

算法习题课

2024.4.18 黄云鹏

P1

1.2

1.2: (3个数的中位数)

输入3个各不相同的整数

1) 题目：请设计一个寻找3个数的中位数的算法。

- 先找最大数，再在剩下的两数中找较大数；（还有其他做法）

```
Algorithm: MIDDLE(A[1,2,3])
max := MAX(A[1],A[2]);      /*需且仅需1次比较*/
min := MIN(A[1],A[2]);      /*需且仅需1次比较*/
if A[3] > max then
    return max;
else
    if A[3] > min then
        return A[3];
    else
        return min;
```

2) 题目：在最坏情况下、平均情况下你的算法分别需要进行多少次的比较？(假设所有可能的输入等概率出现)

- 解析：最坏情况是比较次数最多的全逆序，平均情况可以直接枚举所有情况再求期望(注：不同算法答案可能不同)；
- 解答如下：(以三个整数 \in 集合 1, 2, 3 为例)

最坏情况下： $A[1, 2, 3] = 1, 3, 2$ 或 $3, 1, 2$ 或 $2, 3, 1$ 或 $3, 2, 1$ ，需要3次比较；

平均情况下：共6种情况，每种情况等概率 $1/6$ ，只有 $1, 2, 3, 2, 1, 3$ 种情况只需要2次比较，其余均需要3次比较，故平均需要： $\frac{2}{6} \times 2 + \frac{4}{6} \times 3 = \frac{8}{3}$ 次比较；

3) 题目：在最坏情况下找出3个不同整数的中位数至少需要多少次比较？请证明你的结论。

- 解析：可采取对手论证的思路。

- 解答：最坏情况下求 3 个整数中位数至少需要 3 次比较，证明如下：

设输入的三个整数分别为 a, b, c ，不妨设第 1 次比较结果为 $a > b$ ，

则第 2 次比较在 c 和 a 或 b 之间：1) c 和 a 比较，若 $c > a$ 则中位数

为 a ，若 $c < a$ ，此时 c 和 b 不确定哪个是中位数，故至少还需一次比

较；2) c 和 b 比较，若 $c < b$ 则中位数为 b ，若 $c > b$ ，此时 a 和 c 不

确定哪个是中位数，故至少还需一次比较；

综上：无论什么算法，最坏情况下至少需要 3 次比较；

1.3

* 1.3: (集合最小覆盖问题)

1) 前几大集合可能会有很多重复覆盖, 反例如下:

$$U = \{1, 2, 3, 4, 5\}, S_1 = \{1, 2, 3, 4\}, S_2 = \{1, 2, 3\}, S_3 = \{5\} \Rightarrow$$

输出: $\{S_1, S_2, S_3\}$, 正确答案为: $\{S_1, S_3\}$

2) 只要求输出一个集合覆盖(可行解), 不一定是最小覆盖(最优解), 因此可以设计多种算法:

Alg1: 直接判断最大并集 $\bigcup_{i=1}^m S_i$ 是否为集合覆盖;

Alg2: 针对每一个元素 $i \in \{1, \dots, n\}$, 寻找包含 i 的集合 S_i , 并加入覆盖中;

Alg3: 贪心, 每次寻找当前包含最多未覆盖元素的集合 S_{max} , 并加入覆盖中;

Alg4: 枚举, 搜索所有可能的集合覆盖 $\{S_{i_1}, \dots, S_{i_k}\}$, 并选出最小的一个;

3) 最小集合覆盖是NP问题, 因此上述四种算法只有 $Alg4$ 能保证总是得出最小覆盖, 特别地,
针对 $Alg3$, 给出如下反例, 说明其不能保证总是得到最优解:

$$U = \{1, 2, 3, 4, 5, 6\}, S_1 = \{1, 2, 3\}, S_2 = \{1, 4\}, S_3 = \{2, 5\}, S_4 = \{3, 6\} \Rightarrow$$

输出: $\{S_1, S_2, S_3, S_4\}$, 正确答案为: $\{S_2, S_3, S_4\}$

1.7

1.7: (多项式计算)

题目: HORNER算法是用来计算多项式 $P(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$ 的, 请证明其正确性。

- 解析: HORNER算法本质即为**秦九韶算法**, 采用了迭代的思想, 故正确性证明可采用同样使用了迭代思想的**数学归纳法**;
- 证明: i) 当 $n = 1$ 时, $p = a_0 = P(x)$, 易知正确;

ii) 假设当 $n = k$ 时算法正确, 即 $p = P(x) = \sum_0^k a_i x^i$; 则当 $n = k + 1$

时, 由假设知, 循环的倒数第二次: $p = \sum_1^{k+1} a_i x^i - 1$, 故最后一遍循环:

$$p = p \times x + A[0] = \sum_1^{k+1} a_i x^i + a_0 = \sum_0^{k+1} a_i x^i$$
 综上: 归纳可得算法对一切 n

成立;

1.8

1.8: (整数相乘)

INT-MULT算法用来计算两个非负整数 y 、 z 的乘积

- 解析: 由算法易得算法的递推公式: $F(y, z) = F(cy, \lfloor \frac{z}{c} \rfloor) + y \times (z \% c)$ 设

$z = z_n, z_i = c \times z_{i-1} + r_i$, 即 $\frac{z_i}{c} = z_{i-1}, z_i \% c = r_i$, 特殊地: $z_0 = 0$ 故

递推公式可变形为: $F(y, z_n) = y \times \sum_0^{n-1} c^i \times r_{n-i}$ 通过变形其实可以清楚认

识到**算法的本质其实就是得到 z 的 c 进制形式**, 然后逐位与 y 相乘, 故算法的进行(递归的层数)与 y 无关(y 仅作为一个系数参数)因此任意 y , 当算法对所有的 $z \in N$ 成立时, 即可得算法的正确性;

- 算法正确性证明如下： 证明) i) 当 $z = 1$ 时, $F(y, 1) = y \times c^0 = y \times 1$, 易知正确;

ii) 假设当 $z \leq k(k > 1)$ 时算法成立, 即 $y \times z = F(y, z), (z \leq k)$, 则当 $z = k + 1$ 时, $F(y, k + 1) = F(cy, \lfloor \frac{k+1}{c} \rfloor) + y \times ((k + 1)\%c)$,

令 $\lfloor \frac{k+1}{c} \rfloor = q, (k + 1)\%c = r$, 由带余除法知: $k + 1 = cq + r$, 则 $F(y, k + 1) = F(cy, q) + y \times r = y \times cq + y \times r = y \times (cq + r) = y \times (k + 1)$, 成立;

综上: 归纳可得算法对所有的 $z \in N$ 成立, 故算法正确, 得证;

1.9

1.9: (平均复杂度计算)

将算法执行*operations*的代价次数定义为随机变量 X , 则题干所给出的即是 X 的分布律, 而平均情况下的时间复杂度就是 X 的期望 EX , 按照期望公式计算即可:

$$\begin{aligned} EX(n) &= \sum_{i=1}^n Pr(i)X(i) = \\ \frac{1}{n} \times (10 \times \frac{n}{4}) + \frac{2}{n} \times (20 \times \frac{n}{4}) + \frac{1}{2n} \times (30 \times \frac{n}{4}) + \frac{1}{2n} \times (n \times \frac{n}{4}) \\ &= \frac{n+130}{8} \end{aligned}$$

1.10

* 1.10: (UNIQUE算法分析)- part1

1) 该算法用于判断数组中每个元素是否唯一，最坏时间复杂度直接计算最大循环次数即可：

$$worstCost = (n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n - 1)}{2} \Rightarrow O(n^2)$$

* 1.10: (UNIQUE算法分析)- part2

2) n 个元素中仅2个元素相等，且每种输入等概率，即概率均为 $\frac{1}{C_n^2}$ ，输入代价即为所有可能的

循环次数*i*, ($i \in [1, \frac{n(n-1)}{2}]$), 直接计算代价的期望即可:

$$averageCost = \sum_{i=1}^{\frac{n(n-1)}{2}} \frac{1}{C_n^2} \cdot i = \frac{n^2 - n + 2}{4} \Rightarrow O(n^2)$$

* 1.10: (UNIQUE算法分析)- part3

3) 由于任意位置元素等概率地从 $[1, 2, \dots, k]$ 中选取, 故:

- 任意两个位置 i, j 上元素相等的概率为: $P(A[i] = A[j]) = k \times (\frac{1}{k})^2 = \frac{1}{k}$,
- 任意两个位置 i, j 的元素不相等的概率为 $P(A[i] \neq A[j]) = 1 - \frac{1}{k}$;
- 若循环代价为 i , 则意味着前 $i - 1$ 次比较的元素对不等, 第 i 次相等, 因此循环代价近似服从几何分布 $G(\frac{1}{k})$, 直接通过公式得到期望代价即可:

$$averageCost = \frac{1}{\frac{1}{k}} = k \Rightarrow O(k)$$

* 1.10: (UNIQUE算法分析) - part4

3) 另注：上述循环代价之所以是“近似”服从几何分布，是因为几何分布随机变量的取值范围是正整数集合 N_+ ，而本题中循环代价的取值范围为 $[1, \dots, \frac{n(n-1)}{2}]$ ，因此直接代入几何分布的期望公式不够严谨，最准确的做法是直接计算求和式：

$$\text{averageCost} = \sum_{i=1}^m i(1-p)^i p, \text{ 其中 } p = \frac{1}{k}, m = \frac{n(n-1)}{2}$$

$$\Rightarrow \text{averageCost} = \frac{1}{p} - \left(m + \frac{1}{p}\right)(1-p)^m, \text{ 易知 : }$$

$$\text{当 } n \rightarrow \infty \text{ 时, } m \rightarrow \infty, \text{ averageCost} \rightarrow \frac{1}{p} = k \Rightarrow O(k)$$

2.2

2.2: (取整函数)

题目: 证明: 对于任意整数 $n \geq 1$, $\lceil \log(n+1) \rceil = \lfloor \log n \rfloor + 1$ (提示: 将n划分为 $2^k \leq n \leq 2^{k+1} - 1$)。

- 解析: 根据提示将n进行划分, 根据取整函数的定义用k表示取整函数, 即可证明;
- 证明如下:

因为对于任意整数 $n \geq 1$, 可划分为 $2^k \leq n \leq 2^{k+1} - 1$, 则:

$$\begin{aligned} \textcircled{1} \quad k+1 &= \lceil \log(2^k + 1) \rceil \leq \lceil \log(n+1) \rceil \leq \lceil \log(2^{k+1}) \rceil = k+1 \Rightarrow \\ &\lceil \log(n+1) \rceil = k+1 ; \end{aligned}$$

$$\textcircled{2} \quad k = \lfloor \log(2^k) \rfloor \leq \lfloor \log n \rfloor \leq \lfloor \log(2^{k+1} - 1) \rfloor = k \Rightarrow \lfloor \log n \rfloor = k ;$$

综上: 对于任意整数 $n \geq 1$, $\lceil \log(n+1) \rceil = k+1 = \lfloor \log n \rfloor + 1$, 得证;

2.5

2.5: (二叉树的性质)

对于一棵非空的二叉树 T ，记其中叶节点的个数为 n_0 ，有1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

如果 T 为一棵任意二叉树，请证明 $n_0 = n_2 + 1$ 。

- 证明如下：若 T 是一棵任意二叉树，则由二叉树的结构易知：

若 T 的边数为 e ，则： $e = n - 1 = n_0 + n_1 + n_2 - 1$ 又根据度数与边的关系：

$$e = n_0 \times 0 + n_1 \times 1 + n_2 \times 2$$

联立可得： $n_0 = n_2 + 1$ ，得证；

2.7

2.7: (函数渐近增长率的基本性质)

按照符号定义和关系性质定义证明即可，注意*iff*是“当且仅当”的意思，意味着连接的两个命题是充分必要条件，需要分别从充分性和必要性双向进行说明；

2.8

2.8：(函数渐近增长率排序)

1) 排序如下：

$$\log n < n < n \log n < n^2 \leq n^2 + \log n < n^3 < n - n^3 + 7n^5 < 2^n$$

2) 排序如下：

$$\begin{aligned} \log \log n &< \log n = \ln n < (\log n)^2 < \sqrt{n} < n < n \log n < n^{1+\varepsilon} \\ &< n^2 \leq n^2 + \log n < n^3 < n - n^3 + 7n^5 < 2^{n-1} \leq 2^n < e^n < n! \end{aligned}$$

2.16

* 2.16: (根据递推式求解代价函数的渐近增长率) - part1

1) 根据Master定理: $f(n) = 1 = O(n^{\log_3 2 - \varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_3 2})$

2) 无法用Master定理, 直接展开求和:

$$T(n) = c[\log n + \log \frac{n}{2} + \dots + \log \frac{n}{2^k}], \text{ 其中 } k = \log n$$

$$\begin{aligned} &= c[(k+1) \cdot \log n - \log \prod_{i=0}^k 2^i] = c[k(k+1) - \log 2 \cdot \frac{k(k+1)}{2}] \\ &= \Theta(\log^2 n) \end{aligned}$$

3) 根据Master定理：

$f(n) = cn = \Omega(n^\varepsilon)$, 且任取 $k \in [\frac{1}{2}, 1)$, 有 :

$$\begin{aligned} f\left(\frac{n}{2}\right) &= \frac{cn}{2} \leq k \cdot f(n) = k \cdot cn \\ \Rightarrow T(n) &= \Theta(n) \end{aligned}$$

4) 根据Master定理: $f(n) = cn = \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$

5) 无法用Master定理, 直接展开求和:

$$T(n) = cn[\log n + \log \frac{n}{2} + \dots + \log \frac{n}{2^k}], \text{ 其中 } k = \log n$$

由2)知 : $T(n) = \Theta(n \log^2 n)$

6) 无法用Master定理, 直接展开求和:

$$T(n) = n \cdot \sum_{i=0}^{\log_3 n} \log^3 \frac{n}{3^i} = n \cdot \sum_{i=0}^{\log_3 n} (\log n - \log 3 \cdot i)^3 = \Theta(n \log^4 n)$$

7) 根据Master定理:

$$f(n) = cn^2 = \Omega(n^{1+\varepsilon}), \text{且任取 } k \in [\frac{1}{2}, 1), \text{有:}$$

$$\begin{aligned} 2f\left(\frac{n}{2}\right) &= \frac{cn^2}{2} \leq k \cdot f(n) = k \cdot cn^2 \\ \Rightarrow T(n) &= \Theta(n^2) \end{aligned}$$

8) 根据Master定理:

$$f(n) = n^{\frac{3}{2}} \log n = \Omega(n^{\log_{25}49+\varepsilon}), \text{且任取 } k \in [\frac{49}{125}, 1), \text{有:}$$

$$\begin{aligned} 49f\left(\frac{n}{25}\right) &= \frac{49}{125}n^{\frac{3}{2}} \log \frac{n}{25} \leq kf(n) = kn^{\frac{3}{2}} \log n \\ \Rightarrow T(n) &= \Theta(n^{\frac{3}{2}} \log n) \end{aligned}$$

9) 直接展开求和: $T(n) = T(n - 1) + 2 = \dots = T(1) + 2 \times (n - 1) = 2n - 1 \Rightarrow T(n) = \Theta(n)$

10) 直接展开求和: $T(n) = T(n - 1) + n^c = \dots = T(1) + n^c \times (n - 1) \Rightarrow T(n) = \Theta(n^{c+1})$

11) 直接展开求和: $T(n) = T(n - 1) + c^n = \dots = T(1) + \sum_{i=2}^n c^i \Rightarrow T(n) = \Theta(c^n)$

12) 由于 $f(n) = 2n^3 - 3n^2 + 3n - 1 = \Theta(n^3)$

对 n 的奇偶性讨论如下：

① 若 $n = 2k$, 则: $T(2k) = T(2k - 2) + \Theta(k^3) = T(2(k - 1)) + \Theta(k^3)$, 求和易得:

$$T(2k) = \Theta(k^4)$$

② 若 $n = 2k - 1$, 则: $T(2k - 1) = T(2k - 3) + \Theta(k^3) = T(2(k - 1) - 1) + \Theta(k^3)$, 求和易得:

$$T(2k - 1) = \Theta(k^4)$$

综合①②: $T(n) = \Theta(n^4)$

13) 由替换法: 猜测: $T(n) = \Theta(n)$, 归纳证明如下:

先证: $T(n) = O(n)$, 即证存在常数 $c > 0$ 和 $n_0 > 0$, 使得 $T(n) \leq cn$, 对所有 $n \geq n_0$ 恒成立: i) 对于 $n = 1$, $T(1) = 1$, 易找到 c ; ii) 假设对于某常数 $c > 0$, 当 $n < k$ 时命题均成立, 则当 $n=k$ 时:

$T(k) = T(k/2) + T(k/4) + T(k/8) + k \leq (c/2 + c/4 + c/8 + 1) \times k = (7c/8 + 1) \times k \leq c \times k$ (当 $c \geq 8$ 时), 因此对于任意常数 $c \geq 8$, 命题均成立 $\Rightarrow T(n) = O(n)$ ($c \geq 8$);

再证: $T(n) = \Omega(n)$, 同上理: 对于任意正常数 $c \leq 8$, 命题均成立 $\Rightarrow T(n) = \Omega(n)$ ($c \leq 8$);

综上: $T(n) = O(n) \wedge T(n) = \Omega(n) \iff T(n) = \Theta(n)$, 故取 $c = 8$, 即可得: $T(n) = \Theta(n)$, 猜测得证;

2.18

2.18：(根据递归式计算时间复杂度)

无法用*Master*定理，直接递归树逐层求和：

$$T(n) = \sqrt{n}T(\sqrt{n}) + n = n^{\frac{3}{4}}T(n^{\frac{1}{4}}) + 2n = \dots = n^{1-\frac{1}{2^k}}T(n^{\frac{1}{2^k}}) + kn,$$

$$\text{令 } n^{\frac{1}{2^k}} = 2, \text{ 得到 : } k = \log \log n \Rightarrow T(n) = \Theta(n \log \log n)$$

注：一种错误做法是：利用替换法，猜测 $T(n) = O(n)$ ，

由归纳假设： $T(n) = \sqrt{n}T(\sqrt{n}) + n = \sqrt{n} \cdot O(\sqrt{n}) + n = O(n)$ ，猜测正确；

正确过程：猜测 $T(n) = O(n)$ ，即当 $n \rightarrow +\infty$, $T(n) \leq cn$,

由归纳假设： $T(n) = \sqrt{n}T(\sqrt{n}) + n \leq \sqrt{n} \cdot (c\sqrt{n}) + n = (c+1)n$ ，猜测错误；

2.19

2.19: (Master定理不适用的条件)

可以选: $a = b = 2, f(n) = n \log n$, 易验证此种情况不适用于Master定理;

注: 有同学选: $a = b = 1, f(n) = n \log n$, 此时“递归式”为: $T(n) = T(n) + n \log n$, 易知矛盾;

2.22

2.22：(ALG1和ALG2算法分析)

- 1) 两个算法都是在求数组 A 的最小值；
- 2) 根据递归式求解：
 - 对于ALG1: $T(n) = T(n - 1) + O(1) \Rightarrow T(n) = \Theta(n)$;
 - 对于ALG2: $T(n) = 2T\left(\frac{n}{2}\right) + O(1) \Rightarrow T(n) = \Theta(n)$;

2.24

* 2.24: (多重循环的最坏时间复杂度) - part1

1) MYSTERY:

$$\begin{aligned} time &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=1}^j 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^n j = \sum_{i=1}^{n-1} \frac{(n+i+1)(n-i)}{2} \\ &= \frac{(n-1)n(n+1)}{3} \Rightarrow time = O(n^3) \end{aligned}$$

* 2.24: (多重循环的最坏时间复杂度) - part2

2) PERSKY:

$$\begin{aligned} time &= \sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^{i+j} 1 = \sum_{i=1}^n \sum_{j=1}^i (i+1) = \sum_{i=1}^n i(i+1) \\ &= \sum_{i=1}^n (i^2 + i) = \frac{n(n+1)(n+2)}{3} \Rightarrow time = O(n^3) \end{aligned}$$

* 2.24: (多重循环的最坏时间复杂度) - part3

3) PRESTIFEROUS:

$$\begin{aligned}
 time &= \sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^{i+j} \sum_{l=1}^{i+j-k} 1 = \sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^{i+j} (i + j - k) \\
 &\stackrel{k'=i+j-k}{=} \sum_{i=1}^n \sum_{j=1}^i \sum_{k'=0}^i k' = \sum_{i=1}^n \sum_{j=1}^i \frac{i(i+1)}{2} = \sum_{i=1}^n \frac{i^2(i+1)}{2} \\
 &= \sum_{i=1}^n \frac{(i^3 + i)}{2} = \frac{n(n+1)(n+2)(3n+1)}{24} \Rightarrow time = O(n^4)
 \end{aligned}$$

* 2.24: (多重循环的最坏时间复杂度) - part4

4) CONUNDRUM: (注意各个循环变量真正的取值范围)

$$\begin{aligned}
 time &= \sum_{i=1}^n \sum_{j=i+1}^n \sum_{k=i+j-1}^n 1 = \sum_{i=1}^n \sum_{j=i+1}^n \max(n+2-i-j, 0) \\
 &= \sum_{i=1}^n \sum_{j=i+1}^{n+1-i} (n+2-i) - j \xrightarrow{j'=(n+2-i)-j} \sum_{i=1}^n \sum_{j'=1}^{n+1-2i} j' \\
 &\xrightarrow{n+1-2i \geq 1 \Rightarrow i \leq n/2} \sum_{i=1}^{\frac{n}{2}} \frac{(n+2-2i)(n+1-2i)}{2} \\
 &= \frac{n(n+2)(2n-1)}{24} \Rightarrow time = O(n^3)
 \end{aligned}$$

3.2

3.2: (冒泡排序) - part1

- 1) 利用冒泡排序的性质: 每次冒泡都会确定一个值的最终位置, 来构造循环不变式;
- 2) 最坏情况和平均情况的代价均为总循环次数 $\frac{n(n - 1)}{2} = \Theta(n^2)$;
- 3) 算法改进的目的实际上是尽量减少内循环的次数 $j := 1 \rightarrow \min\{i - 1, k - 1\}$
 - 最坏情况易知为全逆序时, 改进无效, 仍然需要 $\frac{n(n - 1)}{2} = \Theta(n^2)$ 次比较;
 - 平均情况的分析较为复杂, 下面给出一种不够严谨的证明:

3.2: (冒泡排序) - part2

3) 续:

- ① 假设第 i 轮内循环开始前, j 的取值范围为 $[1, k_i - 1]$, 说明此时 $A[k_i + 1, \dots, n]$ 已经有序并且不会再参与比较, 因此问题归为排序 $A[1, \dots, k_i]$ 的子问题;
- ② 第1轮内循环无法改进, 需要 $n - 1$ 次比较, 且循环结束后得到 k_1 , 由①知, 此时问题归为排序 $A[1, \dots, k_1]$ 的子问题;

3.2: (冒泡排序) - part3

3) 续:

③ 为了方便讨论，我们假设在平均情况下，第1轮内循环结束后不会改变排序 $A[1, \dots, k_1]$ 的子问题的平均性能；

④ 在③的前提下，可得代价递归式：

$$T(n) = \sum_{k=1}^{n-1} Pr(n-k)T(n-k) + f(n)$$

其中： $Pr(n-k) = \frac{n-k}{\prod_{i=0}^{k-1} (n-i)}$, $f(n) = n - 1$

3.2: (冒泡排序) - part4

3) 续:

⑤ 猜测 $T(n) = \Theta(n^2)$, 即当 $n \rightarrow +\infty$, $T(n) = cn^2$, 利用替换法证明如下:

$$\begin{aligned} T(n) &= \sum_{k=1}^{n-1} Pr(n-k)T(n-k) + f(n) \\ &= c \cdot \sum_{k=1}^{n-1} Pr(n-k)(n-k)^2 + f(n) = c \cdot \sum_{k=1}^{n-1} \frac{(n-k)^3}{\prod_{i=0}^{k-1} (n-i)} + f(n) \\ &= c \left[\frac{(n-1)^3}{n} + \frac{(n-2)^3}{n(n-1)} + O(1) \right] + f(n) = c[n^2 - 2n + O(1)] + (n-1) \\ &= cn^2 + (1-2c)n + O(1) = cn^2 \end{aligned}$$

取 $c = \frac{1}{2}$, 上式结论成立 $\Rightarrow T(n) = \Theta(n^2)$, 猜测得证;

3.5

* 3.5: (PREVIOUS-LARGER算法分析) - part1

改进: 只需将第5行的语句 $j := j - 1$ 改成 $j := P[j]$ 即可, 因为:

$\forall x \in A[P[j] + 1, \dots, j - 1], x \leq A[j] \leq A[i]$, 因此可以直接跳去比较 $A[P[j]]$ 和 $A[i]$;

正确性证明: 易证原算法的正确性, 而上述改进只是减少了不必要的内循环;

复杂性证明:

* 3.5: (PREVIOUS-LARGER算法分析) - part2

复杂性证明（续）：

- ① 该算法的关键操作是内循环中 $A[j]$ 和 $A[i]$ 的比较，而赋值语句 $j := P[j]$ 的执行次数每次内循环中总是比其少1次，而外循环共 n 次，因此赋值操作相对比较操作共少了 n 次，因此可以更换关键操作为赋值操作，或者说对 P 数组的引用操作，证明其复杂度为 $\Theta(n)$ ；

* 3.5: (PREVIOUS-LARGER算法分析) - part3

复杂性证明 (续) :

② 首先证明引理: 若某次比较 $A[j]$ 和 $A[i]$, 有 $A[j] \leq A[i]$, 则 $\forall x \in A[j+1, \dots, i-1], x < A[j]$

1. 初始: $j = i - 1$, 则 $A[j+1, \dots, i-1] = \emptyset$, 得证;
2. 假设 $j = k$ 时引理成立, 则执行 $j' := P[j]$, 若仍有 $A[j'] \leq A[i]$, 则根据 P 数组定义:

$$\forall x \in A[j'+1, \dots, j-1], x \leq A[j] < A[j']$$

3. 又由归纳假设: $\forall x \in A[j+1, \dots, i-1], x < A[j]$, 结合上式, 有:

$$\forall x \in A[j'+1, \dots, j-1, j, j+1, \dots, i-1], x < A[j']$$

* 3.5: (PREVIOUS-LARGER算法分析) - part4

复杂性证明 (续) :

③ 根据引理, 假设某次比较 $A[j]$ 和 $A[i]$, 有 $A[j] \leq A[i]$, 则 $\forall k \in [j + 1, i - 1]$, 第 k 次外循环

中都不可能引用 $P[j]$, 否则: $A[j] \leq A[k] < A[j]$, 矛盾;

④ 同理, $\forall k \in [i + 1, n]$, 第 k 次外循环中也不可能引用 $P[j]$, 否则:

$A[j] \leq A[k] \Rightarrow \forall x \in A[j + 1, \dots, k - 1], x < A[j]$, 而 $i \in [j + 1, k - 1]$, 故 $A[i] < A[j]$, 矛盾;

⑤ 显然 $\forall k \in [1, j - 1]$, 第 k 次外循环不可能引用 $P[j]$, 故综上所述, 对于 $P[j]$ 的引用只可能出现

在第 i 次外循环中, 意味着 P 数组中每个元素至多被引用一次, 因此该算法复杂度为 $\Theta(n)$;

3.6

* 3.6: (数组调换问题)

给定数组 $A[1, \dots, n]$ 和一个位置下标 k ($1 \leq k \leq n - 1$)，现在需要将数组的左右两部分（以下标 k 为分界线）调换位置

- 1) 题目：请设计一个时间复杂度为 $O(n^2)$ 、空间复杂度为 $O(1)$ 的算法解决这一问题
 - 解析：满足上述要求的算法，易知可以利用冒泡排序的框架，通过相邻位置的若干次交换，使得 $A[k + 1, \dots, n]$ “冒”到 $A[1, \dots, k]$ 前即可；
- 2) 题目：请设计一个时间复杂度为 $O(n)$ 、空间复杂度为 $O(n)$ 的算法解决这一问题
 - 解析：满足上述要求的算法，易知可以构造一个辅助数组，先存储 $A[k + 1, \dots, n]$ ，再接着存储 $A[1, \dots, k]$ ，最后再顺序放回原数组即可；

3) 题目：请设计一个时间复杂度为 $O(n)$ 、空间复杂度为 $O(1)$ 的算法解决这一问题

- 解析：满足上述要求的算法，需要调用一个子函数 $reverse(A, i, j)$ ，该函数的功能是：将子数组 $A[i, .., j]$ 翻转；而为了实现数组的调换，只需三次调用该函数，即： $reverse(A, 1, k)$ ， $reverse(A, k + 1, n)$ ， $reverse(A, 1, n)$ ，易验证三次翻转后即可实现数组按下标 k 的调换；至于复杂度的计算，易知子函数 $reverse(A, i, j)$ 的时间复杂度为 $O(n)$ 、空间复杂度为 $O(1)$ ，则可推知该算法的时间复杂度亦为 $O(n)$ 、空间复杂度亦为 $O(1)$ ；

3.8

3.8: (微博名人问题)

题目：给定 n 个人，我们称一个人为“微博名人”，即他被其他所有人微博关注，但是自己不关注任何人。为了从给定的 n 个人中找到名人，唯一可以进行的操作是：针对两个人A和B，询问“A是否关注了B”，答案只能是YES或者NO
1) 在一群人中，可能有多少个名人？
2) 请设计一个算法找到名人，并试图改进基于遍历的朴素算法

- 解析：根据微博名人的定义进行分析和算法设计（基于遍历，但是可改进以减少遍历次数）
- 解答如下：
 - 1) 只可能至多1个名人，没有名人很正常，而只要存在一个名人A之后，A被其他所有人都关注，而如果存在第二个名人B，B也关注了A，这与“名人不关注任何人”矛盾，故只可能至多1个名人；

2) 算法伪码如下：($A[1,..,n]$ 为 n 个人的关注列表数组)

```
1 Algorithm: MICROBLOG-CELEBRITY(A[1,2,...,n])
2 flag := true;           /*flag 判断被询问者是否为名人*/
3 possible[] := {true};   /*该数组判断每个人是否有可能是名人*/
4 for i:= 1 to n do
5   if possible[i] = true then /*只对可能是名人的进行询问*/
6     for j:= 1 to n do
7       if j != i then        /*遍历其可能的关注的其他人*/
8         if A[i] has no j then
9           possible[j] := false; /*没关注的一定不是名人*/
10        else
11          flag := false; /*关注了别人被询问者一定不是名人*/
12 if flag = true then /*没关注其他人的要么是名人，要么可判断其中没有名人*/
13   for k:=1 to n do
14     if k!=i and A[k] has no i then
15       return 0;
16   return i;
17 return 0;
```

3.9

3.9：(最大和连续子序列)

- 1) 遍历所有可能的子串，并对其求和；
- 2) 遍历所有子串，对共起点的子串累计求和；
- 3) 分成左右两半递归计算，再线性时间算跨越左右两半的部分，三部分返回最大值；
- 4) 一趟遍历，记录非负子串最值；
- 5) 定义以第 k 个元素结尾的最大和连续子序列的和为 $sum[k]$ ，

有动态规划状态转移方程： $sum[i + 1] = \max\{a[i + 1], sum[i] + a[i + 1]\}$ ；

3) (时间复杂度为 $O(n \log n)$)

```
1 Algorithm: MAX-SUBARRAY3(A[1,2,...,k])
2 center := (k+1)/2; /*将串分半*/
3 if center = 0 then /*串仅一个元素，直接返回唯一元素值*/
4     return A[1];
5 leftmax := MAX-SUBARRAY3(A[1,2,...,center-1]) /*左子串最值*/
6 rightmax := MAX-SUBARRAY3(A[center+1,...,k]) /*右子串最值*/
7 midLmax := 0; midRmax := 0; temp := 0;
8 for i:= center downto 1 do /*中间子串左边最值*/
9     temp := temp+A[i];
10    if temp > midLmax then
11        midLmax := temp;
12    else if temp < midLmax then // 开始减小，说明已得最值
13        break;
14 temp := 0;
15 for j:= center+1 to k do /*中间子串右边最值*/
16    temp := temp+A[j];
17    if temp > midRmax then
18        midRmax := temp;
19    else if temp < midRmax then // 开始减小，说明已得最值
20        break;
21 midmax := midLmax+midRmax; /*中间子串最值*/
22 return MAX(leftmax,midmax,rightmax); /*该串最值为三者之最*/
```

4) (时间复杂度为 $O(n)$)

```
1 Algorithm: MAX-SUBARRAY4(A[1,2,...,n])
2 max := 0; temp := 0;
3 for i:= 1 to n do
4     temp := temp + A[i];
5     if temp > max then
6         max := temp;
7     if temp < 0 then /*子串为负即抛弃，因为任何极大子串不可能包含它*/
8         temp := 0;
9 return max;
```

5) 思路类似与4),

P2

4.1、4.4

4.1: (二叉树的叶结点个数)

二叉树越趋近于完美，叶节点个数越多， $L = 2^h$ 在二叉树为完美二叉树时成立；

4.4: (少量元素排序)

- 1) 采用锦标赛排序（逆向归并排序）；
- 2) 同样采用锦标赛排序，最优算法最坏需要7次比较；

4.8、4.9

4.8: (k-sorted问题)

将快速排序进行到 $\log k$ 层即可，但是需要注意，为了保证每段等长为 $\frac{n}{k}$ ，每次partition都得选择中位数才行，而选择中位数有线性算法，因此代价同普通partition；

4.9: (螺钉匹配问题)

利用快排思想，先用某个螺母将螺钉划分成三堆，左边螺钉比螺母小，中间一个刚好匹配，右边比螺母大，然后用和该螺母正好匹配的螺钉再将螺母划分成相应三堆，对左右两边分别进行递归即可；

4.11

* 4.11: (少量逆序对数组排序) - part1

- 1) 利用反证法, 假设 $j - i > 2$, 则 $A[i], A[j]$ 之间任意1个元素 $A[p](i < p < j)$ 对应至少1个逆序对, 故 $A[i, \dots, j]$ 总共至少对应 $j - i$ 个逆序对;
- 2) 为了保证最坏情况比较次数不超过 n , 设计算法时要将可能的情况分析完整

* 4.11: (少量逆序对数组排序) - part2

2) 续:

① 先说明以下引理:

Lemma1: 若存在逆序对 (i, j) , 则 $j = i + 1$ 或 $i + 2$;

Lemma2: 如果存在逆序对 $(i, i + 2)$, 则在 $(i, i + 1)$ 和 $(i + 1, i + 2)$ 中有且仅有一个逆序对;

Lemma3: 若存在逆序对 $(i, i + 1)$, 则 $(i + 1, i + 2)$ 不可能也是逆序对, 否则三元组 $(i, i + 1, i + 2)$ 将会产生3个逆序对;

* 4.11: (少量逆序对数组排序) - part3

2) 续:

② 根据①中引理, 数组中逆序对的情况共5种, 以数组 $A = [1, 2, 3, 4, 5, 6]$ 的排列为例:

case1: 没有逆序对 $\Rightarrow A = [1, 2, 3, 4, 5, 6]$;

case2: 有1个逆序对 $\Rightarrow A = [1, 3, 2, 4, 5, 6]$;

case3: 有2个不连续的逆序对 $\Rightarrow A = [1, 3, 2, 4, 6, 5]$;

case4: 有2个连续的逆序对, 且中位数在最左边 $\Rightarrow A = [1, 3, 4, 2, 5, 6]$;

case5: 有2个连续的逆序对, 且中位数在最右边 $\Rightarrow A = [1, 4, 2, 3, 5, 6]$;

* 4.11: (少量逆序对数组排序) - part4

2) 续: (注: 插入排序在数组基本有序时优势明显, 本题可按其模板) 设计算法如下:

```
1 Algorithm: Inv2Sort(A[1..n])
2
3 cnt := 0, j := 0; // cnt变量可以省去, 不影响算法正确性, 但是逻辑会更晦涩
4 // 最坏n次比较
5 for i:=1 to n-1 do
6
7     if A[i] > A[i+1] then // 此处共n-1次比较
8         swap(A[i],A[i+1]);
9
10    cnt := cnt + 1;
11    if cnt = 1 then // 发现第一个逆序对, 对应case2, 或case3,4,5
12        j := i-1;
13    else if cnt = 2 then // 发现第二个, 对应case3或case5
14        break;
15
16    if cnt = 1 and j>0 and A[j] > A[j+1] then // +1次比较, 对应case4, j>0防止越界
17        swap(A[j],A[j+1]);
18 // if cnt = 0 then 对应case1
19 return A;
```

4.14

4.14: (易位词)

题目：请设计一个算法找出一个很大的英文文件中的所有异位词

- 解析：先将每个单词内部排序，则同组易位词全部转换为字典序最小的单词形式，接着再对排序后的单词进行外部排序，则挨在一起的相同的单词即为同组易位词；

7.1

* 7.1: (逆序对计数问题的推广) - part1

由于本题没有限制时间复杂度，因此可以直接 $O(n^2)$ 朴素遍历，也可以利用教材7.2.1和7.2.2中基于归排和堆排的算法实现，时间复杂度分别为 $O(n \log n)$ 和 $O(n \log^2 n)$ ；以下基于归排，给出一种实现：

```
1 Algorithm: INVERSION_MERGE(A, l, mid, r)
2 // init
3 n1 := mid-l+1; n2 := r-mid;
4 for i:= 1 to n1 do
5   L[i] := A[l+i-1];
6 for j:= 1 to n2 do
7   R[j] := A[mid+j];
8 L[n1+1] = +∞; R[n2+1] = +∞;
9 i:=1; j:= 1; inv := 0; // inv for the number of inversions
10 // merge
11 for k:= l to r do
12   if L[i] > R[j] then // for merge in normal order
13     A[k] := R[j];
14     if L[i] > C*R[j] and L[i] != +∞ then // C >= 1
15       inv := inv + (mid - l - i + 1); // all the L[i+p] > L[i] > C*R[j]
16     j := j + 1;
17   else
18     A[k] := L[i];
19     i := i + 1;
20 return inv;
```

* 7.1: (逆序对计数问题的推广) - part3

另注：这道题除了使用排序框架外，还可以利用树状数组(*Binary Indexed Tree*)的数据结构，同样可以达到 $O(n \log n)$ 的时间复杂度，简要说明如下：

- ① 树状数组可以动态维护前缀和，达到 $O(\log n)$ 时间计算 $\text{getSum}(i)$ 和执行 $\text{update}(i, val)$ ；
- ② 逆序对个数的统计，实际上就是计算 $\sum_{i=1}^n \sum_{j=1}^{i-1} I(A[j] > A[i])$ ，某种变换下相当于计算 $\sum_{i=1}^n \text{getSum}(i)$ ；
- ③ 建立树状数组的时间代价为 $O(n \log n)$ 、空间代价为 $O(n)$ ；共 n 次调用 $\text{getSum}(i)$ ，代价为 $O(n \log n)$ ，“变换”的代价可以做到 $O(n \log n)$ ，因此最终算法的时间复杂度为 $O(n \log n)$ ，空间复杂度为 $O(n)$ ；

7.4

* 7.4: (k 个数组的合并)

假设有 k 个数组，每个数组有 n 个有序元素，现在需要将这些数组合并成一个有序数组

1) 题目：考虑如下方案：。。。请分析该算法的时间复杂度

- 解析：合并两个有序数组 $A[1, \dots, n]$ 和 $B[1, \dots, m]$ 的代价为 $O(m + n)$ ，因此将每次合并的代价相加即可；
- 解答： $worstCost = 2n + 3n + \dots + kn = O(k^2n)$ ；

2) 题目：请给出一个分治算法，在 $O(nk \log k)$ 的时间内完成 k 个数组的合并

- 解析：思路已经提示的很明显了，将第一问“串行”的 $O(k^2n)$ 的算法改进为“并行”的即可，具体来说：将 k 个数组两两分组，分别合并，形成 $\frac{k}{2}$ 个拥有 $2n$ 个元素的有序数组，再重复这个过程，将 $k/2$ 个数组继续两两分组，分别合并。。。直到最后形成一个完整的拥有 kn 个元素的有序数组；

由于第 i 次分组合并的代价为： $\frac{k}{2^i} \times 2^i n = nk$ ，总共要进行 $\log k$ 次，因此总时间为 $O(nk \log k)$

7.5

7.5：(树的高度和直径)

1) 计算树 T 的高度 h_T , 若已知左右子树的高度 h_l 和 h_r , 则有:

$$h_T = \max\{h_l, h_r\} + 1 \cdots (*)$$

根据(*)式可得递归式: $T(n) = T(n_l) + T(n_r) + O(1), n_l + n_r = n \Rightarrow T(n) = O(n)$

2) 计算树 T 的直径 d_T , 若已知左右子树的直径 d_l 和 d_r , 以及左右子树的高度 h_l 和 h_r , 则有:

$$d_T = \max\{d_l, d_r, h_l + h_r + 2\} \cdots (*)$$

可以设置一个全局变量 d_{max} , 记录递归中出现的最大直径, 这样求直径算法可依附在求高算法之中,

时间复杂度仍为 $O(n)$;

- 另注: 也可以两次深度优先遍历, 先找到最大深度结点, 从最大深度结点深度遍历, 再找到它所对应的最大深度点, 其距离即为直径;

7.8

1) 题目：请设计一个算法找出所有的maxima

- 解析：

- ① 若可以进行排序的话，可以先仅对x进行排序，再依照x的排序结果从大到小去比较y，**当且仅当** a_i 的y坐标大于任何 a_j ($j > i$)的y坐标时，其为一个maxima，因此可以维护一个 y_{max} 变量，找到一个maxima就更新一次；
- ② 若不能进行排序，则找到x的中值，然后划分整个数组，分别递归找到左半部分和右半部分的maxima，右半部分的maxima一定是整体的maxima，而左半部分的maxima还要进行筛选，即只有满足①中**黑体**所述条件才是整体的maxima；

- 算法：略

2) 题目：请将上述思路整理成一个算法，并证明其正确性，或者找出上述思路的错误

- 解析：该算法错误，因为需要注意到坐标的分布可能十分不均匀，或者规律性很强（例如当所有的点都在 $x + y = 1$ 这条直线上时，第三象限没有值，算法不能终止；再例如当有几个值的坐标非常极端时，可能会使得maxima也落入第三象限，算法不正确）
 - 解答：略
-

3) 题目：如果你能找出上述思路的错误，能否进一步证明该问题不可能有代价为 $O(n \log n)$ 的算法

- 解析：因为基于比较的排序算法复杂度为 $\Theta(n \log n)$ ，因此同理基于坐标比较的寻找maxima问题的下界不可能到 $O(n \log n)$ ；
- 证明：略

7.12

1) (记二维比特数组为 $bitStr$, 缺失比特串为 $mStr$, 下同);

1. 算法思路:

Lemma1: 当 $k \geq 2$ 时, $bitStr$ 第 j 列的布尔和等于 $mstr$ 的第 j 位比特值, 证明如下:

- ① 若没有缺失比特串且 $k \geq 2$, 则 $bitStr$ 任何一列上的布尔和等于偶数个1的布尔和, 为0;
- ② 若 $mstr[j] = 0$, 则 $bitStr$ 第 j 列的布尔和仍等于偶数个1的布尔和(仅少了一个0), 为0;
- ③ 若 $mstr[j] = 1$, 则 $bitStr$ 第 j 列的布尔和等于奇数个1的布尔和(仅少了一个1), 为1;

综合①~③: Lemma1得证 \Rightarrow 根据Lemma1, 可设计朴素查找算法如下:

- I. 若 $k = 1$, 则 $bitStr$ 仅含0, 1两个单比特串, 缺失一个, 将剩余那个取反即可得 $mstr$;
- II. 若 $k \geq 2$, 则对每一个 $j \in [1, k]$, 有: $mstr[j] = \sum_{i=1}^{n-1} bitStr[i][j]$;

2. 算法伪码:

```
1 Algorithm FindMissingString1(bitStr[1..n-1][1..k]) // => O(nk)
2 // base case: k = 1, n-1 = 1, just return the flipped bitStr[1][1]
3 if k == 1 then
4     return bitStr[1][1] ⊕ 1; // 1 => 0, 0 => 1
5 // bool sum for each bit in column j
6 mstr := [00..0]; // all-zero array of size k
7 for j:=1 to k do
8     sum := 0;
9     for i:=1 to n-1 do
10        sum := sum ⊕ bitStr[i][j];
11     mstr[j] := sum;
12 return mstr;
```

3. 算法分析:

时间复杂度: 易知对 $bitStr$ 每一位仅访问1次 $\Rightarrow O(nk)$;

空间复杂度: 易知只使用了常数额外空间 $\Rightarrow O(1)$;

2)

1. 算法思路:

Lemma2: 当 $k \geq 3$ 时, 若 $mstr[j] = b$, 则所有第 j 位为 b 的比特串, 其第 $j + 1$ 列的布尔和等于 $mstr$ 的第 $j + 1$ 位比特值, 证明如下:

- ① 当 $k \geq 3$ 时, 第 j 位为 b 的比特串一定满足 0 和 1 的数量相等且为偶数, 因此其布尔和为 0;
- ② 故在求和第 $j + 1$ 列时, 只需要考虑那些第 j 位为 0 的比特串, 共 $\lfloor \frac{n-1}{2} \rfloor$ 个;

根据 Lemma2, 可设计分治查找算法如下:

b

- I. 从 $j = 1$ 开始, 当确定 $mstr[j] = b$ 后, 删除 $bitStr$ 第 j 位不是 b 的一半比特串;
- II. 在剩下的一半比特串上计算 $mstr[j+1]$, 重复上述步骤直到比特串只剩下一个;
- III. 若比特串只剩下一个, 显然此时 $j = k$, 直接将其第 k 位取反即得到 $mstr[k]$;

3. 算法分析:

1. 时间代价递归式: $T(n) = T(\frac{n}{2}) + O(n)$, 由 Master 定理得: $T(n) = O(n)$;
2. 空间代价递归式: $T(n, k) = T(\frac{n}{2}, k) + O(nk)$, 由 Master 定理得: $T(n, k) = O(nk)$;

2. 算法伪码:

```
1 Algorithm FindMissingString2(bitStr[1..n][1..k], j, mstr) // => O(n)
2 // base case1: nj = 1, just return the flipped bitStr[1][j]
3 if n == 1 then
4     mstr[j] := bitStr[1][j] ⊕ 1; // 1 => 0, 0 => 1
5     return;
6 // bool sum for column j
7 sum := 0;
8 for i:=1 to n do
9     sum := sum ⊕ bitStr[i][j];
10 mstr[j] := sum;
11 // get subBitStr containing those who hold the same jth bit with mstr
12 subBitStr := [];
13 for i:=1 to n do
14     if bitStr[i][j] == mstr[j] then
15         subBitStr[i] = bitStr[i];
16 // recurse to subBitStr to find (j+1)th bit
17 FindMissingString2(subBitStr[1..n/2][1..k], j+1, mstr);
```

14.1

14.1: (堆结构的数学特性)

题目：请证明，对于所有正整数 h ， $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil =$

$\lceil \log(h + 1) \rceil$

- 解析：利用堆结构的数学特性即可证明
- 证明：由堆的特性知：若假设 h 为某堆中的某节点在堆中的索引，则可知：
 $\lfloor \frac{h}{2} \rfloor$ 为其父亲节点的索引， $\lceil \log(h + 1) \rceil$ 为 h 节点的深度，自然 $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil$ 为其父亲节点的深度

故命题可表述为：“索引为 h 的节点的深度等于其父亲节点的深度+1”，由堆的树形结构知显然正确；

14.2

* 14.2: (堆中第k大的元素)

注意代价只能是 k 的函数，与 n 无关，几种可能的算法思路如下：

- ① 因为堆中第 k 大的元素至多在第 k 层，因此直接对前 k 层（ $2^k - 1$ 个元素）子堆进行堆排序，即可找到第 k 大的元素，代价为 $O(k2^k)$ ；
- ② 同①理，搜索范围为前 k 层（ $2^k - 1$ 个元素）的子堆，但实际上无需全排序，只需要让堆弹出 k 次堆顶（ k 次修复），即可得到第 k 大的元素，代价为 $O(k^2)$ ；

③ 虽然第 k 大的元素只存在于前 k 层，但是出现在第 k 层的概率相当低，我们可以动态利用堆的性质，维护一个优先级队列（其本身也可以用堆实现），每次将堆顶元素入队，然后队首元素出队，将其在堆中对应的左右子节点入队，重复上述操作 k 次，则第 k 次弹出的最大元素即为堆中第 k 大的元素 \Rightarrow 由于优先级队列至多 k 个元素，每次修复代价为 $O(\log k)$ ，因此 k 次总代价为 $O(k \log k)$ ；

14.3

14.3: (d叉堆) - part1

可以定义一个二维坐标 (i, j) 表示第*i*层的第*j*个节点，然后结合其数学特性进行证明，如下所示：

$$\text{令 } P(i) = \lfloor \frac{i-2}{d} + 1 \rfloor, \quad C(i, j) = d(i-1) + j + 1$$

① 设某节点在第*h*层的第*k*位上，则其索引为：

$$i = index(h, k) = (1 + d + d^2 + \cdots + d^{h-2}) + k = \frac{d^{h-1} - 1}{d - 1} + k$$

14.3: (d叉堆) - part2

而易知其父亲节点在第 $h - 1$ 层的第 $\lceil \frac{k}{d} \rceil$ 个，则其父亲的索引为：

$$\begin{aligned} index(h - 1, \lceil \frac{k}{d} \rceil) &= \frac{d^{h-2} - 1}{d - 1} + \lceil \frac{k}{d} \rceil = \frac{d^{h-2} - 1}{d - 1} + \lfloor \frac{k + d - 1}{d} \rfloor \\ &= \lfloor \frac{d^{h-2} - 1}{d - 1} + \frac{k + d - 1}{d} \rfloor = \lfloor \frac{d^{h-1} - d}{d(d - 1)} + \frac{k + d - 1}{d} \rfloor \\ &= \lfloor \frac{d^{h-1} - 1}{d(d - 1)} - \frac{2}{d} + \frac{k}{d} + 1 \rfloor = \lfloor \frac{1}{d} [\frac{d^{h-1} - 1}{(d - 1)} + k - 2] + 1 \rfloor \\ &= \lfloor \frac{i - 2}{d} + 1 \rfloor = P(i) \end{aligned}$$

14.3: (d叉堆) - part3

② 同①设，则节点*i*的第*j*个子节点在第*h*+1层的第(*k*-1)*d*+*j*个，则该子节点的索引为：

$$\begin{aligned} index(h + 1, (k - 1)d + j) &= \frac{d^h - 1}{d - 1} + (k - 1) \times d + j \\ &= d \times \frac{d^{h-1} - 1}{d - 1} + 1 + d \times k - d + j \\ &= d(i - 1) + j + 1 = C(i, j) \end{aligned}$$

14.4

14.4: (所有节点的高度之和) - part1

利用数学归纳法证明，注意不能直接对左右子堆进行归纳假设（否则证明条件不够），
还需利用堆的一个结构特性，即“堆的左子堆和右子堆至少有一个是完美堆”，证明如下：

① 若左子堆和右子堆均完美，即该堆完美：

设该完美堆的高度为 L ，容易推出节点数 $N(L) = 2^{L+1} - 1$ ，所有节点的高度之和

$$\Sigma H(L) = 2^{L+1} - L - 2, \text{ 则 } \Sigma H(L) = 2^{L+1} - L - 2 < 2^{L+1} - 2 = N(L) - 1 = n - 1;$$

14.4: (所有节点的高度之和) - part2

② 若左子堆非完美，右子堆完美，且左子堆的高度为 $L - 1$ ，右子堆的高度为 $L - 2$

$$\Rightarrow \text{右子堆: } N(L - 2) = 2^{L-1} - 1, \Sigma H(L - 2) = 2^{L-1} - L$$

$$\Rightarrow \text{左子堆: } N'(L - 1) = (n - 1) - N(L - 2) = n - 2^{L-1}$$

$$\Sigma H'(L - 1) \leq N'(L - 1) - 1 = n - 2^{L-1} - 1$$

$$\text{则 } \Sigma H(L) = \Sigma H'(L - 1) + \Sigma H(L - 2) + L \leq n - 1;$$

14.4: (所有节点的高度之和) - part3

③ 若右子堆非完美， 左子堆完美， 且左， 右子堆的高度均为 $L - 1$

$$\Rightarrow \text{左子堆: } N(L - 1) = 2^L - 1, \Sigma H(L - 1) = 2^L - L - 1$$

$$\Rightarrow \text{右子堆: } N'(L - 1) = (n - 1) - N(L - 1) = n - 2^L,$$

$$\Sigma H'(L - 1) \leq N'(L - 1) - 1 = n - 2^L - 1$$

$$\text{则 } \Sigma H(L) = \Sigma H'(L - 1) + \Sigma H(L - 1) + L \leq n - 2 < n - 1;$$

综上: n 个节点的堆中所有节点的高度之和最多为 $n - 1$, 同时当 $n = 2^k$ 时, 其高度之和正好是 $n - 1$;

14.5

14.5：(k个链表的合并)

先用各链表中的表头构建一个最小堆($O(k)$)，则此时堆顶元素一定是 n 个元素中的最小值，

接着从堆顶元素所在原链表中取出下一元素替换堆顶并修复($O(\log k)$)，重复这个过程直到所有链空，

接着依次从堆中弹出堆顶直到堆空 \Rightarrow 每个元素都入/出堆各1次，故总代价为 $O(k + n \log k) = O(n \log k)$ ；

另注：此题也可按照习题7.4“ k 个数组的合并”的思路，通过归并排序实现，只是由于链表无法随机访问，

因此向左右两半递归之前，还需先找到链表的中间节点，可以用快慢指针实现（时间为 $O(n)$ ）；

14.6

* 14.6: (动态发现中值)

采用对顶堆的数据结构，具体来说：

① 维护两个堆，左边最大堆 H_l ，右边最小堆 H_r ，始终保持 $H_r.size \leq H_l.size \leq H_r.size + 1$ ，

并且保证 $H_l.top = median$ ；（常数时间发现中位数）

② 插入元素 x ：（对数时间插入）

case1: $H_l.size = H_r.size$ 且 $x > H_l.top$, 先将 x 插入 H_r ，再将 $H_r.top$ 弹出并插入 H_l ；

case2: $H_l.size = H_r.size$ 且 $x \leq H_l.top$, 直接将 x 插入 H_l ；

case3: $H_l.size = H_r.size + 1$ 且 $x > H_l.top$, 直接将 x 插入 H_r ；

case4: $H_l.size = H_r.size + 1$ 且 $x \leq H_l.top$, 先将 x 插入 H_l ，再将 $H_l.top$ 弹出并插入 H_r ；

③ 删除中值 m : (对数时间删除)

case1: $H_l.size = H_r.size$, 先将 m 从 $H_l.top$ 弹出, 再将 $H_r.top$ 弹出并插入 H_l ;

case2: $H_l.size = H_r.size + 1$, 直接将 m 从 $H_l.top$ 弹出;

④ 删除元素 x : (对数时间删除)

由于普通堆并没有删除操作, 因此采用延迟删除技巧, 即记录一个待删除集合 $delSet$,

将待删除元素 x 插入集合, 每次当 $H_l.top$ 或 $delSet$ 更新时, 检查 $delSet.contains(H_l.top)$,

若返回 $True$, 则执行③中算法, 同时 $delSet.erase(H_l.top)$, 直到返回 $False$ 或堆空或集合空;