

4.11 逆序对

1)

证明:

采用反证法。假设 (i, j) 为逆序对且 $j - i > 2$, 则考虑子数组 $A[i], A[i + 1], A[i + 2], \dots, A[j]$. $\forall k \in (i, j)$, $A[k]$ 的取值有三种可能:

- ①若 $A[k] < A[j] < A[i]$, 则有逆序对 (i, k) ;
- ②若 $A[j] < A[k] < A[i]$, 则有逆序对 $(i, k), (k, j)$;
- ③若 $A[j] < A[i] < A[k]$, 则有逆序对 (k, j) 。

为尽量减少逆序对的数量, 假设 $\forall i < k < m < j, A[k] < A[m]$, 则除了 (i, j) 以外, $A[k], A[m]$ 至少贡献两个逆序对, 与原命题矛盾。故假设不成立, 即 $j - i \leq 2$ 。

2)

算法主干为依次比较 $A[i]$ 与 $A[i + 1]$, 最坏情况需比较 $n - 1$ 次; 当有 2 个逆序对时, 可能需要额外比较一次。故最坏情况下需比较 n 次。

```
1  bool is_inversion = 0;
2  for i=1 to n-1 do
3  {
4      if A[i]>A[i+1]
5      {
6          if i==n
7              swap(A[i], A[i+1]);
8              is_inversion=0;
9          else if is_inversion==1
10             swap(A[i-1], A[i+1]);
11             is_inversion=0;
12         else if A[i-1]>A[i+1]
13             swap(A[i-1], A[i], A[i+1]) to (A[i+1], A[i-1], A[i+1]);
14             is_inversion=0;
15         else
16             is_inversion=1;
17     }
18     else
19     {
20         if is_inversion==1
21             if A[i-1]>A[i+1]
22                 swap(A[i-1], A[i], A[i+1]) to (A[i], A[i+1], A[i-1]);
23                 is_inversion=0;
24             else
25                 swap(A[i-1], A[i]);
26                 is_inversion=0;
27     }
28 }
```

4.14 易位词

先提取出英文文件中的所有单词，并按单词长度分组。将单词内部按字典序重排，然后在各个组内对排序后的单词进行排序。相邻且相同的单词即为易位词。

第七章

7.1 逆序对计数问题的推广

对数组进行遍历检查逆序对，时间复杂度为 $O(n^2)$ 。

7.4 合并有序数组

1)

$$T(n) = \sum_{i=1}^{k-1} (i \cdot n + n) = \frac{(k-1)(k+2)}{2} \cdot n = O(k^2 n).$$

2)

同时对 k 个数组采用合并排序中的合并方法，其中采用最小堆选择 k 个元素中的最小者。每次选择最小元素的时间复杂度为 $\log k$ ，一共进行 $n \cdot k$ 次选择，故时间复杂度为 $O(nk \log k)$ 。

7.5 二叉树

1)

记树的高度为 h ，则 $h = \max(h_{left}, h_{right}) + 1$ 。

$$T(n) = T(n_{left}) + T(n_{right}) + 1 = O(n).$$

2)

记树的直径为 d ，则 $d = \max(d_{left}, d_{right}, h_{left} + h_{right} + 2)$ 。

$$S(n) = S(n_{left}) + S(n_{right}) + T(n_{left}) + T(n_{right}) + 1 = O(n).$$

7.8 寻找 maxima

1)

可以排序：先对所有点的横坐标 x 进行排序，得到序列 $X[1 \dots n]$ ，然后按下标从大到小进行遍历

```
1 maxima_y=0;
2 for i=n to 1
3 {
4     if (X[i].y > maxima_y)
5     {
6         X[i] is maxima;
7         maxima_y = X[i].y;
8     }
9 }
```

3)

基于比较的排序算法的时间复杂度为 $\Omega(n \log n)$ ，故基于比较寻找 maxima 的算法的时间复杂度不可能为 $o(n \log n)$ 。

7.12 寻找缺失的比特串

1)

所有可能的 2^k 个 k 比特串应该满足每一位共有 2^{k-1} 个 0 和 2^{k-1} 个 1，故遍历 k 行，找出每一行缺失的 0 或 1，即可拼出缺失的比特串。

2)

假设二位比特数组中的比特串为 $s_i[1 \dots k]$ ($i \in (1, n)$)，缺失的比特串为 $m[1 \dots k]$ ，与 1) 类似地，先遍历第一行，确定 $m[1] = b_1$ ，接下来只考虑集合 $\{\forall i, s_i[1] = b_1 \mid s_i[2 \dots k]\}$ ；重复上述步骤。

$$T(n) = \sum_{i=1}^k 2^k - 1 = 2^{k+1} - 2 - \frac{k(k+1)}{2} = 2n - \frac{k(k+1)}{2} = O(n).$$

第十四章

14.1 log 的性质证明

证明：

对于堆， $\lceil \log(h+1) \rceil$ 为查找一个节点所需的时间复杂度， $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil$ 为查找它的父节点所需的时间复杂度，而后再查找一次即可得到前者，故 $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil + 1 = \lceil \log(h+1) \rceil$ 。

14.2 堆中第 k 大的元素

只取堆的前 k 层，进行 k 次 pop 操作，每次调整的时间复杂度为 $\log 2^k O(k)$ ，总体的时间复杂度为 $O(k^2)$ 。

14.3 d 叉堆

1)

证明：

下标为 m 的节点，它的子节点下标为 $d(m-1)+2$ 到 $dm+1$ 。令子节点下标为 i ，则 $i \in [d(m-1)+2, dm+1]$ ，即 $m = \lfloor \frac{i-2}{d} + 1 \rfloor$ 。

2)

证明：

同上所述，下标为 i 的节点的第 j 个子节点下标为 $d(i-1)+2+j-1 = d(i-1)+j+1$ 。

14.4 节点高度和

证明：

假设堆的高度为 H ，则对于完美堆，共有 $n = 2^{H+1} - 1$ 个节点，所有节点的高度之和为

$$H(n) = \sum_{i=0}^H (i \cdot 2^{H-i}) = 2^{H+1} - H - 2 = n - 1 - H < n - 1$$

当堆不是完美堆时，假设堆比起完美堆缺失了 m ($m \in [1, 2^H - 1]$) 个节点，则 $n = 2^{H+1} - 1 - m$ ， $H(n)$ 减少了 $D(m) = \sum_{i=1}^{\infty} \lfloor \frac{m}{2^i} \rfloor$ ，所有节点的高度之和为

$$H'(n) = H(n) - D(m).$$

考虑 $n - 1 - H'(n) = H - (m - D(m))$ 为与 m 有关的函数，且随 m 增大，函数值不减。因此当 m 取最大 $2^H - 1$ 时，函数取得最小值 0，即 $H'(n) \leq n - 1$ 。

综上所述，原命题成立且在 $m = 2^H - 1$ ， $n = 2^H$ 时取等。

14.5 有序链表合并算法

先用各个链表的表头元素构建 k 个元素的最小堆，弹出堆顶后用堆顶所属链表的下一个元素替换堆顶；重复操作直至所有链表为空；然后依次弹出堆顶直至堆空。构建堆的时间复杂度为 $O(k)$ ，修复最小堆的时间复杂度为 $O(\log k)$ ，共修复 n 次，故算法的时间复杂度为 $O(k + n \log k) = O(n \log k)$ 。

14.6 动态发现中值

假设数组大小为 n ，维护一个最小堆 H_{min} 用于存储前 $\lceil \frac{n}{2} \rceil$ 大的数，和一个最大堆 H_{max} 用于存储后 $\lfloor \frac{n}{2} \rfloor$ 大的数。

插入元素 x ：

若 $H_{min}.size = H_{max}.size$ 且 $x < H_{min}.top$ ，则先 $H_{max}.push(x)$ ，然后弹出 $H_{max}.top$ 并将其插入 H_{min} ；

若 $H_{min}.size = H_{max}.size$ 且 $x \geq H_{min}.top$ ，则 $H_{min}.push(x)$ ；

若 $H_{min}.size > H_{max}.size$ 且 $x < H_{min}.top$ ，则 $H_{max}.push(x)$ 。

若 $H_{min}.size > H_{max}.size$ 且 $x \geq H_{min}.top$ ，则先 $H_{min}.push(x)$ ，然后弹出 $H_{min}.top$ 并将其插入 H_{max} 。

删除中位数：

若 $H_{min}.size = H_{max}.size$ ，则先弹出 $H_{min}.top$ ，然后弹出 $H_{max}.top$ 并将其插入 H_{min} ；

若 $H_{min}.size > H_{max}.size$ ，则弹出 $H_{min}.top$ 。

查找中位数：

若 $H_{min}.size = H_{max}.size$ ，则取 $\frac{H_{min}.top + H_{max}.top}{2}$ ；

若 $H_{min}.size > H_{max}.size$ ，则取 $H_{min}.top$ 。