Homework2

231220030 邢俊书

2025 年 3 月 18 日

Solution 4.1.

不妨记二叉树中总节点个数为 n，叶节点个数为 n_0，有一个子节点的节点个数为 n_1，有两个子节点的节点个数为 n_2，不难得出有如下等式成立：

$$\begin{cases} n = n_0 + n_1 + n_2 \\ n - 1 = n_1 + 2n_2 \end{cases}$$

两式相减，得：$n_2 = n_0 + 1$。因为二叉树高度为 h，所以有 $n \leq 2^{h+1} - 1$ 成立，即 $n_0 + n_1 + n_2 \leq 2^{h+1} - 1$，又 $n_0 + n_1 + n_2 = 2n_0 + n_1 - 1 \geq 2n_0 - 1$，因此 $2n_0 - 1 \leq 2^{h+1} - 1$，$n_0 \leq 2^h$，亦即 $L \leq 2^h$。

Solution 4.4.

为了方便起见，不妨先约定一个 swap() 函数：

```
void swap(int& a, int& b) {
    int tmp = b;
    b = a;
    a = tmp;
}
```

(1) 以下给出的算法本质上是在简单情况下的归并排序：

```
void MySort(int a[]) {
    if (a[0] > a[1]) swap(a[0], a[1]);
    if (a[2] > a[3]) swap(a[2], a[3]);
    int b[4];
    int i = 0, j = 2, k = 0;
    while (i <= 1 && j <= 3) {
        if (a[i] <= a[j]) b[k++] = a[i++];
        else b[k++] = a[j++];
    }
    while (i <= 1) b[k++] = a[i++];
    while (j <= 3) b[k++] = a[j++];
    for (int i = 0; i <= 3; i++) {
        a[i] = b[i];
    }
    return;
}
```

在 while 循环开始前进行了 2 次比较，在 while 循环开始后，由鸽笼原理可知，至多进行 3 次比较，故该算法在最坏情况下可以只利用 5 次比较对 4 个元素进行排序。

(2) 不妨记待排序的五个数分别为 A、B、C、D、E。先比较 A 与 B、C 与 D 的大小关系，不失一般性，假设 $A > B$、$C > D$，再比较 A 与 C 的大小关系，不失一般性，假设 $A > C$，至此进行了 3 次比较操作，得到了这样一个序列 $A > C > D$。现在我们考虑将 E 插入到这个序列之中：易知，仅需先将 E 与 C 比较，再与 A 或 D 比较即可。最后将 B 插入得到的 C、D、E 序列中，操作与将 E 插入 $A > C > D$ 的序列之中类似。故最坏情况下，该算法仅需 7 次比较，即可将 5 个元素排序。具体代码可见附件。

下证在最坏情况下，该算法具有最优性：初始情况下，5 个元素的不同排列共有 $5! = 120$ 种，故决策树的叶节点至少应有 120 个，若仅进行 6 次比较，至多有 $2^6 = 64$ 个叶节点，故最坏情况下，至少需要 7 次比较才能实现对 5 个元素的排序。

Solution 4.8.

首先调用 nth-element() 函数以 O(n) 的时间复杂度计算数组的中位数，随后部分划分数组，使得数组左半元素均小于中位数，右半元素均大于中位数，再递归处理左右两部分直至将数组分为 k 段。

算法的时间复杂度满足 $f(n) = 2f(n/2) + O(n)$，易知，算法运行 logk 次后终止，故总时间复杂度为 O(nlogk)。

Solution 4.9.

首先随机选取一个螺钉，将所有螺母与之匹配，可以找到与该螺钉匹配的螺母，同时也将所有螺母分为两部分，一部分小于该螺钉，一部分大于该螺钉，再取与该螺钉匹配的螺母，将剩余所有螺钉与之匹配，同样可以将螺钉分为两部分，递归处理左右两部分的子问题即可解决问题。

算法每次都会选取一个 pivot 元素，再以 O(n) 的时间复杂度将所需处理的两组数据各自分为大小两部分，故算法本质上等效于两次快排操作，时间复杂度为 $O(nlogn)$。

Solution 4.11.

(1) 不妨假设存在 i、j，使得 (i, j) 为逆序对且 j - i > 2，则 A[i] 与 A[j] 之间至少存在两个元素，记其中两个元素为 A[k]、A[l] 且 k < l，由逆序对性质可知，A[i] > A[j]。

若 A[k] > A[i]，则 (k, j) 也是逆序对，A[k] < A[l] < A[j] 必然不成立，(k, l) 与 (l, j) 之间必然存在一个逆序对，与数组 A 至多有 2 个逆序对矛盾。

若 A[k] < A[i]，则 (i, k) 也是逆序对，若 A[k] < A[l] < A[j] 成立，则 (i, l) 也是逆序对，否则 (k, l) 与 (l, j) 之间必然存在一个逆序对，与数组 A 至多

有 2 个逆序对矛盾。

综上，假设不成立，若 (i, j) 为逆序对，则 $j - i \leq 2$。

(2) 假设 (i, j) 为逆序对，若 j - i = 1，则 A[i] 与 A[j] 相邻；若 j - i = 2，记中间元素为 A[k]，则 A[i] < A[k] < A[j] 必然不成立，(i, k) 与 (k, j) 之间必然存在一个逆序对，即无论 j - i 的取值，数组中必然存在一对逆序对，其两个元素相邻。故算法思路如下：

先遍历数组，找到第一个逆序对 (i, i + 1)，此时 (i - 1, i + 1) 与 (i, i + 2) 之间可能存在逆序对。首先交换 A[i] 与 A[i + 1]，使得数组部分有序，交换后 (i - 1, i) 与 (i + 1, i + 2) 之间可能存在逆序对，(i + 1, i + 2) 在向后遍历的过程中会被检查，故向前检查 A[i - 1] 与 A[i]，若 A[i - 1] > A[i]，则交换 A[i - 1] 与 A[i]，并退出循环；否则继续向后进行比较，直到遍历完数组或找到并交换完第二个逆序对。

由于遍历完数组仅需 n - 1 次比较，且在整个遍历过程中，至多向前比较 1 次，故算法在最坏情况下的比较次数不超过 n 次。

Solution 4.14.

先对所有单词进行预处理，将单词中所有字母以字母序重组，例如，“eat”与“tea”都会被重组为"aet”，再以每个单词重组后得到新字母序列为键值，将所有单词映射至哈希表中，最后遍历所有哈希表，找出其中元素大于等于 2 的即可。

Solution 7.1.

暴力算法仅需要以两层循环遍历每一个二元组即可，易知时间复杂度为 $O(n^2)$。

若要优化时间复杂度，可以参照归并排序求解传统逆序对的思路，修改合并子问题时的判断条件即可，可将时间复杂度优化至 O(nlogn)，具体代码如下：

```
int merge_sort(int A[], int l, int r) {
    int cnt = 0;
    if (l >= r) return 0;
    int mid = (l + r) >> 1;
    cnt += merge_sort(A, l, mid);
    cnt += merge_sort(A, mid + 1, r);
    int i = l, j = mid + 1, k = 0;
    int tmp[r - l + 1];
    while (i <= mid && j <= r) {
        if (A[i] <= C * A[j]) {
            tmp[k++] = A[i++];
        } // 修改的判断条件
        else {
            tmp[k++] = A[j++];
            cnt += mid - i + 1;
        }
    }
    while (i <= mid) tmp[k++] = A[i++];
    while (j <= r) tmp[k++] = A[j++];
    for (int s = l, t = 0; s <= r; s++, t++) {
        A[s] = tmp[t];
    }
    return cnt;
}
```

Solution 7.4.

(1) 将一个长度为 mn 的有序数组与一个长度为 n 的有序数组合并的最坏时间代价为 c(m + 1)n，故题述方案的时间复杂度为 $\sum_{i=1}^{k-1} c(i+1)n = \frac{(k+2)(k-1)}{2}n = O(nk^2)$。

(2) 将需要合并的 k 个数组分为两堆，若堆中恰有两个数组，则以归并排序类似的方式合并两数组，否则继续将堆进行划分，最后将已合并的两个堆合并。时间复杂度满足，f(k) = 2f(k / 2) + nk，易知时间复杂度为 O(nklogk)。

Solution 7.5.

(1) 不妨先约定树节点的结构体为：

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
};
```

算法如下：

```
int getDepth(TreeNode* root) {
    int depth = 0;
    if (root == nullptr) return 0;
    if (root->left != nullptr) {
        depth = max(depth, getDepth(root->left) + 1);
    }
    if (root->right != nullptr) {
        depth = max(depth, getDepth(root->right) + 1);
    }
    return depth;
}
```

(2) 算法如下：

```
int maxDiameter = 0;

int dfs(TreeNode* root) {
    if (!root) return 0;
    int leftDepth = dfs(root->left);
    int rightDepth = dfs(root->right);
    maxDiameter = max(maxDiameter, leftDepth + rightDepth);
    return max(leftDepth, rightDepth) + 1;
}

int treeDiameter(TreeNode* root) {
    maxDiameter = 0;
    dfs(root);
    return maxDiameter;
}
```

Solution 7.8.

(1) 先调用排序算法对所有点按照横坐标大小进行从小到大的排序，接着从大到小开始遍历所有点，横坐标最大的必然是 maxima。向前遍历时，维护一个 y 来记录已遍历点中纵坐标最大的点，若遍历过程中有点的纵坐标大于 y，则该点也是 maxima，同时更新 y；否则某点必然不是 maxima。

先以 O(n) 的时间代价计算横坐标的中位数，根据中位数将所有点划分为两部分，递归解决左右两部分的 maxima。对于右半部分，其 maxima 必为局部的 maxima，对于左半部分，需要与右半部分维护的 y 进行比较。

(2) 算法不正确，因为递归的过程中划分不一定均匀，不可能每次递归都恰好将数组划分为四等块，故递归式并不满足。

Solution 7.12.

(1) 分别遍历 k 行，统计每一行中 1 出现的次数。若为偶数，则缺失的比特串相应位置为 0；否则为 1。

(2) 不难发现，每遍历完一行，均可以将问题规模缩减至 $\frac{n-1}{2}$。被排除的 $\frac{n+1}{2}$ 个元素往后的每一行，0 和 1 出现的次数均相等，否则由鸽笼原理可知，其中必然会存在相同元素。时间复杂度满足 f(n) = f(n / 2) + n，由主定理可知，时间复杂度为 O(n)。

Solution 14.1.

假设一个堆中共有 h 个元素，则堆的高度为 $\lceil log(h+1) \rceil - 1$，删除所有的叶子节点后的高度为 $\lceil log(\lfloor \frac{1}{2}h \rfloor + 1) \rceil - 1$，余下的堆的高度必然比原来的堆少 1，故等式成立。

Solution 14.2.

为方便讨论，不妨假定给定堆已被组织为二叉树形式，并约定树节点的结构体为：

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
};
```

考虑维护一个最大堆 maxHeap，初始情况下，将给定堆的堆顶元素入堆，显然这个元素就是最大元素，由堆的偏序关系可知，第 2 大元素仅可能为这个元素的子节点，将其子节点入堆并将 maxHeap 堆顶元素出堆。如此不断地将 maxHeap 堆顶元素出堆并将其子节点入堆，即可保证所维护的堆中永远是所有可能的第 i 大的元素所构成的集合。算法的时间复杂度为 $\sum_{i=1}^{k} logi = logk! = O(klogk)$。

具体代码如下：

```
int get_kth_element(TreeNode* root, int k) {
    if (!root) return -1;
    maxHeap.push(root);
    int cnt = 0;
    TreeNode* tmp = nullptr;
    while (cnt < k && !maxHeap.empty()) {
        tmp = maxHeap.top();
        maxHeap.pop();
        cnt++;
        if (tmp->left) maxHeap.push(tmp->left);
        if (tmp->right) maxHeap.push(tmp->right);
    }
    return tmp ? tmp->val : -1;
}
```

Solution 14.3.

先证 D-ARY-CHILD(i, j) 的正确性：

记 P(i)：$\forall 1 \leq j \leq d$，D-ARY-CHILD(i, j) = d(i - 1) + j + 1。

显然 D-ARY-CHILD(1, j) = j + 1，所以 P(1) 成立。

假设当 $n \leq i$ 时，P(n) 成立，下证 P(i + 1) 成立。

不难发现 D-ARY-CHILD(i + 1, j) = D-ARY-CHILD(i, d) + j = d(i - 1) + d + 1 + j = di + j + 1，成立。

由数学归纳法可知，D-ARY-CHILD(i, j) 正确。

再证 D-ARY-PARENT(i) 的正确性：

显然根节点成立，接着考虑非根节点，由上述证明可知，对于任意下标为 n 的节点，都可以写为 n = d(i - 1) + j + 1，代入 D-ARY-PARENT() 函数，$\lfloor \frac{i-2}{d} + 1 \rfloor = \lfloor \frac{d(i-1)+j-1}{d} + 1 \rfloor = i$，正确性得证。

Solution 14.4.

不妨先考虑完美二叉树的所有节点高度之和，假设树高度为 h，则所有节点的高度之和为 $\sum_{i=1}^{h-1} i * 2^{h-1-i} = 2^h - h - 1 = n - \lceil logn \rceil$。此时若再加入一个节点，则高度之和恰变为了 n，即题目所需的取等的情况。

记 P(h)：对于高度为 h 的堆，所有节点之和最多为 n - 1。

显然高度为 0 时，所有节点高度之和为 0，P(0) 成立。

假设当 $h \leq k$ 时，P(h) 成立，下证 P(k + 1) 成立。

显然一个堆的根节点的左右子堆大小分别为 n_1、n_2，记左右子堆所有节点的高度之和分别为 H_1、H_2，则该堆所有节点的高度之和为 $H_1 + H_2 + \lceil logn_1 \rceil$。

对左右子树是不是完美二叉树进行讨论，发现会出现两种情况：

左子树为完美二叉树，则 $H_1+H_2+\lceil logn_1 \rceil \leq n_1-\lceil logn_1 \rceil+n_2-1+\lceil logn_1 \rceil < n_1 + n_2 + 1 - 1$，成立。

右子树为完美二叉树，则 $H_1 + H_2 + \lceil logn_1 \rceil \leq n_1 - 1 + n_2 - \lceil logn_1 \rceil + 1 + \lceil logn_1 \rceil = n_1 + n_2 + 1 - 1$，成立。

综上，在一个有 n 个节点的堆中，所有节点的高度之和最多为 n - 1。

Solution 14.5.

首先以 k 个已排序链表的头节点创建一个最小堆 minHeap，每次将堆顶元素出堆并将相应节点的后继节点 (如果存在) 入堆，这样可以保证 minHeap 中元素永远是剩余所有元素中最小元素的可能元素构成的集合。最坏情况下，除剩余元素数小于 k 时外，每次维护堆的代价为 logk，执行 n 次后终止，故时间复杂度为 O(nlogk)。

Solution 14.6.

维护两个堆 maxHeap、minHeap 分别存储较小的与较大的一半元素，同时保证 $minHeap.size() \leq maxHeap.size() \leq minHeap.size() + 1$。当输入元素数为偶数时，中位数为两个堆堆顶元素的平均数；当输入元素数为奇数时，中位数即为 maxHeap 的堆顶元素。

```
void addNum(int num) {
    if (maxHeap.empty() || num <= maxHeap.top()) {
        maxHeap.push(num);
    }
    else minHeap.push(num);
    if (maxHeap.size() > minHeap.size() + 1) {
        minHeap.push(maxHeap.top());
        maxHeap.pop();
    }
    else if (minHeap.size() > maxHeap.size()) {
        maxHeap.push(minHeap.top());
        minHeap.pop();
    }
}

double findMedian() {
    if (maxHeap.size() > minHeap.size()) {
        return maxHeap.top();
    }
    return (maxHeap.top() + minHeap.top()) / 2.0;
}
```

删除操作可以考虑通过哈希表 + 延迟删除实现。