1603. Design Parking System

Easy Design Counting Simulation

Leetcode Link

Problem Description

In this problem, we're asked to design a simple parking system for a parking lot with three different types of parking spaces: big, medium, and small. Each type of parking space has a fixed number of slots that can be occupied by cars of that specific size. The parking system needs to be able to handle two operations:

1. Initializing the parking system with the number of slots for each type of parking space.
2. Adding a car to the parking lot, which is subject to there being an available slot for the car's type.

When a car tries to park, the parking system checks if there is an available slot for that particular size of the car. If an appropriate slot is available, the car parks (i.e., the count of available slots of that type reduces by one), and the system returns `true`. If no slot is available for that car's type, the system returns `false`.

The key to solving the problem is to keep track of the number of available slots for each car type in an efficient way that allows quick updates and queries.

Intuition

The solution approach is straightforward. Since there are only three types of car slots available, we can use an array with three elements, where each element corresponds to the count of available slots for each car type.

1. **Initialization**: We initialize an array of size four, where indices 1, 2, and 3 represent 'big', 'medium', and 'small' slots respectively. The reason for choosing index 1 to 3 instead of 0 to 2 is to map the `carType` directly to the array index, as `carType` is defined to be 1, 2, or 3 in the problem description. We leave index 0 unused. Each element in this array stores the number of available spaces for that type of car.

2. **Adding a Car**: The `addCar` function is called with a `carType`, which is used as the index to directly access the corresponding count in the array. We first check if there's at least one slot available of the given car type by checking if the counter at that index is greater than zero. If it is, we decrement the counter as we've now occupied a slot and return `true`. If the counter is already at zero, it means there are no available slots for that car type and we return `false`.

This array-based system allows constant-time operations for both adding cars and initializing the parking system, which means the time complexity for each operation is O(1), providing us with a very efficient solution.

Solution Approach

The implementation of the solution can be broken down into two parts, following the two major functionalities of the `ParkingSystem` class:

Part 1: Initialization

In the constructor `__init__`, we initialize an instance variable called `self.cnt`. This variable is a list that stores the count of available spots for each car type.

- Big car slots are stored at index 1, hence `self.cnt[1] = big`.
- Medium car slots are stored at index 2, hence `self.cnt[2] = medium`.
- Small car slots are stored at index 3, hence `self.cnt[3] = small`.

The array is initialized with the number of slots for each type of parking space given as arguments to the constructor. The index 0 of the array is not used in this problem.

```
def __init__(self, big: int, medium: int, small: int):
    # Initializing the array with an extra index for convenience in accessing by carType directly
    self.cnt = [0, big, medium, small]
```

Part 2: Adding a Car

The next part of our solution is the `addCar` function. This function's purpose is to process the request of adding a car to the parking lot based on the car's type and the available space.

Here's the step-by-step process of what happens when `addCar` is called:

1. Check if there are available slots for the given `carType` by directly accessing the `self.cnt` array using `carType` as the index.
2. If `self.cnt[carType]` is greater than 0, it implies an available slot. We decrease the count by one using `self.cnt[carType] -= 1` — indicating that we've filled one slot — and return `True`.
3. If `self.cnt[carType]` equals 0, it means there are no slots available, and we return `False`.

```
def addCar(self, carType: int) -> bool:
    # Check if the car type has an available slot
    if self.cnt[carType] == 0:
        # If not, return False
        return False
    # If there is a slot, decrement the counter and return True
    self.cnt[carType] -= 1
    return True
```

The data structure used in this solution, a list in Python (also called an array in some programming languages), is the optimal choice for this scenario due to:

- The fixed number of car types, which corresponds to a fixed number of list indices.
- The need for constant-time access and update operations, both of which lists provide.

Algorithmically, the solution is simple and does not involve complex patterns or algorithms. It leverages direct indexing for fast operations, avoiding any iterations or searches. Through this method, both initialization and adding cars are performed with a time complexity of O(1).

Example Walkthrough

Let's go through an example to illustrate how the solution works.

Suppose the parking lot has the following number of slots for each car type:

- Big: 1
- Medium: 2
- Small: 3

And the sequence of cars that arrive are as follows:

1. A big car
2. A medium car
3. Another medium car
4. A small car
5. Another small car
6. A big car again

We will walk through how the `ParkingSystem` would handle this sequence of cars.

Step 1: Initialization

First, we initialize the parking system with the available slots. Using the solution's `__init__` method:

```
parking_system = ParkingSystem(1, 2, 3)
```

After initialization, our `self.cnt` array looks like this: `[0, 1, 2, 3]`

Step 2: Adding Cars

- When the first big car arrives, we call `addCar(1)`. Since `self.cnt[1]` is 1 (there's one big slot available), the car is parked, `self.cnt` gets updated to `[0, 0, 2, 3]`, and `True` is returned.
- The medium car arrives, we call `addCar(2)`. Since `self.cnt[2]` is 2, the car is parked, `self.cnt` is now `[0, 0, 1, 3]`, and `True` is returned.
- Another medium car arrives, we call `addCar(2)` again. Now `self.cnt[2]` is 1, so the car is parked, `self.cnt` becomes `[0, 0, 0, 3]`, and `True` is returned.
- A small car arrives, we call `addCar(3)`. `self.cnt[3]` is 3, so the car is parked, `self.cnt` updates to `[0, 0, 0, 2]`, and `True` is returned.
- Another small car arrives, `addCar(3)` is called. `self.cnt[3]` is now 2, so this car is also parked, updating `self.cnt` to `[0, 0, 0, 1]`, and `True` is returned.
- Finally, another big car tries to park, so we call `addCar(1)`. But `self.cnt[1]` is 0 because there are no more big slots available after the first car parked, so `False` is returned.

Throughout these operations, each `addCar` call checks and updates the `self.cnt` array in constant time, illustrating both the efficiency and simplicity of the approach.

Python Solution

```
class ParkingSystem:
    def __init__(self, big: int, medium: int, small: int):
        # Initialize a ParkingSystem object with the number of parking spots available for each car size
        self.spots_available = [0, big, medium, small]

    def addCar(self, carType: int) -> bool:
        """Attempt to park a car of a specific type into the parking system.

        Args:
            carType (int): The type of the car (1 = big, 2 = medium, 3 = small).

        Returns:
            bool: True if the car can be parked, False if no spots available for the car type.
        """
        # Check if there are available spots for the given car type
        if self.spots_available[carType] == 0:
            # Return False if there are no spots available for the given car type
            return False
        # Decrease the count of available spots for the car type
        self.spots_available[carType] -= 1
        # Return True since the car has been successfully parked
        return True

# Here is how you create an instance of the ParkingSystem and attempt to add a car of a particular type
# obj = ParkingSystem(big, medium, small)
# result = obj.addCar(carType)  # result will be either True or False depending on the availability of the spot
```

Java Solution

```
// Class representing a parking system with a fixed number of parking spots
// for big, medium, and small cars.
class ParkingSystem {

    // Array to store the number of available spots for each car type.
    private int[] carSpotsAvailable;

    // Constructor for the ParkingSystem class.
    // Initializes the number of parking spots available for each car type.
    // big - number of spots for big cars
    // medium - number of spots for medium cars
    // small - number of spots for small cars
    public ParkingSystem(int big, int medium, int small) {
        // Index 0 is not used for simplicity,
        // indexes 1 to 3 correspond to big, medium, and small car types
        // respectively.
        carSpotsAvailable = new int[]{0, big, medium, small};
    }

    // Method to add a car to the parking if there's available spot for its type.
    // carType - the type of the car (1 for big, 2 for medium, 3 for small)
    // Returns true if a car was successfully parked, false if no spot was available.
    public boolean addCar(int carType) {
        // Check if there is no available space for the car type.
        if (carSpotsAvailable[carType] == 0) {
            return false;
        }
        // Decrease the count of available spots for the car type as one is now taken.
        --carSpotsAvailable[carType];
        return true;
    }
}

// The ParkingSystem class could be used as follows:
// ParkingSystem obj = new ParkingSystem(big, medium, small);
// boolean isParked = obj.addCar(carType);
```

C++ Solution

```
class ParkingSystem {
private:
    vector<int> spotsAvailable; // Vector to hold the available spots for each car type

public:
    // Constructor initializing the number of parking spots for different sizes of cars
    ParkingSystem(int big, int medium, int small) {
        spotsAvailable = {0, big, medium, small}; // Index 0 is ignored for convenience
    }

    // Function to add a car of a specific type to the parking system
    bool addCar(int carType) {
        // Check if there is a spot available for the car type
        if (spotsAvailable[carType] == 0) {
            // If no spots are available, return false
            return false;
        }
        // If there is a spot available, decrease the count and return true
        --spotsAvailable[carType];
        return true;
    }
};

/**
 * Your ParkingSystem object will be instantiated and called as such:
 * ParkingSystem* obj = new ParkingSystem(big, medium, small);
 * bool param_1 = obj->addCar(carType);
 */
```

Typescript Solution

```
// Counts for available parking spots: index 0 for big cars, 1 for medium cars, and 2 for small cars
let parkingSpotCounts: [number, number, number];

// Initializes the parking system with the specified number of parking spots for each type of car
function initializeParkingSystem(big: number, medium: number, small: number): void {
    parkingSpotCounts = [big, medium, small];
}

// Attempts to add a car to the parking system based on car type
// Returns true if parking is successful; false otherwise
function addCarToParkingSystem(carType: number): boolean {
    // Check if the car type is valid (1: big, 2: medium, 3: small)
    if (carType < 1 || carType > 3) {
        return false;
    }

    // Adjusting carType to zero-based index for the array
    const index = carType - 1;

    // Check if there is available spot for the given type of car
    if (parkingSpotCounts[index] === 0) {
        // No available spot for this type of car
        return false;
    }

    // Decrement the count for the given car type spot
    parkingSpotCounts[index]--;
    // Parking successful
    return true;
}

// Example usage:
// Initialize the parking system with 1 big spot, 2 medium spots and 3 small spots
initializeParkingSystem(1, 2, 3);
// Attempt to add a medium car to the parking system
const wasCarAdded: boolean = addCarToParkingSystem(2);
```

Time and Space Complexity

Time Complexity

The time complexity of both the `__init__` method and the `addCar` method is `O(1)`. This is because both methods perform a constant number of operations regardless of the input size:

- `__init__`: Initializes the `cnt` array with three integers, which is a constant-time operation as it involves setting up three fixed indices.
- `addCar`: Accesses and modifies the `cnt` array at the index corresponding to `carType`, which is a constant-time operation due to direct array indexing.

Therefore, overall, the time complexity is `O(1)` for initialization and each car parking attempt.

Space Complexity

The space complexity of the `ParkingSystem` class is `O(1)`. This constant space is due to the `cnt` array which always contains three elements regardless of the number of operations performed or the size of input parameters. The space occupied by the `ParkingSystem` object remains constant throughout its lifetime.