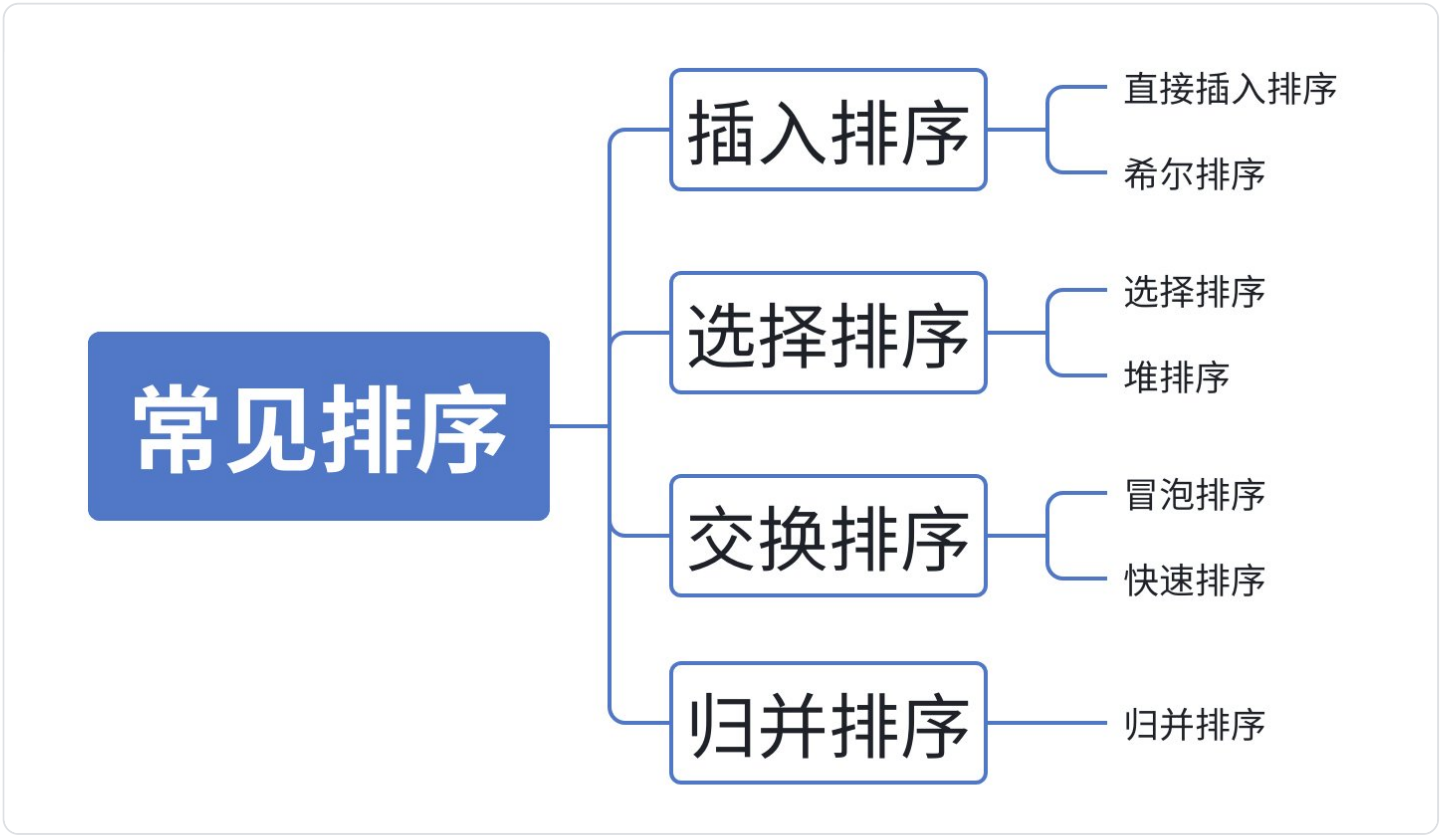
# 排序复习

## 1. 常见排序算法




## 2. 各个排序的详解

### 2.1 插入排序

#### 2.1.1 直接插入排序

基本思想是把待排序的值插入到已经排好的序列中，直到所有数一次插入得到一个完整的排序。

就如扑克牌抓拍时的思路

当插入第i(i>=1)个元素时，前面的array[0],array[1],...,array[i-1]已经排好序，此时用array[i]的排序码与 array[i-1],array[i-2],...的排序码顺序进行比较，找到插入位置即将array[i]插入，原来位置上的元素顺序后移 。

基于vector浅浅写了一版，调用了insert和erase接口，实际复杂度肯定大于O($n^2$)

代码块

```
#include<iostream>
#include<vector>
#include<algorithm>

std::vector<int> CommonArray({ 3,44,38,5,47,15,36,26,27,2,46,4,19,50,48 });

void InsertSort()
{
        // 本质是从先排序好前两个数字，然后从第三个数字不断搜寻适合它在插入的位置前面排序位置
        std::vector<int> tmpArray = CommonArray;
        if(tmpArray[0] > tmpArray[1]) std::swap(tmpArray[0], tmpArray[1]);

        for (int i = 2; i < tmpArray.size(); i++)
        {
                for (int j = 0; j <= i; j++)
                {
```

```
                        if (tmpArray[i] < tmpArray[0])
                        {
                                tmpArray.insert(tmpArray.begin(), tmpArray[i]);
                                tmpArray.erase(tmpArray.begin() + i + 1);
                                break;
                        }

                        else if (j != 0 && tmpArray[i] >= tmpArray[j - 1] &&
tmpArray[i] <= tmpArray[j])
                        {
                                tmpArray.insert(tmpArray.begin() + j,
tmpArray[i]);
                                tmpArray.erase(tmpArray.begin() + i + 1);
                                break;
                        }
                }
        }

        for (int x : tmpArray) std::cout << x << " ";
}

```

牢D写的

代码块

```
#include <iostream>
#include <vector>

void insertionSort(std::vector<int>& arr)
{
    int n = arr.size();
    for (int i = 1; i < n; ++i) {  // 从第二个元素开始
        int key = arr[i];          // 当前待插入的元素
        int j = i - 1;             // 前一个元素的索引

        // 将比 key 大的元素向后移动
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];   // 后移
            j--;
        }
        arr[j + 1] = key;         // 插入 key 到正确位置
    }
}
```

```
20
```

😍 **直接插入排序算法的时间复杂度在 O(n)~O($n^2$)， 平均在O(n), 空间复杂度 O(1), 稳定性方面为较稳定.**

### 2.1.2 希尔排序

希尔排序法又称缩小增量法。希尔排序法的基本思想是：**先选定一个整数，把待排序文件中所有记录分成几个**

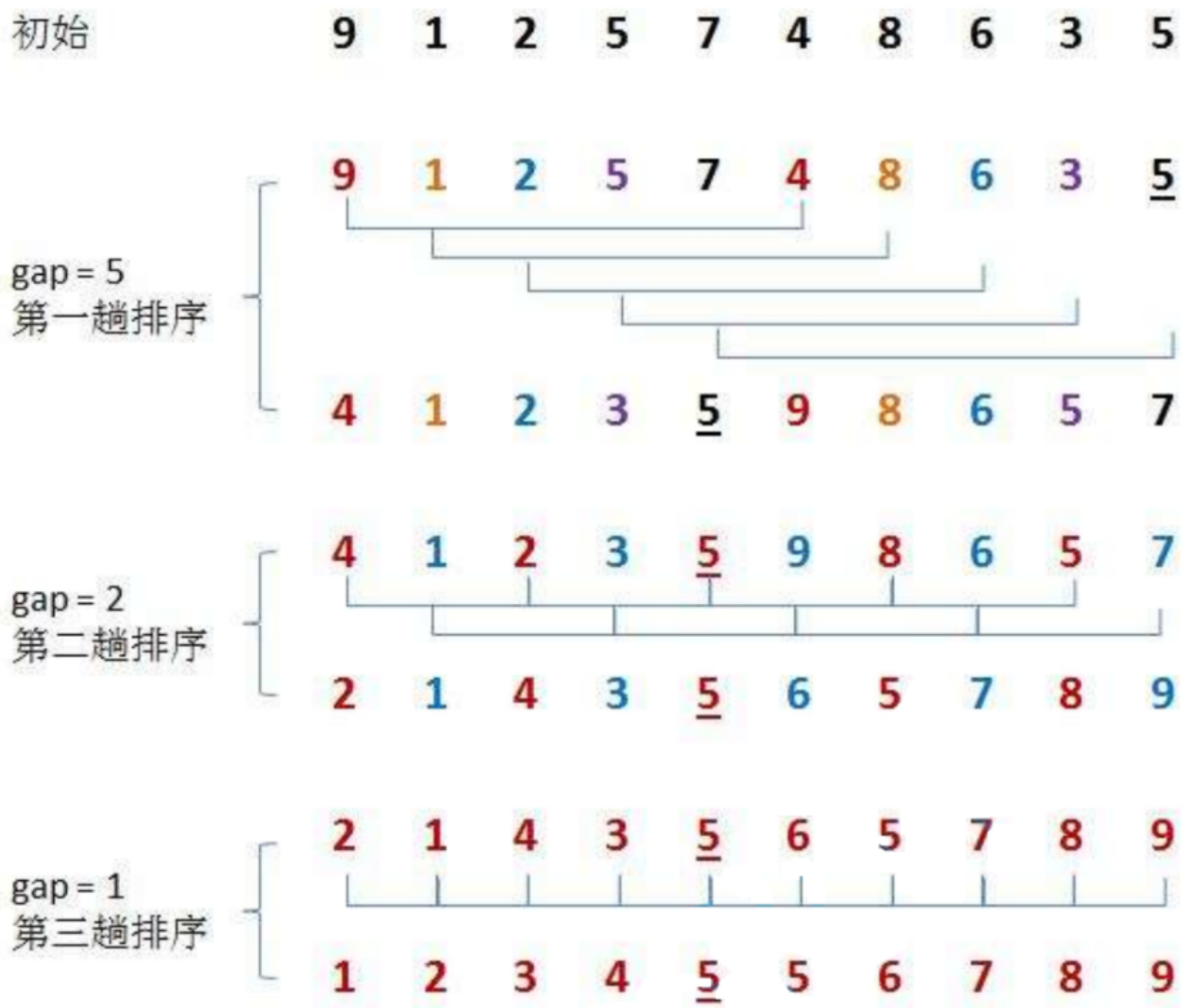
**组，所有距离为的记录分在同一组内，并对每一组内的记录进行排序。然后，取重复上述分组和排序的工 作。当到**

**达=1时，所有记录在统一组内排好序**。

说人话就是

希尔排序是**插入排序的改进版**，由 Donald Shell 于 1959 年提出。其核心思想是：

🎹
1. **分组插入排序**：将数组按一定的**增量（gap）**分成若干子序列，对每个子序列进行**直接插入排序**。
2. **逐步缩小增量**：随着排序的进行，增量不断减小，最终减至 1（即对整个数组进行一次插入排序）。
3. **利用插入排序的高效性**：由于插入排序在**部分有序**或**小规模数据**时效率较高，希尔排序通过**预处理（分组排序）**使数组趋于有序，从而减少最后的插入排序时间。




希尔排序的特性总结：

1. 希尔排序是对直接插入排序的优化。
2. 当gap > 1时都是预排序，目的是让数组更接近于有序。当gap == 1时，数组已经接近有序的了，这样就

   会很快。这样整体而言，可以达到优化的效果。我们实现后可以进行性能测试的对比。
3. 希尔排序的时间复杂度不好计算，因为gap的取值方法很多，导致很难去计算，因此在好些书中给出的

🤩 希尔排序的时间复杂度都不固定： **$O(n \log n) \sim O(n^2)$, 取决于增量序列(如gap = n/2, n/4, ...) 1时最坏, 为 $O(n^2)$）**,稳定性为**不稳定**

《数据结构(C语言版)》--- 严蔚敏

希尔排序的分析是一个复杂的问题，因为它的时间是所取“增量”序列的函数，这涉及一些数学上尚未解决的难题。因此，到目前为止尚未有人求得一种最好的增量序列，但大量的研究已得出一些局部的结论。如有人指出，当增量序列为 dlta[k] = $2^{t-k+1}-1$ 时，希尔排序的时间复杂度为 $O(n^{3/2})$，其中 $t$ 为排序趟数，$1 \leqslant k \leqslant t \leqslant \lfloor \log_2(n+1) \rfloor$。还有人在大量的实验基础上推出：当 $n$ 在某个特定范围内，希尔排序所需的比较和移动次数约为 $n^{1.3}$，当 $n \to \infty$ 时，可减少到 $n(\log_2 n)^2$[2]。增量序列可以有各种取法①，但需注意：应使增量序列中的值没有除 1 之外的公因子，并且最后一个增量值必须等于 1。

《数据结构-用面相对象方法与C++描述》--- 殷人昆

*gap* 的取法有多种。最初 Shell 提出取 $gap = \lfloor n/2 \rfloor$，$gap = \lfloor gap/2 \rfloor$，直到 $gap = 1$，后来 Knuth 提出取 $gap = \lfloor gap/3 \rfloor + 1$。还有人提出都取奇数为好，也有人提出各 *gap* 互质为好。无论哪一种主张都没有得到证明。

对希尔排序的时间复杂度的分析很困难，在特定情况下可以准确地估算关键码的比较次数和对象移动次数，但想要弄清关键码比较次数和对象移动次数与增量选择之间的依赖关系，并给出完整的数学分析，还没有人能够做到。在 Knuth 所著的《计算机程序设计技巧》第 3 卷中，利用大量的实验统计资料得出，当 $n$ 很大时，关键码平均比较次数和对象平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 范围内，这是在利用直接插入排序作为子序列排序方法的情况下得到的。

因为咋们的gap是按照Knuth提出的方式取值的，而且Knuth进行了大量的试验统计，我们暂时就按照：$O(n^{1.25})$ 到 $O(1.6 * n^{1.25})$来算。

代码块

```cpp
#include <iostream>
#include <vector>

void shellSort(std::vector<int>& arr)
{
    int n = arr.size();

    // 初始增量 gap = n/2，并逐步缩小 gap
    for (int gap = n / 2; gap > 0; gap /= 2)
    {
        // 对每个子序列进行插入排序
        for (int i = gap; i < n; i++)
        {
            int temp = arr[i];  // 当前待插入元素

            // 对子序列进行插入排序
            for (int j = i; j >= gap && arr[j - gap] > temp; j -= gap)
            {
                arr[j] = arr[j - gap];  // 较大的元素后移
```

```
                }
                arr[j] = temp;  // 插入到正确位置
            }
        }
    }

```

## 2.2 选择排序

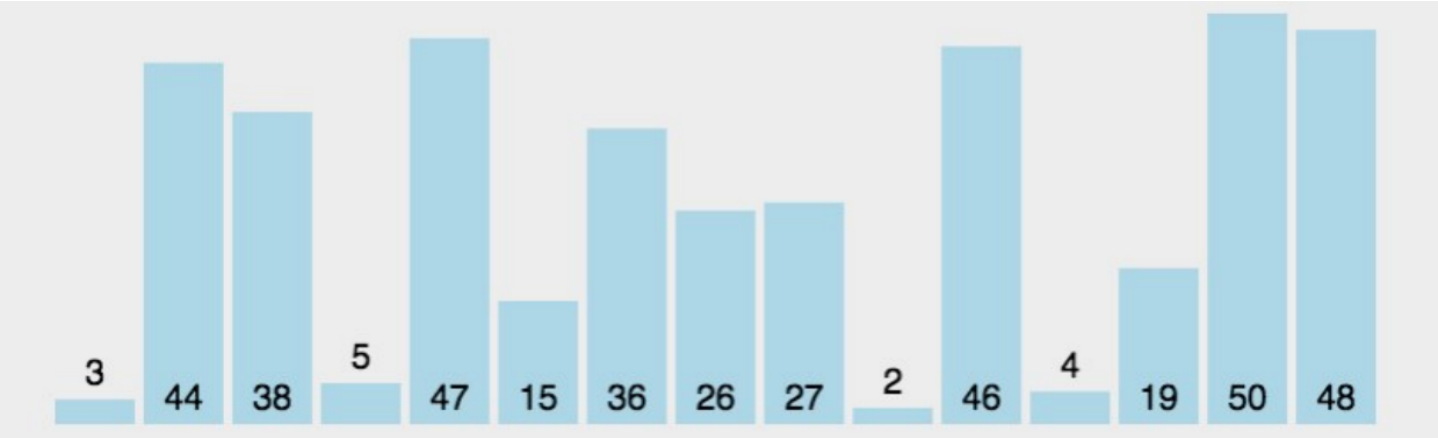
基本思想就是: 每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完 。

### 2.2.1 直接选择排序

直接选择排序是一种简单直观的排序算法，其核心思想是：

1. **从未排序部分选择最小（或最大）元素**：每次遍历未排序部分，找到最小（升序）或最大（降序）的元素。
2. **交换到已排序部分的末尾**：将选中的元素与未排序部分的第一个元素交换，使其成为已排序部分的新成员。
3. **重复直到全部有序**：不断缩小未排序部分的范围，直到所有元素排序完成。

在元素集合array[i]--array[n-1]中选择关键码最大(小)的数据元素, 若它不是这组元素中的最后一个(第一个)元素，则将它与这组元素中的最后一个（第一个）元素交换, 在剩余的array[i]--array[n-（array[i+1]--array[n-1]）集合中，重复上述步骤，直到集合剩余1个元素.




本质上就是不断的找未排序部分的最小值,然后插入到第一个位置

```cpp
#include <iostream>
#include <vector>

void selectionSort(std::vector<int>& arr)
{
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i)
    {  // 只需遍历到倒数第二个元素
        int minIndex = i;  // 假设当前未排序部分的第一个元素是最小的
        for (int j = i + 1; j < n; ++j)
        {
            // 在未排序部分中寻找最小值
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;  // 更新最小值的索引
            }
        }
        // 将最小值交换到已排序部分的末尾
        std::swap(arr[i], arr[minIndex]);
    }
}

```

😇 直接选择排序总结:

a. 思想易理解,但是效率不高,实际应用中很少使用

b. 时间复杂度: **$O(n^2)$**

c. 空间复杂度: **O(1)**

d. 稳定性：**不稳定**

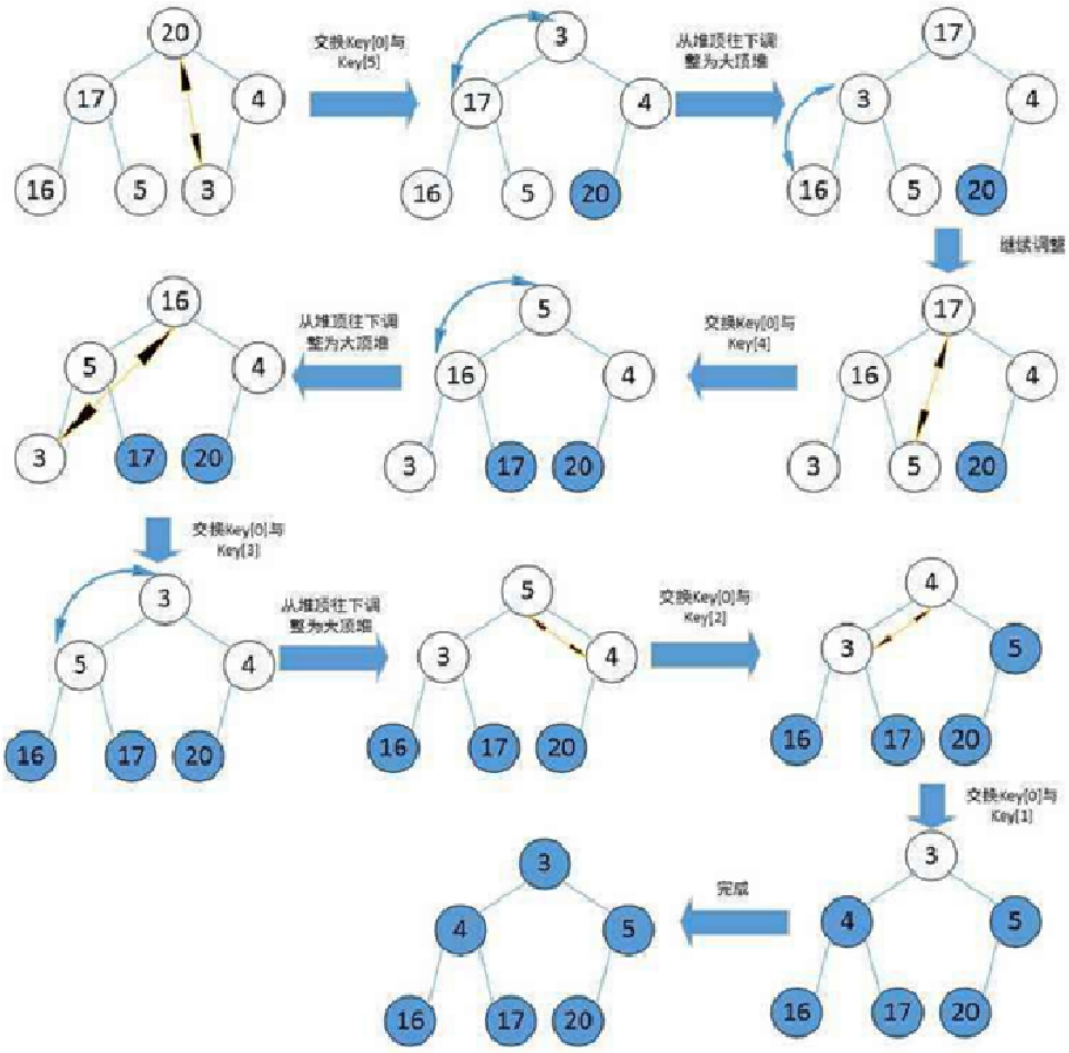
### 2.2.2 堆排序

基本思想：堆排序是一种基于**二叉堆（Binary Heap）**数据结构的排序算法，由 J. W. J. Williams 于 1964 年提出。其核心思想是：

1. **构建最大堆（或最小堆）**：
   - **最大堆**：父节点的值 ≥ 子节点的值（用于升序排序）。
   - **最小堆**：父节点的值 ≤ 子节点的值（用于降序排序）。
2. **交换堆顶与末尾元素**：
   - 将堆顶元素（最大值或最小值）与当前未排序部分的最后一个元素交换。

3. **调整剩余堆：**
    - 对剩余元素重新调整成堆结构，重复步骤 2 直到所有元素有序。

基本逻辑框图：




代码块

```
#include <iostream>
#include <vector>

// 调整堆，使其满足最大堆性质
void heapify(std::vector<int>& arr, int n, int i)
{
    int largest = i;        // 初始化当前节点为最大值
    int left = 2 * i + 1;   // 左子节点
    int right = 2 * i + 2; // 右子节点

    // 如果左子节点比当前节点大，更新最大值
```

```cpp
    if (left < n && arr[left] > arr[largest]) largest = left;

    // 如果右子节点比当前最大值大，更新最大值
    if (right < n && arr[right] > arr[largest]) largest = right;

    // 如果最大值不是当前节点，交换并递归调整
    if (largest != i)
    {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);  // 递归调整受影响的子树
    }
}

// 堆排序主函数
void heapSort(std::vector<int>& arr)
{
    int n = arr.size();

    // 构建最大堆（从最后一个非叶子节点开始）
    for (int i = n / 2 - 1; i >= 0; --i)
    {
        heapify(arr, n, i);
    }

    // 逐个提取堆顶元素并调整堆
    for (int i = n - 1; i > 0; --i)
    {
        std::swap(arr[0], arr[i]);  // 将堆顶元素（最大值）放到末尾
        heapify(arr, i, 0);         // 调整剩余堆
    }
}
```

**适用场景**

- **大规模数据**：时间复杂度 O(n log n) 优于 $O(n^2)$ 的简单排序（如选择排序）。
- **内存受限环境**：空间复杂度 O(1)，适合嵌入式系统等场景。
- **需要部分排序**：堆结构可以高效获取前 k 个最大/最小元素。

**关键点说明**

1. **为什么从 `n/2 - 1` 开始建堆？**
   - 因为叶子节点（无子节点）本身已经是合法的堆，只需从最后一个非叶子节点开始调整。
2. **为什么堆排序不稳定？**

- 例如 `[3a, 3b, 2]` 建堆时可能交换 `3a` 和 `3b`，导致相对顺序改变。

3. **与快速排序对比**：
   - 堆排序最坏情况下仍为 O(n log n)，而快排最坏 $O(n^2)$，但快排的常数因子更小，通常更快。

😜 **堆排总结：**

1. 空间复杂度：**O(n log n)**
2. 空间复杂度：**O(1)**
3. 稳定性：**不稳定**

## 2.3 交换排序

基本思想：所谓交换，就是根据序列中两个记录键值的比较结果来对换这两个记录在序列中的位置，交换排 序的特点是：将键值较大的记录向序列的尾部移动，键值较小的记录向序列的前部移动。

### 2.3.1 冒泡排序

说实话，它不配上桌。😅😅😅

😅 冒泡排序是一种简单的交换排序算法，其核心思想是：

1. **相邻元素比较交换**：从数组的第一个元素开始，依次比较相邻的两个元素，如果顺序错误（如前一个比后一个大），就交换它们。
2. **每一轮冒泡一个最大值**：每一轮遍历后，当前未排序部分的最大值会“冒泡”到数组的末尾。
3. **重复直到全部有序**：重复上述过程，直到没有任何交换发生，说明数组已经完全有序。

代码块

```
#include <iostream>
#include <vector>

void bubbleSort(std::vector<int>& arr)
{
    int n = arr.size();
```

```
    for (int i = 0; i < n - 1; ++i)
    {           // 外层循环控制轮数
        bool swapped = false;                   // 标记本轮是否发生交换
        for (int j = 0; j < n - i - 1; ++j)
        {
        // 内层循环比较相邻元素
            if (arr[j] > arr[j + 1])
            {
                // 如果前一个元素更大
                std::swap(arr[j], arr[j + 1]);  // 交换它们
                swapped = true;                 // 标记发生交换
            }
        }

        if (!swapped) break;  // 如果本轮无交换，说明数组已有序，提前终止
    }
}

```

😅 冒泡总结：

1. 时间复杂度：**O($n^2$)**
2. 空间复杂度：**O(1)**
3. 稳定性：**不稳定**

### 2.3.2 快速排序

🤪 快速排序是一种高效的**分治（Divide and Conquer）**排序算法，由 Tony Hoare 于 1959 年提出。其核心思想是：

1. **选取基准（Pivot）**：从数组中选择一个元素作为基准（通常选第一个、最后一个或随机素）。
2. **分区（Partition）**：将数组分为两部分，使得：
   - 左边部分的元素 ⩽ 基准值，
   - 右边部分的元素 ⩾ 基准值。
3. **递归排序子数组**：对左右子数组递归执行上述过程，直到子数组长度为 1 或 0（已有序）。

#### 1. Horae版本(更加高效)

代码块

```cpp
#include <cstdlib>
#include <ctime>

// 随机选择基准并交换到开头（避免最坏情况）
int choosePivot(std::vector<int>& arr, int low, int high)
 {
    int randomIdx = low + rand() % (high - low + 1);
    std::swap(arr[low], arr[randomIdx]);
    return arr[low];
}

// Hoare 分区方案（比 Lomuto 更高效）
int partitionHoare(std::vector<int>& arr, int low, int high)
{
    int pivot = choosePivot(arr, low, high);
    int i = low - 1, j = high + 1;
    while (true)
    {
        do { ++i; } while (arr[i] < pivot);
        do { --j; } while (arr[j] > pivot);
        if (i >= j) return j;
        std::swap(arr[i], arr[j]);
    }
}

void quickSortHoare(std::vector<int>& arr, int low, int high)
{
    if (low < high)
    {
        int pivotIdx = partitionHoare(arr, low, high);
        quickSortHoare(arr, low, pivotIdx);      // 注意包含 pivotIdx
        quickSortHoare(arr, pivotIdx + 1, high);
    }
}
```

## 2. Lumuto版本

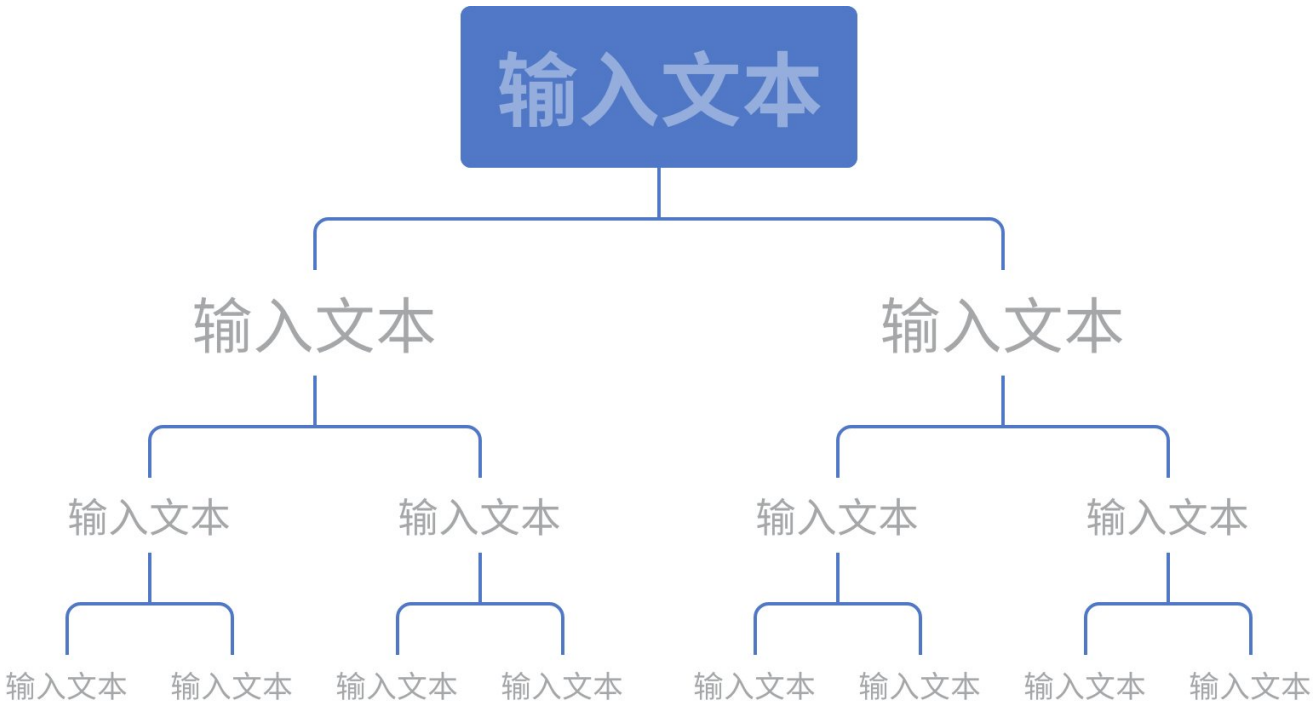
代码块

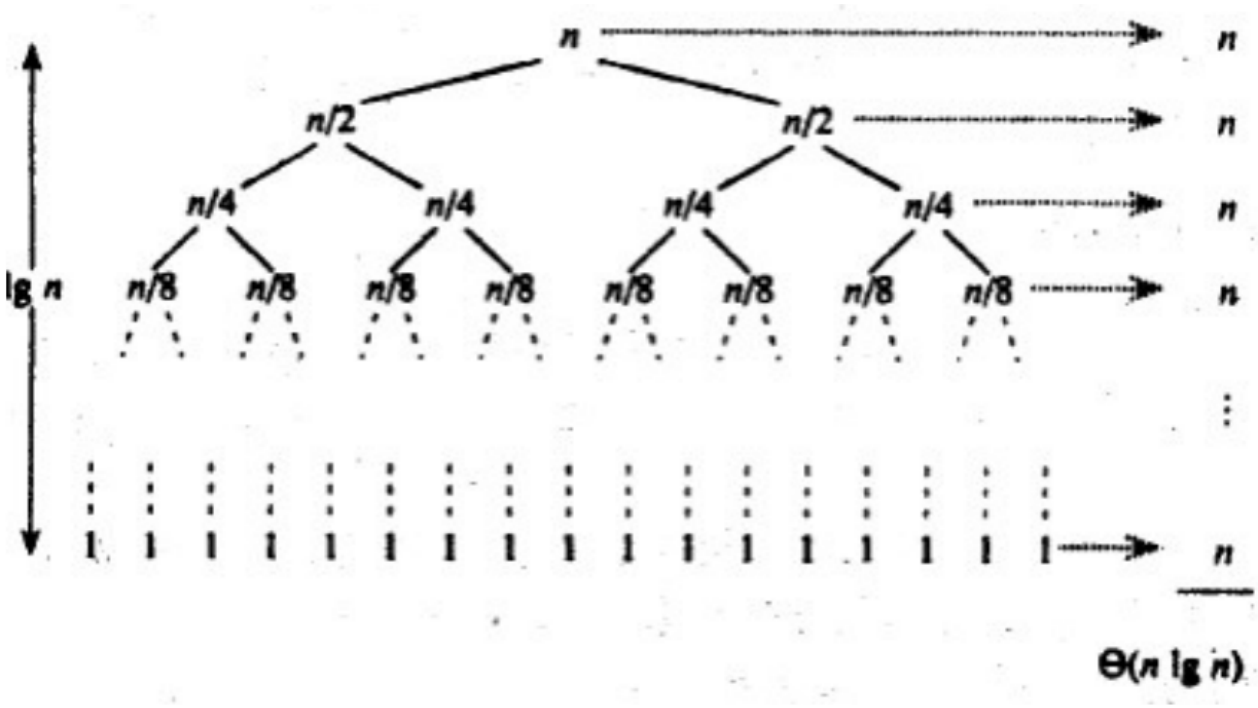
```cpp
#include <iostream>
#include <vector>

// 分区函数（Lomuto 方案，选择最后一个元素为基准）
int partition(std::vector<int>& arr, int low, int high)
{
    int pivot = arr[high];  // 选择最后一个元素为基准
```

```
    int i = low - 1;        // i 是小于基准的区域的右边界

    for (int j = low; j < high; ++j)
    {
        if (arr[j] <= pivot)
        {
            // 当前元素 ≤ 基准
            i++;
            std::swap(arr[i], arr[j]);  // 交换到左侧
        }
    }

    std::swap(arr[i + 1], arr[high]);  // 将基准放到正确位置
    return i + 1;                      // 返回基准的最终索引
}

// 快速排序主函数
void quickSort(std::vector<int>& arr, int low, int high)
{
    if (low < high)
    {
        int pivotIdx = partition(arr, low, high);  // 分区并获取基准位置
        quickSort(arr, low, pivotIdx - 1);         // 递归排序左子数组
        quickSort(arr, pivotIdx + 1, high);        // 递归排序右子数组
    }
}

```

以我自己的理解来说: 快排是分治的一个具体体现

首先可以固定或者随机找一个基准值(用于划分数组为两份的下标, 随机值可以提高避免最坏情况的概率), 利用递归的思想,不断对划分完的数组在进行划分,在递归返回时进行排序,当返回到最上层时,就是一个完整且排好序的数组.




**使用场景**

- **大规模数据**：平均性能优于归并排序和堆排序（常数因子更小）。
- **内存敏感场景**：空间复杂度优于归并排序（O(log n) vs O(n)）。
- **对稳定性无要求**：相同元素的顺序可能改变。

**关键优化**

1. **随机化基准**：避免最坏情况（如数组已有序时选固定基准导致 $O(n^2)$）。
2. **三数取中法**：选择 `low`、`mid`、`high` 的中位数作为基准，进一步平衡分区。
3. **小数组切换插入排序**：当子数组长度较小时（如 $\leqslant 15$），改用插入排序减少递归开销。

**为什么快速排序叫“快速”？**

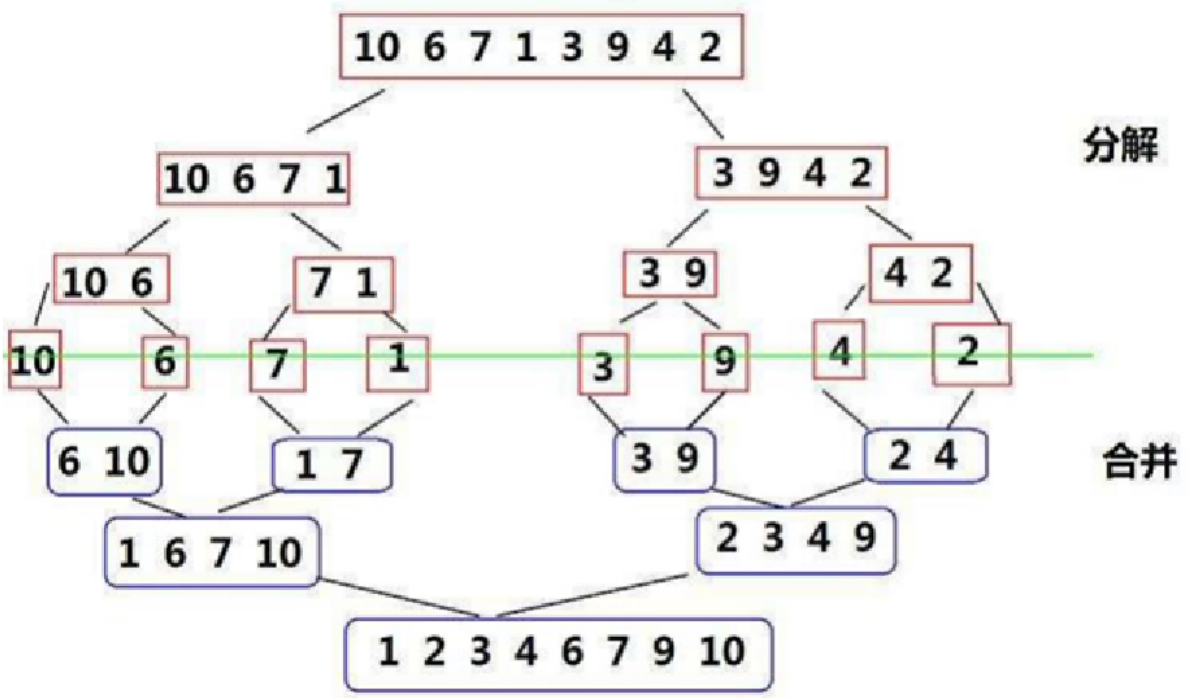
- 尽管最坏情况是 $O(n^2)$，但实际应用中：
  - 分区操作的内部循环非常高效（适合CPU缓存）。
  - 平均情况下比其他 O(n log n) 算法（如堆排序）更快。

| 😚 快排总结: | 值: | 说明: |
| --- | --- | --- |
| 1. 时间复杂度: | **O(n log n)** | **平均情况（分区平衡时）** |
| 2. 最坏情况: | **$O(n^2)$** | **当分区极度不平衡时（如数组已有序且总选第一个元素为基准）** |
| 3. 空间复杂度: | **O(log n)** | **递归栈深度（最坏 O(n)）** |
| 4. 稳定性: | **不稳定** | **分区交换可能改变相同元素的相对顺序（如 `[3a,3b,2]` → `[2,3b,3a]`）** |

## 2.4 归并排序

归并排序是一种基于**分治法（Divide and Conquer）**的高效排序算法，核心步骤如下：

1. **分解（Divide）**：
   - 将数组递归地分成两半，直到每个子数组只包含一个元素（天然有序）。
2. **合并（Merge）**：
   - 将两个已排序的子数组合并成一个有序数组，通过逐个比较元素并按顺序填充。




## 1. 递归版本(标准实现)

代码块

```cpp
#include <iostream>
#include <vector>

// 合并两个有序子数组 arr[left..mid] 和 arr[mid+1..right]
void merge(std::vector<int>& arr, int left, int mid, int right)
{
    int n1 = mid - left + 1;    // 左子数组长度
    int n2 = right - mid;       // 右子数组长度

    // 创建临时数组
    std::vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    // 合并临时数组到 arr[left..right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
```

```
            arr[k] = L[i];
            i++;
        }

        else
        {
            arr[k] = R[j];
            j++;
        }

        k++;
    }

    // 复制剩余元素
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

// 递归排序函数
void mergeSort(std::vector<int>& arr, int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;  // 避免溢出
        mergeSort(arr, left, mid);            // 排序左半部分
        mergeSort(arr, mid + 1, right);       // 排序右半部分
        merge(arr, left, mid, right);         // 合并
    }
}

```

## 2. 迭代版本(非递归,避免栈溢出)

代码块

```
#include <iostream>
#include <vector>

void mergeSortIterative(std::vector<int>& arr)
{
    int n = arr.size();
    for (int currSize = 1; currSize < n; currSize *= 2)
    {
        for (int left = 0; left < n - 1; left += 2 * currSize)
        {
            int mid = std::min(left + currSize - 1, n - 1);
```

```
            int right = std::min(left + 2 * currSize - 1, n - 1);
            merge(arr, left, mid, right);
        }
    }
}
```

## 🧐 算法分析

| 指标 | 值 | 说明 |
| --- | --- | --- |
| **时间复杂度** | **O(n log n)** | 分解和合并各需 O(log n) 层，每层合并总时间为 O(n) |
| **空间复杂度** | **O(n)** | 需要额外临时数组存储合并结果 |
| **稳定性** | **稳定** | 合并时 L[i] <= R[j] 优先取左元素，保证相同元素的相对顺序不变 |

---

### 适用场景

- **大规模数据**：时间复杂度稳定为 O(n log n)，适合外部排序（如处理磁盘大文件）。
- **需要稳定性**：如对结构体按多关键字排序时（例如先按年龄，再按姓名）。
- **链表排序**：归并排序是链表排序的最佳选择（无需随机访问，空间复杂度可优化为 O(1)）。

---

### 优化技巧

1. **小数组切换插入排序**：当子数组长度较小时（如 ⩽ 15），插入排序的常数因子更优。
2. **避免频繁内存分配**：提前分配一个全局临时数组，供所有合并操作复用。
3. **自然归并排序**：利用输入中已有的有序片段（Run）减少合并次数。

> **与其他排序对比**
>
> - **vs 快速排序**：
>   - 归并排序稳定且时间复杂度恒为 O(n log n)，但需要额外空间。
>   - 快排平均更快（常数因子小），但最坏 $O(n^2)$。
> - **vs 堆排序**：
>   - 归并排序稳定且适合外部排序，堆排序原地但不稳定。

对我而言,归并排序和将两个链表合成一个有序链表一样,不过归并得先拆分,所以需要新建两个数组(空间复杂度O(n))用于存放拆分的数组,然后一个一个比对,将值存放原数组中,最后将两个小数组中那个还有值的数组中的所有值放到原数组之后.

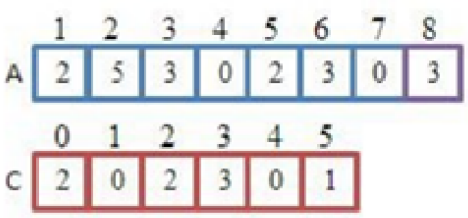
在拆分的步骤(也叫分解)可以利用迭代或者递归,有点类似于快排,对数组的不断拆分.

## 2.5 非比较排序

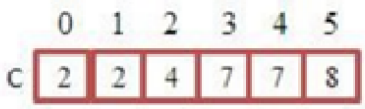
### 2.5.1 计数排序

😀 计数排序是一种**非比较型**整数排序算法，适用于数据范围较小且分布均匀的整数。其核心思想是：

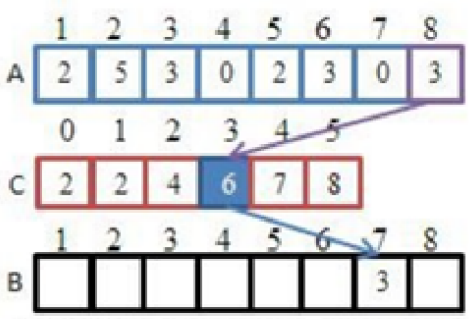
1. **统计频率**：统计每个元素出现的次数，存入计数数组。
2. **计算前缀和**：将计数数组转换为前缀和数组，确定每个元素的最终位置。
3. **反向填充**：根据前缀和数组将元素按顺序放入结果数组，保证稳定性。



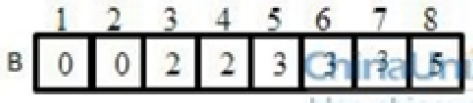

a) A 为待排序的数组。C 记录 A 中各值的元素数目。值为 0 的有 2 个元素，值为 1 的有 0 个元素，…，值为 5 的有一个元素




b) 将 C[i] 转换为值小于等于 i 的元素个数。




c) 为 A 数组从后向前的每个元素找到对应的 B 中的位置。每次从 A 中复制一个元素到 B 中，C 中相应的计数减一。




d) 当 A 中的元素都复制到 B 中后，B 就是排好序的结果。

ChinaUnix 博客
blog.chinaunix.net

代码块

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

void countingSort(std::vector<int>& arr)
{
    // 1. 找到数组中的最大值，确定计数数组大小
    int max_val = *max_element(arr.begin(), arr.end());
    std::vector<int> count(max_val + 1, 0);  // 初始化计数数组

    // 2. 统计每个元素的出现次数
    for (int num : arr)
    {
        count[num]++;
    }

    // 3. 计算前缀和（确定元素的位置）
    for (int i = 1; i <= max_val; ++i)
    {
        count[i] += count[i - 1];
    }

    // 4. 反向填充结果数组（保证稳定性）
    std::vector<int> output(arr.size());
    for (int i = arr.size() - 1; i >= 0; --i)
```

```
    {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

    // 将结果拷贝回原数组
    arr = output;
}

```

`int max_val = *max_element(arr.begin(), arr.end());` 这种写法我是第一次见

### 算法分析

| 指标 | 值 | 说明 |
| --- | --- | --- |
| **时间复杂度** | **O(n + k)** | **n** 是元素个数，**k** 是数据范围（最大元素值） |
| **空间复杂度** | **O(n + k)** | 需要计数数组和输出数组 |
| **稳定性** | **稳定** | 反向填充保证相同元素的原始顺序 |

---

> #### 适用场景
>
> - **数据范围小**：如年龄排序（0~150）、分数排序（0~100）等。
> - **整数排序**：不适用于浮点数或字符串。
> - **需要稳定性**：如对卫星数据（如对象按某整数属性排序）保持原始顺序。

#### 处理负数时,需要将整体数据集进行一个数据偏移

代码块

```
void countingSortWithNegative(std::vector<int>& arr)
{
```

```cpp
    if (arr.empty()) return;

    int min_val = *min_element(arr.begin(), arr.end());
    int max_val = *max_element(arr.begin(), arr.end());
    int range = max_val - min_val + 1;

    std::vector<int> count(range, 0);
    for (int num : arr)
    {
        count[num - min_val]++;  // 平移至非负
    }

    for (int i = 1; i < range; ++i)
    {
        count[i] += count[i - 1];
    }

    std::vector<int> output(arr.size());
    for (int i = arr.size() - 1; i >= 0; --i)
    {
        output[count[arr[i] - min_val] - 1] = arr[i];
        count[arr[i] - min_val]--;
    }

    arr = output;
}
```

> **优化技巧**
>
> 1. **缩小计数数组大小**：若已知数据范围（如成绩0~100），直接固定 `k=101`。
> 2. **原地排序**：通过交换操作减少空间复杂度至 O(k)，但会失去稳定性。
> 3. **与其他算法结合**：如基数排序的高位优先（MSD）中，用计数排序处理每一位。

---

> **局限性**
>
> - **数据范围大时效率低**：若 `k` 远大于 `n`（如 `max_val = 1e9`），空间浪费严重。
> - **仅适用于整数**：浮点数需先离散化，字符串需其他方法（如基数排序）。

# 3. 整体总结

整体排序算法复杂度及稳定性分析:


| 排序方法 | 平均情况 | 最好情况 | 最坏情况 | 辅助空间 | 稳定性 |
|---|---|---|---|---|---|
| 冒泡排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 简单选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 直接插入排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| 希尔排序 | $O(n\log n) \sim O(n^2)$ | $O(n^{1.3})$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 堆排序 | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(1)$ | 不稳定 |
| 归并排序 | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(n)$ | 稳定 |
| 快速排序 | $O(n\log n)$ | $O(n\log n)$ | $O(n^2)$ | $O(\log n) \sim O(n)$ | 不稳定 |

| 排序算法 | 平均时间复杂度 | 最坏时间复杂度 | 最好时间复杂度 | 空间复杂度 | 稳定性 |
|---|---|---|---|---|---|
| 直接插入排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 希尔排序 | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ | $O(1)$ | 不稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 堆排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | 不稳定 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 快速排序 | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ | $O(\log n)$ | 不稳定 |
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | 稳定 |

**说明：**

1. **直接插入排序**：适合小规模或部分有序的数据，稳定。
2. **希尔排序**：插入排序的改进版，通过分组排序减少逆序对，不稳定。
3. **选择排序**：每次选择最小（或最大）元素，不稳定。
4. **堆排序**：利用堆结构实现选择排序，不稳定。
5. **冒泡排序**：通过相邻元素交换实现，稳定但效率低。
6. **快速排序**：分治思想，平均性能优，但最坏情况为$O(n^2)$，不稳定。
7. **归并排序**：分治思想，稳定但需要额外空间。

**关键点：**

- **快速排序**的空间复杂度为递归栈的深度，平均$O(\log n)$，最坏$O(n)$。
- **归并排序**需要额外空间存储临时数组，因此空间复杂度为$O(n)$。

## 4. 几道选择练习题

1. 快速排序算法是基于（　　）的一个排序算法。
A 分治法
B 贪心法
C 递归法
D 动态规划法

2.对记录（54,38,96,23,15,72,60,45,83）进行从小到大的直接插入排序时，当把第8个记录45插入到有序表时，为找到插入位置需比较（ ）次？（采用从后往前比较）
A 3
B 4
C 5
D 6

3.以下排序方式中占用O（n）辅助存储空间的是
A 选择排序

B 快速排序
C 堆排序
D 归并排序

4.下列排序算法中稳定且时间复杂度为O(n2)的是（ ）
A 快速排序
B 冒泡排序
C 直接选择排序
D 归并排序

5.关于排序，下面说法不正确的是
A 快排时间复杂度为O(N*logN)，空间复杂度为O(logN)
B 归并排序是一种稳定的排序,堆排序和快排均不稳定
C 序列基本有序时，快排退化成冒泡排序，直接插入排序最快
D 归并排序空间复杂度为O(N)，堆排序空间复杂度的为O(logN)

6.下列排序法中，最坏情况下时间复杂度最小的是（ ）
A 堆排序
B 快速排序
C 希尔排序
D 冒泡排序

7.设一组初始记录关键字序列为(65,56,72,99,86,25,34,66)，则以第一个关键字65为基准而得到的一趟快速排序结果是（）
A 34, 56, 25, 65, 86, 99, 72, 66
B 25, 34, 56, 65, 99, 86, 72, 66
C 34, 56, 25, 65, 66, 99, 86, 72
D 34, 56, 25, 65, 99, 86, 72, 66

答案:
1.A
2.C
3.D
4.B
5.D
6.A
7.A