to withdraw $600.
   - The ATM will try to fulfill this with the least number of banknotes, starting with the largest denomination. Since we have two $500 notes, the ATM compares$600 with $500.
   - The ATM can only give 1 note of $500 without exceeding$600. After dispensing one $500 note,$100 remains to be given.
   - Next, it looks to the $200 denomination but skips it since it's not available.
   - Then the ATM checks the $100 denomination and sees that it can dispense 1 note to fulfill the remaining amount.
   - The withdrawal array so far will be `[0, 0, 1, 0, 1]` representing 0 units of $20, 0 units of$50, 1 unit of $100, 0 units of$200, and 1 unit of $500 banknotes.
   - After the withdrawal, the ATM updates its counts, resulting in `self.cnt = [0, 0, 0, 0, 1]`.

Let's examine the steps more formally:

- Initialization: ATM `atm` starts with counts as `[0, 0, 0, 0, 0]`.
- Deposit: After depositing `banknotesCount = [0, 0, 1, 0, 2]`, the ATM's counts are `[0, 0, 1, 0, 2]`.
- Withdraw: To fulfill the withdraw request of $600,
  - The ATM iterates from the largest denomination ($500) and decides to give out 1 note, leaving$100 to be given.
  - Skips the $200 denomination as it's not available.
  - Gives out 1 note of $100 to cover the remaining$100 needed.
  - The withdrawal array is `[0, 0, 1, 0, 1]` representing the number of each denomination dispensed.
  - The ATM updates its counts to `[0, 0, 0, 0, 1]`.

If a second withdrawal request arrives, for example, $300, the ATM would go through its denominations:

- It looks at the $500 denomination but using it would exceed the withdrawal amount.
- It skips the $200 and$100 denominations since they are no longer available.
- Since there are no banknotes that can fulfill this request, the ATM would return `[-1]` indicating the withdrawal cannot be made with the available notes.

In summary, the ATM uses a greedy withdrawal strategy, dispensing the largest notes first and updating its internal counts accordingly. If, at any point during withdrawal, it cannot fulfill the requested amount with the available notes, it declines the transaction.

Python Solution

```python
from typing import List

class ATM:
    def __init__(self):
        # Initialize counts of each banknote type to 0
        self.banknote_counts = [0] * 5
        # Define the denominations in ascending order
        self.denominations = [20, 50, 100, 200, 500]

    def deposit(self, banknotesCount: List[int]) -> None:
        """
        Deposits an array of banknotes.
        :param banknotesCount: A list of integers representing the count of each banknote to deposit.
        """
        # Increment the count for each banknote denomination
        for i, count in enumerate(banknotesCount):
            self.banknote_counts[i] += count

    def withdraw(self, amount: int) -> List[int]:
        """
        Withdraws an amount by dispensing the fewest number of banknotes possible.
        :param amount: Integer amount to withdraw.
        :return: List of integers representing the count of each banknote denomination dispensed, or [-1] if transaction can't be m
        """
        # Initialize the result list to 0 for each denomination
        withdrawal_counts = [0] * 5

        # Attempt to withdraw from highest to smallest denomination
        for i in range(4, -1, -1):
            # Calculate the number of banknotes of the current denomination we can dispense
            withdrawal_counts[i] = min(amount // self.denominations[i], self.banknote_counts[i])
            # Reduce the amount by the value dispensed
            amount -= withdrawal_counts[i] * self.denominations[i]

        # Check if it's possible to dispense the full amount
        if amount > 0:
            return [-1]  # Unable to dispense the exact amount with available banknotes

        # Update the ATM's banknote counts after dispensing
        for i, count in enumerate(withdrawal_counts):
            self.banknote_counts[i] -= count

        return withdrawal_counts  # Return the banknotes dispensed


# Example usage:
# atm = ATM()
# atm.deposit([0,2,1,0,1])
# print(atm.withdraw(600))  # Should print the banknote distribution if possible, otherwise [-1]

```

Java Solution

```java
class ATM {
    // Array to hold the count of each banknote denomination
    private long[] banknoteCounts = new long[5];
    // Array to represent the value of each banknote denomination
    private int[] denominations = {20, 50, 100, 200, 500};

    // Constructor for the ATM (not needed in this context as it does nothing)
    public ATM() {
    }

    // Method to deposit a variable number of banknotes
    public void deposit(int[] banknotesCount) {
        // Iterate over the array of banknotes to be deposited
        for (int i = 0; i < banknotesCount.length; ++i) {
            // Add the deposited banknotes to the respective count in banknoteCounts
            banknoteCounts[i] += banknotesCount[i];
        }
    }

    // Method to withdraw a specified amount from the ATM
    public int[] withdraw(int amount) {
        // Array to hold the number of banknotes to dispense of each denomination
        int[] withdrawalArray = new int[5];
        // Iterate over the banknote denominations starting from the largest to the smallest
        for (int i = 4; i >= 0; --i) {
            // Calculate the number of banknotes of the current denomination to dispense
            withdrawalArray[i] = (int) Math.min(amount / denominations[i], banknoteCounts[i]);
            // Reduce the amount by the total value of the dispensed banknotes of the current denomination
            amount -= withdrawalArray[i] * denominations[i];
        }
        // If the exact amount cannot be dispensed (non-zero remainder)
        if (amount > 0) {
            // Return an array containing only -1 as an error code
            return new int[] {-1};
        }
        // If the exact amount can be dispensed, update the banknote counts
        for (int i = 0; i < 5; ++i) {
            // Subtract the dispensed banknotes from the banknoteCounts
            banknoteCounts[i] -= withdrawalArray[i];
        }
        // Return the array of dispensed banknotes
        return withdrawalArray;
    }
}

/**
 * Your ATM object will be instantiated and called as such:
 * ATM obj = new ATM();
 * obj.deposit(banknotesCount);
 * int[] param_2 = obj.withdraw(amount);
 */

```

C++ Solution

```cpp
#include <vector>
using namespace std;

// Class to simulate ATM operations
class ATM {
public:
    // Constructor to initialize the ATM with empty banknote counters
    ATM() {
    }

    // Function to deposit a specific count of banknotes into the ATM
    void deposit(vector<int> banknotesCount) {
        // Iterate through the banknote denominations and add the count of banknotes
        for (int i = 0; i < banknotesCount.size(); ++i) {
            banknoteCounters[i] += banknotesCount[i];
        }
    }

    // Function to withdraw a specific amount from the ATM
    vector<int> withdraw(int amount) {
        vector<int> result(5, 0);  // Stores the number of banknotes to dispense for each denomination

        // Iterate through the banknote denominations from highest to lowest
        for (int i = 4; i >= 0; --i) {
            // Calculate the max number of banknotes of denomination that can be dispensed
            result[i] = min(static_cast<long long>(amount) / denominations[i], banknoteCounters[i]);
            // Subtract the value of the dispensed banknotes from the amount
            amount -= result[i] * denominations[i];
        }

        // Check if the requested amount could not be withdrawn from the available banknotes
        if (amount > 0) {
            return {-1}; // Return {-1} to indicate failure to withdraw the full amount
        }

        // Subtract dispensed banknotes from the banknote counters
        for (int i = 0; i < 5; ++i) {
            banknoteCounters[i] -= result[i];
        }

        return result; // Return the number of banknotes dispensed for each denomination
    }

private:
    // Array to hold the count of each banknote denomination in the ATM
    long long banknoteCounters[5] = {0};
    // Array representing the value of each banknote denomination
    int denominations[5] = {20, 50, 100, 200, 500};
};

/**
 * Example of how the ATM class is used:
 * ATM* atm = new ATM();
 * atm->deposit(banknotesCount);
 * vector<int> withdrawnAmount = atm->withdraw(amount);
 */

```

Typescript Solution

```typescript
// Variables to hold the count and denominations of each banknote in the ATM
let banknoteCounters: Array<number> = [0, 0, 0, 0, 0];
let denominations: Array<number> = [20, 50, 100, 200, 500];

/**
 * Initialize the ATM with an empty banknote counters data structure.
 */
function initATM() {
    banknoteCounters = [0, 0, 0, 0, 0];
}

/**
 * Deposit a specific count of banknotes into the ATM.
 * @param {number[]} banknotesCount - Array containing the count of banknotes to deposit for each denomination.
 */
function deposit(banknotesCount: Array<number>) {
    for (let i = 0; i < banknotesCount.length; i++) {
        banknoteCounters[i] += banknotesCount[i];
    }
}

/**
 * Withdraw a specific amount from the ATM.
 * @param {number} amount - The amount to withdraw from the ATM.
 * @returns {number[]} array containing the number of banknotes to be dispensed for each denomination, or [-1] if it's not possible
 */
function withdraw(amount: number): Array<number> {
    let result = [0, 0, 0, 0, 0]; // Stores the number of banknotes to dispense for each denomination

    // Iterate from highest denomination to lowest
    for (let i = 4; i >= 0; i--) {
        let numBanknotes = Math.min(Math.floor(amount / denominations[i]), banknoteCounters[i]);
        amount -= numBanknotes * denominations[i];
        result[i] = numBanknotes;
    }

    // Check if the full amount could be withdrawn
    if (amount > 0) {
        // Failure to withdraw the full amount
        return [-1];
    }

    // Subtract the dispensed banknotes from the banknote counters
    for (let i = 0; i < 5; i++) {
        banknoteCounters[i] -= result[i];
    }

    // Return the number of banknotes dispensed for each denomination
    return result;
}

// Example usage
// initATM(); // Initializes the ATM
// deposit([10, 10, 10, 10, 10]); // Deposits an initial amount into the ATM
// const withdrawnAmount: Array<number> = withdraw(1500); // Withdraws an amount from the ATM

```

Time and Space Complexity

Time Complexity

The time complexity of `deposit`:

- The function `deposit` iterates over the `banknotesCount` list once, which has a fixed size of 5. The number of operations is constant, regardless of input size. Therefore, the time complexity is `O(1)`.

The time complexity of `withdraw`:

- The withdraw function contains a loop that iterates in reverse over a fixed-size list (`self.d`), which has 5 elements corresponding to the different banknote denominations. The operations inside the loop are constant time.
- The loop runs a fixed number of times regardless of the `amount` because the denominations are given and unchanging. Therefore, this loop contributes a constant time complexity `O(1)`.
- The second loop occurs only if the withdrawal is successful, iterating over the `ans` list to update the `self.cnt` array. Since `ans` has a fixed size of 5, this also contributes `O(1)`.

In summary, both `deposit` and `withdraw` have constant time complexity because they operate over a fixed-size array that represents the banknote denominations and counts. Therefore, the overall time complexity of each operation in the `ATM` class is `O(1)`.

Space Complexity

The space complexity of the `ATM` class:

- The class maintains a fixed-size integer array `self.cnt` which has a length of 5, and the array `self.d` which is also of length 5 and is used for the denominations. These do not scale with input size and are considered constant space `O(1)`.
- Temporary variables used in both `deposit` and `withdraw` methods are also constant in size, not dependent on the input.

Given these observations, the space complexity of the `ATM` class operations (`deposit` and `withdraw`) is `O(1)`. The state of the ATM and the operations do not require additional space that grows with the input size.