第 10 章 排序

郑新、徐鹏飞、李健

2025 年 11 月 17 日

本节内容概览

1. 祖师爷级的排序算法：插入排序、选择排序、冒泡排序
2. 插入排序的两种优化：二分插入排序、希尔排序
3. 选择排序的一种优化：堆排序
4. 冒泡排序的一种优化：快速排序
5. 结构性困境与破局者：归并排序
6. 空间换时间？我也会：计数排序、桶排序、基数排序

插入排序

```c
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j; //通过j从右向左，依次检阅左侧区间

        for(j=i-1; j>=0 && arr[j]>key; --j)
            arr[j+1] = arr[j]; // 元素右移（挪动）

        arr[j+1] = key;  //此时j=-1或a[j]<=key
    }
}
```

插入排序（注解）

- 许多人打牌时采用的排序算法，在小数据量或“近乎有序”时效率极高。

插入排序（注解）

- 许多人打牌时采用的排序算法，在小数据量或“近乎有序”时效率极高。
- 有序区的维护：从左向右，直到覆盖全体

插入排序（注解）

- 许多人打牌时采用的排序算法，在小数据量或“近乎有序”时效率极高。
- 有序区的维护：从左向右，直到覆盖全体
- 核心操作：将紧邻有序区的元素 key“插入”到合适的位置
  - key 从右向左逐个“检阅”有序区中的元素；
  - 若 key（严格）更小，则（向右）挪动被检阅的元素；
  - 否则，结束检阅，并将 key 安顿于该元素右侧。

插入排序（注解）

- 许多人打牌时采用的排序算法，在小数据量或“近乎有序”时效率极高。
- 有序区的维护：从左向右，直到覆盖全体
- 核心操作：将紧邻有序区的元素 key “插入”到合适的位置
    - key 从右向左逐个“检阅”有序区中的元素；
    - 若 key（严格）更小，则（向右）挪动被检阅的元素；
    - 否则，结束检阅，并将 key 安顿于该元素右侧。
- 稳定性：如果两个数相等，插入排序不会改变它们的相对次序。

选择排序

```c
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int k = i;  //i是位置，k指向应归位到i的元素.
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[k])
                k = j; //从未归位元素中选择最小
        int t  = arr[i];
        arr[i] = arr[k];
        arr[k] = t;
    }
}
```

选择排序（注解）

- 比较多而交换少：一轮最多一次交换。

选择排序（注解）

- 比较多而交换少：一轮最多一次交换。
- 选择排序的比较次数，与原始数据的“有序程度”无关。

选择排序（注解）

- 比较多而交换少：一轮最多一次交换。
- 选择排序的比较次数，与原始数据的“有序程度”无关。
- 核心思想：选好再换
  - 在所有“未归位”的元素中选择最小的元素归位。
  - 从左向右（从小到大）枚举归位位置。

选择排序（注解）

- 比较多而交换少：一轮最多一次交换。
- 选择排序的比较次数，与原始数据的“有序程度”无关。
- 核心思想：选好再换
  - 在所有“未归位”的元素中选择最小的元素归位。
  - 从左向右（从小到大）枚举归位位置。
- 不稳定性的来源：归位操作（将后面的小元素与前面的大元素进行交换）破坏了大元素与潜在相等元素之间的次序关系。

`Lost in swap operation!`

选择排序（注解）

- 比较多而交换少：一轮最多一次交换。
- 选择排序的比较次数，与原始数据的“有序程度”无关。
- 核心思想：选好再换
  - 在所有“未归位”的元素中选择最小的元素归位。
  - 从左向右（从小到大）枚举归位位置。
- 不稳定性的来源：归位操作（将后面的小元素与前面的大元素进行交换）破坏了大元素与潜在相等元素之间的次序关系。

`Lost in swap operation!`

两大缺陷：最好情况不够好、不稳定!

冒泡排序

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-1-i; j++) {
            if (arr[j] > arr[j+1]) {
                int t    = arr[j];
                arr[j]   = arr[j+1];
                arr[j+1] = t;
            }
        }
    }
}
```

冒泡排序（注解）

- 对内循环来说，待排序列是 $\mathtt{arr}[0, n-i)$；排序随 $i: 0 \to n-1$ 完成。

冒泡排序（注解）

- 对内循环来说，待排序列是 $\texttt{arr}[0, n-i)$；排序随 $i : 0 \to n-1$ 完成。

- 与选择排序不同，冒泡只交换相邻元素（当相邻元素为“逆序对”时）。
- 稳定性：相邻元素相等时不交换，因此保证了相等元素的相对次序不会改变。

冒泡排序（注解）

- 对内循环来说，待排序列是 $\texttt{arr}[0, n-i)$；排序随 $i: 0 \to n-1$ 完成。
- 与选择排序不同，冒泡只交换相邻元素（当相邻元素为“逆序对”时）。
- 稳定性：相邻元素相等时不交换，因此保证了相等元素的相对次序不会改变。
- 冒泡排序中的每一次交换，都精确地消除了一个逆序对。

冒泡排序（注解）

- 对内循环来说，待排序列是 $\texttt{arr}[0, n - i)$；排序随 $i : 0 \to n - 1$ 完成。

- 与选择排序不同，冒泡只交换相邻元素（当相邻元素为“逆序对”时）。
- 稳定性：相邻元素相等时不交换，因此保证了相等元素的相对次序不会改变。

- 冒泡排序中的每一次交换，都精确地消除了一个逆序对。
- 因此，交换的次数与序列有序程度有关（等于原序列中的“逆序对”个数）。

冒泡排序（注解）

- 对内循环来说，待排序列是 $\texttt{arr}[0, n-i)$；排序随 $i : 0 \to n-1$ 完成。

- 与选择排序不同，冒泡只交换相邻元素（当相邻元素为“逆序对”时）。
- 稳定性：相邻元素相等时不交换，因此保证了相等元素的相对次序不会改变。

- 冒泡排序中的每一次交换，都精确地消除了一个逆序对。
- 因此，交换的次数与序列有序程度有关（等于原序列中的“逆序对”个数）。
- 但是，比较的次数与序列有序程度无关（除非做如下的优化）。

冒泡排序（注解）

- 对内循环来说，待排序列是 `arr`$[0, n-i)$；排序随 $i: 0 \to n-1$ 完成。

- 与选择排序不同，冒泡只交换相邻元素（当相邻元素为“逆序对”时）。
- 稳定性：相邻元素相等时不交换，因此保证了相等元素的相对次序不会改变。

- 冒泡排序中的每一次交换，都精确地消除了一个逆序对。
- 因此，交换的次数与序列有序程度有关（等于原序列中的“逆序对”个数）。
- 但是，比较的次数与序列有序程度无关（除非做如下的优化）。

优化：加一个 `flag` 标记内层循环中是否发生了交换，如果没有则结束排序。
该优化可以改进冒泡排序的“最好时间复杂度”。

排序三祖：冒泡、选择、插入

教学目标

建立排序概念，理解“比较”与“交换”，掌握基础实现。

门派	算法	核心逻辑	稳定性	特点
“换”	冒泡排序	相邻交换上浮	稳定	逻辑最简单，效率最低
“选”	选择排序	全局择优放置	不稳定	交换次数少，但比较多
“挪”	**插入排序**	模拟理牌过程	稳定	**“几乎有序”时效率最高**

插入排序

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j   = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; // 元素右移（挪动）
            j--;    //从右向左检阅arr[j]
        }

        arr[j + 1] = key;
    }
}
```

冒泡排序

```c
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-1-i; j++) {
            if (arr[j] > arr[j+1]) {
                int t     = arr[j];
                arr[j]    = arr[j+1];
                arr[j+1] = t;
            }
        }
    }
}
```

选择排序

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int k = i;  //i是位置，k指向应归位到i的元素.
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[k])
                k = j; //从未归位元素中选择最小
        int t  = arr[i];
        arr[i] = arr[k];
        arr[k] = t;
    }
}
```

本节内容概览

1. 祖师爷级的排序算法：插入排序、选择排序、冒泡排序
2. 插入排序的两种优化：二分插入排序、希尔排序
3. 选择排序的一种优化：堆排序
4. 冒泡排序的一种优化：快速排序
5. 结构性困境与破局者：归并排序
6. 空间换时间？我也会：计数排序、桶排序、基数排序

二分插入排序

二分查找优化

- 插入排序在寻找插入位置的时候，没有利用左侧区间的单调性！
- 通过二分查找来寻找插入位置，就得到“二分插入排序”。

二分插入排序

二分查找优化

- 插入排序在寻找插入位置的时候，没有利用左侧区间的单调性！
- 通过二分查找来寻找插入位置，就得到“二分插入排序”。

- 比较次数：从 $O(n^2)$ 降低到 $O(n \log n)$。
- 挪动次数：不变，依然是 $O(n^2)$。
- 因此，优化后总的时间复杂度依然是 $O(n^2)$。

插入排序 (1945)

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j; //j从右向左，依次检阅左侧区间

        for(j=i-1; j>=0 && arr[j]>key; --j)
            arr[j+1] = arr[j]; // 元素右移（挪动）

        arr[j+1] = key;  //此时j=-1或a[j]<=key
    }
}
```

希尔排序 (1959)

```
void shellSort(int arr[], int n) {
    //gap 从 n/2 开始，每次折半
    for(int gap=n>>1; gap; gap>>=1){
        //gap循环内部 = (步长为gap的) 插入循环
        for(int i=gap; i<n; ++i){
            int key = arr[i], j;
            for(j=i-gap; j>=0 && arr[j]>key; j-=gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = key;
        }
    }
}
```

希尔排序 (1959)

- 按设计者 `Donald Shell` 的名字命名，任何以 1 结束的步长序列都可以工作。
  - 已知的最好步长序列是 `Sedgewick` 提出的 $(1, 5, 19, 41, 109, \ldots)$

希尔排序 (1959)

- 按设计者 `Donald Shell` 的名字命名，任何以 1 结束的步长序列都可以工作。
  - 已知的最好步长序列是 `Sedgewick` 提出的 $(1, 5, 19, 41, 109, \ldots)$
- 稳定性：不稳定。远距离的挪动会改变相等元素的相对次序。

希尔排序 (1959)

- 按设计者 Donald Shell 的名字命名，任何以 1 结束的步长序列都可以工作。
  - 已知的最好步长序列是 Sedgewick 提出的 $(1, 5, 19, 41, 109, \ldots)$
- 稳定性：不稳定。远距离的挪动会改变相等元素的相对次序。
- 时间复杂度：难以严格证明，取决于步长序列的选择。
  - 比插入排序快，甚至在小数组中比快速排序和堆排序还快；
  - 数据量大时比快速排序慢。

希尔排序的经典步长序列与复杂度分析

序列名称	生成公式（递减）	核心特征	复杂度
1. Shell	$h_k = \lfloor N/2^k \rfloor$	简单，但易产生公因子，性能不佳。	$O(N^2)$
2. Hibbard	$h_k = 2^k - 1$	理论分析最优的序列之一。	$O(N^{1.5})$
3. Pratt/Knuth	$h_k = 2^p \cdot 3^q$	**理论复杂度可被证明**，效率较高。	$O(N \log^2 N)$
4. Sedgewick	略	**实践公认最佳**，最快且最稳定。	$O(N^{4/3})$，$O(N^{7/6})$

希尔排序虽然没有严格的 $O(N \log N)$ 证明，但其优秀的实际性能和 $O(1)$ 空间复杂度，使其在工程上仍是优于 $O(N^2)$ 算法的有力选择。

本节内容概览

1. 祖师爷级的排序算法：插入排序、选择排序、冒泡排序
2. 插入排序的两种优化：二分插入排序、希尔排序
3. **选择排序的一种优化：堆排序**
4. 冒泡排序的一种优化：快速排序
5. 结构性困境与破局者：归并排序
6. 空间换时间？我也会：计数排序、桶排序、基数排序

选择排序

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int k = i;  //i是位置，k指向应归位到i的元素.
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[k])
                k = j; //从未归位元素中选择最小
        int t  = arr[i];
        arr[i] = arr[k];
        arr[k] = t;
    }
}
```

第一步：从选择排序到性能瓶颈

堆排序是选择排序的优化版本

- **选择排序的核心思想：**
  - 重复地从待排序序列中选出极值（以最大为例）元素。
  - 将其放置到已排序区的末尾。

第一步：从选择排序到性能瓶颈

堆排序是选择排序的优化版本

- **选择排序的核心思想：**
  - 重复地从待排序序列中选出极值（以最大为例）元素。
  - 将其放置到已排序区的末尾。
- **选择排序的瓶颈：**

问题所在：低效的“查找”

- 为了找到极值，每轮都需要对剩余的 N 个元素进行一次**线性扫描**。
- **时间开销：** 每轮需要 $O(N)$ 时间。
- **总复杂度：** $N \times O(N) = \mathbf{O(N^2)}$

第一步：从选择排序到性能瓶颈

堆排序是选择排序的优化版本

- **选择排序的核心思想**：
  - 重复地从待排序序列中选出极值（以最大为例）元素。
  - 将其放置到已排序区的末尾。
- **选择排序的瓶颈**：

问题所在：低效的“查找”

- 为了找到极值，每轮都需要对剩余的 N 个元素进行一次**线性扫描**。
- **时间开销**：每轮需要 $O(N)$ 时间。
- **总复杂度**： $N \times O(N) = \mathbf{O(N^2)}$

- **目标**：如何将每轮的极值查找时间从 $O(N)$ 加速到 $O(\log N)$ 或更快？

第二步：引入堆（Heap）数据结构

用数据结构将 $O(N)$ 加速到 $O(\log N)$

什么是堆？

- 堆是一种特殊的**完全二叉树**。
- **最大堆（Max-Heap）**：任何父节点的值都大于或等于其子节点的值。

第二步：引入堆（Heap）数据结构

用数据结构将 $O(N)$ 加速到 $O(\log N)$

什么是堆？

- 堆是一种特殊的**完全二叉树**。
- **最大堆（Max-Heap）**：任何父节点的值都大于或等于其子节点的值。

堆如何解决瓶颈？

加速“选择”操作

- **选择极值**：最大值永远位于**根节点**。时间开销：$O(1)$。
- **恢复结构（堆化）**：移除根节点后，通过向下调整操作恢复堆的性质。
- **恢复开销**：时间开销：$\mathbf{O}(\log \mathbf{N})$。

第三步：堆排序（Heap Sort）算法流程

利用 $O(\log N)$ 的选择进行排序

1. **建堆阶段 (Build Heap)**
   - 将待排序的数组调整为一个完全的**最大堆**。
   - 从最后一个非叶子节点开始，依次执行堆化操作。
   - **时间开销**：**O(N)**（这是最巧妙的一步）。

第三步：堆排序（Heap Sort）算法流程

利用 $O(\log N)$ 的选择进行排序

1. **建堆阶段 (Build Heap)**
   - 将待排序的数组调整为一个完全的**最大堆**。
   - 从最后一个非叶子节点开始，依次执行堆化操作。
   - **时间开销**： **O(N)**（这是最巧妙的一步）。
2. **排序阶段 (Selection/Sorting)**
   - 重复 $N-1$ 次以下操作：
   - 将堆顶元素（最大值）与堆末尾元素交换。
   - 堆的大小减一，对新的堆顶执行堆化（Heapify）。
   - **时间开销**： $N \times O(\log N) = \mathbf{O(N \log N)}$。

第三步：堆排序（Heap Sort）算法流程

利用 $O(\log N)$ 的选择进行排序

1. **建堆阶段 (Build Heap)**
   - 将待排序的数组调整为一个完全的**最大堆**。
   - 从最后一个非叶子节点开始，依次执行堆化操作。
   - **时间开销**： **O(N)**（这是最巧妙的一步）。
2. **排序阶段 (Selection/Sorting)**
   - 重复 $N-1$ 次以下操作：
   - 将堆顶元素（最大值）与堆末尾元素交换。
   - 堆的大小减一，对新的堆顶执行堆化（Heapify）。
   - **时间开销**： $N \times O(\log N) = \mathbf{O(N \log N)}$。

核心结论：效率的飞跃

$$\text{选择排序}\ (O(N^2)) \xrightarrow{\text{使用堆结构}} \textbf{堆排序}\ (O(N \log N))$$

第三步：堆排序（Heap Sort）算法流程

利用 $O(\log N)$ 的选择进行排序

1. **建堆阶段 (Build Heap)**
   - 将待排序的数组调整为一个完全的**最大堆**。
   - 从最后一个非叶子节点开始，依次执行堆化操作。
   - **时间开销**： **O(N)**（这是最巧妙的一步）。
2. **排序阶段 (Selection/Sorting)**
   - 重复 $N-1$ 次以下操作：
   - 将堆顶元素（最大值）与堆末尾元素交换。
   - 堆的大小减一，对新的堆顶执行堆化（Heapify）。
   - **时间开销**： $N \times O(\log N) = \mathbf{O(N\log N)}$。

核心结论：效率的飞跃

$$\text{选择排序}\ (O(N^2)) \xrightarrow{\text{使用堆结构}} \textbf{堆排序}\ (O(N\log N))$$

堆排序

```cpp
// 堆排序主函数 (Heap Sort)
void heapSort(int arr[], int n) {
    // 阶段一: 建堆 (从最后一个非叶子节点开始向上堆化)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // 阶段二: 排序(依次提取最大值)
    for (int i = n - 1; i > 0; i--) {
        std::swap(arr[0], arr[i]); //稳定性? 不稳定。
        heapify(arr, i, 0);        //对剩余的i个元素进行堆化
    }
}
```

堆化函数

```
void heapify(int arr[], int n, int i) {
    int best = i;
    int lson = 2*i  + 1;
    int rson = lson + 1;

    if (lson < n && arr[lson] > arr[best]) best = lson;
    if (rson < n && arr[rson] > arr[best]) best = rson;
    if (best == i) return;

    std::swap(arr[i], arr[best]);
    heapify(arr, n, best); //堆排序优雅，但不稳定。
}
```

本节内容概览

1. 祖师爷级的排序算法：插入排序、选择排序、冒泡排序
2. 插入排序的两种优化：二分插入排序、希尔排序
3. 选择排序的一种优化：堆排序
4. 冒泡排序的一种优化：快速排序
5. 结构性困境与破局者：归并排序
6. 空间换时间？我也会：计数排序、桶排序、基数排序

冒泡排序

```c
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-1-i; j++) {
            if (arr[j] > arr[j+1]) {
                int t    = arr[j];
                arr[j]   = arr[j+1];
                arr[j+1] = t;
            }
        }
    }
}
```

第一步：冒泡排序的局限性

同属【换】字诀，效率天壤之别

- **冒泡排序（Bubble Sort）的核心：**
  - 属于交换排序（Exchange Sorts，**【换】**字诀）。
  - 策略：通过**相邻元素**的不断比较和交换来消除逆序对。

第一步：冒泡排序的局限性

同属【换】字诀，效率天壤之别

- **冒泡排序（Bubble Sort）的核心：**
    - 属于交换排序（Exchange Sorts，**【换】字诀**）。
    - 策略：通过**相邻元素**的不断比较和交换来消除逆序对。
- **效率瓶颈：局部交换**

问题所在：低效的元素移动

- 元素每次只能移动一位。
- 元素移动速度太慢（“乌龟问题”）。
- 无论数据如何，需要进行 $O(N^2)$ 次比较。

第一步：冒泡排序的局限性

同属【换】字诀，效率天壤之别

- **冒泡排序（Bubble Sort）的核心：**
  - 属于交换排序（Exchange Sorts，【换】字诀）。
  - 策略：通过**相邻元素**的不断比较和交换来消除逆序对。
- **效率瓶颈：局部交换**

问题所在：低效的元素移动

- 元素每次只能移动一位。
- 元素移动速度太慢（“乌龟问题”）。
- 无论数据如何，需要进行 $O(N^2)$ 次比较。

- **目标**：如何在保持“交换”思想的同时，实现元素的大步幅、远距离移动？

第二步：快速排序的核心机制——分区

用“基准值”实现有目的的远距离交换

1. **基准值（Pivot）的选择：**
   - 从当前待排序序列中选择一个元素作为基准值 P。

第二步：快速排序的核心机制——分区

用“基准值”实现有目的的远距离交换

1. **基准值（Pivot）的选择：**
   - 从当前待排序序列中选择一个元素作为基准值 P。
2. **分区操作（Partitioning）：**
   - 通过一趟扫描，将所有小于 P 的元素交换到 P 的左侧。
   - 将所有大于 P 的元素交换到 P 的右侧。
   - 操作结束后，P 处于其最终的排序位置。

第二步：快速排序的核心机制——分区

用“基准值”实现有目的的远距离交换

1. **基准值（Pivot）的选择：**
   - 从当前待排序序列中选择一个元素作为基准值 P。
2. **分区操作（Partitioning）：**
   - 通过一趟扫描，将所有小于 P 的元素交换到 P 的左侧。
   - 将所有大于 P 的元素交换到 P 的右侧。
   - 操作结束后，P 处于其最终的排序位置。
3. **效率突破点：**

从局部交换到长距离交换

冒泡排序需要 $O(N)$ 步才能将一个元素从头移到尾。而快速排序的分区操作，可以在 $O(N)$ 的时间内，将所有元素一次性交换到 P 的两侧，实现远距离交换。

第三步：快速排序的递归与性能

平均复杂度 $O(N \log N)$ 的交换排序

- **算法流程：递归的分治法**
  1. **分治**：将序列分成两个子序列（左侧小于 P，右侧大于 P）。
  2. **递归**：对左侧子序列和右侧子序列分别递归执行分区操作。
  3. **结束**：当子序列只剩下一个或零个元素时停止。

第三步：快速排序的递归与性能

平均复杂度 $O(N\log N)$ 的交换排序

- **算法流程：递归的分治法**
  1. **分治**：将序列分成两个子序列（左侧小于 P，右侧大于 P）。
  2. **递归**：对左侧子序列和右侧子序列分别递归执行分区操作。
  3. **结束**：当子序列只剩下一个或零个元素时停止。
- **时间复杂度**：
  - **平均情况**：$\mathbf{O(N\log N)}$（性能最佳）。
  - **最坏情况**：$\mathbf{O(N^2)}$（例如，每次选择的 P 都是最大或最小元素）。

第三步：快速排序的递归与性能

平均复杂度 $O(N\log N)$ 的交换排序

- **算法流程：递归的分治法**
  1. **分治**：将序列分成两个子序列（左侧小于 P，右侧大于 P）。
  2. **递归**：对左侧子序列和右侧子序列分别递归执行分区操作。
  3. **结束**：当子序列只剩下一个或零个元素时停止。
- **时间复杂度：**
  - **平均情况**：$\mathbf{O(N\log N)}$（性能最佳）。
  - **最坏情况**：$\mathbf{O(N^2)}$（例如，每次选择的 P 都是最大或最小元素）。
- **关键性质与权衡：**

高效但有代价

- **稳定性**：不稳定（分区操作中的交换会破坏相等元素的相对次序）。
- **空间复杂度**：$O(\log N)$（递归栈空间），原地排序。

第四步：快速排序的通用优化

优化目标：避免最坏情况 $O(N^2)$

问题：当 P 总是选到极值（最大或最小）时，递归深度达到 N，导致 $O(N^2)$。

第四步：快速排序的通用优化

优化目标：避免最坏情况 $O(N^2)$

问题：当 P 总是选到极值（最大或最小）时，递归深度达到 N，导致 $O(N^2)$。

1. **随机化基准值 (Randomized Pivot)**
   - 从序列中随机选取一个元素作为 P。
   - 极大地降低遇到 $O(N^2)$ 最坏情况的概率。

第四步：快速排序的通用优化

优化目标：避免最坏情况 $O(N^2)$

问题：当 P 总是选到极值（最大或最小）时，递归深度达到 N，导致 $O(N^2)$。

1. **随机化基准值 (Randomized Pivot)**
   - 从序列中随机选取一个元素作为 P。
   - 极大地降低遇到 $O(N^2)$ 最坏情况的概率。
2. **三数取中法 (Median-of-Three)**
   - 选取序列的首、中、尾三个元素，取其中位数作为 P。
   - P 处于极端的概率更低，有效减少递归深度。

第四步：快速排序的通用优化

优化目标：避免最坏情况 $O(N^2)$

问题：当 P 总是选到极值（最大或最小）时，递归深度达到 N，导致 $O(N^2)$。

1. **随机化基准值 (Randomized Pivot)**
   - 从序列中随机选取一个元素作为 P。
   - 极大地降低遇到 $O(N^2)$ 最坏情况的概率。
2. **三数取中法 (Median-of-Three)**
   - 选取序列的首、中、尾三个元素，取其中位数作为 P。
   - P 处于极端的概率更低，有效减少递归深度。
3. **小数组优化 (Cutoff to Insertion Sort)**
   - 当子序列长度小于某一阈值（如 $10 \sim 20$）时，切换为插入排序。
   - 插入排序在小数组上常数因子更小，且消除了递归开销。

第五步：三路快排（解决重复元素问题）

问题：重复元素导致的性能下降

当序列中存在大量与 P 相等的元素时，它们会被反复与 P 比较，导致分区不平衡。

第五步：三路快排（解决重复元素问题）

问题：重复元素导致的性能下降

当序列中存在大量与 P 相等的元素时，它们会被反复与 P 比较，导致分区不平衡。

三路快排的分区机制：

- 目的：一次分区将序列分成三个独立区域。
- **分区规则（三指针）**：
  1. **小于区（$< P$）**
  2. **等于区（$= P$）**（无需再动，已归位）
  3. **大于区（$> P$）**

第五步：三路快排（解决重复元素问题）

问题：重复元素导致的性能下降

当序列中存在大量与 P 相等的元素时，它们会被反复与 P 比较，导致分区不平衡。

三路快排的分区机制：

- 目的：一次分区将序列分成三个独立区域。
- **分区规则（三指针）：**
  1. **小于区** ($< P$)
  2. **等于区** ($= P$) (无需再动，已归位)
  3. **大于区** ($> P$)

效率突破：避免对等于区的冗余递归

对于等于区 ($= P$) 的元素，我们无需再进行递归排序。当重复元素较多时，递归深度大大降低，性能接近线性 **O(N)**。

第六步：内省排序（Introsort）

快速排序的最终加固：解决 $O(N^2)$ 的隐患

核心机制：动态切换算法

1. **默认模式**：启动时使用**快速排序**（Quick Sort），利用其优秀的平均性能。
2. **递归监测**：算法实时监测递归的深度（即分治树的高度）。
3. **发现恶化**：一旦递归深度达到预设的阈值，表明可能触发了快排的最坏情况。
4. **安全切换**：算法立即切换为**堆排序**（Heap Sort）。

第六步：内省排序（Introsort）

快速排序的最终加固：解决 $O(N^2)$ 的隐患

核心机制：动态切换算法

1. **默认模式**：启动时使用**快速排序**（Quick Sort），利用其优秀的平均性能。
2. **递归监测**：算法实时监测递归的深度（即分治树的高度）。
3. **发现恶化**：一旦递归深度达到预设的阈值，表明可能触发了快排的最坏情况。
4. **安全切换**：算法立即切换为**堆排序**（Heap Sort）。

- **平均性能**：保持快速排序的极高平均速度 $\mathbf{O(N \log N)}$。
- **最坏情况**：保证最坏时间复杂度为堆排序 $\mathbf{O(N \log N)}$。

内省排序是 C++ 标准库 `std::sort` 等现代库中常用的底层实现。

快速排序 - Lomuto 分区法 (单指针，简洁)

```cpp
int lomuto(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];  // 选择最右侧元素作为基准值
    int i = low-1;          // [low,i]<pivot.
    for (int j = low; j < high; j++)
        if (arr[j] < pivot) std::swap(arr[++i], arr[j]);
    std::swap(arr[i+1], arr[high]);
    return i+1;
}
void quickSort(std::vector<int>& arr, int low, int high) {
    if (low >= high) return;
    int pivotIndex = lomuto(arr, low, high);
    quickSort(arr, low,  pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, high);
}
```

快速排序 - Hoare 分区法 (双指针、最早提出、最少交换)

```cpp
int hoare(std::vector<int>& arr, int low, int high) {
    int pivot = arr[low]; // 选择最左侧元素作为基准值
    int i=low, j=high+1;
    while(true) {
        do i++; while(i <= high && arr[i] < pivot);
        do j--; while(arr[j] > pivot);
        if (i >= j) break;
        std::swap(arr[i], arr[j]);
    }
    std::swap(arr[low], arr[j]);
    return j;
}
void quickSort(std::vector<int>& arr, int low, int high)
```

快速排序 - 三路分区法（解决重复元素）

```cpp
void threeWay(std::vector<int>& arr, int lo, int hi) {
    if (lo >= hi) return;
    int lt = lo; //[lo,lt) < pivot
    int gt = hi; //(gt,hi] > pivot
    int i  = lo+1, pivot = arr[lo];
    while (i <= gt){ //[lt,i) = pivot
        if (arr[i]==pivot){++i; continue;}
        if (arr[i]>pivot) std::swap(arr[i],  arr[gt--]);
        else              std::swap(arr[i++],arr[lt++]);
    }
    threeWay(arr, low,  lt-1);
    threeWay(arr, gt+1, high);
}
```

本节内容概览

1. 祖师爷级的排序算法：插入排序、选择排序、冒泡排序
2. 插入排序的两种优化：二分插入排序、希尔排序
3. 选择排序的一种优化：堆排序
4. 冒泡排序的一种优化：快速排序
5. 结构性困境与破局者：归并排序
6. 空间换时间？我也会：计数排序、桶排序、基数排序

传统排序算法的“两难困境”：效率与稳定性

鱼与熊掌不可兼得的权衡

什么是“两难困境”？

- **高效率目标**：尽可能快地完成排序，理想是 $O(N \log N)$ 甚至 $O(N)$。
- **稳定性目标**：保持相等元素的相对次序。

传统排序算法的“两难困境”：效率与稳定性

鱼与熊掌不可兼得的权衡

什么是“两难困境”？

- **高效率目标**：尽可能快地完成排序，理想是 $O(N \log N)$ 甚至 $O(N)$。
- **稳定性目标**：保持相等元素的相对次序。

选择一：追求稳定性（牺牲效率）

- **代表**：插入排序、冒泡排序。
- **特点**：基于相邻元素的挪动或交换。
- **结果**：确保了稳定性，但 $\mathbf{O(N^2)}$。

选择二：追求效率（牺牲稳定性）

- **代表**：希尔排序、堆排序、快速排序。
- **特点**：允许大步幅、远距离的交换。
- **结果**：$\mathbf{O(N \log N)}$，但不稳定。

归并排序（merge-sort）

$O(N \log N)$，并具有稳定性

分治经典

- **顶部视角**：将序列分成两等份，分而治之（递归调用），再将两者合并。
- **底部视角**：先将序列递归地分成最小单元，再将有序子序列逐层合并。

归并排序（merge-sort）

$O(N \log N)$，并具有稳定性

分治经典

- **顶部视角**：将序列分成两等份，分而治之（递归调用），再将两者合并。
- **底部视角**：先将序列递归地分成最小单元，再将有序子序列逐层合并。

与快排的对比：

- 快排：先分区，后递归。
- 归并：先递归，后合并。 为啥？

merge-sort 主排序函数

```cpp
void mergeSort(std::vector<int>& arr, int lo, int hi){
    if(lo >= hi) return;          //递归终止条件
    int mi = (lo+hi)/2;           //计算中点

    mergeSort(arr, lo,   mi);  //治理左侧
    mergeSort(arr, mi+1, hi);  //治理右侧

    merge(arr, lo, mi, hi);    //合并
}
```

merge-sort 的关键在“治”，即“合并”

前提：辅助空间

- 归并排序需要使用 O(N) 的额外辅助空间进行合并操作。
- 这使得它不必像快排那样进行“原地交换”，从而规避了稳定性难题。

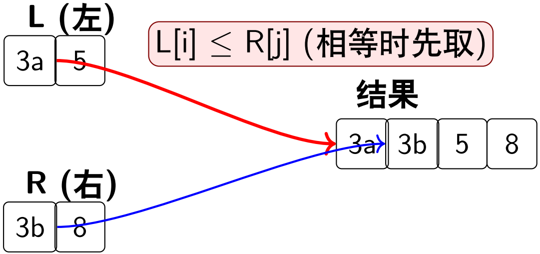
merge-sort 的关键在“治”，即“合并”

前提：辅助空间

- 归并排序需要使用 O(N) 的额外辅助空间进行合并操作。
- 这使得它不必像快排那样进行“原地交换”，从而规避了稳定性难题。

稳定性的核心规则：左侧优先

在合并两个子数组 **L** (左) 和 **R** (右) 时，若 **L** 的当前元素 **L[i]** 与 **R** 的当前元素 **R[j]** 相等，必须优先选择 **L[i]** 放入结果序列。



结论：优先选择左侧 (L[i]) 确保了 3a 永远在 3b 之前，从而保证了稳定性。

merge 函数

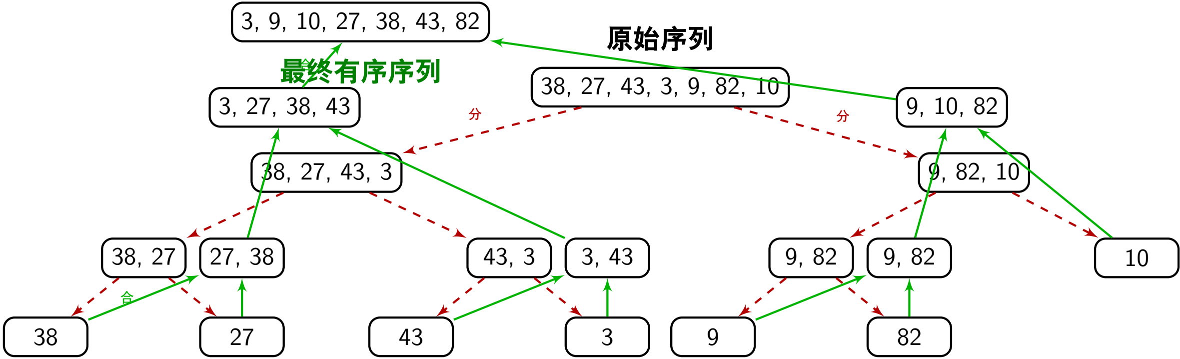
```
void merge(vector<int>& arr, int lo, int mi, int hi) {
    //初始化L,R 分别为arr[lo,mi],arr[mi+1,hi]的副本.
    vector<int> L(arr.begin()+lo,   arr.begin()+mi+1);
    vector<int> R(arr.begin()+md+1, arr.begin()+hi+1);

    int i=0, j=0, k=lo;
    while(i<L.size() && j<R.size())
        arr[k++] = L[i]<=R[j] ? L[i++] : R[j++];

    while(i<L.size()) arr[k++]=L[i++];
    while(j<R.size()) arr[k++]=R[j++];
}
```

归并排序的可视化：分而治之，再合并

一个简单的例子：[38, 27, 43, 3, 9, 82, 10]



merge-sort 的时间复杂度分析

1. **分 (Divide)：**
   - 递归地将当前序列二分成两半，直到每个子序列只包含一个元素（天然有序）。
   - **深度：** $\mathbf{O}(\log \mathbf{N})$。

merge-sort 的时间复杂度分析

1. **分 (Divide)**:
   - 递归地将当前序列二分成两半，直到每个子序列只包含一个元素（天然有序）。
   - **深度**： $\mathbf{O}(\log \mathbf{N})$。
2. **合 (Conquer / Merge)**:
   - 将两个相邻的有序子序列合并成一个新的有序序列。
   - 核心操作: 使用辅助空间进行双指针扫描和合并。
   - **广度**： 每层合并操作的总耗时是 $\mathbf{O}(\mathbf{N})$。

merge-sort 的时间复杂度分析

1. **分 (Divide)**：
   - 递归地将当前序列二分成两半，直到每个子序列只包含一个元素（天然有序）。
   - **深度**： $\mathbf{O}(\log \mathbf{N})$。
2. **合 (Conquer / Merge)**：
   - 将两个相邻的有序子序列合并成一个新的有序序列。
   - **核心操作**：使用辅助空间进行双指针扫描和合并。
   - **广度**： 每层合并操作的总耗时是 $\mathbf{O}(\mathbf{N})$。

时间复杂度：最优与最坏的一致

- **总时间复杂度**： $O(\text{深度}) \times O(\text{每层总工作量}) = O(\log N) \times O(N) = \mathbf{O}(\mathbf{N} \log \mathbf{N})$
- **特点**： 无论数据初始状态如何，时间复杂度恒定，性能非常稳定。

本节内容概览

1. 祖师爷级的排序算法：插入排序、选择排序、冒泡排序
2. 插入排序的两种优化：二分插入排序、希尔排序
3. 选择排序的一种优化：堆排序
4. 冒泡排序的一种优化：快速排序
5. 结构性困境与破局者：归并排序
6. **空间换时间？我也会：计数排序、桶排序、基数排序**

计数排序 (Counting Sort) 概览

打破 O(N log N) 限制的线性时间排序

算法定义与核心约束

- **非比较排序**：计数排序不依赖元素间的比较操作，而是利用元素的数值定位。
- **适用条件**：
  1. 输入元素必须是整数。
  2. 整数的范围 K (即 0 到 max) 必须相对较小且已知。

计数排序 (Counting Sort) 概览

打破 $O(N \log N)$ 限制的线性时间排序

算法定义与核心约束

- **非比较排序**：计数排序不依赖元素间的比较操作，而是利用元素的数值定位。
- **适用条件**：
  1. 输入元素必须是整数。
  2. 整数的范围 K (即 0 到 max) 必须相对较小且已知。

计数排序的优势与特性

- **复杂度**： $\mathbf{O(N+K)}$ (其中 N 是输入元素数量，K 是范围)。
  - ✓ 当 $K = O(N)$ 时，算法达到理想的线性时间 $\mathbf{O(N)}$。
- **稳定性**：计数排序是稳定的。

计数排序的三个关键步骤

从计数到定位 (假设输入 A 的最大值为 K)

步骤一：计数 (Counting)

- 初始化一个大小为 $K+1$ 的计数数组 **C**，用于记录每个值出现的次数。
- 遍历输入数组 A，对于每个元素 $A[i]$，执行：$\mathbf{C[A[i]]} \leftarrow \mathbf{C[A[i]]} + \mathbf{1}$。

计数排序的三个关键步骤

从计数到定位 (假设输入 A 的最大值为 K)

步骤一：计数 (Counting)

- 初始化一个大小为 $K+1$ 的计数数组 **C**，用于记录每个值出现的次数。
- 遍历输入数组 A，对于每个元素 $A[i]$，执行：$\mathbf{C[A[i]]} \leftarrow \mathbf{C[A[i]]} + \mathbf{1}$。

步骤二：累加求和 (Cumulative Sum)

- 目标：确定每个元素在最终输出数组中的**结束位置**。
- 操作：遍历 **C** 数组，对 $j = 1$ 到 K，执行：$\mathbf{C[j]} \leftarrow \mathbf{C[j]} + \mathbf{C[j-1]}$。

示例演示 (假设 $A = [2, 5, 3, 0, 2, 3, 0]$，$K = 5$)

计数排序的三个关键步骤

从计数到定位 (假设输入 A 的最大值为 K)

步骤一： 计数 (Counting)

- 初始化一个大小为 $K+1$ 的计数数组 **C**，用于记录每个值出现的次数。
- 遍历输入数组 A，对于每个元素 $A[i]$，执行： $\mathbf{C[A[i]]} \leftarrow \mathbf{C[A[i]]} + \mathbf{1}$。

步骤二： 累加求和 (Cumulative Sum)

- 目标： 确定每个元素在最终输出数组中的**结束位置**。
- 操作： 遍历 **C** 数组，对 $j=1$ 到 K，执行： $\mathbf{C[j]} \leftarrow \mathbf{C[j]} + \mathbf{C[j-1]}$。

示例演示 (假设 $A=[2,5,3,0,2,3,0]$， $K=5$)

C 数组 (Step 1 计数后):

值	0	1	2	3	4	5
C	2	0	2	2	0	1

计数排序的三个关键步骤

从计数到定位 (假设输入 A 的最大值为 K)

步骤一：计数 (Counting)

- 初始化一个大小为 $K+1$ 的计数数组 **C**，用于记录每个值出现的次数。
- 遍历输入数组 A，对于每个元素 $A[i]$，执行：$\mathbf{C[A[i]]} \leftarrow \mathbf{C[A[i]]} + \mathbf{1}$。

步骤二：累加求和 (Cumulative Sum)

- 目标：确定每个元素在最终输出数组中的**结束位置**。
- 操作：遍历 **C** 数组，对 $j=1$ 到 K，执行：$\mathbf{C[j]} \leftarrow \mathbf{C[j]} + \mathbf{C[j-1]}$。

示例演示 (假设 $A = [2, 5, 3, 0, 2, 3, 0]$，$K = 5$)

C 数组 (Step 1 计数后):

值	0	1	2	3	4	5
C	2	0	2	2	0	1

C 数组 (Step 2 累加后):

值	0	1	2	3	4	5
C	2	2	4	6	6	7

计数排序的三个关键步骤

从计数到定位 (假设输入 A 的最大值为 K)

步骤一：计数 (Counting)

- 初始化一个大小为 $K+1$ 的计数数组 **C**，用于记录每个值出现的次数。
- 遍历输入数组 A，对于每个元素 $A[i]$，执行：$\mathbf{C[A[i]]} \leftarrow \mathbf{C[A[i]]} + \mathbf{1}$。

步骤二：累加求和 (Cumulative Sum)

- 目标：确定每个元素在最终输出数组中的**结束位置**。
- 操作：遍历 **C** 数组，对 $j = 1$ 到 K，执行：$\mathbf{C[j]} \leftarrow \mathbf{C[j]} + \mathbf{C[j-1]}$。

示例演示 (假设 $A = [2, 5, 3, 0, 2, 3, 0]$，$K = 5$)

C 数组 (Step 1 计数后):

值	0	1	2	3	4	5
C	2	0	2	2	0	1

C 数组 (Step 2 累加后):

值	0	1	2	3	4	5
C	2	2	4	6	6	7

$\mathbf{C[3] = 6}$ 表示 ≤ 3 的元素共有 6 个。

步骤三：放置

逆序枚举，以保证稳定性

- 目标：将 A 中的元素按 C 数组提供的索引放入输出数组 **B** 中。
- 关键：必须从 A 的末尾向前遍历 (即 i 从 $N-1$ 到 0)。
- 对于 $A[i]$：
  1. 定位：$\text{pos} = \mathbf{C[A[i]]}$
  2. 放置：$\mathbf{B}[\text{pos} - \mathbf{1}] \leftarrow \mathbf{A[i]}$
  3. 更新：$\mathbf{C[A[i]]} \leftarrow \mathbf{C[A[i]]} - \mathbf{1}$

步骤三：放置

逆序枚举，以保证稳定性

- 目标：将 A 中的元素按 C 数组提供的索引放入输出数组 **B** 中。
- **关键**：必须从 A 的末尾向前遍历 (即 i 从 $N-1$ 到 0)。
- 对于 $A[i]$：
  1. 定位：$\text{pos} = \mathbf{C[A[i]]}$
  2. 放置：$\mathbf{B}[\text{pos} - \mathbf{1}] \leftarrow \mathbf{A[i]}$
  3. 更新：$\mathbf{C[A[i]]} \leftarrow \mathbf{C[A[i]]} - \mathbf{1}$

稳定性保证：倒序遍历 A 数组 (从右向左) 配合 C 数组的递减，确保了相等元素在 A 中的先后顺序在 B 中保持不变。

counting-sort：复杂度分析

O(N+K) 时间和 O(N+K) 空间

复杂度总结

- **时间复杂度 (Time):**
  - 步骤一 (计数): $O(N)$
  - 步骤二 (累加): $O(K)$
  - 步骤三 (放置): $O(N)$

$$\text{总时间复杂度} = \mathbf{O(N+K)}$$

- **空间复杂度 (Space):**
  - 输出数组 B: $O(N)$
  - 计数数组 C: $O(K)$

$$\text{总空间复杂度} = \mathbf{O(N+K)}$$

couting-sort

```
void countingSort(int *A, int n, int k) {
    ++k; // 0...k, 计数数组需要k+1个元素
    int *C = (int *)calloc(k,sizeof(int));//自动初始化为0
    int *B = (int *)malloc(n*sizeof(int));

    for(int i=0; i<n; i++) C[A[i]]++;
    for(int i=1; i<k; i++) C[i] += C[i-1];
    for(int i=n-1; i>=0; i--) B[--C[A[i]]] = A[i];
    for(int i=0; i<n; i++) A[i] = B[i];
    free(B);
    free(C);
}
```

桶排序 (Bucket Sort) 概览

依赖数据分布的线性时间排序

算法定义与核心机制

- **机制**：它将输入数据分配到有限数量的桶中，然后对每个桶内的数据独立排序，最后合并所有桶中的数据。
- **特点**：桶排序是结合“分治思想”和“分配思想”的一种排序算法。

桶排序 (Bucket Sort) 概览

依赖数据分布的线性时间排序

算法定义与核心机制

- **机制**：它将输入数据分配到有限数量的桶中，然后对每个桶内的数据独立排序，最后合并所有桶中的数据。
- **特点**：桶排序是结合“分治思想”和“分配思想”的一种排序算法。

关键假设

- 输入数据的分布比较均匀，使得每个桶中的元素数量大致相等。

桶排序的三个关键步骤

从分散 (Scattering) 到聚合 (Gathering)

1. **分散（Scattering）**
   - **目标**： 将输入数组 A 中的元素放入 K 个桶 $B_0, B_1, \ldots, B_{K-1}$ 中。
   - **映射**： 通常使用线性映射函数将 $A[i]$ 映射到桶的索引 j。$j = \lfloor \frac{A[i]-\text{Min}}{\text{Max}-\text{Min}} \times K \rfloor$

桶排序的三个关键步骤

从分散 (Scattering) 到聚合 (Gathering)

1. **分散（Scattering）**
   - **目标**：将输入数组 A 中的元素放入 K 个桶 $B_0, B_1, \ldots, B_{K-1}$ 中。
   - **映射**：通常使用线性映射函数将 $A[i]$ 映射到桶的索引 j。$j = \lfloor \frac{A[i]-\mathrm{Min}}{\mathrm{Max}-\mathrm{Min}} \times K \rfloor$
2. **桶内排序（Sorting Buckets）**
   - 对每个非空桶 $\mathbf{B_j}$ 中的元素进行排序。
   - 由于每个桶中的元素很少，通常使用高效的插入排序 (insertion-sort)。
   - 插入排序虽然理论时间复杂度是 $O(N^2)$，但在小规模排序问题中表现优异。

桶排序的三个关键步骤

从分散 (Scattering) 到聚合 (Gathering)

1. **分散（Scattering）**
   - **目标**：将输入数组 A 中的元素放入 K 个桶 $B_0, B_1, \ldots, B_{K-1}$ 中。
   - **映射**：通常使用线性映射函数将 $A[i]$ 映射到桶的索引 j。$j = \lfloor \frac{A[i]-\text{Min}}{\text{Max}-\text{Min}} \times K \rfloor$
2. **桶内排序（Sorting Buckets）**
   - 对每个非空桶 $\mathbf{B_j}$ 中的元素进行排序。
   - 由于每个桶中的元素很少，通常使用高效的插入排序 (insertion-sort)。
   - 插入排序虽然理论时间复杂度是 $O(N^2)$，但在小规模排序问题中表现优异。
3. **聚合（Gathering）**
   - 目标：按顺序将 $B_0, B_1, \ldots, B_{K-1}$ 中的元素取出，放入输出数组中。
   - 操作：由于桶是根据范围划分的，聚合后的数组即为最终的有序数组。

复杂度分析

性能与数据分布的强关联

- **Step 1 & 3 (分散与聚合): $\mathbf{O(N+K)}$**
- **Step 2 (桶内排序):** 设 N 个元素均匀分到 K 个桶，每个桶约有 N/K 个元素。
  桶内排序总时间 $\mathbf{T(N)} = \sum_{i=1}^{K} O((\frac{N}{K})^2)$ (插入排序)。
- 如果均匀分布，平均时间复杂度为：$\mathbf{O(N + \frac{N^2}{K} + K)}$
- **最坏情况**：若所有元素落入一个桶中，退化为桶内排序的复杂度，$\mathbf{O(N^2)}$。

复杂度分析

性能与数据分布的强关联

- **Step 1 & 3 (分散与聚合): $\mathbf{O(N+K)}$**
- **Step 2 (桶内排序):** 设 N 个元素均匀分到 K 个桶，每个桶约有 N/K 个元素。
  桶内排序总时间 $\mathbf{T(N)} = \sum_{i=1}^{K} O((\frac{N}{K})^2)$ (插入排序)。
- 如果均匀分布，平均时间复杂度为：$\mathbf{O(N + \frac{N^2}{K} + K)}$
- **最坏情况**：若所有元素落入一个桶中，退化为桶内排序的复杂度，$\mathbf{O(N^2)}$。

空间与稳定性

- **空间复杂度: $\mathbf{O(N+K)}$** (用于 N 个元素的存储和 K 个桶的结构)。
- **稳定性**：如果桶内使用稳定的排序算法，则桶排序是稳定的。

桶排序：常见问题解答 (FAQ)

聚焦复杂度和算法选择的考点

问：为什么桶内排序通常不使用归并排序？

桶排序：常见问题解答 (FAQ)

聚焦复杂度和算法选择的考点

问：为什么桶内排序通常不使用归并排序？

- **桶内规模**：在数据均匀分布情况下，每个桶中的元素数量 $\mathbf{N_i}$ 非常小 ($\approx N/K$)。
- **插入排序优势**：
  - 插入排序具有极低的常数因子和 $O(1)$ 额外空间的优势。
  - 当桶内元素数 $\approx N/K \leq 10$ 时，插入排序的 $O(N^2)$ 劣势被大大掩盖。

桶排序：常见问题解答 (FAQ)

聚焦复杂度和算法选择的考点

问：为什么桶内排序通常不使用归并排序？

- **桶内规模**：在数据均匀分布情况下，每个桶中的元素数量 $\mathbf{N_i}$ 非常小 ($\approx N/K$)。
- **插入排序优势**：
  - 插入排序具有极低的常数因子和 $O(1)$ 额外空间的优势。
  - 当桶内元素数 $\approx N/K \leq 10$ 时，插入排序的 $O(N^2)$ 劣势被大大掩盖。
- **归并排序劣势**：
  - 递归开销高：对微小数组进行递归调用会增加不必要的系统开销。
  - 内存开销大：归并排序的合并操作需要 $\mathbf{O(N_i)}$ 的辅助空间。对每个小桶都进行内存分配和复制，总体的内存管理开销会非常高，抵消了其渐进复杂度优势。

基数排序演示

待排数列：$\{170, 045, 075, 090, 802, 024\}$

待排数位 ↓	排序后的序列 ↓
0, **5**, **5**, **0**, **2**, **4**	17**0**, 09**0**, 80**2**, 02**4**, 04**5**, 07**5**
7, **9**, **0**, **2**, **4**, **7**	8**0**2, 0**2**4, 0**4**5, 1**7**0, 0**7**5, 0**9**0
8, **0**, **0**, **1**, **0**, **0**	**0**24, **0**45, **0**75, **0**90, **1**70, **8**02

最终结果：$\{024, 045, 075, 090, 170, 802\}$

基数排序演示

待排数列：$\{170, 045, 075, 090, 802, 024\}$

待排数位 ↓	排序后的序列 ↓
0, 5, 5, 0, 2, 4	17**0**, 09**0**, 80**2**, 02**4**, 04**5**, 07**5**
7, 9, 0, 2, 4, 7	8**0**2, 0**2**4, 0**4**5, 1**7**0, 0**7**5, 0**9**0
8, 0, 0, 1, 0, 0	**0**24, **0**45, **0**75, **0**90, **1**70, **8**02

最终结果：$\{024, 045, 075, 090, 170, 802\}$

注意：在第 2 轮中，170 必须保持在 075 之前，
直到第 3 轮才会被正确交换位置。

基数排序 (Radix Sort) 概览

基数排序：用多轮的稳定排序（如计数排序）代替单轮排序

- **核心思想**：从低到高对每个数位（Digit）进行多轮的稳定排序。
- **基数定义**：基数 **B** 是指该数位可能拥有的不同值的数量（类似进制的底数）。
  - 例如：处理十进制数时，$\mathbf{B} = \mathbf{10}$ (有 0-9 共 10 种可能值)。

基数排序 (Radix Sort) 概览

基数排序：用多轮的稳定排序（如计数排序）代替单轮排序

- **核心思想**：从低到高对每个数位（Digit）进行多轮的稳定排序。
- **基数定义**：基数 **B** 是指该数位可能拥有的不同值的数量（类似进制的底数）。
  - 例如：处理十进制数时，$\mathbf{B} = \mathbf{10}$ (有 0-9 共 10 种可能值)。

关键步骤与稳定排序

- **位数**：确定待排对象在基数 **B** 下的最大数位 D。
- **机制**：迭代 D 轮，每轮用一个稳定排序算法（如计数排序）处理当前位。
- **稳定**：稳定，才能确保低位排序产生的顺序不会被后续的高位排序破坏。

LSD 基数排序：步骤与机制

Least Significant Digit (LSD)：从个位开始

1. **确定位数** D：根据最大值或预先知道的数据范围在基数 **B** 下的位数。

LSD 基数排序：步骤与机制

Least Significant Digit (LSD)：从个位开始

1. **确定位数** D：根据最大值或预先知道的数据范围在基数 **B** 下的位数。
2. **迭代** D **次**：从最低位（个位 $d = 1$）开始，向最高位（$d = D$）进行 D 次循环。

LSD 基数排序：步骤与机制

Least Significant Digit (LSD)：从个位开始

1. **确定位数** D：根据最大值或预先知道的数据范围在基数 **B** 下的位数。
2. **迭代** D **次**：从最低位（个位 $d=1$）开始，向最高位（$d=D$）进行 D 次循环。
3. **稳定排序**：每次循环中，以当前位 d 的值为键对序列执行稳定排序。

LSD 基数排序：步骤与机制

Least Significant Digit (LSD)：从个位开始

1. **确定位数** D：根据最大值或预先知道的数据范围在基数 **B** 下的位数。
2. **迭代 D 次**：从最低位（个位 $d = 1$）开始，向最高位（$d = D$）进行 D 次循环。
3. **稳定排序**：每次循环中，以当前位 d 的值为键对序列执行稳定排序。

稳定性的核心作用：维护低位已建立的正确相对顺序

输入序列：$L = \{\mathbf{38}, \mathbf{31}\}$ (最终正确顺序：$\{31, 38\}$)

1. **第 1 轮 (个位 8 vs 1):** 个位值不同，排序结果为 $\{\mathbf{31}, \mathbf{38}\}$。

LSD 基数排序：步骤与机制

Least Significant Digit (LSD)：从个位开始

1. **确定位数** D：根据最大值或预先知道的数据范围在基数 **B** 下的位数。
2. **迭代** D **次**：从最低位（个位 $d = 1$）开始，向最高位（$d = D$）进行 D 次循环。
3. **稳定排序**：每次循环中，**以当前位 d 的值**为键对序列执行稳定排序。

稳定性的核心作用：维护低位已建立的正确相对顺序

输入序列：$L = \{\mathbf{38}, \mathbf{31}\}$ (最终正确顺序：$\{31, 38\}$)

1. **第 1 轮 (个位 8 vs 1):** 个位值不同，排序结果为 $\{\mathbf{31}, \mathbf{38}\}$。
2. **第 2 轮 (十位 3 vs 3): 十位相同 (3=3)**，它们的相对顺序由前一轮决定。
   - ✓ **稳定排序**： 能够维护 **31** 在 **38** 之前的顺序。 结果：$\{\mathbf{31}, \mathbf{38}\}$
   - ✗ **不稳定排序**：可能会破坏之前的相对顺序，导致结果是 $\{\mathbf{38}, \mathbf{31}\}$

LSD 基数排序：步骤与机制

Least Significant Digit (LSD)：从个位开始

1. **确定位数** D：根据最大值或预先知道的数据范围在基数 **B** 下的位数。
2. **迭代** D **次**：从最低位（个位 $d=1$）开始，向最高位（$d=D$）进行 D 次循环。
3. **稳定排序**：每次循环中，**以当前位 d 的值**为键对序列执行稳定排序。

稳定性的核心作用：维护低位已建立的正确相对顺序

输入序列： $L = \{\mathbf{38}, \mathbf{31}\}$ (最终正确顺序：$\{31, 38\}$)

1. **第 1 轮 (个位 8 vs 1):** 个位值不同，排序结果为 $\{\mathbf{31}, \mathbf{38}\}$。
2. **第 2 轮 (十位 3 vs 3): 十位相同 (3=3)**，它们的相对顺序由前一轮决定。
   - ✓ **稳定排序**： 能够维护 31 在 38 之前的顺序。 结果：$\{\mathbf{31}, \mathbf{38}\}$
   - ✗ **不稳定排序**：可能会破坏之前的相对顺序，导致结果是 $\{\mathbf{38}, \mathbf{31}\}$

结论：对于基数排序来说，稳定性是必须的。

复杂度分析与变体

D 次 $O(N+B)$ 的稳定排序

- B：排序所基于的基数 (Base)，通常 $B=10$ 或 $B=2^k$ (计算机中常取 256)。
- D：数字的最大位数 (Max Digits)。
- N：输入元素的总数。

$$\text{总时间复杂度} = \mathbf{O(D \cdot (N+B))}$$

复杂度分析与变体

D 次 $O(N+B)$ 的稳定排序

- B：排序所基于的基数 (Base)，通常 $B=10$ 或 $B=2^k$ (计算机中常取 256)。
- D：数字的最大位数 (Max Digits)。
- N：输入元素的总数。

$$\text{总时间复杂度} = \mathbf{O(D\cdot(N+B))}$$

MSD 变体 (Most Significant Digit)

- 从最高位开始排序，将数据分成 B 个桶，然后递归地对每个桶进行排序。
- 处理字符串或位数差异很大的数字时更高效（无需处理尾部大量的 0 补齐）。
- 而 LSD 不用递归，实现简单，内存管理高效，因此在整数排序中更常用。

radix-sort-byte

```cpp
void radixSortByte(std::vector<int>& arr) {
    const int B = 256;
    for(int shift = 0; shift < 32; shift += 8){
        std::vector<int> cnt(B, 0);
        std::vector<int> aux(arr.size());
        for (int x : arr) cnt[(x >> shift) & 0xFF]++;
        for (int i=1; i<B; i++) cnt[i] += cnt[i-1];
        for (int i = arr.size()-1; i>=0; i--) {
            int v = (arr[i] >> shift) & 0xFF;
            aux[--cnt[v]] = arr[i];
        }
        arr = aux;
    }
}
```

总结：排序算法全景概览 (Cheat Sheet)

阶段	算法名称	核心逻辑	时间复杂度	空间复杂度	稳定性	教学关键点
1. 基础	冒泡排序	相邻交换	$O(n^2)$	$O(1)$	稳定	理解复杂度与交换
	选择排序	寻找最值	$O(n^2)$	$O(1)$	不稳定	减少了交换次数
	插入排序	维护有序区	$O(n^2)$	$O(1)$	稳定	**小数据量/几近有序时极快**
2. 进阶	**归并排序**	分治、递归	$O(n\log n)$	$O(n)$	稳定	空间换时间，典型分治
	快速排序	分区 (Partition)	$O(n\log n)$	$O(\log n)$	不稳定	**工业界最常用的基础**
3. 结构	堆排序	利用二叉堆	$O(n\log n)$	$O(1)$	不稳定	适合 TopK 问题，原地排序
4. 线性	计数/基数	统计与映射	$O(n+k)$	$O(k)$	稳定	突破比较排序的极限

第一阶段：直觉与基础 ($O(n^2)$)

教学目标

建立排序概念，理解“比较”与“交换”，掌握基础实现。

算法	**核心逻辑**	**稳定性**	**特点**
冒泡排序	相邻交换上浮	稳定	逻辑最简单，效率最低
选择排序	全局择优放置	不稳定	交换次数最少，但比较多
插入排序	模拟理牌过程	稳定	**几乎有序时效率最高**

第二阶段：分治法的革命 ($O(n\log n)$)

教学目标

突破性能瓶颈，理解递归、分治思想及 $O(n\log n)$ 的由来。

算法	核心操作	空间	稳定性	关键痛点/优势
归并排序	Merge (合并)	$O(n)$	稳定	稳定但耗内存
快速排序	Partition (分区)	$O(\log n)$	不稳定	平均最快，但需防最坏情况
堆排序	Heapify (堆化)	$O(1)$	不稳定	节省空间，常用于系统底层

第三阶段：突破理论极限（线性时间）

思考

如果不进行“元素间的比较”，还能排序吗？

- **计数排序 (Counting Sort)**:
  - 统计每个数出现的频次。
  - 适用于：量大但范围小（如考分统计）。
- **基数排序 (Radix Sort)**:
  - 按位（个位、十位...）进行桶分配。
  - 适用于：长整数、字符串。

结论：利用数据的自身特征，可以达到 $O(n)$ 的速度。

欢迎提问！