计算机算法设计与分析

081203M04001H

Chap 2 课程作业

2022 年 10 月 7 号

Professor: 卜东波

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

两个各有 n 个数值元素的独立数据库, 假设你 (持有数值 k) 可以用 $O(1)$ 的时间在其中任意一个数据库 (一次) 查询出第 k 小的元素[1]. 现要求以 $O(\log n)$(n 为查询次数) 的时间来确定这两个数据库 (共 $2n$ 个元素) 的中位数 (即第 n 小的元素).

Solution: 算法的思路是在第 1 个数据库中找出 1 个元素 a_i, 在第 2 个数据库中找出 1 个元素 b_j, 并保证小于等于 a_i 和 b_j 的元素总数为 N 个, 然后在所有小于 a_i 的元素中找出最大的元素 a_m, 在所有小于 b_j 的元素中找出最大的元素 b_n, 两个数据库所有的元素中第 n 小的元素就是 $\max(a_m, b_n)$. 算法1的伪代码描述如下:

Algorithm 1 算法 **FindMedian**(A, B)

Input: 数据库 A 和数据库 B 及预言机 **Oracle**
Output: 两个数组中的中位数 (即第 n 小的元素)

int $i = 1,\ j = n$;
if $a[n] \le b[1]$ **then** ▷ 使用 **Oracle** 查询数据库 A 中第 n 小的元素 $a[n]$ 和 B 中第 1 小的元素 $b[1]$
    return $a[n]$;
end if
if $a[1] \ge b[n]$ **then** ▷ 使用 **Oracle** 查询数据库 A 中第 1 小的元素 $a[1]$ 和 B 中第 n 小的元素 $b[n]$
    return $b[n]$;
end if
while $j \ge 3$ **do**
    int $k := (i + j)/2$;
    int flag $= a[k] - b[n-k+1]$; ▷ 使用 **Oracle** 查询 $a[k]$ 和 $b[n-k+1]$
    if flag > 0 **then**
        $j := k$;
    else if flag < 0 **then**
        $i := k$;
    else
        return $a[k]$;
    end if
end while
return $\max(a[k], b[n-k])$;
end {FindMedian}

设数据库 A 中第 k 小的元素为 a_k, 数据库 B 中第 k 小的元素为 b_k, 定义函数 f: $f(k) = a_k - b_{n-k+1}$, 显然 $f(k)$ 是关于 k 的单调递增函数. 特别地, 当 $f(1) \ge 0$ 时, b_n 就是第 n 小的元素; 当 $f(n) \le 0$ 时, a_n 就是第 n 小的元素. 若能找到一个 k, 使得: $f(k) \le 0$ 且 $f(k+1) \ge 0$. 这表明: $a_1 \le ... \le a_k \le b_{n-k+1}$ 且 $b_1 \le ... \le b_{n-k} \le a_{k+1}$. 设集合 $S = \{a_1, a_2, ..., a_k, b_1, b_2, ..., b_{n-k}\}$, S 的元素个数为 n. 由上可知: $\max(a_k, b_{n-k})$ 就是第 n 小的元素. 显然查询次数的函数 K 满足递推方程: $K(n) = K(n/2) + O(1)$, 根据主定理解得 $K(n) = O(\log n)$(其中 n 为查询次数).

注释: 如果不使用预言机, 而是调用 **PartSelect** 算法, 则查询第 k 小元素的需要消耗 $O(n)$ 的时间 (课上讲过), 从而 **FindMedian** 算法的时间复杂度递推方程为 $T(n) = T(n/2) + O(n)$, 解得 $T(n) = O(n)$(其中 n 为数据库元素个数).

[1]可以将这两个数据库都视作**预言机 (Oracle Machine)**, 即为可以在单一运算之内解答特定问题的黑盒式数据库.

Problem 2

给定一个二叉树 T, 请给出一个 $O(n)$ 算法来反转二叉树. 例如下面图1反转左二叉树, 我们得到右二叉树.

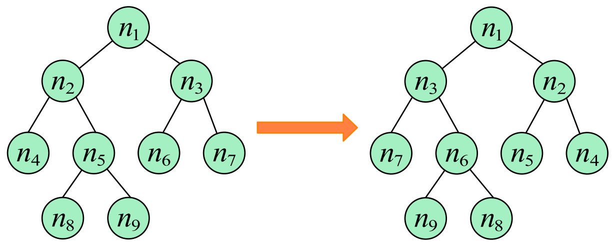

图 1: 翻转二叉树

Solution: 这是一道很经典的二叉树问题. 显然, 我们从根节点开始, 递归地对树进行遍历, 并从叶子节点先开始翻转. 如果当前遍历到的节点 root 的左右两棵子树都已经翻转, 那么我们只需要交换两棵子树的位置, 即可完成以 root 为根节点的整棵子树的翻转. 算法对应的伪代码过于简单就不予描述了, 直接给出其 C++ 代码:

```cpp
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};
/* 以上是二叉树的 ADT 定义 */
TreeNode* invertTree(TreeNode* root) {
    if (root == nullptr) {
        return nullptr;
    }
    TreeNode* left = invertTree(root->left);
    TreeNode* right = invertTree(root->right);
    root->left = right;
    root->right = left;
    return root;
}
```

现在分析一下时空复杂度: 由于要遍历二叉树中的每个节点, 且在其上交换两颗子树的时间是常数, 所以时间复杂度为 $T(n) = O(n)$; 递归所使用的栈空间主要由递归栈的深度决定, 而递归栈的深度就等于二叉树的高度 (即 $\log n$), 但最坏情形下树形呈链状, 则递归栈深度为 $O(n)$, 所以空间复杂度为 $O(n)$.

Problem 3

监狱有 N 个房间, 每个房间关押一个犯人, 有 M 种宗教, 每个犯人会信仰其中一种. 如果相邻房间的犯人的宗教信仰相同, 就可能发生越狱, 求有多少种状态可能发生越狱. 例如, 有 3 个房间和 2 种宗教, 然后会发生 6 种不同的状态.

Solution: 这本质上就是个排列组合问题. 每个房间的犯人都有 M 中信仰, 故总的状态数为 M^N. 根据题意, 只要有一对相邻房间的犯人宗教信仰相同, 则就会发生越狱事件. 考虑越狱的状态数会比较复杂, 所以**正难则反** (容斥原理), 考虑不发生越狱[2]的状态数: 第 1 个房间的犯人有 M 种选择, 而第 2 个房间的犯人有 $M-1$ 种选择, 且第 3 个房间的犯人有 $M-1$ 种选择[3], 同理第 4 个房间的犯人也有 $M-1$ 种选择 $\cdots$. 因此, 不发生越狱的状态数为 $M\cdot(M-1)^{N-1}$, 从而越狱的状态数为 $M^N - M\cdot(M-1)^{N-1}$.

显然 $O(1)$ 的公式已经给出来了, 接下来只需要套一下快速幂的模板 (递归 + 分治) 即可:

Algorithm 2 算法 **QuickMul**(x, N)

Input: 正实数 x (double), 正整数 N (long long)
Output: x^N

if $N == 0$ **then** ▷ 递归终止条件为 $N = 0$
  return 1.0;
end if
double $y =$ **QuickMul**$(x, N/2)$; ▷ 要计算 x^N 时, 先递归地计算出 $y = x^{\lfloor N/2 \rfloor}$
if $N\%2 == 0$ **then** ▷ 如果 N 为偶数, 则 $x^N = y^2$
  return $y * y$;
else ▷ 如果 N 为奇数, 则 $x^N = y^2 \times x$
  return $y * y * x$;
end if
end {QuickMul}

最后只需要调用一下即可: **return QuickMul**$(M, N) - M *$ **QuickMul**$(M-1, N-1)$. 上述伪代码稍微改一下就可以写出 C++ 代码, 就不予以展示了. 显然, 此公式调用了两次快速幂算法[4], 所以该问题解法的时间复杂度为 $T(N) = O(\log N) + O(\log N) = O(\log N)$, 空间复杂度 $S(N) = O(\log N)$.

[2] 即任意相邻房间对的犯人宗教信仰都不相同.
[3] 只要第 3 个房间犯人的信仰跟第 2 个房间犯人的信仰不同即可, 即使跟第 1 个房间相同也没事 (因为不相邻而不会发生越狱).
[4] 快速幂算法需要 $O(\log N)$ 的时间和 $O(\log N)$ 的空间.

Problem 4

给定一棵二叉树, 假设两个相邻节点之间的距离为 1, 请给出求解二叉树中任意两个节点的最大距离的方法.

Solution: 此题即求二叉树的直径, 任意一条路径均可以被看作由某个节点为起点, 从其左儿子和右儿子向下遍历的路径拼接得到. 如下图2, 我们可以知道路径 $[9, 4, 2, 5, 7, 8]$ 可以被看作以 2 为起点, 从其左儿子向下遍历的路径 $[2, 4, 9]$ 和从其右儿子向下遍历的路径 $[2, 5, 7, 8]$ 拼接得到.

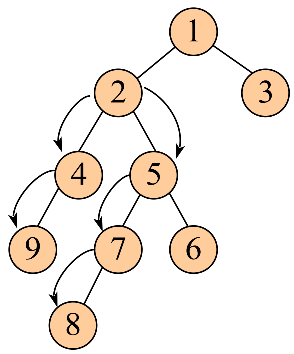

图 2: 路径的看法示意图

我们利用**深度优先搜索 (dfs)** 来遍历这颗二叉树, 对于所遍历到的每个非叶子节点: 再计算求得以其左儿子为根的子树深度 L 和右儿子为根的子树深度 R, 则以该节点为起点的路径长度的最大值 (即以该节点为根的子树的直径) 为 $L + R$.

而计算以 node 节点为根的子树深度: 我们可以定义递归函数 **dfs**(node), 先递归调用该函数求得以 node 左儿子和右儿子为根的子树深度 L, R, 则返回以 node 节点为根的子树深度 $\max(L, R) + 1$.

最后递归搜索 (dfs) 这颗二叉树的每个节点, 并设置全局变量 Diam 来记录并更新迭代各节点对应的子树直径, 从而求得整个二叉树的最大直径. 算法伪码 (C++ 代码与此类似, 就不罗列了) 描述如下:

Algorithm 3 算法 **dfs**(TreeNode node)

Input: 一棵二叉树 root ▷ TreeNode 数据类型的定义在前面已给出
Output: 二叉树 root 的直径 Diam

global Diam = 0; ▷ 定义全局变量来记录并更新以各节点为根的子树直径的最大值
if node == **null then** ▷ 递归终止条件
  return 0; ▷ 以叶子节点为根的子树深度为 0
end if
int Left = **dfs**(node→left); ▷ 递归计算以 node 左儿子为根的子树深度
int Right = **dfs**(node→right); ▷ 递归计算以 node 右儿子为根的子树深度
Diam = **max**(Left + Right, Diam); ▷ 更新迭代子树直径以求得最大值 (即整个二叉树的直径)
return max(Left, Right) + 1; ▷ 返回以当前节点为根的子树深度
end {dfs}

现在来分析以下算法的复杂度: 深度优先搜索二叉树, 时间复杂度显然就是 $O(N)$(N 为二叉树的节点个数); 由于递归函数在递归过程中需要为每一层递归函数分配栈空间, 所以这里需要额外的空间且该空间取决于递归的深度, 而递归的深度显然为二叉树的高度, 并且每次递归调用的函数里又只用了常数个变量, 所以所需空间复杂度为 $O(H)$(H 为二叉树的高度).

Problem 5

请给出三路快速排序的算法 (算法的时间复杂度显然仍是 $O(n\log n)$, 空间复杂度为 $O(\log n)$).

Solution: 一趟三路快排的结果: 随机化选取一个 pivot, 完成一趟三路快排, 数组会分为低 (<pivot 的元素)、中 (=pivot 的元素)、高 (>pivot 的元素) 三个分区. 一趟三路快排的思想核心是**三指针**, 具体来说是先将随机化选取好的 pivot 放到数组头部, pivot 紧右边的是低分区, 然后是中分区, 再然后是未知区域, 数组最右边的是高分区. 指针 lt 用来控制低分区的扩张, 指针 gt 用来控制高分区的扩张, 指针 i 用来遍历未知区域 (具体如下图3(a) 中所示). 一趟三路快排需用 **while** 循环, 循环条件为 $i < gt$. 当指针 i 所指元素 nums[i]==pivot 时, 中分区扩张 (即 i++); 当 nums[i] >pivot 时, 交换 nums[i] 和 nums[$gt-1$] 以使得元素 nums[i] 向高分区靠拢, 然后高分区扩张 (即 gt- -), 此时指针 i 不用动, 因为换过来的还是未知元素 (需要继续检索); 当 nums[i] <pivot 时, 交换 nums[i] 和 nums[$lt+1$] 以使得元素 nums[i] 向低分区靠拢, 与此同时低分区扩张 (即 lt++), 且此时指针 i 所指元素是已知的 (=pivot), 所以需要 i++ 来继续检索未知区域. 经过一趟三路快排则数组演变为下图3(b) 中所示. 算法伪代码见如下[5]:

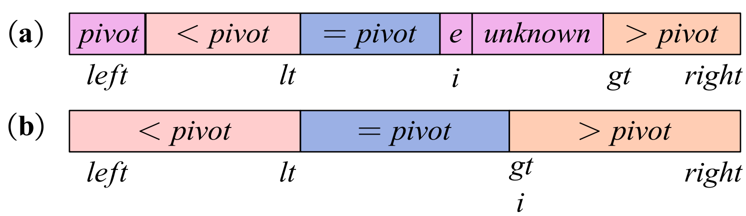

图 3: 三路快排分区示意图

Algorithm 4 三路快速排序 **QuickSort3** 算法

Input: 数组 $A[0,\cdots,n-1]$ 的子段 A[left, right]
Output: 排序后的数组子段 A[left, right]

```
if left >= right then                                        ▷ 递归终止条件
    return ;
end if
int rand_index = left + rand()%(right − left + 1);           ▷ 随机化选取主元 pivot
swap(A[left], A[rand_index]);                                ▷ 将选好的主元放到数组左端
int pivot = A[left];
int lt = left − 1, i = left, gt = right + 1;                 ▷ 三指针的初始化
while i < gt do                                              ▷ 当指针 i 和指针 gt 对撞时结束循环
    if A[i] == pivot then
        i++;                                                 ▷ 此时只需扩张中分区
    else if A[i] > pivot then                                ▷ 此时使元素 nums[i] 向高分区靠拢并扩张高分区
        swap(A[i], A[gt − 1]), gt- -;                        ▷ 并且指针 i 不动来继续检索当前所指的未知元素
    else if A[i] < pivot then                                ▷ 此时令元素 nums[i] 向低分区靠拢并扩张低分区
        swap(A[lt + 1], A[i]), lt++, i++;                    ▷ 并且指针 i 继续向前 (右移一位) 检索未知元素
    end if
end while
QuickSort3(A, left, lt);                                     ▷ 继续递归地对左边数组进行快排
QuickSort3(A, gt, right);                                    ▷ 继续递归地对右边数组进行快排
end {QuickSort3}
```

---

[5]当数组中存在大量的重复元素时, 双路快排的交换操作就显得很多余, 而三路快速排序恰好能解决这个问题.

以下是三路快排的 C++ 代码, 使用的话只需要执行 **QuickSort3**(nums, 0, nums.size() − 1) 即可.

```cpp
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
void QuickSort3(vector<int>& nums, int left, int right) {
    if(left >= right) {
        return;
    }
    int rand_index = left + rand()%(right - left + 1);
    swap(nums[left], nums[rand_index]);
    int pivot = nums[left];
    int lt = left - 1, i = left, gt = right + 1;
    while(i < gt) {
        if(nums[i] == pivot) {
            i++;
        }
        else if(nums[i] > pivot) {
            swap(nums[i], nums[gt - 1]);
            gt--;
        }
        else if(nums[i] < pivot) {
            swap(nums[lt + 1], nums[i]);
            lt++, i++;
        }
    }
    QuickSort3(nums, left, lt);
    QuickSort3(nums, gt, right);
}
```

并且借助一趟三路快排的思想可以解决**荷兰三色国旗问题** (伪码见如下算法5) 且能优化" 选取第 k 小元素" 的算法.

Algorithm 5 三色国旗问题的 **ThreeColor** 算法

Input: n 个实数构成的数组 $A[0, \cdots, n-1]$

Output: 负数排在正数前面且 0 排在中间的数组 $A[0, \cdots, n-1]$

int pivot = 0;
int lt = −1, i = 0, gt = n;
while i < gt **do**
  if $A[i]$ == pivot **then**
    i++;
  else if $A[i]$ > pivot **then**
    swap($A[i]$, $A[\text{gt}-1]$), gt- -;
  else if $A[i]$ < pivot **then**
    swap($A[\text{lt}+1]$, $A[i]$), lt++, i++;
  end if
end while
end {ThreeColor}

Problem 6

有 n 条绳子, 它们的长度分别为 L_i. 如果从它们中切割出 m 条长度相同的绳子, 这 m 条绳子每条最长能有多长?

Solution: 一个很直接的想法: 假如所切长度选的较短, 那么切割出来的段数就会 $\geq m$; 假如所切长度选的较长, 那么切割出来的段数就会 $< m$. 所以我们需要通过在区间 $[0, \textbf{INT\_MAX}]$ 中对所切长度进行二分搜索. **搜索 (二分) 准则**是: 取所切长度为区间中值 (mid), 按照此长度计算所有绳子中切出来的段数总和, 如果段数总和 $\geq m$, 则解位于右半区间 (即所切长度选短了)[6]; 若段数总和 $< m$, 则解位于左半区间 (即所切长度选长了). 于是我们可以给出以下算法6的伪代码:

Algorithm 6 切绳子 **CutRope** 算法

Input: n 个绳子长度构成的数组 $L[0, \cdots, n-1]$
Output: 割出 m 条最大长度且长度相同的绳段

```
int low = 0, high = INT_MAX;                      ▷ 初始化搜索区间
while low ≤ high do
    int mid = (low + high) >> 1;
    if mid == 0 then
        break;                                    ▷ 这是因为后面计算段数总和时, 分母是不能为零的!
    end if
    int cnt = 0;                                  ▷ 每次循环都需清 0
    for int i = 0; i < n; i++ do
        cnt += ⌊L[i]/mid⌋;                        ▷ 计算所有绳子按当前 mid 值所切的段数总和
    end for
    if cnt ≥ m then                               ▷ 舍弃左半区间, 在右半区间搜索最优可行解
        low = mid + 1;
    else                                          ▷ 舍弃右半区间, 在左半区间搜索最优可行解
        high = mid − 1;
    end if
end while
return high;                                      ▷ 返回所找到的最优可行解
end {CutRope}
```

不妨设 n 条绳子中的长度最大值为 $L_{\max}$, 则搜索区间可初始化为 $[0, L_{\max}]$, 于是 **CutRope** 算法 (二分) 的时间复杂度显然为 $O(\log L_{\max})$, 算法只需要常数级的额外辅助空间, 故空间复杂度为 $O(1)$.

至此, Chap 2 的作业解答完毕.

[6] 也可以认为此情形下的 mid 值就是**可行解**, 但不是**最优可行解**. 也就是说, 我们需要在另一半区间中继续二分搜索来找出**最优可行解**.