

## 1207. 独一无二的出现次数（简单）

### 1. 题目描述

给你一个整数数组 `arr`，请你帮忙统计数组中每个数的出现次数。

如果每个数的出现次数都是独一无二的，就返回 `true`；否则返回 `false`。

#### 示例 1：

输入：arr = [1,2,2,1,1,3]

输出：true

解释：在该数组中，1 出现了 3 次，2 出现了 2 次，3 只出现了 1 次。没有两个数的出现次数相同。

#### 示例 2：

输入：arr = [1,2]

输出：false

#### 示例 3：

输入：arr = [-3,0,1,-3,1,1,1,-3,10,0]

输出：true

#### 提示：

- 1 <= arr.length <= 1000
- 1000 <= arr[i] <= 1000

### 2. 简单实现

```
class Solution {
public:
    bool uniqueOccurrences(vector<int>& arr) {
        vector<int> v = vector<int>(2001, 0);
        for(int i = 0; i < arr.size(); i++)
            v[arr[i]+1000]++;

        unordered_set<int> s;
        for(int i = 0; i < v.size(); i++)
            if(v[i] > 0){
                if(s.count(v[i]) == 0)
                    s.insert(v[i]);
                else
                    return false;
            }
        return true;
    }
};
```

## 1208. 尽可能使字符串相等 (中等)

### 1. 题目描述

给你两个长度相同的字符串，`s` 和 `t`。

将 `s` 中的第 `i` 个字符变到 `t` 中的第 `i` 个字符需要  $|s[i] - t[i]|$  的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。

用于变更字符串的最大预算是 `maxCost`。在转化字符串时，总开销应当小于等于该预算，这也意味着字符串的转化可能是不完全的。

如果你可以将 `s` 的子字符串转化为它在 `t` 中对应的子字符串，则返回可以转化的最大长度。

如果 `s` 中没有子字符串可以转化成 `t` 中对应的子字符串，则返回 0。

#### 示例 1:

输入: `s = "abcd", t = "bcdf", cost = 3`  
输出: 3  
解释: `s` 中的 "abc" 可以变为 "bcd"。开销为 3，所以最大长度为 3。

#### 示例 2:

输入: `s = "abcd", t = "cdef", cost = 3`  
输出: 1  
解释: `s` 中的任一字符要想变成 `t` 中对应的字符，其开销都是 2。因此，最大长度为 1。

#### 示例 3:

输入: `s = "abcd", t = "acde", cost = 0`  
输出: 1  
解释: 你无法作出任何改动，所以最大长度为 1。

#### 提示:

- `1 <= s.length, t.length <= 10^5`
- `0 <= maxCost <= 10^6`
- `s` 和 `t` 都只含小写英文字母。

### 2. 简单实现

相当于统计某个数组（索引存储  $\text{abs}(s[i] - t[i])$ ）中数字总和不超过 `maxCost` 的最大子数组长度，因此用滑动窗口实现即可

```
class Solution {
public:
    int equalSubstring(string s, string t, int maxCost) {
        int len = s.size();
        int l = 0, r = 0, temp_cost = 0;
        int ans = 0;
        while(r < len && l < len){
            while(r < len && temp_cost + abs(s[r] - t[r]) <= maxCost){
                temp_cost += abs(s[r] - t[r]);
            }
            ans = max(ans, r - l);
            temp_cost -= abs(s[l] - t[l]);
            l++;
        }
        return ans;
    }
};
```

```

        r++;
    }
    ans = max(ans, r - 1);
    temp_cost -= abs(s[l] - t[l]);
    l++;
}
return ans;
}
};

```

## 1209. 删除字符串中的所有相邻重复项 II (中等)

### 1. 题目描述

给你一个字符串 `s`，「`k` 倍重复项删除操作」将会从 `s` 中选择 `k` 个相邻且相等的字母，并删除它们，使被删去的字符串的左侧和右侧连在一起。

你需要对 `s` 重复进行无限次这样的删除操作，直到无法继续为止。

在执行完所有删除操作后，返回最终得到的字符串。

本题答案保证唯一。

#### 示例 1:

输入: `s = "abcd"`, `k = 2`

输出: `"abcd"`

解释: 没有要删除的内容。

#### 示例 2:

输入: `s = "deeedbbcccbdaa"`, `k = 3`

输出: `"aa"`

解释:

先删除 `"eee"` 和 `"ccc"`, 得到 `"ddbbbdaa"`

再删除 `"bbb"`, 得到 `"dddaa"`

最后删除 `"ddd"`, 得到 `"aa"`

#### 示例 3:

输入: `s = "pbbcggtttciippooaaais"`, `k = 2`

输出: `"ps"`

#### 提示:

- `1 <= s.length <= 10^5`
- `2 <= k <= 10^4`
- `s` 中只含有小写英文字母。

### 2. 简单实现

用栈，每满足`k`个就消掉

```

class Solution {
public:
    string removeDuplicates(string s, int k) {
        stack<pair<char, int>> cache; // 每个字符的数量
        char temp_ch = s[0]; // 当前字符
        int temp_len = 1; // 奇数
        int idx = 1; // 遍历到的索引
        while(idx < s.size()){
            if(s[idx] == temp_ch){ // 相同, 计数加1
                temp_len++;
                idx++;
            }
            else{ // 不同, 处理
                // 旧字符处理
                temp_len %= k;
                if(temp_len > 0) // 入栈
                    cache.push(make_pair(temp_ch, temp_len));
                // 新字符计数
                temp_ch = s[idx];
                temp_len = 1;
                if(!cache.empty() && temp_ch == cache.top().first){
                    // 新字符可以与当前栈顶字符合并计数
                    temp_len += cache.top().second;
                    cache.pop();
                }
                idx++;
            }
        }
        // 最新的字符未在循环中处理
        temp_len %= k;
        if(temp_len > 0)
            cache.push(make_pair(temp_ch, temp_len));
        // 栈内剩下的就是答案
        string ans = "";
        while(!cache.empty()){
            string cur = "";
            char ch = cache.top().first;
            int n = cache.top().second;
            cache.pop();
            while(n--){
                cur += ch;
            }
            ans = cur + ans;
        }
        return ans;
    }
};

```

### 3. 自我改进

生成ans时从栈生成需要从后往前加, 这对于字符串很浪费时间, 故可以把栈换成双向队列, 处理字符串时用队尾, 生成答案时用队头

```

class Solution {

```

```

public:
    string removeDuplicates(string s, int k) {
        deque<pair<char, int>> cache;
        char temp_ch = s[0];
        int temp_len = 1;
        int idx = 1;
        while(idx < s.size()){
            if(s[idx] == temp_ch){
                temp_len++;
                idx++;
            }
            else{
                temp_len %= k;
                if(temp_len > 0)
                    cache.push_back(make_pair(temp_ch, temp_len));
                temp_ch = s[idx];
                temp_len = 1;
                if(!cache.empty() && temp_ch == cache.back().first){
                    temp_len += cache.back().second;
                    cache.pop_back();
                }
                idx++;
            }
        }
        temp_len %= k;
        if(temp_len > 0)
            cache.push_back(make_pair(temp_ch, temp_len));
        string ans = "";
        for(auto it = cache.begin(); it != cache.end(); it++){
            string cur = "";
            char ch = cache.front().first;
            int n = cache.front().second;
            cache.pop_front();
            while(n--){
                ans += ch;
            }
        }
        return ans;
    }
};

```

## 1210. 穿过迷宫的最少移动次数（困难）

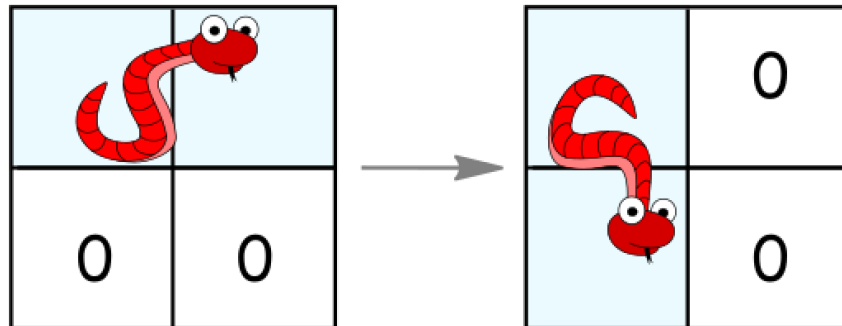
### 1. 题目描述

你还记得那条风靡全球的贪吃蛇吗？

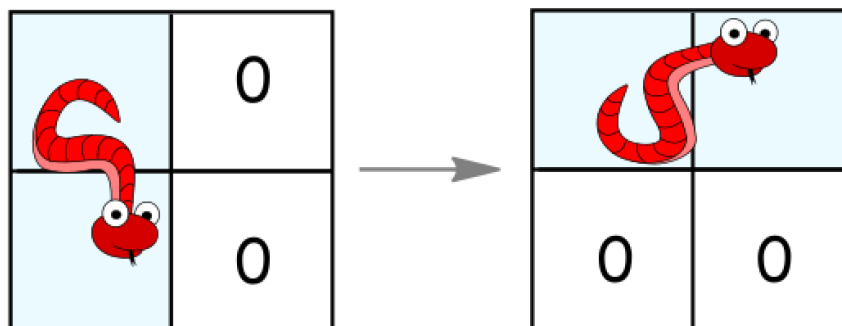
我们在一个  $n \times n$  的网格上构建了新的迷宫地图，蛇的长度为 2，也就是说它会占去两个单元格。蛇会从左上角  $(0, 0)$  和  $(0, 1)$  开始移动。我们用 0 表示空单元格，用 1 表示障碍物。蛇需要移动到迷宫的右下角  $(n-1, n-2)$  和  $(n-1, n-1)$ 。

每次移动，蛇可以这样走：

- 如果没有障碍，则向右移动一个单元格。并仍然保持身体的水平 / 竖直状态。
- 如果没有障碍，则向下移动一个单元格。并仍然保持身体的水平 / 竖直状态。
- 如果它处于水平状态并且其下面的两个单元都是空的，就顺时针旋转 90 度。蛇从  $((r, c), (r, c+1))$  移动到  $((r, c), (r+1, c))$ 。



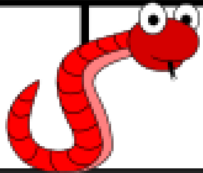
- 如果它处于竖直状态并且其右面的两个单元都是空的，就逆时针旋转 90 度。蛇从  $((r, c), (r+1, c))$  移动到  $((r, c), (r, c+1))$ 。



返回蛇抵达目的地所需的最少移动次数。

如果无法到达目的地，请返回 -1。

**示例 1:**

START					
		0	0	0	1
1	1	0	0	1	0
0	0	0	0	1	1
0	0	1	0	1	0
0	1	1	0	0	0
0	1	1	0	0	0
		FINISH			

输入: grid = [[0,0,0,0,0,1],  
               [1,1,0,0,1,0],  
               [0,0,0,0,1,1],  
               [0,0,1,0,1,0],  
               [0,1,1,0,0,0],  
               [0,1,1,0,0,0]]

输出: 11

解释:

一种可能的解决方案是 [右, 右, 顺时针旋转, 右, 下, 下, 下, 下, 逆时针旋转, 右, 下]。

示例 2:

输入: grid = [[0,0,1,1,1,1],  
              [0,0,0,0,1,1],  
              [1,1,0,0,0,1],  
              [1,1,1,0,0,1],  
              [1,1,1,0,0,1],  
              [1,1,1,0,0,0]]

输出: 9

### 提示:

- `2 <= n <= 100`
- `0 <= grid[i][j] <= 1`
- 蛇保证从空单元格开始出发。

### 2. 简单实现

BFS

```
class Solution {
public:
    int minimumMoves(vector<vector<int>>& grid) {
        int n = grid.size();
        if(grid[n-1][n-2] == 1 || grid[n-1][n-1] == 1)
            return -1;
        if(n == 2)
            return 1;
        // {x,y,0} 表示蛇头在 grid[x][y], 且身体水平, {x,y,1} 则表示身体竖直
        queue<vector<int>> q;
        q.push({0,1,0});
        vector<vector<vector<bool>>> visited =
            vector<vector<vector<bool>>>(n, vector<vector<bool>>(n, vector<bool>(2,
false)));
        visited[0][1][0] = true;
        int ans = -1;
        while(!q.empty()){
            ans++;
            int size = q.size();
            for(int i = 0; i < size; i++){
                int x = q.front()[0];
                int y = q.front()[1];
                int dir = q.front()[2];
                q.pop();
                if(dir == 0){ // 当前身体水平
                    if(x == n-1 && y == n-1) // 到达迷宫右下角
                        return ans;
                    if(y+1 < n && grid[x][y+1] == 0 && visited[x][y+1][0] == false)
                    {
                        // 水平向右移动一格
                        visited[x][y+1][0] = true;
                        q.push({x,y+1,0});
                    }
                }
                if(x+1 < n && grid[x+1][y-1] == 0 && grid[x+1][y] == 0){
                    if(visited[x+1][y][0] == false) { // 竖直向下移动一格
                        visited[x+1][y][0] = true;
                    }
                }
            }
        }
    }
};
```



```

        q.push({x+1,y,0});
    }
    if(visited[x+1][y-1][1] == false){//旋转
        visited[x+1][y-1][1] = true;
        q.push({x+1,y-1,1});
    }
}
}
else{//当前身体竖直
    if(x+1 < n && grid[x+1][y] == 0 && visited[x+1][y][1] == false)

        //竖直向下移动一格
        visited[x+1][y][1] = true;
        q.push({x+1,y,1});
    }
    if(y+1 < n && grid[x-1][y+1] == 0 && grid[x][y+1] == 0){
        if(visited[x][y+1][1] == false){//水平向右移动一格
            visited[x][y+1][1] = true;
            q.push({x,y+1,1});
        }
        if(visited[x-1][y+1][0] == false){//旋转
            visited[x-1][y+1][0] = true;
            q.push({x-1,y+1,0});
        }
    }
}
}
}
return -1;
}
};

```