

排序

排序的基本概念

排序(Sorting)是将数据的任意序列，重新排列成一个按照关键字有序(非递增有序或非递减有序)的序列的过程。

假设含有 n 个记录的序列为 $\{R_1, R_2, \dots, R_n\}$ ，其相对应的关键字序列为 $\{K_1, K_2, \dots, K_n\}$ ，这些关键字相互之间可以进行比较。若在它们之间存在着这样一个关系 $K_{S1} \leq K_{S2} \leq \dots \leq K_{Sn}$ ，则按此固有关系将 n 个记录的序列重新排列为 $\{R_{S1}, R_{S2}, \dots, R_{Sn}\}$ 的操作称为排序。

若关键字 K_i 是记录 R_i ($i = 1, 2, \dots, n$)的主关键字，则每个关键字可以唯一地标识一个记录，关键字各不相同。若 K_i 为次关键字，则在待排序的记录序列中可能有多个记录的关键字相同。在按照某种方法排序的过程中，若关键字相同的记录的相对位置不发生改变，则称所用的排序方法是稳定的；反之，称所用的排序方法是不稳定的。

在排序过程中，若待排序记录序列全部读入内存中处理，则称此类排序问题为内部排序；反之，若参加排序的记录数量较大，在排序过程中仅有部分记录在内存中，还要对外存中的记录进行访问，则称此类排序问题为外部排序。内部排序适用于记录个数不是很多的文件，而外部排序适用于记录个数很多的大文件，整个排序过程需要在内、外存之间多次交换数据才能得到排序的结果。

内部排序的方法通常可分为以下5类：

1. **插入排序**：将无序序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序序列的长度。
2. **交换排序**：通过“交换”无序序列中的相邻记录从而得到其中关键字最小或最大的记录，并将它增加到有序序列中，以增加记录的有序序列的长度。
3. **选择排序**：从记录的无序序列中“选择”关键字最小或最大的记录，并将它加到有序子序列中，以增加记录的有序序列的长度。
4. **归并排序**：通过“归并”两个或两个以上的有序序列，逐步增加有序序列的长度。
5. **分配排序**：通过对无序序列中的记录反复进行“分配”和“收集”操作，逐步使得无序序列变为有序序列。

一般来说，在排序过程中有两种基本操作：**比较关键字的大小**，以及**移动记录的位置**。前一种操作对大多数排序方法都是必要的，而后一种操作可通过改变记录的存储方式来避免。

评价排序算法效率的标准主要有两条：

1. 执行算法所需要的时间开销
2. 执行算法所需要的额外存储空间

插入排序

插入排序(Insertion Sorting)的基本思想是：首先将第一个记录看作一个有序序列，然后每次将下一个待排序的记录有序插入已排好序的有序序列中，使得有序序列逐步扩大，直到所有记录都加到有序序列中。主要有三种插入排序方法：直接插入排序、这班插入排序和希尔排序。

直接插入排序

直接插入排序(Straight Insertion Sort)是一种比较简单的插入排序方法。假设在排序过程中，记录序列为 $R[0, \dots, n-1]$ ，首先将第一个记录 $R[0]$ 看作一个有序子序列，然后依次将记录 $R[i]$ ($1 \leq i \leq n-1$)插入有序子序列 $R[0, \dots, i-1]$ 中，使记录的有序子序列从 $R[0, \dots, i-1]$ 变为 $R[0, \dots, i]$ 。

```
template <class Type>
void straightInsertSort(Type R[], int size){
    int pos, j; // pos为待插入位置记录
    Type temp;
    for(pos = 1, pos < size; pos++){
        temp = R[pos]; // 记录待插入的元素
        for(j = pos - 1; j >= 0 && R[j] > temp; j--) // 从后往前查找插入位置
            R[j + 1] = R[j]; // 将R[j]后移
        R[j + 1] = temp; // 插入R[pos]到正确位置
    }
}
```

直接插入排序算法简单，容易实现，是一种稳定的排序方法。当待排序记录数量 n 很小且局部有序时较为适用。当 n 很大时，其效率不高。它的基本操作有两种：比较关键字和移动记录。

最好的情况是：初始序列为有序，关键字总比较次数为最小值 $(n-1)$ ，即 $\sum_{i=2}^n 1$ ，无需后移记录，但在一趟排序开始时要将待排序记录放进临时变量中，在一趟排序结束时再将待排序记录放到合适位置，需要移动记录 $2(n-1)$ 次。

最坏的情况是，初始序列为逆序，关键字总比较次数为最大值 $\sum_{i=2}^n i = \frac{(n+2)(n-1)}{2}$ ，记录移动次数为最大值 $\sum_{i=2}^n i - 1 + 2 = \frac{(n+4)(n-1)}{2}$ 。

由此可见，直接插入排序算法最好情况的时间复杂度为 $O(n)$ ，最坏情况时间复杂度和平均时间复杂度为 $O(n^2)$ 。在直接插入排序中，只需一个记录大小的辅助空间用于存放待插入的记录，因此空间复杂度为 $O(1)$ 。

折半插入排序

对直接插入排序算法进行改进，可从减少比较和移动次数这两方面着手。当前面有 $i-1$ 个有序记录序列时，要插入第 i 个序列，可利用折半查找方式确定插入位置，以减少比较次数。这种排序方法称为折半排序插入(Binary Insertion Sort)，其算法如下：

```
template <class Type>
void binaryInsertSort(Type R[], int size){
    int pos, j, low, high, mid;
    Type temp;
    for(pos = 1; pos < size; pos++){
        temp = R[pos];
        low = 0; high = pos - 1;
        while(low <= high){
            mid = (low + high) / 2; // 查找中间位置
            if(R[mid] > temp) high = mid - 1; // 在左边继续查找
            else low = mid + 1; // 在右边继续查找
        }
        for(j = pos - 1; j >= low; j--)
            R[j + 1] = R[j]; // 将R[j]后移
        R[low] = temp; // 插入R[pos]到正确位置
    }
}
```

折半插入排序比直接插入排序明显地减少了关键字间的比较次数，但记录移动的次数不变，故其时间复杂度仍然为 $O(n^2)$ 。

希尔排序

希尔排序(Shell Sort)又称为缩小增量排序。Shell从“减少记录个数”和“基本有序”两方面对直接插入排序进行了更改。其基本思想是：将待排序的记录划分成几组，间距相同的记录分在一组中，对各组分别实施直接插入排序。当经过几次分组排序后，记录的排序已经基本有序，再对所有的记录实施最后的直接插入排序。通过分组，一方面减少了参与直接插入排序的数据量；另一方面可以先比较那些间距较大的记录，避免频繁的移动相邻记录。当待排序记录个数较少且待排序记录已基本有序时，直接插入排序的效率是最高的。对于有 n 个记录的初始序列，希尔排序的具体步骤如下：

1. 首先取一个整数 $gap < n$ 作为增量。
2. 将全部记录分为 gap 个子序列，所有间距为 gap 的记录分在同一个子序列中，对每个子序列分别实施直接插入排序。
3. 然后缩小增量 gap ，冲入步骤2的子序列划分和排序工作，直到最后 gap 等于1，将所有的记录放在一组中，进行最后一次直接插入排序。

Shell提出从 $gap = \lfloor \frac{n}{2} \rfloor$ 开始划分子序列，每次缩小增量 $gap = \lfloor \frac{gap}{2} \rfloor$ ，直到 $gap=1$ 为止。这种划分方式带来的问题是：

1. 可能存在大量重复的划分，即已经分在同一组中的记录下一趟可能仍然分在同一组中。
2. 直到 $gap=1$ 时，相邻的奇数位置的记录才会与偶数位置的记录进行比较，效率较低。

目前对于如何分组(即 gap 如何取值)没有统一意见，只有如下共识：第一个增量 gap 小于表长，最后一个 gap 等于1，增量序列中的值没有除1以外的公因子。若用Shell提出的分组方法，则希尔排序的算法如下：

```

template <class Type>
void shellSort(Type R[], int size){
    int gap, pos, j;
    Type temp;
    for(gap = size / 2; gap > 0; gap /= 2){
        for(pos = gap, pos < size; pos++){
            temp = R[pos]; // 记录待插入的元素
            for(j = pos - gap; j >= 0 && R[j] > temp; j -= gap) // 从后往前查找插入位置
                R[j + gap] = R[j]; // 将R[j]后移
            R[j + gap] = temp; // 插入R[pos]到正确位置
        }
    }
}

```

希尔排序适用于待排序的记录数量较大的情况，是一种不稳定的排序方法。希尔排序的时间性能与其选定的增量序列有关，有人在大量测试的基础上推导出，希尔排序的时间复杂度约为 $O(n^{1.3})$ 。其空间复杂度为 $O(1)$ 。

交换排序

交换排序(Exchange Sort)的基本思想是：对待排序记录序列中元素间关键字比较，若发现记录逆序，则交换。主要有两种交换排序方法：冒泡排序和快速排序。

冒泡排序

冒泡排序(Bubble Sort)是一种比较简单的交换排序方法。它的基本思想是：对所有相邻记录的关键字进行比较，若不满足排序要求(即逆序)，则将其交换，直到所有记录排好序为止。对于由 n 个记录组成的记录序列，冒泡排序的步骤如下：

1. 将整个待排序的记录序列划分成有序区和无序区，初始状态有序区为空，无序区包括所有待排序的记录。
2. 每一趟冒泡排序，对无序区从头到尾比较相邻记录的关键字，若逆序，则将关键字小的记录换到前面，关键字大的记录换到后面。一趟排序后，无序区中关键字最大的记录进入有序区。
3. 重复执行步骤2，若在某一趟排序中没有发生交换操作，则说明待排序记录已经全部有序，排序提前结束；否则，最多需要经过 $n - 1$ 趟冒泡排序，才能将这 n 个记录重新按照关键字排好序。

冒泡排序算法如下：

```

template <class Type>
void bubbleSort(Type R[], int size){
    int i, j;
    bool flag = true; // 标志变量，表示是否发生交换
    for(i = 1; i < size && flag; i++){
        flag = false; // 每次开始时假设没有交换
        for(j = 0; j < size - i; j++){
            if(R[j+1] < R[j]){
                swap(R[j], R[j+1]);
                flag = true; // 如果发生交换，设置标志为true
            }
        }
    }
}

```

冒泡排序是一种稳定的排序方法，关键字的比较次数和记录的交换次数与记录的初始顺序有关。最好的情况是，初始序列为有序，比较次数为 $n - 1$ ，交换次数为0；最坏的情况是，初始序列为逆序，比较次数和交换次数均为 $\sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}$ ，记录的移动次数为 $\frac{3n(n-1)}{2}$ ，因此，最好情况的时间复杂度为 $O(n)$ ，最坏情况时间复杂度和平均时间复杂度为 $O(n^2)$ 。在冒泡排序过程中，只需要一个记录大小的辅助空间用于交换，因此空间复杂度为 $O(1)$ 。

快速排序

快速排序(Quick Sort)，也称分区交换排序，是对冒泡排序的改进。在冒泡排序中，记录的比较和移动是在相邻的位置进行的，记录每次交换只能消除一个逆序，因而总的比较和移动次数较多。在快速排序中，通过分区的一次交换能消除多个逆序，它是目前最快的内部排序算法，其基本思想如下：

1. 在待排序记录序列中选取一个记录作为枢轴(pivot)，并以该记录的关键字(key)为基准。
2. 凡关键字小于枢轴的记录均移动至枢轴之前，凡关键字大于枢轴的记录均移动至枢轴之后。一趟排序之后，记录序列被划分为两个子序列L和R，使得L中所有记录的关键字都小于或等于key，R中所有记录的关键字都大于或等于key，枢轴处于子序列L和R之间，刚好在最终位置。
3. 对子序列L和R分别继续进行快速排序，直到子序列中只有一个记录为止。

一趟快速排序的具体做法是：设置两个指针low, high分别用来指示将要与枢轴进行比较的左侧记录和右侧记录，首先从high所指位置开始向前查找关键字小于枢轴关键字的记录，将其与枢轴进行交换，再从low所指位置开始向后查找关键字大于枢轴关键字的记录，将其与枢轴进行交换，反复执行以上两步，直到low和high相等。在这个过程中，记录交换都是与枢轴之间发生的，每次交换要移动3次记录。我们可以先用临时变量暂存枢轴，只移动要与枢轴交换的记录，直到最后再将临时变量中保存的枢轴放入最终位置，这种做法可减少排序中记录的移动次数。

以最左侧记录作为枢轴的快速排序算法如下：

一趟快速排序：

```
template <class Type>
int partition(Type S[], int low, int high){
    Type temp = S[low]; // 暂存枢轴
    while(low != high){
        while(low < high && S[high] >= temp) high--; // 从右向左找第一个小于temp的元素
        if(low < high){
            S[low] = S[high]; // 将该元素移到左边
            low++; // 移动low指针
        }
        while (low < high && S[low] <= temp) low++; // 从左向右找第一个大于temp的元素
        if(low < high){
            S[high] = S[low]; // 将该元素移到右边
            high--; // 移动high指针
        }
    }
    S[low] = temp; // 将temp放到正确的位置
    return low; // 返回分区分点
}
```

递归快速排序：

```
template <class Type>
void quickSort(Type S[], int low, int high){
    int pivot;
    if(low >= high) return; // 如果区间无效，直接返回
    pivot = partition(S, low, high); // 获取分区分点
    quickSort(S, low, pivot - 1); // 对左半部分递归排序
    quickSort(S, pivot + 1, high); // 对右半部分递归排序
}
```

快速排序的接口函数：

```
template <class Type>
void quickSort(Type S[], int size){
    quickSort(S, 0, size - 1); // 调用快速排序
}
```

若每次选取序列中最左端记录作为枢轴，当待排序记录的初始状态为按关键字有序时，快速排序将退化为冒泡排序，因此快速排序的最坏时间复杂度为 $O(n^2)$ 。也就是说，依次划分后枢轴两侧记录数量越接近，排序速度将越快。那么，枢轴的选择将非常重要，它决定了一趟排序后两个子序列的长度，进而影响整个算法的效率。为避免出现一趟排序后记录集中在枢轴一侧的情况，常采用“三者取中”法，即比较左端、右端和中间位置上三个记录的关键字，然后去关键字为中间记录作为枢轴，这将改善算法在最差情况下的性能。

下面分析快速排序的时间性能。假设一次划分所得枢轴在位置*i*处，则对*n*个记录进行快速排序所需时间为：

$$T(n) = T(i) + T(n - i - 1) + Cn$$

其中： Cn 为对*n*个记录进行一次划分所需时间， $T(i)$ 和 $T(n - i - 1)$ 为一次划分后继续对两个子序列进行快速排序的时间，若待排序列中记录的关键字是随机分布的，则*i*取 $0 \sim n - 1$ 中任意一值的可能性相同，因此可得快速排序所需时间的平均值为：

$$\begin{aligned}
T_{avg}(n) &= \frac{1}{n} \sum_{i=0}^{n-1} [T_{avg}(i) + T_{avg}(n-i-1)] + Cn = \frac{2}{n} \sum_{i=0}^{n-1} T_{avg}(i) + Cn \\
\Rightarrow nT_{avg}(n) &= 2 \sum_{i=0}^{n-1} T_{avg}(i) + Cn^2 \\
\Rightarrow (n-1)T_{avg}(n-1) &= 2 \sum_{i=0}^{n-2} T_{avg}(i) + C(n-1)^2 \\
&\Rightarrow nT_{avg}(n) - (n-1)T_{avg}(n-1) = 2T_{avg}(n-1) + C(2n-1) \\
&\Rightarrow nT_{avg}(n) = (n+1)T_{avg}(n-1) + C(2n-1) \\
&\Rightarrow \frac{T_{avg}(n)}{n+1} = \frac{T_{avg}(n-1)}{n} + \frac{2C}{n+1} \\
&\Rightarrow \frac{T_{avg}(n)}{n+1} = \frac{T_{avg}(1)}{2} + 2C \sum_{i=3}^{n+1} \frac{1}{i} \\
&\Rightarrow T_{avg}(n) = \frac{T_{avg}(1)}{2}(n+1) + 2C(n+1) \sum_{i=3}^{n+1} \frac{1}{i}
\end{aligned}$$

因此，快速排序的平均时间复杂度为 $O(n \log n)$ 。就平均而言，快速排序是目前最好的内部排序方法。

选择排序

选择排序(Selection Sort)的基本思想是：依次从待排序记录序列中选出关键字最小(或最大)的记录、关键字次之的记录.....，并分别将它们定位到序列左侧(或右侧)的第1个位置、第2个位置.....，直至序列中只剩下一个最小(或最大)的记录为止，从而使待排序的记录称为按关键字大小排列的有序序列。主要有三种选择排序方法：直接选择排序，堆排序和锦标赛排序。

直接选择排序

直接选择排序(Straight Selection Sort)是一种比较简单选择排序方法。它的基本思想是：对于由 n 个记录组成的记录序列，第一趟，从 n 个记录中选取关键字最小的记录与第1个记录互换；第二趟，从剩余的 $n-1$ 个记录中选取关键字最小的记录与第2个记录互换；第 i 趟，从剩余的 $n-i+1$ 个记录中选取关键字最小的记录与第 i 个记录互换。重复以上过程，直到剩余记录仅有一个为止。

直接选择排序算法如下：

```

template <class Type>
void straightSelectionSort(Type R[], int size){
    int pos, min, j; // min为一趟排序中最小记录的下标
    for(pos = 0; pos < size - 1; pos++){
        min = pos;
        for(j = pos + 1; j < size; j++)
            if(R[j] < R[min]) min = j; // 找到最小记录的下标
        if(pos != min) swap(R[pos], R[min]); // 如果最小记录不是当前记录，则交换
    }
}

```

在直接选择排序过程中存在大跨度的数据移动，是不稳定的排序方法，关键字的比较次数和记录的初始顺序无关。其比较次数为 $\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$ ，交换次数不超过 $n-1$ 次。当待排序记录初始为正序时，不发生交换；当初始为逆序时，发生 $n-1$ 次交换，即移动次数最多 $3(n-1)$ 次。因此，直接选择排序的平均时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ 。

堆排序

锦标赛排序

归并排序

基数排序

各种内部排序方法的比较

外部排序

置换选择排序

多路并归排序