第一次作业

2.3.

> **改进的二分搜索算法**
>
> 给定一个已排序的数组 `a[0:n-1]`，设计一个二分搜索算法，使得：
>
> 1. 当搜索的元素 `x` 不在数组中时，返回小于 `x` 的最大元素的位置 `i` 和大于 `x` 的最小元素的位置 `j`。
> 2. 当搜索的元素 `x` 在数组中时，`i` 和 `j` 的值相同，均为 `x` 在数组中的位置。
>
> 要求：编写该算法。

C++版本

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

std::pair<int, int> improvedBinarySearch(const std::vector<int>& a, int
x) {
 // 初始化左、右指针
 int left = 0, right = a.size() - 1;
 int mid;
 // 当左指针小于或等于右指针时，继续搜索
 while (left <= right) {
     // 计算中间位置
     mid = left + (right - left) / 2;
     // 如果中间元素与x相等，返回其位置
     if (a[mid] == x) {
         return {mid, mid};
     }
     // 如果中间元素小于x，更新左指针来搜索右半部分
     else if (a[mid] < x) {
         left = mid + 1;
     }
     // 如果中间元素大于x，更新右指针来搜索左半部分
     else {
         right = mid - 1;
     }
 }
 // 如果没有找到x：
 // 此时，right指针会指向小于x的最大元素
 // 而left指针会指向大于x的最小元素
 int i = right;
 int j = left;
 return {i, j}; // 返回两个指针的位置
}
int main() {
 std::vector<int> a = {1, 3, 5, 7, 9, 11};
 int x = 6;
 // 执行改进的二分搜索
 auto result = improvedBinarySearch(a, x);
```

```
    // 打印结果
    std::cout << "i: " << result.first << ", j: " << result.second <<
std::endl;
    return 0;
}
```

Python版本

```python
from typing import List, Tuple

def improvedBinarySearch(a: List[int], x: int) -> Tuple[int, int]:
    # 初始化左、右指针
    left, right = 0, len(a) - 1
    # 当左指针小于或等于右指针时，继续搜索
    while left <= right:
        # 计算中间位置
        mid = left + (right - left) // 2
        # 如果中间元素与x相等，返回其位置
        if a[mid] == x:
            return mid, mid
        # 如果中间元素小于x，更新左指针来搜索右半部分
        elif a[mid] < x:
            left = mid + 1
        # 如果中间元素大于x，更新右指针来搜索左半部分
        else:
            right = mid - 1
    # 如果没有找到x：
    # 此时，right指针会指向小于x的最大元素
    # 而left指针会指向大于x的最小元素
    i, j = right, left
    return i, j  # 返回两个指针的位置
def main():
    a = [1, 3, 5, 7, 9, 11]
    x = 6
    # 执行改进的二分搜索
    i, j = improvedBinarySearch(a, x)
    # 打印结果
    print(f"i: {i}, j: {j}")
if __name__ == "__main__":
    main()
```

2.9.

寻找数组的主元素

给定一个包含 (n) 个元素的数组 (T[0:n-1])。对于数组中的任意元素 (x)，定义 (S(x) = {i | T[i] = x})，即所有等于 (x) 的元素的索引集合。

当 (|S(x)| > \frac{n}{2}) 时，我们称 (x) 为数组 (T) 的主元素。

任务：设计一个线性时间复杂度的算法，确定数组 (T[0:n-1]) 是否存在一个主元素，并返回该主元素（如果存在）。

要求：编写该算法。

C++版本

```cpp
#include <iostream>
#include <vector>
int findCandidate(const std::vector<int>& T) {
    int candidate = T[0];  // 初始选择第一个元素为候选主元素
    int count = 1;
    for (int i = 1; i < T.size(); i++) {
        // 当计数器为0时，更改候选主元素为当前元素
        if (count == 0) {
            candidate = T[i];
            count = 1;
        } else if (T[i] == candidate) {  // 当前元素等于候选主元素时，增加计数
            count++;
        } else {  // 当前元素不等于候选主元素时，减少计数
            count--;
        }
    }
    return candidate;
}
bool isMajority(const std::vector<int>& T, int candidate) {
    int count = 0;
    for (int i = 0; i < T.size(); i++) {
        if (T[i] == candidate) count++;  // 统计候选主元素在数组中出现的次数
    }
    // 判断该候选元素是否满足主元素的条件
    return count > T.size() / 2;
}
int main() {
    std::vector<int> T = {3, 3, 4, 2, 4, 4, 2, 4, 4};
    int candidate = findCandidate(T);  // 获取数组的候选主元素
    // 验证该候选主元素是否真的满足主元素的定义
    if (isMajority(T, candidate)) {
        std::cout << "主元素是: " << candidate << std::endl;
    } else {
        std::cout << "没有主元素." << std::endl;
    }
    return 0;
}
```

Python版本

```python
from typing import List
def find_candidate(T: List[int]) -> int:
    candidate = T[0]  # 初始选择第一个元素为候选主元素
    count = 1
    for num in T[1:]:
        # 当计数器为0时，更改候选主元素为当前元素
        if count == 0:
            candidate = num
            count = 1
        # 当前元素等于候选主元素时，增加计数
        elif num == candidate:
            count += 1
```

```
        # 当前元素不等于候选主元素时，减少计数
        else:
            count -= 1
    return candidate
def is_majority(T: List[int], candidate: int) -> bool:
    count = T.count(candidate)  # 统计候选主元素在列表中出现的次数
    # 判断该候选元素是否满足主元素的条件
    return count > len(T) // 2
def main():
    T = [3, 3, 4, 2, 4, 4, 2, 4, 4]
    candidate = find_candidate(T)  # 获取列表的候选主元素
    # 验证该候选主元素是否真的满足主元素的定义
    if is_majority(T, candidate):
        print(f"主元素是: {candidate}")
    else:
        print("没有主元素.")
if __name__ == "__main__":
    main()
```

2.10.

寻找无序数组的主元素

背景:

给定一个包含 (n) 个元素的无序数组 (T[0:n-1])。对于数组中的任意元素 (x)，定义 (S(x) = {i | T[i] = x})，即所有等于 (x) 的元素的索引集合。

当 (|S(x)| > \frac{n}{2}) 时，我们称 (x) 为数组 (T) 的主元素。

任务:

1. 设计一个算法，其计算复杂性为 (O(\log n))，来确定数组 (T) 是否存在主元素。
2. 进一步地，能否设计一个线性时间复杂度 (O(n)) 的算法来解决这个问题?

限制:

在数组 (T) 中，元素之间不存在序关系。你只能测试任意两个元素是否相等，不能进行其他比较操作。

C++版本1

```
#include <iostream>
#include <vector>
int findMajorityElement(const std::vector<int>& T) {
    int candidate = -1;  // 当前的主元素候选
    int count = 0;  // 为候选元素“投票”的计数器
    // 第一次遍历，找到可能的主元素候选
    for (int val : T) {
        if (count == 0) {
            candidate = val;
        }
        count += (val == candidate) ? 1 : -1;  // 如果与候选相同，投票，否则撤票
    }
    // 第二次遍历，确认这个候选元素是否真的是主元素
```

```
    count = 0;
    for (int val : T) {
        if (val == candidate) count++;
    }
    if (count > T.size() / 2) return candidate;  // 如果是主元素，返回它
    return -1;  // 否则，返回-1表示没有找到主元素
}
int main() {
    std::vector<int> T = {3, 3, 4, 2, 4, 3, 2, 4, 4};
    int majority = findMajorityElement(T);
    if (majority != -1) {
        std::cout << "主元素是: " << majority << std::endl;
    } else {
        std::cout << "没有主元素." << std::endl;
    }
    return 0;
}
```

C++ 版本2

```
#include <iostream>
#include <vector>
#include <unordered_map>
// 辅助函数，用于统计数组中某元素的出现次数
int countOccurrence(const std::vector<int>& T, int num) {
    int count = 0;
    for (int val : T) {
        if (val == num) count++;
    }
    return count;
}
int findMajorityElementRec(const std::vector<int>& T, int start, int end)
{
    // 基本情况：只有一个元素
    if (start == end) return T[start];
    // 将数组分为两部分并递归查找
    int mid = (start + end) / 2;
    int left = findMajorityElementRec(T, start, mid);
    int right = findMajorityElementRec(T, mid + 1, end);
    // 如果两部分的主元素相同，则返回该元素
    if (left == right) return left;
    // 否则，统计两个主元素的出现次数，并返回次数较多的那个
    int leftCount = countOccurrence(T, left);
    int rightCount = countOccurrence(T, right);
    if (leftCount > (end - start + 1) / 2) return left;
    if (rightCount > (end - start + 1) / 2) return right;
    // 如果没有主元素，返回-1（表示没有找到）
    return -1;
}
int main() {
    std::vector<int> T = {3, 3, 4, 2, 4, 3, 2, 4, 4};
    int majority = findMajorityElementRec(T, 0, T.size() - 1);
    if (majority != -1 && countOccurrence(T, majority) > T.size() / 2) {
        std::cout << "主元素是: " << majority << std::endl;
    } else {
```

```
        std::cout << "没有主元素." << std::endl;
    }
    return 0;
}
```

PS: 摩尔投票算法是为这个问题提供了$O(n)$时间复杂度的最佳已知解决方案