排序技术 2

Tan Yiqing

2025 年 12 月 22 日

1 交换排序 (续)

1.1 快速排序 (续)

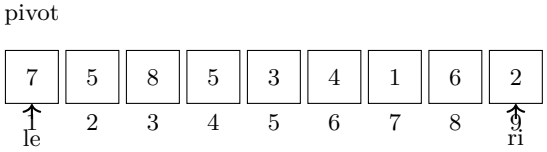
1.1.1 快速排序的代码流程

笔者觉得快排比较重要而且相对比较复杂，所以这里给出代码流程说明。

Listing 1: 快速排序的一次划分过程

```
template <class T>
int quickPivot(T* &arr, int n, int sIndex, int eIndex){
    //use first element as pivot, put pivot in correct position, return index
    int le = sIndex;
    int ri = eIndex;
    while(le < ri){
        // 1) 先从右往左找：寻找一个“比左侧当前元素 arr[le] 小”的元素
        while(le < ri && arr[le] <= arr[ri]) ri--;
        //把“小元素”放到左侧区域
        if(le < ri){
            T tmp = arr[le];
            arr[le] = arr[ri];
            arr[ri] = tmp;
            le++;     //左侧“已处理区”扩大
        }
        // 2) 再从左往右找：寻找一个“比右侧当前元素 arr[ri] 大”的元素
        while(le < ri && arr[le] <= arr[ri]) le++;
        //把“大元素”放到右侧区域
        if(le < ri){
            T tmp = arr[le];
            arr[le] = arr[ri];
            arr[ri] = tmp;
            ri--;     //右侧“已处理区”扩大
        }
    }
    return le;
}
```

例子演示： 以数组 $[7,5,8,5,3,4,1,6,2]$ 为例，演示一次 quickPivot 划分过程（枢轴为第一个元素 7）。



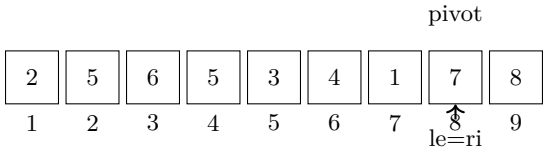
初始状态：le=1，ri=9，pivot=7。

1. 从右往左找，直到 $arr[ri] < arr[le]$，找到 $arr[9] = 2 < 7$，交换 $arr[1]$ 和 $arr[9]$，数组变为 $[2,5,8,5,3,4,1,6,7]$，le=2。
2. 从左往右找，直到 $arr[le] > arr[ri]$，le 依次移动到 $arr[3] = 8 > 7$，交换 $arr[3]$ 和 $arr[9]$，数组变为 $[2,5,7,5,3,4,1,6,8]$，ri=8。

3. 从右往左找，直到 $arr[ri] = 6 < 7$，交换 $arr[3]$ 和 $arr[8]$，数组变为 $[2, 5, 6, 5, 3, 4, 1, 7, 8]$，le=4。

4. 从左往右找，le 移动到 $arr[7] = 1 < 7$，le=8。

5. le=ri=8，划分结束，pivot 7 已到正确位置。



最终状态：pivot 7 已到正确位置（第 8 位）。

1.1.2 快速排序的性能分析

时间复杂度 分析如下：

1. **最好情况**：每一次划分对一个记录定位后，该记录的左侧子表与右侧子表的长度相同，为 $O(n \log_2 n)$。

$$T(n) \leq 2T\left(\frac{n}{2}\right) + n$$

展开递推式如下：

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + n \\ &\leq 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 4T\left(\frac{n}{4}\right) + 2n \\ &\leq 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n = 8T\left(\frac{n}{8}\right) + 3n \\ &\cdots \\ &\leq nT(1) + n\log_2 n = O(n\log_2 n) \end{aligned}$$

上述递推式表示：每一层递归都将问题规模减半，并且每层需要 $O(n)$ 的比较和移动操作。递归的层数为 $\log_2 n$，每层总操作量为 n，因此总的时间复杂度为 $O(n \log_2 n)$。

2. **最坏情况**：每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列为空），为 $O(n^2)$。

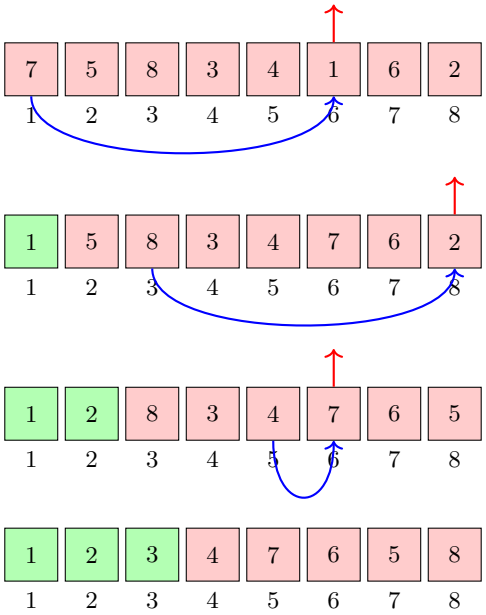
$$\Sigma_{i=1}^{n-1} n - i = \frac{n(n-1)}{2} = O(n^2)$$

一个反直觉的事情是最坏情况的典型例子是正序和反序！

3. **平均情况**：时间复杂度为 $O(n \log_2 n)$。

2 选择排序 (selection sort)

选择排序的主要操作是选择，其主要思想是：每趟排序在当前待排序序列中选出关键码最小的记录，添加到有序序列中。



2.1 性能分析

时间复杂度 分析如下：

1. 最好情况：

   (a) 比较次数：$\frac{n(n-1)}{2}$ 次。

   (b) 移动次数：0 次。

   时间复杂度为 $O(n^2)$。

2. 最坏情况：

   (a) 比较次数：$\frac{n(n-1)}{2}$ 次。

   (b) 移动次数：$3(n-1)$ 次。

   时间复杂度为 $O(n^2)$。

3. 平均情况：时间复杂度为 $O(n^2)$。

空间复杂度 需要一个辅助空间存放临时变量，空间复杂度为 $O(1)$。

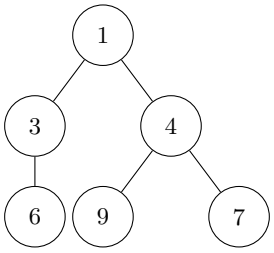
稳定性 选择排序是**不稳定**的排序算法! 反例：对 [(f,1),(b,2),(c,3),(f,4),(a,1)] 进行选择排序，第一次选择后得到 [(a,1),(b,2),(c,3),(f,4),(f,1)]，两个 (f,1) 和 (f,4) 的相对位置发生了改变。

2.2 改进方法

关键在于减少关键码的比较次数，可以采用**堆排序**来实现选择排序的改进。

2.2.1 堆的定义

堆是具有下列性质的完全二叉树：每个结点的值都小于或等于其左右孩子结点的值（称为小根堆），或每个结点的值都大于或等于其左右孩子结点的值（称为大根堆）。



小根堆的根结点是所有结点的最小者。较小结点靠近根结点，但不绝对。(大根堆反之)

堆采用顺序结构存储，比如上面可以用层序存储为数组 $[1,3,4,6,9,7]$。

2.2.2 基本思想与代码实现

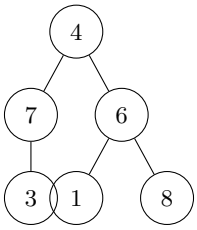
首先将待排序的记录序列构造成一个堆，此时，选出了堆中所有记录的**最大者**，然后将它从堆中移走，并将剩余的记录再调整成堆，这样又找出了**次小的记录**，以此类推，直到堆中只有一个记录。

所以，问题来到如何**构造堆**和**调整堆**。

堆构造

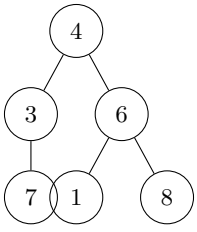
1. 从最后一个非叶节点开始，依次向上对每个节点进行“向下调整”。
2. 每次调整时，将当前节点与其左右孩子中较小者比较，若孩子更小则交换，并递归对子节点继续调整。
3. 重复上述过程，直到根节点也调整完毕，整个序列就变成了一个堆。

假设初始无序序列为 $[4,7,6,3,1,8]$，其完全二叉树结构如下：



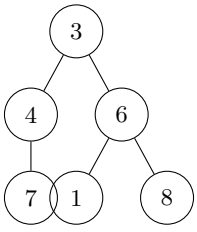
步骤 1：从最后一个非叶节点（7）开始向下调整。

7 的左孩子为 3，比 7 小，交换 7 和 3。



步骤 2：向上回到根节点（4），向下调整。

4 的左孩子为 3，右孩子为 6，3 最小，交换 4 和 3。

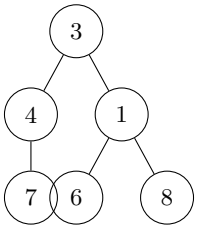


步骤 3：继续调整 4 的子树。

4 的左孩子为 7，右孩子为空，无需调整。

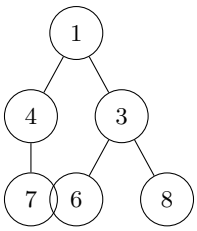
步骤 4：调整右子树（6）。

6 的左孩子为 1，比 6 小，交换 6 和 1。



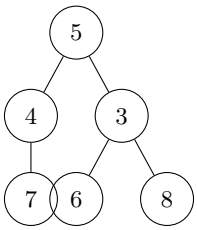
步骤 5：回到根节点，检查是否需要继续调整。

根节点 3 的左右孩子为 4 和 1，1 最小，交换 3 和 1。

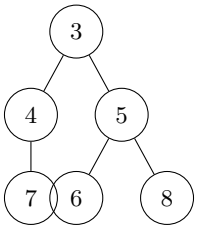


最终得到小根堆结构。

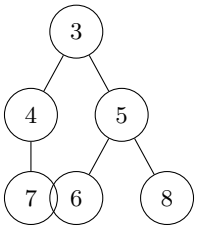
堆调整 在一棵完全二叉树中，根结点的左右子树均是堆，如何调整根结点，使整个完全二叉树成为一个堆？



此时，根节点 5 大于右孩子 3，需要与 3 交换，调整如下：



继续对 5 进行向下调整。5 的左孩子为 6，右孩子为 8，5 已经小于它们，无需再调整，最终得到小根堆：



总结：只需将根节点与左右子树中较小的根交换，并递归对子树调整，即可使整棵树成为堆。

以上过程都是在二叉树上的可视化，如何在顺序结构上实现呢？先给出如下的对应关系：

1. 堆构建：

   for (i=n/2; i>=1; i–) sift(r, i, n) ;

2. 堆调整：
   sift(r, i, n):
3. 处理堆顶记录：
   swap(r[1], r[n]) ;

Listing 2: 堆排序的顺序结构实现

```
/* 向下调整，使以i为根的子树成为小根堆 */
void sift(int r[], int i, int n) {
    int tmp = r[i];
    int child = 2 * i;
    while (child <= n) {
        // 选出左右孩子中较小的
        if (child + 1 <= n && r[child + 1] < r[child]) {
            child++;
        }
        if (r[child] < tmp) {
            r[i] = r[child];
            i = child;
            child = 2 * i;
        } else {
            break;
        }
    }
    r[i] = tmp;
}

/* 堆排序主过程 */
void heapSort(int r[], int n) {
    // 1. 堆构建（下标从1开始）
    for (int i = n / 2; i >= 1; i--) {
        sift(r, i, n);
    }
    // 2. 反复取堆顶元素放到末尾，并调整堆
    for (int i = n; i > 1; i--) {
        swap(r[1], r[i]); // 处理堆顶记录
        sift(r, 1, i - 1); // 堆调整
    }
}
```

2.2.3 堆排序性能分析

时间复杂度 分析如下：

1. 第一个 for 是堆构造：$O(n)$。
2. 第二个 for 是输出堆顶重建堆，共需要取 n-1 次堆顶记录，第 i 次取堆顶记录重建堆需要 $O(\log_2 i)$ 时间，所以需要 $O(n \log_2 n)$ 时间；

因此整个堆排序的时间复杂度为 $O(n \log_2 n)$，这是堆排序的最好、最坏和平均的时间代价。

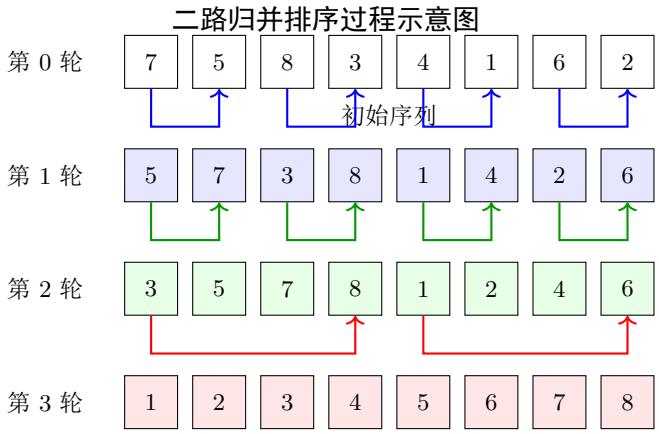
3 归并排序 (mergesort)

归并是指将两个或两个以上的有序序列合并成一个有序序列的过程。

归并排序的主要操作是归并，其主要思想是：将若干有序序列逐步归并，最终得到一个有序序列。

3.1 二路归并排序

将一个具有 n 个待排序记录的序列看成是 n 个长度为 1 的有序序列，然后进行两两归并，得到 n/2 个长度为 2 的有序序列，再进行两两归并，得到 n/4 个长度为 4 的有序序列，以此类推，直至得到一个长度为 n 的有序序列为止。



在归并过程中，可能会破坏原来的有序序列，所以，将归并的结果存入另外一个数组中。

3.2 伪代码

Listing 3: 二路归并排序伪代码

```
void MergeSort2(int r[ ], int r1[ ], int s, int t)
{
    if (s==t) r1[s]=r[s];                    //递归出口，用有序长度来控制
    else {
        m=(s+t)/2;
        MergeSort2(r, r1, s, m);              //归并排序前半个子序列
        MergeSort2(r, r1, m+1, t);            //归并排序后半个子序列
        Merge(r1, r, s, m, t);                将两个已排序的子序列归并
    }
}
```

3.3 性能分析

1. 时间复杂度：一趟归并操作是将 r[1] r[n] 中相邻的长度为 h 的有序序列进行两两归并，并把结果存放到 r1[1] r1[n] 中，这需要 $O(n)$ 时间。整个归并排序需要进行趟 $\lceil \log_2 n \rceil$，因此，总的时间代价是 $O(n \log_2 n)$。这是归并排序算法的最好、最坏、平均的时间性能。

2. 空间复杂度：算法在执行时，需要占用与原始记录序列同样数量的存储空间，因此空间复杂度为 $O(n)$。

4 分配排序 (distribution sort)

桶是指一个用于临时存放数据的容器。

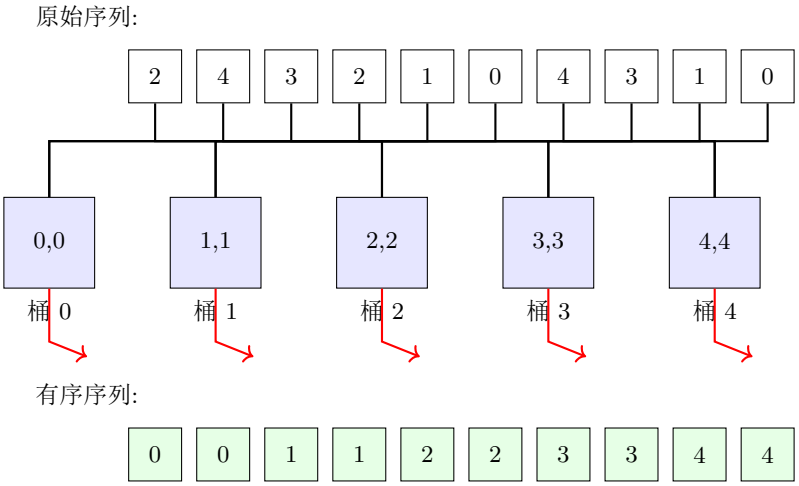
分配排序是基于分配和收集的排序方法，其基本思想是：先将待排序记录序列分配到不同的桶里，然后再把各桶中的记录依次收集到一起。

4.1 桶排序 (bucket sort)

假设待排序记录的值都在 0～m-1 之间，设置 m 个桶，首先将值为 i 的记录分配到第 i 个桶中，然后再将各个桶中的记录依次收集起来。

实际写的使用，用静态链队列来实现每个桶是个不错的选择。

桶排序过程示意图



4.1.1 桶排序性能分析

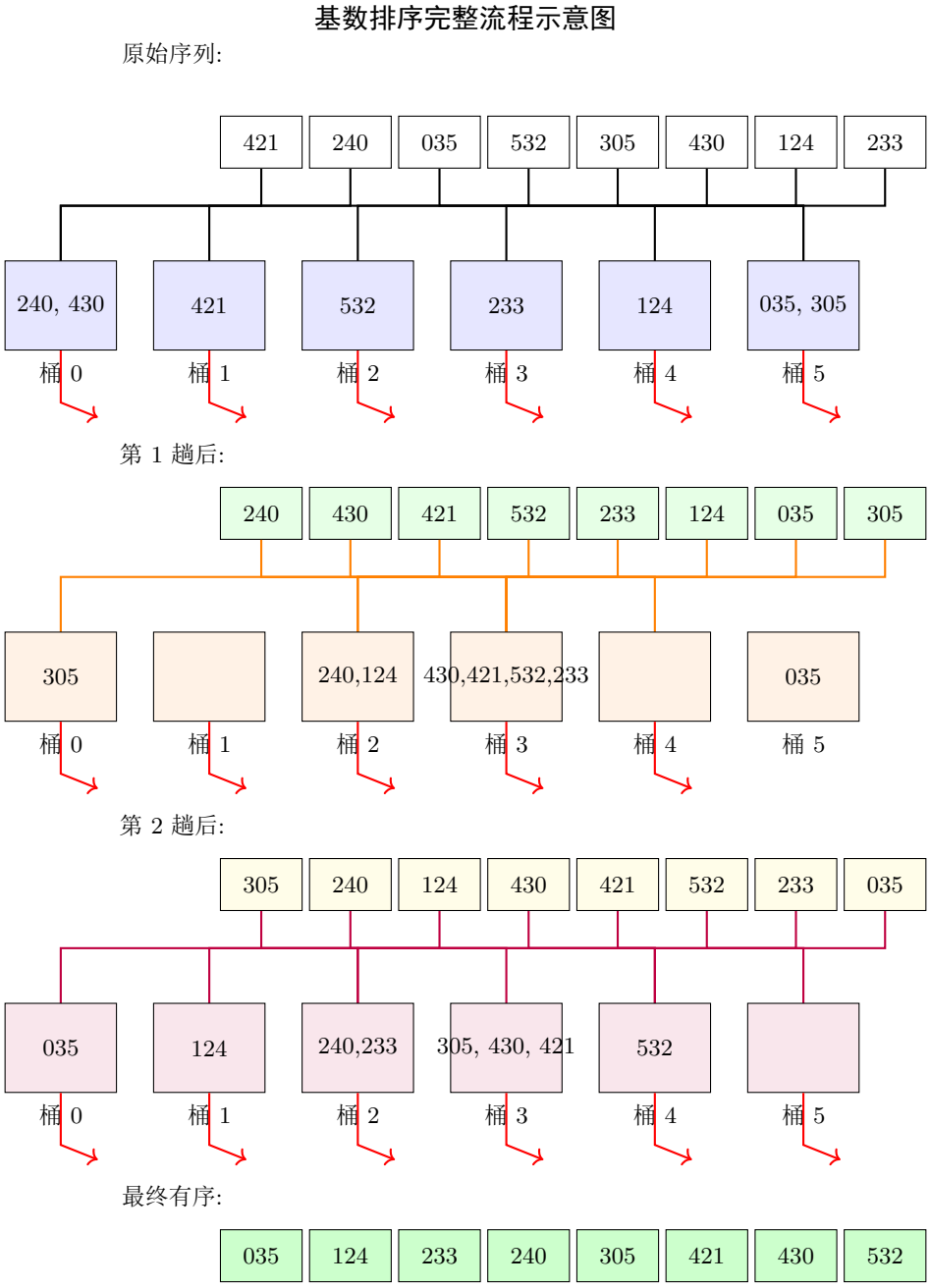
1. **时间复杂度**：假定用一个静态链表存储所有桶，用静态链队列实现桶。桶式排序第一个循环初始化静态链表，时间性能为 $O(n)$，第二个循环初始化静态链队列，时间性能为 $O(m)$，进行分配需要遍历静态链表，时间性能为 $O(n)$，进行收集需要遍历静态链表和静态链队列，因此，桶式排序的时间复杂度为 $O(n+m)$。
2. **空间复杂度**：桶式排序的空间复杂度是 $O(m)$，用来存储 m 个静态队列表示的桶。
3. **稳定性**：桶排序是**稳定**的排序算法。

4.2 基数排序 (radix sort)

基本思想是：将关键码看成由若干个子关键码复合而成，然后借助分配和收集操作。具体来说：

1. 第 1 趟排序：按最低位关键码 $kd-1$ 将具有相同码值的记录分配到一个队列（桶）中，然后依次收集起来，得到一个按关键码 $kd-1$ 有序的序列；
2. 第 i 趟排序：按关键码 $kd-i$ 将具有相同码值的记录分配到一个队列（桶）中，然后依次收集起来，得到一个按关键码 $kd-i$ 有序的序列。

上面是 PPT 上的，人话的话可以举个例子：先把个位一样的放一起，再把十位一样的放一起……



如上图所示，基数排序每一趟都按对应位分配到桶中，再收集起来，经过所有关键码位的分配与收集后，最终得到有序序列。

4.2.1 基数排序性能分析

1. **时间复杂度：**假设待排序记录的关键码由 d 个子关键码复合而成，每个子关键码的取值范围为 m 个，则基数排序的时间复杂度为 $O(d(n+m))$，其中每一趟分配的时间复杂度是 $O(n)$，每一趟收集的时间复杂度为 $O(n+m)$，整个排序需要执行 d 趟。
2. **空间复杂度：**基数排序共需要 m 个队列，因此空间复杂度为 $O(m)$。
3. **稳定性：**由于桶采用队列作为存储结构，基数排序是**稳定**的排序算法。

5 总结

排序方法	平均情况	最好情况	最坏情况
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$
希尔排序	$O(n\log_2 n)$~$O(n^2)$	$O(n^{1.3})$?	$O(n^2)$
起泡排序	$O(n^2)$	$O(n)$	$O(n^2)$
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
基数排序	$O(d(n+m))$	$O(d(n+m))$	$O(d(n+m))$

排序方法	辅助空间
直接插入排序	$O(1)$
希尔排序	$O(1)$
起泡排序	$O(1)$
快速排序	$O(\log_2 n)$ ~$O(n)$
简单选择排序	$O(1)$
堆排序	$O(1)$
归并排序	$O(n)$
基数排序	$O(m)$

稳定性比较

所有排序方法可分为两类，

（1）一类是稳定的，包括直接插入排序、起泡排序和归并排序；

（2）另一类是不稳定的，包括希尔排序、简单选择排序、快速排序和堆排序。