Java 语言程序设计 作业 3-Java 基础

封钰震 (1951362)

2021 年 11 月 1 日

1. 编程环境

硬件环境

- 型号名称：MacBook Pro
- 处理器名称：Dual-Core Intel Core i5
- 内存：8 GB

软件环境

- 操作系统：macOS 10.15.7

运行环境

- JDK 14.0.2

2. 设计思想

第一问

首先，考察计算后一个排列的`nextArrange`函数，算法的步骤如下：

1. 对该排列 $A[1], A[2], \cdots, A[A.length]$ 从后往前遍历，直到 $A[i] > A[i-1]$，此步的时间复杂度为 $O(n)$；
2. 从 $A[i], A[i+1], \cdots, A[A.length]$ 中寻找到 $A[j] > A[i-1]$ 且如果 $j < A.length$，则 $A[j+1] \leq A[i-1]$，此步的时间复杂度为 $O(n-i+1) = O(n)$；

i	1	2	3	4
$A[i]$	2	1	4	3

表 1: 排列 2143

i	1	2	3	4
$A[i]$	2	3	1	4

表 2: 排列 2314

3. 将 $A[i-1]$ 与 $A[j]$ 交换，此步的时间复杂度为 $O(1)$；

4. 将 $A[i], A[i+1], \cdots, A[A.length]$ 排成顺序，即 $A'[i] \leq A'[i+1] \leq \cdots \leq A'[A.length]$。由于在上述步骤后，$A[i], A[i+1], \cdots, A[A.length]$ 是逆序的，因此该步的时间复杂度为 $O(n)$。

例如，若对于排列 2143 寻找下一个排列，即 A 如表1。第一步，从后往前找到 $A[3] > A[2]$，即 $i = 3$；第二步，在 $A[3], A[4]$ 中有 $A[3] > A[2], A[4] > A[2]$，因此 $j = 4$；第三步，将 $A[2]$ 与 $A[4]$ 进行交换，现在排列变为 2341；最后，将 $A[3], A[4]$ 顺序排列，最终得到下一排列为 2314。

接着，考察计算前一个排列的`previousArrange`函数，只需要对`nextArrange`函数进行微调，算法的步骤如下：

1. 对该排列 $A[1], A[2], \cdots, A[A.length]$ 从后往前遍历，直到 $A[i] < A[i-1]$，此步的时间复杂度为 $O(n)$；

2. 从 $A[i], A[i+1], \cdots, A[A.length]$ 中寻找到 $A[j] < A[i-1]$ 且如果 $j < A.length$，则 $A[j+1] \geq A[i-1]$，此步的时间复杂度为 $O(n-i+1) = O(n)$；

3. 将 $A[i-1]$ 与 $A[j]$ 交换，此步的时间复杂度为 $O(1)$；

4. 将 $A[i], A[i+1], \cdots, A[A.length]$ 排成逆序，即 $A'[i] \geq A'[i+1] \geq \cdots \geq A'[A.length]$。由于在上述步骤后，$A[i], A[i+1], \cdots, A[A.length]$ 是顺序的，因此该步的时间复杂度为 $O(n)$。

例如，若对于排列 2314 寻找前一个排列，即 A 如表2。第一步，从后往前找到 $A[3] < A[2]$，即 $i = 3$；第二步，在 $A[3], A[4]$ 中有 $A[3] < A[2], A[4] \geq A[2]$，因此 $j = 3$；第三步，将 $A[2]$ 与 $A[3]$ 进行交换，现在排列变为 2134；最后，将 $A[3], A[4]$ 逆序排列，最终得到前一排列为 2143。

在这两个方法中，输入为一个`Integer`数组（因此是 Reference），方法对数组进行操作。若要对文件进行输出，只需要对数组进行输出即可，因此扩展性较强。同时，两方法的时间复杂度均为 $O(n)$。

i	1	2	3	4
$A[i]$	1	4	3	1

表 3: 排列 1431

第二问

对于该问，设计了`fullPermutation`方法，其中只需要将集合中的元素排序为顺序数组，并作为参数，使用`nextArrange`函数进行迭代（即上一次的结果作为下一次的参数），并输出每次迭代的结果。

在`fullPermutation`方法中，输入为一个`Set<Integer>`，方法对该集合进行处理，`nextArrange`函数的迭代次数为 $n!$ 次。

第三问

第一问的算法对于有重复数据的场景完全适用。

例如，若对于排列 1431 寻找下一个排列，即 A 如表3。第一步，从后往前找到 $A[2] > A[1]$，即 $i = 2$；第二步，在 $A[2], A[3], A[4]$ 中有 $A[2] > A[1], A[3] > A[1], A[4] \leq A[1]$，因此 $j = 3$；第三步，将 $A[1]$ 与 $A[3]$ 进行交换，现在排列变为 3411；最后，将 $A[2], A[3], A[4]$ 顺序排列，最终得到下一排列为 3114。

同样将序列中的元素排序为顺序数组，并作为参数，使用`nextArrange`函数进行迭代，并输出每次迭代的结果。

对于该问，设计了`fullPermutationWithRepeatedElem`函数，输入为一个`Integer`数组与最大迭代次数`N`，函数对该数组进行处理，`nextArrange`函数的迭代次数为 N 次。

3. 执行过程

第一问

对于第一问，对两个方法分别设计了两组样例：对于`nextArrange`函数，输入排列 1234 与 2143；对于`previousArrange`函数,输入排列 4321 与 2134。若要测试其他样例,可以修改源代码中`testForQuestion1`函数并在`main`函数中调用。得到结果如图1所示。

对于第二问，设计了三个样例，分别是集合 $\{1, 2, 3, 4\}, \{5, 7, 9\}, \{7, 9, 5, 1\}$。若要测试其他样例，可以修改源代码中`testForQuestion2`函数并在`main`函数中调用。得到结果如图2所示。

```
--------------------Test for Question 1--------------------
----------Next Arrangement of {1, 2, 3, 4}----------
1
2
4
3
----------Next Arrangement of {2, 1, 4, 3}----------
2
3
1
4
----------Previous Arrangement of {4, 3, 2, 1}----------
4
3
1
2
----------Previous Arrangement of {2, 1, 3, 4}----------
1
4
3
2
```

图 1: 输出结果

```
--------------------Test for Question 2--------------------
----------Full Permutation of {1, 2, 3, 4}----------
1234 1243 1324 1342 1423 1432 2134 2143 2314 2341 2413 2431 3124 3142 3214 3241 3412 3421 4123 4132 4213 4231 4312 4321
----------Full Permutation of {5, 7, 9}----------
579 597 759 795 957 975
----------Full Permutation of {7, 9, 5, 1}----------
1579 1597 1759 1795 1957 1975 5179 5197 5719 5791 5917 5971 7159 7195 7519 7591 7915 7951 9157 9175 9517 9571 9715 9751
```

图 2: 输出结果

```
--------------------Test for Question 3--------------------
----------Next Arrangement of {1, 4, 3, 1}----------
3
1
1
4
----------Previous Arrangement of {1, 4, 3, 1}----------
1
4
1
3
----------Full Permutation of {1, 1, 3, 4}----------
1134 1143 1314 1341 1413 1431 3114 3141 3411 4113 4131 4311
----------Full Permutation of {1, 1, 2, 3, 4, 4, 4, 5}----------
11234445 11234454 11234544 11235444 11243445 11243454 11243544 11244345 11244354 11244435 11244453 11244534 11244543 112
```

图 3: 输出结果（部分）

对于第三问，首先对`nextArrange`函数和`previousArrange`函数测试排列 1431，再对序列 [1, 1, 3, 4] 与 [1, 1, 2, 3, 4, 4, 4, 5] 分别输出其全排列。两个测试样例的迭代次数（即全排列数）应分别为 $\frac{4!}{2!} = 12, \frac{8!}{2!\times 3!} = 3360$。得到结果如图3所示。

4. 主要函数源代码

- `long factorial(int number)`用于计算`number`的阶乘；
- `void nextArrange(Integer[] nums)`用于计算排列`nums`的后一个排列（适用于包含重复数字的场景）；
- `void previousArrange(Integer[] nums)`用于计算排列`nums`的前一个排列（适用于包含重复数字的场景）；
- `fullPermutation(Set<Integer> set)`用于输出集合`set`的全排列；

- `fullPermutationWithRepeatedElem(Integer[] nums, int numbers_of_arrangements)`用于输出序列`nums`的全排列，其中全排列个数为`numbers_of_arrangements`。

```
public class Main {
    public static long factorial(int number)
    {
        if (number <= 1)
            return 1;
        else
            return number * factorial(number - 1);
    }

    private static void nextArrange(Integer[] nums)
    {
        for (int i = nums.length - 1; i > 0; i--)
        {
            if (nums[i] > nums[i - 1])
            {
                Integer temp = nums[i - 1];
                Integer[] sorted_nums = new Integer[nums.length - i
                    ];
                boolean is_sorted = false;
                for (int j = i; j < nums.length; j++)
                {
                    if ((j + 1 == nums.length || nums[j + 1] <=
                        temp) && !is_sorted)
                    {
                        nums[i - 1] = nums[j];
                        nums[j]  = temp;
                        is_sorted = true;
                    }
                    sorted_nums[nums.length - j - 1] = nums[j];
                }
                System.arraycopy(sorted_nums, 0, nums, i,
                    sorted_nums.length);
```

```
                break;
            }
        }
    }

    private static void previousArrange(Integer[] nums)
    {
        for (int i = nums.length - 1; i > 0; i--)
        {
            if (nums[i] < nums[i - 1])
            {
                Integer temp = nums[i - 1];
                Integer[] sorted_nums = new Integer[nums.length - i
                    ];
                boolean is_sorted = false;
                for (int j = i; j < nums.length; j++)
                {
                    if ((j + 1 == nums.length || nums[j + 1] >=
                        temp) && !is_sorted)
                    {
                        nums[i - 1] = nums[j];
                        nums[j]  = temp;
                        is_sorted = true;
                    }
                    sorted_nums[nums.length - j - 1] = nums[j];
                }
                System.arraycopy(sorted_nums, 0, nums, i,
                    sorted_nums.length);
                break;
            }
        }
    }

    private static void fullPermutation(Set<Integer> set)
    {
```

```
            Integer[] nums = set.toArray(new Integer[0]);
            Arrays.sort(nums);
            long fact = factorial(nums.length);
            for (long i = 0; i < fact; i++)
            {
                for (Integer num : nums) {
                    System.out.print(num);
                }
                System.out.print(' ');
                nextArrange(nums);
            }
            System.out.println();
        }

        private static void fullPermutationWithRepeatedElem(Integer[]
           nums, int numbers_of_arrangements)
        {
            Arrays.sort(nums);
            for (long i = 0; i < numbers_of_arrangements; i++)
            {
                for (Integer num : nums) {
                    System.out.print(num);
                }
                System.out.print(' ');
                nextArrange(nums);
            }
            System.out.println();
        }
    }
```