849. Maximize Distance to Closest Person

Medium Array

Leetcode Link

Problem Description

In this problem, we are provided with an array `seats` that represents a row of seats. Each seat can be either occupied (denoted by `1`) or empty (denoted by `0`). We know that there is at least one empty seat and at least one occupied seat in the array. The objective is to find a seat for Alex where, if he sits down, the distance to the nearest person already seated is as large as possible. The distance is defined as the number of seats between Alex's seat and the closest seated person, and we need to maximize this distance to ensure maximum possible personal space for Alex. Our goal is to calculate and return this maximum distance.

Intuition

The intuition behind the solution is to identify the longest stretch of consecutive empty seats, with consideration of the edge cases where the longest stretch might actually start at the beginning of the row or end at the last seat in the row.

Here are key insights to form a strategy:

1. Identify stretches of empty seats and find the longest stretch. This corresponds to finding the largest gap between two occupied seats.
2. Special consideration for the first and last seats: If the first or last seats are empty, the gap is counted from the edge of the row to the first or last occupied seat respectively. This needs special handling since the "distance" in this case is not halved—it is taken as the entire length of the stretch.
3. Find the longest stretch: To maximize the distance to the nearest person, Alex can sit exactly in the middle of the longest stretch of empty seats. The maximum distance to the closest person would then be half the length of this longest stretch. However, if this stretch is at the beginning or end of the row, he will sit at the first or last available seat, resulting in the distance being the entire length of the stretch.

The approach to solving the problem involves iterating through the `seats` array to identify the first and last occupied seats, and in the process, also recording the maximum distance between two consecutive occupied seats. The maximum distance Alex can sit away from the closest person would then be the maximum of the following three values:

- The distance from the start of the row to the first occupied seat.
- The distance from the last occupied seat to the end of the row.
- Half of the distance of the longest stretch of empty seats between two occupied seats (since Alex would sit exactly in the middle of this stretch).

This method efficiently calculates the optimal seat for Alex in a single pass through the `seats` array, avoiding the need for multiple iterations or complex calculations.

Solution Approach

The solution approach involves a single pass through the row of `seats` using a for-loop. During this pass, the code keeps track of the index of the first and last occupied seats, as well as the distance to the last occupied seat seen so far when examining each seat. The following elements contribute to the solution:

Variables Used:

- `first`: This holds the index of the first occupied seat found in the row. Initially, it is set to `None`.
- `last`: This stores the index of the most recently found occupied seat as we iterate through the `seats`. It is also initially set to `None`.
- `d`: This variable maintains the maximum distance between two occupied seats as the iteration progresses. We initialize it to `0`.

The Algorithm:

1. Iterate through the `seats` array with the index `i` and the value `c` using the `enumerate` function, which provides both the index and value during the iteration.
2. Check if the current seat `c` is occupied (i.e., `c` is `1`):
   - If an occupied seat is found and it's not the first one (`last` is not `None`), update `d` with the maximum value between the current `d` and the difference between the current index `i` and the index of the last occupied seat `last`.
   - If this is the first occupied seat found (`first` is `None`), store its index in `first`.
   - Update `last` with the current index `i` because it is the latest occupied seat.
3. After completing the iteration, calculate the maximum distance Alex can be from the closest person by taking the maximum of three potential values:
   - The distance from the start of the row to the first occupied seat (`first`).
   - The distance from the last occupied seat to the end of the row (`len(seats) - last - 1`).
   - Half the distance of the longest stretch of empty seats between two occupied seats (`d // 2`).

The Code:

```
class Solution:
    def maxDistToClosest(self, seats: List[int]) -> int:
        first = last = None
        d = 0
        for i, c in enumerate(seats):
            if c == 1:
                if last is not None:
                    d = max(d, i - last)
                if first is None:
                    first = i
                last = i
        return max(first, len(seats) - last - 1, d // 2)
```

This implementation uses a greedy approach to find the maximum distance in an efficient manner. No additional data structure is required aside from a few variables to keep track of indices and the maximum distance found. Because of this, the space complexity is O(1), and since we are iterating through the array only once, the time complexity is O(n), where `n` is the number of seats in the row.

Example Walkthrough

Imagine we have an array `seats` that looks like this: `[0, 1, 0, 0, 1, 0, 1, 0, 0, 0]`.

To apply our solution approach, let's walk through the algorithm step by step:

1. Initialize: `first = last = None` and `d = 0`.
2. Begin iterating through the `seats` array with indices and values.
3. At index `0`, the seat is empty, so no changes are made.
4. At index `1`, the seat is occupied so:
   - Since this is the first occupied seat we've found, set `first` to the current index (`first = 1`).
   - Update `last` to be the current index as well (`last = 1`).
5. Continue iterating. Seats `2` and `3` are empty, so no changes.
6. At index `4`, we encounter another occupied seat.
   - Since `last` is not `None`, we calculate the distance from the previous occupied seat at index `1` to the current seat at index `4`, which yields a distance of `3`. Update `d` to be the maximum of `d` and `3`, so `d = 3`.
   - Update `last` to the current index (`last = 4`).
7. Move on to seat `5` which is empty, so no changes.
8. At index `6`, there is another occupied seat.
   - Again, calculate the distance (`1`) and compare it with `d`. Since `1` is less than `3`, we do not update `d`.
   - Update `last` to `6`.
9. Seats `7`, `8`, and `9` are empty, so no changes occur during these steps.

Once we've finished iterating through the array, we consider the maximum distance from the following:

- The distance from the start of the row to the first occupied seat (`first = 1`).
- The distance from the last occupied seat to the end of the row (`len(seats) - last - 1 = 10 - 6 - 1 = 3`).
- Half the distance of the longest stretch of empty seats between two occupied seats (`d // 2 = 3 // 2 = 1`).

The maximum of these values is `3`, which is accounted for by both the distance from the last occupied seat to the end of the row and from the first seat in the row to the first occupied seat. Alex can choose either seat at the start or the end of the row for the maximum distance to the closest person, hence we return `3` as the result.

Here's the walkthrough reflected in the provided code:

```
class Solution:
    def maxDistToClosest(self, seats: List[int]) -> int:
        first = last = None
        d = 0
        for i, c in enumerate(seats):
            if c == 1:
                if last is not None:
                    d = max(d, i - last)  # At index 4 and index 1, d becomes 3
                if first is None:
                    first = i
                last = i  # At index 1, last becomes 1
                          # Updates at indices 3, 4, and 6
        return max(first, len(seats) - last - 1, d // 2)  # Returns max(1, 3, 1) which is 3
```

By iterating through the array only once and using a simple set of variables to keep track, we find the maximum distance where Alex can sit with a time complexity of O(n) and a space complexity of O(1).

Python Solution

```
class Solution:
    def maxDistToClosest(self, seats: List[int]) -> int:
        # Initialize the first occupied seat index and last occupied seat index to None
        first_occupied_seat_index = None
        last_occupied_seat_index = None
        max_distance = 0  # Initialize the maximum distance to 0

        # Iterate over each seat and its index in the seats list
        for i, seat in enumerate(seats):
            # Check if the seat is occupied
            if seat == 1:
                # If there was a previously occupied seat, calculate the distance between
                # the current occupied seat and the last occupied seat and update max_distance
                if last_occupied_seat_index is not None:
                    distance_between_seats = i - last_occupied_seat_index
                    max_distance = max(max_distance, distance_between_seats)

                # If this is the first occupied seat we've found, store its index
                if first_occupied_seat_index is None:
                    first_occupied_seat_index = i

                # Update the last occupied seat index to the current index
                last_occupied_seat_index = i

        # Calculates the maximum distance by considering the first and last seats as well.
        # If they are unoccupied, and compare it with the maximum distance found between two occupants.
        # The distance for the first and last seats is handled differently, hence they are compared separately.
        # max_distance // 2 is used for the middle section since a person can sit in the middle of two occupied seats.
        return max(
            first_occupied_seat_index,  # Max distance from the start to the first occupied seat
            len(seats) - last_occupied_seat_index - 1,  # Max distance from the last occupied seat to the end
            max_distance // 2  # Max distance between two occupied seats divided by 2
        )

# Note: The provided code assumes that 'List' is properly imported from the 'typing' module.
# If the code is run as-is, make sure to add: from typing import List
```

Java Solution

```
class Solution {
    public int maxDistToClosest(int[] seats) {
        // Initialize variables for the first occupied seat and last occupied seat.
        int firstOccupied = -1, lastOccupied = -1;

        // Initialize the maximum distance to the closest person to 0.
        int maxDistance = 0;

        // Get the number of seats.
        int seatCount = seats.length;

        // Iterate over each seat.
        for (int i = 0; i < seatCount; ++i) {

            // Check if the current seat is occupied.
            if (seats[i] == 1) {

                // Update the maximum distance if there is a last occupied seat to compare with.
                if (lastOccupied != -1) {
                    maxDistance = Math.max(maxDistance, i - lastOccupied);
                }

                // If it's the first occupied seat encountered, record its index.
                if (firstOccupied == -1) {
                    firstOccupied = i;
                }

                // Update the last occupied seat index.
                lastOccupied = i;
            }
        }

        // Calculate the distance from the first occupied seat and the last seat.
        int distanceFromStart = firstOccupied;
        int distanceFromEnd = seatCount - lastOccupied - 1;

        // Get the maximum distance from start, end, and between occupied seats.
        // The maximum distance in between seats is divided by 2 since it's between two people.
        return Math.max(maxDistance / 2, Math.max(distanceFromStart, distanceFromEnd));
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm> // For using max()

using std::vector;
using std::max;

class Solution {
public:
    // Finds the maximal distance to the closest person.
    int maxDistToClosest(vector<int>& seats) {
        int firstOccupied = -1;  // To mark the position of the first occupied seat.
        int lastOccupied = -1;   // To mark the position of the last occupied seat so far.
        int maxDistance = 0;     // To keep track of the maximum distance to the closest person so far.
        int seatCount = seats.size(); // Total number of seats.

        // Iterate over the seats to find the maximum distance.
        for (int i = 0; i < seatCount; ++i) {
            if (seats[i] == 1) {  // When the seat is occupied
                if (lastOccupied != -1) // For all occupied seats except the first one
                    // Calculate the distance from the last occupied seat and update maxDistance if the distance is larger.
                    maxDistance = max(maxDistance, i - lastOccupied);

                // Update the first occupied seat position upon finding the first occupied seat.
                if (firstOccupied == -1)
                    firstOccupied = i;

                // Update the last occupied seat.
                lastOccupied = i;
            }
        }

        // Calculate the max distance for the edge seats and compare it with the previously found max.
        // If an edge seat is the farthest from any occupied seat, update maxDistance accordingly.
        // The distance from an edge to the first/last occupied seat is directly the index difference.
        int edgeMaxDistance = max(firstOccupied, seatCount - lastOccupied - 1);

        // Return the maximum distance among the adjacent seats and the edge seats.
        return max(maxDistance / 2, edgeMaxDistance);
    }
};
```

Typescript Solution

```
/**
 * Finds the maximum distance to the closest person in a row of seats.
 * Seats are represented by an array where 1 is a person sitting and 0 is an empty seat.
 * There is at least one empty seat and at least one person sitting.
 * @returns {number} - The maximum distance to the closest person.
 */
function maxDistToClosest(seats: number[]): number {
    let firstOccupiedSeatIndex = -1; // Index of the first occupied seat
    let lastOccupiedSeatIndex = -1;  // Index of the last occupied seat
    let maxDistance = 0;             // Maximum distance to the closest person
    let totalSeats = seats.length;   // Total number of seats

    for (let i = 0; i < totalSeats; ++i) {
        if (seats[i] === 1) {
            if (lastOccupiedSeatIndex !== -1) {
                // Update max distance calculating the half distance between two occupied seats
                maxDistance = Math.max(maxDistance, i - lastOccupiedSeatIndex);
            }
            if (firstOccupiedSeatIndex === -1) {
                // Update the index of the first occupied seat found
                firstOccupiedSeatIndex = i;
            }
            // Update the index of the last occupied seat found
            lastOccupiedSeatIndex = i;
        }
    }
    // Calculate the max distance considering the edge seats and the max distance
    // found between two occupied seats, where distance is halved since you sit in-between.
    return Math.max(
        firstOccupiedSeatIndex,                     // Distance from the first seat to the first person
        totalSeats - lastOccupiedSeatIndex - 1,     // Distance from the last person to the last seat
        Math.floor(maxDistance / 2)                 // Max distance divided by 2 for the middle seat between two people
    );
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `seats` list. This is derived from the fact that there is a single for loop that iterates over each element of the `seats` list exactly once to find the maximum distance to the closest person.

Space Complexity

The space complexity of the code is `O(1)` which means it uses constant extra space. This is because the variables `first`, `last`, and `d` are used to keep track of positions and distance within the algorithm, regardless of the input size. The list itself is not copied or modified, and no additional data structures are utilized that grow with the input size.