数据结构与算法

Quaternijkon

数据结构与算法刷题本

July 02, 2024

主要来源于力扣

目录

1. 数据结构

数据结构是为实现对计算机数据有效使用的各种数据组织形式，服务于各类计算机操作。不同的数据结构具有各自对应的适用场景，旨在降低各种算法计算的时间与空间复杂度，达到最佳的任务执行效率。

1.1. 数组

1.1.1. ☐数组中的重复元素

? 寻找文件副本

设备中存有 n 个文件，文件 id 记于数组 documents。若文件 id 相同，则定义为该文件存在副本。请返回任一存在副本的文件 id。

 示例 **1**

输入：documents = [2, 5, 3, 0, 5, 0]

输出：0 或 5

 提示

0 ≤ documents[i] ≤ n-1

2 <= n <= 100000

 方法 **1:** 遍历

遍历中，第一次遇到数字 x 时，将其交换至索引 x 处；而当第二次遇到数字 x 时，一定有 documents[x]=x ，此时即可得到一组重复数字。

cpp

```cpp
class Solution {
public:
    int findRepeatDocument(vector<int>& documents) {
        int i = 0;
        while(i < documents.size()) {
            if(documents[i] == i) {
                i++;
                continue;
            }
            if(documents[documents[i]] == documents[i])
                return documents[i];
            swap(documents[i],documents[documents[i]]);
        }
        return -1;
    }
};
```

1.2. 链表

1.2.1. ☐反转链表

? 反转链表

给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。

示例 1

输入：head = [1,2,3,4,5]

输出：[5,4,3,2,1]

示例 2

输入：head = [1,2]

输出：[2,1]

示例 3

输入：head = []

输出：[]

提示

链表中节点的数目范围是 [0, 5000]

-5000 <= Node.val <= 5000

方法 1: 迭代

考虑遍历链表，并在访问各节点时修改 `next` 引用指向，算法流程见注释。

```cpp
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *cur = head, *pre = nullptr;
        while(cur != nullptr) {
            ListNode* tmp = cur->next; // 暂存后继节点 cur.next
            cur->next = pre;           // 修改 next 引用指向
            pre = cur;                 // pre 暂存 cur
            cur = tmp;                 // cur 访问下一节点
```

```
        }
        return pre;
    }
};
```

方法 2: 递归

考虑使用递归法遍历链表，当越过尾节点后终止递归，在回溯时修改各节点的 `next` 引用指向。

`recur(cur, pre)` 递归函数：
1. 终止条件：当 `cur` 为空，则返回尾节点 `pre` （即反转链表的头节点）；
2. 递归后继节点，记录返回值（即反转链表的头节点）为 `res` ；
3. 修改当前节点 `cur` 引用指向前驱节点 `pre` ；
4. 返回反转链表的头节点 `res` ；

`reverseList(head)` 函数：

调用并返回 `recur(head, null)` 。传入 `null` 是因为反转链表后， `head` 节点指向 `null` ；

```cpp
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        return recur(head, nullptr);           // 调用递归并返回
    }
private:
    ListNode* recur(ListNode* cur, ListNode* pre) {
        if (cur == nullptr) return pre;        // 终止条件
        ListNode* res = recur(cur->next, cur); // 递归后继节点
        cur->next = pre;                       // 修改节点引用指向
        return res;                            // 返回反转链表的头节点
    }
};
```

1.3. 栈与队列

1.3.1. ☐ 中位数 ↗

? 数据流中的中位数

中位数是有序整数列表中的中间值。如果列表的大小是偶数，则没有中间值，中位数是两个中间值的平均值。

例如，

- [2,3,4] 的中位数是 3
- [2,3] 的中位数是 $\frac{2+3}{2} = 2.5$

设计一个支持以下两种操作的数据结构：

- `void addNum(int num)` - 从数据流中添加一个整数到数据结构中。
- `double findMedian()` - 返回目前所有元素的中位数。

 示例 **1**

输入：["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]

[[],[1],[2],[],[3],[]]

输出：[null,null,null,1.50000,null,2.00000]

 示例 **2**

输入：["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]

[[],[1],[2],[],[3],[]]

输出：[null,null,null,1.50000,null,2.00000]

 提示

最多会对 `addNum`、`findMedian` 进行 50000 次调用。

 方法 **1:** 堆

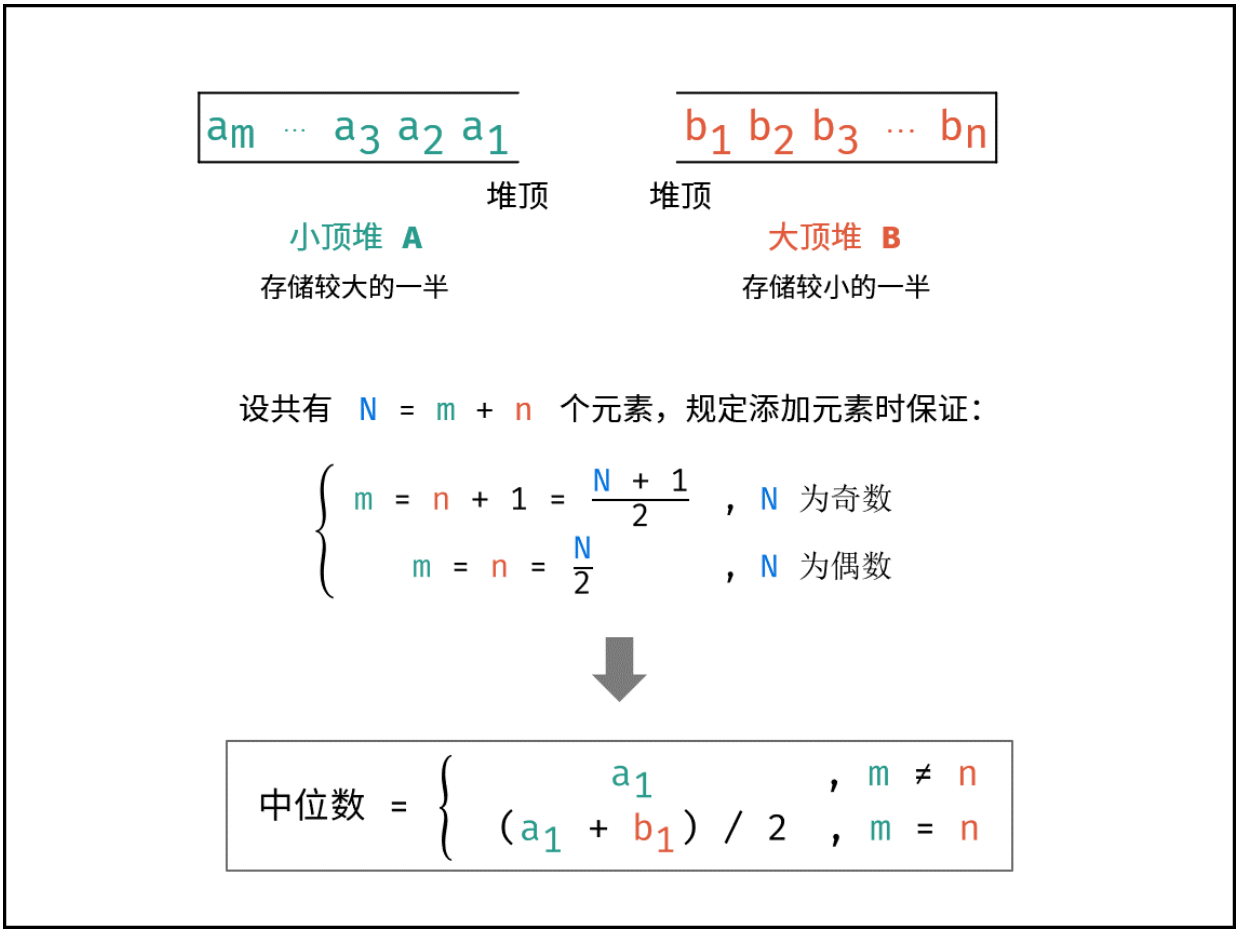

图 1　堆

`addNum(num)` 函数：

- 当 $m = n$（即 N 为 偶数）：需向 A 添加一个元素。实现方法：将新元素 `num` 插入至 B，再将 B 堆顶元素插入至 A；
- 当 $m \neq n$（即 N 为 奇数）：需向 B 添加一个元素。实现方法：将新元素 `num` 插入至 A，再将 A 堆顶元素插入至 B；

`findMedian()` 函数：

- 当 $m = n$（N 为 偶数）：则中位数为 (A 的堆顶元素 + B 的堆顶元素)/2。
- 当 $m \neq n$（N 为 奇数）：则中位数为 A 的堆顶元素。

```cpp
class MedianFinder {
public:
    priority_queue<int, vector<int>, greater<int>> A; // 小顶堆，保存较大的一半
    priority_queue<int, vector<int>, less<int>> B; // 大顶堆，保存较小的一半
    MedianFinder() { }
    void addNum(int num) {
        if(A.size() != B.size()) {
            A.push(num);
```

```
            B.push(A.top());
            A.pop();
        } else {
            B.push(num);
            A.push(B.top());
            B.pop();
        }
    }
    double findMedian() {
        return A.size() != B.size() ? A.top() : (A.top() + B.top()) / 2.0;
    }
};
```

1.4. 字符串

1.4.1. ☐最长回文子串

? 最长回文子串

给你一个字符串 s，找到 s 中最长的回文子串。

示例 1

输入：s = “babad”

输出："bab"

解释："aba" 同样是符合题意的答案。

示例 2

输入：s = “cbbd”

输出："bb"

提示

1 <= s.length <= 1000

s 仅由数字和英文字母组成

方法 1： 动态规划

对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。例如对于字符串 “ababa”，如果我们已经知道 “bab” 是回文串，那么 “ababa” 一定是回文串，这是因为它的首尾两个字母都是 “a”。

根据这样的思路，我们就可以用动态规划的方法解决本题。我们用 `P(i,j)` 表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 `s[i:j]`）是否为回文串：

$$\begin{cases} P(i,j) = \text{true} & \text{如果子串} S_i \ldots S_j \text{是回文串} \\ P(i,j) = \text{false} & \text{其它情况} \end{cases} \tag{1}$$

这里的「其它情况」包含两种可能性：

- `s[i,j]` 本身不是一个回文串；
- $i > j$，此时 `s[i,j]` 本身不合法。

那么我们就可以写出动态规划的状态转移方程：

$$P(i,j) = P(i+1,j-1) \wedge (S_i == S_j) \tag{2}$$

也就是说，只有 `s[i+1:j-1]` 是回文串，并且 s 的第 i 和 j 个字母相同时，`s[i:j]` 才会是回文串。

上文的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们就可以写出动态规划的边界条件：

$$\begin{cases} P(i,i) = \text{true} \\ P(i,i+1) = \left(S_i == S_{\{i+1\}}\right) \end{cases} \tag{3}$$

根据这个思路，我们就可以完成动态规划了，最终的答案即为所有 `P(i,j)=true` 中 `j-i+1`（即子串长度）的最大值。注意：在状态转移方程中，我们是从长度较短的字符串向长度较长的字符串进行转移的，因此一定要注意动态规划的循环顺序。

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

using namespace std;

class Solution {
public:
    string longestPalindrome(string s) {
        int n = s.size();
        if (n < 2) {
            return s;
        }

        int maxLen = 1;
        int begin = 0;
        // dp[i][j] 表示 s[i..j] 是否是回文串
        vector<vector<int>> dp(n, vector<int>(n));
        // 初始化：所有长度为 1 的子串都是回文串
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }
        // 递推开始
        // 先枚举子串长度
        for (int L = 2; L <= n; L++) {
            // 枚举左边界，左边界的上限设置可以宽松一些
```

```
27              for (int i = 0; i < n; i++) {
28                  // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
29                  int j = L + i - 1;
30                  // 如果右边界越界，就可以退出当前循环
31                  if (j >= n) {
32                      break;
33                  }
34
35                  if (s[i] != s[j]) {
36                      dp[i][j] = false;
37                  } else {
38                      if (j - i < 3) {
39                          dp[i][j] = true;
40                      } else {
41                          dp[i][j] = dp[i + 1][j - 1];
42                      }
43                  }
44
45                  // 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是回文，此时记录回文长
    度和起始位置
46                  if (dp[i][j] && j - i + 1 > maxLen) {
47                      maxLen = j - i + 1;
48                      begin = i;
49                  }
50              }
51          }
52          return s.substr(begin, maxLen);
53      }
54  };
```

方法 2: 中心扩展算法

我们仔细观察一下方法一中的状态转移方程：

$$\begin{cases} P(i,j) = \text{true} \\ P(i,i+1) = \left(S_i == S_{\{i+1\}}\right) \\ P(i,j) = P(i+1,j-1) \wedge \left(S_i == S_j\right) \end{cases} \tag{4}$$

找出其中的状态转移链：

$$P(i,j) \leftarrow P(i+1,j-1) \leftarrow P(i+2,j-2) \leftarrow \cdots \leftarrow \text{某一边界情况} \tag{5}$$

可以发现，所有的状态在转移的时候的可能性都是唯一的。也就是说，我们可以从每一种边界情况开始「扩展」，也可以得出所有的状态对应的答案。

边界情况即为子串长度为 1 或 2 的情况。我们枚举每一种边界情况，并从对应的子串开始不断地向两边扩展。如果两边的字母相同，我们就可以继续扩展，例如从 `P(i+1,j-1)` 扩展到 `P(i,j)`；如果两边的字母不同，我们就可以停止扩展，因为在这之后的子串都不能是回文串了。

聪明的读者此时应该可以发现，「边界情况」对应的子串实际上就是我们「扩展」出的回文串的「回文中心」。方法二的本质即为：我们枚举所有的「回文中心」并尝试「扩展」，直到无法扩展为止，此时的回文串长度即为此「回文中心」下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

```cpp
class Solution {
public:
    pair<int, int> expandAroundCenter(const string& s, int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left;
            ++right;
        }
        return {left + 1, right - 1};
    }

    string longestPalindrome(string s) {
        int start = 0, end = 0;
        for (int i = 0; i < s.size(); ++i) {
            auto [left1, right1] = expandAroundCenter(s, i, i);
            auto [left2, right2] = expandAroundCenter(s, i, i + 1);
            if (right1 - left1 > end - start) {
                start = left1;
                end = right1;
            }
            if (right2 - left2 > end - start) {
                start = left2;
                end = right2;
            }
        }
        return s.substr(start, end - start + 1);
    }
};
```

方法 3：Manacher 算法

略

```cpp
class Solution {
public:
    int expand(const string& s, int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left;
            ++right;
        }
        return (right - left - 2) / 2;
    }

    string longestPalindrome(string s) {
        int start = 0, end = -1;
        string t = "#";
        for (char c: s) {
            t += c;
            t += '#';
        }
        t += '#';
        s = t;

        vector<int> arm_len;
        int right = -1, j = -1;
        for (int i = 0; i < s.size(); ++i) {
            int cur_arm_len;
            if (right >= i) {
                int i_sym = j * 2 - i;
                int min_arm_len = min(arm_len[i_sym], right - i);
                cur_arm_len = expand(s, i - min_arm_len, i + min_arm_len);
            } else {
                cur_arm_len = expand(s, i, i);
            }
            arm_len.push_back(cur_arm_len);
            if (i + cur_arm_len > right) {
                j = i;
                right = i + cur_arm_len;
            }
            if (cur_arm_len * 2 + 1 > end - start) {
                start = i - cur_arm_len;
                end = i + cur_arm_len;
            }
        }

```

```
43          string ans;
44          for (int i = start; i ⩽ end; ++i) {
45              if (s[i] ≠ '#') {
46                  ans += s[i];
47              }
48          }
49          return ans;
50      }
51  };
```

1.5. 哈希表

1.5.1. ☐同构字符串

? 同构字符串

给定两个字符串 s 和 t，判断它们是否是同构的。

如果 s 中的字符可以按某种映射关系替换得到 t，那么这两个字符串是同构的。

每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。不同字符不能映射到同一个字符上，相同字符只能映射到同一个字符上，字符可以映射到自己本身。

 示例 1

输入：s = “egg”, t = “add”

输出：true

 示例 2

输入：s = “foo”, t = “bar”

输出：false

 示例 3

输入：s = “paper”, t = “title”

输出：true

 提示

1 <= s.length <= $5 * 10^4$

t.length == s.length

s 和 t 由任意有效的 ASCII 字符组成

 方法 1: 散列表

使用两个映射表分别记录字符串 `s` 和 `t` 中每个字符的第一次出现位置，然后检查两个字符串对应位置的字符是否具有相同的第一次出现位置，以判断是否同构。

```cpp
class Solution {
public:
```

```
3       bool isIsomorphic(string s, string t) {
4           int n=s.size();
5           unordered_map<char,int> smap;
6           unordered_map<char,int> tmap;
7
8           for(int i=0;i<n;++i)
9               if(smap[s[i]]==0) smap[s[i]]=i+1;
10          for(int i=0;i<n;++i)
11              if(tmap[t[i]]==0) tmap[t[i]]=i+1;
12          for(int i=0;i<n;++i)
13              if(smap[s[i]]≠tmap[t[i]]) return false;
14          return true;
15      }
16  };
```

1.6. 堆

1.6.1. ☐数组中的第 K 个最大元素

? 数组中的第 K 个最大元素

给定整数数组 nums 和整数 k，请返回数组中第 k 个最大的元素。

请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

 示例 1

输入: [3,2,1,5,6,4], k = 2

输出: 5

 示例 2

输入: [3,2,3,1,2,4,5,5,6], k = 4

输出: 4

 提示

$1 <= k <= nums.length <= 10^5$

$-10^4 <= nums[i] <= 10^4$

 方法 1: 基于快速排序的选择方法

我们可以用快速排序来解决这个问题，先对原数组排序，再返回倒数第 k 个位置，这样平均时间复杂度是 O(nlogn)，但其实我们可以做的更快。

首先我们来回顾一下快速排序，这是一个典型的分治算法。我们对数组 `a[l…r]` 做快速排序的过程是（参考《算法导论》）：

- 分解：将数组 `a[l…r]` 「划分」成两个子数组 `a[l…q-1]`、`a[q+1…r]`，使得 `a[l…q-1]` 中的每个元素小于等于 `a[q]`，且 `a[q]` 小于等于 `a[q+1…r]` 中的每个元素。其中，计算下标 q 也是「划分」过程的一部分。
- 解决：通过递归调用快速排序，对子数组 `a[l…q-1]` 和 `a[q+1…r]` 进行排序。
- 合并：因为子数组都是原址排序的，所以不需要进行合并操作，`a[l…r]` 已经有序。
- 上文中提到的「划分」过程是：从子数组 `a[l…r]` 中选择任意一个元素 x 作为主元，调整子数组的元素使得左边的元素都小于等于它，右边的元素都大于等于它，x 的最终位置就是 q。

由此可以发现每次经过「划分」操作后，我们一定可以确定一个元素的最终位置，即 x 的最终位置为 q，并且保证 `a[l…q-1]` 中的每个元素小于等于 `a[q]`，且 `a[q]` 小于等于 `a[q+1…r]` 中的每个元素。所以只要某次划分的 q 为倒数第 k 个下标的时候，我们就已经找到了答案。我们只关心这一点，至于 `a[l…q-1]` 和 `a[q+1…r]` 是否是有序的，我们不关心。

因此我们可以改进快速排序算法来解决这个问题：在分解的过程当中，我们会对子数组进行划分，如果划分得到的 q 正好就是我们需要的下标，就直接返回 `a[q]`；否则，如果 q 比目标下标小，就递归右子区间，否则递归左子区间。这样就可以把原来递归两个区间变成只递归一个区间，提高了时间效率。

```cpp
class Solution {
public:
    int quickselect(vector<int> &nums, int l, int r, int k) {
        if (l == r)
            return nums[k];
        int partition = nums[l], i = l - 1, j = r + 1;
        while (i < j) {
            do i++; while (nums[i] < partition);
            do j--; while (nums[j] > partition);
            if (i < j)
                swap(nums[i], nums[j]);
        }
        if (k <= j)return quickselect(nums, l, j, k);
        else return quickselect(nums, j + 1, r, k);
    }

    int findKthLargest(vector<int> &nums, int k) {
        int n = nums.size();
        return quickselect(nums, 0, n - 1, n - k);
    }
};
```

方法 2：基于堆排序的选择方法

我们也可以使用堆排序来解决这个问题——建立一个大根堆，做 k-1 次删除操作后堆顶元素就是我们要找的答案。

```cpp
class Solution {
public:
    void maxHeapify(vector<int>& a, int i, int heapSize) {
        int l = i * 2 + 1, r = i * 2 + 2, largest = i;
        if (l < heapSize && a[l] > a[largest]) {
            largest = l;
```

```
7           }
8           if (r < heapSize && a[r] > a[largest]) {
9               largest = r;
10          }
11          if (largest ≠ i) {
12              swap(a[i], a[largest]);
13              maxHeapify(a, largest, heapSize);
14          }
15      }
16
17      void buildMaxHeap(vector<int>& a, int heapSize) {
18          for (int i = heapSize / 2; i ⩾ 0; --i) {
19              maxHeapify(a, i, heapSize);
20          }
21      }
22
23      int findKthLargest(vector<int>& nums, int k) {
24          int heapSize = nums.size();
25          buildMaxHeap(nums, heapSize);
26          for (int i = nums.size() - 1; i ⩾ nums.size() - k + 1; --i) {
27              swap(nums[0], nums[i]);
28              --heapSize;
29              maxHeapify(nums, 0, heapSize);
30          }
31          return nums[0];
32      }
33  };
```

1.7. 树

1.8. 图

2. 算法

2.1. 二分查找

2.2. 排序算法

2.3. 滑动窗口与双指针

2.4. 递归与分治

2.5. 广度优先搜索

2.6. 深度优先搜索

2.7. 贪心算法

2.8. 动态规划

2.9. 前缀和

2.10. 数学

参考文献

Index of Figures