layout: post title: "排序算法" data: 2019-06-17 09:54:36 categories: IntroduceAlgorithm(算法导论) tags: 排序 excerpt: 九种内部排序算法 author: Lydia mathjax: true

- content {:toc}

排序算法

给一列元素排序算是算法中一个基础问题。排序算法可分为两种，**内部排序**和**外部排序**，内部排序是所有数据能够一次加入内存中，直接进行排序的算法；外部排序是指数据不能够一次加载到内存中，例如数据量太大等，这时候就需要采取一些办法。当我们讨论一个排序算法时，除了需要讨论**稳定性**，最好**时间复杂度**，最差**时间复杂度**和**空间复杂度**，讨论空间复杂度时，可以得到是否是**原址排序**(如果输入数组中仅有常数个元素需要再排序过程中存储在数组之外，则称排序算法是原址的(in place))，还应该去讨论算法的适用范围和改进点，也就是在什么时候会出现最好时间复杂和最坏时间复杂度以及改进策略，通常算法的最坏事件复杂度发生在待排元素完全逆序的时候。另外，没有一种排序算法是在所有情况下都是最好的，选择合适的排序算法才是最重要的。算法的稳定性并不是一成不变的，例如插入排序中，我们使用严格小于/严格大于时，才能保证算法稳定，当使用小于等于/大于等于时，算法就是不稳定的。

为了方便讨论，我们只讨论基于比较的排序(相对的面是什么？？？)，即所有的元素都是可比较的，并且假定排序的结果是从小到大的。

简单排序算法

这三种排序算法特点是代码简单，容易实现，但是在性能上不是很好，但是 **1)** 可以作为高级排序算法中小规模数据时应用，**2)** 在此基础上改进，例如对希尔排序的子数组应用直接插入排序，堆排序是简单选择排序的优化。

直接插入排序

直接插入排序的主要思想是将元素插入到**合适**的位置，逐步构建有序序列。将元素分成两个部分，前半部分是有序的，后半部分是无序的。每次从后半部分取出一个元素，在有序序列中从后向前扫描**比较**，找到合适的位置**插入**。 可以把插入排序想象成抓牌的过程，通常我们左手拿牌，右手抓牌，每抓到一张牌，从右向左比较牌面大小，如果牌面大于抓到的牌，就将这张牌向右移动一个位置(留出一个空位)，当牌面小于等于抓到的牌时，就将抓到的牌放入。

- 对于一组包含N个元素的非递减有序序列，采用插入排序成非递增序列，比较次数至多是 $\frac{1+N-2}{2} \times (N-2)$ ，移动的次数至多为 $\frac{1+N-2}{2} \times (N-2) = \frac{(N-1)\times(N-2)}{2}$
- 简单插入排序的时间复杂度是$O(N^2)$
- 当待排序列是有序时，取得最好时间复杂度$O(N)$
- 在简单插入算法中，只需要申请临时变量即可，空间复杂度是$O(N)$
- 当保证严格小于/严格大于时，算法是稳定的

对于下标$i < j$,如果$A[i] > A[j]$,，则称(i, j)是一对逆序对(inversion),例如序列$34, 8, 64, 51, 32, 21$中有九个逆序对，插入排序需要做九次交换，交换的次数和逆序对的数目是相同的，交换两个元素正好消去1个逆序对。因此插入排序 $T(N, I) = O(N + I), I$是逆序对的个数,因此，

- **如果序列基本有序，则插入排序简单且高效。**(当然，如果序列完全逆序，此时 I是 N^2 级别)

伪码

```
insertion_sort(A, length)
```

```
for j = 1 to length-1
    key = A[j]
    //将key 插入有序子序列序列A[0...j-1]
    i = j
    while i-1 >= 0 && A[i-1] > key
        A[i] = A[i-1]
        i--
    A[i] = key
```

时间复杂度下界

- 定理：任意N个不同元素组成的序列平均具有 $N(N-1)/4$ 个逆序对。
- 定理：任何仅以交换相邻两元素来排序的算法，其平均时间复杂度为 $\Omega(N^2)$ 。
- 这意味着：
- 要提高算法效率，我们必须每次消去不止1个逆序对！
- 每次交换相隔较远的2个元素！

简单选择排序

简单选择排序就跟它的名字一样，关键在于**选择**合适的元素，逐步构建有序序列。将序列分成两个部分，前半部分是有序的，每次从无序的序列中选取最小元素，追加在有序序列末尾(与无序第一个元素交换)。在寻找最小元素时，需要遍历无序序列$\Theta(N^2)$，成为提高效率的瓶颈。

- 简单选择排序的时间复杂度是$T(N)=\Theta(N^2)$,在找最小值时，无论如何要遍历整个无序序列
- 空间复杂度是 $O(1)$,只需要申请一个临时变量保存当前最小值
- 简单排序算法是不稳定的
- 简单排序算法并不是一种优秀的算法，适用于数据量比较小，并且对稳定性没有要求的情况

伪码

```
selection_sort(A, length)
```

```
for i = 0 to length-1
    min_idx = i
    for j = i+1 to length-1
        if(A[j] < A[min_idx])
            min_idx = j
    A[i] <-> A[min_idx]
```

冒泡排序

冒泡排序是一种直观的排序方法，每次比较相邻的两个元素，如果逆序就将他们顺序交换，一轮冒泡后，最大的元素放在序列尾部，序列右边是排序好的子序列。重复这个过程，直到所有的元素都排好。 如果序列已经是有序的，在这个排序过程中并不能发现，因此我们给每一轮排序加一个标记，如果整轮排序中都没有交换过，说明序列已经排好了，那么停止排序。

- 冒泡排序的最好时间复杂度$O(N)$,序列有序时
- 冒泡排序的最好时间复杂度$O(N^2)$,序列逆序时
- 冒泡排序的空间复杂度是$O(1)$
- 冒泡排序是稳定的
- 冒泡排序交换的次数也是逆序对的数目，因此，冒泡排序适用于序列基本有序的情况

伪码

```
bubble_sort(A, length)
```

```
for i = N-1 to 0
    flag = 0 //标识是否发生交换
    for j = 0 to i-1
        if(A[j] > A[j+1])
            A[j] <-> A[j+1]
            flag = 1
    if flag == 0
        break   //全程无交换
```

高级排序算法

希尔排序

希尔排序是对直接插入排序的一种优化，实质是将直接插入排序变成了分组插入排序。其基本思想就是将待排元素按照步长(gap)分割成N个组，对每个组进行直接插入排序，然后再减小步长进行直接插入排序，直到gap达到最小值，即数组基本有序时，再对数组进行直接插入排序，此时直接插入排序可以达到最高效率。当gap=1时，希尔排序退化成直接插入排序。因此，我们可以1)让gap>1,然后跳转到直接插入排序，或者2)gap减少至1，流畅地进入直接插入排序。
所有的gap值组成的序列叫做增量序列。

- 不同的增量序列得到不同的时间复杂度
- 增量元素不互质，则小的增量元素可能根本不起作用
- 原始增量序列，最坏事件复杂度是$\Theta(N^2)$，每一次插入排序都不起作用，最后退化成简单插入排序
- Sedgewick增量序列$1, 5, 19, 41, 109, \dots$，$9 \times 4^i - 9 \times 2^i + 1$,$4^i - 3 \times 2^i + 1$的最差时间复杂度猜想是$O(N^{\frac{4}{3}})$
- 希尔排序的空间复杂度是$O(1)$
- 希尔排序是不稳定的排序，分组排序导致它不稳定

伪码

1.原始增量序列 $D_0 = \lfloor \frac{length}{2} \rfloor, D_k = \lfloor \frac{D_{k+1}}{2} \rfloor$

```
shell_sort(A, length)
```

```
D = length/2
while D > 0 //gap逐渐减小
    for i = D to length - 1
        /// 一轮插入排序
        key = A[D]
        j=D
        while j-D >= 0 && A[j-D]>key ///j-D >= 0 保证索引值>0
            A[j] = A[j-D] ///空出A[j]
            j-=D
        A[j] = key
    D = D / 2
```

2.Sedgewick增量序列

`shell_sort(A, length)`

```
    int Si, D, P, i;
    ElementType Tmp;
    //这里只列出一小部分增量
    Sedgewick[] = {929, 505, 209, 109, 41, 19, 5, 1, 0};
    Si = 0
    ///增量序列小于序列长度
    while Sedgewick[Si] > length
        Si++
    while Si >= 0
        D = Sedgewick[Si]
        for i = D to length - 1
            /// 一轮插入排序
            key = A[i]
            j=i
            while j-D >= 0 && A[j-D]>key ///j-D >= 0 保证索引值>0
                A[j] = A[j-D]
                j-=D
            A[j] = key
        Si--
```

堆排序

堆排序是选择排序的改进，在选择排序中，找最小元的操作是提高速度的瓶颈。但是他利用了堆的性质，父节点的值大于子节点，且满足完全二叉树(除最后一层外，其它层都有$2n$个节点，最后一层的节点都连续集中在树的左边，堆是一棵完全二叉树，存储效率很高)，而从堆中取得最大/最小值时间复杂度为$O(1)$，大大提高了找最小元的效率。

二叉排序树:父节点的值大于**所有左子树**的值，小于**所有右子树**的值
大顶堆:父节点的值大于**子节点**的值
小顶堆: 父节点的值小于**子节点**的值

- 堆排序中，升序用大顶堆，降序用小顶堆，可以降低空间复杂度$O(N)->O(1)$
- 堆排序是选择排序的一种，它利用了数组的特点快速定位指定索引的元素
- 堆排序的时间复杂度为$O(Nlog_2N)$，空间复杂度是$O(1)$
- 不稳定排序(???)
- 虽然堆排序给出最佳平均时间复杂度，但实际效果不如用Sedgewick增量序列的希尔排序(???)
- max_heapify 时间复杂度是$O(log_2N)$(推导:)

伪代码

`heap_sort(A, length)`

```
///建立最大堆
i = length/2 ///下界
while i >= 0
    max_heapify(A, i, length-1)
    i--
i = length - 1
while i > 0
    A[0] <-> A[i]
    ///维护最大堆性质
    max_heapify(A, 0, i)
    i--
```

`max_heapify(A , i, j)`

```
///向下过滤函数:将数组中以A[i]为根的子堆调整为最大堆
///调整根节点、两个子节点的位置，让它满足大顶堆
Parent = i
while Parent*2+1 <= j
    Child = Parent*2+1
    if Child!=j && A[Child] < A[Child+1]
        Child++ //Child指向左右结点中较大的
    if A[Parent] < A[Child]
        ///交换最大项和根
        A[Parent] <-> A[Child]
    Parent = Child
```

快速排序

快排是一种在实际应用中经常用到的算法，它的应用场景是大规模的数据排序，并且实际性能要优于归并排序，快排可以看做冒泡排序的改进。快排采用了分而治之的思想，它的基本思路是，从数组中选取一个主元，把所有大于主元的元素都放到它的后面，所有小于主元的元素都放到它的前面，主元将数列分成两部分，再分别对这两部分进行相同的操作，直到数组不能切分为止。此时数组为有序数组。不同的主元选择，影响时间复杂度。

- 快速排序是不稳定

- 最好情况下，每次选择的主元都将数组分成等长的两部分，此时数组的长度减小最快，时间复杂度为$O(log_2N)$
- 主元的选择：如果选取数组的头或尾作为主元，而且数组是有序的，那么退化成冒泡排序，时间复杂度退化成$O(N^2)$；我们可以选择，头、尾、中间元素的中值作为主元
- 快排的空间复杂度是O(log_{2}N)
- 快排在给数组分组时，有两种方法:单边扫描和双边扫描
- 当数组基本有序时，插入排序的速度很快，我们可以设置一个截断值(cutoff)，当数组长度小于截断值时，采用插入排序，不同截断值对快排的四度影响也不同；同时，用快排小规模数据还不如插入排序快
- 截断值(cutoff)通常设定是10

伪代码

`quick_sort(A, length)` 双边扫描方法

```
left = 0
right = length - 1
if left < right
QSort(A, left, right)
```

`QSort(A, left, right)`

```
///调整A[left] A[right] A[median] 的位置，让A[left] <= A[median] <= A[right]
///pivot = A[right]
if cutoff < right - left + 1
    pivot = Median(A, left, right)
    ///双边扫描
    i =  left + 1
    j = right - 1
    while i < j
        while A[i] <= pivot
            i++
        while A[j] >= piort
            j++
        A[i] <-> A[j]
    A[i] <-> A[right]
    QSort(A, left, i - 1)
    QSort(A, i + 1, right)
else
    insertion_sort()
```

归并排序

归并排序的核心是将两个有序表合并，如果有序表总共有N个元素，那么，时间复杂度是$O(N)$。归并排序的核心思想是分而治之，他的基本思路是将两个有序子数列合并(Merge)成一个有序数列，如果说快排是自顶向下的排序，那么为了得到有序子序列，归并排序就是自低向下的排序。

- 归并排序是一种稳定的排序方法，Merge过程没有破坏稳定性

- 因为归并的过程需要借助一个等长的数组归并排序不是原址排序，它的空间复杂度是$O(N)$，同时，它能排序的元素规模小了一半
- 归并排序的时间复杂度是$O(NlogN)$

伪代码

1.递归版本的归并排序

```
merge_sort(A, length)
```

```
    //申请新的空间
    new tempA
    if tempA != NULL
       MSort(A, tempA, 0, length - 1)
    else
        print("empty")
```

```
MSort(A, tempA, left, right_end)
```

```
    if left < right_end
        center =  (left + right_end) / 2
        MSort(A, tempA, left, center)
        MSort(A, tempA, center + 1, right_end)
        Merge(A, tempA, left, center + 1, right_end)
```

```
Merge(A, tempA, left, right, right_end)
```

```
//A[left,...right-1]和A[right,...,right_end]merge到A[left,...,right_end]
i = left
j = right
num = right_end - left + 1
temp = left
//归并过程
while i <=  left-1 && j <= right_end
    if A[i] <= A[j]
        tempA[temp++] = A[i++]
    else
        tempA[temp++] = A[j++]
while i <= left-1   //直接复制左边剩下的
    tempA[temp++] = A[i++]
while j <= right_end    //直接复制右边剩下的
    tempA[temp++] = A[j++]
temp = right_end
//拷贝数组
while temp >= right_end - num + 1
    A[temp] = tempA[temp]
```

2.非递归版本的归并排序

非递归版本的归并排序更体现了归并排序自底向上的特质

`merge_sort(A, length)`

```
//申请新内存
new tempA
子数组长度
sub_length = 1
if tempA!=NULL
    while sub_length < length
        Merge_pass(A, tempA, length, sub_length)
        sub_length = sub_length * 2
        Merge_pass(tempA, A, length, sub_length)  //如果sub_length > length
Merge_pass函数会将数组原封不动的复制过去
        sub_length = sub_length * 2
else
 print("empty)
```

`Merge_pass(A, tempA, N, length)`

```
//按照length长度作为子序列长度进行merge,归并的结果方放在tempA中
left = 0
right = left + length
right_end = left + 2 * length - 1
while left <= N - 2 * length //不能让right_end >= N
    Merge1(A ,tempA, left, right, right_end)
    left = left + 2*length
    right = left + length
    right_end = left + 2 * length - 1
if right < N ///尾部还有两个序列
    Merge1(A ,tempA, left, right, N - 1)
else //尾部只有一个序列
    for i = left to N - 1
        tempA[i] = A[i]
```

桶排序

桶排序不是一种基于比较的排序，假设我们知道待排元素的范围为$[1, N]$，那么我们建立N个桶，扫描一遍所有元素，将元素放到合适的桶中，最后再扫描一遍所有的桶，依次打印桶中的元素。

- 假设有M个元素，元素的范围是$[1, N]$，那么时间复杂度是$O(M+N)$，这个时间复杂度看似是线性的，但是当$N >> M$时，时间复杂度就是指数级别的
- 另外桶排序也不是一种原址排序，但它可以做成稳定排序

伪代码

`bucket_Sort(A, length)`

```
count[]初始化;
while 读入1个学生成绩grade
    将该生插入count[grade]链表;
for i = 1 to M
    if ( count[i] )
        输出整个count[i]链表;
```

计数排序

基数排序