【贪心】
社交距离
题目描述与示例

题目描述

疫情期间，需要大家保证一定的社交距离，公司组织开交流会议，座位有一排共 `N` 个座位，编号分别为 `[0, N-1]`，要求员工一个接着一个进入会议室，并且可以在任何时候离开会议室。

满足：每当一个员工进入时，需要坐到最大社交距离的座位（例如：位置 `A` 与左右有员工落座的位置距离分别为 `2` 和 `2`，位置 `B` 与左右有员工落座的位置距离分别为 `2` 和 `3`，影响因素都为 `2` 个位置，则认为座位 `A` 和 `B` 与左右位置的社交距离是一样的)；如果有多个这样的座位，则坐到索引最小的那个座位。

输入描述

会议室座位总数 `seatNum`，`(1 ≤ seatNums ≤ 500)`

员工的进出顺序 `seatOrLeave` 数组，元素值为 `1`：表示进场；元素值为负数，表示出场（特殊：位置 `0` 的员工不会离开），例如 `-4` 表示坐在位置 `4` 的员工离开（保证有员工坐在该座位上）

输出描述

最后进来员工，他会坐在第几个位置，如果位置已满，则输出 `-1`

示例

输入

```
10
[1, 1, 1, 1, -4, 1]
```

输出

```
1 5
```

说明

`seat->0`，坐在任何位置都行，但是要给他安排索引最小的位置，也就是座位 `0`

`seat->9`，要和旁边的人距离最远，也就是座位 `9`。

`seat->4`，位置 `4` 与 `0` 和 `9` 的距离为(`4` 和 `5`)，位置 `5` 与 `0` 和 `9` 的距离(`5` 和 `4`)，所以位置 `4` 和 `5` 都是可以选择的座位，按照要求需素引最小的那个座位，也就是作为 `4`

`seat->2`，位置 `2` 与 `0` 和 `4` 的距离为(`2` 和 `2`),位置 `6` 与 `4` 和 `9` 的距离(`2` 和 `3`)，位置 `7` 与 `4` 和 `9` 的距离(`3` 和 `2`)，影响因素都为 `2` 个位置，按照要求需素引最小的那个座位，也就是座位 `2`。

`leave(4)`， `4` 号座位的员工离开。

`seat->5`，员工最后坐在 `5` 号座位上。

解题思路

操作数组的遍历

本题涉及到**动态的模拟过程**，即对一个长度为 `seatNum` 的数组，不断进行元素添加和删除（即落座和离开）。

我们把落座和离开定义为**操作**，用数组 `operations` 储存。

我们可以构建一个长度为 `seatNum` 的数组 `seats`，表示整个会议室的落座情况。其中

- `seats[i] == 0` 表示第 `i` 个位置为空，没有人坐下
- `seats[i] == 1` 表示第 `i` 个位置不为空，已经有人坐下

题目已经说明，位置 `0` 的员工落座之后不会离开，且 `seats` 数组一开始为空，每一次离开操作时，座位上必然有人，故操作数组的第一个操作，必然是第一个员工落座到 `seats` 数组中位置 `0`。

即一定存在 `operations[0] = 1` 成立。即 `seats` 的初始化为

```
1 seats = [0] * n
2 seats[0] = 1
```

另外，题目要求输出的内容是：**最后一个进场的人落座的位置**。

显然最后一次落座发生之后，如果后面还发生了离场，也不会影响最后一个进场的人落座的位置。故我们遍历操作数组，只需要遍历到最后一次落座发生即可。

我们可以逆序遍历操作数组 `operations` ，找到最后一个 `operations[i] == 1` 的位置 `i` ，记录为 `last_in_operation_idx` 。代码为

```
for i in range(len(operations)-1, -1, -1):
    if operations[i] > 0:
        last_in_operation_idx = i
        break
```

结合 `operations[0] = 1` 一定成立这件事情。

显然，如果整个操作过程**有且只有第一个人进入了会议室，那么这个人也是最后一个进入会议室的人**，应该输出 `0` 作为这第一个人也是最后一个人的落座位置 `0` 。对于这种情况我们需要做一个特殊判断，即

```
if last_in_operation_idx == 0:
    print(0)
```

排除了这种特殊情况之后，我们需要根据操作数组 `operations[1:last_in_operation_idx+1]` 的所有操作，来修改座位数组 `seats` 。

我们需要判断

- `i` 是否为 `last_in_operation_idx` 。
  - 若是，此时是最后一个人落座，需要输出答案
  - 若不是，则考虑此时是落座还是离开
    - 若 `operations[i] == 1` ，则是落座。需要找到落座位置
    - 若 `operations[i] < 0` ，则是离开。需要令对应位置的人离开

故整体的框架为

```python
for i in range(1, last_in_operation_idx+1):
    # 落座，且是最后一个人
    if i == last_in_operation_idx:
        pass
    op = operations[i]
    # 落座，但不是最后一个人
    if op > 0:
        pass
    # 离开
    else:
        pass
```

元素添加（落座）

注意到**添加元素的过程**，总是会选择距离左右两边已存在元素最远的那个下标，该子过程和题目【贪心】2023C-停车找车位是完全一致的。

故对于每一次有**新的人进入会议室落座**，我们可以构建如下的函数`update_seats_in(seats)`，来贪心地找到每一次应该落座的位置

```python
def update_seats_in(seats):
    for i, num in enumerate(seats[::-1]):
        if num == 1:
            right = n-1-i
            break

    ans_idx = n-1
    max_dis = n-1-right

    pre = 0
    for i, num in enumerate(seats[1:right + 1], 1):
        if num == 1:
            cur_dis = (i - pre) // 2
            if max_dis < cur_dis:
                max_dis = cur_dis
                ans_idx = pre + (i - pre) // 2
            pre = i

    return ans_idx if max_dis > 0 else -1
```

有几个需要注意的点：

1. 题目明确说明，位置 `0` 的员工不会离开，这意味
   - 在第一个员工落座之后（坐到位置 `0` 之后），`seats` 数组的左端点 `seats[0]` 一定为 `1`。
   - 在【贪心】2023C-停车找车位 中关于左端点 `left` 的计算就可以不用考虑了。
   - 直接设置 `left` 为 `0`，表示数组中最左边的 `1`，一定位于 `seats` 数组中 `0` 的位置。
   - 初始化 `pre` 的时候，也设置为 `0` 即可
2. 关于右端点 `right` 的计算仍然不能省略，需要通过**逆序遍历**，找到 `seats` 数组中最右边的第一个 `1`
3. 遍历过程中，除了储存全局的最大距离 `max_dis`，还需要同时储存这个最大距离对应的落座位置 `ans_idx`，且这个函数需要返回的正是 `ans_idx`。
4. 虽然题目没有明确说明，**如果会议室人满了，继续往会议室中加人应该如何处理**
   - 但输出描述中有一句“**如果位置已满，则输出** `-1`”
   - 因此这种情况发生时，考虑数组 `seats` 仍然为满的状态，不发生任何变化，返回的 `ans_idx` 为 `-1`
   - 人满的情况可以通过判断全局最大距离 `max_dis` 是否大于 `0` 来判断
   - 事实证明，这种考虑是可以和题目用例是对应得上的

每次调用函数 `update_seats_in(seats)` 之后，会返回一个下标 `idx`。若

- 此时是最后一个人落座，即 `i = last_in_operation_idx`，那么则直接输出 `idx`
- 此时不是最后一个人落座，则判断 `idx`。若
  - `idx = -1`，说明本次落座之前会议室人满，直接跳过
  - `idx != -1`，说明本次落座位置为 `idx`，将 `seats[idx]` 从 `0` 改为 `1`

整体代码为

```python
for i in range(1, last_in_operation_idx+1):
    # 落座，且是最后一个人
    if i == last_in_operation_idx:
        ans = update_seats_in(seats)
        print(ans)
        break
    op = operations[i]
    # 落座，但不是最后一个人
    if op > 0:
```

```
        idx = update_seats_in(seats)
        if idx != -1:
            seats[idx] = 1
    # 离开
    else:
        pass
```

元素删除（离开）

这个就很简单了，在遍历 `operations` 的过程中若出现 `operations[i] < 0`，则发生了离开。

离开人的位置为 `idx = -operations[i]`。

只需要把 `seat[idx]` 从 `1` 改为 `0` 即可。即

```
for i in range(1, last_in_operation_idx+1):
    # 落座，且是最后一个人
    if i == last_in_operation_idx:
        pass
    op = operations[i]
    # 落座，但不是最后一个人
    if op > 0:
        pass
    # 离开
    else:
        idx = -op
        seats[idx] = 0
```

代码

Python

```
# 题目：2024D-社交距离
# 分值：200
# 作者：许老师-闭着眼睛学数理化
# 算法：贪心
```

```python
# 代码看不懂的地方，请直接在群上提问


# 有人进场时，更新数组seats的函数
def update_seats_in(seats):
    # 题目明确了位置0一定不会离开，即位置0始终会被人占据
    # 因此可以不用判断最左边的情况
    # 即不用判断left了，直接取left为0即可
    # 寻找seats最右边的1对应的下标right
    for i, num in enumerate(seats[::-1]):
        if num == 1:
            right = n-1-i
            break

    # 初始化答案，落座在最右边，即n-1
    ans_idx = n-1
    # 初始化当前seats中能够达到的全局的最大距离
    max_dis = n-1-right

    # 查看那些两边都有1的位置，即需要判断任意两个1之间的距离
    # pre表示找到区间中，上一个1的位置，初始化为0
    pre = 0
    # 遍历剩下的区间seats[1:right+1]
    for i, num in enumerate(seats[1:right + 1], 1):
        # 找到一个1，计算i和pre之间的距离并取半，
        # 即为停在i和pre正中间位置距离两边最近车辆的最远距离
        if num == 1:
            # 计算当前i和pre之间能够找到的最大距离
            cur_dis = (i - pre) // 2
            # 若当前最大距离，大于全局的最大距离，那么
            if max_dis < cur_dis:
                # 更新全局的最大距离
                max_dis = cur_dis
                # 更新应该落座的位置
                ans_idx = pre + (i - pre) // 2
            # 找到了一个1之后，当前i位置的1变成了下一个1的前一个1，pre修改为i
            pre = i

    # 返回落座的位置
    # 可能存在一种极端情况，即seats本身已经全为1
    # 那么max_dis的值将始终为0，此时返回-1而非落座的下标ans_idx
    return ans_idx if max_dis > 0 else -1

# 输入会议室大小
n = int(input())
# 输入一系列操作，其中1表示有有人进入，负数表示有人离开
# 注意输入包含了
```

```python
operations = list(map(int, input()[1:-1].split(",")))

# 构建长度为n的数组seats，表示会议室的情况
seats = [0] * n
# operations[0]必然为1，即一开始一定有人会进场
# 这个人将坐在seats[0]的位置，且之后都不会离开
seats[0] = 1


# 逆序遍历操作数组operations，找到最后一个人进入会议室对应的操作的下标
# 记为last_in_operation_idx
# 显然last_in_operation_idx之后的离开操作（无论有多少人离开），都无需再考虑
for i in range(len(operations)-1, -1, -1):
    if operations[i] > 0:
        last_in_operation_idx = i
        break

# 一种特殊情况：
# 只有最开始进来了一个人，后面都没有人再进入
# 即第一个人就是最后一个人
# 直接输出0
if last_in_operation_idx == 0:
    print(0)
# 否则，遍历所有剩下的操作
else:
    for i in range(1, last_in_operation_idx+1):
        # 如果是最后一个人进入会议室，调用update_seats_in()函数
        # 函数返回的结果，即为最后一个人进入的位置，输出该结果即为答案
        if i == last_in_operation_idx:
            ans = update_seats_in(seats)
            print(ans)
            break
        op = operations[i]
        # 有人进入会议室
        if op > 0:
            # 调用update_seats_in()函数
            # 计算得到落座的位置idx，修改seats[idx]为1
            #  （前提是有座位可以落座，即idx不为-1）
            idx = update_seats_in(seats)
            if idx != -1:
                seats[idx] = 1
        # 有人离开会议室
        else:
            # -op为离开的下标idx，修改seats[idx]为0
            idx = -op
            seats[idx] = 0
```

Java

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {
    public static int updateSeatsIn(List<Integer> seats) {
        int n = seats.size();
        int right = n - 1;
        for (int i = n - 1; i >= 0; i--) {
            if (seats.get(i) == 1) {
                right = i;
                break;
            }
        }

        int ansIdx = n - 1;
        int maxDis = n - 1 - right;
        int pre = 0;
        for (int i = 1; i <= right; i++) {
            if (seats.get(i) == 1) {
                int curDis = (i - pre) / 2;
                if (maxDis < curDis) {
                    maxDis = curDis;
                    ansIdx = pre + (i - pre) / 2;
                }
                pre = i;
            }
        }
        return maxDis > 0 ? ansIdx : -1;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        scanner.nextLine(); // Consume newline
        String line = scanner.nextLine();
        String[] operationsStr = line.substring(1, line.length() - 1).split(",
");
        List<Integer> operations = new ArrayList<>();
        for (String op : operationsStr) {
            operations.add(Integer.parseInt(op));
        }

        List<Integer> seats = new ArrayList<>(n);
```

```
        for (int i = 0; i < n; i++) {
            seats.add(0);
        }
        seats.set(0, 1);

        int lastInOperationIdx = -1;
        for (int i = operations.size() - 1; i >= 0; i--) {
            if (operations.get(i) > 0) {
                lastInOperationIdx = i;
                break;
            }
        }

        if (lastInOperationIdx == 0) {
            System.out.println(0);
        } else {
            for (int i = 1; i <= lastInOperationIdx; i++) {
                if (i == lastInOperationIdx) {
                    int ans = updateSeatsIn(seats);
                    System.out.println(ans);
                    break;
                }
                int op = operations.get(i);
                if (op > 0) {
                    int idx = updateSeatsIn(seats);
                    if (idx != -1) {
                        seats.set(idx, 1);
                    }
                } else {
                    int idx = -op;
                    seats.set(idx, 0);
                }
            }
        }
    }
}

```

C++

```
#include <iostream>
#include <vector>

using namespace std;

```

```cpp
int updateSeatsIn(vector<int>& seats) {
    int n = seats.size();
    int right = n - 1;
    for (int i = n - 1; i >= 0; i--) {
        if (seats[i] == 1) {
            right = i;
            break;
        }
    }

    int ansIdx = n - 1;
    int maxDis = n - 1 - right;
    int pre = 0;
    for (int i = 1; i <= right; i++) {
        if (seats[i] == 1) {
            int curDis = (i - pre) / 2;
            if (maxDis < curDis) {
                maxDis = curDis;
                ansIdx = pre + (i - pre) / 2;
            }
            pre = i;
        }
    }
    return maxDis > 0 ? ansIdx : -1;
}

int main() {
    int n;
    cin >> n;
    cin.ignore(); // Consume newline
    string line;
    getline(cin, line);
    vector<int> operations;
    size_t start = 1;
    size_t end;
    while ((end = line.find(", ", start)) != string::npos) {
        operations.push_back(stoi(line.substr(start, end - start)));
        start = end + 1;
    }
    operations.push_back(stoi(line.substr(start)));

    vector<int> seats(n);
    seats[0] = 1;

    int lastInOperationIdx = -1;
    for (int i = operations.size() - 1; i >= 0; i--) {
        if (operations[i] > 0) {
```

```
53              lastInOperationIdx = i;
54              break;
55          }
56      }
57
58      if (lastInOperationIdx == 0) {
59          cout << 0 << endl;
60      } else {
61          for (int i = 1; i <= lastInOperationIdx; i++) {
62              if (i == lastInOperationIdx) {
63                  int ans = updateSeatsIn(seats);
64                  cout << ans << endl;
65                  break;
66              }
67              int op = operations[i];
68              if (op > 0) {
69                  int idx = updateSeatsIn(seats);
70                  if (idx != -1) {
71                      seats[idx] = 1;
72                  }
73              } else {
74                  int idx = -op;
75                  seats[idx] = 0;
76              }
77          }
78      }
79
80      return 0;
81  }
82
```

时空复杂度

时间复杂度：`O(NM)`。操作数组`operations`的长度为`M`，每一次`update_seats_in(seats)`的操作需要`O(N)`的时间复杂度。

空间复杂度：`O(N)`。`seats`数组所占空间为`O(N)`。