最长递增子序列实验报告

# 一、实验环境

操作系统：Windows10 专业版 64位

处理器：Intel(R) Core(TM) i7-8700 @3.2GHZ

内存：16G

编程语言： Java 1.8

# 二、算法描述

## 1. 单调递增

在本次实验中将单调递增理解为严格单调递增，即不允许有两个数相等，在本次实验中如没有特别说明，所有的单调递增都是严格单调递增。

单调递增函数的定义：设函数的定义域为I，如果对于定义域I某个区间D上的任意两个自变量的值，当时，都有，那么就说函数在区间D上是增函数。

## 2. 算法描述

### 动态规划O(n2)

记以ai结尾的最长单调递增子序列长度为Ai，输入数组为N，我们可以得到以下递推式：

在计算Ai时，我们需要遍历所有的，比较，如果，那么就是一个候选的递增子序列长度，找出的最大值就可以找到最长递增子序列长度。由于遍历i到N长度的复杂度为O(n)，遍历K到i的复杂度也为O(n)，所以该算法最终的复杂度为O(n2)。

算法的关键代码为：

|  |
| --- |
| for (int i = 1; i < numbers.length; i++) {  maxLen[i] = 1; //用来描述Ai  tags[i] = -1;//用来保存以i结尾最长子序列前一个数的位置  for (int j = 0; j < i; j++) {  if (numbers[j] < numbers[i]) {//如果N[k]<N[i]  int length = maxLen[j]+1;  if (length > maxLen[i]) {  maxLen[i] = length;//更新Ai并保存前一个数的位置  tags[i] = j;  if (length > maxLength) {  maxIndex = i;  maxLength = length;  }  }  };  }  } |

### 贪心算法O(lgn)

由题目的提示，一个长度为i的候选子序列的尾元素至少不比一个长度为i-1的候选子序列的尾元素小，换句话说，对于一个固定长度为i的递增子序列，只有尾元素尽可能小，我们才更有可能找到第i+1个元素放到这个子序列后面成为更长的递增子序列。再换句话说，如果出现了多种长度为i的递增子序列，我们优先选择尾元素最小的那种，这样在后面找到更长递增子序列的概率会更大。

要实现这种方法，我们需要建立一个临时数组minTail，第i个位置用来保存长度为i的递增子序列的最小尾元素的位置。需要一个临时变量maxLen用来记录现在已经构造的最长递增子序列。依次遍历N的每个元素ai，如果ai比现在可以构造的最长子序列的尾元素还要大，即ai>N[minTail[maxLen]]，就说明我们可以构造一个更长的最长子序列，这时可以设置minTail[maxLen+1]=i，maxLen=maxLen+1，如果ai<N[minTail[maxLen]]，说明如果要构造一个长度不超过maxLen的递增子序列，尾元素可以更小，我们需要找到一个合适长度的递增子序列，将尾元素替换为ai。因为“一个长度为i的候选子序列的尾元素至少不比一个长度为i-1的候选子序列的尾元素小”，所以我们需要找到满足ai<N[minTail[j]]的最小的j，将j-1替换为i，换句话说，minTail标记的元素的值必须是递增的。

由于minTail标记的元素值是有序的，所有找到一个合适的插入位置可以使用二分查找，算法复杂度为O(lgn)，遍历一遍所有的元素算法复杂度为O(n)，该算法最终复杂度为O(nlgn)。

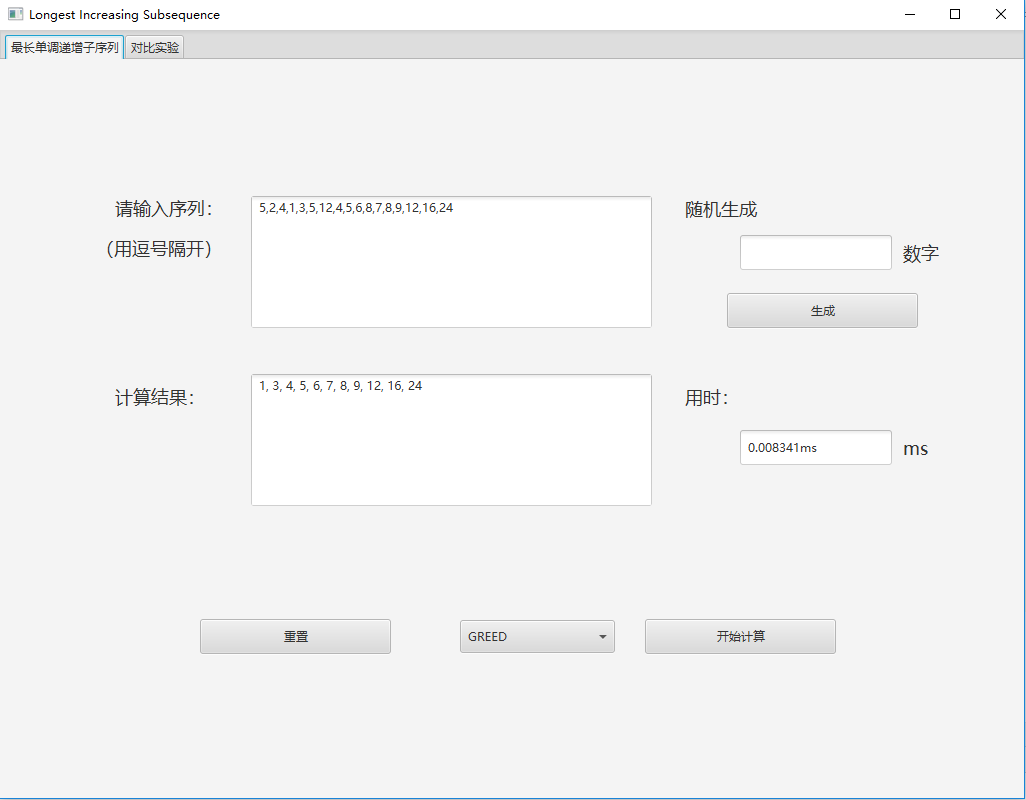
算法的关键代码如下：

|  |
| --- |
| for (int i = 1; i < numbers.length; i++) {  if (numbers[i] > numbers[minTailIndex[maxLen-1]]) {//如果大于最大的尾元素  minTailIndex[maxLen++] = i; //说明可以构造更长的子序列  tags[i] = maxLen; //记录第i个元素可以构造子序列长度，用于寻找最终子序列  }  else { //否则使用二分查找找出替换的位置  int replaceIndex = binarySearch(numbers, minTailIndex, 0,maxLen-1,numbers[i]);  tags[i] = tags[minTailIndex[replaceIndex]];  minTailIndex[replaceIndex] = i;  }  } |

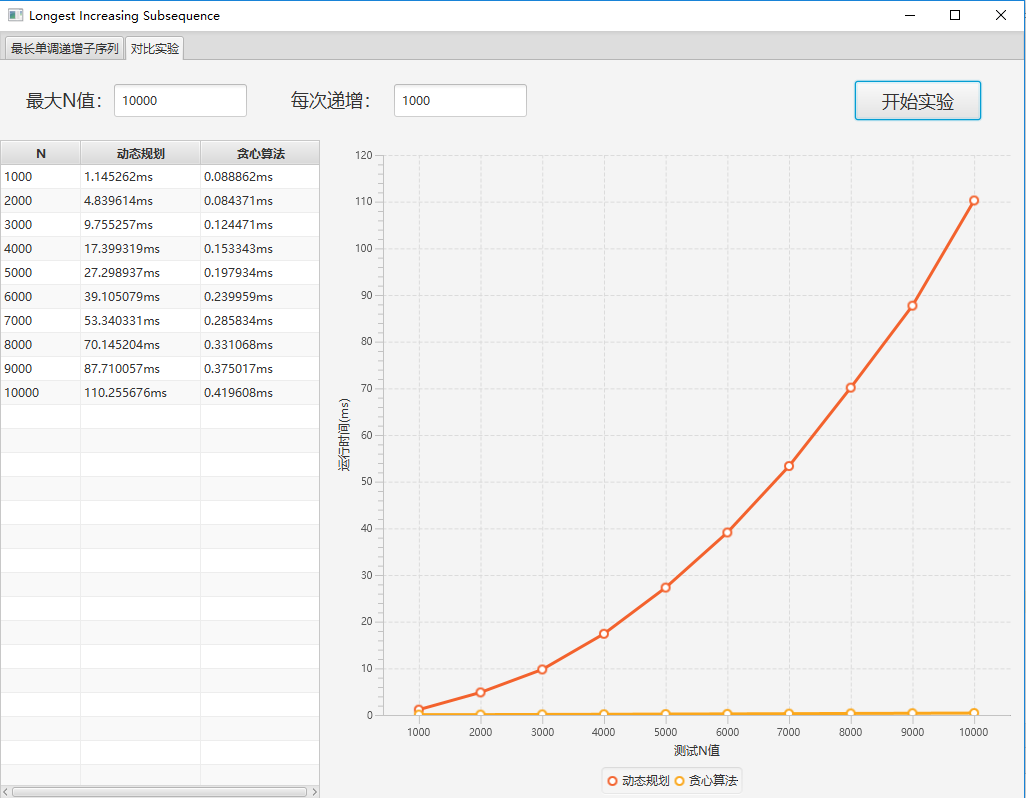
# 三、实验界面

实验程序可以手动输入序列，也可以随机生成0~231-1范围内的整数序列，可以自由选择贪心算法和动态规划算法，结果显示找到的一个最长子序列和计算时间。





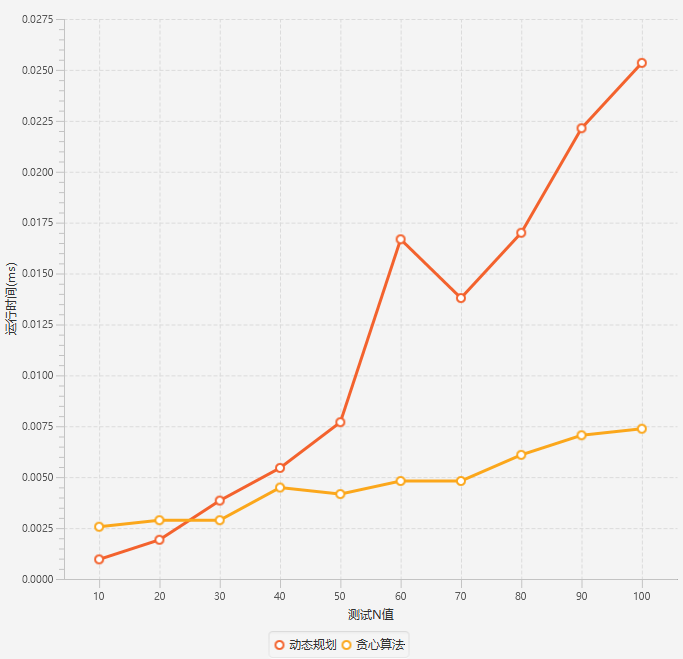
除此以外，实验程序也支持实验对比，实验对比两种算法的界面如下：



# 四、实验结果

## 1. 10~100数据量

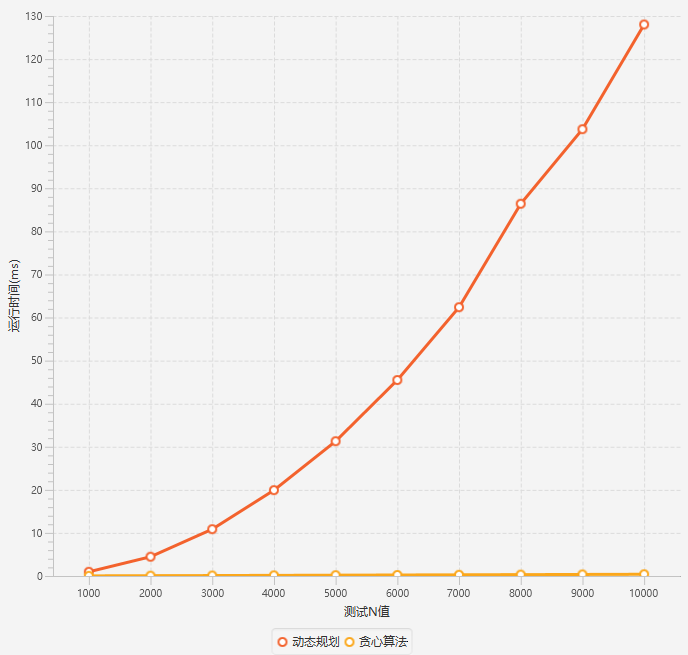
|  |  |  |
| --- | --- | --- |
| N | 动态规划 | 贪心算法 |
| 10 | 9.63E-4ms | 0.002566ms |
| 20 | 0.001925ms | 0.002887ms |
| 30 | 0.00385ms | 0.002887ms |
| 40 | 0.005454ms | 0.004491ms |
| 50 | 0.007699ms | 0.00417ms |
| 60 | 0.016682ms | 0.004812ms |
| 70 | 0.013795ms | 0.004812ms |
| 80 | 0.017002ms | 0.006096ms |
| 90 | 0.022135ms | 0.007058ms |
| 100 | 0.025344ms | 0.007378ms |



由实验可知，数据量较小时两种方法差别不大，数据量小于40时运行时间基本都在0.005ms以下，当数据量超过50时贪心算法运行时间就确保小于动态规划的时间。

## 2. 1000~10000数据量

|  |  |  |
| --- | --- | --- |
| N | 动态规划 | 贪心算法 |
| 1000 | 0.972991ms | 0.034326ms |
| 2000 | 4.470371ms | 0.069614ms |
| 3000 | 10.865231ms | 0.107147ms |
| 4000 | 19.922423ms | 0.146286ms |
| 5000 | 31.281689ms | 0.187669ms |
| 6000 | 45.489673ms | 0.233544ms |
| 7000 | 62.392391ms | 0.277494ms |
| 8000 | 86.411131ms | 0.317915ms |
| 9000 | 103.714851ms | 0.357694ms |
| 10000 | 128.027124ms | 0.403248ms |



数据量逐渐扩大之后贪心算法与动态规划的运行时间差距剧烈扩大，尽管两种算法的理论运行时间差距只是O(n2)与O(nlgn)，但在实际运行时间上贪心算法的效率确高出很多。3.

## 3. 最长子序列结果演示





