10.5 归并排序

核心思路：基于**分治**思想，将两个或两个以上的**有序序列合并**为一个新的有序序列。

归并次序：

- 自顶向下：将序列拆分直到有序；然后使用归并算法得到排序结果。
- 自底向上：将序列看成多条有序子序列，并将子序列两两合并。

应用场景：

- 求逆序数对：通过归并排序，快速计算序列中逆序数对的数量。

10.5.1 二路归并

二路归并算法将两个有序序列合并为一个新的有序序列。

对两有序序列，分别取出这两个序列中**键值最小**的项，并选出两个数据项中最小的放置到新的序列中；循环上述动作，直到两个序列中的所有数据**都已被放置**到新的序列中。此时，新的序列即为排序结果。

对于长度分别为n和m的有序序列，二路归并算法的主循环最多执行$n+m$次，因此其复杂度为$\boldsymbol{O(n+m)}$。

二路归并伪代码

算法10-8 二路归并 TwoWayMerge(a, l_x, r_x, l_y, r_y)

输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

输出： 将两个有序序列合并后的新有序序列t

InitList(t) //新序列t, 存放合并后的有序序列
$i \leftarrow l_x$
$j \leftarrow l_y$
$k \leftarrow 0$
while $i \le r_x$ **or** j $\le r_y$ **do**
| **if** $j > r_y$ **or** $(i \le r_x$ **and** $a_i \le a_j)$ **then**
| | $t_k \leftarrow a_i$ //将a_i添加至t
| | $i \leftarrow i + 1$
| **else**
| | $t_k \leftarrow a_j$ //将a_j添加至t
| | $j \leftarrow j + 1$
| **end**
| $k \leftarrow k + 1$
end
return t

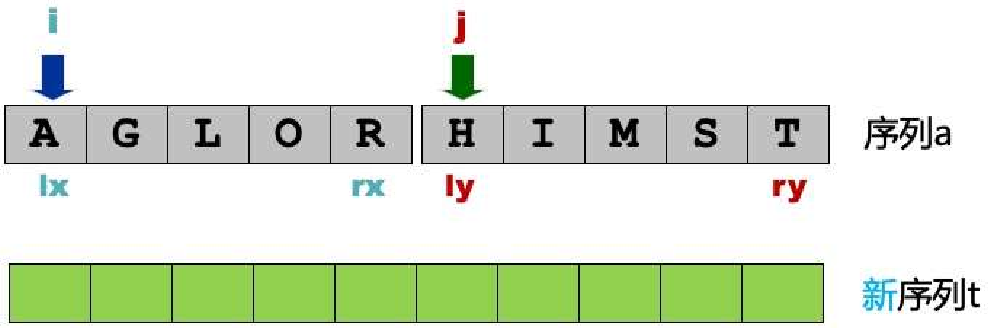
思考：可否不使用新序列，直接在原序列a上合并

二路归并示例

算法10-8 二路归并 $\text{TwoWayMerge}(a, l_x, r_x, l_y, r_y)$

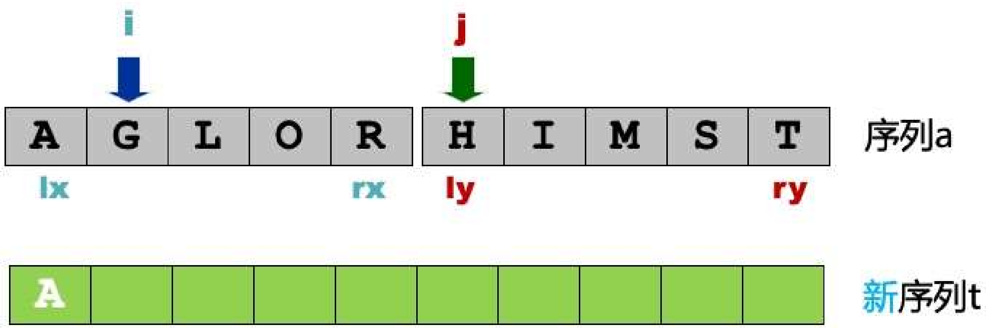
输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

输出： 将两个有序序列合并后的新有序序列t



二路归并示例

算法10-8 二路归并 TwoWayMerge(a, l_x, r_x, l_y, r_y)
输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y
输出： 将两个有序序列合并后的新有序序列t

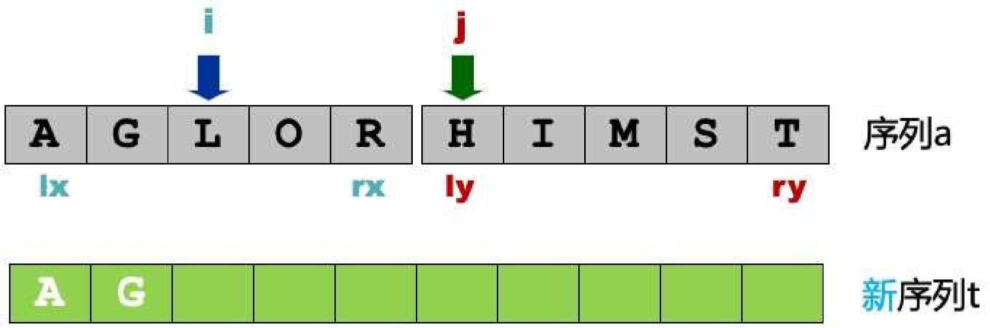


二路归并示例

算法10-8 二路归并 TwoWayMerge(a, l_x, r_x, l_y, r_y)

输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

输出： 将两个有序序列合并后的新有序序列t

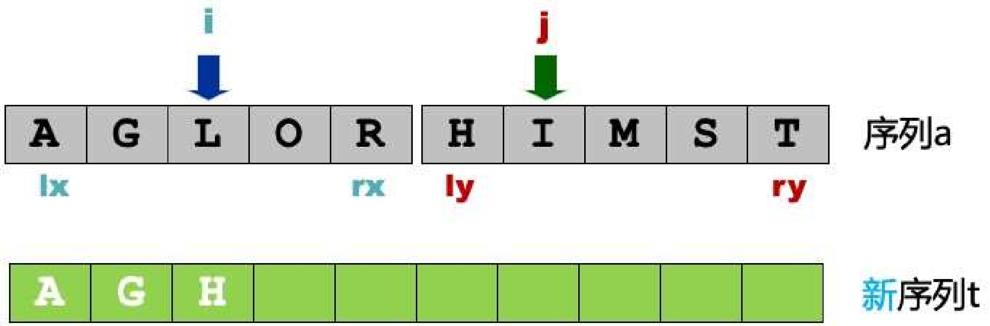


二路归并示例

算法10-8 二路归并 TwoWayMerge(a, l_x, r_x, l_y, r_y)

输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

输出： 将两个有序序列合并后的新有序序列t

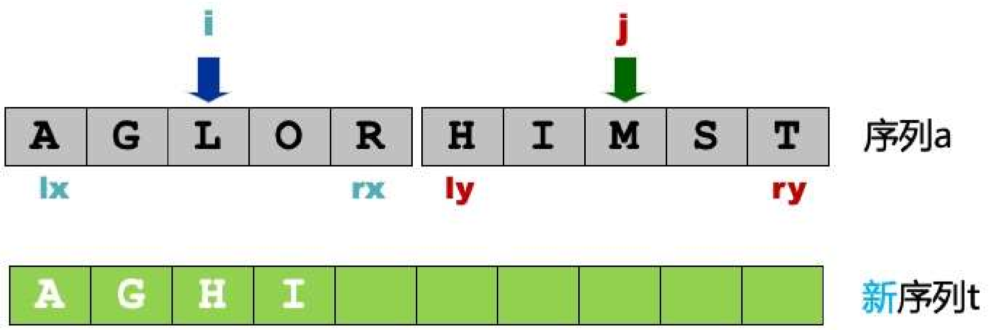


二路归并示例

算法10-8 二路归并 TwoWayMerge(a, l_x, r_x, l_y, r_y)

输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

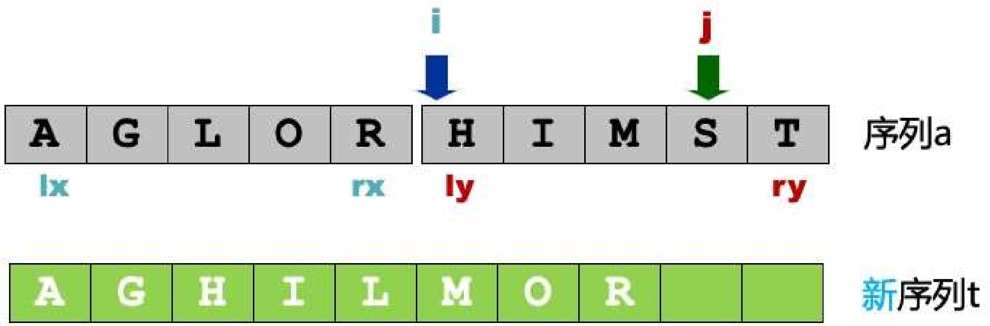
输出： 将两个有序序列合并后的新有序序列t



二路归并示例

算法10-8 二路归并 TwoWayMerge(a, l_x, r_x, l_y, r_y)
输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y
输出： 将两个有序序列合并后的新有序序列t

- $i > r_x$ 子序列$a[l_x, r_x]$合并结束



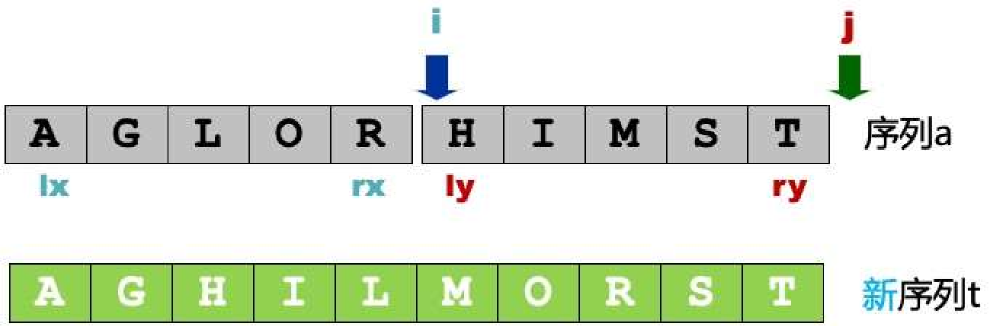
- 子序列$a[l_y, r_y]$中剩下的元素可以按序移至新序列t（不需要比较！）

二路归并示例

算法10-8 二路归并 TwoWayMerge(a, l_x, r_x, l_y, r_y)

输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y

输出： 将两个有序序列合并后的新有序序列t



- 合并结束

- 比较次数：$O(r_y - l_y + r_x - l_x + 2)$

高等教育出版社

单选题 1分

根据合并算法，合并两个长度均为n的有序序列，元素之间比较次数的最大值和最小值是

A 2n 和 n

B 2n 和 1

C 2n-1 和 n

D 2n-1 和 1

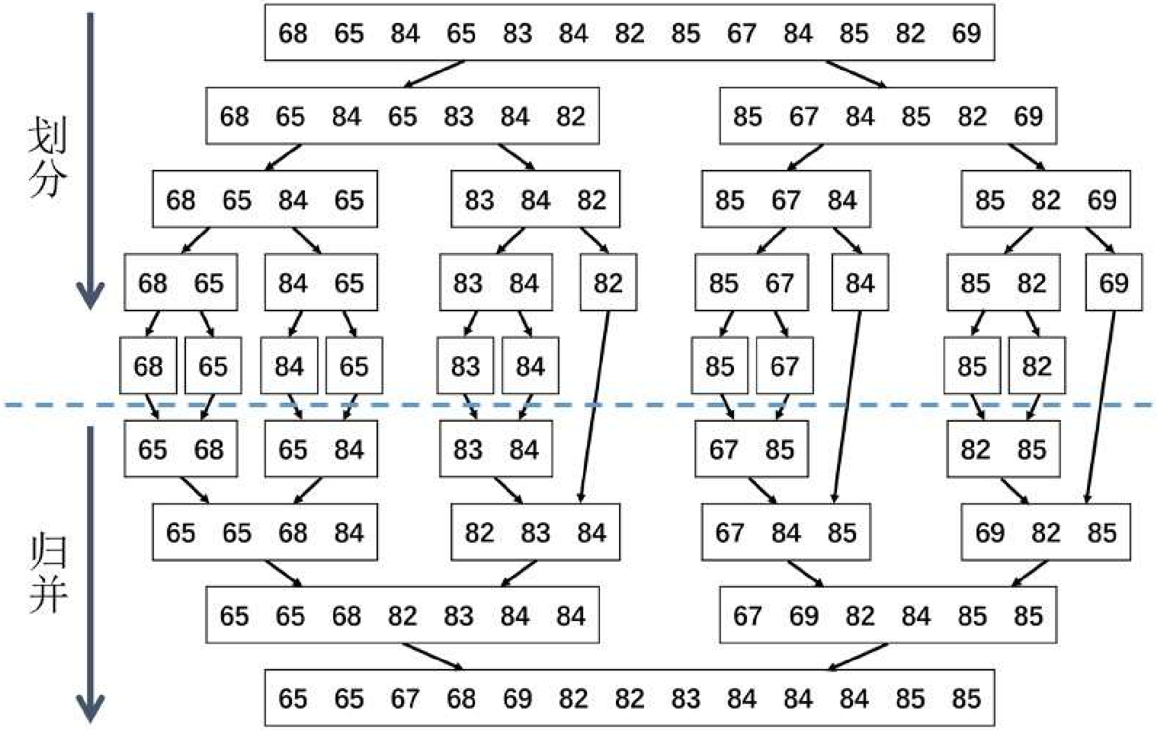
10.5.2 归并排序

利用上述归并思想和二路归并算法来实现的排序算法称为归并排序。

- 算法首先将序列拆分，直到被拆分的子序列长度为1，此时子序列显然有序。
- 然后，使用上述二路归并算法，将有序的短序列依次合并为长序列，最终完成序列的排序。

据归并时对子序列划分方式的不同，归并排序算法又可分为**自顶向下**和**自底向上**两种。

自顶向下归并排序示例


划分
归并
68 65 84 65 83 84 82 85 67 84 85 82 69
68 65 84 65 83 84 82
85 67 84 85 82 69
68 65 84 65
83 84 82
85 67 84
85 82 69
68 65
84 65
83 84
82
85 67
84
85 82
69
68
65
84
65
83
84
85
67
85
82
65 68
65 84
83 84
67 85
82 85
65 65 68 84
82 83 84
67 84 85
69 82 85
65 65 68 82 83 84 84
67 69 82 84 85 85
65 65 67 68 69 82 82 83 84 84 84 85 85

自顶向下归并排序伪代码

算法10-9 归并排序 $\mathrm{MergeSort}(a, l, r)$

输入： 序列a，左端点下标l，右端点下标r

输出： 调整$a_l, a_{l+1}, \dots, a_r$元素顺序，使元素按照非递减顺序排列

if $l < r$ **then** //序列中有至少两个元素待排

| $m \leftarrow (l+r)/2$

| $\mathrm{MergeSort}(a, l, m)$

| $\mathrm{MergeSort}(a, m+1, r)$

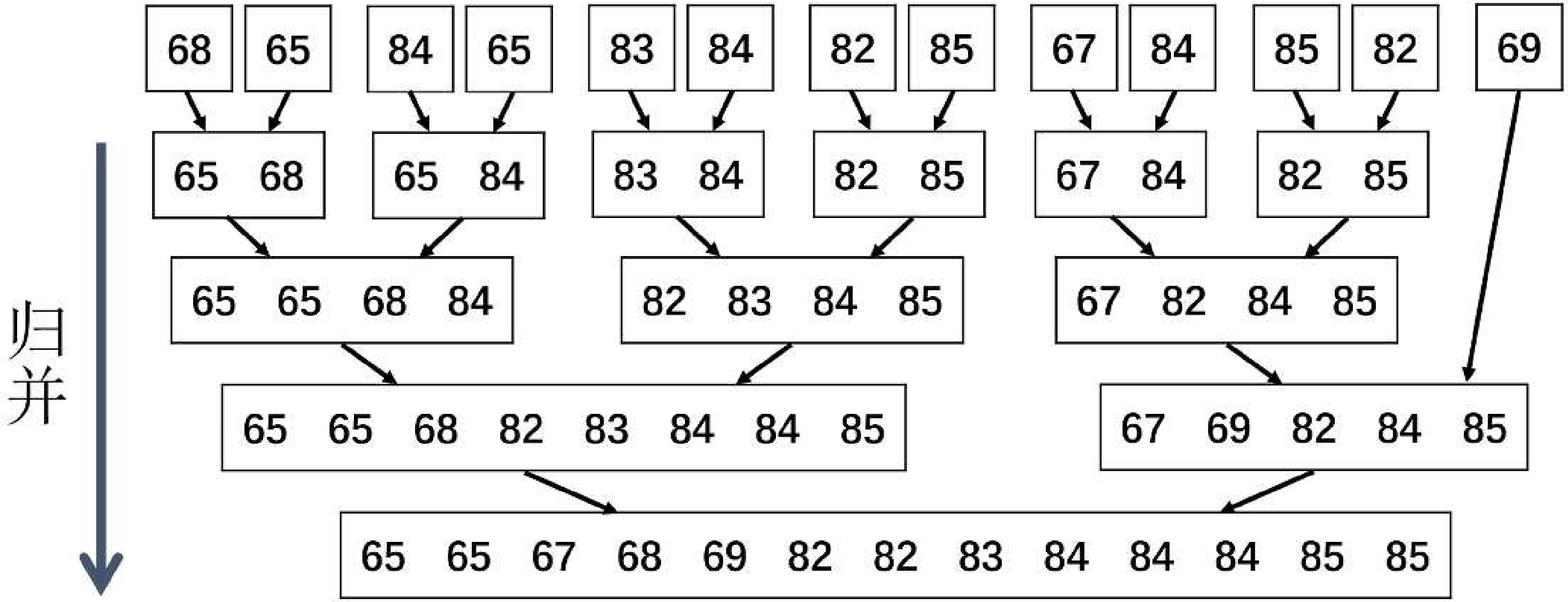
| $\langle a_l, \dots, a_r \rangle \leftarrow \mathrm{TwoWayMerge}(a, l, m, m+1, r)$

end

- 合并后的新序列复制回原序列

高等教育出版社

自底向上归并排序示例


68 65 84 65 83 84 82 85 67 84 85 82 69
65 68
65 84
83 84
82 85
67 84
82 85
65 65 68 84
82 83 84 85
67 82 84 85
65 65 68 82 83 84 84 85
67 69 82 84 85
65 65 67 68 69 82 82 83 84 84 84 85 85
归并

自底向上归并排序伪代码

算法10-9 自底向上归并排序 MergeSortBottomUp(a, l, r)

输入： 序列a，左端点下标l，右端点下标r

输出： 调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
sorted_len ← 1 //当前有序子列长度
n ← r-l+1 //待排元素个数，即序列长度
while sorted_len < n do //当前有序子列长度小于序列长度，则相邻两子序列归并
| lx ← 1 //左子序列从最左端开始
| while lx ≤ r-sorted_len do
| | rx ← lx+sorted_len-1  //左子序列的右端点
| | ly ← rx+1   //右子序列的左端点
| | ry ← Min(ly+sorted_len-1, r)  //右子序列的右端点
| | ⟨a_lx, …, a_ry⟩ ← TwoWayMerge(a, lx, rx, ly, ry)  //归并
| | lx ← ry+1  //下一对子序列的左子序列的左端点
| end
| sorted_len ← sorted_len × 2  //有序子列长度加倍
end
```

归并排序性能分析

由于二路归并算法的复杂度为两个序列长度之和$O(n+m)$，因此我们分析每个元素至多会被二路归并算法调用几次。

自顶向下：

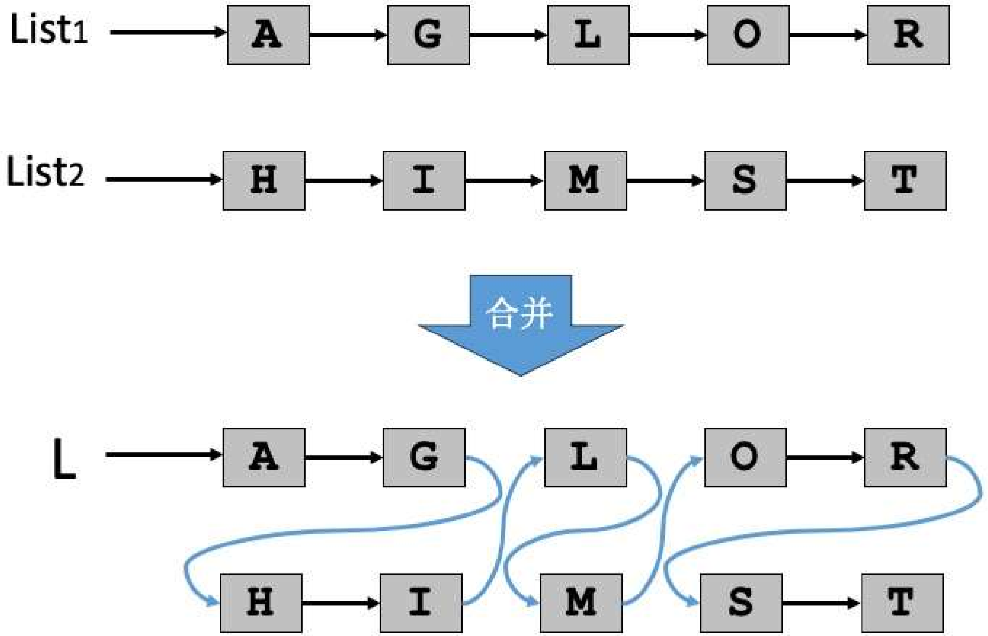
递归的深度每多一层，其待排序序列的长度就**减半**；同时，对于任意深度相同的递归调用，它们所覆盖的元素不相交。因此，递归的最大深度为$\lceil \log(n) \rceil$，且每层递归最多覆盖n个元素，其时间复杂度为$O(n \log n)$。

自底向上：

其外循环的最大循环次数为$\lceil \log(n) \rceil$，内循环将每个元素进行二路归并最多一次，因此复杂度也为$O(n \log n)$。

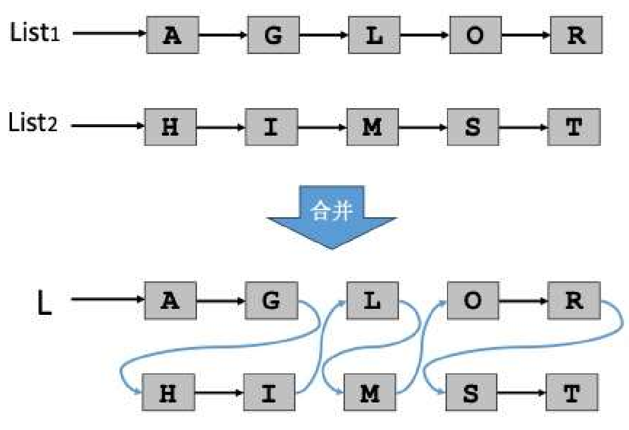
二路归并的应用：合并有序单链表

- 两个有序序列用单链表存放，将两个链表合并为一个有序单链表



二路归并的应用：合并有序单链表

- 两个有序序列用单链表存放，将两个链表合并为一个有序单链表



- 时间复杂度：O(|List1|+|List2|)
- 空间复杂度：O(1)

```
p1 ← List1
p2 ← List2
while p2 ≠ NIL do    //将List2中的所有结点归并至List1
|  pre ← NIL             //pre的初始化
|  while p1 ≠ NIL 且 p1.data ≤ p2.data do  //遍历List1
|  |  pre ← p1
|  |  p1 ← p1.next
|  end         //找到大于p2的结点或者到达List1末尾
|  tmp ← p2          //取出p2的头结点
|  p2 ← p2.next
|  tmp.next ← p1   //将tmp插在p1前面
|  if pre = NIL then    //pre为空表示 p1=List1 (?)
|  |  List1 ← tmp  //tmp成为List1的新头结点
|  else
|  |  pre.next ← tmp      //tmp连在pre后面
|  end
|  p1 ← tmp        //p1指向新插入的结点
end
return List1
```

投票 最多可选1项

合并两个顺序表，通常需要使用与两个顺序表的总长度相同大小的额外空间，而合并单链表不需要其它存储空间，主要的原因是什么？

A 在顺序表中插入或删除元素需要移动大量的数据

B 顺序表的长度固定不变

经典面试题

问题描述

有两个按升序排列的序列，长度分别是n和m且各自存储在顺序表中。设第一个顺序表的容量为n+m，如下所示。要求将两个序列合并在第一个顺序表中，时间复杂度为O(n+m)，空间复杂度O(1)。

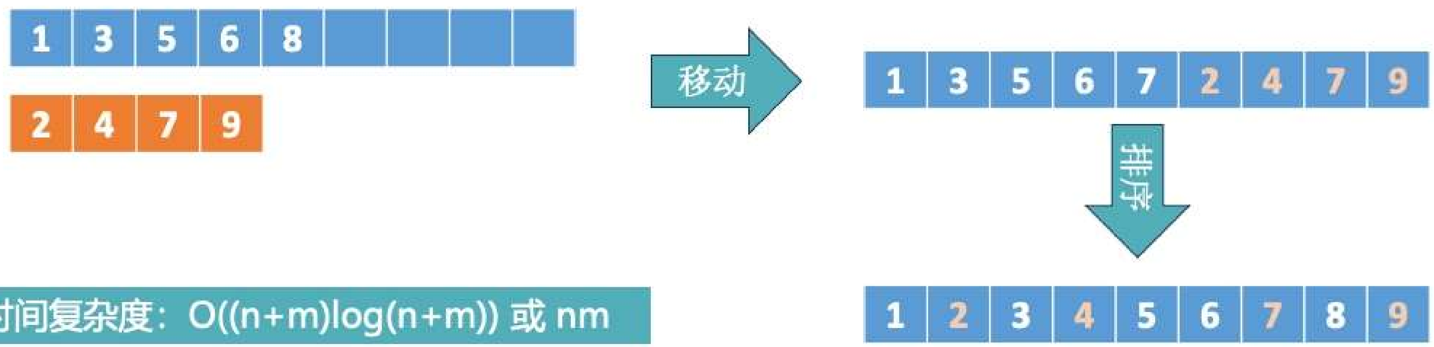
高等教育出版社

经典面试题

问题描述

有两个按升序排列的序列，长度分别是n和m且各自存储在顺序表中。设第一个顺序表的容量为n+m，如下所示。要求将两个序列合并在第一个顺序表中，时间复杂度为O(n+m)，空间复杂度O(1)。

算法1：将第二个顺序表的元素先移到第一个顺序表后段，然后排序！



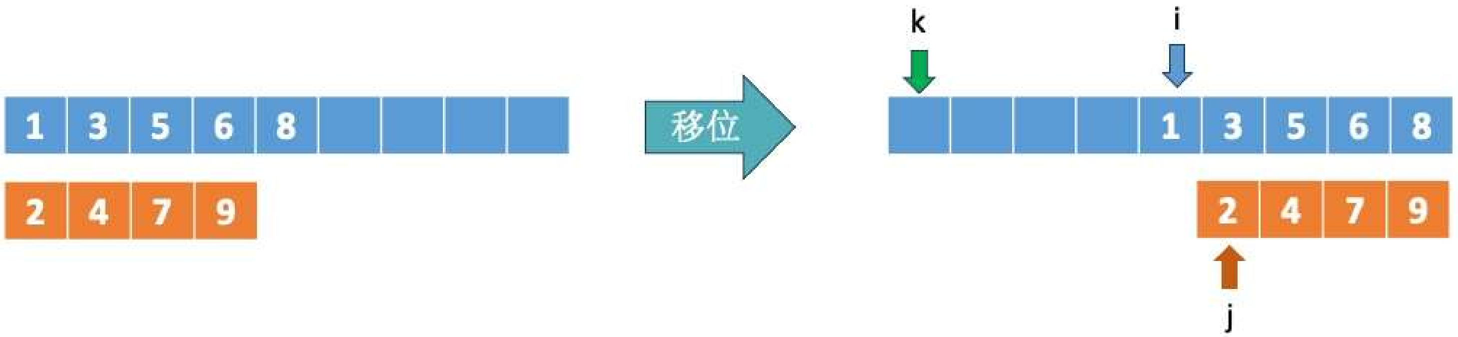
- 时间复杂度：O((n+m)log(n+m)) 或 nm

经典面试题

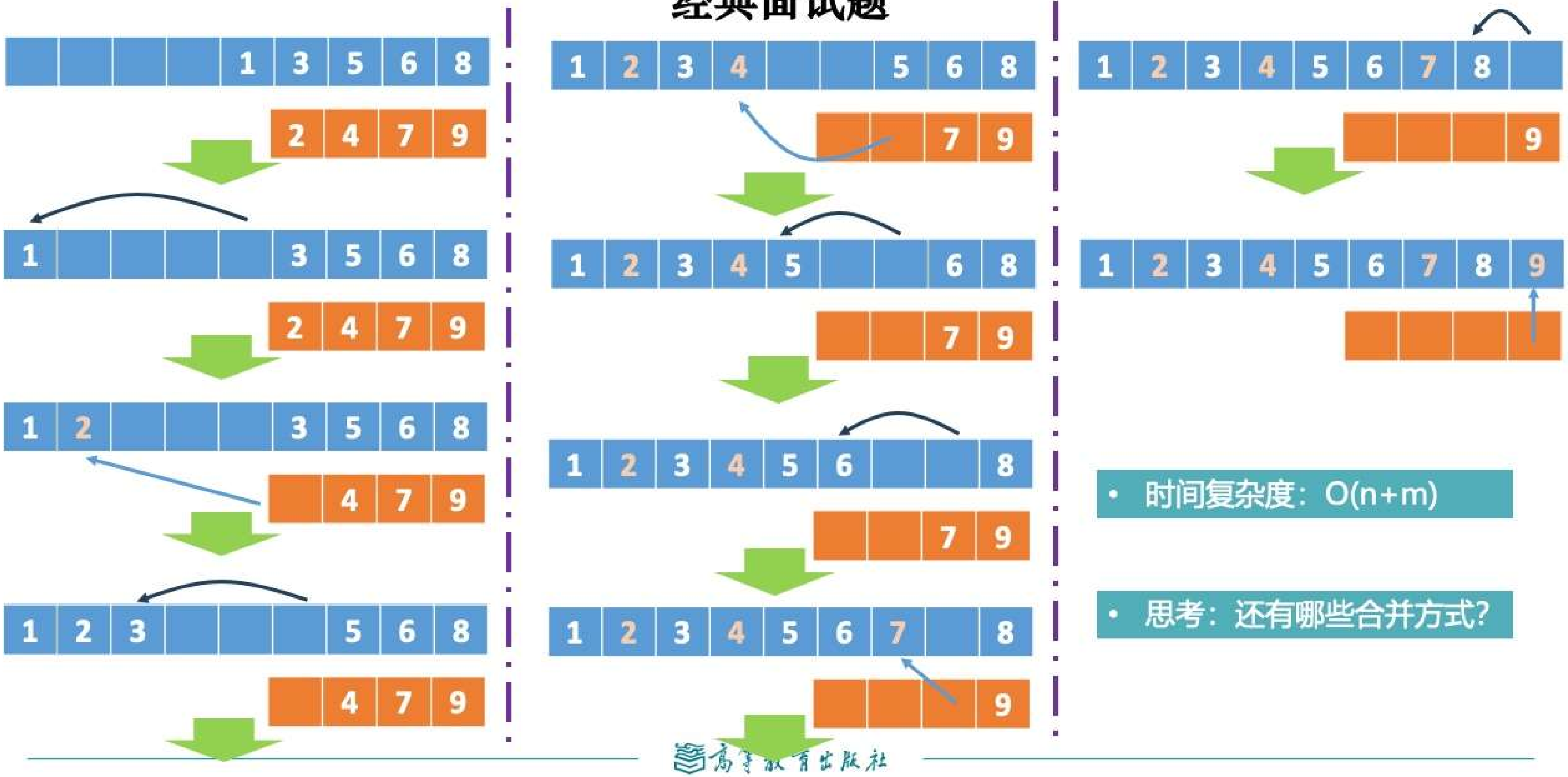
问题描述

有两个按升序排列的序列，长度分别是n和m且各自存储在顺序表中。设第一个顺序表的容量为n+m，如下所示。要求将两个序列合并在第一个顺序表中，时间复杂度为O(n+m)，空间复杂度O(1)。

算法2：先将第一个顺序表的所有元素移到右端，然后合并两个序列



经典面试题



- 时间复杂度：O(n+m)
- 思考：还有哪些合并方式?

归并排序的改进

从二路归并的伪代码可以看到，需要使用一个临时数组来存储归并结果，并在归并排序完成后将结果拷贝回原数组，共进行$2n + 2m$次拷贝（归并次数x2）。在序列元素占用内存较大时，拷贝可能花费大量时间。实际上，可以通过互换当前数组和临时数组的技巧来**减少一半**的拷贝时间。

以自底向上的归并排序为例，当外循环进行**奇数次**时，a作为有序子序列，t作为存放合并结果的临时序列；当外循环进行**偶数次**时，t作为有序子序列，a作为存放合并结果的临时序列。

改进后二路归并的伪代码

算法10-11 改进二路归并 TwoWayMergeImproved$(a, t, l_x, r_x, l_y, r_y)$

输入： 序列a及其有序子序列x和y的下标范围l_x, r_x和l_y, r_y，辅助序列t

输出： 将两个有序序列合并至序列t

$i \leftarrow l_x$ //左子序列当前待比较的元素位置
$j \leftarrow l_y + 1$ //右子序列当前待比较的元素位置
$k \leftarrow l_x$ //结果序列当前待放入的元素位置
while $i \le r_x$ or $j \le r_y$ **do**
| **if** $j > r_y$ or $(i \le r_x$ and $a_i \le a_j)$ **then**
| | $t_k \leftarrow a_i$
| | $i \leftarrow i + 1$
| **else**
| | $t_k \leftarrow a_j$
| | $j \leftarrow j + 1$
| **end**
| $k \leftarrow k + 1$
end

改进后自底向上归并排序的伪代码

算法10-12 改进自底向上归并排序 MergeSortBottomUp(a, l, r)
输入： 序列a，左端点下标l，右端点下标r
输出： 调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1.  sorted_len ← 1 //当前有序子列长度
2.  n ← r-l+1 //待排元素个数，即序列长度
3.  count ← 0
4.  InitList(t)        //辅组序列的初始化
5.  while sorted_len < n do //当前有序子列长度小于序列
    长度，则相邻两子序列归并
6.  | count ← count + 1
7.  | l_x ← 1 //左子序列从最左端开始
8.  | while l_x ≤ r-sorted_len do
9.  | | r_x ← l_x+sorted_len-1  //左子序列的右端点
10. | | l_y ← r_x + 1 //右子序列的左端点
11. | | r_y ← Min(r_x+sorted_len, r)  //右子序列的右端点
12. | | if count%2=1 then
13. | | | TwoWayMergeImproved(a, t, l_x, r_x, l_y, r_y)  //a并入t
14. | | else
15. | | | TwoWayMergeImproved(t, a, l_x, r_x, l_y, r_y)  //t并入a
16. | | end
17. | | l_x ← r_y+1  //下一对子序列的左子序列的左端点
18. | end
19. | sorted_len ← sorted_len × 2  //有序子列长度加倍
20. end
```

归并排序的经典应用：求逆序对数量

归并排序的一个经典应用是求逆序对数量。

在一个序列中，两个元素a_i和a_j逆序指它们满足$i < j$和$a_i > a_j$，称这两个元素为一个**逆序对**。求一个序列的逆序对数量则是找到序列中有多少组不同的i和j，满足a_i和a_j是逆序对。

显然，当序列有序时，序列的逆序对数量为零。一种最简单的方法是枚举所有可能的i和j，如果逆序则答案加一。这种算法的时间复杂度为$\boldsymbol{O(n^2)}$。

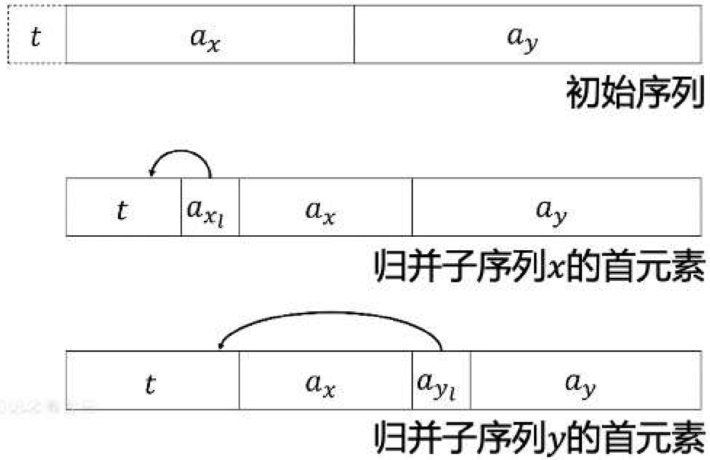
基于归并排序算法思想，可以在$\boldsymbol{O(n \log n)}$的时间里找到序列的逆序对数量。算法的核心思想在于：统计两个子序列进行二路归时序列逆序对数量**减少了多少**。

归并排序的经典应用:求逆序对数量

考虑二路归并时，输入有序子序列x,y的区间分别为l_x,r_x和l_y,r_y。

由于在归并排序中待归并的子序列满足**首尾相连**，规定$l_x \le r_x < l_y \le r_y$，就有$r_x + 1 = l_y$。

为了便于理解，可以将辅助序列t看作拼接于待排序子序列前，且每次将元素插入t末尾的操作看成将元素从原序列首移动至序列末尾，如图所示。

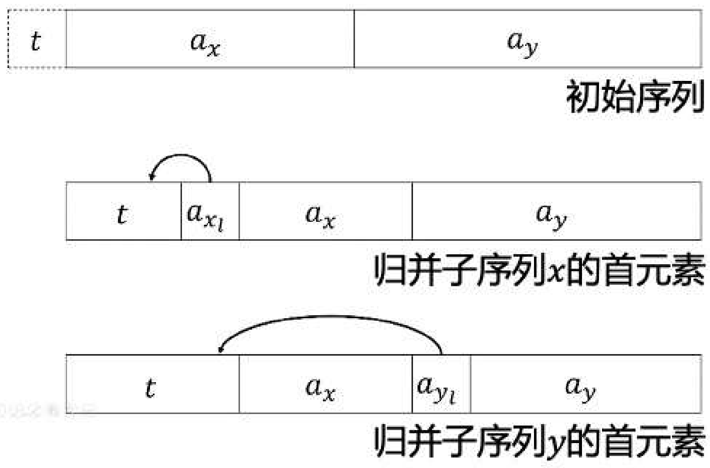


归并排序的经典应用:求逆序对数量

- 初始时，辅助序列t中不包含任何元素。
- 若将x的a_{l_x}移至t末尾，相当于序列t和子序列x的分界符l_x右移，序列元素顺序**没有发生改变**。
- 若将y的a_{l_y}移至t末尾，相当于将该元素使用插入排序插至l_x处，此时序列中元素间**两两顺序**关系发生改变的只有a_{l_y}**和子序列x剩余元素**之间。
- 由于二路归并每次选择的元素是子序列剩余元素中**最小的**，逆序对数量减少了x目前**剩余的元素数量**。
- 最后，二路归并这两个子序列时，如果一个逆序对的两个元素未被两个子序列完全包含，则二路归并后它们**仍然为逆序对**。

因此，通过二路归并，将两个子序列合并为有序序列后，求出了逆序对**减少的数量**，同时**没有影响**其它未被两个子序列完全包含的逆序对数量。

算法时间复杂度和归并排序相同，为$O(n\log n)$。



高等教育出版社

二路归并求逆序对减量的伪代码

算法10-13 二路归并求逆序对减量 $\mathrm{TwoWayInversionCount}(a, t, l, m, r)$

输入： 序列a，相邻两个有序子序列范围l, m和$m+1, r$，辅组序列t

输出： 将两个有序序列合并，并返回减少的逆序对数量

```
i ← l
j ← m + 1
k ← l
count ← 0    //记录逆序对数量
while i ≤ m or j ≤ r do
| if j > r or (i ≤ m and a_i ≤ a_j) then
| | t_k ← a_i
| | i ← i + 1
| else   //a_i > a_j
| | t_k ← a_j
| | j ← j + 1
| | count ← count + (m − i + 1)  //减少的逆序对
| end
| k ← k + 1
end
for i ← l to r do
| a_i ← t_i   //将合并好的序列复制回序列a
end
return count
```

- 由于函数返回的是逆序对数量，需要在函数内将合并好的序列复制给序列a
- 时间复杂度：O(r-l+1)

高等教育出版社

归并排序兼求逆序对数量的伪代码

算法10-14 归并排序兼求逆序对数量 $\mathrm{InversionCount}(a, t, l, r)$
输入： 序列a，左端点下标l，右端点下标r，辅组序列t
输出： 调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列，同时返回序列中逆序对数量

$count \leftarrow 0$
if $l < r$ **then** //序列中至少有2个元素时才执行
| $m \leftarrow (l + r)/2$
| $count \leftarrow count + \mathrm{InversionCount}(a, t, l, m)$
| $count \leftarrow count + \mathrm{InversionCount}(a, t, m + 1, r)$
| $count \leftarrow count + \mathrm{TwoWayInversionCount}(a, t, l, m, r)$
end
return count

二路归并求逆序对算法的应用：翻转对(LeetCode493)

问题描述： 给定一个数组 a，如果 i < j 且 a[i] > 2a[j]，就将 (i, j) 称作一个**翻转对**。

输入： 数组a，长度不超过10^5

输出： 返回给定数组中的翻转对数量

例：输入 [1,3,2,4,1]，**输出** 2

--翻转对(3,1), (4,1)

思路：改写求逆序对函数 $\mathrm{TwoWayInversionCount}(a, t, l, m, r)$

- 时间复杂度：O(n log(n))

```
i ← l
j ← m + 1
p ← l  //指向翻转对的开始位置
k ← 0
count ← 0   //记录翻转对数量
while i ≤ m or j ≤ r do
| if j > r or (i ≤ m and a[i] ≤ a[j]) then
| | t[k] ← a[i]
| | i ← i + 1
| else    // a[i] > a[j]
| | t[k] ← a[j]
| | while p ≤ m and a[p] ≤ 2 * a[j] do
| | | p ← p + 1
| | end    //查找a[j]的最小翻转对
| | j ← j + 1
| | count ← count + (m − p + 1)
| end
| k ← k + 1
end
for i ← l to r do
| a[i] ← t[i]
end
return count
```

高等教育出版社

10.6 基于比较排序的复杂度分析

核心问题：是否存在时间复杂度优于 $O(n\log n)$ 的基于比较排序的算法?

- 基于比较的排序的复杂度下界
- 基于比较的排序的平均复杂度
- 最少比较排序

10.6.1 基于比较的排序的复杂度下界

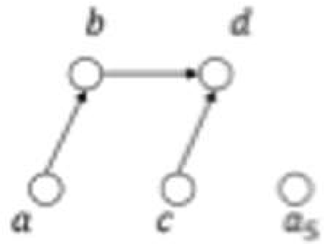
核心问题：每次操作可以询问两个元素 a_i 是否大于 a_j，对于任意的输入数组，在最坏情况下，至少需要进行多少次操作才能确定输入数组中所有元素的次序。

理论分析：对于每次询问操作，都可以将剩下可能的次序分成“满足条件”和“不满足条件”两类，在最坏情况下，我们每次最多只能排除一半的可能次序。对于 $n!$ 种可能次序，至少需要进行 $O(\log(n!))$ 次操作。

$$\log(n!) = \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \frac{1}{12n} + \frac{1}{288n^2} + \cdots\right)\right) \sim O(n\log n)$$

10.6.3 最少比较排序

示例（$n = 5$）：7 次比较。先比较 a_1、a_2 和 a_3、a_4，再将两者的较大者进行比较，得到如下图所示次序图，其中 $a \to b$ 表示 $a < b$。再通过最多两次比较将 a_5 插入到 a、b、d 中的适当位置。继续通过最多两次比较将 c 插入到 a、b、d、a_5 中的适当位置。一共进行最多七次比较完成排序。



思考：当 $n = 6$ 时，如何设计一个最小比较排序算法[1]。

[1] Ford L R , Johnson S M .A Tournament Problem[J].American Mathematical Monthly, 1959, 66(5):387-389.DOI:10.2307/2308750.