5 排序

1. 排序概念及运用

1.1 概念

排序：所谓排序，就是使一串记录，按照其中的某个或某些关键字的大小，递增或递减的排列起来的操作。

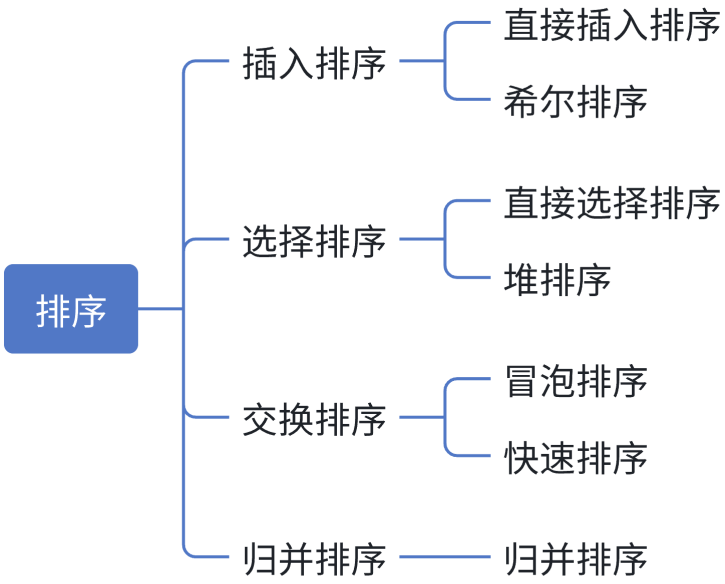
1.2 运用

购物筛选排序

院校排名

排名	院校名称	总分	类型	所在地	批次	历年排名
1	北京大学	100	综合类	北京	本科一批	
2	清华大学	99.58	理工类	北京	本科一批	
3	浙江大学	82.56	综合类	浙江	本科一批	
4	复旦大学	82.17	综合类	上海	本科一批	
5	中国人民大学	81.51	综合类	北京	本科一批	
6	上海交通大学	81.5	综合类	上海	本科一批	
7	武汉大学	81.49	综合类	湖北	本科一批	
8	南京大学	80.7	综合类	江苏	本科一批	
9	解放军国防科学技术大学	80.31	军事类	湖南	本科一批	
10	中山大学	76.16	综合类	广东	本科一批	
11	吉林大学	75.99	综合类	吉林	本科一批	
12	华中科技大学	75.04	综合类	湖北	本科一批	
13	四川大学	74.5	综合类	四川	本科一批	
14	天津大学	73.54	理工类	天津	本科一批	
15	南开大学	73.5	综合类	天津	本科一批	
16	西安交通大学	73.5	综合类	陕西	本科一批	

1.3 常见排序算法



2. 实现常见排序算法

```
1 int a[] = {5, 3, 9, 6, 2, 4, 7, 1, 8};
```

2.1 插入排序

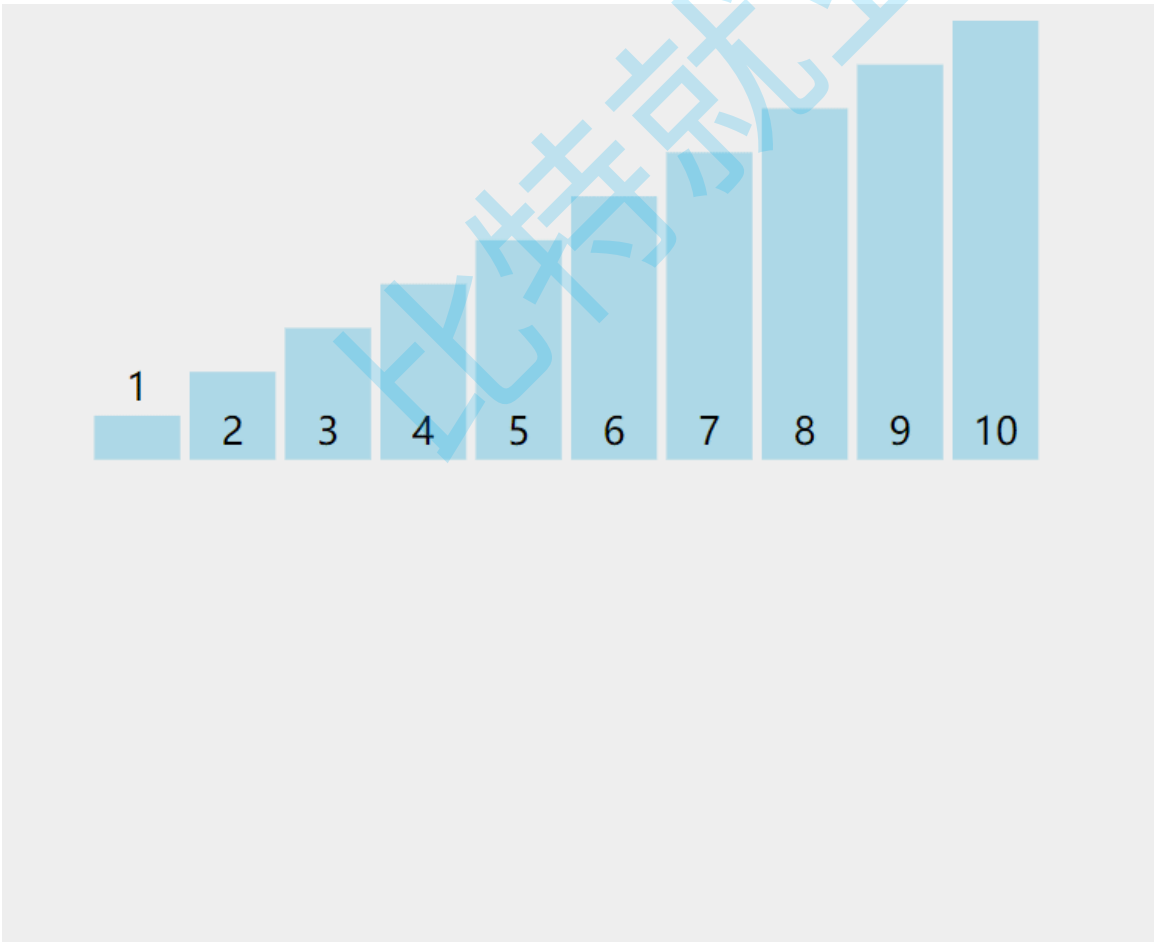
> 基本思想

直接插入排序是一种简单的插入排序法，其基本思想是：把待排序的记录按其关键码值的大小逐个插入到一个已经排好序的有序序列中，直到所有的记录插入完为止，得到一个新的有序序列。

实际中我们玩扑克牌时，就用了插入排序的思想

2.1.1 直接插入排序

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



代码实现

```c
void InsertSort(int* a, int n)
{
    for (int i = 0; i < n-1; i++)
    {
        int end = i ;
        int tmp = a[end + 1];
        while (end >= 0)
        {
            if (a[end] > tmp)
            {
                a[end + 1] = a[end];
                end--;
            }
            else
            {
                break;
            }
        }
        a[end + 1] = tmp;
    }
}
```

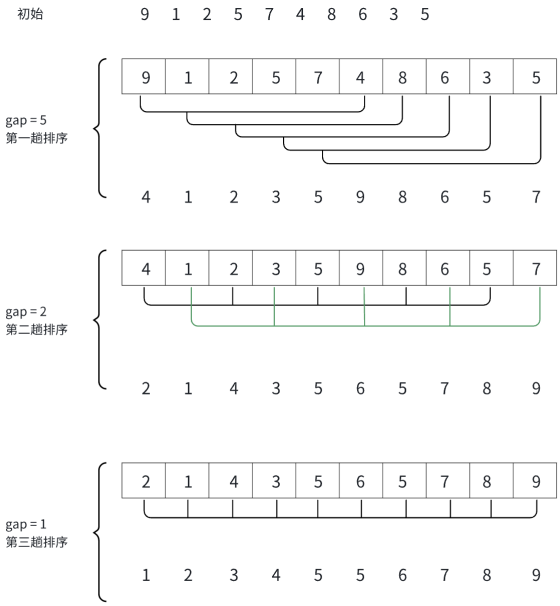
> 直接插入排序的特性总结

1. 元素集合越接近有序，直接插入排序算法的时间效率越高
2. 时间复杂度：O(N^2)
3. 空间复杂度：O(1)

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（通常是gap = n/3+1），把待排序文件所有记录分成各组，所有的距离相等的记录分在同一组内，并对每一组内的记录进行排序，然后gap=gap/3+1得到下一个整数，再将数组分成各组，进行插入排序，当gap=1时，就相当于直接插入排序。

它是在直接插入排序算法的基础上进行改进而来的，综合来说它的效率肯定是要高于直接插入排序算法的。



> 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。

代码实现

```
代码实现：
void ShellSort(int* a, int n)
```

```
{
    int gap = n;
    while (gap > 1)
    {
        //推荐写法：除3
        gap = gap / 3 + 1;

        for (int i = 0; i < n - gap; i++)
        {
            int end = i;
            int tmp = a[end + gap];
            while (end >= 0)
            {
                if (a[end] > tmp)
                {
                    a[end + gap] = a[end];
                    end -= gap;
                }
                else
                {
                    break;
                }
            }
            a[end + gap] = tmp;
        }
    }

}
```

2.1.2.1 希尔排序的时间复杂度计算

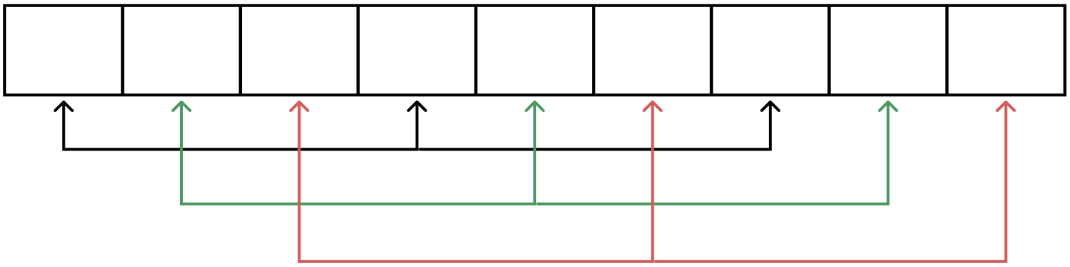
希尔排序的时间复杂度估算：

外层循环：

外层循环的时间复杂度可以直接给出为： $O(\log_2 n)$ 或者 $O(\log_3 n)$ ，即 $O(\log n)$

内层循环：

假设n = 9 ，若gap为3，则一共gap组，每组n/gap个数据



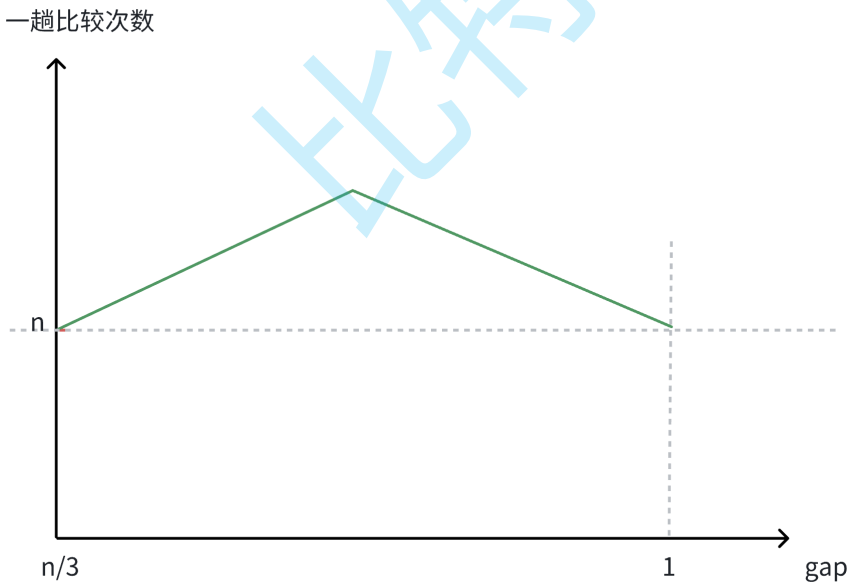
假设一共有n个数据，合计gap组，则每组为n/gap个；在每组中，插入移动的次数最坏的情况下为 $1+2+3+....+(\frac{n}{gap}-1))$ ，一共是gap组，因此：

总计最坏情况下移动总数为： $gap*[1+2+3+....+(\frac{n}{gap}-1)]$

gap取值有（以除3为例）：n/3 n/9 n/27 ..... 2 1

- 当gap为n/3时，移动总数为： $\frac{n}{3}*(1+2)=n$
- 当gap为n/9时，移动总数为： $\frac{n}{9}*(1+2+3+....+8)=\frac{n}{9}*\frac{8(1+8)}{2}=4n$
- 最后一躺，gap=1即直接插入排序，内层循环排序消耗为n

通过以上的分析，可以画出这样的曲线图：



因此，希尔排序在最初和最后的排序的次数都为n，即前一阶段排序次数是逐渐上升的状态，当到达某一顶点时，排序次数逐渐下降至n，而该顶点的计算暂时无法给出具体的计算过程

希尔排序时间复杂度不好计算，因为 `gap` 的取值很多，导致很难去计算，因此很多书中给出的希尔排序的时间复杂度都不固定。《数据结构(C语言版)》--- 严蔚敏书中给出的时间复杂度为：

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 $\mathrm{dlta}[k]=2^{t-k+1}-1$ 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。

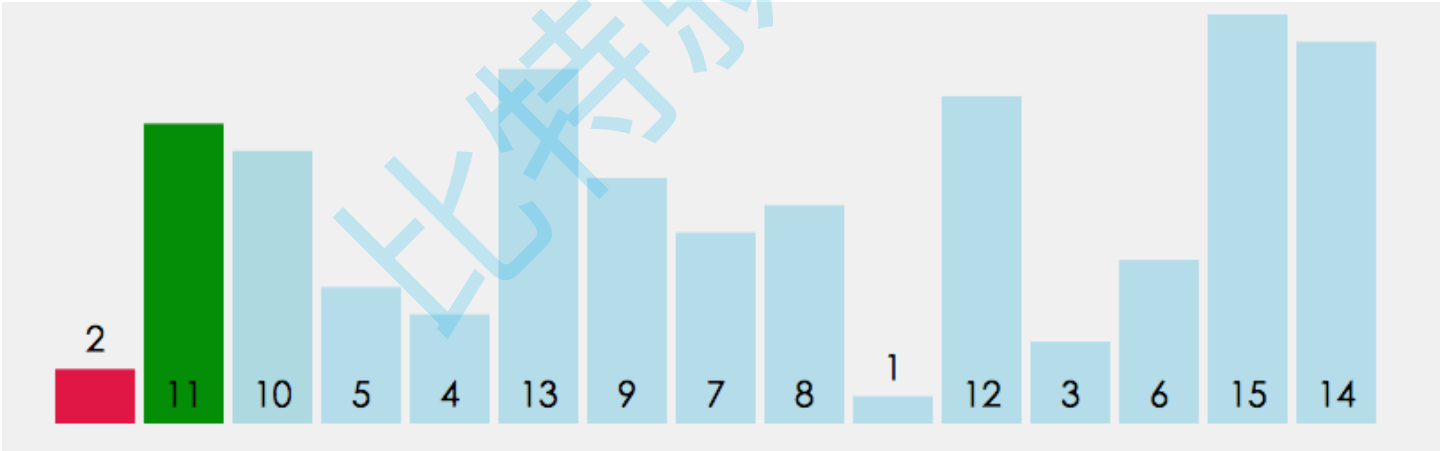
2.2 选择排序

选择排序的基本思想：

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

1. 在元素集合 `array[i]--array[n-1]` 中选择关键码最大(小)的数据元素
2. 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
3. 在剩余的 `array[i]--array[n-2]` (`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



代码实现：

```
1  void SelectSort(int* a, int n)
2  {
3      int begin = 0, end = n - 1;
4      while (begin < end)
5      {
6          int mini = begin, maxi = begin;
7          for (int i = begin; i <= end; i++)
8          {
```

```
            if (a[i] > a[maxi])
            {
                maxi = i;
            }
            if (a[i] < a[mini])
            {
                mini = i;
            }
        }
        if (begin == maxi)
        {
            maxi = mini;
        }
        swap(&a[mini], &a[begin]);
        swap(&a[maxi], &a[end]);

        ++begin;
        --end;
    }
}
```

> 直接选择排序的特性总结：

1. 直接选择排序思考非常好理解，但是效率不是很好。实际中很少使用
2. 时间复杂度：$O(N^2)$
3. 空间复杂度：$O(1)$

2.2.2 堆排序

堆排序(Heapsort)是指利用堆积树（堆）这种数据结构所设计的一种排序算法，它是选择排序的一种。它是通过堆来进行选择数据。需要注意的是排升序要建大堆，排降序建小堆。

在二叉树章节我们已经实现过堆排序，这里不再赘述

2.3 交换排序

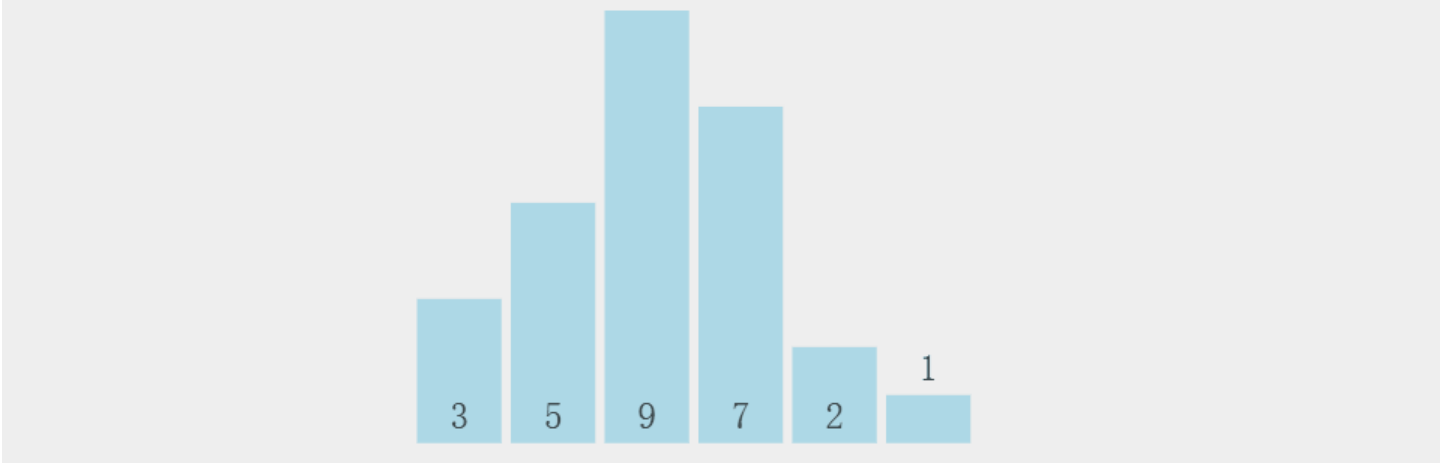
交换排序基本思想：

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

前面在算法题中我们已经接触过冒泡排序的思路了，冒泡排序是一种最基础的交换排序。之所以叫做冒泡排序，因为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数组的一侧移动。



代码实现：

```
1  void BubbleSort(int* a, int n)
2  {
3      int exchange = 0;
4      for (int i = 0; i < n; i++)
5      {
6          for (int j = 0; j <n-i-1 ; j++)
7          {
8              if (a[j] > a[j + 1])
9              {
10                 exchange = 1;
11                 swap(&a[j], &a[j + 1]);
12             }
13         }
14         if (exchange == 0)
15         {
16             break;
17         }
18     }
19 }
```

> 冒泡排序的特性总结

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(1)$

2.3.2 快速排序

快速排序是Hoare于1962年提出的一种二叉树结构的交换排序方法，其基本思想为：任取待排序元素序列中的某元素作为基准值，按照该排序码将待排序集合分割成两子序列，左子序列中所有元素均小于基准值，右子序列中所有元素均大于基准值，然后最左右子序列重复该过程，直到所有元素都排列在相应位置上为止。

快速排序实现主框架：

```
//快速排序
void QuickSort(int* a, int left, int right)
{
    if (left >= right) {
        return;
    }

    //_QuickSort用于按照基准值将区间[left,right)中的元素进行划分
    int meet = _QuickSort(a, left, right);
    QuickSort(a, left, meet - 1);
    QuickSort(a, meet + 1, right);
}
```

将区间中的元素进行划分的 `_QuickSort` 方法主要有以下几种实现方式：

2.3.2.1 hoare版本

> 算法思路：

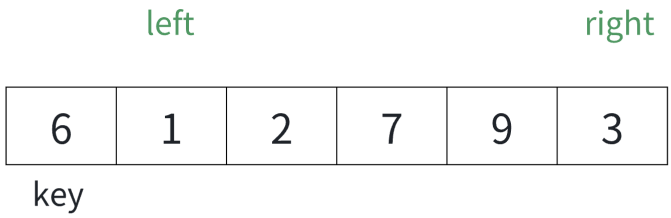
1）创建左右指针，确定基准值

2）从右向左找出比基准值小的数据，从左向右找出比基准值大的数据，左右指针数据交换，进入下次循环

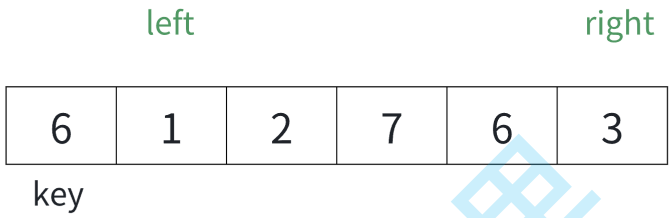
> 问题1：为什么跳出循环后right位置的值一定不大于key?

当 `left > right` 时，即right走到left的左侧，而left扫描过的数据均不大于key，因此right此时指向的数据一定不大于key

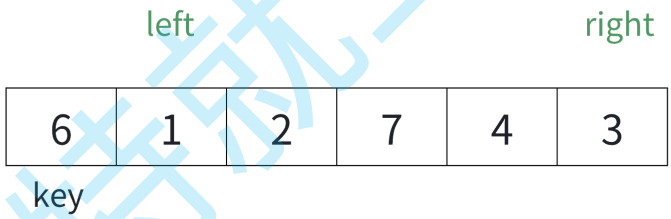
场景1：相遇值大于key值



场景2：相遇值等于key值

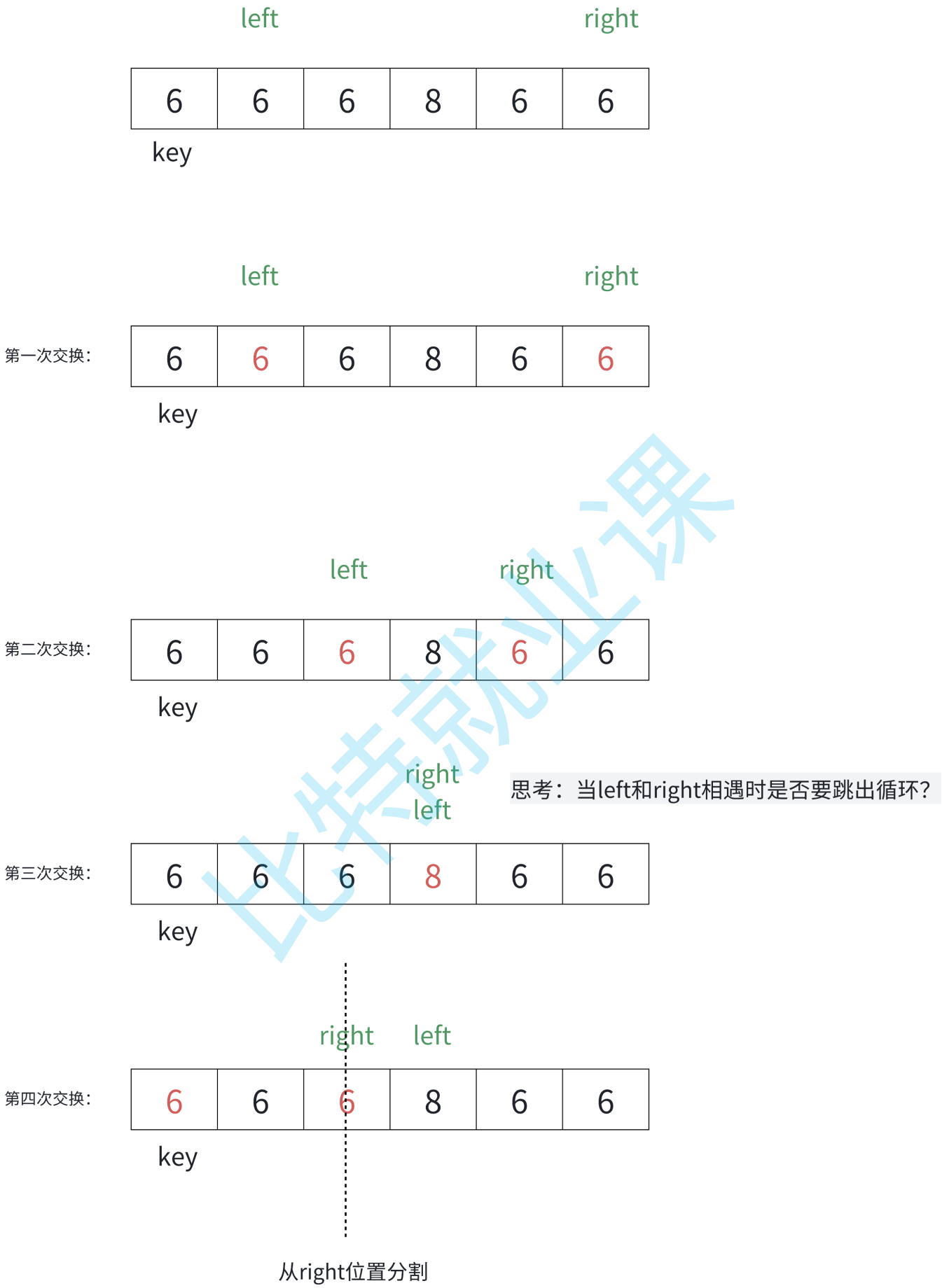


场景3：相遇值小于key值



> 问题2：为什么left 和 right指定的数据和key值相等时也要交换？

相等的值参与交换确实有一些额外消耗。实际还有各种复杂的场景，假设数组中的数据大量重复时，无法进行有效的分割排序。


left
right
6 6 6 8 6 6
key
left
right
第一次交换：
6 6 6 8 6 6
key
left
right
第二次交换：
6 6 6 8 6 6
key
right
left
思考：当left和right相遇时是否要跳出循环？
第三次交换：
6 6 6 8 6 6
key
right
left
第四次交换：
6 6 6 8 6 6
key
从right位置分割

代码实现：

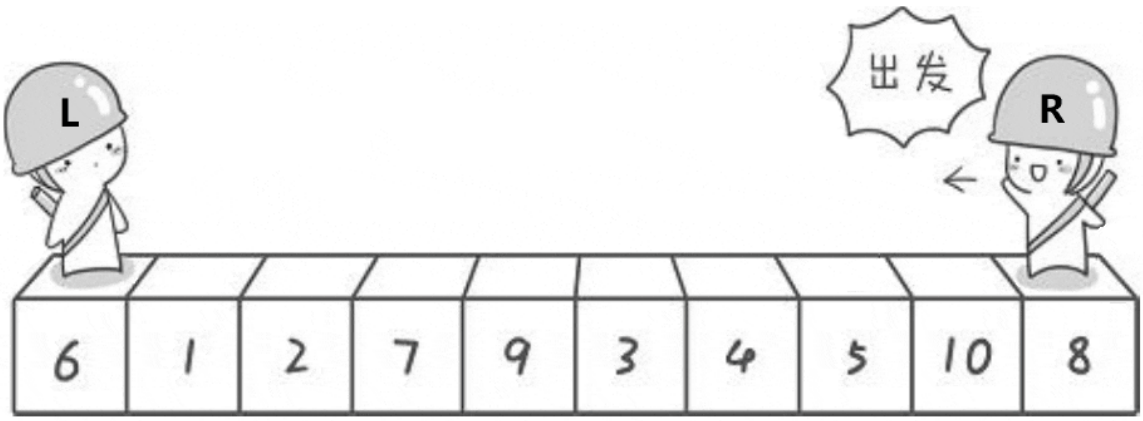
```c
int _QuickSort(int* a, int left, int right)
{
    int begin = left;
    int end = right;


    int keyi = left;
    ++left;

    while (left <= right)
    {
        // 右边找小
        while (left <= right && a[right] > a[keyi])
        {
            --right;
        }

        // 右边找小
        while (left <= right && a[left] < a[keyi])
        {
            ++left;
        }

        if (left <= right)
        {
            swap(&a[left++], &a[right--]);
        }
    }

    swap(&a[keyi], &a[right]);
    return right;
}
```


2.3.2.2 挖坑法

> 思路：
>
> 创建左右指针。首先从右向左找出比基准小的数据，找到后立即放入左边坑中，当前位置变为新的"坑"，然后从左向右找出比基准大的数据，找到后立即放入右边坑中，当前位置变为新的"坑"，结束循环后将最开始存储的分界值放入当前的"坑"中，返回当前"坑"下标（即分界值下标）

先将第一个数据存放在临时变量 key 中，形成一个坑位

key =

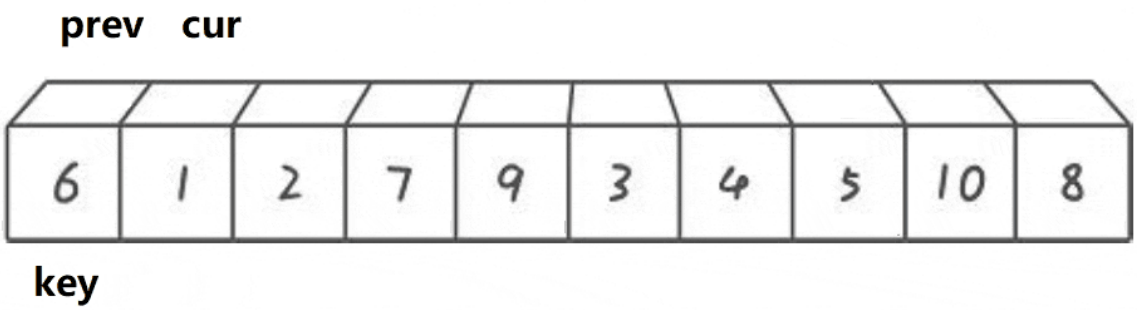



```
1  int _QuickSort(int* a, int left, int right)
2  {
3      int mid = a[left];
4      int hole = left;
5      int key = a[hole];
6      while (left < right)
7      {
8          while (left < right && a[right] >= key)
9          {
10             --right;
11         }
12         a[hole] = a[right];
13         hole = right;
14         while (left < right && a[left] <= key)
15         {
16             ++left;
17         }
18         a[hole] = a[left];
19         hole = left;
20     }
21     a[hole] = key;
22     return hole;
23 }
```


2.3.2.3 lomuto前后指针

> 创建前后指针，从左往右找比基准值小的进行交换，使得小的都排在基准值的左边。

初始时，prev指针指向序列开头，
cur指针指向prev指针的后一个位置




```
int _QuickSort(int* a, int left, int right)
{
    int prev = left, cur = left + 1;
    int key = left;
    while (cur <= right)
    {
        if (a[cur] < a[key] && ++prev != cur)
        {
            swap(&a[cur], &a[prev]);
        }
        ++cur;
    }
    swap(&a[key], &a[prev]);

    return prev;
}
```

> 快速排序特性总结：

1. 时间复杂度：$O(nlogn)$
2. 空间复杂度：$O(logn)$

2.3.2.4 非递归版本

非递归版本的快速排序需要借助数据结构：栈

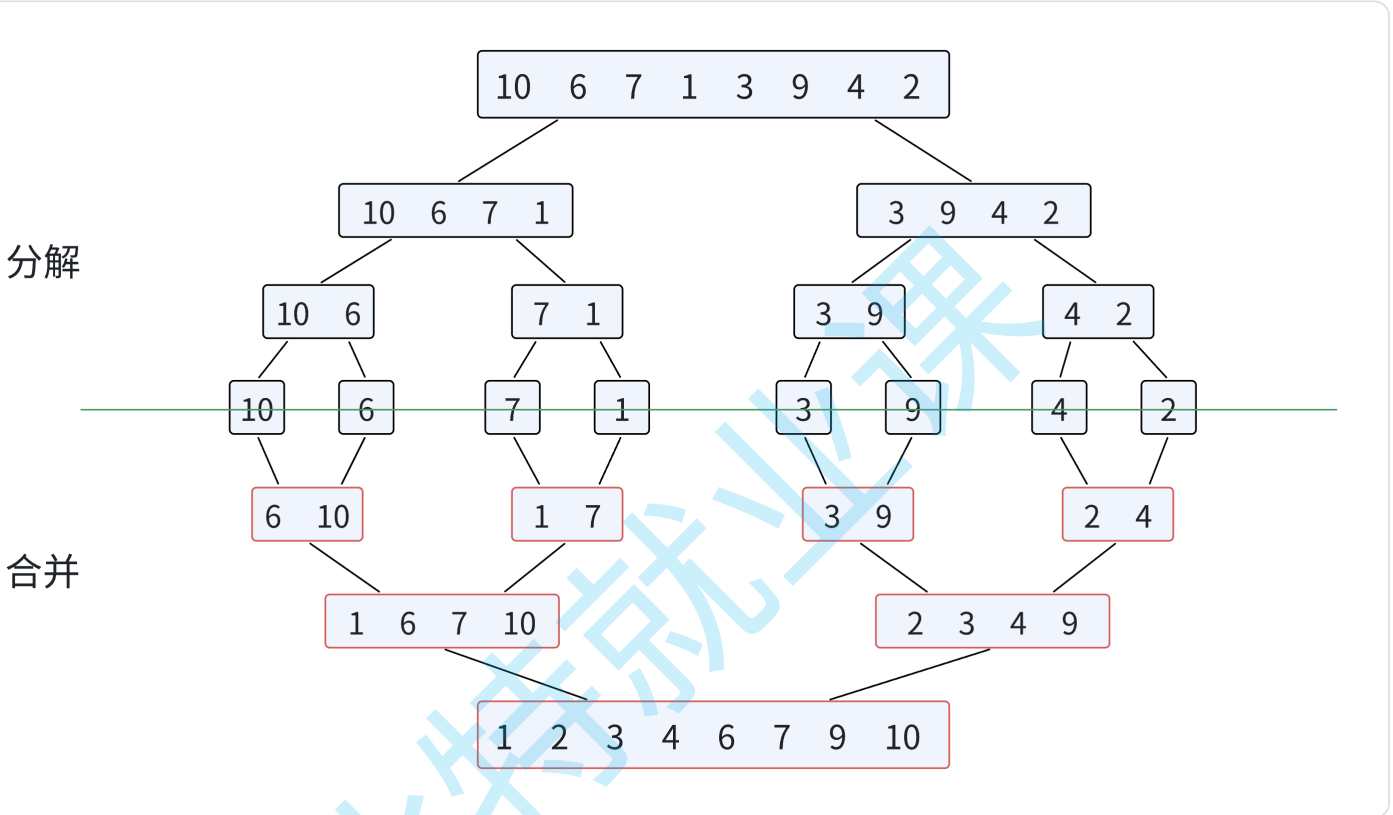
代码实现

```c
void QuickSortNonR(int* a, int left, int right)
{
    ST st;
    STInit(&st);
    STPush(&st, right);
    STPush(&st, left);

    while (!STEmpty(&st))
    {
        int begin = STTop(&st);
        STPop(&st);

        int end = STTop(&st);
        STPop(&st);

        // 单趟
        int keyi = begin;
        int prev = begin;
        int cur = begin + 1;

        while (cur <= end)
        {
            if (a[cur] < a[keyi] && ++prev != cur)
                    Swap(&a[prev], &a[cur]);

            ++cur;
        }

        Swap(&a[keyi], &a[prev]);
        keyi = prev;

        // [begin, keyi-1] keyi [keyi+1, end]
        if (keyi + 1 < end)
        {
            STPush(&st, end);
            STPush(&st, keyi + 1);
        }

        if (begin < keyi-1)
        {
            STPush(&st, keyi-1);
            STPush(&st, begin);
        }
    }

    STDestroy(&st);
}
```

2.4 归并排序

归并排序算法思想：

归并排序（MERGE-SORT）是建立在归并操作上的一种有效的排序算法,该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。 归并排序核心步骤：



代码实现

```
void _MergeSort(int* a, int left, int right, int* tmp)
{
    if (left >= right)
    {
        return;
    }
    int mid = (right + left) / 2;
    //[left,mid] [mid+1,right]
    _MergeSort(a, left, mid, tmp);
    _MergeSort(a, mid + 1, right, tmp);

    int begin1 = left, end1 = mid;
    int begin2 = mid + 1, end2 = right;
    int index = begin1;

```

```
    //合并两个有序数组为一个数组
    while (begin1 <= end1 && begin2 <= end2)
    {
        if (a[begin1] < a[begin2])
        {
            tmp[index++] = a[begin1++];
        }
        else
        {
            tmp[index++] = a[begin2++];
        }
    }
    while (begin1 <= end1)
    {
        tmp[index++] = a[begin1++];
    }
    while (begin2 <= end2)
    {
        tmp[index++] = a[begin2++];
    }
    for (int i = left; i <= right; i++)
    {
        a[i] = tmp[i];
    }
}
void MergeSort(int* a, int n)
{
    int* tmp = new int[n];
    _MergeSort(a, 0, n - 1, tmp);
    delete[] tmp;
}

```

> 归并排序特性总结：

1. 时间复杂度：$O(nlogn)$
2. 空间复杂度：$O(n)$

2.5 测试代码：排序性能对比

```
// 测试排序的性能对比
void TestOP()
{
    srand(time(0));
    const int N = 100000;
```

```
6     int* a1 = (int*)malloc(sizeof(int)*N);
7     int* a2 = (int*)malloc(sizeof(int)*N);
8     int* a3 = (int*)malloc(sizeof(int)*N);
9     int* a4 = (int*)malloc(sizeof(int)*N);
10    int* a5 = (int*)malloc(sizeof(int)*N);
11    int* a6 = (int*)malloc(sizeof(int)*N);
12    int* a7 = (int*)malloc(sizeof(int)*N);
13    for (int i = 0; i < N; ++i)
14    {
15        a1[i] = rand();
16        a2[i] = a1[i];
17        a3[i] = a1[i];
18        a4[i] = a1[i];
19        a5[i] = a1[i];
20        a6[i] = a1[i];
21        a7[i] = a1[i];
22    }
23    int begin1 = clock();
24    InsertSort(a1, N);
25    int end1 = clock();
26
27    int begin2 = clock();
28    ShellSort(a2, N);
29    int end2 = clock();
30
31    int begin3 = clock();
32    SelectSort(a3, N);
33    int end3 = clock();
34
35    int begin4 = clock();
36    HeapSort(a4, N);
37    int end4 = clock();
38
39    int begin5 = clock();
40    QuickSort(a5, 0, N-1);
41    int end5 = clock();
42
43    int begin6 = clock();
44    MergeSort(a6, N);
45    int end6 = clock();
46
47    int begin7 = clock();
48    BubbleSort(a7, N);
49    int end7 = clock();
50
51    printf("InsertSort:%d\n", end1 - begin1);
52    printf("ShellSort:%d\n", end2 - begin2);
```

```
53      printf("SelectSort:%d\n", end3 - begin3);
54      printf("HeapSort:%d\n", end4 - begin4);
55      printf("QuickSort:%d\n", end5 - begin5);
56      printf("MergeSort:%d\n", end6 - begin6);
57      printf("BubbleSort:%d\n", end7 - begin7);
58
59      free(a1);
60      free(a2);
61      free(a3);
62      free(a4);
63      free(a5);
64      free(a6);
65      free(a7);
66  }
```

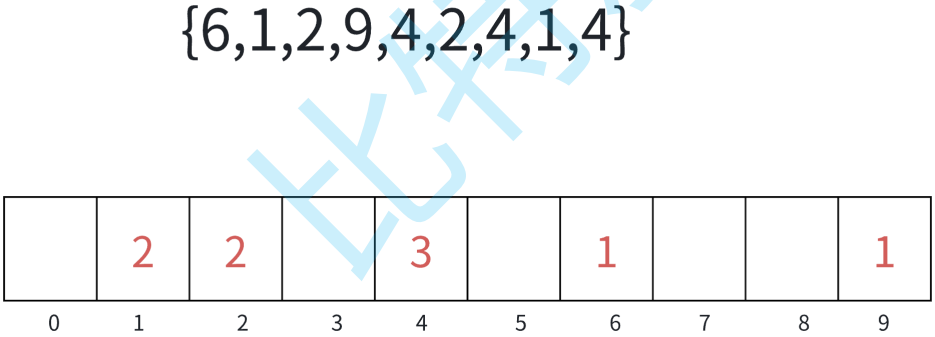
2.6 非比较排序

2.6.1 计数排序

计数排序又称为鸽巢原理，是对哈希直接定址法的变形应用。 操作步骤：

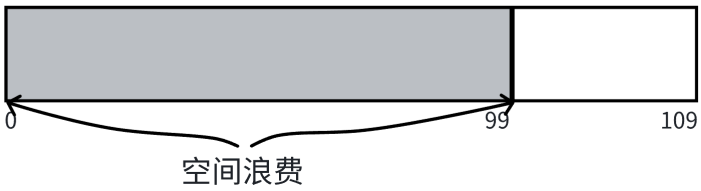
1）统计相同元素出现次数

2）根据统计的结果将序列回收到原来的序列中



{100,101,109,105,101,105}

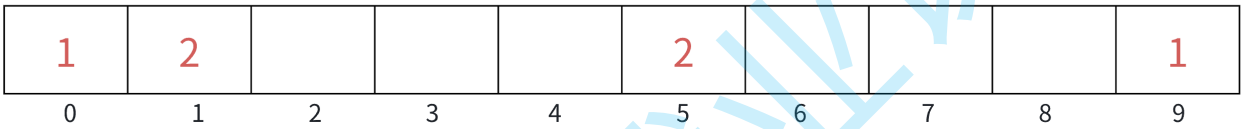
1）按数值开空间



2）按范围开空间

max 109　　min 100

range = max - min = 9



代码实现

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

        if (a[i] < min)
                min = a[i];
    }

    int range = max - min + 1;
    int* count = (int*)malloc(sizeof(int) * range);
    if (count == NULL)
    {
        perror("malloc fail");
        return;
    }

```

```
    memset(count, 0, sizeof(int) * range);

    // 统计次数
    for (int i = 0; i < n; i++)
    {
        count[a[i] - min]++;`
    }

    // 排序
    int j = 0;
    for (int i = 0; i < range; i++)
    {
        while (count[i]--)
        {
            a[j++] = i + min;
        }
    }
}
```

> 计数排序的特性：
>
> 计数排序在数据范围集中时，效率很高，但是适用范围及场景有限。
>
> 时间复杂度：$O(N+range)$
>
> 空间复杂度：$O(range)$
>
> 稳定性：稳定

3. 排序算法复杂度及稳定性分析

稳定性：假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。

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
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	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

稳定性验证案例

> 直接选择排序：5 8 5 2 9
>
> 希尔排序：5 8 2 5 9
>
> 堆排序：2 2 2 2
>
> 快速排序：5 3 3 4 3 8 9 10 11