

算法设计与分析作业二

作者：吴润泽 学号：181860109

Email: 181860109@smail.nju.edu.cn

2020 年 8 月 12 日

目录

PART I	2
problem 6.8	2
problem 6.9	4
problem 6.10	6
problem 6.13	9
problem 6.15	11
 PART II	 14
problem 7.1	14
problem 7.2	15
problem 7.3	16
problem 7.4	17
problem 7.6	18

PART I

problem 6.8

算法分析 假定 n 总是 k 的倍数, 且 n 和 k 都是 2 的幂。

利用快排的思想, 将数组从中间划分为两段 $A[0 \cdots n/2]$, $A[n/2+1 \cdots n]$, 且左段元素小于右段元素。

对于子序列继续递归划分, 得到 $A[0 \cdots n/2^m]$, $A[n/2^m+1 \cdots n/2^{m-1}] \cdots A[n - n/2^m + 1 \cdots n]$, 当 $2^m = k \rightarrow m = \log k$ 时, 划分完成。

因此寻找中位数划分的函数时间复杂度应为 $O(n)$, 划分函数的递归方程为 $W(n) = 2W(n/2) + O(n)$, 划分左右子段 $\log k$ 次, 方能使得总的时间复杂度达到 $O(n \log k)$ 。

具体算法实现请见算法 **k-sorted**: [k-sorted 算法](#)

算法时间复杂度 对于 `findk_pos`, 每次递归代价为 $O(n)$, 每次子问题缩小为原来一半的规模, 且子问题只有一个, 可列出递归方程为 $T(n) = T(n/2) + O(n)$, 由主定理可以得出 $T(n) = O(n)$ 。

对于 `k_sorted`, 每次递归代价为 $O(n)$, 每次子问题缩小为原来一半, 而需要划分左右两序列, 子问题为两个, 可列出递归方程为 $W(n) = 2W(n/2) + O(n)$, 注意结束条件为递归调用了 $\log k$ 层, 每层代价均为 $O(n)$, 因此时间复杂度为 $O(n \log k)$ 满足题目要求。

算法 1 k-sorted 算法

输入: 待划分序列 $A[1 \cdots n]$, 划分段数 k **输出:** 划分后的的序列 A

```

1: function FINDK_POS( $A, k\_pos, begin, end$ ) \\ 返回该段数组第  $k$  小
2: /* 利用快排思想, 选定一个 key, 将大于 key 的元素放在其右边, 小于
   key 放于左边。
3: 判断 key 插入的位置是否为  $k$ , 如果是则函数返回, 如果插入位置大于
    $k$  说明第  $k$  小位于左子序列对左边递归寻找, 否则对右子序列递归寻找。
   */
4:    $split \leftarrow begin, key \leftarrow A[begin]$ 
5:   for  $i \leftarrow begin + 1$  to  $end$  do
6:      $A[i] \leq key ? swap(A[begin], A[i])$ 
7:   end for
8:    $split > k\_pos ? \text{return } findk\_pos(A, k\_pos, begin, split - 1)$ 
9:    $split < k\_pos ? \text{return } findk\_pos(A, k\_pos, split + 1, end)$ 
10:  return  $split$ 
11: end function
12: function K_SORTED( $A, begin, end, k, count = 1$ )
13: /* count 记录当前的段数, 每次调用 findk_pos,  $A$  被分为  $[begin, mid]$ 
   和  $[mid+1, end]$  两段, 段数变为原来两倍, 且左段元素小于右段, 调用
   层数达到  $\log k$  层算法结束, 否则继续划分左右子序列 */
14:    $mid \leftarrow (end - begin)/2 + begin, count \leftarrow count * 2$ 
15:    $findk\_pos(A, mid, begin, end)$ 
16:   if  $count == k$  then
17:     划分  $k$  段, 算法结束
18:     return  $A$ 
19:   end if
20:    $k\_sorted(A, begin, mid, k, count)$ 
21:    $k\_sorted(A, mid + 1, end, k, count)$ 
22: end function

```

problem 6.9

算法分析 同样利用快速排序的思想，令螺钉为 A，螺母为 B：

1. 在 A 数组中拿一个，根据 A 和螺母的大小关系，可以分成三部分，B1：比螺钉小的，B2：比螺钉大的，B3：完全匹配的。
2. 用 B3，同样可以把 A 分为三部分，A1：比螺母小的，A2：比螺母大的，A3：完全匹配的。
3. B1 与 A1 匹配，B2 与 A2 匹配，分别执行上述算法，直至全部匹配。

具体算法实现请见算法 match： [match 算法](#)

算法时间复杂度 对于每次递归代价：

1. 首先寻找分割点，遍历了一次数组，时间复杂度为 $O(n)$ 。
 2. 之后根据分割点遍历 A,B 两数组将其分为两部分，时间复杂度为 $O(n)$ 。
 3. 最后将分割后两序列进行递归操作继续划分。因此每次递归操作总的代价为 $O(n)$ 。
- 因此可推得算法的递推方程为 $T(n) = 2T(n/2) + O(n)$ 。根据主定理可得时间复杂度为 $O(n \log n)$ 满足题目要求。

算法 2 match 算法

输入: 螺钉数组 A , 螺母数组 B **输出:** 螺钉螺母对应下标一一匹配后的数组

```

1: function MATCH( $A, B, l, r$ )
2: /*找到分割点, mark 记录 B 等于 A 首元素的下标,
3: count 记录 B 中小于 A 的个数*/
4:    $count \leftarrow 0, mark \leftarrow 0$ 
5:   for  $i \leftarrow l$  to  $r$  do
6:      $A[l] == B[i] ? mark = i$ 
7:      $A[l] > B[i] ? count++ = 1$ 
8:   end for
9: /*为 B 和 A 的左半部分分配 count 个元素*/
10:   $swap(A[l], A[l + count]), swap(B[mark], B[l + count])$ 
11:   $mark \leftarrow mark + count, i \leftarrow l, j \leftarrow r$ 
12:  while  $i < mark$  and  $j > mark$  do\\将 a 分成两部分
13:    while  $i < mark$  and  $a[i] < b[mark]$  do
14:       $i \leftarrow i + 1$ 
15:    end while
16:    while  $j > mark$  and  $a[j] < b[mark]$  do
17:       $j \leftarrow j - 1$ 
18:    end while
19:     $swap(a[i + +], a[j - -])$ 
20:  end while
21:   $i \leftarrow l, j \leftarrow r$ 
22:  while  $i < mark$  and  $j > mark$  do\\将 b 分成两部分
23:    while  $i < mark$  and  $b[i] < a[mark]$  do
24:       $i \leftarrow i + 1$ 
25:    end while
26:    while  $j > mark$  and  $b[j] < a[mark]$  do
27:       $j \leftarrow j - 1$ 
28:    end while
29:     $swap(b[i + +], b[j - -])$ 
30:  end while
31:   $l < mark ? match(A, B, l, mark - 1)$ 
32:   $r > mark ? match(A, B, mark + 1, r)$ 
33: end function

```

problem 6.10

(1)

算法分析 利用归并排序的思想，在归并排序中合并左右数组 A, B

1. 由归并排序定义可知，A、B 两数组已经有序。
2. 在合并过程中，就需要计算 $a[i]$, $b[j]$ 分别来自左右两部分的逆序对数，同时遍历两个数组，对于遍历的两个数进行比较大，如果 $a[i] < b[j]$ 显然没有逆序。
4. 如果 $a[i] > b[j]$ ，那么 $a[i+1 \cdots n] > b[j]$ 均成立，即逆序对数有 $n-i+1$ 个。因此遍历时如遇到 $a[i] > b[j]$ ，逆序对数加 $n-i+1$ 即可。

具体算法实现请见算法 MergeSort: [MergeSort 算法](#)

算法时间复杂度 与归并排序算法相同，在合并函数中仅添加一条赋值语句，复杂度仍为 $O(n \log n)$ 符合题目要求。

(2)

算法分析 利用归并排序的思想，在归并排序中合并左右数组 A, B。若 $a[i] > b[j] \& a[i] > C \cdot b[j]$ ，则 $a[i+1 \cdots n] > C \cdot b[j]$ 均成立，与 [MergeSort 算法](#) 相似。当 $a[i] > C \cdot b[j]$ 时，遍历 a 数组，找到 $a[tem] > C \cdot b[j]$ ，更新总的逆序对个数即可。

具体算法实现请见算法 GeneralPair: [GeneralPair 算法](#)

算法时间复杂度 在 $a[i] > C \cdot b[j]$ 时，遍历左数组，每次合并的时间复杂度为 $O(n^2)$ 。总的递推方程为 $T(n) = 2T(\frac{n}{2}) + O(n^2)$ ，时间复杂度变为 $O(n^2)$ 。

算法 3 MergeSort 算法

输入: 无序序列 A, l, r **输出:** 总逆序对数和 sum

```
1:  $sum \leftarrow 0$ 
2: function MERGESORT( $A, l, r$ )
3:    $l == r$  ? return
4:    $mid \leftarrow \frac{l+r}{2}$ 
5:   MergeSort( $A, l, mid$ ), MergeSort( $A, mid + 1, r$ )
6:   Merge( $A, l, mid, r$ )
7: end function
8: function MERGE( $A, l, mid, r$ )
9:    $n1 \leftarrow mid - l + 1$ ,  $n2 \leftarrow r - mid$ 
10: Let  $L[1..(n1+1)]$  and  $R[1..(n2+1)]$  be new arrays
11:   for  $i \leftarrow 1$  to  $n1$  do
12:      $L[i] \leftarrow A[l - i + 1]$ 
13:   end for
14:   for  $i \leftarrow 1$  to  $n2$  do
15:      $R[i] \leftarrow A[mid + i]$ 
16:   end for
17:    $L[n1 + 1] \leftarrow \infty$ ,  $R[n2 + 1] \leftarrow \infty$ 
18:    $i \leftarrow 1$ ,  $j \leftarrow 1$ 
19:   for  $k \leftarrow l$  to  $r$  do
20:     if  $L[i] < R[j]$  then
21:        $A[k] \leftarrow L[i++]$ 
22:     else
23:        $A[k] \leftarrow R[j++]$ 
24:        $sum \leftarrow sum + n1 - i + 1$  \\ 仅添加这句, 更新 sum 逆序对和
25:     end if
26:   end for
27: end function
```

算法 4 GeneralPair 算法

输入: 无序序列 A, l, r, C 输出: 总广义逆序对数和 sum

```

1:  $sum \leftarrow 0$ 
2: function MERGESORT( $A, l, r, C$ )
3:    $l == r$  ? return
4:    $mid \leftarrow \frac{l+r}{2}$ 
5:   MergeSort( $A, l, mid, C$ ), MergeSort( $A, mid + 1, r, C$ )
6:   Merge( $A, l, mid, r$ )
7: end function
8: function MERGE( $A, l, mid, r$ )
9:    $n1 \leftarrow mid - l + 1$ ,  $n2 \leftarrow r - mid$ 
10: Let  $L[1..(n1+1)]$  and  $R[1..(n2+1)]$  be new arrays
11:   for  $i \leftarrow 1$  to  $n1$  do
12:      $L[i] \leftarrow A[l - i + 1]$ 
13:   end for
14:   for  $i \leftarrow 1$  to  $n2$  do
15:      $R[i] \leftarrow A[mid + i]$ 
16:   end for
17:    $L[n1 + 1] \leftarrow \infty$ ,  $R[n2 + 1] \leftarrow \infty$ 
18:    $i \leftarrow 1$ ,  $j \leftarrow 1$ 
19:   for  $k \leftarrow l$  to  $r$  do
20:     if  $L[i] < R[j]$  then
21:        $A[k] \leftarrow L[i++]$ 
22:     else
23:       for  $tem \leftarrow i$  to  $n1$  do\\找到广义逆序对
24:         if  $L[i] > C \cdot R[j]$  then
25:            $sum \leftarrow sum + n1 - i + 1$ \\更新 sum 逆序对和
26:         end if
27:       end for
28:        $A[k] \leftarrow R[j++]$ 
29:     end if
30:   end for
31: end function

```

problem 6.13

(1)

①证明 易知当 R 中元素个数为 13 倍数相同元素均为 k 个, 则 R 中存在 13 个常见元素。假设可以有 14 个常见元素: 则这 14 个常见元素个数总和 $sum \geq 14 \cdot \lceil \frac{n}{13} \rceil > n$, 与总元素个数为 n 矛盾。

因此 R 中存在最多 13 个不同的常见元素, 证毕。

②证明 x 为 $R[1..n]$ 中常见元素, 则设 x 在 $R[1..\lfloor \frac{n}{2} \rfloor]$ 中有 k_1 个, 在 $R[\lfloor \frac{n}{2} \rfloor + 1..n]$ 中有 k_2 个, 则 $k_1 + k_2 \geq \lceil \frac{n}{13} \rceil$, 假设 x 在两个数组中均不为常见元素, 则 $k_1 < \lceil \frac{n}{26} \rceil$, $k_2 < \lceil \frac{n}{26} \rceil$, 即 $k_1 + k_2 < 2 \cdot \lceil \frac{n}{26} \rceil < \lceil \frac{n}{13} \rceil$, 产生矛盾, 因此 x 至少是两个数组中一个数组的常见元素。

③算法设计

1. 当数组元素小于 k 时, 返回数组元素, 否则递归划分为两部分。
2. 得到的两部分结果依次与原数组进行统计, 如果元素出现次数不低于 $\lceil \frac{n}{k} \rceil$, 则加入结果集合, 重复前两步直至算法完成。
3. 时间复杂度合并时两部分结果个数不超过 2k, 遍历原始数组统计个数, 即合并复杂度为 $O(n)$, 递归产生两个子问题, 子问题规模为原来一半, 可列出递归方程为 $T(n) = 2T(\frac{n}{2}) + O(n)$, 根据主定理可得 $T(n) = O(n \log n)$ 。

算法 5 Majority 算法

```

1. Function Majority (Ori, A, n, L, R, k)           \Ori 为原始数组拷贝
2.   if len(A) ≤ k do return A                     \递归终点, 返回数组即可
4.   mid ← (L + R)/2
5.   LS ← Majority(Ori, A, L, mid, k)               \进行递归划分
6.   RS ← Majority(Ori, A, mid + 1, R, k)
7.   ans ← set_init()                               \初始化集合 ans
8.   for each item in LS and RS do
9.       if count(item, Ori) ≥  $\lceil \frac{n}{k} \rceil$  do      \count 函数用于统计元素在原数组中出现次数
10.          ans.add(item)                            \将满足结果的 item 加入集合 ans
11.  return ans

```

(2)

正常工作

(3)

存在 采用摩尔投票算法

1. 假设数组个数为奇数，其中次数 $\geq (n+1)/2$ 的数最多 1 个，如果个数为偶数，则选取最后一个元素，判断是否为众数，这样数组个数再次变为奇数。
2. 预设众数为第一个，遍历数组，当众数和当前数相同，频率加一。
3. 如果不相同，则频率减一 (相当于两元素抵消)，如果频率变为 0，则当前元素作为众数，直到遍历所有元素。
4. 再进行一次循环判断结果是否出现超过 $n/2$ 次即可，时间复杂度为 $O(n)$ 。

算法 6 MooreMajority 算法

```

1. Function MooreMajority ( $A[1 \cdots n]$ )
2.   if  $n \% 2 == 0$  do
3.     if  $\text{count}(A[n]) \geq \frac{n}{2}$  return  $A[n]$ 
4.     else  $n \leftarrow n - 1$ 
5.    $\text{res} \leftarrow A[1], \text{count} \leftarrow 1$ 
6.   for  $i \leftarrow 2$  to  $n$  do
7.     if  $\text{res} == A[i]$  then  $\text{count} \leftarrow \text{count} + 1$ 
8.     else if  $\text{count} == 0$  then  $\text{res} \leftarrow A[i], \text{count} \leftarrow 1$ 
9.     else  $\text{count} \leftarrow \text{count} - 1$ 
10.  for  $i \leftarrow 1$  to  $n$  do
11.    if  $\text{res} == A[i]$  then  $\text{count} \leftarrow \text{count} + 1$ 
12.  return  $\text{count} \geq \lceil \frac{n}{2} \rceil ? \text{res} : 0$ 

```

(4)

由 (3) 可知采用类似的摩尔投票算法，常见元素问题的下界为 $\Omega(n)$ 。而比较排序的下界为 $\Omega(n \log n)$ ，因此比较排序更容易。

problem 6.15

(1)

①排序算法 1. 对输入的 n 个点进行排序, 排序规则为按 x 由小到大排序 (x 相等时, 按 y 由小到大排序)。

2. x 最大且 y 最大的点 (即排序后的最后一个点) 一定为 *maxima*, 且前面点的 x 坐标小于等于后面的 x 坐标。

3. 从第 N 个点向前遍历每个点, $maxy$ 记录当前极大点中 y 的最大值, 当第 i 个点的大于 $maxy$ 并且第 $i-1$ 个点 (如果存在) 的横坐标不等于第 i 个点则该点即为极大点, 直至遍历所有点。

4. 时间复杂度: 排序采取归并排序时间复杂度为 $O(n \log n)$, 遍历排序后数组时间复杂度为 $O(n)$, 因此时间复杂度为 $O(n \log n)$ 。

算法 7 *MaximaSort* 算法

输入: 结构体数组 A 输出: *maxima* 点集 ans

```

1. Function MaximaSort ( $A[1 \cdots n]$ )
2.   使用归并排序结构数组  $A$  进行排序
3.    $maxy \leftarrow 1, A[0].x \leftarrow -\infty \setminus A[0]$  作为哨兵
4.   for  $i \leftarrow n$  to 1 do
5.       if  $A[i].y > maxy$  and  $A[i-1].x < A[i].x$  then
6.            $maxy \leftarrow y$ 
7.            $ans.add(A[i])$ 
8.   return  $ans$ 

```

②分治算法 1. 当数组 A 中仅有一个点时返回该点。

2. 取数组 A 中所有点的 x 坐标的中值点, 将所有点划分为两部分 $S1, S2$ 。

3. 递归方式继续划分 $A1$ 和 $A2$, 得到 *maxima* 集合 $R1$ 和 $R2$ 。

4. 显然 $R2$ 中 *maxima* 即为当前 A 的 *maxima*, 仅需要判断 $R1$ 中的 *maxima* 是否符合即可。

5. 取 $R2$ 中的 y 坐标最大值 y_{max} , 遍历 $R1$ 中所有坐标, 若点的 y 坐标大于 y_{max} , 则加入 *maxima* 集合。

6. 回到第 1 步, 直至算法完成。

7. **时间复杂度:** 合并的比较代价为 $O(n)$, 递归产生两个子问题, 子问题规模为原来一半, 可列出递归方程为 $T(n) = 2T(\frac{n}{2}) + O(n)$, 根据主定理可得 $T(n) = O(n \log n)$ 。

算法 8 *Maxima* 算法

输入: 结构体数组 A **输出:** maxima 点集 ans

1. **Function** *Maxima* ($A[1 \cdots n]$)
 2. **find** x_{mid} **in** A
 3. **divide** A **into** $A1, A2$ **based on** x_{mid}
 4. $R1 \leftarrow \text{Maxima}(A1)$ **and** $R2 \leftarrow \text{Maxima}(A2)$
 5. $ans.add(R2)$ **and find the** y_{max} **in** $R2$
 8. **for each** $item$ **in** $R1$ **do**
 9. **if** $item.y > y_{max}$ **then** $ans.add(item)$
 10. **return** ans
-

(2)

算法 9 *WrongMaxima* 算法

```

1. Function WrongMaxima ( $A[1 \cdots n]$ )
2.   if  $n \leq 1$  then return  $A$ 
3.   find  $mid\_point$  in  $A$ 
4.   divide  $A$  into  $A1, A2, A3, A4$  based on  $mid\_point$ 
5.    $R2 \leftarrow Maxima(A2)$  and  $R3 \leftarrow Maxima(A3)$  and  $R4 \leftarrow Maxima(A4)$ 
6.    $ans.add(R4)$ , find  $x_{max}, y_{max}$  in  $R4$ 
6.   for each  $item$  in  $R2$  do \\\text{左上大于右上的最大纵坐标则满足}
7.       if  $item.y > y_{max}$  then  $ans.add(item)$ 
6.   for each  $item$  in  $R3$  do \\\text{右下大于右上的最大值横坐标则满足}
7.       if  $item.x > x_{max}$  then  $ans.add(item)$ 
8.   return  $ans$ 

```

不正确: 原问题与划分后的子问题可能均没有左下象限的点, 则子问题的规模没有变为原来的 $3/4$, 递归方程退化为 $T(n) = 3T(\frac{n}{3}) + O(n) = O(n \log n)$ 。

(3)

算法核心也是一种基于比较的算法, 下界应与基于比较的排序算法下界相同, 均为 $\Omega(n \log n)$, 具体证明方法由于自身能力有限, 故不能给出。

PART II

problem 7.1

证明

假设 $2^k \leq h \leq 2^{k+1} - 1$, $k \in N$

$$2^{k-1} + 1 \leq \lfloor \frac{h}{2} \rfloor + 1 \leq 2^k$$

$$\lceil \log(2^{k-1} + 1) \rceil = k, \lceil \log(2^k) \rceil = k$$

$$\Rightarrow \lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil = k \quad (1)$$

$$2^k + 1 \leq h + 1 \leq 2^{k+1} \rightarrow \log(2^k + 1) > k, \log(2^{k+1}) = k + 1$$

$$\Rightarrow \lceil \log(h + 1) \rceil = k + 1 \quad (2)$$

因此 $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil + 1 = \lceil \log(h + 1) \rceil = k + 1$ 证毕。

堆的层数计算 h 可以理解为堆的节点个数，设堆高度为 k

则根据堆的完全二叉树性质可知 $2^k \leq h \leq 2^{k+1} - 1$ ，则 $\lceil \log(h + 1) \rceil$ 即为计算堆层数即 $k+1$ 层。

由于 $2^{k-2} - 1 < \lfloor \frac{h}{2} \rfloor < 2^k - 1$ ，因此具有 $\lfloor \frac{h}{2} \rfloor$ 结点个数的堆高度为 $k-1$ ，即 $\lceil \log(\lfloor \frac{h}{2} \rfloor + 1) \rceil$ 即堆层数为 k 层，加上最后一层则为 $k+1$ 层，因此两式均为计算堆的层数。

problem 7.2**算法设计**

1. 给定最大堆为 heap1, 新建最大堆 heap2, 存储结构元素 (value, index), 比较原则为 value。
2. 从 heap1 中取出根节点将其放入 heap2。
3. 从 heap2 中弹出根节点 Node, K 减一, 当 K 减为 0, 返回 Node 的 value 即可。
4. 插入根节点对应的左右子节点到 heap2 中。
5. 回到第 2 步。

算法分析 每次插入元素均为 heap1 当前最大元素, heap2 pop 第 i 次即为第 i 大元素。

算法 11 *KthMaxheap* 算法

输入:最大堆 heap1[1...n], K **输出:**第 K 大元素

1. **Function** *KthMaxheap* (heap1, K)
 2. *heap2* \leftarrow *init*(heap1[0], 0)
 3. **for** *i* \leftarrow 1 **to** K **do**
 4. (*value*, *index*) \leftarrow *heap2.pop*()
 5. *heap2.insert*(heap1[*index* * 2 + 1], *index* * 2 + 1)
 6. *heap2.insert*(heap1[*index* * 2], *index* * 2)
 7. **return** *value*
-

时间复杂度 heap2 的元素个数最大为 k+1, 因此每次插入删除操作复杂度均为 $O(\log(k))$ 。

最多 2k 次插入和 k 次删除, 因此总的时间复杂度为 $O(k \log k)$, 符合要求。

problem 7.3

对于 d 叉堆中第 m 层的第 n 个元素 (m, n 均从 1 开始), 则它在数组中的下标为:

$$I(m, n) = (1 + d + d^2 + \cdots + d^{m-2}) + n = \frac{d^{m-1} - 1}{d - 1} + n$$

对于其父节点, 则为第 $m-1$ 层的第 $\lfloor \frac{n}{d} \rfloor$ 个元素, 因此下标为:

$$\begin{aligned} PARENT(m, n) &= I(m-1, \lfloor \frac{n}{d} \rfloor) \\ &= \frac{d^{m-2} - 1}{d - 1} + \lfloor \frac{n}{d} \rfloor \\ &= \frac{d^{m-1} - d}{d(d-1)} + \lfloor \frac{n + d - 1}{d} \rfloor \\ &= \lfloor \frac{d^{m-1} - d}{d(d-1)} + \frac{n + d - 1}{d} \rfloor \\ &= \lfloor \frac{1}{d} \frac{d^{m-1} - d}{d-1} + n + d - 1 \rfloor \\ &= \lfloor \frac{1}{d} \frac{d^{m-1} - 1}{d-1} + n + d - 2 \rfloor \\ &= \lfloor \frac{I(m, n) + d - 2}{d} \rfloor \end{aligned}$$

对于其子节点, 则为第 $m+1$ 层的第 $d(n-1)+j$ 个元素, 因此下标为:

$$\begin{aligned} CHILD(m, n, j) &= (1 + d + d^2 + \cdots + d^{m-1}) + d(n-1) + j \\ &= \frac{d^m - 1}{d - 1} + d(n-1) + j \\ &= \frac{d^m - d}{d - 1} + d(n-1) + 1 + j \\ &= d(\frac{d^{m-1} - 1}{d - 1} + n - 1) + j + 1 \\ &= d(I(m, n) - 1) + j + 1 \end{aligned}$$

因此当元素数组下标为 i 的父节点和第 j 个字节节点下标为,

$$D-ARY-PARENT(i) = \lfloor \frac{i-2}{d} + 1 \rfloor$$

$$D-ARY-CHILD(i, j) = d(i-1) + j + 1, \text{ 得证。}$$

problem 7.4

对于完美二叉树，设其树的高度为 h ，则最底层共有 2^h 个树叶。
其节点总数为 $2^{h+1} - 1$ ，其所有节点的高度和为 $2^{h+1} - h - 2$. (1)

对堆节点个数 n 进行归纳，易得 $n = 1$ $0 \leq 1 - 1 = 0$ ，满足。

假设 $n \leq k$ ，令堆的总高度和与节点总数差最大值为 $gap(k)$ ，均有 $gap(n) \leq -1$ 。

当 $n = k + 1$ 时，根据堆的性质，堆的左右子树至少有一个为完美二叉树。

1. 若堆为完美二叉树，设堆高度为 h ，高度和与节点总数差为：

$$gap(k + 1) = 2^{h+1} - h - 2 - (2^{h+1} - 1) = -h - 1 \leq -1.$$

2. 当左子树为完美二叉树，其高度为 $h - 1$ ，其高度和为 $2^h - h - 1$ ，节点总数为 $2^h - 1$ ，对右子树由归纳假设，其高度和与节点总数差最大值为 -1 。
则：

$$gap(k + 1) \leq h - 1 + (2^h - h - 1) - (2^h - 1) + \max(gap(k_i)) = -2 \leq -1$$

3. 当左子树不为完美二叉树而右子树为完美二叉树，高度为 $h - 2$ ，其高度和为 $2^{h-1} - h$ ，节点总数为 $2^{h-1} - 1$ ，对左子树由归纳假设，其高度和与节点总数差最大值为 -1 。

则：

$$gap(k + 1) \leq h - 1 + (2^{h-1} - h) - (2^{h-1} - 1) + \max(gap(k_i)) = -1$$

当最底层仅有一个节点时，节点个数和所有节点高度和的差值为

$$gap(n) = h - 1 + ((2^{h-1} - 1) - (2^{h-1})) + ((2^{h-1} - h) - (2^{h-1} - 1)) = -1$$

即 n 个节点的堆中，所有节点的高度之和最多为 $n-1$ ，在最底层仅有一个节点时等号成立。

problem 7.6**算法设计**

1. 使用归并排序将每个单词内部字母按照从小到大排序, 使得查找易位词时方便操作。

2. 建立结构体 *Data*, 其中包含原始字符串 *ori*, 排序后字符串 *key*, 以及该字符串出现的次数 *count*。

3. 依次遍历每个单词, 构造结构 *data_cur*, 以判断是否在集合 *data_set* 中:

①. 如果在集合中, 结构体为 *data_old*, 根据题意换为字典序小的字符串, 并更新出现次数, 即 $count + 1$ 。

②. 如果没有在集合中, 则将该结构体加入 *data_set* 即可。

4. 遍历整棵树, 将 $count > 0$ 的节点的 *ori* 输出。算法完成。

具体算法实现请见算法 FindTrans: [算法 FindTrans](#)

时间复杂度 对每个单词内部进行归并排序, 设单词总数为 num , 单词长度为 $len(x_i)$ 时间复杂度为 $\sum_{i=0}^{num} len(x_i) \log(len(x_i)) = O(n \log n)$; 集合每次的插入操作时间复杂度为 $\log n$, 进行次数为 $O(n)$, 因此维护集合有序性的时间复杂度为 $O(n \log n)$ 。因此总的时间复杂度为 $O(n \log n)$ 。

算法 12 FindTransposition 算法

输入: 单词序列 oris, 单词个数 num

输出: 易位词序列 res

```
1: function FINDTRANS(keys,num)
2:   data_set  $\leftarrow$  set_init(), res  $\leftarrow$  set_init()
3:   for i  $\leftarrow$  0 to num do
4:     key  $\leftarrow$  merge_sort(oris[i])\\得到归并排序后的关键词 key
5:     data_cur(key,oris[i],0)\\构造结构体 data_cur
6:     if find(data_set,data_old) == True then
7:       /*如果找到相同的 key, 则返回对应的结构体data_old*/
8:       data_old.count++
9:       data_old.ori  $\leftarrow$  min(data_old.ori,oris[i])
10:      /*更新 count 计数, 并将 ori 设为字典序较小的字符串*/
11:    else
12:      data_set.add(data_cur)\\如果没有则将 data_cur 加入集合中
13:    end if
14:  end for
15:  for all item in data_set do
16:    if item.count  $\geq$  1 then
17:      res.add(item.ori)\\如果出现次数不为 0, 即为易位词, 加入 res
18:    end if
19:  end for
20:  return res
21: end function
```
