**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 ith seat, and seats[i] = 0 represents that the ith 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:**

A picture containing text, clock

Description automatically generated

**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 \* 104
* seats[i] is 0 or 1.
* At least one seat is **empty**.
* At least one seat is **occupied**.

**Solution:**

* Approach #1: Next Array [Accepted]
* **Intuition**
* Let left[i] be the distance from seat i to the closest person sitting to the left of i. Similarly, let right[i] be the distance to the closest person sitting to the right of i. This is motivated by the idea that the closest person in seat i sits a distance min(left[i], right[i]) away.
* **Algorithm**
* To construct left[i], notice it is either left[i-1] + 1 if the seat is empty, or 0 if it is full. right[i] is constructed in a similar way.

class Solution {

public 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;

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;

}

}

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of seats.
* Space Complexity: O(N)*O*(*N*), the space used by left and right.

#### Approach #2: Two Pointer [Accepted]

**Intuition**

As we iterate through seats, we'll update the closest person sitting to our left, and closest person sitting to our right.

**Algorithm**

Keep track of prev, the filled seat at or to the left of i, and future, the filled seat at or to the right of i.

Then at seat i, the closest person is min(i - prev, future - i), with one exception. i - prev should be considered infinite if there is no person to the left of seat i, and similarly future - i is infinite if there is no one to the right of seat i.

class Solution {

public int maxDistToClosest(int[] seats) {

int N = seats.length;

int prev = -1, future = 0;

int ans = 0;

for (int i = 0; i < N; ++i) {

if (seats[i] == 1) {

prev = i;

} else {

while (future < N && seats[future] == 0 || future < i)

future++;

int left = prev == -1 ? N : i - prev;

int right = future == N ? N : future - i;

ans = Math.max(ans, Math.min(left, right));

}

}

return ans;

}

}

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of seats.
* Space Complexity: O(1)*O*(1).

#### Approach #3: Group by Zero [Accepted]

**Intuition**

In a group of K adjacent empty seats between two people, the answer is (K+1) / 2.

**Algorithm**

For each group of K empty seats between two people, we can take into account the candidate answer (K+1) / 2.

For groups of empty seats between the edge of the row and one other person, the answer is K, and we should take into account those answers too.

class Solution {

public int maxDistToClosest(int[] seats) {

int N = seats.length;

int K = 0; //current longest group of empty seats

int ans = 0;

for (int i = 0; i < N; ++i) {

if (seats[i] == 1) {

K = 0;

} else {

K++;

ans = Math.max(ans, (K + 1) / 2);

}

}

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

ans = Math.max(ans, i);

break;

}

for (int i = N-1; i >= 0; --i) if (seats[i] == 1) {

ans = Math.max(ans, N - 1 - i);

break;

}

return ans;

}

}

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of seats.
* Space Complexity: O(1)*O*(1).