1386. Cinema Seat Allocation

Medium Greedy Bit Manipulation Array Hash Table [Leetcode Link]

Problem Description

In the given scenario, a cinema consists of `n` rows, each with 10 seats. These seats are labelled from 1 to 10. The challenge is to find out the maximum number of four-person families that we can seat in the cinema without having anyone sit in a seat that has already been reserved. The array `reservedSeats` provides us with the information about seats that have been taken (for example, [3,8] means row 3, seat 8 is reserved).

However, there are specific rules we must follow:

- A family of four must sit in the same row and side by side.
- The aisle is something to consider since a family can only be split by the aisle if it separates them two by two (two on one side and the other two on the opposite side of the aisle).

Given this setup, we need to determine how many groups of four can be seated in an optimal arrangement.

Intuition

To solve this problem, we can use bitmasking and a greedy approach. The intuition behind this approach can be broken down into the following insights:

- First, we recognize that there are only a few configurations of four seats that can fit a family: either starting from seat 2, seat 4, or seat 6. These seats will be respectively represented by the masks 0b0111100000, 0b0001111000, and 0b0000011110 in binary where a 1 denotes an occupied seat.
- Next, we consider that rows without any reserved seats can obviously fit two families (one on each end avoiding the aisle in the middle).
- For rows with reserved seats, we use a dictionary to keep track of which seats are occupied using bitmasking. A `defaultdict` is used to store a bitmask per row where a set bit (1) represents an occupied seat.
- Then, we iterate through each row with at least one reserved seat and check against our family seat masks to see if there's room for a family. For every match, we increment our answer by one and update the mask to reflect the newly occupied seats to ensure we do not double-count space for another family in the same four-seat configuration.
- If none of the masks fit, it means that we cannot seat a family of four in that specific row without using the exception (2 people on each side of the aisle).
- By iterating through all the reserved seats and performing the above checks, we can find the maximum number of four-person family groups that can be seated in the cinema.

Solution Approach

The solution approach makes use of a hashmap (in Python, a `defaultdict` of integer type) and bitwise operations. This is executed through the following steps:

1. **Dictionary Creation**: A dictionary `d` is created to keep track of reserved seats for each row. The key is the row number, and the value is a bitmask where each bit corresponds to a seat in the row, and a set bit (`1`) means the seat is reserved. The bitmask is built by shifting a `1` left by ((10 - j)) places, where (j) is the seat number. The expression `1 << (10 - j)` turns into a bitmask where all bits are `0` except the bit that corresponds to the reserved seat.

2. **Defining Seat Masks**: Since there are only certain configurations where a family of four can sit together (with no one sitting in the aisle), three masks are predefined to represent these configurations:

   - 2nd to 5th seats: `0b0111100000`
   - 4th to 7th seats: `0b0001111000`
   - 6th to 9th seats: `0b0000011110`

   These masks represent the seats that form a valid group without any of them being an aisle seat.

3. **Initial Count**: The initial count `ans` is set to twice the number of rows without any reservations, as it's possible to fit two families in an empty row.

4. **Row Iteration and Bitmask Checking**: The algorithm iterates over each reserved row (x) in the dictionary. For each row, it will then check against each of the three predefined masks to see if any groups of four seats are available:

   - If `(x & mask) == 0`, this means the seats corresponding to the current mask are all unreserved, and a family can be placed there.
   - The bitmask of the current row is updated with the mask to mark these seats as occupied (`x |= mask`) to prevent other families from being placed in the same seats.
   - For each successful check, the answer `ans` is incremented by one.
5. **Final Answer**: After all the reserved rows have been processed and the available spaces for families have been calculated, the algorithm returns the total number of families that can be seated, `ans`.

By utilizing bitmasks to efficiently check seat availability and update seat reservations, and by keeping track of reserved seats per row in a dictionary, the algorithm is able to calculate the maximum number of four-person families that can be seated in the cinema in a time-efficient manner.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Assume a cinema has 3 rows (`n = 3`), and we have the following reserved seats: `reservedSeats = [[1,2], [1,3], [1,8], [2,6], [3,1], [3,10]]`.

- Row 1 has seats 2 and 3 reserved.
- Row 2 has seat 6 reserved.
- Row 3 has seats 1 and 10 reserved.

Step 1: Dictionary Creation

We create a dictionary to keep track of the reserved seats with bitmasking:

- For Row 1: reserved seats are 2 and 3, so the bitmask is `0b0000001100`.
- For Row 2: the reserved seat is 6, so the bitmask is `0b0000010000`.
- For Row 3: reserved seats are 1 and 10, so the bitmask is `0b1000000001`.

Step 2: Defining Seat Masks

Three predefined masks:

- For 2nd to 5th seats: `mask1 = 0b0111100000`.
- For 4th to 7th seats: `mask2 = 0b0001111000`.
- For 6th to 9th seats: `mask3 = 0b0000011110`.

Step 3: Initial Count

Initially, there are no rows without reservations, so `ans = 0`.

Step 4: Row Iteration and Bitmask Checking

- For Row 1 with bitmask `0b0000001100`:

  - Check against `mask1`: `(0b0000001100 & mask1) != 0`, no family can be seated.
  - Check against `mask2`: `(0b0000001100 & mask2) == 0`, one family can be seated, bitmask updates to `0b0001111100`.
  - Check against `mask3`: `(0b0001111100 & mask3) != 0`, no additional family.
  - We placed one family in Row 1, so now `ans = 1`.
- For Row 2 with bitmask `0b0000010000`:

  - Check against `mask1`: `(0b0000010000 & mask1) == 0`, one family can be seated, bitmask updates to `0b0111110000`.
  - Check against `mask2`: `(0b0111110000 & mask2) != 0`, no additional family.
  - Check against `mask3`: `(0b0000010000 & mask3) != 0`, no family.
  - We placed one family in Row 2, so now `ans = 2`.
- For Row 3 with bitmask `0b1000000001`:

  - All seats except aisle ones are available, so we can place two families, one in `mask1` and one in `mask3`. But since we only need non-aisle seats for a group of 4, we can ignore this case for simplicity.
  - The bitmask updates to `0b1111111111`, marking all seats as taken.
  - We placed two families in Row 3, so now `ans = 4`.

Step 5: Final Answer

After checking all the reserved rows, we conclude that we can seat `ans = 4` families of four in this cinema.

Therefore, in our small example of a 3-row cinema with the given reserved seats, we can optimally fit 4 four-person families.

Python Solution

```python
from collections import defaultdict

class Solution:
    def maxNumberOfFamilies(self, n: int, reserved_seats: list[list[int]]) -> int:
        # Create a dictionary to keep track of reserved seats by row.
        reservations_dict = defaultdict(int)

        # Populate the reservations_dict with the bitwise representation of the reserved seats.
        for row, seat in reserved_seats:
            reservations_dict[row] |= 1 << (10 - seat)

        # Define masks for the three possible ways a family can sit.
        # These are bit masks that check sets of 4 seats that are together.
        family_masks = (0b0111100000, 0b0000011110, 0b0001111000)

        # Pre-calculate the number of families that can sit without any reservation conflicts.
        # If a row has no reservations, two families can sit there.
        answer = (n - len(reservations_dict)) * 2

        # Check each row with reservations to see how many more families can sit.
        for reserved_row in reservations_dict.values():
            for mask in family_masks:
                # If the family can sit in the checked pattern (mask) without conflicting with reserved seats...
                if (reserved_row & mask) == 0:
                    # Mark the seats as taken to avoid double counting.
                    reserved_row |= mask
                    # One more family can sit in this row.
                    answer += 1

        # Return the total number of families that can sit together.
        return answer

```

Java Solution

```java
class Solution {
    public int maxNumberOfFamilies(int n, int[][] reservedSeats) {
        // Define a map to keep track of the reserved seats in each row
        Map<Integer, Integer> reservedMap = new HashMap<>();

        // Iterate over the list of reserved seats and update the map
        // The key is the row number, and the value is a bit mask representing reserved seats
        for (int[] seat : reservedSeats) {
            int row = seat[0], seatNum = seat[1];
            // Set the corresponding bit for the reserved seat
            reservedMap.merge(row, 1 << (10 - seatNum), (oldValue, value) -> oldValue | value);
        }

        // Define the masks that represent the available blocks of 4 seats
        int[] masks = {
            0b0111100000, // left block: seats 2-5
            0b0000011110, // right block: seats 6-9
            0b0001111000  // middle block: seats 4-7
        };

        // Start with the maximum number of families that can be seated without any reservations
        int ans = (n - reservedMap.size()) * 2;

        // Iterate over the rows with reservations and find available spots
        for (int reserved : reservedMap.values()) {
            for (int mask : masks) {
                // If all seats represented by a mask are not reserved, increment ans
                if ((reserved & mask) == 0) {
                    reserved |= mask; // Update the reserved seats
                    ++ans;
                    break; // Once a family is seated, no need to check other masks for this row
                }
            }
        }
        return ans; // Return the maximum number of families that can be seated
    }
}

```

C++ Solution

```cpp
#include <unordered_map>
#include <vector>

class Solution {
public:
    int maxNumberOfFamilies(int numRows, std::vector<std::vector<int>>& reservedSeats) {
        // Create a map to hold the occupied seats for each row
        std::unordered_map<int, int> occupiedSeats;

        // Process the reserved seats and mark them in the occupiedSeats map
        for (const auto& seat : reservedSeats) {
            int row = seat[0], col = seat[1];
            // Create a bitmask and mark the seat as occupied
            occupiedSeats[row] |= 1 << (10 - col); // Shift bits to the column position
        }

        // Family seating patterns that can occupy four consecutive seats
        // Pattern 1: Seats 2, 3, 4, 5 (between aisle seats)
        // Pattern 2: Seats 6, 7, 8, 9 (between aisle seats)
        // Pattern 3: Seats 4, 5, 6, 7 (middle seats)
        int seatingPatterns[3] = {0b0111100000, 0b0000011110, 0b0001111000};

        // Every row not in occupiedSeats can fit 2 families
        int maxFamilies = (numRows - occupiedSeats.size()) * 2;

        // Go through the occupiedSeats to find the number of families that can fit
        for (auto& [row, occupied] : occupiedSeats) {
            for (int& pattern : seatingPatterns) {
                // Check if the pattern fits in the row
                if ((occupied & pattern) == 0) {
                    // If yes, update the occupied bitmask and increase count
                    occupied |= pattern;
                    ++maxFamilies;
                    break; // Only one pattern can fit if another family has already occupied some seats
                }
            }
        }

        return maxFamilies;
    }
};

```

Typescript Solution

```typescript
function maxNumberOfFamilies(n: number, reservedSeats: number[][]): number {
    // Map for storing the reserved seats information per row.
    const reservedMap: Map<number, number> = new Map();

    // Populating the reservedMap. A bit-mask is created for each row.
    for (const [row, seat] of reservedSeats) {
        reservedMap.set(row, (reservedMap.get(row) ?? 0) | (1 << (10 - seat)));
    }

    // Calculate the maximum number of families that can sit in unreserved rows.
    let maxFamilies = (n - reservedMap.size) * 2;

    // Define bit-masks for valid family seating positions.
    const familySeatMasks = [0b0111100000, 0b0000011110, 0b0001111000];

    // Iterate over reserved rows to check if there's a seating configuration
    // available for a family.
    for (const [_, reservedSeatsBitmask] of reservedMap) {
        for (const mask of familySeatMasks) {
            // If a valid family seating position is empty.
            if ((reservedSeatsBitmask & mask) === 0) {
                // Mark the position as occupied to prevent double counting.
                reservedSeatsBitmask |= mask;
                // Increment the number of families as we found a spot.
                maxFamilies++;
            }
        }
    }

    return maxFamilies;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code consists of two parts:

1. **Iterating through the reserved seats (`reservedSeats` list):** This part involves iterating through each seat reservation in the provided list, which is of size `m`, and setting a corresponding bit in a dictionary `d`. The bit manipulation operation for a single seat is constant; hence, this part of the algorithm is `O(m)`, where `m` is the number of reserved seats.

2. **Processing the dictionary (`d`) to count the number of families that can be accommodated:** Here, we iterate through each row that has at least one reserved seat (not more than `m` such rows) and check against three different masks to find a suitable spot for a family of 4. This also involves constant time operations for each row. Therefore, the complexity for this part is also `O(m)`.

Combining the two, the overall time complexity of the algorithm is `O(m)`.

Space Complexity

For space complexity, we mainly consider the dictionary `d` that is used to store rows with reserved seats:

- **Dictionary (`d`) to store the reserved seats:** The space used by this dictionary depends on how many different rows have reserved seats, which in the worst-case scenario can be `m`. Therefore, the space complexity is `O(m)`, where `m` is the number of reserved rows.

Overall, the space complexity of the code is `O(m)`.