排序

一、排序基础概念

排序定义：将无序序列调整为有序序列（升序/降序）
内部排序：全部数据在内存中完成（本章重点）
外部排序：数据量太大需借助外存
关键操作：比较 + 移动

```c
// 记录类型定义
typedef int KeyType;  // 关键字类型

typedef struct {
    KeyType key;      // 关键字
    // 可添加其他数据项（如：InfoType otherinfo）
} RcdType;            // 记录类型

typedef struct {
    RcdType r[1001];  // 存储空间(r[0]闲置)
    int length;        // 当前长度
} SqList;              // 顺序表
```

二、插入排序

1. 直接插入排序

思想：将无序区首个元素插入有序区合适位置，如同扑克牌排序
原理：

- 数组分为有序区（左）和无序区（右）
- 每轮取无序区首元素，反向遍历有序区找到插入点
- 插入点后元素后移，插入当前元素

代码实现：

```c
void InsertSort(SqList *L) {
    int i, j;
    for (i = 2; i <= L->length; i++) {  // 从第2个元素开始（下标1为初始有序区）
        if (L->r[i].key < L->r[i-1].key) {
            L->r[0] = L->r[i];  // 哨兵暂存待插入元素

            // 从后向前找插入位置
            for (j = i-1; L->r[0].key < L->r[j].key; j--) {
                L->r[j+1] = L->r[j];  // 元素后移
            }

            L->r[j+1] = L->r[0];  // 插入到正确位置
        }
    }
}
```

时间复杂度：

- 最优 $O(n)$（已有序）
- 最差 $O(n^2)$（逆序）
- 平均 $O(n^2)$
  稳定性：稳定

---

2. 折半插入排序

优化点：用二分查找替代顺序查找插入位置
原理：

- 在有序区使用二分法快速定位插入点
- 移动元素操作不变

代码实现：

```c
void BInsertSort(SqList *L) {
    int i, j, low, high, mid;
    for (i = 2; i <= L->length; i++) {
        L->r[0] = L->r[i];  // 哨兵
        low = 1; high = i-1;  // 二分查找区间

        while (low <= high) {  // 查找插入位置
            mid = (low + high) / 2;
            if (L->r[0].key < L->r[mid].key)
                high = mid - 1;
            else
                low = mid + 1;
        }

        // 移动元素（high+1到i-1后移）
        for (j = i-1; j >= high+1; j--) {
            L->r[j+1] = L->r[j];
        }
        L->r[high+1] = L->r[0];  // 插入
    }
}
```

时间复杂度：比较次数降至 $O(n \log n)$，移动次数仍为 $O(n^2)$
适用场景：数据量较大且比较开销大

---

3. 希尔排序

思想：分组插入排序，逐步缩小增量
原理：

- 按增量d分组（如d=5,3,1）
- 每组内进行直接插入排序
- d递减至1时整体有序

代码实现：

```c
/**
 * 希尔排序的插入函数（单次增量排序）
 * @param L 顺序表指针
 * @param dk 当前增量值（分组间隔）
 */

//增量序列还是需要注意
void ShellInsert(SqList *L, int dk) {
    int i, j;
    // 从第dk+1个元素开始，遍历所有分组（相当于对每个分组进行插入排序）
    for (i = dk + 1; i <= L->length; i++) {
        // 当前元素需要插入到分组的有序区（类似直接插入排序）
        if (L->r[i].key < L->r[i - dk].key) {  // 如果 L->r[i] 小于它前面第 dk 个元素 L->r[i - dk].key，说明逆序，需要进行插入操作。
            L->r[0] = L->r[i];  // 将待插入元素暂存到哨兵位置（下标0）

            /**
             * 在分组内寻找插入位置（按增量dk跳跃比较）
             * j = i - dk：从分组内前一个元素开始
             * j > 0：确保不越界
             * L->r[0].key < L->r[j].key：哨兵元素比当前元素小
             * j -= dk：按增量向前跳跃（在分组内向前移动）
             */
            for (j = i; j > 0 && L->r[0].key < L->r[j].key; j -= dk) {
                L->r[j ] = L->r[j-dk];  // 分组内元素后移（间隔dk位置）
            }

            // 将哨兵元素插入到正确位置
            L->r[j + dk] = L->r[0];
        }
    }
}

/**
 * 希尔排序主函数
 * @param L 顺序表指针
 * @param dlta 增量序列数组（如{5,3,1}）
 * @param t 增量序列的长度（元素个数）
 */
void ShellSort(SqList *L, int dlta[], int t) {
    // 遍历增量序列（从大到小使用每个增量）
    for (int k = 0; k < t; k++) {
        // 使用当前增量dlta[k]执行一趟插入排序
        ShellInsert(L, dlta[k]);
    }
}
```

时间复杂度：$O(n^{1.3}$至$O(n^2)$
特点：适合中等规模数据，不稳定排序

三、交换排序

1. 冒泡排序

思想：通过相邻元素交换将最大/小值"冒泡"到顶端
原理：

- 每轮遍历无序区，相邻元素比较交换
- 每轮将最大值沉底，有序区扩大

代码实现：

```
void BubbleSort(SqList *L) {
    int i = L->length;
    int lastExchange = 1;  // 记录最后一次交换位置

    while (i > 1) {  // 无序区长度>1时循环
        int bound = i;  // 无序区上界
        i = 1;  // 若未发生交换则提前结束

        for (int j = 1; j < bound; j++) {
            if (L->r[j+1].key < L->r[j].key) {
                // 交换
                RcdType tmp = L->r[j];
                L->r[j] = L->r[j+1];
                L->r[j+1] = tmp;

                i = j;  // 更新最后一次交换位置
            }
        }
    }
}
```

时间复杂度：$O(n^2)$
优化点：记录最后交换位置减少无效比较

---

2. 快速排序

思想：分治法，选取枢轴分割序列
原理：

1. 选枢轴（如首元素）
2. 一趟划分：将小于枢轴放左侧，大于放右侧
3. 递归处理左右子序列

划分过程：

```
int Partition(RcdType R[], int low, int high) {
    R[0] = R[low];        // 枢轴存入R[0]
    KeyType pivotkey = R[low].key;

    while (low < high) {
        while (low < high && R[high].key >= pivotkey)
            high--;
```

```
        R[low] = R[high];  // 小记录移左侧

        while (low < high && R[low].key <= pivotkey)
            low++;
        R[high] = R[low];  // 大记录移右侧
    }

    R[low] = R[0];  // 枢轴归位
    return low;
}
```

递归排序：

```
void QSort(RcdType R[], int s, int t) {
    if (s < t) {
        int pivotloc = Partition(R, s, t);
        QSort(R, s, pivotloc-1);  // 左子序列
        QSort(R, pivotloc+1, t);  // 右子序列
    }
}

void QuickSort(SqList *L) {
    QSort(L->r, 1, L->length);
}
```

时间复杂度：平均 $O(n \log n)$，最差 $O(n^2)$（有序时退化）
优化：三数取中法选枢轴

四、选择排序

1. 简单选择排序

思想：每轮选最小元素交换到有序区末尾
原理：

- 有序区在前，无序区在后
- 每轮遍历无序区找最小值下标
- 与无序区首元素交换

代码实现：

```
void SelectSort(SqList *L) {
    for (int i = 1; i < L->length; i++) {
        int minIdx = i;

        // 在无序区找最小值下标
        for (int j = i+1; j <= L->length; j++) {
            if (L->r[j].key < L->r[minIdx].key)
                minIdx = j;
        }

        // 交换
        if (minIdx != i) {
            RcdType tmp = L->r[i];
```

```
            L->r[i] = L->r[minIdx];
            L->r[minIdx] = tmp;
        }
    }
}
```

时间复杂度：$O(n^2)$
特点：移动次数少，不稳定

---

2. 堆排序

堆定义：完全二叉树，父节点值 ≥ 子节点值（大顶堆）
思想：

1. 建堆：从最后一个非叶节点调整
2. 交换堆顶与末尾元素
3. 调整剩余元素为新堆
4. 重复2-3至有序

关键操作：

```
// 调整以s为根的子树为大顶堆
void HeapAdjust(RcdType R[], int s, int m) {
    RcdType rc = R[s];  // 暂存根节点

    for (int j = 2*s; j <= m; j *= 2) {  // 沿较大子节点筛选
        if (j < m && R[j].key < R[j+1].key)
            j++;  // j指向较大子节点

        if (rc.key >= R[j].key) break;  // 根大于子节点，无需调整

        R[s] = R[j];  // 大孩子上移
        s = j;        // 继续向下筛选
    }

    R[s] = rc;  // 原根节点放入最终位置
}

// 堆排序主函数
void HeapSort(SqList *L) {
    // 建堆（从最后一个非叶节点开始）
    for (int i = L->length/2; i > 0; i--) {
        HeapAdjust(L->r, i, L->length);
    }

    // 逐步输出堆顶
    for (int i = L->length; i > 1; i--) {
        // 交换堆顶和末尾
        RcdType tmp = L->r[1];
        L->r[1] = L->r[i];
        L->r[i] = tmp;

        // 调整剩余元素为新堆
        HeapAdjust(L->r, 1, i-1);
```

```
    }
}
```

时间复杂度：建堆 $O(n)$，排序 $O(n\log n)$
特点：不稳定，适合大规模数据

五、归并排序

思想：分治法，两两归并有序子序列
原理：

1. 递归分割序列至单元素
2. 两两合并有序子序列

合并操作：

```c
// 合并两个有序子序列
void Merge(RcdType SR[], RcdType TR[], int i, int m, int n) {
    int j = m+1, k = i;  // j: 右序列起点, k: TR下标

    // 比较两个子序列元素
    while (i <= m && j <= n) {
        if (SR[i].key <= SR[j].key)
            TR[k++] = SR[i++];
        else
            TR[k++] = SR[j++];
    }

    // 复制剩余元素
    while (i <= m) TR[k++] = SR[i++];
    while (j <= n) TR[k++] = SR[j++];
}
```

递归排序：

```c
void MSort(RcdType SR[], RcdType TR1[], int s, int t) {
    if (s == t) {
        TR1[s] = SR[s];
    } else {
        int m = (s + t) / 2;
        RcdType TR2[1001];  // 临时数组

        MSort(SR, TR2, s, m);      // 左子序列排序
        MSort(SR, TR2, m+1, t);    // 右子序列排序
        Merge(TR2, TR1, s, m, t); // 合并
    }
}

void MergeSort(SqList *L) {
    MSort(L->r, L->r, 1, L->length);
}
```

时间复杂度：$O(n \log n)$
空间复杂度：$O(n)$（需辅助数组）
特点：稳定，适合外部排序

六、基数排序

思想：多关键字排序（LSD最低位优先）
原理：

1. 按个位分配到10个队列
2. 按十位重新分配
3. 重复至最高位
4. 收集得有序序列

数据结构：

```
// 队列节点
typedef struct node {
    RcdType data;
    struct node *next;
} Node;

// 队列结构
typedef struct {
    Node *front, *rear;
} Queue;
```

操作步骤：

1. 初始化10个队列（0-9桶）
2. 从低位到高位：
    - 分配：按当前位数字入队
    - 收集：按队列顺序链接
3. 最终链表即为有序序列

时间复杂度：$O(d(n + rd))$（d为位数，r为基数）
特点：稳定，适合多关键字排序

七、排序算法比较

算法	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
直接插入	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	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

算法	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$	稳定

> **注**：加粗算法为高效排序（$O(n\log n)$ 级别）

八、外部排序

适用场景：数据量 > 内存容量
核心过程：

1. **生成归并段**：
    - 分段读入内存
    - 内部排序后写回外存
2. **多路归并**：
    - k路归并减少趟数
    - 归并趟数 = $\lceil \log_k m \rceil$（m为初始归并段数）

时间分析：
总时间 = 内部排序时间 + I/O时间 + 归并时间
优化方向：

- 增加归并路数k
- 减少初始归并段数量m
- 优化I/O（如缓冲技术）

关键概念总结

1. **稳定性**：相等元素排序后相对位置不变（重要用于多关键字排序）
2. **时间下限**：基于比较的排序算法最低时间复杂度为 $O(n\log n)$
3. **算法选择**：
    - 小规模：插入/冒泡
    - 中等规模：希尔排序
    - 大规模：快速/堆/归并
    - 多关键字：基数排序
4. **实战建议**：

```cpp
// 快速排序模板（建议记忆）
void quick_sort(int q[], int l, int r) {
    if (l >= r) return;
    int i = l-1, j = r+1, x = q[(l+r)>>1];
    while (i < j) {
        do i++; while (q[i] < x);
        do j--; while (q[j] > x);
        if (i < j) swap(&q[i], &q[j]);
    }
    quick_sort(q, l, j);
    quick_sort(q, j+1, r);
}
```