实验二 典型排序算法训练： 快速排序、 计数排序 实验报告

姓名：陈矛　　学号：SA22225116

一、实验原理

1. 快速排序

1) 算法流程图：

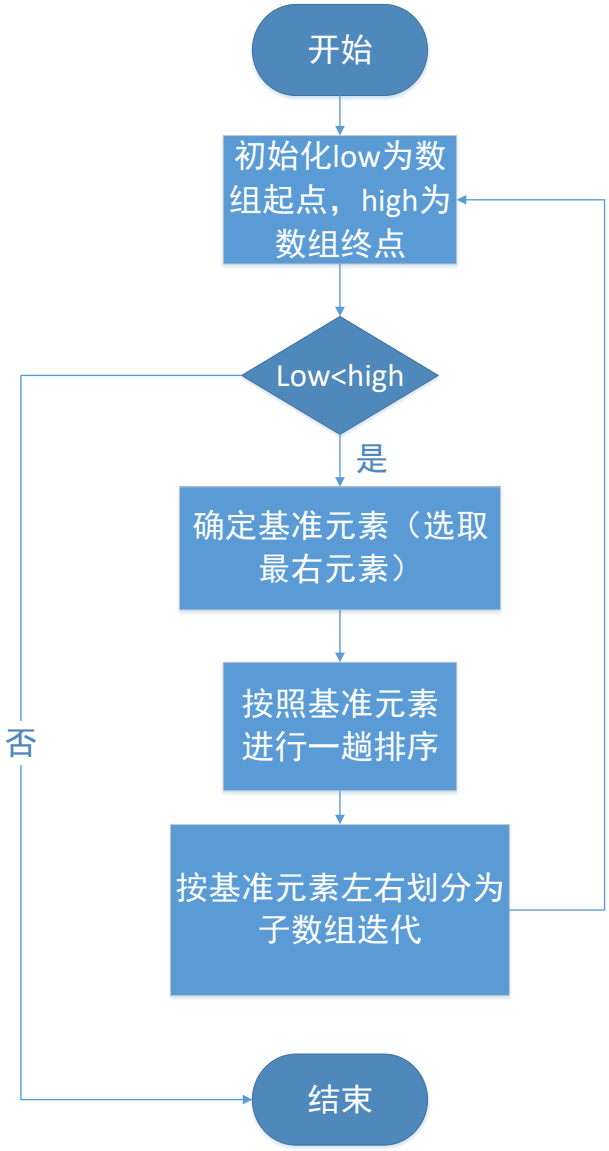

图 快速排序流程图

2) 时间复杂度（详细分析见后续分析）：
平均时间复杂度：O(nlogn)
最佳时间复杂度：O(nlogn)
最差时间复杂度：O(n^2)

3) 空间复杂度：O(logn)

2. 计数排序：

1) 算法流程图

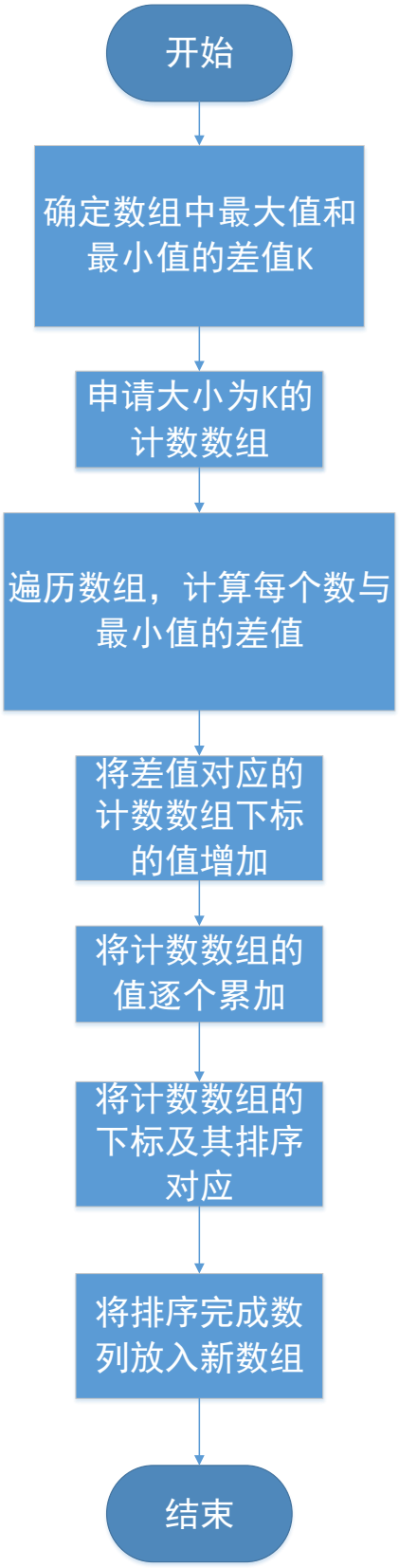

图 计数排序流程图

2) 时间复杂度（详细分析见后续分析）：
   平均时间复杂度：O(n + k)
   最佳时间复杂度：O(n + k)
   最差时间复杂度：O(n + k)
3) 空间复杂度：O(n + k)

3. 快速排序原理

快速排序的思想是任找一个元素作为基准，对待排数组进行分组，使基准元素左边的数据比基准数据要小，右边的数据比基准数据要大，这样基准元素就放在了正确的位置上。 然后对基准元素左边和右边的组进行相同的操作，最后将数据排序完成。

4. 计数排序原理

计数排序是由额外空间的辅助和元素本身的值决定的。计数排序过程中不存在元素之间的比较和交换操作，根据元素本身的值，将每个元素出现的次数记录到辅助空间后，通过对辅助空间内数据的计算，即可确定每一个元素最终的位置。

算法过程：

（1）根据待排序集合中最大元素和最小元素的差值范围，申请额外空间；

（2）遍历待排序集合，将每一个元素出现的次数记录到元素值对应的额外空间内；

（3）对额外空间内数据进行计算，得出每一个元素的正确位置；

（4）将待排序集合每一个元素移动到计算出的正确位置上。

计数排序的一个重要性质就是它是稳定的：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的次序相同，即在输入数组中先出现的，在输出数组中也位于前面。

二、运行结果及分析

1. 实现对数组[-2,1,-3,4,-1,2,1,-5,4]的快速排序并画出流程图。

流程图：

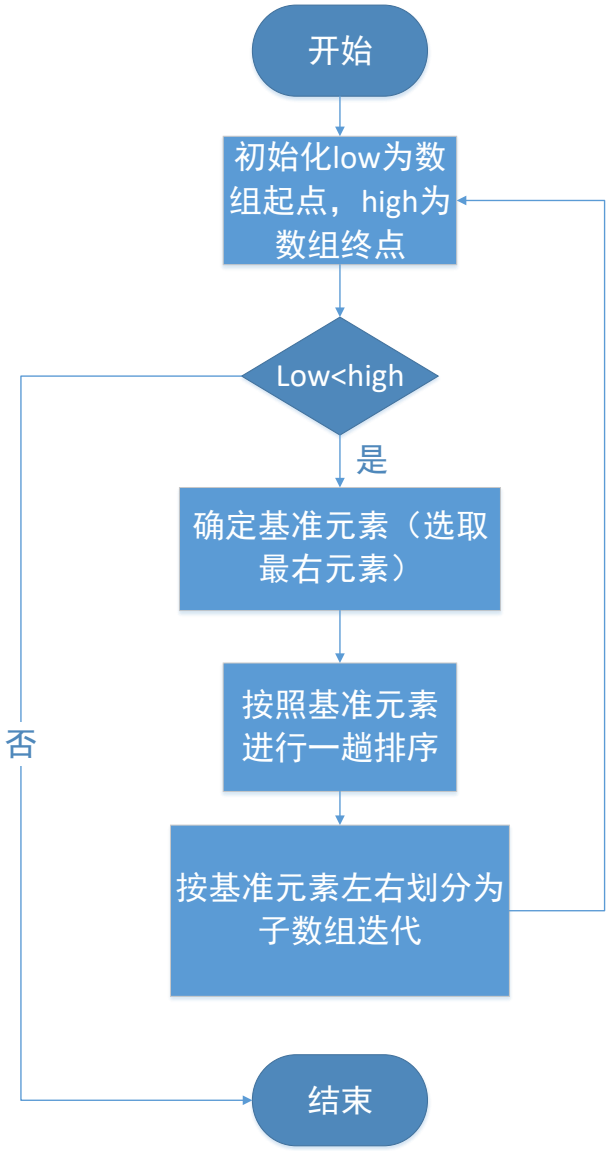

图 快速排序流程图

实验结果：

```
E:\University\java\develop_tools\jdk1.8.0_131\bin\java ...
快速排序实验结果: [-5, -3, -2, -1, 1, 1, 2, 4, 4]

Process finished with exit code 0
```

关键代码：

```
public void quicksort(int[] nums, int p, int r){
    if (p < r){
        int pos = partition(nums,p,r);
        quicksort(nums,p,pos-1);
        quicksort(nums,pos+1,r);
    }
}
```

```java
public int partition(int[] nums, int p, int r){
    int pivot = nums[r];
    int i = p, j = p;
    while( j < r){
        if(nums[j] < pivot){
            swap(nums,i,j);
            i++;
        }
        j++;
        //如果比pivot 小，j 用来检查每一个数，i 用来记录下一个数被交换过来时应该放的位置
        //当找到大于等于pivot 的数时，i 停在这个位置，让j 继续往后找，找到比pivot 小的数时，换到这个位置上
    }

    swap(nums,i,r);
    return i;
}

public void swap(int[] a,int i,int j){
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

2. 实现对数组[95, 94, 91, 98, 99, 90, 99, 93, 91, 92]的计数排序并画出流程图。

流程图：

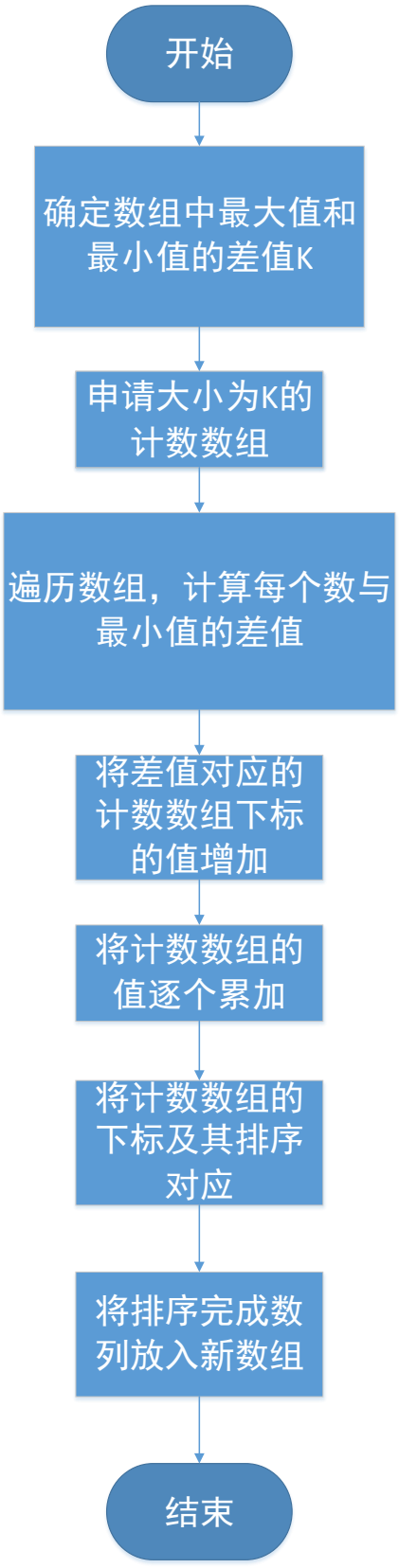

图 计数排序流程图

运行结果：

```
E:\University\java\develop_tools\jdk1.8.0_131\bin\java ...
计数排序实验结果：[90, 91, 91, 92, 93, 94, 95, 98, 99, 99]

Process finished with exit code 0
```

关键代码：

```java
public int[] countingSort(int[] nums,int[] res,int k){
    int[] count = new int[k];
    int maxN = nums[0];
    int minN = nums[0];

    //找最大值和最小值
    for(int i = 1;i<nums.length;i++){
        maxN = Math.max(nums[i],maxN);
        minN = Math.min(nums[i],minN);
    }

    //计数，此时result数组里存放着每个数的数量
    for (int i = 0;i<nums.length;i++){
        count[nums[i] - minN] ++;
    }

    //对所有计数累加，累加之后，count[i]里面就存放着比该元素小的元素数量
    for (int i = 1;i < k; i++){
        count[i] = count[i] + count[i - 1];
    }

    //反向填充目标数组：将每个元素i放在新数组的第count(i)项，每放一个元素就将count(i)减去1
    for (int i = nums.length - 1;i >= 0 ;i--){
        res[count[nums[i] - minN] - 1] = nums[i];
        //通过count找到这个元素应该放在哪个位置
        count[nums[i] - minN] --;
    }

    return res;

}
```

3. 以上两种排序算法的区别有哪些？ 分别的时间和空间复杂度是多少？

快速排序原理

快速排序的思想是任找一个元素作为基准，对待排数组进行分组，使基准元

素左边的数据比基准数据要小，右边的数据比基准数据要大，这样基准元素就放在了正确的位置上。 然后对基准元素左边和右边的组进行相同的操作，最后将数据排序完成。

计数排序原理

计数排序是由额外空间的辅助和元素本身的值决定的。计数排序过程中不存在元素之间的比较和交换操作，根据元素本身的值，将每个元素出现的次数记录到辅助空间后，通过对辅助空间内数据的计算，即可确定每一个元素最终的位置。

算法过程：

（1）根据待排序集合中最大元素和最小元素的差值范围，申请额外空间；

（2）遍历待排序集合，将每一个元素出现的次数记录到元素值对应的额外空间内；

（3）对额外空间内数据进行计算，得出每一个元素的正确位置；

（4）将待排序集合每一个元素移动到计算出的正确位置上。

计数排序的一个重要性质就是它是稳定的：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的次序相同，即在输入数组中先出现的，在输出数组中也位于前面。而快速排序是不稳定的。

时间复杂度和空间复杂度：

快速排序

1) 时间复杂度：
   平均时间复杂度：O(nlogn)
   最佳时间复杂度：O(nlogn)
   最差时间复杂度：O(n^2)
2) 空间复杂度：O(logn)

快速排序的本质采用的是“分治”的思想，每一趟递归会确定一个元素的位置，然后再对它左边（所有比它小的元素）和右边（所有比它大的元素）的两个区间继续进行快速排序，就这样，每一次都会将一个问题分成两个子问题，直到拆分到最小子问题为止。在平均的情况下，由于对于每一趟递归而言，时间复杂度都为 O(n)，而总共会进行 logn 次递归（每次拆分成两个子问题）。所以平均时间复杂度为 O(nlogn)。由于总共进行了 logn 次递归，也就是需要空间复杂度为 O(logn)的栈来存放数据。

在最差的情况下，每一次的拆分都出现很“不均衡”的情况。这时递归树的深度会达到 n，这样的情况下，最差的时间复杂度为 O(n^2)。

计数排序

1) 时间复杂度：
   平均时间复杂度：O(n+k)
   最佳时间复杂度：O(n+k)
   最差时间复杂度：O(n+k)
2) 空间复杂度：O(n+k)

其中 k 是待排序数的范围（max-min）。

对于一个待排序的元素 x，我们可以确定小于等于 x 的个数 i，根据这个个数 i，我们就可以把 x 放到索引 i 处。那么只需要确定小于等于 x 的个数，我们可以专门开辟一个数组 c[]，然后遍历数组，确定数组 a 中每个元素中出现的频率，然后就可以确定对于 a 中每个元素 x，小于等于这个元素的个数。然后就可以把元素 x 放到对应位置了。当然，元素 x 的大小是可能重复的，为了保证算法的稳定性，这样就需要我们对数组 c 的值访问之后减 1，保证和 x 一样大的元素能放在其前面。

计数排序的运行过程：

第一步：对于数组 A，我们首先统计每个值的个数，将 A 的值作为 C 元素索引，值的个数作为 C 数组的值。比如对于数组 A 中的元素 2，在数组 A 中出现了 2 次，所以 c[2] = 2，而元素 5 出现了以此，所以 c[5] = 1。

第二步：至此为止，数组 C 中已经统计了各个元素的出现次数，那么我们就可以根据各个元素的出现次数，累加出比该元素小的元素个数，更新到数组 C 中。C[0]=2 表示出现 0 的次数为 2，C[1]=0 表示出现 1 的次数为 0，那么小于等于 1 的元素个数为 C[0]+C[1]=2，我们把 C[1]更新为 2，同理 C[2]=2 表示出现 2 的次数为 2，那么小于等于 2 的元素个数为 C[1]+C[2]=4，继续把 C[2]更新为 4，以此类推……

第三步：到这里，我们得到了存储小于等于元素的个数的数组 C。现在我们开始从**尾部到头部**遍历数组 A，比如首先我们看 A[7] = 3，然后查找 C[3]，发现 C[3] = 7，说明有 7 个元素小于等于 3。我们首先需要做一步 C[3] = C[3] - 1，因为这里虽然有 7 个元素小于等于 3，但是 B 的索引是从 0 开始的，而且这样减一可以保证下次再找到一个 3，可以放在这个 3 的前面。然后 B[C[3]] = 3，就把第一个 3 放到了对的位置。后面以此类推，直到遍历完数组 B。

从整个算法的流程可以看出，整个排序过程并未发生过元素之间的比较。算法的所有开销都来源于数组的遍历。在第一步中，统计了数组 A 中的数，时间开销为 O(n)。在第二步中，对所有的元素进行累加，而数组 C 的长度为 k，时间开销为 O(k)。第三步中依次把所有元素放到正确的位置上，总开销为 O(n)。综上，整个算法的时间复杂度为 O(n+k)，空间复杂度为 O(n+k)。

三、实验总结

（针对本次实验，客观评估自己的代码，叙述本次实验收获）

本次实验，通过课本上两种排序算法的伪代码写了可以成功运行的 java 代码，其中快速排序是比较熟悉的算法，代码实现上没有遇到比较大的困难。对于计数排序，一开始的实现思想是把所有数的个数都数出来，然后再根据这个计数结果从前往后覆盖原数组。但是这样子做的话，实际上放回的并不是“原数据”，而是直接对数组赋值，这种做法不能保证稳定性，在实际工作中也是不合理的。于是重新修改了代码，实现了算法的稳定性和正确性。

快速排序和计数排序是两种常用的排序算法。下面是它们的一些对比总结：

1) 原理：快速排序是一种分治算法，它通过不断地选取一个基准元素并将数组划分为两个子数组来实现排序。计数排序是一种线性排序算法，它通过计算每个数字在数组中出现的次数来实现排序。
2) 时间复杂度：快速排序的时间复杂度为 O(n*log n)，其中 n 是数组的大

小。计数排序的时间复杂度为 O(n)，因此它在某些情况下比快速排序更快。

3) 空间复杂度：快速排序的空间复杂度为 O(log n)，因为它需要使用递归来实现分治。计数排序的空间复杂度为 O(n)，因为它需要使用一个计数数组来存储每个数字出现的次数。
4) 适用情况：快速排序适用于各种情况，包括随机数组和有序数组。计数排序只适用于数组中的数字都在一个较小的范围内，因为它需要为每个可能出现的数字分配一个位置。
5) 稳定性：快速排序是不稳定的排序算法，这意味着它可能会改变相等元素的相对顺序。计数排序是稳定的排序算法，这意味着它会保留相等元素的相对顺序。
6) 代码实现：快速排序的代码实现相对较复杂，需要实现递归和分治的逻辑。计数排序的代码实现较为简单，只需要统计每个数字的出现次数并将它们按顺序放回原数组即可。

总的来说，快速排序是一种适用范围广泛的排序算法，但它的代码实现较为复杂。计数排序是一种简单易实现的线性排序算法，但它的适用范围有限，只能用于数字较小的情况。在选择排序算法时，需要根据需要排序的数据的特点来决定使用哪种算法。

实验日期：2022 年 11 月 27 日

附录

实验源代码：

1. SolutionForLab2.java

```java
import java.util.Arrays;

/**
 * @author ChenMao
 * @create 2022-11-27 20:26
 */
public class SolutionForLab2 {

    public static void main(String[] args) {
        int[] nums1 = {-2,1,-3,4,-1,2,1,-5,4};
```

```
        SolutionForLab2 test = new SolutionForLab2();

        //快速排序结果
        test.quicksort(nums1,0,nums1.length - 1);
//        System.out.println("快速排序实验结果: " +
Arrays.toString(nums1));


        int [] nums2 = {95, 94, 91, 98, 99, 90, 99, 93, 91, 92};

        int[] res2 = new int[nums2.length];

        int k = nums2[0];
        for(int i = 1;i<nums2.length;i++){
            k = Math.max(nums2[i],k);
        }

        test.countingSort(nums2,res2,k);


        System.out.println("计数排序实验结果: " +
Arrays.toString(res2));




    }

    public int[] countingSort(int[] nums,int[] res,int k){
        int[] count = new int[k];
        int maxN = nums[0];
        int minN = nums[0];

        //找最大值和最小值
        for(int i = 1;i<nums.length;i++){
            maxN = Math.max(nums[i],maxN);
            minN = Math.min(nums[i],minN);
        }

        //计数，此时result 数组里存放着每个数的数量
        for (int i = 0;i<nums.length;i++){
            count[nums[i] - minN] ++;
        }
```

```
        //对所有计数累加，累加之后，count[i]里面就存放着比该元素小的元素数量
        for (int i = 1;i < k; i++){
            count[i] = count[i] + count[i - 1];
        }

        //反向填充目标数组：将每个元素 i 放在新数组的第 count(i)项，每放一个元素就将 count(i)减去 1
        for (int i = nums.length - 1;i >= 0 ;i--){
            res[count[nums[i] - minN] - 1] = nums[i];
            //通过 count 找到这个元素应该放在哪个位置
            count[nums[i] - minN] --;
        }

        return res;

    }

    public void quicksort(int[] nums, int p, int r){
        if (p < r){
            int pos = partition(nums,p,r);
            quicksort(nums,p,pos-1);
            quicksort(nums,pos+1,r);
        }
    }

    public int partition(int[] nums, int p, int r){
        int pivot = nums[r];
        int i = p, j = p;
        while( j < r){
            if(nums[j] < pivot){
                swap(nums,i,j);
                i++;
            }
            j++;
            //如果比 pivot 小，j 用来检查每一个数，i 用来记录下一个数被交换过来时应该放的位置
            //当找到大于等于 pivot 的数时,i 停在这个位置,让 j 继续往后找，找到比 pivot 小的数时，换到这个位置上
        }

        swap(nums,i,r);
        return i;
    }
```

```
    public void swap(int[] a,int i,int j){
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }


}
```