## 第8章 线性时间排序

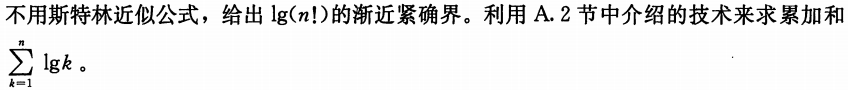
#### 练习 8.1-1



**问题分析**

在比较排序算法中，要确定一个元素的位置，该元素至少要进行两次比较（首尾元素除外）：与左边元素比较，与右边元素比较。而首尾元素只需要至少进行一次比较。因此，在一个*n*个元素的序列中，所有元素参与比较的次数之和至少为2*n*−2。而一次比较有2个元素参与，所以至少需要的比较次数为(2*n*−2) / 2 = *n*−1。所以在一棵比较排序算法的决策树中，一个叶结点可能的最小深度为*n*−1。

#### 练习 8.1-2



**问题分析**

#### 练习 8.1-3



**问题分析**

一棵高度为*h*的决策树至多有2*h*个叶结点。如果叶结点为*m*个，这意味着排序的输入的情况有*m*种，那么决策树的高度*h*满足2*h* ≥ *m*，即*h* ≥ lg*m*。

1) 输入情况有*n*!/2种



2) 输入情况有*n*!/*n*种

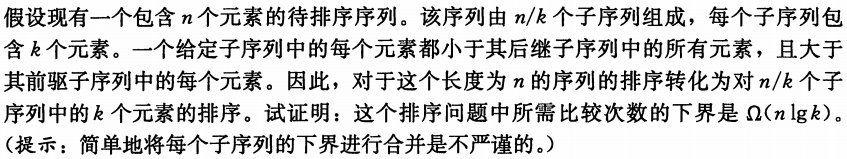


3) 输入情况有*n*!/2*n*种



所以，对于以上3种情况，比较排序都不能达到线性时间。

#### 练习 8.1-4



**问题分析**

对于长度为*k*的子数组，一共有*k*!种排列。一共有*n*/*k*个长度为*k*的子数组，所以整个数组一共有(*k*!) *n*/*k*种排列。所以，决策树的高度满足



所以，这个排序问题所需比较时间的下界是Ω(*n*lg*k*)。

#### 练习 8.2-1



**问题分析**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| A | 6 | 0 | 2 | 0 | 1 | 3 | 4 | 6 | 1 | 3 | 2 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 2 | 2 | 2 | 1 | 0 | 2 |

(a)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 4 | 6 | 8 | 9 | 9 | 11 |

(b)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  |  |  |  |  | 2 |  |  |  |  |  |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 4 | 5 | 8 | 9 | 9 | 11 |

(c)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  |  |  |  |  | 2 |  | 3 |  |  |  |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 4 | 5 | 7 | 9 | 9 | 11 |

(d)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  |  |  | 1 |  | 2 |  | 3 |  |  |  |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 3 | 5 | 7 | 9 | 9 | 11 |

(e)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  |  |  | 1 |  | 2 |  | 3 |  |  | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 3 | 5 | 7 | 9 | 9 | 10 |

(f)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  |  |  | 1 |  | 2 |  | 3 | 4 |  | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 3 | 5 | 7 | 8 | 9 | 10 |

(g)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  |  |  | 1 |  | 2 | 3 | 3 | 4 |  | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 3 | 5 | 6 | 8 | 9 | 10 |

(h)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  |  | 1 | 1 |  | 2 | 3 | 3 | 4 |  | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 2 | 2 | 5 | 6 | 8 | 9 | 10 |

(i)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  | 0 | 1 | 1 |  | 2 | 3 | 3 | 4 |  | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 1 | 2 | 5 | 6 | 8 | 9 | 10 |

(j)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B |  | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 |  | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 1 | 2 | 4 | 6 | 8 | 9 | 10 |

(k)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 |  | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 1 | 2 | 4 | 6 | 8 | 9 | 10 |

(l)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| B | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 6 | 6 |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| C | 1 | 2 | 4 | 6 | 8 | 9 | 9 |

(m)

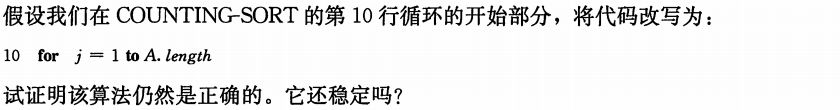
#### 练习 8.2-2



**问题分析**

假设原始数组中*A*[*i*]和*A*[*j*]两个元素是相等的，它们的值都为*x*，并且*i* < *j*，这说明*A*[*i*]排列在*A*[*j*]之前。在计数排序算法中，*A*[*j*]会被先取出，它被排在第*C*[*x*]位，然后*C*[*x*]会被减1。*A*[*i*]在*A*[*j*]之后被取出，此时*C*[*x*]的值已经被减小了，那么在排序后的数组中，*A*[*i*]一定会排在*A*[*j*]的前面。所以计数排序算法是稳定的。

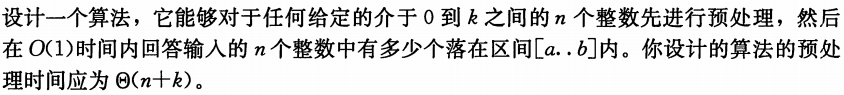
#### 练习 8.2-3



**问题分析**

如果for循环改为从1到*A*.*length*，计数排序算法仍然正确，但是不稳定。不稳定是原因是相等元素中原先排在前面的元素会被先取出，排在后面；而原先排在后面的元素会被后取出，排在前面。

#### 练习 8.2-4



**问题分析**

COUNTING(*A*)

Let *C*[-1 .. *k*] be a new array

**for** *i* = -1 🡪 *k*

*C*[*i*] = 0

**for** *j* = 1 🡪 *A*.*length*

*C*[*A*[*j*]]++

**for** *i* = 1 🡪 *k*

*C*[*i*] += *C*[*i*-1]

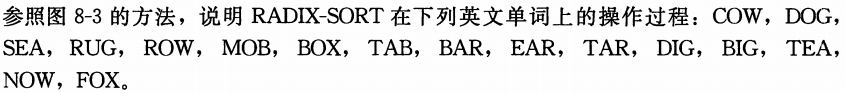
**return** *C*

INTERVAL\_NUM(*A*, *a*, *b*)

*C* = COUNTING(*A*)

**return** *C*[*b*]–*C*[*a*-1]

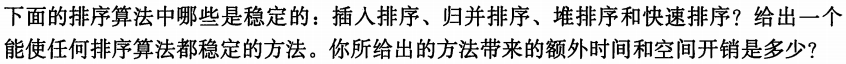
#### 练习 8.3-1



**问题分析**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| COW  DOG  SEA  RUG  ROW  MOB  BOX  TAB  BAR  EAR  TAR  DIG  BIG  TEA  NOW  FOX | **==>** | SEA  TEA  MOB  TAB  DOG  RUG  DIG  BIG  BAR  EAR  TAR  COW  ROW  NOW  BOX  FOX | **==>** | TAB  BAR  EAR  TAR  SEA  TEA  DIG  BIG  MOB  DOG  COW  ROW  NOW  BOX  FOX  RUG | **==>** | BAR  BIG  BOX  COW  DIG  DOG  EAR  FOX  MOB  NOW  ROW  RUG  SEA  TAB  TAR  TEA |

#### 练习 8.3-2



**问题分析**

插入排序和归并排序是稳定的，而堆排序和快速排序是不稳定的。

一个能使任何排序算法都稳定的方法是：记录数组中每个元素的下标，在比较2个元素大小的关系时候，先比较元素的值，如果相等，再比较下标。这会需要额外的Θ(*n*)空间来存储数组元素的下标，但是在时间上只会增加常量因子。

#### 练习 8.3-3



**问题分析**

#### 练习 8.3-4



**问题分析**

定义一种*n*进制的3位数*xyz*，每位数可能取值为0 ~ *n*−1，其数值为(*xn*2 + *yn* + *z*)。显然，这样的3位数最小值为0，最大值为(*n*−1)*n*2 + (*n*−1)*n* + (*n*−1) = *n*3−1，所以其取值范围为0 ~ *n*3−1。

对这样定义的3位数进行基数排序，位数*d* = 3，每一位数有*k* = *n*个可能的取值。排序的时间复杂度为Θ(*d*(*n*+*k*)) = Θ(3(*n*+*n*)) = Θ(*n*)。

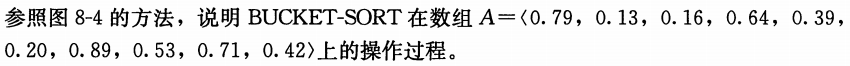
#### 练习 8.3-5\*



**问题分析**

本节给出的第一个卡片排序算法是先按最高有效位来进行排序。第一轮排序对最高有效位进行，最坏情况会得到10堆卡片。对每一堆卡片各自进行一轮排序，排序对次高有效位进行。这样每一堆卡片在最坏情况下又被分为10堆。以此类推，每进行一轮排序，会得到新的10堆卡片。那么，对最高有效位的排序需要进行1轮，对次高有效位的排序需要进行10轮，而对次次高有效位的排序需要进行102轮，以此类排，直到对最低有效位进行排序，需要进行10*d*−1轮，加起来一共是(10*d*−1)/9。

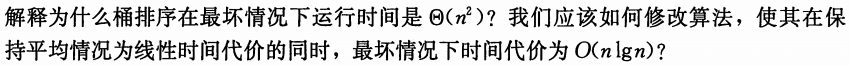
#### 练习 8.4-1



**问题分析**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A |  |  | B |  |  |  |  |  |  |
| 1 | 0.79 |  | 0 | / |  |  |  |  |  |  |
| 2 | 0.13 |  | 1 |  |  | 0.13 |  |  | 0.16 | / |
| 3 | 0.16 |  | 2 |  |  | 0.20 | / |  |  |  |
| 4 | 0.64 |  | 3 |  |  | 0.39 | / |  |  |  |
| 5 | 0.39 |  | 4 |  |  | 0.42 | / |  |  |  |
| 6 | 0.20 |  | 5 |  |  | 0.53 | / |  |  |  |
| 7 | 0.89 |  | 6 |  |  | 0.64 | / |  |  |  |
| 8 | 0.53 |  | 7 |  |  | 0.71 |  |  | 0.79 | / |
| 9 | 0.71 |  | 8 |  |  | 0.89 | / |  |  |  |
| 10 | 0.42 |  | 9 | / |  |  |  |  |  |  |

#### 练习 8.4-2

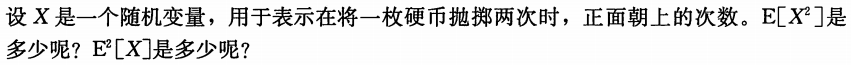


**问题分析**

桶排序的最坏情况发生在所有元素都放在一个桶中的时候，这时候对桶内的元素调用插入排序的时间为Θ(*n*2)。

如果要求桶排序在最坏情况下的时间复杂度为O(*n*lg*n*)，可以考虑在对桶内元素进行排序时采用归并排序和堆排序等渐近时间为Θ(*n*lg*n*)的排序算法。

#### 练习 8.4-3



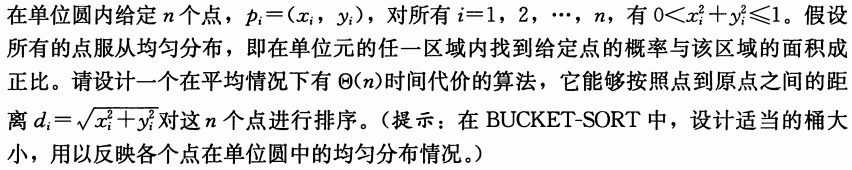
**问题分析**

下表列出了随机变量*X*的取值及其概率。

|  |  |
| --- | --- |
| *X* | *P*(*X*) |
| 0 | 1/4 |
| 1 | 1/2 |
| 2 | 1/4 |



#### 练习 8.4-4\*



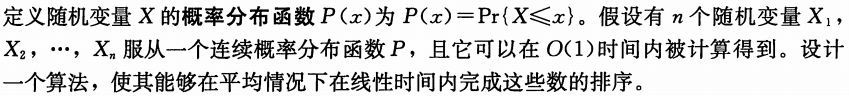
**问题分析**

可以按照面积将圆等分为*n*等份。由于我们要考察的是点到圆心的距离，所以可以用同心圆来将圆*n*等分，如下图所示。

等分的目标是使同心圆之间区域的面积相等。单位圆的面积S = *π*∙12 = *π*。*n*等分之后，每个区间的面积*s* = *π*/*n*。可以算得同心圆的半径分别为、、…、。对于第*i*个等分区间，它内部的点到圆心的距离在[*ri*-1, *ri*)之中。因此，可以对单位圆内的*n*个点进行桶排序，将到圆心距离位于[*ri*-1, *ri*)之中的点放入第*i*个桶中。

如果按照距离来把点放入相应的桶中，还需要根据距离来查找相应的区间，对每个点来说，查找花费的时间为O(lg*n*)，因此算法的代价为O(*n*lg*n*)。而如果利用面积作为参考，可以在Θ(1)时间内将一个点放入到相应的桶中。对于点(*xi*, *yi*)，经过该点的同心圆的面积*Ai* = *π*(*xi*2+*yi*2)，那该放入的桶的序号为。

#### 练习 8.4-5\*

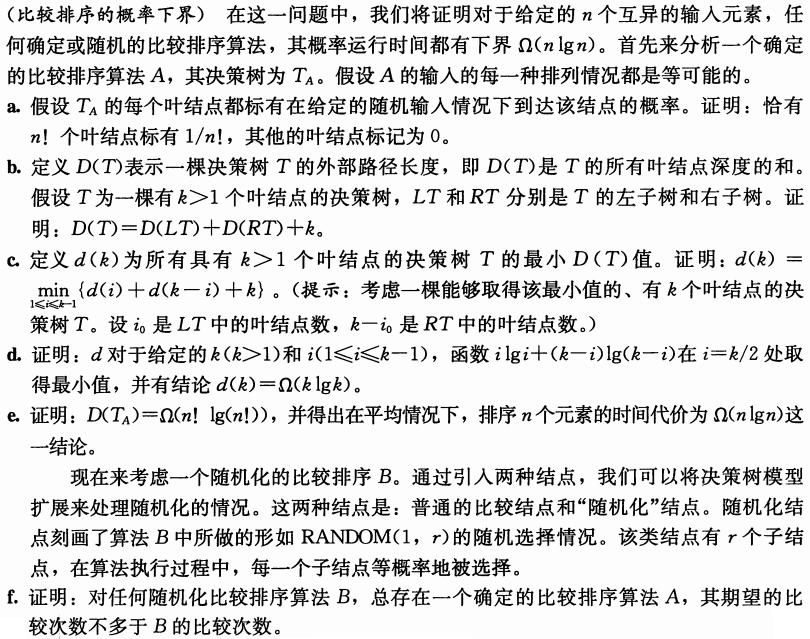


**问题分析**

按照概率分布将元素划分到*n*个桶中。元素*Xi*被划分到第个桶中。按照桶排序的算法流程即可进行排序。

假如两个元素*Xi* < *Xj*，根据概率分布函数的定义，必定有*P*(*Xi*) ≤ *P*(*Xj*)。因此，要么*Xi*和*Xj*被放入同一个桶中，要么*Xi*放在一个比*Xj*所在的桶的下标更小的桶中。所以按照概率分布来划分，这样的桶排序算法是正确的。

#### 思考题 8-1



**a-问题分析**

*n*个互异的输入元素，一共有*n*!种排列。由于每种排列是等可能的，所以任意一种排列的概率为1/*n*!。在决策树中，每一个叶结点对应一种特定的排列，同时每一种特定排列也只能到达一个叶结点，也就是说，元素的排列和叶结点是一一对应的。所以，有*n*!个叶结点，其中每一个叶结点的概率为1/*n*!。而如果还有其他叶结点，其概率一定为0。

**b-问题分析**

由于*T*的叶结点个数*k* > 1。所以*T*的根结点必然至少包含一个非空的子树，因为如果*T*的根结点的2个子树都为空，那么*T*只有1个叶结点，与题目要求不符。又由于决策树是一个完全二叉树，所以*T*的2个子树都非空。根据这一分析，*T*的根结点肯定不会是叶结点，*T*的叶结点肯定只位于*T*的根结点的子树上。*T*的左子树的叶结点深度之和为*D*(*LT*)，右子树的叶结点深度之和为*D*(*RT*)。而根结点的加入，会让每个叶结点的深度加1，由于一共有*k*个叶结点，所以叶结点的深度之和会增加*k*。故*D*(*T*) = *D*(*LT*) + *D*(*RT*) + *k*。

**c-问题分析**

根据问题b的分析，*T*的叶结点深度之和*D*(*T*) = *D*(*LT*) + *D*(*RT*) + *k*。



**d-问题分析**

对函数*f*(*i*) = *i*lg*i* + (*k*-*i*)lg(*k*-*i*)求导，*f’*(*i*) = lg*i* – lg(*k*-*i*)。显然，导数在*i* = *k*/2处为零；当*i* < *k*/2时导数小于0；当*i* > *k*/2时导数大于0。这说明函数*f*(*i*)在*i* = *k*/2处取最小值，并且*f*(*i*)min = *k*lg(*k*/2)。

可以用代入法来证明*d*(*k*) = Ω(*k*lg*k*)。

对于初始情况*k* = 1，假设*d*(1) = 1。无论*c*取何值，显然有*d*(1) > *c*·1·lg1 = 0。

下面进入归纳阶段。



所以*d*(*k*) = Ω(*k*lg*k*)。

**e-问题分析**

根据问题a的分析，决策树*TA*有*n*!个叶结点，其中每一个叶结点的概率为1/*n*!。而如果还有其他叶结点，其概率一定为0。忽略概率为0的叶结点，并不影响最终的结果。显然有



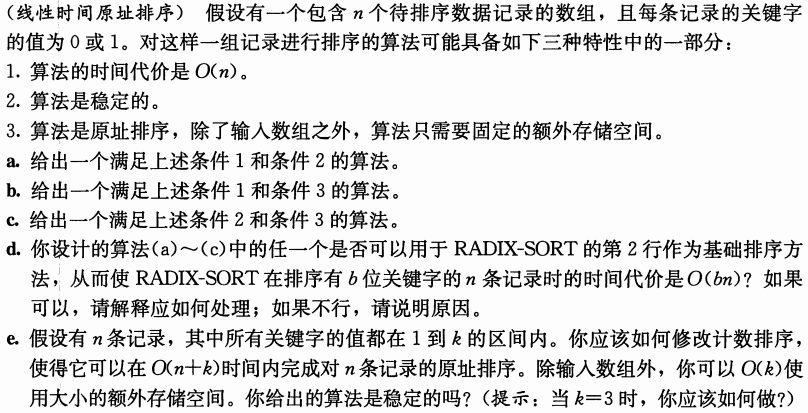
*D*(*TA*)是所有叶结点的深度总和，而每个叶结点的概率为1/*n*!，所以叶结点的平均深度为



排序的平均时间代价也就是叶结点的平均深度，即Ω(*n*lg*n*)。

**f-问题分析**

#### 思考题 8-2



**a-问题分析**

计数排序

**b-问题分析**

快速排序。因为数组只包含0和1，所以快速排序进行一轮划分即可结束，时间复杂度为O(*n*)。

**c-问题分析**

插入排序。

**d-问题分析**

1) 计数排序：可以。

2) 快速排序：不可以，因为排序不稳定。

3) 插入排序：不可以，因为时间代价不为O(*n*)。

**e-问题分析**

IN\_PLACE\_COUNTING\_SORT(*A*, *k*)

let *C*[1 .. *k*] be a new array

**for** *i* = 1 🡪 k

*C*[*i*] = 0

**for** *j* = 1 🡪 *A*.*length*

*C*[*A*[*j*]]++

**for** *i* = 2 🡪 k

*C*[*i*] += *C*[*i*-1]

*j* = *A*.*length*

**while** *j* >= 1

**if** *C*[*A*[*j*]] != *j*

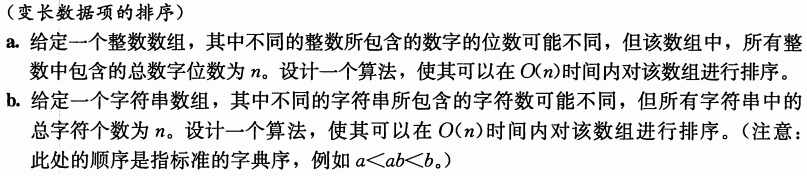
exchange *A*[*j*] with *A*[*C*[*A*[*j*]]]

**else**

*C*[*A*[*j*]]--

*j*--

#### 思考题 8-3



**a-问题分析**

考虑采用基数排序。在基数排序中，在第*i*轮排序中，要排序的是所有数字的第*i*位。如果某个数字的第*i*位为0，那么它会被排列在数组的前部。并且，在之后每一轮(*i*+1, *i*+2, …)排序中，这个数字参与排序的数位都为0，所以它在数组中的位置不再发生改变。利用这个特点，可以修改基数排序：在第*i*轮排序中，如果遇到第*i*位为0的数字，那么在第*i*轮排序之后，这些数字不参与之后每一轮的排序。

VARIABLE\_LENGTH\_RADIX\_SORT(*A*, *d*)

*sort\_start* = 1

**for** *i* = 1 🡪 *d*

use a stable sort to sort array *A*[*sort\_start* .. *A*.*length*] on digit *i*

count the number *n*0 of elements whose digit *i* is 0

*sort\_start* += *n*0

在以上算法中，假如一共有*m*个数字，如果第*j*个数字有*dj*位，那么这个数字一共会参与(*dj*+1)轮排序，因此这个数字对算法时间复杂度的贡献为O(*dj*+1)。对所有数字的算法时间复杂度的贡献求和，可以得到算法的时间复杂度



注意，在以上算法中，每次迭代都会统计第*i*位为0的元素的个数，如果采用计数排序，这一统计可以在O(1)时间内完成。因此在计算算法的时间复杂度时，可以不用考虑这一步的时间。

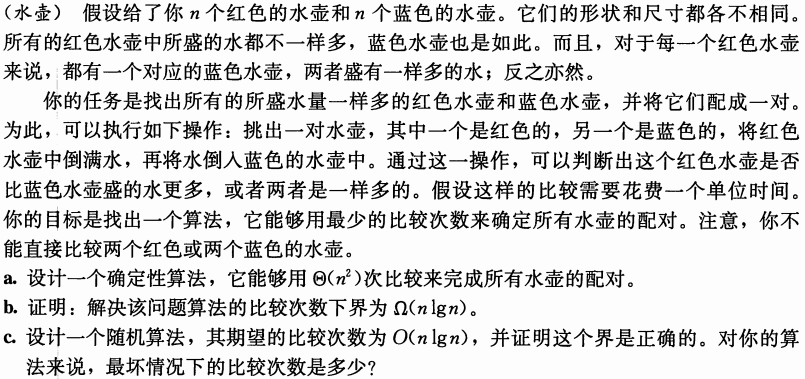
**b-问题分析**

同样考虑采用基数排序。在对数字的基数排序中，排序从低位开始，再到高位。与此不同，在对字符串的基数排序中，排序先从最左边的字符开始，再到右边的字符。下面给出一个例子。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| fix  fox  box  I  at  fit  bat  a | **==>** | at  a  box  bat  fix  fox  fit  I | **==>** | a  at  bat  box  fix  fit  fox  I | **==>** | a  at  bat  box  fit  fix  fox  I |

注意，第一轮排序对所有的字符串的最左边的字符进行排序。在第一轮排序后，所有的字符串按照最左边的字符分组，每一组的字符串的最左边的字符相同，并且排在前面的组中的字符串在最终的排序中一定排在后面的组中的字符串的前面。利用这一特点，在第二轮排序中，只针对同一个组内的字符串进行排序。依此递归，最后可以排序所有字符串。

#### 思考题 8-4



**a-问题分析**

依次选取红色水壶，将其与所有的蓝色水壶进行比较，从而挑出与其容量一样的蓝色水壶。这一算法的时间复杂度为Θ(*n*2)。

JUG\_MATCH(*R*, *B*)

let *M*[1 .. *n*] be a new array

*n* = *R*.*length*

**for** *i* = 1 🡪 *n*

**for** *j* = 1 🡪 *n*

**if** *R*[*i*] == *B*[*j*]

*M*[*i*] = (*i*, *j*)

**break**

**return** *M*

**b-问题分析**

**c-问题分析**

按照题目意思，每一个红色水壶都有一个蓝色水壶与之容量相等。比较简单的一个想法是，将所有红色水壶排序，将所有蓝色水壶排序，排序后处于相同位置的红色水壶和蓝色水壶一定是容量相等的。排序可以采用随机化的快速排序，期望时间复杂度为O(*n*lg*n*)。然而，这个算法违反了题目要求。因为题目只允许红色水壶和蓝色水壶之间的比较，而这个算法需要红色水壶与红色水壶的比较，以及蓝色水壶与蓝色水壶的比较，这超出了题目的限制。

为了满足题目要求，可以考虑对快速排序做出修改，详见下面的伪代码RANDOM\_QUICK\_JUG\_MATCH。每次调用RANDOM\_QUICK\_JUG\_MATCH，从红色水壶*R*中随机选出一个*r*，将*r*与所有蓝色水壶*B*比较，从而将蓝色水壶划分为3类：容量小于*r*的蓝色水壶*B*<，容量大于*r*的蓝色水壶*B*>，容量等于*r*的蓝色水壶*b*。反过来，用蓝色水壶*b*将红色水壶也分为3类：容量小于*b*的蓝色水壶*R*<，容量大于*b*的蓝色水壶*R*>，容量等于*b*的蓝色水壶*r*。红色水壶*r*与蓝色水壶*b*的容量相等，二者配成一对。再用递归的方式对*R*<和*B*<进行排序，以及对*R*>和*B*>进行排序。

RANDOM\_JUG\_PARTITION(*R*, *B*, *p*, *r*)

*k* = RANDOM(*p*, *r*)

*i* = *p*-1

*j* = *p*

**while** *j* <= *r*

**if** *B*[*j*] < *R*[*k*]

*i*++

exchange *B*[*i*] with *B*[*j*]

*j*++

**elseif** *B*[*j*] == *R*[*k*]

exchange *B*[*r*] with *B*[*j*]

**else**

*j*++

*k* = *i*+1

exchange *B*[*k*] with *B*[*r*]

*i* = *p*-1

*j* = *p*

**while** *j* <= *r*

**if** *R*[*j*] < *B*[*k*]

*i*++

exchange *R*[*i*] with *R*[*j*]

*j*++

**elseif** *R*[*j*] == *B*[*k*]

exchange *R*[*r*] with *R*[*j*]

**else**

*j*++

exchange *R*[*k*] with *R*[*r*]

**return** *k*

RANDOM\_QUICK\_JUG\_MATCH (*R*, *B*, *p*, *r*)

**if** *p* < *r*

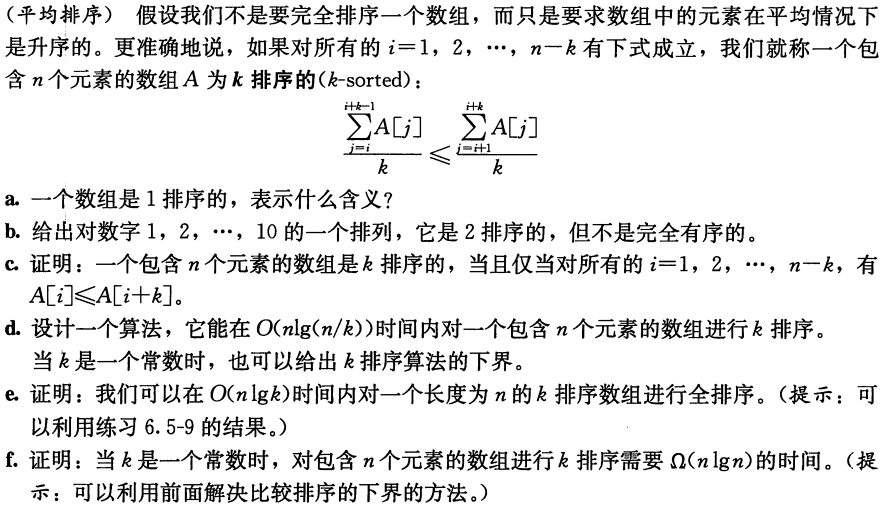
*q* = RANDOM\_JUG\_PARTITION(*R*, *B*, *p*, *r*)

RANDOM\_QUICK\_JUG\_MATCH (*R*, *B*, *p*, *q*-1)

RANDOM\_QUICK\_JUG\_MATCH (*R*, *B*, *q*+1, *r*)

该算法实际上是一个随机化的快速排序过程，只不过需要排序的是两个*n*元素的数组，期望时间复杂度仍然为O(*n*lg*n*)。

#### 思考题 8-5



**a-问题分析**

一个数组*A*是1排序的，说明对所有*i* = 1, 2, … , *n*-1，有*A*[*i*] ≤ *A*[*i*+1]，这说明数组是完全排序的。

**b-问题分析**

2 1 4 3 6 5 8 7 10 9

**c-问题分析**

1) 先证。

由于*A*[*i*] ≤ *A*[*i*+*k*]，显然有*A*[*i*] + (*A*[*i*+1] + *A*[*i*+2] + … + *A*[*i*+*k*−1]) ≤ (*A*[*i*+1] + *A*[*i*+2] + … + *A*[*i*+*k*−1]) + *A*[*i*+*k*]，所以有，所以成立。

2) 再证

由于，即*A*[*i*] + (*A*[*i*+1] + *A*[*i*+2] + … + *A*[*i*+*k*−1]) ≤ (*A*[*i*+1] + *A*[*i*+2] + … + *A*[*i*+*k*−1]) + *A*[*i*+*k*]，不等式左右两边括号内的和完全相等，所以*A*[*i*] ≤ *A*[*i*+*k*]成立。

**d-问题分析**

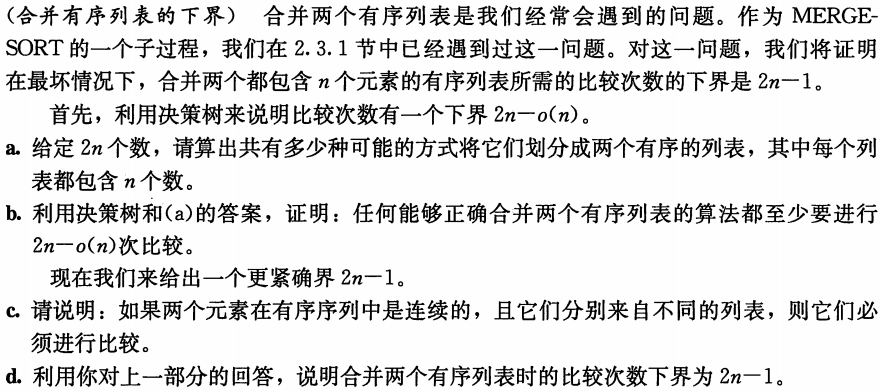
*k*排序可以这样进行，将数组分为*k*个子数组，如下图所示。然后对每个子数组分别进行排序。显然，这样这排序后的数组满足*A*[*i*] ≤ *A*[*i*+*k*] (*i* = 1, 2, … , *n*−*k*)，因此这样的*k*排序算法是正确的。每个子数组的大小为*n*/*k*，可以在O((*n*/*k*)lg(*n*/*k*))内对一个子数进行排序。一共有k个子数组，因此对整个数组进行*k*排序的时间为O(*n*lg(*n*/*k*))。

|  |  |
| --- | --- |
| 原数组 | 1 2 … *k* *k*+1 *k*+2 … 2*k* 2*k*+1 2*k*+2 … 3*k* … … |
| 子数组1 | 1 *k*+1 2*k*+1 … … |
| 子数组2 | 2 *k*+2 2*k*+2 … … |
| **∙**  **∙**  **∙** | **∙**  **∙**  **∙** |
| 子数组*k* | *k* 2*k* 3*k* … … |

**e-问题分析**

**f-问题分析**

#### 思考题 8-6



**a-问题分析**

**b-问题分析**

**c-问题分析**

**d-问题分析**