Monday Feb 28th

Maximize Distance to Closest Person

Solution

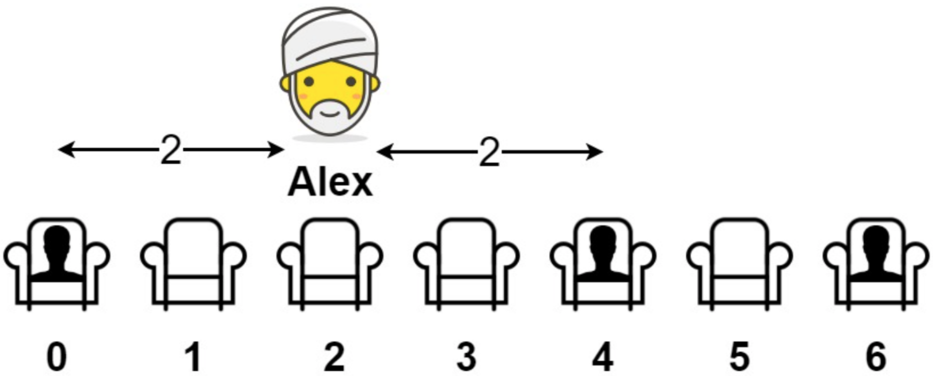
You are given an array representing a row of `seats` where `seats[i] = 1` represents a person sitting in the `i`th seat, and `seats[i] = 0` represents that the `i`th seat is empty **(0-indexed)**.

There is at least one empty seat, and at least one person sitting.

Alex wants to sit in the seat such that the distance between him and the closest person to him is maximized.

Return *that maximum distance to the closest person*.

Example 1:




```
Input: seats = [1,0,0,0,1,0,1]
Output: 2
Explanation:
If Alex sits in the second open seat (i.e. seats[2]), then the closest person has distance 2
If Alex sits in any other open seat, the closest person has distance 1.
Thus, the maximum distance to the closest person is 2.
```

Example 2:

```
Input: seats = [1,0,0,0]
Output: 3
Explanation:
If Alex sits in the last seat (i.e. seats[3]), the closest person is 3 seats away.
This is the maximum distance possible, so the answer is 3.
```

Example 3:

```
Input: seats = [0,1]
Output: 1
```

Constraints:

- `2 <= seats.length <=` $2 * 10^4$
- `seats[i]` is `0` or `1`.
- At least one seat is **empty**.
- At least one seat is **occupied**.

The Bruteforce Solution: $O(n^2)$

```
package leetcode;
public class Problem0228_Solution {
      public static void main(String[] args) {
            int[] seats = { 0, 0, 1 };
            System.out.println(findClosestSeat(seats, 0));
      }
      public static int maxDistToClosest(int[] seats) {
            int maxDistance = 0;
            // Edge case: When there are two seats and one of them is empty
            if (seats.length == 2) {
                  maxDistance = 1;
            } // When there are more than two seats
            else {
                  for (int i = 0; i < seats.length; i++) {
                        // When the seat ith is empty
                        if (seats[i] == 0) {
                              // Check the closest neightbour
                              int localMaxDistance = findClosestSeat(seats, i);
                              maxDistance = Math.max(maxDistance, localMaxDistance);
                        }

                  }
            }
            return maxDistance;
      }
      /*
       * This method takes in an array representing a row of seats, an index i
       * indicating an empty seat, and returns the closest distance between the
       * current seat and an occupied seat. This method assumes there are more
       * than two seats, at least one seat is empty, and at least one seat is
       * occupied.
       */
      public static int findClosestSeat(int[] seats, int i) {
            int closestDist = 0;
            int seatCount = seats.length;
            for (int j = i - 1; j >= 0; j--) {// Left pointer searches to the left
                  if (seats[j] == 1) {
                        closestDist = i - j;
                        break;
                  }
            }
            for (int j = i + 1; j < seatCount; j++) {// Right pointer searches to the right
```

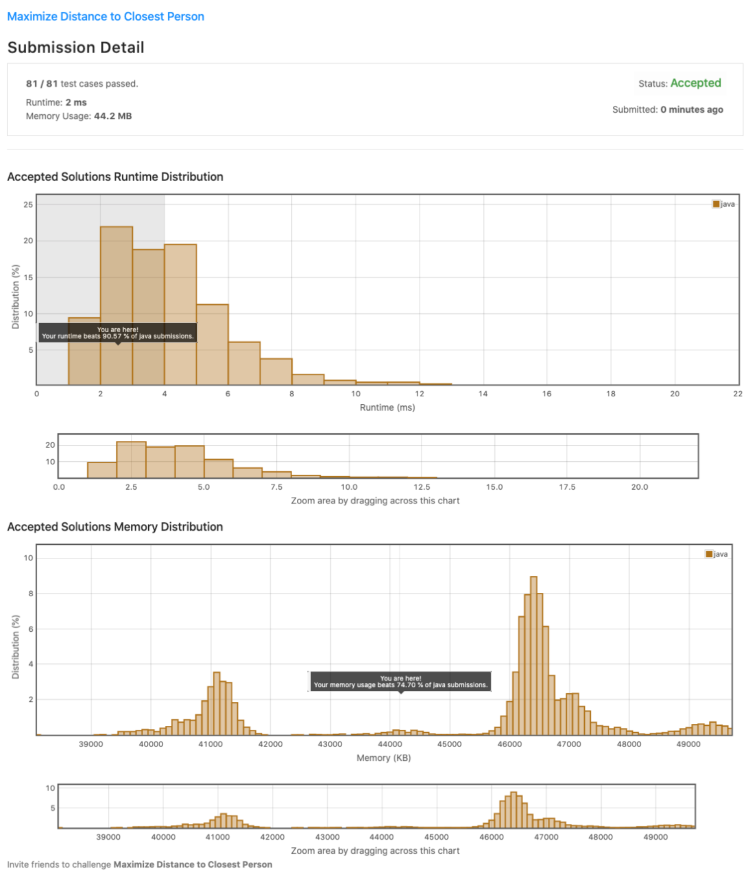
Commented [SL1]: Time complexity: O(n^2)

Commented [SL2]: Time complexity: O(1)

Commented [SL3]: Time complexity: O(n*n)

Commented [SL4]: Time complexity: O(n)

```java
            if (seats[j] == 1) {
                if (closestDist == 0) {
                    // Given the assumption, it's not possible when closetDist == 0, because we
                    //know there must be at least one occupied seat. Hence in this case, it
                  means that we are given the first seat index.
                    closestDist = j - i;
                } else {
                    closestDist = Math.min(closestDist, j - i);
                }
                break;
            }
        }
        return closestDist;
    }
}
```



The Next Array Solution: $O(n)$

The problem is reduced to finding the closest left/right distance of each empty seat.
When a seats[i] is occupied (i.e., seats[i] == 1), then the closest left/right distance is 0, because we cannot sit in that seat.
When a seats[i] is unoccupied (i.e., seats[i] == 0), then the closest left distance is left[i] = left[i-1] + 1, the closest right distance is right[i] = right[i+1] + 1;
For the leftmost seat (i.e., seats[0]), if it is unoccupied, left[0] = N;
For the rightmost seat (i.e., seats[N-1]), if it is unoccupied, right[N-1] = N.

Apparently, when seats[0] = 1 and seats[N-1] = 1, this algorithm works.
Now we need to show that the algorithm works for seats[0] = 0 or seats[N-1] = 0.

Case 1: When seats[0] = 0 and seats[N-1] = 1.
  Case 1.1 Seats = [0, 0, ..., 0,0,0, ..., 0, 1]
  Case 1.2 Seats = [0, 0, ..., 0,1,0, ..., 0, 1]
    left = [N, N+1, ..., N+n, **0**, 1, ..., (N-n-3), **0**]
    right = [n+1, n, ..., 1, **0**, (N-n-3), ..., 1, **0**]
Case 2: When seats[0] = 1 and seats[N-1] = 0.

Case 3: When seats[0] = 1 and seats[N-1] = 1.

```java
package leetcode;

import java.util.Arrays;

public class Problem0228_Solution2 {
    public static void main(String[] args) {
        int[] seats = { 0, 0, 1 };
        System.out.println(maxDistToClosest(seats));
    }

    public static int maxDistToClosest(int[] seats) {
        int N = seats.length;
        int[] left = new int[N], right = new int[N];
        Arrays.fill(left, N);
        Arrays.fill(right, N);

        for (int i = 0; i < N; ++i) {
            if (seats[i] == 1)
                left[i] = 0;
```

Commented [SL5]: To fill complete the array with a particular value N.

Commented [SL6R5]: However, I couldn't explain in an intuitive way why we should start with default N for both left and right at each position.

```
            else if (i > 0)
                left[i] = left[i - 1] + 1;
        }
        for (int i = N - 1; i >= 0; --i) {
            if (seats[i] == 1)
                right[i] = 0;
            else if (i < N - 1)
                right[i] = right[i + 1] + 1;
        }
        int ans = 0;
        for (int i = 0; i < N; ++i)
            if (seats[i] == 0)
                ans = Math.max(ans, Math.min(left[i], right[i]));
        return ans;
    }
}
```

Commented [SL7]: Time complexity: O(n)
Construct left[i]: the closest person to the left of an empty seat ith.

Commented [SL8]: Time complexity: O(n)
Construct right[i]: the closest person to the right of an empty seat ith.

Commented [SL9]: Time complexity: O(n).
The closest person to an empty seat ith is of a distance min(left[i], right[i]) away.

Example 1:

Seats = {1, 0, 0, 1, 0}

Seat Index	0	1	2	3	4
Occupancy	1	0	0	1	0

Left:

Seat Index	0	1	2	3	4
Closest distance	0	1	2	0	1

Right:

Seat Index	0	1	2	3	4
Closest distance	0	2	1	0	5

The closest distance to:

Seat Index	0	1	2	3	4
Closest distance	0	1	1	0	1

Example 2:

Seats = {1, 0, 0, 0, 0}

Seat Index	0	1	2	3	4
Occupancy	1	0	0	0	0

Left:

Seat Index	0	1	2	3	4
Closest distance	0	1	2	3	4

Right:

Seat Index	0	1	2	3	4
Closest distance	0	8	7	6	5

The closest distance to:

Seat Index	0	1	2	3	4
Closest distance	0	1	2	3	4

Example 2':

Seats = {1, 0, 0, 0, 0}

Seat Index	0	1	2	3	4
Occupancy	1	0	0	0	0

Left:

Seat Index	0	1	2	3	4
Closest distance	0	1	2	3	4

Right:

Seat Index	0	1	2	3	4
Closest distance	5	3	2	1	0

The closest distance to:

Seat Index	0	1	2	3	4
Closest distance	0	1	2	1	0

Example 3:

Seats = {0, 1, 0, 0, 0}

Seat Index	0	1	2	3	4
Occupancy	0	1	0	0	0

Left:

Seat Index	0	1	2	3	4
Closest distance	5	0	1	2	3

Right:

Seat Index	0	1	2	3	4
Closest distance	1	0	7	6	5

The closest distance to:

Seat Index	0	**1**	2	3	4
Closest distance	1	**0**	1	2	3

Example 4:

Seats = {0, 1, 0, 1, 0}

Seat Index	0	**1**	2	**3**	4
Occupancy	**0**	**1**	**0**	**1**	**0**

Left:

Seat Index	0	**1**	2	**3**	4
Closest distance	5	**0**	1	**0**	1

Right:

Seat Index	0	**1**	2	**3**	4
Closest distance	1	**0**	1	**0**	5

The closest distance to:

Seat Index	0	**1**	2	**3**	4
Closest distance	1	**0**	1	**0**	1

The Two Pointers Solution: $O(n)$

The problem is reduced to finding the maximum distance between two continuous 1 in an array, and just return half of that maximum value. We also need to consider two edge cases.

Commented [SL10]: Not sure how to work with the edge case.

```java
package leetcode;

public class Problem0228_Solution3 {
	public static void main(String[] args) {
		int[] seats = { 1, 0, 0, 0 };
		System.out.println("Result: " + maxDistToClosest(seats));
	}
	public static int maxDistToClosest(int[] seats) {
		int left = -1, maxDis = 0;
		int len = seats.length;

		for (int i = 0; i < len; i++) {
			if (seats[i] == 0)
				continue;
			if (left == -1) {
				maxDis = Math.max(maxDis, i);
			} else {
				maxDis = Math.max(maxDis, (i - left) / 2);
			}
```

```
                left = i;
            }

            if (seats[len - 1] == 0) {
                maxDis = Math.max(maxDis, len - 1 - left);
            }
            return maxDis;
        }
}
```

Example 1:

Seats = {1, 0, 0, 1, 0}

Seat Index	0	1	2	3	4
Occupancy	1	0	0	1	0

left = -1, maxDis = 0, len = 5
i = 0, left = -1
  maxDis = max(0,0) = 0
  left = 0
i = 1, continue
i = 2, continue
i = 3, left = 0
  maxDis = max(0,1) = 1
  left = 3
i = 4, continue
Because seats[4] == 0
  len - 1 – left = 5 – 1 – 3 = 1
  maxDis = max(1, 1) = 1

Example 2:

Seats = {1, 0, 0, 0, 0}

Seat Index	0	1	2	3	4
Occupancy	1	0	0	0	0

left = -1, maxDis = 0, len = 5
i = 0, left = -1
  maxDis = max(0,0) = 0
  left = 0
i = 1, continue
i = 2, continue
i = 3, continue
i = 4, continue
Because seats[4] == 0
  len - 1 – left = 5 – 1 – 0 = 4

maxDis = max(0, 4) = 4