第二章算法实现题

学号：2209060322　　姓名：梁桐　　班级：计算机 2203

2-1

题目：

2-1　众数问题。

问题描述：给定含有 n 个元素的多重集合 S，每个元素在 S 中出现的次数称为该元素的重数。多重集 S 中重数最大的元素称为众数。例如，$S=\{1, 2, 2, 2, 3, 5\}$。多重集 S 的众数是 2，其重数为 3。

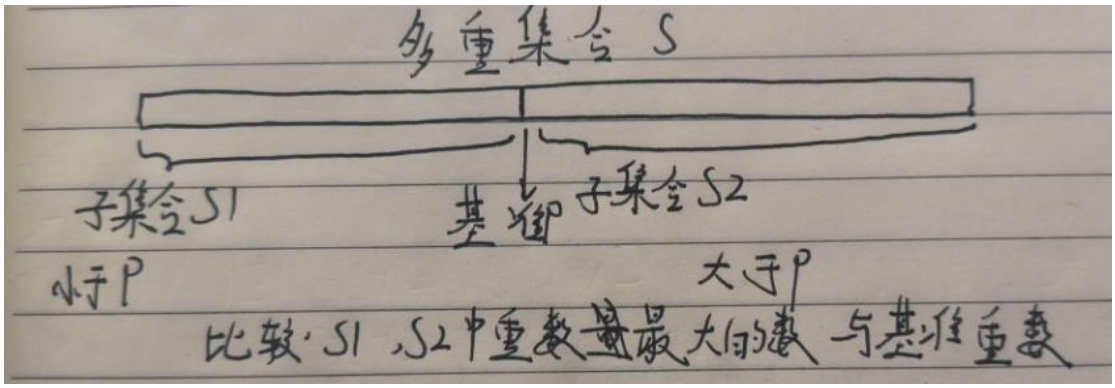
算法设计：对于给定的由 n 个自然数组成的多重集 S，计算 S 的众数及其重数。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行为多重集 S 中元素个数 n；在接下来的 n 行中，每行有一个自然数。

结果输出：将计算结果输出到文件 output.txt。输出文件有 2 行，第 1 行是众数，第 2 行是重数。

输入文件示例	输出文件示例
input.txt	output.txt
6	2
1	3
2	
2	
2	
3	
5	

算法思想：



代码：

```
#include <iostream>
#include <fstream>
#include <vector>
#include <unordered_map>

std::pair<int, int> findMode(const std::vector<int>& data) {
    std::unordered_map<int, int> frequency;
    int maxCount = 0;
    int mode = data[0];
```

```
    for (const int& number : data) {
        frequency[number]++;
        if (frequency[number] > maxCount) {
            maxCount = frequency[number];
            mode = number;
        }
    }

    return std::make_pair(mode, maxCount); // 使用 std::make_pair
}

int main() {
    std::ifstream inputFile("input.txt");
    std::ofstream outputFile("output.txt");

    if (!inputFile) {
        std::cerr << "无法打开输入文件.\n";
        return 1;
    }

    int n;
    inputFile >> n;

    std::vector<int> data(n);
    for (int i = 0; i < n; i++) {
        inputFile >> data[i];
    }

    std::pair<int, int> modeData = findMode(data);
    outputFile << modeData.first << "\n" << modeData.second << "\n"; // 使用 first 和 second 访问

    inputFile.close();
    outputFile.close();
    return 0;
}
```

代码解析:

算法思想

该程序的目标是找到给定多重集中的众数（出现频率最高的元素）及其频率。算法使用哈希表来统计每个元素的出现次数。通过遍历输入数据并更新频率，最后确定出现次数最多的元素。这样，时间复杂度为 O(n)，空间复杂度也是 O(n)。

各个函数的作用

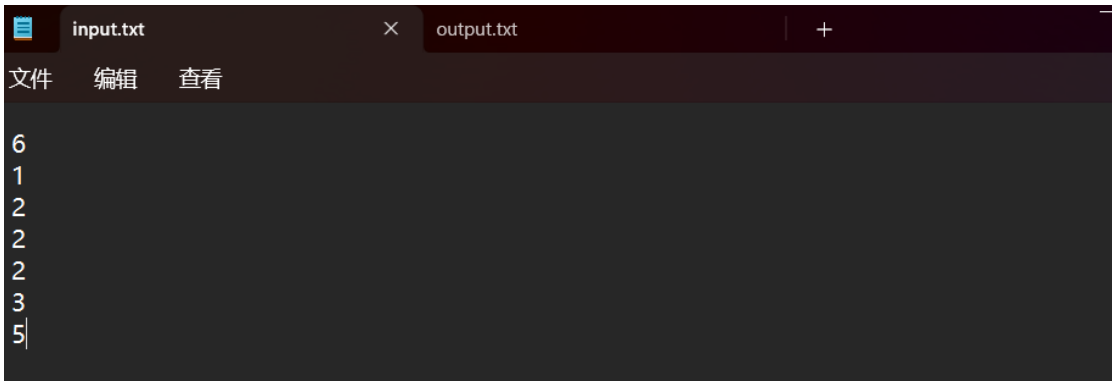
1. findMode(const std::vector<int>& data)
   作用：计算输入数据的众数及其出现频率。
   参数：接收一个整数向量 data，表示多重集的元素。
   返回值：返回一个 std::pair<int, int>，第一个元素是众数，第二个元素是众数的频率。
   实现细节：
   使用 std::unordered_map 来存储每个元素的频率。
   遍历数据并更新频率。
   在遍历过程中，实时跟踪当前的众数及其最大频率。

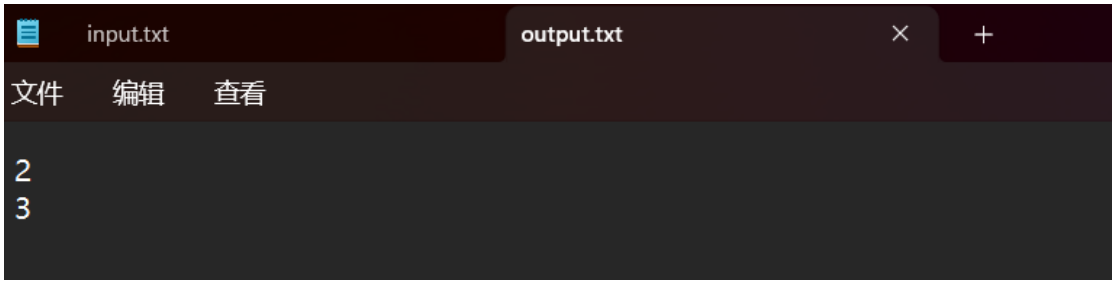
2. main()
   作用：程序的入口点，负责数据的输入和输出。
   实现细节：
   从名为 input.txt 的文件中读取数据，第一行表示元素的个数。
   将后续行中的整数存储在一个向量中。
   调用 findMode 函数获取众数和频率。
   将结果写入名为 output.txt 的文件中，第一行为众数，第二行为其频率。

整体流程
1. 程序启动，读取输入文件，获取元素个数和元素值。
2. 调用 findMode 函数，统计每个元素的频率并找到众数。
3. 将计算结果写入输出文件，结束程序。

这种结构清晰的算法和函数设计有助于提高代码的可读性和可维护性。

运行结果：





2-3

题目：

2-3　半数集问题。

问题描述：给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数如下：

（1）$n \in$ set(n)；

（2）在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半；

（3）按此规则进行处理，直到不能再添加自然数为止。

例如，set(6)={6, 16, 26, 126, 36, 136}。半数集 set(6)中有 6 个元素。注意，该半数集是多重集。

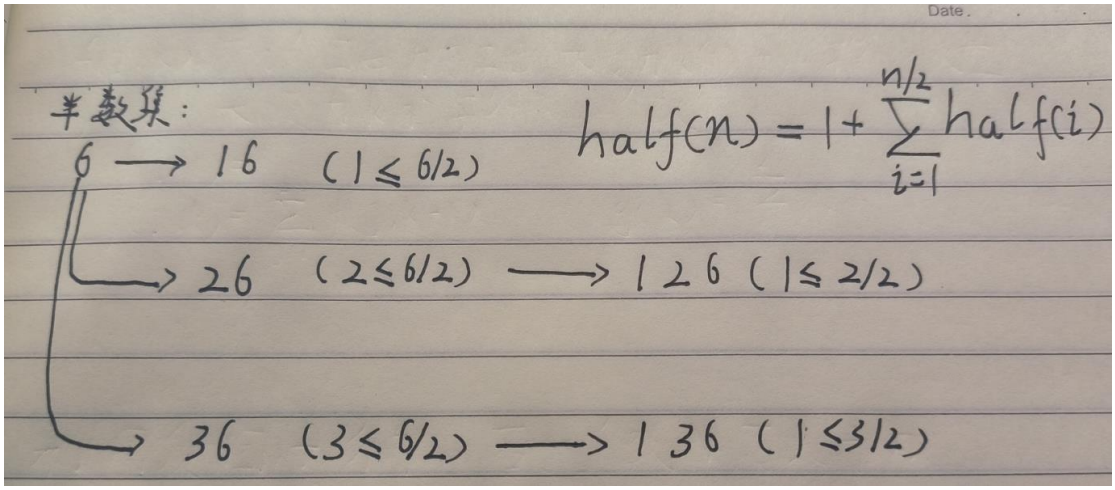
算法设计：对于给定的自然数 n，计算半数集 set(n)中的元素个数。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。每个文件只有一行，给出整数 n（$0<n<1000$）。

结果输出：将计算结果输出到文件 output.txt。输出文件只有一行，给出半数集 set(n)中的元素个数。

输入文件示例	输出文件示例
input.txt	output.txt
6	6

算法思想：

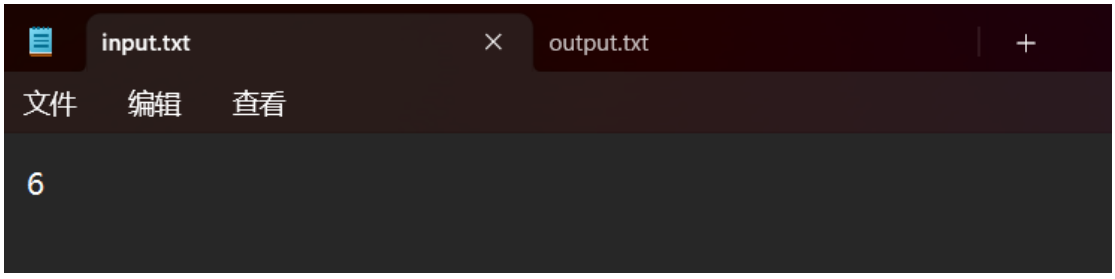


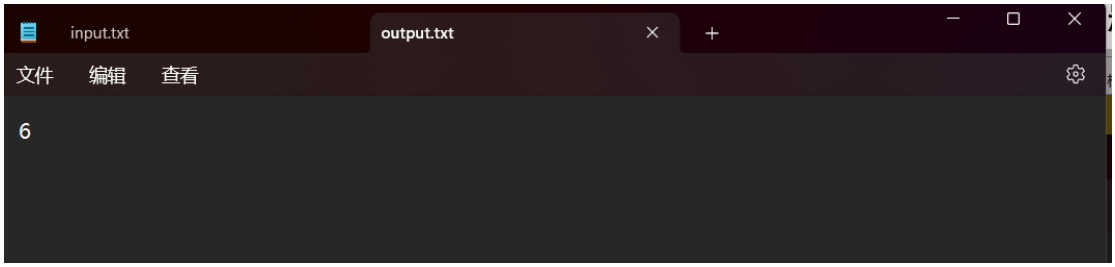
代码：

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_NUM 1000 // 最大数值

int count = 0; // 统计半数集元素个数

// 生成半数集的递归函数
void generate_half_set(int current, int last) {
    count++; // 计数增加
```

```
    for (int i = 1; i <= last / 2; i++) {
        generate_half_set(current * 10 + i, i); // 递归生成下一个数
    }
}

int main() {
    FILE* input_file;  // 输入文件指针
    FILE* output_file; // 输出文件指针

    // 打开输入文件
    if (fopen_s(&input_file, "input.txt", "r") != 0) {
        perror("Failed to open input file");
        return EXIT_FAILURE;
    }

    // 打开输出文件
    if (fopen_s(&output_file, "output.txt", "w") != 0) {
        perror("Failed to open output file");
        fclose(input_file);
        return EXIT_FAILURE;
    }

    int n; // 输入的自然数
    fscanf_s(input_file, "%d", &n); // 读取整数 n
    fclose(input_file); // 关闭输入文件

    generate_half_set(n, n); // 生成半数集

    fprintf(output_file, "%d\n", count); // 写入输出文件
    fclose(output_file); // 关闭输出文件

    return 0; // 程序结束
}
```

运行结果：





2-5

题目:

2-5　有重复元素的排列问题。

问题描述：设 $R=\{r_1, r_2, \cdots, r_n\}$ 是要进行排列的 n 个元素。其中元素 $r_1, r_2, \cdots, r_n$ 可能相同。试设计一个算法，列出 R 的所有不同排列。

算法设计：给定 n 及待排列的 n 个元素。计算出这 n 个元素的所有不同排列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n，$1 \leqslant n \leqslant 500$。接下来的 1 行是待排列的 n 个元素。

结果输出：将计算出的 n 个元素的所有不同排列输出到文件 output.txt。文件最后 1 行中的数是排列总数。

输入文件示例	输出文件示例
input.txt	output.txt
4	aacc
aacc	acac
	acca
	caac
	caca
	ccaa
	6

算法思想:

描述算法的思想:

1. 全排列的基本思路

对于给定的 n 个元素，我们可以将每个元素依次选作第一个元素，然后对剩下的 n 减去 1 个元素进行排列。

具体步骤是

选择第一个元素（当前元素）。

对剩下的 n 减去 1 个元素递归调用排列函数，生成其所有可能的排列。

直到只剩下最后一个元素，此时输出当前排列结果。

2. 递归与回溯

在每一次递归中，我们固定一个元素为当前元素，然后对剩余元素进行排列。

在生成排列时，每次选择新元素前，需要进行交换操作，排列完成后再进行回溯，以恢复状态。

3. 处理重复元素

由于输入的元素可能有重复，需在每次选择当前元素前检查是否有重复元素。具体方法是

在固定当前元素后，遍历已选择的元素，判断是否有与当前元素相同的元素。

如果有相同元素，则跳过此次置换，避免生成重复排列。

这样，最终生成的排列是唯一的，避免了重复的排列输出。

4. 递归终止条件

递归的终止条件是当前选择的元素的索引等于最后一个元素的索引，此时输出当前的排列结果。

代码:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int num = 0;  // 记录排列的总数

// 交换函数
void swap(char* a, char* b) {
    char temp = *a;
    *a = *b;
    *b = temp;
}

// 判断是否需要跳过重复元素
int shouldSkip(char* arr, int start, int current) {
    for (int i = start; i < current; i++) {
        if (arr[i] == arr[current]) {
            return 1;  // 跳过
        }
    }
    return 0;  // 不跳过
}

// 递归生成排列
void permute(char* arr, int start, int end, FILE* output) {
    if (start == end) {
        num++;
        fprintf(output, "%s\n", arr);
    }
```

```
    else {
        for (int i = start; i <= end; i++) {
            if (!shouldSkip(arr, start, i)) {
                swap(&arr[start], &arr[i]);
                permute(arr, start + 1, end, output);
                swap(&arr[start], &arr[i]);  // 回溯
            }
        }
    }
}

int main() {
    FILE* input;
    FILE* output;

    // 使用 fopen_s
    if (fopen_s(&input, "input.txt", "r") != 0) {
        printf("无法打开输入文件。\n");
        return 1;
    }
    if (fopen_s(&output, "output.txt", "w") != 0) {
        printf("无法打开输出文件。\n");
        fclose(input);
        return 1;
    }

    int n;
    fscanf_s(input, "%d\n", &n);  // 读取元素个数

    char* elements = (char*)malloc((n + 1) * sizeof(char)); // +1 用于存放字符串结束符
    fgets(elements, n + 1, input); // 读取元素
    fclose(input);

    // 移除换行符
    size_t len = strlen(elements);
    if (elements[len - 1] == '\n') {
        elements[len - 1] = '\0';
    }

    // 排列并输出结果
    permute(elements, 0, n - 1, output);
    fprintf(output, "%d\n", num); // 输出排列总数
    fclose(output);
```

```
    free(elements);
    return 0;
}
```

运行结果：