# 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$\{R_1, R_2, \ldots, R_n\}$，其相对应的关键字序列为$\{K_1, K_2, \ldots, K_n\}$，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$K_{S1} \leq K_{S2} \leq \cdots \leq K_{Sn}$，则按此固有关系将$n$个记录的序列重新排列为$\{R_{S1}, R_{S2}, \ldots, R_{Sn}\}$的操作称为排序。

若关键字$K_i$是记录$R_i$ $(i = 1, 2, \ldots, n)$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$R[0, \ldots, n-1]$，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$R[0, \ldots, i-1]$中，使记录的有序子序列从$R[0, \ldots, i-1]$变为$R[0, \ldots, i]$。

```
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; // 在左半边继续查找
            else low = mid + 1; // 在右半边继续查找
        }
        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=$\left\lfloor\frac{n}{2}\right\rfloor$开始划分子序列，每次缩小增量gap=$\left\lfloor\frac{gap}{2}\right\rfloor$，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，通过分区间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的记录作为枢轴，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：

$$T_{avg}(n)=\frac{1}{n}\sum_{i=0}^{n-1}[T_{avg}(i)+T_{avg}(n-i-1)]+Cn=\frac{2}{n}\sum_{i=0}^{n-1}T_{avg}(i)+Cn$$

$$\Rightarrow nT_{avg}(n)=2\sum_{i=0}^{n-1}T_{avg}(i)+Cn^2$$

$$\Rightarrow (n-1)T_{avg}(n-1)=2\sum_{i=0}^{n-2}T_{avg}(i)+C(n-1)^2$$

$$\Rightarrow nT_{avg}(n)-(n-1)T_{avg}(n-1)=2T_{avg}(n-1)+C(2n-1)$$

$$\Rightarrow nT_{avg}(n)=(n+1)T_{avg}(n-1)+C(2n-1)$$

$$\Rightarrow \frac{T_{avg}(n)}{n+1}=\frac{T_{avg}(n-1)}{n}+\frac{2C}{n+1}$$

$$\Rightarrow \frac{T_{avg}(n)}{n+1}=\frac{T_{avg}(1)}{2}+2C\sum_{i=3}^{n+1}\frac{1}{i}$$

$$\Rightarrow T_{avg}(n)=\frac{T_{avg}(1)}{2}(n+1)+2C(n+1)\sum_{i=3}^{n+1}\frac{1}{i}$$

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)是一种比较简单的选择排序方法。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

利用堆排序(Heap Sort)选出关键字最小的记录只需要$O(\log n)$的时间。其基本思路如下：

1. 由存储于数组中的初始序列，建成一个大根堆(或小根堆)。
2. 将堆顶的最大(或最小)记录与序列中最后一个记录互换，最大(或最小)记录存放在数组末尾的有序段中。
3. 将序列中除末尾的有序段之外的剩余记录重新调整成堆。
4. 反复执行步骤2、3共$n-1$次，若初始建立的是大根堆，则得到一个非递减的序列；若初始建立的是小根堆，则得到一个非递增序列。

堆排序的过程就是建堆和反复调整堆的过程。那么如何建堆呢？具有$n$个结点的完全二叉树，若编号从0到$n-1$，则其最后一个分支结点的编号是$\lfloor\frac{n-1}{2}\rfloor$，从该结点到根结点(编号为0)反复调整堆，就建立了初始堆。设结点编号从0开始，对于编号为$i$的结点，其双亲和做、右孩子若存在，则双亲的编号为$\lfloor\frac{i-1}{2}\rfloor$，左孩子的编号为$2i+1$，右孩子的编号为$2i+2$。

利用大根堆实现堆排序的算法如下：

向下调整成堆：

```
template <class Type>
void siftDown(Type R[], int pos, int size){
    int child;
    Type temp = R[pos];
    for(; pos * 2 + 1 < size; pos = child){
        child = pos * 2 + 1; // 左孩子
        if(child + 1 < size && R[child] < R[child + 1]) // 如果右孩子存在且大于左孩子
            child++; // 则选择右孩子
        if(R[child] > temp) R[pos] = R[child]; // 如果孩子大于父节点，则将孩子上移
        else break; // 否则停止
    }
    R[pos] = temp; // 将父节点放到正确位置
}
```

堆排序：

```
template <class Type>
void heapSort(Type R[], int size){
    for(int i = size / 2 - 1; i >= 0; i--) // 建立初始堆
        siftDown(R, i, size);
    for(int i = size - 1; i > 0; i--){
        swap(R[0], R[i]); // 将堆顶元素与最后一个元素交换
        siftDown(R, 0, i); // 对剩余的堆进行调整
    }
}
```

d皮埃叙的时间主要是由建堆和反复调整堆两部分时间开销构成。由$n$个关键字建成的堆，其深度为$h=\lfloor \log n \rfloor +1$，根结点向下调整时能移动的最大层数为$h-1$，关键字的最大比较次数为$2(h-1)$次；第$i$层最多有$2^{i-1}$个结点，对于第$i$层的结点，向下调整堆进行的关键字比较的次数最多为$2(h-i)$次。因此，建堆的关键字比较的次数最多为$T(n)$，它满足下式：

$$T(n)=\sum_{i=1}^{i=h-1}2^{i-1}\times(h-i)\leq\sum_{i=1}^{i=h-1}2^{i}\times(h-1)<4n$$

因此，建堆的时间复杂度为$O(n)$。反复调整堆的操作体现在`heapSort`的第二个`for`循环中，共计调用`siftDown`函数$n-1$次，该循环的时间复杂度为$O(n\log n)$。因此，堆排序的时间复杂度为$O(n\log n)$。在堆排序过程中存在大跨度的数据移动，是一种不稳定的排序方法，适合记录数较多的情况，其最坏时间复杂度为$O(n\log n)$。堆排序只需要一个记录大小的辅助存储空间，空间复杂度为$O(1)$。

### 锦标赛排序

锦标赛排序(Tournament Sort)也称树选择排序(Tree Selection Sort)，是一种按照锦标赛的思想进行选择排序的方法。它的基本思想是：首先对$n$个待排序记录的关键字进行两两比较，从中选出$\lceil \frac{n}{2} \rceil$个较小者再两两比较，直到选出关键字最小的记录为止，此为一趟排序。我们将一趟选出的关键字最小的记录称为“冠军”，在输出冠军后，将冠军所在叶结点的关键字改为最大值(如$\infty$)，使它不能再战胜其他结点，然后更新从冠军对应的叶结点到根结点的路径上所有比赛，继续进行锦标赛排序，选出关键字次小的记录，如此重复，直到输出全部有序序列为止。

对$n$个记录的锦标赛排序，每选择一个记录仅需进行$\lceil \log n \rceil$次比较，因此它的时间复杂度和堆排序一样都是$O(n\log n)$，但是这种方法需要$O(n)$的辅助存储空间，并且与胜者要进行多次多余的比较。

## 归并排序

归并排序(Merging Sort)时利用归并技术进行的排序。所谓归并，是指将两个或两个以上的有序表合并成一个新的有序表。在内部排序中，通常采用的是2-路归并排序。

2-路归并排序的基本思想是：将一个具有$n$个待排序记录的序列看成$n$个长度为1的有序序列，然后对相邻的子序列进行两两归并，得到$\lceil \frac{n}{2} \rceil$个长度为2的有序子序列，继续对相邻的子序列进行两两归并，得到$\lceil \frac{n}{4} \rceil$个长度为4的有序序列，重复归并过程，直到得到一个长度为$n$的有序序列为止。

2-路归并序列算法如下：

归并相邻的两个有序子序列。将有序序列`R[low, mid-1]`和`R[mid, high]`归并为有序序列`[low, high]`：

```
template <class Type>
void merge(Type R[], Type temp[], int low, int mid, int high){
    int i = low, j = mid, k = 0;
    while(i < mid && j <= high){ // R中记录由小到大复制到temp中
        if(R[i] < R[j]) temp[k++] = R[i++];
        else temp[k++] = R[j++]; // 将较小的元素放入临时数组
    }
    while(i < mid) temp[k++] = R[i++]; // 将左半部分剩余元素复制到temp
    while(j <= high) temp[k++] = R[j++]; // 将右半部分剩余元素复制到temp
    for(i = 0, k = low; k <= high)
        R[k++] = temp[i++]; // 将临时数组中的元素复制回原数组
}
```

递归2-路归并排序。通过递归调用实现堆子序列R[low, high]的排序过程，将其归并为有序段：

```
template <class Type>
void mergeSort(Type R[], Type temp[], int low, int high){
    if (low == high) return; // 如果区间无效，直接返回
    int mid = (low + high) / 2; // 计算中间位置
    mergeSort(R, temp, low, mid); // 对左半部分递归排序
    mergeSort(R, temp, mid + 1, high); // 对右半部分递归排序
    merge(R, temp, low, mid + 1, high); // 合并两个已排序的子数组
}
```

2-路归并排序的接口函数。参数为待排序序列R，序列大小size：

```
template <class Type>
void mergeSort(Type R[], int size){
    Type* temp = new Type[size]; // 创建临时数组
    mergeSort(R, temp, 0, size - 1); // 调用归并排序
    delete[] temp; // 释放临时数组
}
```

2-路归并序列进行$i$趟排序之后，有序子序列长度小于等于$2^i$。因此，当待排序记录数量为$n$时，需要进行$\lceil \log n \rceil$。每趟归并所需时间与记录个数成正比，因此2-路归并排序的时间复杂度为$O(n \log n)$。由于2-路归并排序每次划分的两个子序列长度基本一致，因此其最好、最差、平均时间复杂度都是$O(n \log n)$。在排序过程中借助了一个与初始序列相同大小的辅助数组，因此空间复杂度为$O(n)$。归并序列是一种稳定的排序方法。

# 各种内部排序方法的比较

比较前面讨论的各种内部排序方法，我们有以下五点结论：

1. 若$n$较小(如$n \leq 50$)，可采用直接插入排序或直接选择排序。由于直接插入排序所需记录移动操作较直接选择排序多，因此当记录本身信息量较大时，宜选用直接选择排序。
2. 若待排序记录的初始状态已经按照关键字基本有序，则选用直接插入排序或冒泡排序为宜。
3. 当$n$较大时，若关键字有明显结构特征(如字符串、整数等)，切关键字位数较少，易于分解，则采用基数排序比较好。若关键字无明显结构特征或取值范围属于某个无穷集合(例如实数型关键字)，则应借助于比较的方法来进行排序，可采用时间复杂度为$O(n \log n)$的排序方法：快速排序、堆排序或归并排序。快速排序是基于比较的内部排序中目前被认为是最好的方法，当待排序记录的关键字是随机分布时，快速排序的平均时间最短。堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。若要求排序稳定，则可选用归并排序。归并排序既适合内部排序也适合外部排序。
4. 对于按照主关键字进行排序的记录序列，所用的排序方法是否稳定无关紧要；而对于按照次关键字进行排序的记录序列，应根据具体问题慎重选择排序方法。应该指出的是，稳定性是由方法本身决定的。
5. 前面讨论的排序方法，大多是用一维向量实现的。若记录本身信息量大，为了避免移动记录耗费大量时间，可以使用链式存储结构。例如插入排序和归并排序都容易在链表上实现。但是像快速排序和堆排序这样的排序方法，很难在链表上实现。一种解决方法是提取关键字建立索引表，然后对索引表进行排序。

| 排序方法 | 平均时间 | 最坏情况 | 辅助空间 | 稳定性 |
| --- | --- | --- | --- | --- |
| 直接插入排序 | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 折半插入排序 | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 直接选择排序 | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 希尔排序 | $O(n^{1.3})$ | $O(n^{1.3})$ | $O(1)$ | 不稳定 |
| 快速排序 | $O(n\log n)$ | $O(n^2)$ | $O(\log n)$ | 不稳定 |
| 堆排序 | $O(n\log n)$ | $O(n\log n)$ | $O(1)$ | 不稳定 |
| 2-路并归排序 | $O(n\log n)$ | $O(n\log n)$ | $O(n)$ | 稳定 |
| 基数排序 | $O(d\times(\text{radix}+n))$ | $O(d\times(\text{radix}+n))$ | $O(\text{radix}+n)$ | 稳定 |

# 外部排序

以上讨论的排序都是内部排序。整个排序过程不设计数据的内、外存交换，待排序的记录全部存放在内存中。若待排序的文件很大，就无法将整个文件的所有记录同时调入内存中进行排序，只能将文件存放在外存中，在排序过程中需要进行多次内、外存之间的交换，这称为外部排序。外部排序也叫文件排序，通常经过两个独立的阶段完成。

1. 生成尽可能长的初始顺串：根据内存大小，每次把文件中一部分记录读入内存中，用有效的内部排序方法将其拍成有序段。每产生一个顺串，就把它写回外存中。
2. 归并阶段：每次读入一些顺串到内存中，通过合并操作，将它们合并成更长的顺串，归并操作需要频繁地对外存做读写操作。经过多次反复合并后，最终得到一个有序文件。

在一般情况下，假设待排序记录含有$m$个初始归并段，外部排序时采用k-路归并，则归并趟数$s=\lceil\log_k m\rceil$。显然，随着$k$增大，归并的趟数$s$将减少，减小$m$也将减少归并的趟数。

## 置换选择排序

如果能让每个初始归并段包含更多的记录，就能减少归并段的个数，也就是减小$m$，从而减少排序时间。但是初始归并段是通过内部排序进行的，这依赖于内存工作区的大小。下面介绍的排序能够生成平均长度为内存工作区两倍大小的初始归并段。置换选择排序(Replacement Selection Sorting)是在数选择排序的基础上得来的。其特点是：在整个排序(得到初始归并段)的过程中，选择最小(或最大)关键字和输入、输出交叉或平行进行。

假设初始待排序文件为输入文件FI，初始归并段文件为输出文件FO，内存工作区为WA，FO和WA的初始状态为空，WA可容纳$N$个记录，则置换选择排序的过程如下：

1. 从FI中输入$N$个记录到工作区WA中。
2. 从WA中选出最小关键字记录，记为MIN。
3. 将此最小关键字记录MIN输出到FO中。
4. 若FI不为空，则从FI中输入下一个记录到WA中。
5. 从所有关键字比MIN大的记录中选出最小关键字记录，作为新的记录MIN。
6. 重复步骤2～5，直到WA中选不出关键字比MIN大的记录为止。此时得到一个初始归并段，在其后加一个归并段结束标志输出到FO中。
7. 重复步骤2～6，直至WA为空，从而得到全部初始归并段。

对包含$n$个记录的输入文件进行置换选择排序，在大小为$M$的内存工作区中用堆排序生成初始归并段，每输出一个记录后调整堆的时间代价为$O(\log M)$，因此生成初始归并段的时间代价是$O(n\log M)$。当输入的记录序列已经按照关键字有序时，使用置换选择排序只能生成一个初始归并段。

## 多路归并排序

由前面的讨论，当增大$k$时，即增加归并的路数，可减少归并的趟数$s$，从而减少块读写次数，加快排序速度。

对于2-路归并，$n$个记录分布在两个归并段中，按照内部归并进行排序，要得到含有$n$个记录的归并段需要进行$n-1$次比较。而对于k-路排序，$n$个记录分布在$k$个归并段中，从$k$个记录中选出一个最小记录需要进行$k-1$次比较，一趟归并要得到全部$n$个记录共需要进行$(n-1)(k-1)$次比较。在k-路归并排序的内部归并过程中，进行总的比较次数为$s(n-1)(k-1)$。故有：

$$\lceil\log_k m\rceil(k-1)(n-1)=\left\lceil\frac{\log m}{\log k}\right\rceil(k-1)(n-1)$$

当$k$增大，$\frac{k-1}{\log k}$增大，内部归并排序时间也增加。这就抵消了因增大$k$而减少外存读写次数所节省的外部操作时间。也就是说，用上述方法增大$k$并不能减少对外存的读写次数。