数据结构与算法

Data Structure and Algorithm

极夜酱

目录

I 基础篇 **1**

1 排序算法 **2**

1.1 排序算法 2
1.2 冒泡排序 4
1.3 选择排序 9
1.4 插入排序 12
1.5 归并排序 15
1.6 快速排序 17
1.7 计数排序 21
1.8 桶排序 23

Part I

基础篇

Chapter 1 排序算法

1.1 排序算法

1.1.1 排序算法

应用到排序的常见比比皆是，例如当开发一个学生管理系统时需要按照学号从小到大进行排序，当开发一个电商平台时需要把同类商品按价格从低到高进行排序，当开发一款游戏时需要按照游戏得分从多到少进行排序。

根据时间复杂度的不同，主流的排序算法可以分为三类：

1. $O(n^2)$：冒泡排序、选择排序、插入排序
2. $O(nlogn)$：归并排序、快速排序、堆排序
3. $O(n)$：计数排序、桶排序、基数排序

在算法界还存在着更多五花八门的排序，它们有些基于传统排序变形而来，有些则是脑洞大开，如鸡尾酒排序、猴子排序、睡眠排序等。

例如睡眠排序，对于待排序数组中的每一个元素，都开启一个线程，元素值是多少，就让线程睡多少毫秒。当这些线程陆续醒来的时候，睡得少的线程线性来，睡得多的线程后醒来。睡眠排序虽然挺有意思，但是没有任何实际价值。启动大量线程的资源消耗姑且不说，数值接近的元素也未必能按顺序输出，而且一旦遇到很大的元素，线程睡眠时间可能超过一个月。

1.1.2 稳定性

排序算法还可以根据其稳定性，划分为稳定排序和不稳定排序：

- 稳定排序：值相同的元素在排序后仍然保持着排序前的顺序。
- 不稳定排序：值相同的元素在排序后打乱了排序前的顺序。

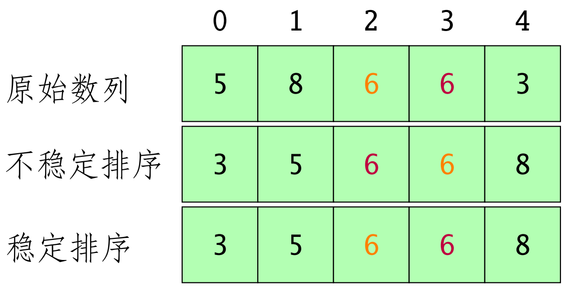


图 1.1: 排序稳定性

1.2 冒泡排序

1.2.1 冒泡排序（Bubble Sort）

冒泡排序是最基础的交换排序。冒泡排序之所以叫冒泡排序，正是因为这种排序算法的每一个元素都可以像小气泡一样，根据自身大小，一点一点向着数组的一侧移动。

按照冒泡排序的思想，要把相邻的元素两两比较，当一个元素大于右侧相邻元素时，交换它们的位置；当一个元素小于或等于右侧相邻元素时，位置不变。

例如一个有 8 个数字组成的无序序列，进行升序排序。

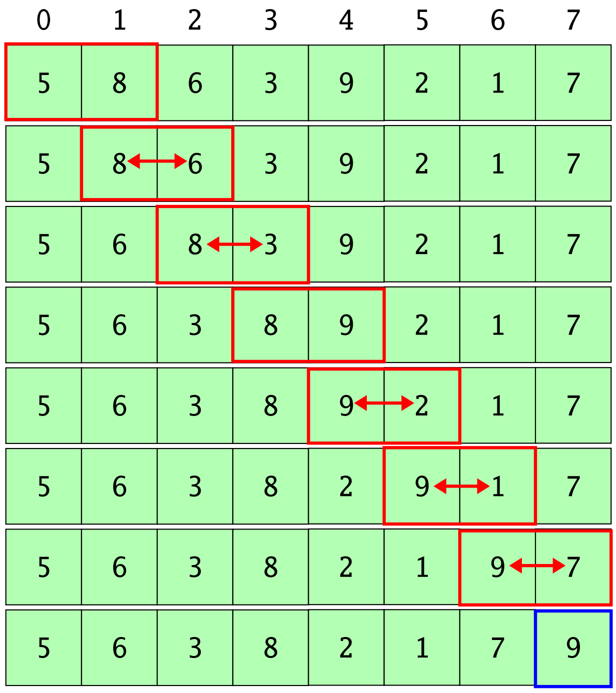


图 1.2: 冒泡排序第 1 轮

这样一来，元素 9 作为数列中最大的元素，就像是汽水里的小气泡一样，浮到了最右侧。这时，冒泡排序的第 1 轮就结束了。数列最右侧元素 9 的位置可以认为是一个有序区域，有序区域目前只有 1 个元素。

接着进行第 2 轮排序：

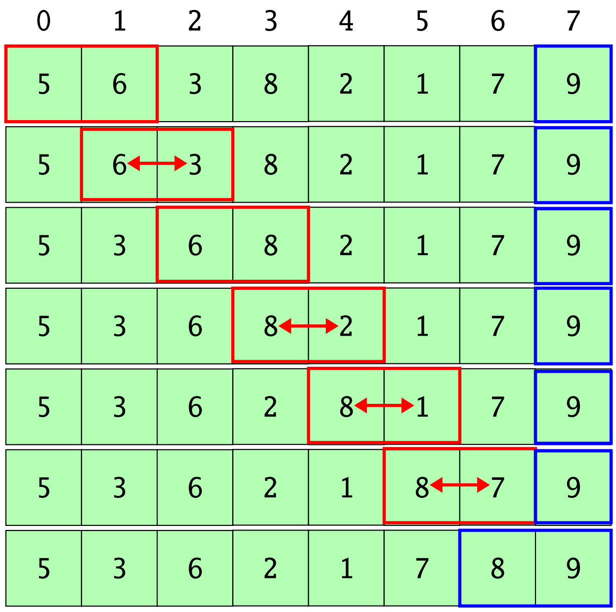


图 1.3: 冒泡排序第 2 轮

第 2 轮排序结束后，数列右侧的有序区有了 2 个元素。

根据相同的方法，完成剩下的排序：

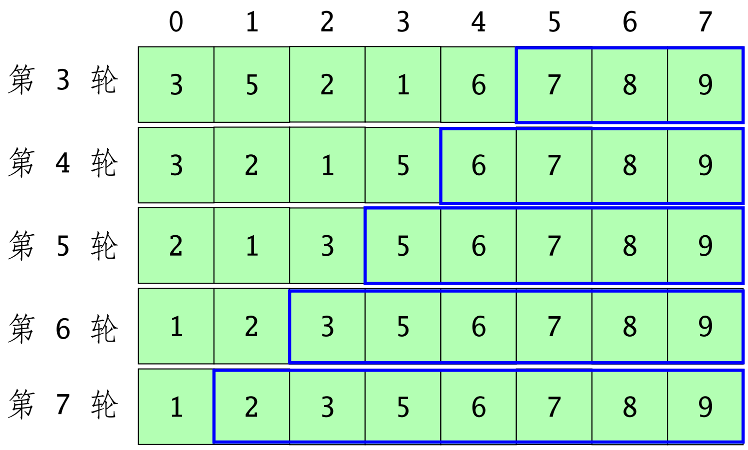


图 1.4: 冒泡排序第 3 ~ 7 轮

1.2.2 算法分析

冒泡排序是一种稳定排序，值相等的元素并不会打乱原本的顺序。由于该排序算法的每一轮都要遍历所有元素，总共遍历 n - 1 轮。

时间复杂度	**空间复杂度**	**稳定性**	**设计思想**
$O(n^2)$	$O(1)$	稳定	贪心法

表 1.1: 冒泡排序算法分析

冒泡排序

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

逆序对

假设数组有 n 个元素，如果 A[i] > A[j], i < j，那么 A[i] 和 A[j] 就被称为逆序对（inversion）。

```
int countInversions(int *arr, int n) {
    int cnt = 0;        // 逆序对数
    for(int i = 0; i < n-1; i++) {
        for(int j = i+1; j < n; j++) {
            if(arr[i] > arr[j]) {
                cnt++;
            }
        }
    }
    return cnt;
}
```

1.2.3 冒泡排序第一次优化

常规的冒泡排序需要进行 n - 1 轮循环，即使在中途数组已经有序，但是还是会继续剩下的循环。例如当数组是 {2, 1, 3, 4, 5} 时，在经过一轮排序后已经变为有序状态，再进行多余的循环就会浪费时间。

为了解决这个问题，可以在每一轮循环中设置一个标志。如果该轮循环中有元素发生过交换，那么就有必要进行下一轮循环。如果没有发生过交换，说明当前数组已经完成排序。

冒泡排序第一次优化

```c
void bubbleSortOptimize1(int *arr, int n) {
    for(int i = 0; i < n - 1; i++) {
        bool isSorted = false;  // 标记是否发生交换
        for(int j = 0; j < n - i - 1; j++) {
            if(arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                isSorted = true;    // 发生交换
            }
        }
        // 该轮未发生交换，已经有序
        if(!isSorted) {
            return;
        }
    }
}
```

1.2.4 冒泡排序第二次优化

在经过一次优化后，算法还存在一个问题，例如数组 {2, 3, 1, 4, 5, 6} 在经过一轮交换后变为 {2, 1, 3, 4, 5, 6}，但是在下一轮时后面有很多次比较都是多余的，因为并没有产生交换操作。

为了解决这个问题，可以再设置一个标志位，用于记录当前轮所交换的最后一个元素的下标。在下一轮排序中，只需比较到该标志位即可，因此之后的元素在上一轮中没有交换过，在这一轮中也不可能交换了。

冒泡排序第二次优化

```
void bubbleSortOptimize2(int *arr, int n) {
    int len = n - 1;        // 内层循环执行次数
    for(int i = 0; i < n - 1; i++) {
        bool flag = false;  // 标记是否发生交换
        int last = 0;       // 标记最后一次发生交换的位置
        for(int j = 0; j < len; j++) {
            if(arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                flag = true;    // 发生交换
                last = j;
            }
        }
        // 该轮未发生交换，已经有序
        if(!flag) {
            return;
        }
        len = last;         // 最后一次发生交换的位置
    }
}
```

1.3 选择排序

1.3.1 选择排序（Selection Sort）

有了冒泡排序为什么还要发明选择排序？冒泡排序有个很大的弊端，就是元素交换次数太多了。

想象一个场景，假设你是一名体育老师，正在指挥一群小学生按照个头从矮到高的顺序排队。采用冒泡排序的方法需要频繁交换相邻学生的位置，同学们心里恐怕会想："这体育老师是不是有毛病啊？"

在程序运行的世界里，虽然计算机并不会产生什么负面情绪，但是频繁的数组元素交换意味着更多的内存读写操作，严重影响了代码运行效率。

有一个简单的办法，就是每一次找到个子最矮的学生，直接交换到队伍的前面。

例如一个有 8 个数字组成的无序序列，进行升序排序。

	0	1	2	3	4	5	6	7
原数组	5	8	6	3	9	2	1	7
第 1 轮	1	8	6	3	9	2	5	7
第 2 轮	1	2	6	3	9	8	5	7
第 3 轮	1	2	3	6	9	8	5	7
第 4 轮	1	2	3	5	9	8	6	7
第 5 轮	1	2	3	5	6	8	9	7
第 6 轮	1	2	3	5	6	7	9	8
第 7 轮	1	2	3	5	6	7	8	9

图 1.5: 选择排序

1.3.2 算法分析

算法每一轮选出最小值，再交换到左侧的时间复杂度是 $O(n)$，一共迭代 n - 1 轮，总的时间复杂度是 $O(n^2)$。

由于算法所做的是原地排序，并没有利用额外的数据结构，所以空间复杂度是 $O(1)$。

时间复杂度	空间复杂度	稳定性	设计思想
$O(n^2)$	$O(1)$	不稳定	减治法

表 1.2: 选择排序算法分析

选择排序

```
void selectionSort(int *arr, int n) {
    for(int i = 0; i < n-1; i++) {
        int minIndex = i;
        for(int j = i+1; j < n; j++) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        if(i != minIndex) {
            swap(&arr[i], &arr[minIndex]);
        }
    }
}
```

1.3.3 选择排序优化

选择排序的整体思想是在一个序列当中选出一个最小的元素，和第一个元素交换，然后在剩下的找最小的，和第二个元素交换。这样最终就可以得到一个有序序列。但是为了更加高效，可以每次选择出一个最小值和一个最大值，分别放在序列的最左和最右边。

选择排序优化

```c
void selectionSortOptimize(int *arr, int n) {
    int left = 0;
    int right = n - 1;
    while(left < right) {
        int min = left;
        int max = right;
        for(int i = left; i <= right; i++) {
            if(arr[i] < arr[min]) {
                min = i;
            }
            if(arr[i] > arr[max]) {
                max = i;
            }
        }
        swap(&arr[max], &arr[right]);
        // 考虑特殊情况，最小值在最右位置
        if(min == right) {
            min = max;
        }
        swap(&arr[min], &arr[left]);
        left++;
        right--;
    }
}
```

1.4 插入排序

1.4.1 插入排序（Insertion Sort）

如何对扑克牌进行排序呢？例如现在手上有红桃 6, 7, 9, 10 这四张牌，已经处于升序排序状态。这时候抓到了一张红桃 8，如何让手上的五张牌重新变成升序呢？

使用冒泡排序？选择排序？恐怕正常人打牌的时候都不会那么做。最自然最简单的方式，是在已经有序的四张牌中找到红桃 8 应该插入的位置，也就是 7 和 9 之间，把红桃 8 插入进去。

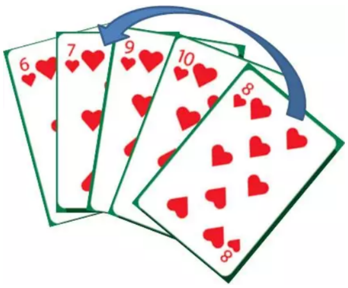


图 1.6: 理牌

例如一个有 8 个数字组成的无序序列，进行升序排序。

	0	1	2	3	4	5	6	7
原数组	5	8	6	3	9	2	1	7
第 1 轮	5	8	6	3	9	2	1	7
第 2 轮	5	6	8	3	9	2	1	7
第 3 轮	3	5	6	8	9	2	1	7
第 4 轮	3	5	6	8	9	2	1	7
第 5 轮	2	3	5	6	8	9	1	7
第 6 轮	1	2	3	5	6	8	9	7
第 7 轮	1	2	3	5	6	7	8	9

图 1.7: 插入排序

1.4.2 算法分析

插入排序要进行 n - 1 轮，每一轮在最坏情况下的比较复制次数分别是 1 次、2 次、3 次、4 次... 一直到 n - 1 次，所以最坏时间复杂度是 $O(n^2)$。

至于空间复杂度，由于插入排序是在原地进行排序，并没有引入额外的数据结构，所以空间复杂度是 $O(1)$。

时间复杂度	空间复杂度	稳定性	设计思想
$O(n^2)$	$O(1)$	稳定	减治法

表 1.3: 插入排序算法分析

插入排序

```c
void insertionSort(int *arr, int n) {
    for(int i = 1; i < n; i++) {
        int temp = arr[i];
        int j = i - 1;
        while(j >= 0 && temp < arr[j]) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp;
    }
}
```

1.4.3 折半插入排序（Binary Insertion Sort）

折半插入排序是对插入排序的改进，其过程就是不断依次将元素插入前面已经排好序的序列中，在寻找插入点时采用了折半查找。

折半插入排序

```c
void binaryInsertionSort(int *arr, int n) {
```

```
    for(int i = 1; i < n; i++) {
        int temp = arr[i];
        int start = 0;
        int end = i - 1;
        while(start <= end) {
            int mid = start + (end - start) / 2;
            if(arr[mid] > temp) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        int j;
        for(j = i - 1; j > end; j--) {
            arr[j+1] = arr[j];
        }
        arr[j+1] = temp;
    }
}
```

1.5 归并排序

1.5.1 归并排序（Merge Sort）

归并排序算法采用分治法：

1. 分解：将序列每次折半划分。
2. 合并：将划分后的序列两两按序合并。

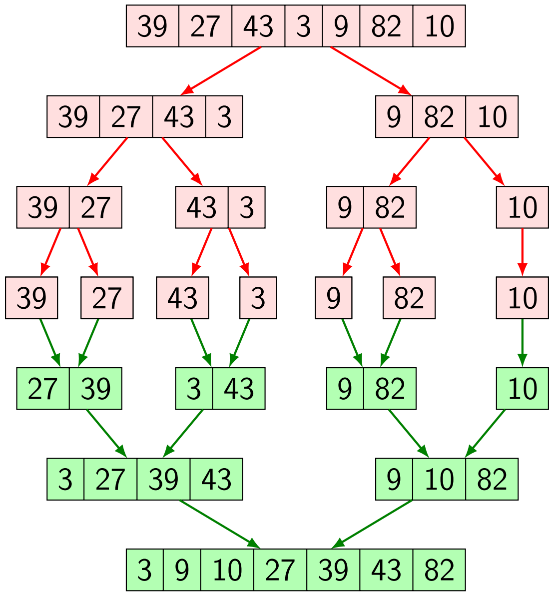


图 1.8: 归并排序

1.5.2 算法分析

归并排序每次将数组折半对分，一共分了 $logn$ 次，每一层进行合并操作的运算量是 n，所以时间复杂度为 $O(nlogn)$。归并排序的速度仅次于快速排序。

时间复杂度	**空间复杂度**	**稳定性**	**设计思想**
$O(nlogn)$	$O(n)$	稳定	分治法

表 1.4: 归并排序算法分析

归并排序

```c
void merge(int *arr, int start, int mid, int end, int *temp) {
    int i = start;
    int j = mid + 1;
    int k = 0;

    while(i <= mid && j <= end) {
        if(arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while(i <= mid) {
        temp[k++] = arr[i++];
    }
    while(j <= end) {
        temp[k++] = arr[j++];
    }

    for(int i = 0; i < k; i++) {
        arr[start+i] = temp[i];
    }
}

void mergeSort(int *arr, int start, int end, int *temp) {
    if(start < end) {
        int mid = start + (end - start) / 2;
        mergeSort(arr, start, mid, temp);
        mergeSort(arr, mid+1, end, temp);
        merge(arr, start, mid, end, temp);
    }
}
```

1.6 快速排序

1.6.1 快速排序（Quick Sort）

快速排序是很重要的算法，与傅里叶变换等算法并称二十世纪十大算法。

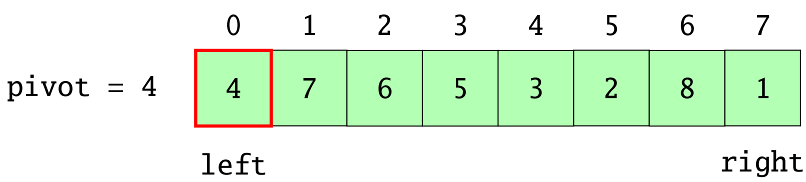
快速排序之所以快，是因为它使用了分治法。快速排序在每一轮挑选一个基准（pivot）元素，并让其它比它小的元素移动到数列一边，比它大的元素移动到数列的另一边，从而把数列拆解成了两个部分。

选择基准元素最简单的方式是选择数列的第一个元素。这种选择在绝大多数情况下是没有问题的，但是如果对一个原本逆序的数列进行升序排序，整个数列并没有被分成一半，每一轮仅仅确定了基准元素的位置。这种情况下数列第一个元素要么是最小值，要么是最大值，根本无法发挥分治法的优势。在这种极端情况下，快速排序需要进行 n 轮，时间复杂度退化成了 $O(n^2)$。

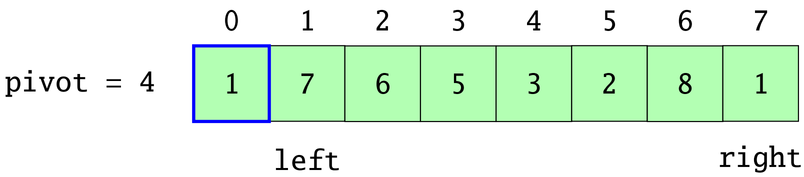
如何避免这种极端情况呢？可以不选择数列的第一个元素，而是随机选择一个元素作为基准元素。这样一来，即使是在数列完全逆序的情况下，也可以有效地将数列分成两部分。当然，即使是随机选择，每一次也有极小的几率选到数列的最大值或最小值，同样会对分治造成一定影响。

确定了基准值后，如何实现将小于基准的元素都移动到基准值一边，大于基准值的都移动到另一边呢？

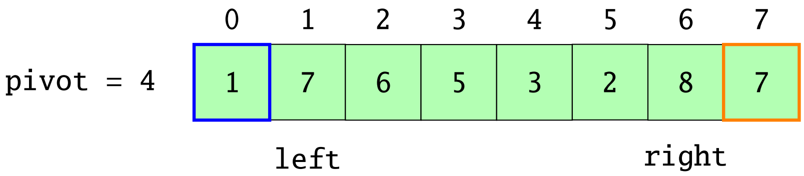
例如一个有 8 个数字组成的无序序列，进行升序排序。选定基准元素 pivot，设置两个指针 left 和 right，指向数列的最左和最右两个元素。



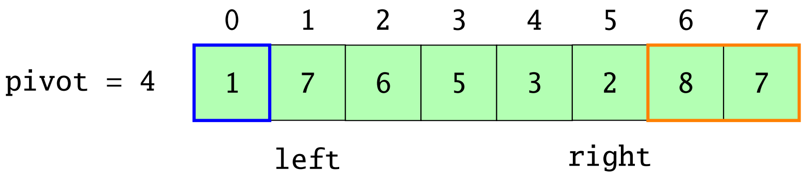
从 right 指针开始，把指针所指向的元素和基准元素做比较。如果比 pivot 大，则 right 指针向左移动；如果比 pivot 小，则把 right 所指向的元素填入 left 指针所指向的位置，同时 left 向右移动一位。

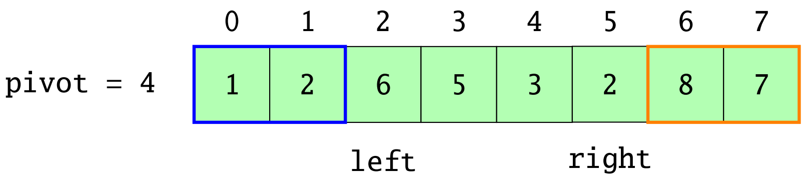


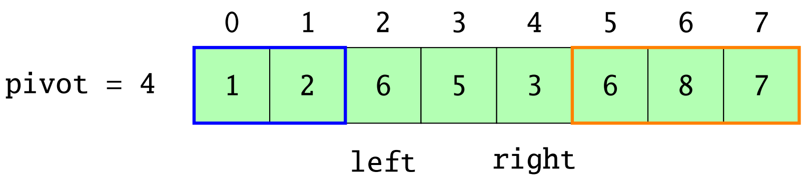
接着，切换到 left 指针进行比较，把指针所指向的元素和基准元素做比较。如果小于 pivot，则 left 指针向右移动；如果大于 pivot，则把 left 所指向的元素填入 right 指针所指向的位置，同时 right 向左移动一位。

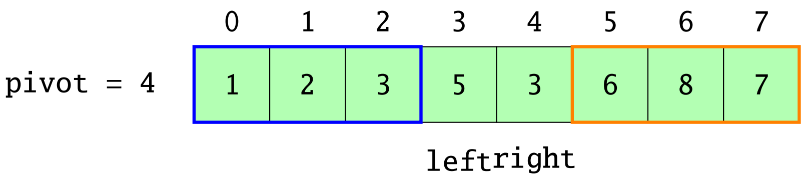


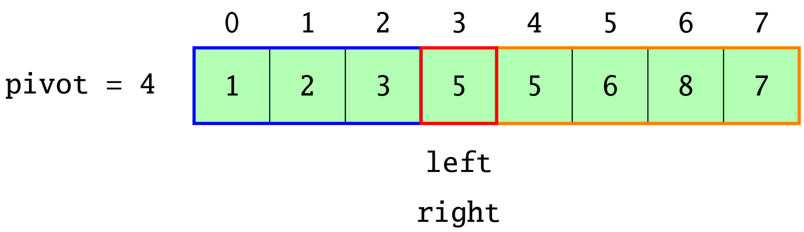
重复之前的步骤继续排序：











当 left 和 right 指针重合在同一位置的时候，把之前的 pivot 元素的值填入该重合的位置。此时数列左边的元素都小于基准元素，数列右边的元素都大于基准元素。

1.6.2 算法分析

分治法的思想下，原数列在每一轮被拆分成两部分，每一部分在下一轮又被拆分成两部分，直到不可再分为止。这样平均情况下需要 $logn$ 轮，因此快速排序算法的平均时间复杂度是 $O(nlogn)$。

时间复杂度	空间复杂度	稳定性	设计思想
$O(nlogn) \sim O(n^2)$	$O(logn) \sim O(n)$	不稳定	分治法

表 1.5: 快速排序算法分析

快速排序

```
void quickSort(int *arr, int start, int end) {
    if(start < end) {
        int i = start;
        int j = end;
        int pivot = arr[start];

        while(i < j) {
            while(i < j && arr[j] > pivot) {
                j--;
            }
            if(i < j) {
                arr[i] = arr[j];
                i++;
            }
            while(i < j && arr[i] < pivot) {
                i++;
            }
            if(i < j) {
                arr[j] = arr[i];
                j--;
```

```
            }
        }
        arr[i] = pivot;
        quickSort(arr, start, i-1);
        quickSort(arr, i+1, end);
    }
}
```

1.7 计数排序

1.7.1 计数排序（Counting Sort）

基于比较的排序算法的最优下界为 $\Omega(nlogn)$。计数排序是一种不基于比较的排序算法，而是利用数组下标来确定元素的正确位置。

遍历数列，将每一个整数按照其值对号入座，对应数组下标的元素加 1。数组的每一个下标位置的值，代表了数列中对应整数出现的次数。有了这个统计结果，直接遍历数组，输出数组元素的下标值，元素的值是多少就输出多少次。

从功能角度，这个算法可以实现整数的排序，但是也存在一些问题。如果只以最大值来决定统计数组的长度并不严谨，例如数列 {95, 94, 91, 98, 99, 90, 99, 93, 91, 92}，这个数列的最大值是 99，但最小值是 90。如果创建长度为 100 的数组，前面的从 0 到 89 的空间位置都浪费了。

因此，不应再以数列的 max + 1 作为统计数组的长度，而是以数列 max - min + 1 作为统计数组的长度。同时，数列的最小值作为一个偏移量，用于统计数组的对号入座。

计数排序适用于一定范围的整数排序，在取值范围不是很大的情况下，它的性能甚至快过那些 $O(nlogn)$ 的排序算法。

计数排序

```
void countingSort(int *arr, int n) {
    int max = arr[0];
    int min = arr[0];
    for(int i = 1; i < n; i++) {
        if(arr[i] > max) {
            max = arr[i];
        }
        if(arr[i] < min) {
```

```
            min = arr[i];
        }
    }

    int range = max - min + 1;
    int table[range];
    memset(table, 0, sizeof(table));

    for(int i = 0; i < n; i++) {
        table[arr[i] - min]++;
    }

    int cnt = 0;
    for(int i = 0; i < range; i++) {
        while(table[i]--) {
            arr[cnt++] = i + min;
        }
    }
}
```

1.8 桶排序

1.8.1 桶排序（Bucket Sort）

桶排序是计数排序的扩展版本。计数排序可以看成每个桶只存储相同元素，而桶排序每个桶存储一定范围的元素。

每一个桶代表一个区间范围，里面可以承载一个或多个元素。通过划分多个范围相同的区间，将每个子区间自排序，最后合并。桶排序需要尽量保证元素分散均匀，否则当所有数据集中在同一个桶中时，桶排序失效。