

基于分治策略的快速排序及其衍生算法

实验内容

本实验要求基于算法设计与分析的一般过程（亦即待求解问题的描述、算法设计、算法描述、算法正确性证明、算法分析、算法实现与测试），完成以下算法的设计与分析：a) **基于分治策略的快速排序算法**，鼓励采用非递归式的算法；b) **在快排算法的基础上，基于分治法的众数（即[如整数等]数据集中出现次数最多的数）求解算法**。另外，若仍有余力，可以挑战基于其它经典分治算法，诸如循环日程表、棋盘覆盖问题等等。

实验目的

理解分治法的核心思想以及分治法求解过程

实现语言

C++

实验步骤/实验结果

步骤一：理解问题，给出问题的描述

a)

快速排序（QuickSort） 是一种高效的排序算法，基于 **分治（Divide and Conquer）** 思想。传统快速排序的核心思想是：

- 选择一个 **基准元素（Pivot）**。
- 通过 **分区（Partition）** 操作，将数据集划分为 **左子数组（小于基准值）** 和 **右子数组（大于基准值）**。
- 递归地对两个子数组进行排序。

b)

众数（Mode） 是指在数据集中**出现次数最多**的元素。例如，在数组 `[1, 3, 3, 2, 3, 2, 2, 2, 2]` 中，**众数是 2**，因为它出现了 **5 次**，比其他数字都多。

要求： 我们需要利用 **分治策略（Divide and Conquer）** 设计一种高效的众数求解算法。思路类似于快速排序：

- 划分数组：** 选择一个**基准值**，将数组划分为左右两部分。
- 递归统计：** 分别计算左右子数组的众数。
- 合并结果：** 比较左右子数组的众数，并计算全局出现次数，从而确定最终的众数。

目标 是通过**分治思想**提高众数求解的效率，使其优于**暴力计数法（ $O(n^2)$ ）**，尽可能接近**线性时间复杂度 $O(n)$** 。

步骤2：算法设计，包括算法策略与数据结构的选择

算法策略

1. a)

- 用首元素 X 作划分标准，将输入数组 A 划分为不超过 X 的元素构成的数据 AL，大于 X 的元素构成的数组 AR，其中，AL，AR 从左到右存放在数据 A 的位置
- 递归的对子问题 AL 和 AR 进行相同的操作，直到子问题规模为 1 时停止

2. b)

- 首先采用快速排序进行排序，规定为升序
- 分解：将数组分成两个子数组，分别在左右子数组中寻找众数。
- 解决：递归地计算左右子数组中的众数。
- 合并：比较左右子数组的众数，计算它们在整个数组中的出现次数，从而确定全局的众数。

数据结构

数组, 栈

步骤3：描述算法。希望采用源代码以外的形式，如伪代码等

a) 描述

```
def quicksort_non_recursive(A):
    stack = [(0, len(A) - 1)]

    while stack:
        low, high = stack.pop()
        if low < high:
            pivot_index = partition(A, low, high)
            if pivot_index + 1 < high:
                stack.append((pivot_index + 1, high))
            if pivot_index - 1 > low:
                stack.append((low, pivot_index - 1))

def partition(A, low, high):
    pivot = A[high]
    i = low - 1
    for j in range(low, high):
        if A[j] <= pivot:
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[high] = A[high], A[i + 1]
    return i + 1
```

b) 描述

```
function majorityElement(arr, left, right):
    if left == right:
        return arr[left] // 只有一个元素，返回它作为众数
```

```

mid = (left + right) / 2

// 递归求解左半部分的众数
left_major = majorityElement(arr, left, mid)

// 递归求解右半部分的众数
right_major = majorityElement(arr, mid + 1, right)

// 若左右众数相同，则返回它
if left_major == right_major:
    return left_major

// 否则，统计左右众数在整个区间的出现次数
left_count = count_occurrences(arr, left, right, left_major)
right_count = count_occurrences(arr, left, right, right_major)

// 返回出现次数较多的作为最终众数
return left_major if left_count > right_count else right_major

```

步骤4：算法的正确性证明。需要这个环节，在理解的基础上对算法的正确性给予证明

循环不变式主要用来帮助我们理解算法的正确性。关于循环不变式，我们必须证明三条性质：

初始化：循环的第一次迭代之前，它为真。

保持：如果循环的某次迭代之前它为真，那么下次迭代之前它仍为真。

终止：在循环终止时，不变式为我们提供一个有用的性质，该性质有助于证明算法是正确的。

——《算法导论（第3版）》第2.1节 插入排序

a) 循环不变式证明

- 初始化

只把整个数组 `[0, n-1]` 入栈，这保证了整个数组会被处理，满足“待处理子区间被正确管理到栈中”。

- 保持性

选定 `pivot` 并确保 `pivot` 归位，`pivot` 左侧所有元素 $\leq pivot$ ，右侧所有元素 $> pivot$ ：任何已处理的 `pivot` 位置都是正确的

将子区间 `[low, pivot-1]` 和 `[pivot+1, high]` 入栈（如果区间大小 > 1 ）：未排序的部分仍然存入栈，保证它们将被进一步处理。

- 终止性

当栈为空时：所有子区间都已处理完毕，每个 `pivot` 的位置正确，整个数组已排序

b)循环不变式证明

- 初始化

当 `left == right` (即子数组只有一个元素时), 显然该元素就是该子数组的众数, 算法直接返回 `arr[left]`, 正确性成立

- 保持性

递归过程中, 数组被分为左右两部分, 划分结束后, 反向逐层比较, 如果 `left_major == right_major`, 则直接返回该众数, 否则, 分别统计 `left_major` 和 `right_major` 在整个 `arr[left] ~ arr[right]` 之间的出现次数, 返回出现次数较多的那个。如果左右子数组的众数不同, 整体众数要么属于 `left_major`, 要么属于 `right_major`, 不会出现新的众数。因此统计出现次数后选取出现次数较多的候选者是正确的。

- 终止性

由于循环不变式保证了每个子区间计算出的众数都是正确的, 最终合并的结果也是正确的, 当循环终止时, 算法返回的值是正确的众数, 证明算法正确

步骤5: 算法复杂性分析, 包括时间复杂性和空间复杂性

a)的相关分析

时间复杂性

- 划分 (partition) 的时间复杂度:

- 每次调用 `partition` 时, 我们需要遍历 `A[low:high]` 共 $O(n)$ 次。

- 整个快速排序的时间复杂度:

- 最优情况 ($O(n \log n)$):

- 如果每次 `partition` 后的 `pivot` 位置较均匀 (如总是选到中位数), 则递归树的高度是 $O(\log n)$, 每层的 `partition` 操作是 $O(n)$, 所以总时间复杂度是:

$$O(n \log n)$$

- 最坏情况 ($O(n^2)$):

- 如果 `partition` 选择的 `pivot` 总是最小或最大元素, 递归树变成 $O(n)$ 深度, 每层的 `partition` 仍然是 $O(n)$, 所以总时间复杂度退化为:

$$O(n^2)$$

- 平均情况 ($O(n \log n)$):

- 随机选择 `pivot` 可以避免最坏情况, 通常会达到 $O(n \log n)$ 的复杂度。

空间复杂性

- 主要消耗空间的是 **栈 stack**，用于存储待排序的区间。
- **最优情况下 ($O(\log n)$)**：
 - 每次 **partition** 使得子区间大小对半减少，栈的最大深度是 $O(\log n)$ ，所以空间复杂度是 $O(\log n)$ 。
- **最坏情况下 ($O(n)$)**：
 - 如果 **partition** 选取的 **pivot** 使得一边子区间始终为空，栈的深度就会达到 $O(n)$ ，导致 $O(n)$ 的空间复杂度。
- **平均情况下 ($O(\log n)$)**：
 - 平均而言，快速排序的划分是对数级别的，因此空间复杂度为 $O(\log n)$ 。

b)的相关分析

时间复杂度

- **递归调用**：每次递归划分数组为两半，总共执行 $O(\log n)$ 次。
- **计数函数**：每次合并阶段统计两个候选众数的出现次数，需要 $O(n)$ 时间。
- **总复杂度**：

$$T(n) = 2T(n/2) + O(n)$$

这是一个 **分治递归方程**，解的结果是 $O(n \log n)$ 。

最坏情况（如果每次都需要完整扫描子数组），时间复杂度仍然是 $O(n \log n)$ 。

空间复杂度

- 由于是**递归实现**，递归深度最多为 $O(\log n)$ 。
- **额外的存储开销很少**（除了递归栈空间，没有额外的数组存储）。

总空间复杂度： $O(\log n)$ 。

步骤6：算法实现与测试。附上代码或以附件的形式提交，同时贴上算法运行结果截图；

a) 代码和运行结果截图

- 代码

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

using namespace std;
```

```

int partition(vector<int>& A, int low, int high) {
    int pivot = A[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (A[j] <= pivot) {
            i++;
            swap(A[i], A[j]);
        }
    }
    swap(A[i + 1], A[high]);
    return i + 1;
}

void quicksort_non_recursive(vector<int>& A) {
    stack<pair<int, int>> s;
    s.push({ 0, A.size() - 1 });

    while (!s.empty()) {
        int low = s.top().first, high = s.top().second;
        s.pop();
        if (low < high) {
            int pivot_index = partition(A, low, high);
            if (pivot_index - 1 > low) s.push({ low, pivot_index - 1 });
            if (pivot_index + 1 < high) s.push({ pivot_index + 1, high });
        }
    }
}

int main() {
    vector<int> arr = { 10, 7, 8, 9, 1, 5 };
    quicksort_non_recursive(arr);
    cout << "Sorted array: ";
    for (int num : arr) cout << num << " ";
    cout << endl;
    return 0;
}

```

- 结果



B) 代码和运行结果截图

- 代码

```

#include <iostream>
#include <vector>

using namespace std;

// 统计 `x` 在 `arr[left] ~ arr[right]` 之间的出现次数

```

```

int countOccurrences(const vector<int>& arr, int left, int right, int x) {
    int count = 0;
    for (int i = left; i <= right; i++) {
        if (arr[i] == x) count++;
    }
    return count;
}

// 分治求解众数
int majorityElement(vector<int>& arr, int left, int right) {
    // 递归终止条件：当区间内只有一个元素时，返回该元素
    if (left == right) {
        return arr[left];
    }

    // 分割数组
    int mid = left + (right - left) / 2;

    // 递归求解左右子数组的众数
    int left_major = majorityElement(arr, left, mid);
    int right_major = majorityElement(arr, mid + 1, right);

    // 如果左右部分的众数相同，则返回该众数
    if (left_major == right_major) {
        return left_major;
    }

    // 统计左众数和右众数在整个区间中的出现次数
    int left_count = countOccurrences(arr, left, right, left_major);
    int right_count = countOccurrences(arr, left, right, right_major);

    // 返回出现次数较多的元素作为全局众数
    return (left_count > right_count) ? left_major : right_major;
}

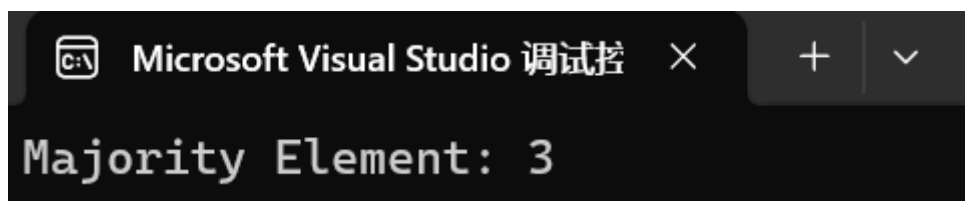
// 入口函数
int main() {
    vector<int> arr = {2,3,1,5,3,2,3};

    int result = majorityElement(arr, 0, arr.size() - 1);
    cout << "Majority Element: " << result << endl;

    return 0;
}

```

- 结果



实验总结

(1) (2) /技术上、分析过程中等各种心得体会与备忘，需要言之有物。

1. 递归 vs. 非递归

- 传统的快速排序基于递归，利用函数调用栈来维护待排序的子区间，而非递归版本则需要显式使用栈来模拟递归行为。
- 关键在于如何管理子区间的入栈顺序，以减少栈的深度，避免最坏情况下的 $O(n)$ 空间开销。

2. 去递归化的思考

- 我们用一个栈记录每个子区间的众数信息
- 采用自底向上的合并方式，每次从栈中取出两个子区间，合并成更大的区间
- 通过 `count_occurrences` 统计 `mode1` 和 `mode2` 的出现次数，选择出现较多的作为合并后的众数

3. 快排的有个缺陷是对于基准值的要求比较高，如果基准值选择不当，就会导致排序效率严重降低。

4. C++ 和 Python 处理方式的不同

- Python
 - `stack.pop()` 直接返回元组，可以 `left, right, mode = stack.pop()`
 - `sum(1 for i in range(left, right+1) if arr[i] == x)` 可以用列表解析快速统计
 - C++
 - `stack.top()` 取出但不删除，需要 `stack.pop()` 结合 `struct`
 - 需要手写循环统计 `count_occurrences`
5. 求众数的算法还可以优化，比如单调栈或者投票

(3) /如何运用分治法举一反三

了解原理后，可应用到二分查找、归并排序、大数据计算.....