西南石油大学课程设计报告

课程	姓名	学号
算法分析与设计		
专业班级	任课老师	成绩
软件工程	王欣	

一、主要元素

题目要求找出数组中占比超过一半的元素，且时间复杂度为$O(N)$、空间复杂度为$O(1)$，针对这类问题，可以采用**摩尔投票算法**。

二、摩尔投票算法

摩尔投票算法的主要思想是利用一个计数器来跟踪当前候选的主要元素的“投票”情况，当遍历到与当前候选者相同的元素时，计数器递增；遇到不同的元素时，计数器递减。

本题主要元素的定义是指超过数组长度一般的元素，假设数组长度为n，那么主要元素出现的次数必定大于$n/2$，那么在整个数组的遍历过程中，最终的候选者将会是这个元素，因为它的出现次数可以抵消掉其它元素的“投票”。

但是在找到候选者后，还需要进行第二次遍历来验证这个候选者是否真的满足主要元素的条件，因为在第一次遍历中，候选者可能并不一定是主要元素，尤其是在元素分布比较均匀的情况下，所以验证候选者的出现次数是否超过$n/2$是有必要的。

上面的描述可能有点抽象，这里给出该算法原论文 MJRTY: A Fast Majority Vote Algorithm中给出的一个例子（翻译成中文）：

> 想象一个会议中心充满了代表（即选民），每个人都举着一个牌子，上面写着他支持的候选人的名字。假设一场混乱的场面随之发生，不同派别的代表开始用他们的牌子互相击倒对方。假设每个击倒对方成员的代表同时被对手击倒。显然，如果任何候选人派出的代表人数超过其他所有人的总和，那么这位候选人就会赢得这场混乱的战斗，当混乱平息后，唯一站着的代表将来自多数派。如果没有候选人拥有明显的多数，结果就不那么明确了；在战斗结束时，最多只有一个候选人的支持者，比如说被提名人，会站着——但被提名人可能并不代表所有代表的多数。因此，一般来说，如果有人在这样的战斗结束时仍然站着，会议主席有义务计算被提名人的牌子数量（包括被击倒代表手中的牌子），以确定是否存在多数。

三、算法实现

该算法的 `C++` 实现如下：

```cpp
// 识别候选元素
int identifyCandidate(vector<int>& nums)
{
    // 初始化候选者和计数器
    int candidate = 0, count = 0;

    // 遍历元素
    for (int num : nums)
        if (count == 0)
        {
            candidate = num;
            count = 1;
        }
        else if (num == candidate)
            count++;
        else
            count--;

    return candidate;
}

// 验证候选元素
int verifyCandidate(vector<int>& nums, int candidate)
{
    // 初始化计数器
    int count = 0;

    // 遍历元素
    for (int num : nums)
        if (num == candidate)
            count++;

    // 出现次数大于N/2
    if (count > nums.size() / 2)
        return candidate;

    return -1;
}
```

四、运行结果

以`3 3 4 2 4 4 2 4 4`为例，运行结果如下：

```
D:\COURSE\assignment\CourseDesign\cmake-build-debug\Q1.exe
4

Process finished with exit code 0
```

输出结果正确，为4

五、体会

通过此题目，我理解了摩尔投票算法的原理，而且注意到在主要元素确定存在的情况下，该算法可以实时处理数据而不需要存储投票以便后续进行批量处理，即可以忽略验证阶段。同时，摩尔投票算法还可以扩展到其他需要频率计数的问题，该算法的一般化版本可以找到获得超过n/k票的候选人。而在实际应用中，摩尔投票算法还可以用于处理社交媒体数据、用户行为分析等场景。

一、最小路径和

该题目要求在一个包含非负整数的$m \times n$的网格grid中，找出一条从左上角到右下角的路径，使得路径 上的数字总和为最小。可使用动态规划的思想解决此题。

二、动态规划

1. 定义状态：设$dp[i][j]$表示到达位置(i, j)的最小路径和。
2. 状态转移方程：

- 从左上角`(0, 0)`开始，更新每个位置的最小路径和：$dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1])$
- 其中，$dp[i-1][j]$ 是从上方到达 (i, j) 的路径和，$dp[i][j-1]$ 是从左方到达 (i, j) 的路径和

3. 特殊情况

- $dp[0][0] = grid[0][0]$，即起始位置的路径和就是其自身的值
- 第一行和第一列的值需要特殊处理，因为它们只能通过从左或从上方移动到达

4. 最终结果：最终的结果将存储在 $dp[m-1][n-1]$ 中，即到达右下角的最小路径和

三、算法实现

该算法的 `C++` 实现如下：

```
// 最小路径和
int minPathSum(vector<vector<int>>& grid)
{
    // 网格为空，返回0
    if (grid.empty() || grid[0].empty())
        return 0;

    // 网格的长和宽
    int m = grid.size();
    int n = grid[0].size();

    // 创建一个 dp 数组
    vector<vector<int>> dp(m, vector<int>(n, 0));

    // 初始化 dp[0][0]，即起始位置的路径和就是本身的值
    dp[0][0] = grid[0][0];

    // 填充第一行
    for (int j = 1; j < n; ++j)
    {
        dp[0][j] = dp[0][j - 1] + grid[0][j];
    }

    // 填充第一列
    for (int i = 1; i < m; ++i)
    {
        dp[i][0] = dp[i - 1][0] + grid[i][0];
    }

    // 填充剩余的 dp 表
    for (int i = 1; i < m; ++i)
    {
        for (int j = 1; j < n; ++j)
        {
            // dp[i-1][j]是从上方到达(i,j)的路径和，dp[i][j-1]是从左方到达[i,j]的路径和，取其中更小的值
            dp[i][j] = grid[i][j] + min(dp[i - 1][j], dp[i][j - 1]);
        }
    }

    // 返回右下角的最小路径和
    return dp[m - 1][n - 1];
```

```
}
```

四、运行结果

以 `[[1,3,1],[1,5,1],[4,2,1]]` 为例，运行结果如下：

```
D:\COURSE\assignment\CourseDesign\cmake-build-debug\Q2.exe
7

Process finished with exit code 0
```

1-3-1-1-1为和最小的路径，输出结果正确

五、体会

动态规划是一种强大的算法设计理念，通过将问题分解成更小的子问题并存储其结果，可以有效地避免重复计算，这个过程让我深刻理解了如何利用状态转移方程来构建解决方案。同时，边界条件的处理在动态规划中常常是一个难点，必须仔细考虑如何从初始状态开始推进到最终目标。除此之外，动态规划的思想不仅限于路径问题，还可以扩展到许多其他问题，如背包问题、最长公共子序列等。通过掌握动态规划的基本原理，我对解决类似问题的经验增加了。