拼接所有字符串产生字典顺序最小的大字符串

package test;

import java.util.Arrays;

import java.util.Comparator;

class MergeComparator implements Comparator<String> { @Override public int compare(String arg0, String arg1) { return (arg0 + arg1).compareTo(arg1 + arg0); }

}

public class MergeStringsLowestLexicography { public static String lowestString(String[] strs) { Arrays.sort(strs, new MergeComparator()); String res = ""; for (int i = 0; i != strs.length; i++) { res += strs[i]; } return res; } public static void main(String[] args) { String[] strArr = { "jibw", "ji", "jp", "bw", "jibw" }; String result = lowestString(strArr); System.out.println(result); }

}

证明最关键的步骤是证明这种比较的方式具有传递性。

假设有a,b,c三个字符串，他们有如下的关系：

a.b < b.a

b.c < c.b

所谓传递性证明是指，如果有以上的两个关系，能否证明 a.c < c.a

证明传递性过程：

字符串a,b的连接记为a.b，如果将字符串看做一个K进制数，那么字符串之间的加减乘除都可以按照数字的方式进行。

那么a.b这个字符串中a作为高位，b作为低位，可以进行如下的替换

a.b = a\*(K^length(b))+b。其中a\*(K^length(b))表示，a这个K进制数，向左位移了b的长度。

我们现在把K^length(b)记为moveBit(b)，则a.b = a\*moveBit(b)+b，那么

a.b < b.a => a\*moveBit(b)+b < b\*moveBit(a)+a 不等式1

b.c < c.b => b\*moveBit(c)+c < c\*moveBit(b)+b 不等式2

现在要证明a.c < c.a，也就是证明a\*moveBit(c)+c < c\*moveBit(a)+a

我们把不等式1的左右两边同时减去b再乘以c，则不等式1可以变形为：

a\*moveBit(b)\*c < b\*moveBit(a)\*c+a\*c-b\*c

我们把不等式2的左右两边同时减去b再乘以a，则不等式2可以变形为：

b\*moveBit(c)\*a+c\*a-b\*a < c\*moveBit(b)\*a

现在a,b,c都是K进制数，所以服从乘法交换律。

所以不等式1中的a\*moveBit(b)\*c等于不等式2中的c\*moveBit(b)\*a。

所以，b\*moveBit(c)\*a+c\*a-b\*a < b\*moveBit(a)\*c+a\*c-b\*c

所以，b\*moveBit(c)\*a-b\*a < b\*moveBit(a)\*c-b\*c

所以，a\*moveBit(c)-a < c\*moveBit(a)-c

所以，a\*moveBit(c)+c < c\*moveBit(a)+a => a.c < c.a

证明a.c < c.a完成。

现在我们知道这种比较大小的方式是有传递性的，那么根据这种传递性可知，在一个排序过的序列中，任意两个字符串Str1与Str2，只要Str1排在Str2的前面，就有Str1.Str2 < Str2.Str1。

好，现在我们有了传递性，接下来需要证明：在通过这种排序方式之后所得到的字符串序列中，交换任意两个字符串之后的那个总字符串，都会比未交换前的那个总字符串，拥有更大的字典顺序。

假设通过如上的比较方式，我们得到了一组字符串的序列：…A.M1.M2…M(n-1).M(n).L…，该序列表示，代号为A的字符串之前和代号为L的字符串之后都有若干字符串，A和L中间有若干字符串（用M1..M(n)表示）。

现在我们交换A和L这两个字符串，那么交换之前和交换之后两个总字符串就分别为：

…A.M1.M2…M(n-1).M(n).L… 换之前

…L.M1.M2…M(n-1).M(n).A… 换之后

现在我们需要证明换之后的总字符串字典顺序大于换之前的。

证明：

因为在原始序列中，M1排在L的前面，所以有M1.L<L.M1，所以有…L.M1.M2…M(n-1).M(n).A… > …M1.L.M2…M(n-1).M(n).A…

因为在原始序列中，M2排在L的前面，所以有M2.L<L.M2，所以有…M1.L.M2…M(n-1).M(n).A… > …M1.M2.L…M(n-1).M(n).A…

…

所以有：…M1.M2…M(n-1).M(n).L.A… > …M1.M2…M(n-1).M(n).A.L…

因为在原始序列中，A排在M(n)的前面，所以有A.M(n)<M(n).A，所以有…M1.M2…M(n-1).M(n).A.L… > …M1.M2…M(n-1).A.M(n).L…

…

所以有：…M1.A.M2…M(n-1).M(n).L… > …A.M1.M2…M(n-1).M(n).L…

通过上面不等式之间的连接，可证明换之后>换之前，证明结束，该方法有效。

那么整个解法的时间复杂度就是排序本身的复杂度，O(N\*logN)。

本题的解法非常简单，但是该题的重点，解法有效性的证明确比较复杂。在这里不得不向读者进行一点提醒，这道题的解题方法，可以划进贪心算法的范畴，这种有效的比较方式，就是我们的贪心策略。

所以算法的时间复杂度就是排序的时间复杂度，O(N\*logN)

从5随机到7随机及其扩展

【题目】

给定一个等概率随机产生1~5的随机函数rand1To5如下：

|  |  |
| --- | --- |
| 1  2  3 | public int rand1To5() {      return (int) (Math.random() \* 5) + 1;  } |

除此之外不能使用任何额外的随机机制，请用rand1To5实现等概率随机产生1~7的随机函数rand1To7。

【补充题目】

给定一个以p概率产生0，以1-p概率产生1的随机函数rand01p如下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | public int rand01p() {      // you can change p as you like      double p = 0.83;      return Math.random() < p ? 0 : 1;  } |

除此之外不能使用任何额外的随机机制，请用rand01p实现等概率随机产生1~6的随机函数rand1To6。

【进阶题目】

给定一个等概率随机产生1~M的随机函数rand1ToM如下：

|  |  |
| --- | --- |
| 1  2  3 | public int rand1ToM(int m) {      return (int) (Math.random() \* m) + 1;  } |

除此之外不能使用任何额外的随机机制。有两个输入参数分别为m和n，请用rand1ToM(m)实现等概率随机产生1~n的随机函数rand1ToN。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | public int rand1To5() {      return (int) (Math.random() \* 5) + 1;  }  public int rand1To7() {      int num = 0;      do {          num = (rand1To5() - 1) \* 5 + rand1To5() - 1;      } while (num > 20);      return num % 7 + 1;  }            public int rand01p() {      // you can change p as you like      double p = 0.83;      return Math.random() < p ? 0 : 1;  }  public int rand01() {      int num;      do {          num = rand01p();      } while (num == rand01p());      return num == 1 ? 1 : 0;  }  public int rand0To3() {      return rand01() \* 2 + rand01();  }  public int rand1To6() {      int num = 0;      do {          num = rand0To3() \* 4 + rand0To3();      } while (num > 11);      return num % 6 + 1;  }          public int rand1ToM(int m) {      return (int) (Math.random() \* m) + 1;  }  public int rand1ToN(int n, int m) {      int[] nMSys = getMSysNum(n - 1, m);      int[] randNum = getRanMSysNumLessN(nMSys, m);      return getNumFromMSysNum(randNum, m) + 1;  }  // 把value转成m进制的数  public int[] getMSysNum(int value, int m) {      int[] res = new int[32];      int index = res.length - 1;      while (value != 0) {          res[index--] = value % m;          value = value / m;      }      return res;  }  // 等概率随机产生一个0~nMsys范围上的数，只不过是m进制表达的。  public int[] getRanMSysNumLessN(int[] nMSys, int m) {      int[] res = new int[nMSys.length];      int start = 0;      while (nMSys[start] == 0) {          start++;      }      int index = start;      boolean lastEqual = true;      while (index != nMSys.length) {          res[index] = rand1ToM(m) - 1;          if (lastEqual) {              if (res[index] > nMSys[index]) {                  index = start;                  lastEqual = true;                  continue;              } else {                  lastEqual = res[index] == nMSys[index];              }          }          index++;      }      return res;  }  // 把m进制的数转成10进制  public int getNumFromMSysNum(int[] mSysNum, int m) {      int res = 0;      for (int i = 0; i != mSysNum.length; i++) {          res = res \* m + mSysNum[i];      }      return res;  } |

【题目】

给定一个无序数组arr，求出需要排序的最短子数组长度。

例如：

arr = [1，5，3，4，2，6，7]

返回4，因为只有[5，3，4，2]需要排序。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public int getMinLength(int[] arr) {      if (arr == null || arr.length < 2) {          return 0;      }      int min = arr[arr.length - 1];      int noMinIndex = -1;      for (int i = arr.length - 2; i != -1; i--) {          if (arr[i] > min) {              noMinIndex = i;          } else {              min = Math.min(min, arr[i]);          }      }      if (noMinIndex == -1) {          return 0;      }      int max = arr[0];      int noMaxIndex = -1;      for (int i = 1; i != arr.length; i++) {          if (arr[i] < max) {              noMaxIndex = i;          } else {              max = Math.max(max, arr[i]);          }      }      return noMaxIndex - noMinIndex + 1;  } |

最大的leftMax与rightMax之差的绝对值

【题目】

给定一个长度为N(N>1)的整型数组arr，可以划分成左右两个部分，左部分arr[0..K]，右部分arr[K+1..N-1]，K可以取值的范围是[0,N-2]。求这么多划分方案中，左部分中的最大值减去右部分最大值的绝对值，最大是多少？

例如[2,7,3,1,1]，当左部分为[2,7]，右部分为[3,1,1]时，左部分中的最大值减去右部分最大值的绝对值为4。当左部分为[2,7,3]，右部分为[1,1]时，左部分中的最大值减去右部分最大值的绝对值为6。还有很多划分方案，但最终返回6。

方法一，时间复杂度O(N^2)，额外空间复杂度O(1)。这是最笨的方法，在数组的每个位置(i)都做一次这种划分，找到arr[0..i]的最大值maxLeft，找到arr[i+1..N-1]的最大值maxRight，然后计算两个值相减的绝对值。每次划分都这样求一次，自然可以得到最大的相减的绝对值。具体请参看如下代码中的maxABS1方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public int maxABS1(int[] arr) {      int res = Integer.MIN\_VALUE;      int maxLeft = 0;      int maxRight = 0;      for (int i = 0; i != arr.length - 1; i++) {          maxLeft = Integer.MIN\_VALUE;          for (int j = 0; j != i + 1; j++) {              maxLeft = Math.max(arr[j], maxLeft);          }          maxRight = Integer.MIN\_VALUE;          for (int j = i + 1; j != arr.length; j++) {              maxRight = Math.max(arr[j], maxRight);          }          res = Math.max(Math.abs(maxLeft - maxRight), res);      }      return res;  } |

方法二，时间复杂度O(N)，额外空间复杂度O(N)。使用预处理数组的方法，先从左到右遍历一次生成lArr，lArr[i]表示arr[0..i]中的最大值。再从右到左遍历一次生成rArr，rArr[i]表示arr[i..N-1]中的最大值。最后一次遍历看看哪种划分的情况下可以得到两部分最大的相减的绝对值，因为预处理数组已经保存了所有划分的max值，所以过程得到了加速。具体请参看如下代码中的maxABS2方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public int maxABS2(int[] arr) {      int[] lArr = new int[arr.length];      int[] rArr = new int[arr.length];      lArr[0] = arr[0];      rArr[arr.length - 1] = arr[arr.length - 1];      for (int i = 1; i < arr.length; i++) {          lArr[i] = Math.max(lArr[i - 1], arr[i]);      }      for (int i = arr.length - 2; i > -1; i--) {          rArr[i] = Math.max(rArr[i + 1], arr[i]);      }      int max = 0;      for (int i = 0; i < arr.length - 1; i++) {          max = Math.max(max, Math.abs(lArr[i] - rArr[i + 1]));      }      return max;  } |

方法三，最优解，时间复杂度O(N)，额外空间复杂度O(1)。先求整个arr的最大值max，因为max是全局最大值，所以不管怎么划分，max要么会成为左部分的最大值，要么会成为右部分的最大值。如果max作为左部分的最大值，接下来我们只要让右部分的最大值尽量小就可以了。右部分的最大值怎么尽量小呢？右部分只含有arr[N-1]的时候就是尽量小的时候。同理，如果max作为右部分的最大值，只要让左部分的最大值尽量小就可以了，左部分只含有arr[0]的时候就是尽量小的时候。所以整个求解过程会变得异常简单，具体请参看如下代码中的maxABS3方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public int maxABS3(int[] arr) {      int max = Integer.MIN\_VALUE;      for (int i = 0; i < arr.length; i++) {          max = Math.max(arr[i], max);      }      return max - Math.min(arr[0], arr[arr.length - 1]);  } |

【题目】

现在有一种新的二叉树节点类型如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Node {      public int value;      public Node left;      public Node right;      public Node parent;      public Node(int data) {          this.value = data;      }  } |

该结构比普通二叉树节点结构多了一条指向父节点的parent指针。假设有一棵Node类型的节点组成的二叉树，树中每个节点的parent指针都正确的指向自己的父节点，头节点的parent指向null。只给一个在二叉树中的某个节点node，请实现返回node的后继节点的函数。在二叉树的中序遍历的序列中，node的下一个节点叫做node的后继节点。

例如，下图的二叉树：

\_\_\_\_\_\_\_\_\_\_6\_\_\_\_\_\_\_\_\_\_

  / \

  \_\_\_3\_\_\_   \_\_\_9\_\_\_

 /       \  /       \

1\_\_       4\_\_   \_\_8  10

\  \  /

2   5 7

中序遍历的结果为：1，2，3，4，5，6，7，8，9，10

所以节点1的后继为节点2，节点2的后继为节点3，…，节点10的后继为null

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public Node getNextNode(Node node) {      if (node == null) {          return node;      }      if (node.right != null) {          return getLeftMost(node.right);      } else {          Node parent = node.parent;          while (parent != null && parent.left != node) {              node = parent;              parent = node.parent;          }          return parent;      }  }  public Node getLeftMost(Node node) {      if (node == null) {          return node;      }      while (node.left != null) {          node = node.left;      }      return node;  } |