# 各种排序算法

不稳定的排序算法：快排、希尔、堆排、选择

1. **import** java**.**util**.**List**;**
2. **import** java**.**util**.**ArrayList**;**
3. **import** java**.**util**.**Arrays**;**
4. public class Test **{**
5. static int**[]** arr **=** **{**22**,**18**,**9**,**8**,**7**,**1**,**5**,**4**,**3**,**2**,**6**};**
6. public static void main**(**String**[]** args**)** **{**
7. // bubbleSort(arr);
8. // selectSort(arr);
9. // InsertSort(arr);
10. // qSort(arr, 0, arr.length-1);
11. // mergeSort(arr, 0, arr.length-1);
12. mergeSort2**(**arr**);**
13. // heapSort(arr);
14. // shellSort(arr);
15. System**.**out**.**println**(**Arrays**.**toString**(**arr**));**
16. // System.out.println(getLcs("123", "a2b3"));
17. **}**
19. static void bubbleSort**(**int**[]** arr**)** **{**
20. int len **=** arr**.**length**;**
21. **for** **(**int i**=**len**-**2**;** i**>=**0**;** i**--)** **{**
22. **for** **(**int j**=**0**;** j**<=**i**;** j**++)** **{**
23. **if** **(**arr**[**j**]** **>** arr**[**j**+**1**])** **{**
24. swap**(**arr**,** j**,** j**+**1**);**
25. **}**
26. **}**
27. **}**
28. **}**
29. static void selectSort**(**int**[]** arr**)** **{**
30. int len **=** arr**.**length**;**
31. **for** **(**int i**=**0**;** i**<**len**-**1**;** i**++)** **{**
32. int min **=** i**;**
33. **for** **(**int j**=**i**+**1**;** j**<**len**;** j**++)** **{**
34. **if** **(**arr**[**j**]** **<** arr**[**min**])** **{**
35. min **=** j**;**
36. **}**
37. **}**
38. **if** **(**min **!=** i**)**
39. swap**(**arr**,** min**,** i**);**
40. **}**
41. **}**
43. static void InsertSort**(**int**[]** arr**)** **{**
44. int len **=** arr**.**length**;**
45. **for** **(**int i**=**0**;** i**<**len**;** i**++)** **{**
46. int k **=** i**;**
47. **while(**k **>=** 1**)** **{**
48. **if** **(**arr**[**k**-**1**]** **>** arr**[**k**])** **{**
49. swap**(**arr**,** k**,** k**-**1**);**
50. **}** **else** **{**
51. **break;**
52. **}**
53. k**--;**
54. **}**
56. // for (int j=i; j>=1; j--) {
57. // if (arr[j-1] > arr[j]) {
58. // swap(arr, j, j-1);
59. // } else
60. // break;
61. // }
63. **}**
64. **}**

67. static int partion**(**int**[]** arr**,** int left**,** int right**)** **{**
68. int mid **=** left **-** 1**;**
69. // int id = left;
70. **while** **(**left **<=** right**)** **{**
71. **if** **(**arr**[**left**]** **<=** arr**[**right**])** **{**
72. swap**(**arr**,** left**,** **++**mid**);**
73. **}**
74. left**++;**
75. **}**
76. **return** mid**;**
77. **}**
79. static void qSort**(**int**[]** arr**,** int left**,** int right**)** **{**
80. **if** **(**left **>=** right**)** **return;**
81. int mid **=** partion**(**arr**,** left**,** right**);**
82. qSort**(**arr**,** left**,** mid**-**1**);**
83. qSort**(**arr**,** mid**+**1**,** right**);**
84. **}**
86. static void mergeSort**(**int**[]** arr**,** int left**,** int right**)** **{**
87. **if** **(**left **>=** right**)** **return;**
88. int mid **=** **(**left **+** right**)** **/** 2**;**
89. mergeSort**(**arr**,** left**,** mid**);**
90. mergeSort**(**arr**,** mid**+**1**,** right**);**
91. merge**(**arr**,** left**,** mid**,** right**);**
92. **}**
94. static void mergeSort2**(**int**[]** arr**)** **{**
95. int len **=** arr**.**length**;**
96. int sl **=** 1**,** dl **=** 1**;**
97. **while** **(**sl **<** len**)** **{**
98. dl **=** sl **\*** 2**;**
99. int i **=** 0**;**
100. **while** **(**i **+** dl **<** len**)** **{**
101. merge**(**arr**,** i**,** i**+**sl**-**1**,** i**+**dl**-**1**);**
102. i **+=** dl**;**
103. **}**
104. **if** **(**i **+** sl **<** len**)** **{**
105. merge**(**arr**,** i**,** i**+**sl**-**1**,** len**-**1**);**
106. **}**
107. sl **=** dl**;**
108. **}**
109. **}**
111. private static void merge**(**int**[]** arr**,** int left**,** int mid**,** int right**)** **{**
112. int**[]** temp **=** **new** int**[**right**-**left**+**1**];**
113. int id **=** 0**;**
114. int l **=** left**,** r **=** mid**+**1**;**
115. **while** **(**l **<=** mid **&&** r **<=** right**)** **{**
116. **if** **(**arr**[**l**]** **<** arr**[**r**])** **{**
117. temp**[**id**++]** **=** arr**[**l**++];**
118. **}** **else** **{**
119. temp**[**id**++]** **=** arr**[**r**++];**
120. **}**
121. **}**
123. **while** **(**l **<=** mid**)** **{**
124. temp**[**id**++]** **=** arr**[**l**++];**
125. **}**
127. **while** **(**r **<=** right**)** **{**
128. temp**[**id**++]** **=** arr**[**r**++];**
129. **}**
131. **for** **(**int i**=**0**;** i**<**temp**.**length**;** i**++)** **{**
132. arr**[**i**+**left**]** **=** temp**[**i**];**
133. **}**
134. **}**
136. static void heapSort**(**int**[]** arr**)** **{**
137. int len **=** arr**.**length**;**
138. **for** **(**int i**=**len**/**2**;** i**>=**0**;** i**--)** **{**
139. heapAdjust**(**arr**,** i**,** len**);**
140. **}**
142. **for** **(**int i**=**len**-**1**;** i**>**0**;** i**--)** **{**
143. swap**(**arr**,** i**,** 0**);**
144. heapAdjust**(**arr**,** 0**,** i**);**
145. **}**
146. **}**
147. private static void heapAdjust**(**int**[]** arr**,** int i**,** int len**)** **{**
148. int child**;**
149. **while** **((**child **=** i**\***2 **+** 1**)** **<** len**)** **{**
150. **if** **(**child **+** 1 **<** len **&&** arr**[**child**+**1**]** **>** arr**[**child**])** **{**
151. child**++;**
152. **}**
154. **if** **(**arr**[**child**]** **>** arr**[**i**])** **{**
155. swap**(**arr**,** i**,** child**);**
156. i **=** child**;**
157. **}** **else** **{**
158. **break;**
159. **}**
160. **}**
161. **}**
162. static void shellSort**(**int**[]** arr**)** **{**
163. int len **=** arr**.**length**;**
164. int feet **=** len **/** 2**;**
165. **while** **(**feet **>** 0**)** **{**
166. **for** **(**int i**=**feet**;** i**<**len**;** i**++)** **{**
167. int insert **=** i**;**
168. **while** **(**insert **>=** feet**)** **{**
169. **if** **(**arr**[**insert**-**feet**]** **>** arr**[**insert**])** **{**
170. swap**(**arr**,** insert**,** insert**-**feet**);**
171. insert **-=** feet**;**
172. **}** **else** **{**
173. **break;**
174. **}**
175. **}**
176. **}**
177. feet **/=** 2**;**
178. **}**
179. **}**
181. private static void swap**(**int**[]** arr**,** int a**,** int b**)** **{**
182. // TODO Auto-generated method stub
183. // arr[a] = arr[a] ^ arr[b];
184. // arr[b] = arr[b] ^ arr[a];
185. // arr[a] = arr[a] ^ arr[b];
187. **if** **(**arr**[**a**]** **==** arr**[**b**])** **return;**
188. arr**[**a**]** **^=** arr**[**b**];**
189. arr**[**b**]** **^=** arr**[**a**];**
190. arr**[**a**]** **^=** arr**[**b**];**
192. // int t = arr[a];
193. // arr[a] = arr[b];
194. // arr[b] = t;
195. **}**

# LCS算法

获取最长公共子串，不连续的。

static int getLcs**(**String s1**,** String s2**)** **{**

int len1 **=** s1**.**length**();**

int len2 **=** s2**.**length**();**

int**[][]** dp **=** **new** int**[**len1**+**1**][**len2**+**1**];**

**for** **(**int i**=**1**;** i**<=**len1**;** i**++)** **{**

**for** **(**int j**=**1**;** j**<=**len2**;** j**++)** **{**

**if** **(**s1**.**charAt**(**i**-**1**)** **==** s2**.**charAt**(**j**-**1**))** **{**

dp**[**i**][**j**]** **=** dp**[**i**-**1**][**j**-**1**]** **+** 1**;**

**}** **else** **{**

dp**[**i**][**j**]** **=** Math**.**max**(**dp**[**i**-**1**][**j**],** dp**[**i**][**j**-**1**]);**

**}**

**}**

**}**

getLcsStr**(**dp**,** s1**,** s2**);**

**return** dp**[**len1**][**len2**];**

**}**

static void getLcsStr**(**int**[][]** dp**,** String s1**,** String s2**)** **{**

int len1 **=** s1**.**length**();**

int len2 **=** s2**.**length**();**

StringBuilder sBuilder **=** **new** StringBuilder**();**

**while** **(**len1 **>** 0 **&&** len2 **>** 0**)** **{**

**if** **(**s1**.**charAt**(**len1**-**1**)** **==** s2**.**charAt**(**len2**-**1**))** **{**

sBuilder**.**insert**(**0**,** s1**.**charAt**(**len1**-**1**));**

len1**--;**

len2**--;**

**}** **else** **if** **(**dp**[**len1**][**len2**]** **==** dp**[**len1**-**1**][**len2**])** **{**

len1**--;**

**}** **else** **{**

len2**--;**

**}**

**}**

System**.**out**.**println**(**sBuilder**.**toString**());**

**}**

# 重建二叉树

前序遍历：根 左 右 ，中序遍历：左 中 右 ，后序遍历：左右根

public TreeNode reConstructBinaryTree**(**int **[]** pre**,**int **[]** in**)** **{**

**if** **(**pre **==** **null** **||** in **==** **null)** **return** **null;**

**return** getTree**(**pre**,** 0**,** pre**.**length**-**1**,** in**,** 0**,** in**.**length**-**1**);**

**}**

private TreeNode getTree**(**int**[]** pre**,** int preLeft**,** int preRight**,**

int**[]** in**,** int inLeft**,** int inRight**)** **{**

int rootValue **=** pre**[**preLeft**];**

TreeNode rootNode **=** **new** TreeNode**(**rootValue**);**

**if** **(**preLeft **==** preRight **&&** inLeft **==** inRight**)** **{**

**return** rootNode**;**

**}**

int rootId **=** 0**;**

**while** **(**rootId **<=** inRight **&&** in**[**rootId**]** **!=** rootValue**)** **{**

rootId**++;**

**}**

//关键点在这里，递归遍历的边界确定

int preLeftLen **=** rootId **-** inLeft**;**

int preLeftEnd **=** preLeft **+** preLeftLen**;**

**if** **(**preLeftLen **>** 0**)** **{**

rootNode**.**left **=** getTree**(**pre**,** preLeft**+**1**,** preLeftEnd**,**

in**,** inLeft**,** rootId**-**1**);**

**}**

**if** **(**rootId **<** inRight**)** **{**

rootNode**.**right **=** getTree**(**pre**,** preLeftEnd**+**1**,** preRight**,**

in**,** rootId**+**1**,** inRight**);**

**}**

**return** rootNode**;**

**}**

**class** TreeNode {

**int** val;

TreeNode left;

TreeNode right;

TreeNode(**int** x) { val = x; }

}

# 二叉树的各种遍历方法 非递归

public void preSer**(**TreeNode root**,** List**<**Integer**>** list**)** **{**//先序,确实已参数形式传进去，会好做些

**if** **(**root **==** **null)** **{**

**return** **;**

**}**

Stack**<**TreeNode**>** stack **=** **new** Stack**<>();**

TreeNode cur **=** root**;**

stack**.**push**(**cur**);**

**while** **(!**stack**.**isEmpty**())** **{**

cur **=** stack**.**pop**();**

list**.**add**(**cur**.**val**);**

**if** **(**cur**.**right **!=** **null)** stack**.**push**(**cur**.**right**);**

**if** **(**cur**.**left **!=** **null)** stack**.**push**(**cur**.**left**);**

**}**

**}**

public void midSer**(**TreeNode root**,** List**<**Integer**>** list**)** **{**//中序,确实已参数形式传进去，会好做些

**if** **(**root **==** **null)** **{**

**return** **;**

**}**

Stack**<**TreeNode**>** stack **=** **new** Stack**<>();**

TreeNode cur **=** root**;**

**while** **(!**stack**.**isEmpty**()** **||** cur **!=** **null)** **{**

**while** **(**cur **!=** **null)** **{**

stack**.**push**(**cur**);**

cur **=** cur**.**left**;**

**}**

**if** **(!**stack**.**isEmpty**())** **{**

cur **=** stack**.**pop**();**

list**.**add**(**cur**.**val**);**

cur **=** cur**.**right**;**

**}**

**}**

**}**

public void aftSer**(**TreeNode root**,** List**<**Integer**>** list**)** **{**//后序,确实已参数形式传进去，会好做些

**if** **(**root **==** **null)** **{**

**return** **;**

**}**

Stack**<**TreeNode**>** stack **=** **new** Stack**<>();**

Stack**<**Integer**>** stackReverse **=** **new** Stack**<>();**

TreeNode cur **=** root**;**

stack**.**push**(**cur**);**

**while** **(!**stack**.**isEmpty**())** **{**

cur **=** stack**.**pop**();**

stackReverse**.**push**(**cur**.**val**);**

**if** **(**cur**.**left **!=** **null)** stack**.**push**(**cur**.**left**);**

**if** **(**cur**.**right **!=** **null)** stack**.**push**(**cur**.**right**);**

**}**

**while** **(!**stackReverse**.**isEmpty**())** **{**

list**.**add**(**stackReverse**.**pop**());**

**}**

**}**

# LRU算法

1. **import** java**.**util**.**HashMap**;**
2. **import** java**.**util**.**LinkedHashMap**;**
3. **import** java**.**util**.**LinkedList**;**
4. public class LRU **{**
5. public static void main**(**String**[]** args**)** **{**
6. // TODO Auto-generated method stub
7. HashMap hashMap **=** **new** HashMap**<**Integer**,** String**>();**
8. hashMap **=** **new** HashMap**<**Integer**,** Integer**>();**
9. hashMap**.**put**(**1**,** 2**);**
10. hashMap**.**put**(**3**,** 2**);**
11. hashMap**.**put**(**2**,** 2**);**
12. hashMap**.**put**(**7**,** 2**);**
13. hashMap**.**put**(**4**,** 2**);**
14. System**.**out**.**println**(**hashMap**);**
16. LinkedHashMap**<**Integer**,** Integer**>** linkedHashMap **=**
17. **new** LinkedHashMap**<>(**10**,** 0.5f**,** **true);**
18. linkedHashMap**.**put**(**1**,** 2**);**
19. linkedHashMap**.**put**(**3**,** 2**);**
20. linkedHashMap**.**put**(**2**,** 2**);**
21. linkedHashMap**.**put**(**7**,** 2**);**
22. linkedHashMap**.**put**(**4**,** 2**);**
23. linkedHashMap**.**get**(**3**);**
24. System**.**out**.**println**(**linkedHashMap**);**
26. LruCache**<**Integer**,** Integer**>** lruCache **=** **new** LruCache**(**3**);**
27. lruCache**.**put**(**2**,** 3**);**
28. lruCache**.**put**(**3**,** 3**);**
29. lruCache**.**put**(**4**,** 3**);**
30. lruCache**.**put**(**5**,** 3**);**
31. lruCache**.**put**(**6**,** 3**);**
32. lruCache**.**get**(**4**);**
33. lruCache**.**put**(**7**,** 3**);**
34. System**.**out**.**println**(**lruCache**);**
36. LinkedList**<**Integer**>** list **=** **new** LinkedList**<>();**
37. list**.**add**(**2**);**
38. list**.**add**(**3**);**
39. list**.**add**(**4**);**
40. list**.**add**(**5**);**
41. list**.**add**(**9**);**
42. System**.**out**.**println**(**list**);**
43. //remove的参数为Object和int,如果传入泛型为Integer.使用remove会混淆，
44. list**.**remove**(**2**);**
45. System**.**out**.**println**(**list**);**
46. list**.**remove**(**Integer**.**valueOf**(**5**));**
47. System**.**out**.**println**(**list**);**
49. LRUCache2**<**Integer**,** Integer**>** lruCache2 **=** **new** LRUCache2**(**3**);**
50. lruCache2**.**add**(**1**,** 3**);**
51. lruCache2**.**add**(**2**,** 4**);**
52. lruCache2**.**add**(**3**,** 5**);**
53. lruCache2**.**get**(**3**);**
54. lruCache2**.**add**(**6**,** 6**);**
55. lruCache2**.**add**(**7**,** 7**);**
56. System**.**out**.**println**(**lruCache2**);**
58. //更为实用的LRU
59. int cacheSize **=** 3**;**
60. int capacity **=** **(**int**)** **(**Math**.**ceil**(**cacheSize**/**0.75**)+**1**);**
61. LinkedHashMap**<**Object**,** Object**>** lru **=** **new** LinkedHashMap**<**Object**,** Object**>**
62. **(**capacity**,** 0.75f**,** **true)** **{**
63. private static final long serialVersionUID **=** 1L**;**
64. @Override
65. protected boolean removeEldestEntry**(**java**.**util**.**Map**.**Entry**<**Object**,** Object**>** eldest**)** **{**
66. // TODO Auto-generated method stub
67. **return** size**()** **>** cacheSize**;**
68. **}**
69. **};**
70. **}**

73. **}**
74. //对应单纯的整数输入 可另 key=value=输入整数
75. class LRUCache2 **<**K**,** V**>** **{**
76. private int cacheSize**;**
77. private HashMap**<**K**,** V**>** map**;**
78. private LinkedList**<**K**>** linkedList**;**
80. public LRUCache2**(**int size**)** **{**
81. cacheSize **=** size**;**
82. map **=** **new** HashMap**<>();**
83. linkedList **=** **new** LinkedList**<>();**
84. **}**
86. public void add**(**K key**,** V value**)** **{**
87. //缓存已经满了 需要清理
88. **if** **(**curSize**()** **>=** cacheSize**)** **{**
89. //已经存在，命中，移到链表尾
90. **if** **(**map**.**containsKey**(**key**))** **{**
91. linkedList**.**remove**(**key**);**
92. linkedList**.**add**(**key**);**
93. **}** **else** **{**//没有命中，移除最旧（链表头）的数据,添加新的数据
94. K old **=** linkedList**.**removeFirst**();**
95. linkedList**.**add**(**key**);**
96. map**.**remove**(**old**);**
97. map**.**put**(**key**,** value**);**
98. **}**
99. **}** **else** **{**//正常添加
100. linkedList**.**add**(**key**);**
101. map**.**put**(**key**,** value**);**
102. **}**
103. **}**
105. public V get**(**K key**)** **{**
106. V value **=** map**.**get**(**key**);**
107. //已经存在，命中，移到链表尾
108. **if** **(**value **!=** **null)** **{**
109. linkedList**.**remove**(**key**);**
110. linkedList**.**add**(**key**);**
111. **}**
112. **return** value**;**
113. **}**
115. public V remove**(**K key**)** **{**
116. V v **=** map**.**get**(**key**);**
117. linkedList**.**remove**(**key**);**
118. map**.**remove**(**key**);**
119. **return** v**;**
120. **}**

123. @Override
124. public String toString**()** **{**
125. // TODO Auto-generated method stub
126. **return** linkedList**.**toString**();**
127. **}**
129. private int curSize**()** **{**
130. **return** linkedList**.**size**();**
131. **}**
132. **}**
133. class LruCache**<**K**,** V**>** **extends** LinkedHashMap**<**K**,** V**>** **{**
134. private final int cacheSize**;**
135. public LruCache**(**int size**)** **{**
136. // TODO Auto-generated constructor stub
137. **super((**int**)** **(**Math**.**ceil**(**size **/** 0.75**)** **+** 1**),** 0.75f**,** **true);**
138. cacheSize **=** size**;**
139. **}**
141. @Override
142. protected boolean removeEldestEntry**(**java**.**util**.**Map**.**Entry**<**K**,** V**>** eldest**)** **{**
143. // TODO Auto-generated method stub
144. **return** size**()** **>** cacheSize**;**
145. **}**
146. **}**

# 翻转单词顺序列

如：pig is good -> good is pig

先翻转所有字符，再逐个单词翻转，逐个反转单词时候需要把握单词的首尾位置

private static String reverse**(**String string**)** **{**

char**[]** cs **=** string**.**toCharArray**();**

int len **=** cs**.**length**;**

int head **=** 0**,** tail **=** 0**,** id **=** 0**;**

reversePart**(**cs**,** 0**,** len**-**1**);**

**while** **(**id **<** len**)** **{**

**while** **(**id **<** len **&&** cs**[**id**]** **==** ' '**)** id**++;**

head **=** id**;**

**while** **(**id **<** len **&&** cs**[**id**]** **!=** ' '**)** id**++;**

tail **=** id **-** 1**;**

reversePart**(**cs**,** head**,** tail**);**

**}**

**return** **new** String**(**cs**);**

**}**

private static void reversePart**(**char**[]** cs**,** int l**,** int r**)** **{**

**if** **(**l **>=** r**)** **return;**

**while** **(**l **<** r**)** **{**

char t **=** cs**[**l**];**

cs**[**l**]** **=** cs**[**r**];**

cs**[**r**]** **=** t**;**

l**++;**

r**--;**

**}**

**}**

# 二维数组查找数值

该二维数组每行元素从左到右递增，每列元素从上到下递增。判断某个数值是否存在其中

根据规律，从右上角开始 或者左下角：

public boolean Find**(**int **[][]** array**,**int target**)** **{**

int row **=** array**.**length**,** column **=** array**[**0**].**length**;**

int r **=** 0**,** c **=** column **-** 1**;**

**while** **(**r **<** row **&&** c **>=** 0**)** **{**

**if** **(**array**[**r**][**c**]** **>** target**)** **{**

c**--;**

**}** **else** **if** **(**array**[**r**][**c**]** **<** target**)** **{**

r**++;**

**}** **else** **{**

**return** **true;**

**}**

**}**

**return** **false;**

**}**

# 旋转数组的最小数字

例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

//4,5,6,7,1,2,3

static public int minNumberInRotateArray**(**int **[]** array**)** **{**

**if** **(**array **==** **null** **||** array**.**length **==** 0**)** **return** 0**;**

int index **=** 0**;**

int left **=** 0**,** right **=** array**.**length**-**1**;**

**while** **(**left **<** right**)** **{**

**if** **(**right **-** left **==** 1**)** **{**

index **=** right**;**

**break;**

**}**

int mid **=** **(**right **+** left**)** **/** 2**;**

**if** **(**array**[**left**]** **==** array**[**mid**]** **&&** array**[**mid**]** **==** array**[**right**])** **{**

**return** minInorder**(**array**,** left**,** right**);**

**}**

**if** **(**array**[**mid**]** **>=** array**[**left**])** **{**

left **=** mid**;**

**}**

**if** **(**array**[**mid**]** **<=** array**[**right**])** **{**

right **=** mid**;**

**}**

**}**

**return** array**[**index**];**

**}**

static private int minInorder**(**int**[]** arr**,** int left**,** int right**)** **{**

**for** **(**int i**=**left**;** i**<**right**;** i**++)** **{**

**if** **(**arr**[**i**]** **>** arr**[**i**+**1**])** **{**

**return** arr**[**i**+**1**];**

**}**

**}**

ret

# 跳台阶

是斐波那契数列的变种，斐波那契数列：0,1,1,2,3……第一个0表示第0项。f(0) =0,f(1)=1

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

0 1 2 3……

public int JumpFloor**(**int n**)** **{**

**if** **(**n **==** 0**)** **return** 0**;**

**if** **(**n **==** 1**)** **return** 1**;**

**if** **(**n **==** 2**)** **return** 2**;**

int a1**=**1**,**a2**=**2**,**a3**=**0**;**

**for** **(**int i**=**3**;** i**<=**n**;** i**++)** **{**

a3 **=** a2 **+** a1**;**

a1 **=** a2**;**

a2 **=** a3**;**

**}**

**return** a3**;**

**}**

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

公式 f(n) = 2 (n-1)

public int JumpFloorII**(**int n**)** **{**

**if** **(**n **==** 0**)** **return** 0**;**

int res**=**1**;**

**for** **(**int i**=**1**;** i**<**n**;** i**++)** **{**

res **\*=** 2**;**

**}**

**return** res**;**

**}**

# 二进制中1的个数

n&(n-1)清除n的二进制最右位的1

public int NumberOf1**(**int n**)** **{**

int count **=** 0**;**

**while** **(**n **!=** 0**)** **{**

count**++;**

n **=** n **&** **(**n**-**1**);**

**}**

**return** count**;**

**}**

变种1：判断一个整数是不是2的整数次方：其二进制只有一个1

变种2：对2个整数m,n，求m需要改变其二进制位几次才能变成n。 求m^n中1的个数

# 数值的整数次方

给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

需要考虑base为0和exp为负数的情况

**import** java**.**util**.**HashMap**;**

public class Test02 **{**

public static void main**(**String**[]** args**)** **{**

System**.**out**.**println**(**Double**.**compare**(**1**-**0.9**,** 0.1**));**

System**.**out**.**println**(**equalDouble**(**1**-**0.9**,** 0.1**));**

System**.**out**.**println**(**Power**(**2**,** **-**3**));**

**}**

static boolean invalidInput **=** **false;**

static public double Power**(**double base**,** int exponent**)** **{**

invalidInput **=** **false;**

**if** **(**equalDouble**(**base**,** 0.0**)** **&&** exponent **<** 0**)** **{**

invalidInput **=** **true;**

**return** 0**;**

**}**

int positiveExp **=** Math**.**abs**(**exponent**);**

double res **=** powerWithUnsignedExp**(**base**,** positiveExp**);**

**if** **(**exponent **<** 0**)** res **=** 1**/**res**;**

**return** res**;**

**}**

static public double powerWithUnsignedExp**(**double base**,** int exp**)** **{**

**if** **(**exp **==** 0**)** **return** 1**;**

**if** **(**exp **==** 1**)** **return** base**;**

double res **=** powerWithUnsignedExp**(**base**,** exp **/** 2**);**

res **\*=** res**;**

**if** **((**exp **&** 0x1**)** **!=** 0**)**

res **\*=** base**;**

**return** res**;**

**}**

static boolean equalDouble**(**double a**,** double b**)** **{**

**if** **((**a **-** b **<** 0.0000001**)** **&&**

**(**a **-** b **>** **-**0.0000001**))** **{**

**return** **true;**

**}**

**return** **false;**

**}**

**}**

# 打印1到最大的n位数

输入数字n，打印从1到最大的n位十进制数字，如输入3，打印1到999。

char数组全排列

package review**;**

package review**;**

**import** java**.**util**.**Arrays**;**

**import** java**.**util**.**HashMap**;**

public class Test02 **{**

public static void main**(**String**[]** args**)** **{**

printDigits**(**15**);**

**}**

static void printDigits**(**int n**)** **{**

char**[]** digits **=** **new** char**[**n**];**

dfsPrint**(**0**,** digits**);**

**}**

//dfs组合数字 每一位有0~9种可能，n位

private static void dfsPrint**(**int n**,** char**[]** digits**)** **{**

**if** **(**n **==** digits**.**length**)** **{**

//System.out.println(Arrays.toString(digits));

outDigit**(**digits**);**

**return;**

**}**

**for** **(**char c**=**'0'**;** c**<=**'9'**;** c**++)** **{**

digits**[**n**]** **=** c**;**

dfsPrint**(**n**+**1**,** digits**);**

**}**

**}**

//打印数字 忽略前面的0

private static void outDigit**(**char**[]** ds**)** **{**

int id **=** 0**;**

**if** **(**ds**[**0**]** **==** '0'**)** **{**

id **=** 1**;**

**while** **(**id **<** ds**.**length **&&** ds**[**id**]** **==** '0'**)**id**++;**

**}**

**while** **(**id **<** ds**.**length**)** **{**

System**.**out**.**print**(**ds**[**id**++]);**

**}**

System**.**out**.**println**();**

**}**

**}**

# 调整数组顺序使奇数位于偶数前面

所有的奇数位于数组的前半部分，偶数位于数组的后半部分

package review**;**

**import** java**.**util**.**Arrays**;**

public class Test02 **{**

public static void main**(**String**[]** args**)** **{**

int**[]** arr **=** **{**1**,**2**,**3**,**4**,**5**,**7**,**8**,**9**};**

reOrderArray**(**arr**);**

System**.**out**.**println**(**Arrays**.**toString**(**arr**));**

**}**

static public void reOrderArray**(**int **[]** array**)** **{**

//顺序改变

// int odds = -1;

// int id = 0;

// while (id < array.length) {

// if (check(array[id])) {

// swap(array, id, ++odds);

// }

// id++;

// }

//次序不变 类似插入排序

**for** **(**int i**=**1**;** i**<**array**.**length**;** i**++)** **{**

int t **=** array**[**i**];**

**if** **(**check**(**t**))** **{//是奇数开始检查**

**for** **(**int j**=**i**;** j**>=**1**;** j**--)** **{**

**if** **(!**check**(**array**[**j**-**1**]))** **{//前面是偶数，交换**

swap**(**array**,** j**,** j**-**1**);**

**}** **else** **{**

**break;**

**}**

**}**

**}**

**}**

**}**

private static void swap**(**int**[]** array**,** int id**,** int i**)** **{**

**if** **(**array**[**id**]** **==** array**[**i**])** **return;**

array**[**id**]** **^=** array**[**i**];**

array**[**i**]** **^=** array**[**id**];**

array**[**id**]** **^=** array**[**i**];**

**}**

//判断奇数

private static boolean check**(**int i**)** **{**

**return** **(**i **&** 0x1**)** **==** 1 **?** **true** **:** **false;**

**}**

**}**

# 删除链表某一节点 O(1)时间复杂度

思路：单向链表，删除del节点，找到del节点的next节点，另del.val = next.val，del.next = next.next；即先复制next节点到del节点，然后再删除了next节点。需要考虑到2种特殊情况：1、被删除节点是尾节点，只能使用正常遍历进行删除；2、如果该链表仅有一个节点，即头节点，则直接删除头节点即可。

时间复制度为 （（n-1）O(1) + O（n））/ n；还是O(1)。

# 链表中倒数第k个结点

该题关键是考察代码的鲁棒性，head为空，k=0,或者k大于链表长度。

对与链表的题目，应先思考可否使用2个不同步长指针进行处理。

如求链表的中间节点，可令p1一次走一步，p2一次走2步。

public ListNode FindKthToTail**(**ListNode head**,**int k**)** **{**

**if** **(**head **==** **null** **||** k **==** 0**)** **return** **null;**

ListNode fast **=** head**;**

ListNode target **=** head**;**

//先走k-1步

**for** **(**int i**=**0**;** i**<**k**-**1**;** i**++)** **{**

**if** **(**fast**.**next **!=** **null)** **{**

fast **=** fast**.**next**;**

**}** **else** **{**

**return** **null;**

**}**

**}**

**while** **(**fast**.**next **!=** **null)** **{**

fast **=** fast**.**next**;**

target **=** target**.**next**;**

**}**

**return** target**;**

**}**

# 反转链表

输入一个链表，反转链表后，输出链表的所有元素。

static public ListNode ReverseList**(**ListNode head**)** **{**

**if** **(**head **==** **null)** **return** **null;**

**if** **(**head**.**next **==** **null)** **return** head**;**

ListNode pre **=** **null;**

ListNode cur **=** head**;**

ListNode next **=** **null;**

**while** **(true)** **{**

next **=** cur**.**next**;**

**if** **(**next **!=** **null)** **{**

cur**.**next **=** pre**;**

pre **=** cur**;**

cur **=** next**;**

**}** **else** **{**

cur**.**next **=** pre**;**

head **=** cur**;**

**break;**

**}**

**}**

**return** head**;**

**}**

# 合并两个排序的链表

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则

注意2个链表为空的情况

public ListNode Merge**(**ListNode list1**,**ListNode list2**)** **{**

**//已经包括都为null的情况**

**if** **(**list1 **==** **null)**

**return** list2**;**

**if** **(**list2 **==** **null)**

**return** list1**;**

ListNode root **=** **null;**

**if** **(**list1**.**val **<=** list2**.**val**)** **{**

root **=** list1**;**

root**.**next **=** Merge**(**list1**.**next**,** list2**);**

**}** **else** **{**

root **=** list2**;**

root**.**next **=** Merge**(**list1**,** list2**.**next**);**

**}**

**return** root**;**

**}**

# 树的子结构

输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

class TreeNode **{**

int val **=** 0**;**

TreeNode left **=** **null;**

TreeNode right **=** **null;**

public TreeNode**(**int val**)** **{**

**this.**val **=** val**;**

**}**

**}**

static public boolean HasSubtree**(**TreeNode root1**,**TreeNode root2**)** **{**

boolean result **=** **false;**

**if** **(**root1 **!=** **null** **&&** root2 **!=** **null)** **{**

**if** **(**root1**.**val **==** root2**.**val**)**

result **=** doTree1HasTree2**(**root1**,** root2**);**

**if** **(**result **==** **false)**

result **=** HasSubtree**(**root1**.**left**,** root2**);**

**if** **(**result **==** **false)**

result **=** HasSubtree**(**root1**.**right**,** root2**);**

**}**

**return** result**;**

**}**

private static boolean doTree1HasTree2**(**TreeNode root1**,** TreeNode root2**)** **{**

**if** **(**root2 **==** **null)**

**return** **true;**

**if** **(**root1 **==** **null)**

**return** **false;**

**if** **(**root1**.**val **!=** root2**.**val**)**

**return** **false;**

**return** doTree1HasTree2**(**root1**.**left**,** root2**.**left**)**

**&&** doTree1HasTree2**(**root1**.**right**,** root2**.**right**);**

**}**

# 二叉树的镜像

public void Mirror**(**TreeNode root**)** **{**

**if** **(**root **==** **null)** **return;**

**if** **(**root**.**left **==** **null** **&&** root**.**right **==** **null)** **return;**

TreeNode temp **=** root**.**left**;**

root**.**left **=** root**.**right**;**

root**.**right **=** temp**;**

Mirror**(**root**.**left**);**

Mirror**(**root**.**right**);**

**}**

# 包含min函数的栈

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的min函数。

**import** java**.**util**.**Stack**;**

public class Solution **{**

Stack**<**Integer**>** mainStack **=** **new** Stack**<>();**

Stack**<**Integer**>** minStack **=** **new** Stack**<>();**

int min **=** Integer**.**MAX\_VALUE**;**

public void push**(**int node**)** **{**

mainStack**.**push**(**node**);**

min **=** Math**.**min**(**node**,** min**);**

minStack**.**push**(**min**);**

**}**

public void pop**()** **{**

**if** **(**mainStack **==** **null)** **return;**

mainStack**.**pop**();**

minStack**.**pop**();**

**}**

public int top**()** **{**

**return** mainStack**.**peek**();**

**}**

public int min**()** **{**

**return** minStack**.**peek**();**

**}**

**}**

# 栈的压入、弹出序列

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4，5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）

public boolean IsPopOrder**(**int **[]** pushA**,**int **[]** popA**)** **{**

Stack**<**Integer**>** stack **=** **new** Stack**<>();**

stack**.**push**(**pushA**[**0**]);**

int idPush **=** 1**,** idPop **=** 0**;**

**while** **(true)** **{**

//这里不能省略，否则peek可能报错

**if** **(**stack**.**isEmpty**())** **{**

**if** **(**idPush **<** pushA**.**length**)** **{**

stack**.**push**(**pushA**[**idPush**++]);**

**}**

**}**

**while** **(**stack**.**peek**()** **!=** popA**[**idPop**])** **{**

**if** **(**idPush **<** pushA**.**length**)** **{**

stack**.**push**(**pushA**[**idPush**++]);**

**}** **else** **{**//已经完全压入，但仍没有找到匹配值，匹配失败

**return** **false;**

**}**

**}**

stack**.**pop**();**//匹配一次，进行下一次匹配

idPop**++;**

**if** **(**idPop **==** popA**.**length**)** **{**

**return** **true;**

**}**

**}**

**}**

该题可以用于生成数列的出栈序列，先进行全排列，再用该函数进行判断是否符合出栈规则即可。

static public void getPopSer**(**int**[]** push**)** **{**

int**[]** book **=** **new** int**[**push**.**length**];**

dfs**(**0**,** push**,new** int**[**push**.**length**],** book**);**

**}**

private static void dfs**(**int n**,** int**[]** push**,** int**[]** out**,** int**[]** book**)** **{**

**if** **(**n **==** push**.**length**)** **{**

**if** **(**isPopOrder**(**push**,** out**))** **{**

System**.**out**.**println**(**Arrays**.**toString**(**out**));**

get**++;**

**}**

all**++;**

**return;**

**}**

**for** **(**int i**=**0**;** i**<**push**.**length**;** i**++)** **{**

**if** **(**book**[**i**]** **==** 0**)** **{**

out**[**n**]** **=** push**[**i**];**

book**[**i**]** **=** 1**;**

dfs**(**n**+**1**,** push**,** out**,** book**);**

book**[**i**]** **=** 0**;**

// out[n] = 0;

**}**

**}**

**}**

static public boolean isPopOrder**(**int **[]** pushA**,**int **[]** popA**)** **{**

Stack**<**Integer**>** stack **=** **new** Stack**<>();**

int idPush **=** 0**,** idPop **=** 0**;**

**while** **(true)** **{**

//这里不能省略，否则peek可能报错

**if** **(**stack**.**isEmpty**())** **{**

**if** **(**idPush **<** pushA**.**length**)** **{**

stack**.**push**(**pushA**[**idPush**++]);**

**}**

**}**

**while** **(**stack**.**peek**()** **!=** popA**[**idPop**])** **{**

**if** **(**idPush **<** pushA**.**length**)** **{**

stack**.**push**(**pushA**[**idPush**++]);**

**}** **else** **{**//已经完全压入，但仍没有找到匹配值，匹配失败

**return** **false;**

**}**

**}**

stack**.**pop**();**//匹配一次，进行下一次匹配

idPop**++;**

**if** **(**idPop **==** popA**.**length**)** **{**

**return** **true;**

**}**

**}**

**}**

# 火车进站

给定一个正整数N代表火车数量，0<N<10，接下来输入火车入站的序列，一共N辆火车，每辆火车以数字1-9编号。要求以字典序排序输出火车出站的序列号。

1. 求出全排列
2. 判断是否符合出栈规则
3. 将数列按字典排序
4. 输出，每行最后一个字符后不能加空格

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**Arrays**;**

**import** java**.**util**.**Collections**;**

**import** java**.**util**.**List**;**

**import** java**.**util**.**Scanner**;**

**import** java**.**util**.**Stack**;**

public class Main **{**

public static void main**(**String**[]** args**)** **{**

// TODO Auto-generated method stub

Scanner in **=** **new** Scanner**(**System**.**in**);**

**while** **(**in**.**hasNextInt**())** **{**

int n **=** in**.**nextInt**();**

int**[]** push **=** **new** int**[**n**];**

**for** **(**int i**=**0**;** i**<**n**;** i**++)** **{**

push**[**i**]** **=** in**.**nextInt**();**

**}**

// Arrays.sort(push);

int**[]** book **=** **new** int**[**push**.**length**];**

dfs**(**0**,** push**,new** int**[**push**.**length**],** book**);**

Collections**.**sort**(**lists**);**

// System.out.println(lists);

**for** **(**int i**=**0**;** i**<**lists**.**size**();** i**++)** **{**

System**.**out**.**println**(**lists**.**get**(**i**));**

**}**

**}**

**}**

static List**<**String**>** lists **=** **new** ArrayList**<>();**

private static void dfs**(**int n**,** int**[]** push**,** int**[]** out**,** int**[]** book**)** **{**

**if** **(**n **==** push**.**length**)** **{**

**if** **(**isPopOrder**(**push**,** out**))** **{**

lists**.**add**(**change**(**out**));**

**}**

**return;**

**}**

**for** **(**int i**=**0**;** i**<**push**.**length**;** i**++)** **{**

**if** **(**book**[**i**]** **==** 0**)** **{**

out**[**n**]** **=** push**[**i**];**

book**[**i**]** **=** 1**;**

dfs**(**n**+**1**,** push**,** out**,** book**);**

book**[**i**]** **=** 0**;**

**}**

**}**

**}**

private static String change**(**int**[]** out**)** **{**

StringBuilder sb **=** **new** StringBuilder**();**

**for** **(**int i**=**0**;** i**<**out**.**length**-**1**;** i**++)** **{**

sb**.**append**(**out**[**i**]+**" "**);**

**}**

sb**.**append**(**out**[**out**.**length**-**1**]);**

**return** sb**.**toString**();**

**}**

private static void print**(**Integer**[]** out**)** **{**

**for** **(**int i**=**0**;** i**<**out**.**length**;** i**++)** **{**

System**.**out**.**print**(**out**[**i**]+**" "**);**

**}**

System**.**out**.**println**();**

**}**

static public boolean isPopOrder**(**int **[]** pushA**,**int**[]** out**)** **{**

Stack**<**Integer**>** stack **=** **new** Stack**<>();**

int idPush **=** 0**,** idPop **=** 0**;**

**while** **(true)** **{**

//这里不能省略，否则peek可能报错

**if** **(**stack**.**isEmpty**())** **{**

**if** **(**idPush **<** pushA**.**length**)** **{**

stack**.**push**(**pushA**[**idPush**++]);**

**}**

**}**

**while** **(**stack**.**peek**()** **!=** out**[**idPop**])** **{**

**if** **(**idPush **<** pushA**.**length**)** **{**

stack**.**push**(**pushA**[**idPush**++]);**

**}** **else** **{**//已经完全压入，但仍没有找到匹配值，匹配失败

**return** **false;**

**}**

**}**

stack**.**pop**();**//匹配一次，进行下一次匹配

idPop**++;**

**if** **(**idPop **==** out**.**length**)** **{**

**return** **true;**

**}**

**}**

**}**

**}**

# 构造回文

给定一个字符串s，你可以从中删除一些字符，使得剩下的串是一个回文串。如何删除才能使得回文串最长呢？  
输出需要删除的字符个数。

求该字符串和逆序字符串的最长公共子序列

private static int getLen**(**String str**)** **{**

StringBuilder sb **=** **new** StringBuilder**(**str**);**

sb **=** sb**.**reverse**();**

int len **=** str**.**length**();**

int**[][]** dp **=** **new** int**[**len**+**1**][**len**+**1**];**

**for** **(**int i**=**1**;** i**<=**len**;** i**++)** **{**

**for** **(**int j**=**1**;** j**<=**len**;** j**++)** **{**

**if** **(**str**.**charAt**(**i**-**1**)** **==** sb**.**charAt**(**j**-**1**))** **{**

dp**[**i**][**j**]** **=** dp**[**i**-**1**][**j**-**1**]** **+** 1**;**

**}** **else** **{**

dp**[**i**][**j**]** **=** Math**.**max**(**dp**[**i**-**1**][**j**],** dp**[**i**][**j**-**1**]);**

**}**

**}**

**}**

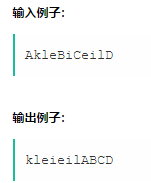
// System.out.println(dp[len][len]);

**return** len **-** dp**[**len**][**len**];**

**}**

# 字符移位

小Q最近遇到了一个难题：把一个字符串的大写字母放到字符串的后面，各个字符的相对位置不变，且不能申请额外的空间。  
你能帮帮小Q吗？



使用类似插入排序的思路

private static String change**(**String str**)** **{**

char**[]** cs **=** str**.**toCharArray**();**

**for** **(**int i**=**1**;** i**<**cs**.**length**;** i**++)** **{**

char ch **=** cs**[**i**];**

**if** **(**Character**.**isLowerCase**(**ch**))** **{**

**for** **(**int j**=**i**;** j**>=**1**;** j**--)** **{**

**if** **(**Character**.**isUpperCase**(**cs**[**j**-**1**]))** **{**

swap**(**cs**,** j**-**1**,** j**);**

**}**

**}**

**}**

**}**

**return** **new** String**(**cs**);**

**}**

private static void swap**(**char**[]** cs**,** int i**,** int j**)** **{**

char t **=** cs**[**i**];**

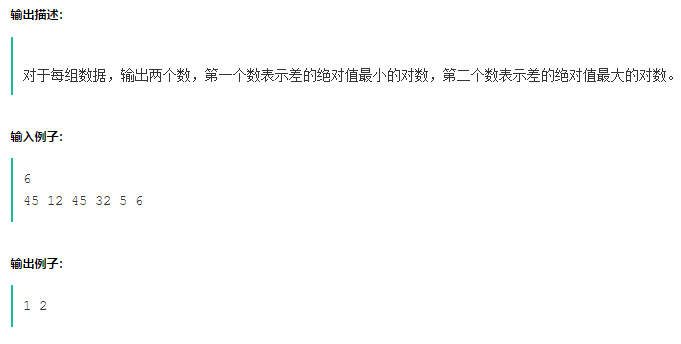
cs**[**i**]** **=** cs**[**j**];**

cs**[**j**]** **=** t**;**

**}**

# 有趣的数字

小Q今天在上厕所时想到了这个问题：有n个数，两两组成二元组，差的绝对值最小的有多少对呢？差的绝对值最大的呢？



解题思路：

1. 使用快排排序
2. 找到差的绝对值最大的对数：找到最小值个数m，最大值个数n，另m\*n;
3. 差值最小的，由于是有序数组，必定是相邻的差值较小，故由快排后的有序数组生成差值数组（即相邻的两两相减）。如果差值数组中0，则查看差值数组中连续0的组数，可以由排列组合知识计算总的差值最小的对数；如果差值数组中没有0，则只需计算差值数组中有多少个最小值即可。注：差值数组必定都是非负数。

# 确定字符互异

请实现一个算法，确定一个字符串的所有字符是否全都不同。这里我们要求不允许使用额外的存储结构。

给定一个string **iniString**，请返回一个bool值,**True**代表所有字符全都不同，**False**代表存在相同的字符。保证字符串中的字符为ASCII字符。字符串的长度小于等于3000。

1. 使用哈希法，int[256],另字符对应的下标值++，如果大于1则返回false
2. 使用快排，在partition算法中判断是否有相等情况。

# 从上往下打印二叉树

BFS 广度优先遍历

public ArrayList**<**Integer**>** PrintFromTopToBottom**(**TreeNode root**)** **{**

ArrayList**<**Integer**>** result **=** **new** ArrayList**<>();**

**if** **(**root **==** **null)** **return** result**;**

LinkedList**<**TreeNode**>** queue **=** **new** LinkedList**<>();**

queue**.**add**(**root**);**

**while** **(!**queue**.**isEmpty**())** **{**

TreeNode temp **=** queue**.**poll**();**

result**.**add**(**temp**.**val**);**

**if** **(**temp**.**left **!=** **null)** queue**.**add**(**temp**.**left**);**

**if** **(**temp**.**right **!=** **null)** queue**.**add**(**temp**.**right**);**

**}**

**return** result**;**

**}**

# 二叉搜索树的后序遍历序列

public boolean VerifySquenceOfBST**(**int **[]** sequence**)** **{**

**if** **(**sequence **==** **null** **||** sequence**.**length **==** 0**)** **return** **false;**

**return** verify**(**sequence**,** 0**,** sequence**.**length**-**1**);**

**}**

private boolean verify**(**int**[]** sequence**,** int left**,** int right**)** **{**

**if** **(**left **>=** right**)** **return** **true;**

int rightStart **=** left**;**

int root **=** sequence**[**right**];**

**while** **(**rightStart **<** right **&&** sequence**[**rightStart**]** **<** root**)** rightStart**++;**

**for** **(**int i**=**rightStart**;** i**<**right**;** i**++)** **{**

**if** **(**sequence**[**i**]** **<** root**)** **{**

**return** **false;**

**}**

**}**

**return** verify**(**sequence**,** left**,** rightStart**-**1**)**

**&&** verify**(**sequence**,** rightStart**,** right**-**1**);**

**}**

# 二叉树中和为某一值的路径

输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

该题使用了二叉树遍历算法，前序遍历，dfs算法。

public class Solution **{**

ArrayList**<**ArrayList**<**Integer**>>** lists **=** **new** ArrayList**<**ArrayList**<**Integer**>>();**

public ArrayList**<**ArrayList**<**Integer**>>** FindPath**(**TreeNode root**,**int target**)** **{**

**if** **(**root **==** **null)** **return** lists**;**

path**(**root**,** 0**,** **new** ArrayList**<**Integer**>(),** target**);**

**return** lists**;**

**}**

//root准备遍历的节点

//sum是已经遍历的路径和，不包括当前结点

private void path**(**TreeNode root**,** int sum**,** ArrayList**<**Integer**>** path**,** int target**)** **{**

**if** **(**root **==** **null)** **return;**

//到达叶结点

**if** **(**root**.**left **==** **null** **&&** root**.**right **==** **null)** **{**

**if** **(**sum **+** root**.**val **==** target**)** **{**

path**.**add**(**root**.**val**);**//加入结点

lists**.**add**(new** ArrayList**<>(**path**));** //记录正确路径

path**.**remove**(**path**.**size**()-**1**);**//移除结点，恢复状态，不影响之后的遍历使用

**}**

**}**

//遍历左结点

path**.**add**(**root**.**val**);**

path**(**root**.**left**,** sum**+**root**.**val**,** path**,** target**);**

path**.**remove**(**path**.**size**()-**1**);**

//遍历右结点

path**.**add**(**root**.**val**);**

path**(**root**.**right**,** sum**+**root**.**val**,** path**,** target**);**

path**.**remove**(**path**.**size**()-**1**);**

**}**

**}**

# 复杂链表的复制

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

/\*

public class RandomListNode {

int label;

RandomListNode next = null;

RandomListNode random = null;

RandomListNode(int label) {

this.label = label;

}

}

\*/

**import** java**.**util**.\*;**

public class Solution **{**

public RandomListNode Clone**(**RandomListNode pHead**)**

**{**

**if** **(**pHead **==** **null)** **return** **null;**

RandomListNode newHead **=** **new** RandomListNode**(**pHead**.**label**);**

HashMap**<**RandomListNode**,** RandomListNode**>** map **=** **new** HashMap**<>();**

map**.**put**(**pHead**,** newHead**);**

RandomListNode newPre **=** newHead**;**

RandomListNode cur **=** pHead**.**next**;**

**while** **(**cur **!=** **null)** **{**

RandomListNode newCur **=** **new** RandomListNode**(**cur**.**label**);**

newPre**.**next **=** newCur**;**

map**.**put**(**cur**,** newCur**);**

cur **=** cur**.**next**;**

newPre **=** newCur**;**

**}**

RandomListNode newIt **=** newHead**;**

RandomListNode it **=** pHead**;**

**while** **(**newIt **!=** **null)** **{**

newIt**.**random **=** map**.**get**(**it**.**random**);**

newIt **=** newIt**.**next**;**

it **=** it**.**next**;**

**}**

**return** newHead**;**

**}**

**}**

public class Solution **{**

public RandomListNode Clone**(**RandomListNode pHead**){**

**if(**pHead**==null)**

**return** **null;**

RandomListNode pCur **=** pHead**;**

//复制next 如原来是A->B->C 变成A->A'->B->B'->C->C'

**while(**pCur**!=null){**

RandomListNode node **=** **new** RandomListNode**(**pCur**.**label**);**

node**.**next **=** pCur**.**next**;**

pCur**.**next **=** node**;**

pCur **=** node**.**next**;**

**}**

pCur **=** pHead**;**

//复制random pCur是原来链表的结点 pCur.next是复制pCur的结点

**while(**pCur**!=null){**

**if(**pCur**.**random**!=null)**

pCur**.**next**.**random **=** pCur**.**random**.**next**;**

pCur **=** pCur**.**next**.**next**;**

**}**

RandomListNode head **=** pHead**.**next**;**

RandomListNode cur **=** head**;**

pCur **=** pHead**;**

//拆分链表 pCur**.**next因为之前的复制 所以不为空

**while(**pCur**!=null){**

pCur**.**next **=** pCur**.**next**.**next**;**

**if(**cur**.**next**!=null)**

cur**.**next **=** cur**.**next**.**next**;**

cur **=** cur**.**next**;**

pCur **=** pCur**.**next**;**

**}**

**return** head**;**

**}**

**}**

# 二叉搜索树与双向链表

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

二叉搜索树的中序遍历结果就是一个递增序列，只需要保存上一次节点即可，让上一节点的next指向当前遍历节点即可。

/\*\*

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

\*/

public class Solution **{**

//二叉搜索树的中序遍历结果就是一个递增序列，只需要保存上一次节点即可

TreeNode head**,** pre**;**

public TreeNode Convert**(**TreeNode pRootOfTree**)** **{**

**if** **(**pRootOfTree **==** **null)** **return** **null;**

converSub**(**pRootOfTree**);**

**return** head**;**

**}**

private void converSub**(**TreeNode node**)** **{**

**if** **(**node **==** **null)** **return;**

converSub**(**node**.**left**);**

**if** **(**head **==** **null)** **{**

head **=** node**;**

pre **=** node**;**

**}** **else** **{**

node**.**left **=** pre**;**//当前结点大于上一节点，且在链表中相邻

pre**.**right **=** node**;**

pre **=** node**;**

**}**

converSub**(**node**.**right**);**

**}**

**}**

# 字符串的排列

输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串abc,则打印出由字符a,b,c所能排列出来的所有字符串abc,acb,bac,bca,cab和cba。 结果请按字母顺序输出。

1. 全排列用dfs算法生成
2. 顺序用treeSet控制，同时清除重复的字符串。

**import** java**.**util**.\*;**

public class Solution **{**

static ArrayList**<**String**>** result**;**

static Set**<**String**>** set**;**

static public ArrayList**<**String**>** Permutation**(**String str**)** **{**

set **=** **new** TreeSet**<>();**

result **=** **new** ArrayList**<>();** //每次都需要初始化

**if** **(**str **==** **null** **||** str**.**length**()** **==** 0**)** **return** result**;**

char**[]** cs **=** str**.**toCharArray**();**

dfs**(**cs**,** **new** char**[**str**.**length**()],**

**new** int**[**str**.**length**()],** 0**);**

**for** **(**String string **:** set**)** **{**

result**.**add**(**string**);**

**}**

**return** result**;**

**}**

static void dfs**(**char**[]** cs**,** char**[]** out**,** int**[]** book**,** int id**)** **{**

**if** **(**id **==** cs**.**length**)** **{**

set**.**add**(new** String**(**out**));**

**return;**

**}**

**for** **(**int i**=**0**;** i**<**cs**.**length**;** i**++)** **{**

**if** **(**book**[**i**]** **==** 0**)** **{**

book**[**i**]** **=** 1**;**

out**[**id**]** **=** cs**[**i**];**

dfs**(**cs**,** out**,** book**,** id**+**1**);**

book**[**i**]** **=** 0**;**

**}**

**}**

**}**

**}**

# 8皇后问题

要求在8\*8的棋盘上摆放8个皇后，使其不能相互攻击，即任意2个皇后不在同一行或者同一列，或同一对角线上

关键是对问题的抽象，用一个数组queen[8],放置0~7的全排列，其中queen[i]表示位于第i行皇后的列号。这样就保证任意2个皇后不在同一行，同一列了。如果处于同一对角线则满足|i-j| = queen[i] – queen[j]。

# 数组中出现次数超过一半的数字

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

public class Solution **{**

public int MoreThanHalfNum\_Solution**(**int **[]** arr**)** **{**

**if** **(**arr **==** **null** **||** arr**.**length **==** 0**)** **return** 0**;**

**if** **(**arr**.**length **==** 1**)** **return** arr**[**0**];**

int num **=** 1**,** val **=** arr**[**0**];**

**for** **(**int i**=**1**;** i**<**arr**.**length**;** i**++)** **{**

**if** **(**arr**[**i**]** **==** val**)** **{**

num**++;**

**}** **else** **{**

num**--;**

**if** **(**num **==** 0**)** **{**

val **=** arr**[**i**];**

num **=** 1**;**

**}**

**}**

**}**

//检查是否出现次数超过一半

int times **=** 0**;**

**for** **(**int i**=**0**;** i**<**arr**.**length**;** i**++)** **{**

**if** **(**arr**[**i**]** **==** val**)**

times**++;**

**}**

**if** **(**times **<=** arr**.**length **/** 2**)** **return** 0**;**

**return** val**;**

**}**

**}**

有成熟的O(n)算法可以找到数组中任意第K大的数字：

该题可以转换为找数组中第N = （len/2）大的数字，然后对该数字验证。find(int[] arr, int id)是找到arr数组排序后下标为id的数字，算法复杂度为O(n)。

public class Solution **{**

public int MoreThanHalfNum\_Solution**(**int **[]** arr**)** **{**

**if** **(**arr **==** **null** **||** arr**.**length **==** 0**)** **return** 0**;**

**if** **(**arr**.**length **==** 1**)** **return** arr**[**0**];**

//超过一半长度的下标

int middle **=** arr**.**length **/** 2**;**

int val **=** find**(**arr**,** middle**);**

//检查是否出现次数超过一半

int times **=** 0**;**

**for** **(**int i**=**0**;** i**<**arr**.**length**;** i**++)** **{**

**if** **(**arr**[**i**]** **==** val**)**

times**++;**

**}**

**if** **(**times **<=** arr**.**length **/** 2**)** **return** 0**;**

**return** val**;**

**}**

static int find**(**int**[]** arr**,** int id**)** **{**

int left **=** 0**,** right **=** arr**.**length **-** 1**;**

int mid **=** partition**(**arr**,** left**,** right**);**

**while** **(**mid **!=** id**)** **{**

**if** **(**mid **>** id**)** **{**

right **=** mid **-** 1**;**

mid **=** partition**(**arr**,** left**,** right**);**

**}** **else** **{**

left **=** mid **+** 1**;**

mid **=** partition**(**arr**,** left**,** right**);**

**}**

**}**

**return** arr**[**mid**];**

**}**

static int partition**(**int**[]** arr**,** int left**,** int right**)** **{**

int mid **=** left **-**1**;**

**while** **(**left **<=** right**)** **{**

**if** **(**arr**[**left**]** **<=** arr**[**right**])** **{**

swap**(**arr**,** left**,** **++**mid**);**

**}**

left**++;**

**}**

**return** mid**;**

**}**

private static void swap**(**int**[]** arr**,** int left**,** int i**)** **{**

int t **=** arr**[**left**];**

arr**[**left**]** **=** arr**[**i**];**

arr**[**i**]** **=** t**;**

**}**

**}**

# 最小的K个数

输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。

1. 使用最大堆 nlog(K) 或者快排 NlogN 或者patition方法 N+KlogK

## 使用最大堆 nlog(K)

**import** java**.**util**.\*;**

public class Solution **{**

public static ArrayList**<**Integer**>** GetLeastNumbers\_Solution**(**int **[]** input**,** int k**)** **{**

ArrayList**<**Integer**>** list **=** **new** ArrayList**<>();**

**if** **(**input **==** **null** **||** k **>** input**.**length **||** input**.**length **==** 0 **||** k **==** 0**)** **return** list**;**

**return** get**(**input**,** k**);**

**}**

private static ArrayList**<**Integer**>** get**(**int**[]** input**,** int k**)** **{**

int**[]** heap **=** **new** int**[**k**];**

**for** **(**int i**=**0**;** i**<**k**;** i**++)** **{**

heap**[**i**]** **=** input**[**i**];**

**}**

//构建最大堆

**for** **(**int i**=**k**/**2**;** i**>=**0**;** i**--)** **{**

heapAdjust**(**heap**,** i**,** k**);**

**}**

//求最小的k个数

**for** **(**int i**=**k**;** i**<**input**.**length**;** i**++)** **{**

**if** **(**input**[**i**]** **<** heap**[**0**])** **{**

heap**[**0**]** **=** input**[**i**];**

heapAdjust**(**heap**,** 0**,** k**);**

**}**

**}**

//输出最大堆

ArrayList**<**Integer**>** out **=** **new** ArrayList**<>();**

**for** **(**int i**=**k**-**1**;** i**>=**0**;** i**--)** **{**

out**.**add**(**0**,** heap**[**0**]);**

swap**(**heap**,** 0**,** i**);**

heapAdjust**(**heap**,** 0**,** i**);**

**}**

**return** out**;**

**}**

//最大堆

static void heapAdjust**(**int**[]** arr**,** int id**,** int len**)** **{**

int i**=**0**;**

int cl**;**

**while** **((**cl **=** id**\***2 **+** 1**)** **<** len**)** **{**

// int cl = id\*2 + 1;

**if** **(**cl**+**1 **<** len **&&** arr**[**cl**+**1**]** **>** arr**[**cl**])** **{**

cl**++;**

**}**

**if** **(**arr**[**cl**]** **>** arr**[**id**])** **{**

swap**(**arr**,** cl**,** id**);**

id **=** cl**;**

**}** **else** **{**

**break;**

**}**

**}**

**}**

private static void swap**(**int**[]** arr**,** int left**,** int i**)** **{**

int t **=** arr**[**left**];**

arr**[**left**]** **=** arr**[**i**];**

arr**[**i**]** **=** t**;**

**}**

**}**

## patition方法 N+KlogK

**import** java**.**util**.\*;**

public class Solution **{**

public static ArrayList**<**Integer**>** GetLeastNumbers\_Solution**(**int **[]** input**,** int k**)** **{**

ArrayList**<**Integer**>** list **=** **new** ArrayList**<>();**

**if** **(**input **==** **null** **||** k **>** input**.**length **||** input**.**length **==** 0 **||** k **==** 0**)** **return** list**;**

// return get(input, k);

**return** partitionGet**(**input**,** k**);**

**}**

private static ArrayList**<**Integer**>** partitionGet**(**int**[]** input**,** int k**)** **{**

int**[]** heap **=** **new** int**[**k**];**

find**(**input**,** k**-**1**);** //找到input排序后的第k-1下标的元素,且该元素左边的值都比input[k-1]小

**for** **(**int i**=**0**;** i**<**k**;** i**++)** **{**

heap**[**i**]** **=** input**[**i**];**

**}**

//排序

Arrays**.**sort**(**heap**);**

//输出最大堆

ArrayList**<**Integer**>** out **=** **new** ArrayList**<>();**

**for** **(**int i**=**0**;** i**<**k**;** i**++)** **{**

out**.**add**(**heap**[**i**]);**

**}**

**return** out**;**

**}**

//找到数组中第id+1大的数(排序后下标为id)，O(n)

static int find**(**int**[]** arr**,** int id**)** **{**

int left **=** 0**,** right **=** arr**.**length **-** 1**;**

int mid **=** partition**(**arr**,** left**,** right**);**

**while** **(**mid **!=** id**)** **{**

**if** **(**mid **>** id**)** **{**

right **=** mid **-** 1**;**

mid **=** partition**(**arr**,** left**,** right**);**

**}** **else** **{**

left **=** mid **+** 1**;**

mid **=** partition**(**arr**,** left**,** right**);**

**}**

**}**

**return** arr**[**mid**];**

**}**

static int partition**(**int**[]** arr**,** int left**,** int right**)** **{**

int mid **=** left **-**1**;**

**while** **(**left **<=** right**)** **{**

**if** **(**arr**[**left**]** **<=** arr**[**right**])** **{**

swap**(**arr**,** left**,** **++**mid**);**

**}**

left**++;**

**}**

**return** mid**;**

**}**

private static void swap**(**int**[]** arr**,** int left**,** int i**)** **{**

int t **=** arr**[**left**];**

arr**[**left**]** **=** arr**[**i**];**

arr**[**i**]** **=** t**;**

**}**

**}**

# 连续子数组的最大和

HZ偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为8(从第0个开始,到第3个为止)。你会不会被他忽悠住？

动态规划方法

public class Solution **{**

static public int FindGreatestSumOfSubArray**(**int**[]** array**)** **{**

**if** **(**array **==** **null** **||** array**.**length **==** 0**)** **return** 0**;**

**return** find**(**array**);**

**}**

//计算连续子向量的最大和 {6,-3,-2,7,-15,1,2,2}

//如果向量和为负数，则下一个元素不需要接着前面的向量了，重新构造子向量

static private int find**(**int**[]** array**)** **{**

int **[]** dp **=** **new** int**[**array**.**length**];** //dp[i]表示 截止到array[i]时，出现的最大序列和

int max **=** array**[**0**];**

dp**[**0**]** **=** array**[**0**];**

**for** **(**int i**=**1**;** i**<**array**.**length**;** i**++)** **{**

//如果当前子向量和为负数，则下一个元素不需要添加到前面的向量了，重新构造子向量

//此处的dp[i-1]可用一个变量替代，因此也不需要dp数组了

**if** **(**dp**[**i**-**1**]** **<** 0**)** **{**

dp**[**i**]** **=** array**[**i**];**

**}** **else** **{**//为正 继续添加

dp**[**i**]** **=** dp**[**i**-**1**]** **+** array**[**i**];**

**}**

max **=** Math**.**max**(**max**,** dp**[**i**]);**

**}**

**return** max**;**

**}**

**}**

# 整数中1出现的次数（从1到n整数中1出现的次数）

求出1~13的整数中1出现的次数,并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、12、13因此共出现6次,但是对于后面问题他就没辙了。ACMer希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中1出现的次数。

这是一个数学问题：

public class Solution **{**

//主要思路：设定整数点（如1、10、100等等）作为位置点i（对应n的各位、十位、百位等等），

//分别对每个数位上有多少包含1的点进行分析

//根据设定的整数位置，对n进行分割，分为两部分，高位a=n/i，低位b=n%i

//例如当当i表示百位，有三种情况：

//1.当百位对应的数k>=2时,如n=31456,i=100，则a=314,b=56，

//此时百位为1的次数有32（最高两位0~31），每一次都包含100个连续的点(个位和十位范围0~99)

//即共有(a/10+1)\*100个点的百位为1

//2.当百位对应的数为1时，如n=31156,i=100，则a=311,b=56，

//当高位为0~30时，百位为1，低位可以为0~99，当高位为31时，百位为1，低位可以为0~56，

//故此时百位为1的点的个数为(a/10)\*100+(b+1)

//3.当百位对应的数为0,如n=31056,则a=310,b=56，

//只有当高位为0~30时，百位才为1，此时低位可以为0~99，故总的点数：(a/10)\*100

public int NumberOf1Between1AndN\_Solution**(**int n**)** **{**

int count **=** 0**;**

**for** **(**int i**=**1**;** i**<=**n**;** i**\*=**10**)** **{**

int a **=** n**/**i**,** b **=** n**%**i**;**

**if** **(**a **%** 10 **==** 0**)** **{**

count **+=** **(**a**/**10**)\***i**;**

**}** **else** **if** **(**a **%** 10 **==** 1**)** **{**

count **+=** **(**a**/**10**)\***i **+** b **+** 1**;**

**}** **else** **{**

count **+=** **(**a**/**10 **+** 1**)\***i**;**

**}**

**}**

**return** count**;**

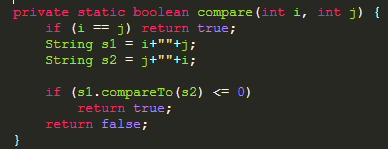
**}**

**}**

# 把数组排成最小的数

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3，32，321}，则打印出这三个数字能排成的最小数字为321323。

变种快排，比较2个数字拼接后的字符串的大小



**import** java**.**util**.**ArrayList**;**

public class Solution **{**

public static String PrintMinNumber**(**int **[]** numbers**)** **{**

**if** **(**numbers **==** **null)** **return** **null;**

qSort**(**numbers**,** 0**,** numbers**.**length**-**1**);**

StringBuilder sb **=** **new** StringBuilder**();**

**for** **(**int i**=**0**;** i**<**numbers**.**length**;** i**++)** **{**

sb**.**append**(**numbers**[**i**]);**

**}**

**return** sb**.**toString**();**

**}**

static int partion**(**int**[]** arr**,** int left**,** int right**)** **{**

int mid **=** left **-** 1**;**

**while** **(**left **<=** right**)** **{**

**if** **(**compare**(**arr**[**left**],** arr**[**right**]))** **{**

swap**(**arr**,** left**,** **++**mid**);**

**}**

left**++;**

**}**

**return** mid**;**

**}**

private static boolean compare**(**int i**,** int j**)** **{**

**if** **(**i **==** j**)** **return** **true;**

String s1 **=** i**+**""**+**j**;**

String s2 **=** j**+**""**+**i**;**

**if** **(**s1**.**compareTo**(**s2**)** **<=** 0**)**

**return** **true;**

**return** **false;**

**}**

static void qSort**(**int**[]** arr**,** int left**,** int right**)** **{**

**if** **(**left **>=** right**)** **return;**

int mid **=** partion**(**arr**,** left**,** right**);**

qSort**(**arr**,** left**,** mid**-**1**);**

qSort**(**arr**,** mid**+**1**,** right**);**

**}**

private static void swap**(**int**[]** arr**,** int a**,** int b**)** **{**

int t **=** arr**[**a**];**

arr**[**a**]** **=** arr**[**b**];**

arr**[**b**]** **=** t**;**

**}**

**}**

使用系统排序函数：

public class Solution **{**

public String PrintMinNumber**(**int **[]** numbers**)** **{**

**if** **(**numbers **==** **null)** **return** **null;**

**return** printMinNumber**(**numbers**);**

**}**

static public String printMinNumber**(**int**[]** numbers**)** **{**

List**<**Integer**>** list **=** **new** ArrayList**<>();**

**for** **(**int i **:** numbers**)** **{**

list**.**add**(**i**);**

**}**

Collections**.**sort**(**list**,** **new** Comparator**<**Integer**>()** **{**

@Override

public int compare**(**Integer o1**,** Integer o2**)** **{**

// TODO Auto-generated method stub

String s1 **=** o1 **+** "" **+** o2**;**

String s2 **=** o2 **+** "" **+** o1**;**

**return** s1**.**compareTo**(**s2**);**

**}**

**});**

StringBuilder sbBuilder **=** **new** StringBuilder**();**

**for** **(**Integer integer **:** list**)** **{**

sbBuilder**.**append**(**integer**);**

**}**

**return** sbBuilder**.**toString**();**

**}**

**}**

# 丑数

把只包含因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含因子7。 习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第N个丑数。

对一个已经生成的丑数序列，假设其最后一个也就是当前最大值的是M,则该序列必定存在一个丑数T2, 如T2的前一个丑数为F2, 满足T2 \* 2 > M，F2 \* 2 <M。 同理存在T3 \* 3 > M，F3 \* 3 <M。T5 \* 5 > M，F5 \* 5 <M。则下一个丑数是Min（T2 \*2, T3\*3, T5\*5），因此在丑数生成的过程中，只需要动态维护T2,T3,T5即可。

**import** java**.**util**.\*;**

**import** java**.**math**.\*;**

public class Solution **{**

public static int GetUglyNumber\_Solution**(**int index**)** **{**

**if** **(**index **<=** 0**)** **return** 0**;**

ArrayList**<**Integer**>** array **=** **new** ArrayList**<>();**

array**.**add**(**1**);**//1是第一个丑数

int t2Id**=**0**,** t3Id **=** 0**,** t5Id **=** 0**;**

int uglyMax**;**

**while** **(**array**.**size**()** **<** index**)** **{**

uglyMax **=** Math**.**min**(**array**.**get**(**t2Id**)\***2**,**

Math**.**min**(**array**.**get**(**t3Id**)\***3**,** array**.**get**(**t5Id**)\***5**));**

array**.**add**(**uglyMax**);**

**while** **(**array**.**get**(**t2Id**)\***2 **<=** uglyMax**)** t2Id**++;**

**while** **(**array**.**get**(**t3Id**)\***3 **<=** uglyMax**)** t3Id**++;**

**while** **(**array**.**get**(**t5Id**)\***5 **<=** uglyMax**)** t5Id**++;**

**}**

**return** array**.**get**(**index**-**1**);** //1是第一个丑数

**}**

**}**

# 第一个只出现一次的字符

在一个字符串(1<=字符串长度<=10000，全部由大写字母组成)中找到第一个只出现一次的字符,并返回它的位置。

1. 第一次遍历字符串，用256的数组记录每个字符出现的次数。
2. 第二次遍历字符串，找到第一个数组记录为1的字符。

public class Solution **{**

public int FirstNotRepeatingChar**(**String str**)** **{**

**if** **(**str **==** **null)** **return** **-**1**;**

int**[]** book **=** **new** int**[**256**];**

**for** **(**int i**=**0**;** i**<**str**.**length**();** i**++)** **{**

book**[**str**.**charAt**(**i**)]++;**

**}**

**for** **(**int i**=**0**;** i**<**str**.**length**();** i**++)** **{**

**if** **(**book**[**str**.**charAt**(**i**)]** **==** 1**)**

**return** i**;**

**}**

**return** **-**1**;**

**}**

**}**

# 数组中的逆序对

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P。并将P对1000000007取模的结果输出。 即输出P%1000000007

采用归并排序的方法，只是在合并数组时，从右往左合并，并且当左边的数大于右边数时候，逆序对数目 += 当前右边数组的剩余元素个数。

public class Solution **{**

public static int InversePairs**(**int **[]** arr**)** **{**

count **=** 0**;**

mergeSort**(**arr**,** 0**,** arr**.**length**-**1**);**

// System.out.println(Arrays.toString(arr));

// System.out.println(count % 1000000007);

**return** count**;**

**}**

static void mergeSort**(**int**[]** arr**,** int left**,** int right**)** **{**

**if** **(**left **>=** right**)** **return;**

int mid **=** **(**left **+** right**)** **/** 2**;**

mergeSort**(**arr**,** left**,** mid**);**

mergeSort**(**arr**,** mid**+**1**,** right**);**

merge**(**arr**,** left**,** mid**,** right**);**

**}**

static void mergeSort2**(**int**[]** arr**)** **{**

int len **=** arr**.**length**;**

int sl **=** 1**,** dl **=** 1**;**

**while** **(**sl **<** len**)** **{**

dl **=** sl **\*** 2**;**

int i **=** 0**;**

**while** **(**i **+** dl **<** len**)** **{**

merge**(**arr**,** i**,** i**+**sl**-**1**,** i**+**dl**-**1**);**

i **+=** dl**;**

**}**

**if** **(**i **+** sl **<** len**)** **{**

merge**(**arr**,** i**,** i**+**sl**-**1**,** len**-**1**);**

**}**

sl **=** dl**;**

**}**

**}**

static int count **=** 0**;**

private static void merge**(**int**[]** arr**,** int left**,** int mid**,** int right**)** **{**

int**[]** temp **=** **new** int**[**right**-**left**+**1**];**

int id **=** temp**.**length **-** 1**;**

int l **=** mid**,** r **=** right**;**

**while** **(**l **>=** left **&&** r **>** mid**)** **{**

**if** **(**arr**[**l**]** **>** arr**[**r**])** **{**

count **+=** r **-** mid**;**

**if** **(**count **>** 1000000007**)** **{**

count **=** count **%** 1000000007**;**

**}**

temp**[**id**--]** **=** arr**[**l**--];**

**}** **else** **{**

temp**[**id**--]** **=** arr**[**r**--];**

**}**

**}**

**while** **(**l **>=** left**)** **{**

temp**[**id**--]** **=** arr**[**l**--];**

**}**

**while** **(**r **>** mid**)** **{**

temp**[**id**--]** **=** arr**[**r**--];**

**}**

**for** **(**int i**=**0**;** i**<**temp**.**length**;** i**++)** **{**

arr**[**i**+**left**]** **=** temp**[**i**];**

**}**

**}**

private static void swap**(**int**[]** arr**,** int a**,** int b**)** **{**

int t **=** arr**[**a**];**

arr**[**a**]** **=** arr**[**b**];**

arr**[**b**]** **=** t**;**

**}**

**}**

# 两个链表的第一个公共结点

输入两个链表，找出它们的第一个公共结点。

2个链表有公共节点，那么它们公共节点后的链表部分是一样的。

1. 借助2个栈，将2个链表分别入栈，这样栈顶就是每个链表的尾尾节点，依次出栈，并比较出栈的元素，相同则为公共节点。
2. 先分别遍历2个链表，求得它们的长度m,n，如果m>n，就先让m链表走m-n步，再一起遍历，遇到相同的那个节点就是公共结点了。
3. /\*
4. public class ListNode {
5. int val;
6. ListNode next = null;
7. ListNode(int val) {
8. this.val = val;
9. }
10. }\*/
11. public class Solution **{**
12. public ListNode FindFirstCommonNode**(**ListNode pHead1**,** ListNode pHead2**)** **{**
14. **if** **(**pHead1 **==** **null** **||** pHead2 **==** **null)** **return** **null;**
15. int len1 **=** 0**;**
16. int len2 **=** 0**;**
17. ListNode node **=** pHead1**;**
18. **while** **(**node **!=** **null)** **{**
19. len1**++;**
20. node **=** node**.**next**;**
21. **}**
22. node **=** pHead2**;**
23. **while** **(**node **!=** **null)** **{**
24. len2**++;**
25. node **=** node**.**next**;**
26. **}**
28. **if** **(**len1 **>** len2**)** **{**
29. int diff **=** len1 **-** len2**;**
30. **for** **(**int i**=**0**;** i**<**diff**;** i**++)** pHead1 **=** pHead1**.**next**;**
31. **}** **else** **{**
32. int diff **=** len2 **-** len2**;**
33. **for** **(**int i**=**0**;** i**<**diff**;** i**++)** pHead2 **=** pHead2**.**next**;**
34. **}**
36. **while** **(**pHead1 **!=** pHead2**)** **{**
37. pHead1 **=** pHead1**.**next**;**
38. pHead2 **=** pHead2**.**next**;**
39. **}**
41. **if** **(**pHead1 **==** pHead2**)**
42. **return** pHead1**;**
43. **else**
44. **return** **null;**
46. **}**
47. **}**

# 数字在排序数组中出现的次数

统计一个数字在排序数组中出现的次数。

采用二分查找，分别查找最左和最右的下标。

public static int GetNumberOfK**(**int **[]** arr **,** int k**)** **{**

**if** **(**arr **==** **null** **||** arr**.**length **==** 0**)** **return** 0**;**

int first **=** binaryFind**(**arr**,** k**,** 0**);**

int last **=** binaryFind**(**arr**,** k**,** 1**);**

**if** **(**first **!=** **-**1 **&&** last **!=** **-**1**)** **{**

**return** last**-**first**+**1**;**

**}**

**return** 0**;**

**}**

//二分查找 type 0 查找最左边的下标 1查找最右边的下标

static int binaryFind**(**int**[]** arr**,** int value**,** int type**)** **{**

int left **=** 0**,** right **=** arr**.**length **-** 1**;**

int mid **=** 0**;**

**while** **(**left **<=** right**)** **{**

mid **=** **(**left **+** right**)** **/** 2**;**

**if** **(**arr**[**mid**]** **>** value**)** **{**

right **=** mid **-** 1**;**

**}** **else** **if** **(**arr**[**mid**]** **<** value**){**

left **=** mid **+** 1**;**

**}** **else** **{**

**if** **(**type **==** 0**)** **{**//左边k

**if** **(**mid **==** 0 **||** **(**mid **-** 1 **>=** 0 **&&** arr**[**mid**-**1**]** **!=** arr**[**mid**]))** **{**

**return** mid**;**

**}** **else** **{**

right **=** mid **-** 1**;**

**}**

**}** **else** **{**//右边k

**if** **(**mid **==** arr**.**length**-**1 **||** **(**mid **+** 1 **<** arr**.**length **&&** arr**[**mid**+**1**]** **!=** arr**[**mid**]))** **{**

**return** mid**;**

**}** **else** **{**

left **=** mid **+** 1**;**

**}**

**}**

**}**

**}**

**return** **-**1**;**

**}**

# 二叉树的深度

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

简单的递归：

public class Solution **{**

public static int TreeDepth**(**TreeNode root**)**

**{**

**if** **(**root **==** **null)** **return** 0**;**

int left **=** TreeDepth**(**root**.**left**);**

int right **=** TreeDepth**(**root**.**right**);**

**return** Math**.**max**(**left**,**right**)** **+** 1**;**

**}**

**}**

# 按层遍历树

**import** java**.**util**.\*;**

public class Solution **{**

public static int TreeDepth**(**TreeNode root**)** **{**

**if** **(**root **==** **null)** **return** 0**;**

Queue**<**TreeNode**>** queue **=** **new** ArrayDeque**<>();**

queue**.**add**(**root**);**

int depth **=** 0**;**

int outCount **=** 0**;**//记录当前层输出的结点数

int curCount **=** 1**;**//换行判断，记录当前层的结点个数

TreeNode node**;**

**while** **(!**queue**.**isEmpty**())** **{**

node **=** queue**.**poll**();**

outCount**++;**

**if** **(**node**.**left **!=** **null)** **{**

queue**.**add**(**node**.**left**);**

**}**

**if** **(**node**.**right **!=** **null)** **{**

queue**.**add**(**node**.**right**);**

**}**

**if** **(**outCount **==** curCount**)** **{**

curCount **=** queue**.**size**();** //更新层的结点数，并切换到下一层

outCount **=** 0**;**

depth**++;**

**}**

**}**

**return** depth**;**

**}**

**}**

# 平衡二叉树

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

public class Solution **{**

public boolean IsBalanced\_Solution**(**TreeNode root**)** **{**

**if** **(**root **==** **null)** **return** **true;**

int leftDepth **=** treeDepth**(**root**.**left**);**

int rightDepth **=** treeDepth**(**root**.**right**);**

//左右子树的高度差大于1 就不是平衡树了

**if** **(**Math**.**abs**(**leftDepth **-** rightDepth**)** **>** 1**)** **return** **false;**

**return** IsBalanced\_Solution**(**root**.**left**)** **&&** IsBalanced\_Solution**(**root**.**right**);**

**}**

public static int treeDepth**(**TreeNode root**)**

**{**

**if** **(**root **==** **null)** **return** 0**;**

int left **=** treeDepth**(**root**.**left**);**

int right **=** treeDepth**(**root**.**right**);**

**return** Math**.**max**(**left**,**right**)** **+** 1**;**

**}**

**}**

# 数组中只出现一次的数字

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

异或的使用：即两个相同的数异或结果为0。异或满足分配律，任意数和0异或不变。

//num1,num2分别为长度为1的数组。传出参数

//将num1[0],num2[0]设置为返回结果

**import** java**.**util**.\*;**

public class Solution **{**

public void FindNumsAppearOnce**(**int **[]** arr**,**int num1**[]** **,** int num2**[])** **{**

**if** **(**arr **==** **null** **||** arr**.**length **<** 2**)** **{**

num1**[**0**]** **=** 0**;**

num2**[**0**]** **=** 0**;**

**return;**

**}**

int xor **=** arr**[**0**]^**arr**[**1**];**

**for** **(**int i**=**2**;** i**<**arr**.**length**;** i**++)** **{**

xor **=** xor **^** arr**[**i**];**

**}**

int flag **=** 1**;**

**for** **(**int i**=**0**;** i**<**32**;** i**++)** **{**

**if** **((**flag **&** xor**)** **!=** 0**)** **{**

**break;**

**}**

flag **=** flag **<<** 1**;**

**}**

List**<**Integer**>** a1 **=** **new** ArrayList**<**Integer**>();**

List**<**Integer**>** a2 **=** **new** ArrayList**<**Integer**>();**

**for** **(**int i**=**0**;** i**<**arr**.**length**;** i**++)** **{**

**if** **((**arr**[**i**]** **&** flag**)** **!=** 0**)** **{**

a1**.**add**(**arr**[**i**]);**

**}** **else** **{**

a2**.**add**(**arr**[**i**]);**

**}**

**}**

int temp **=** a1**.**get**(**0**);**

**for** **(**int i**=**1**;** i**<**a1**.**size**();** i**++)** **{**

temp **=** temp **^** a1**.**get**(**i**);**

**}**

num1**[**0**]** **=** temp**;**

temp **=** a2**.**get**(**0**);**

**for** **(**int i**=**1**;** i**<**a2**.**size**();** i**++)** **{**

temp **=** temp **^** a2**.**get**(**i**);**

**}**

num2**[**0**]** **=** temp**;**

**}**

**}**

# 和为S的两个数字

输入一个递增排序的数组和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。

**import** java**.**util**.**ArrayList**;**

public class Solution **{**

public ArrayList**<**Integer**>** FindNumbersWithSum**(**int **[]** arr**,**int sum**)** **{**

ArrayList**<**Integer**>** result **=** **new** ArrayList**<>();**

int left **=** 0**,** right **=** arr**.**length **-** 1**;**

//从数组两端取两个数求和，交替向中间缩进遍历

//满足条件的第一组的乘积也是最小

**while** **(**left **<** right**)** **{**

**if** **(**arr**[**left**]** **+** arr**[**right**]** **>** sum**)** **{**

right**--;**

**}** **else** **if** **(**arr**[**left**]** **+** arr**[**right**]** **<** sum**)** **{**

left**++;**

**}** **else** **{**

result**.**add**(**arr**[**left**]);**

result**.**add**(**arr**[**right**]);**

**break;**

**}**

**}**

**return** result**;**

**}**

**}**

# 和为S的连续正数序列

小明很喜欢数学,有一天他在做数学作业时,要求计算出9~16的和,他马上就写出了正确答案是100。但是他并不满足于此,他在想究竟有多少种连续的正数序列的和为100(至少包括两个数)。没多久,他就得到另一组连续正数和为100的序列:18,19,20,21,22。现在把问题交给你,你能不能也很快的找出所有和为S的连续正数序列? Good Luck!

输出所有和为S的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序

**import** java**.**util**.**ArrayList**;**

public class Solution **{**

static public ArrayList**<**ArrayList**<**Integer**>** **>** FindContinuousSequence**(**int sum**)** **{**

ArrayList**<**ArrayList**<**Integer**>>** lists **=** **new** ArrayList**<>();**

ArrayList**<**Integer**>** seq **=** **new** ArrayList**<>();**

int smallId **=** 0**,** bigNum**=**1**,** tempSum**=**0**;**

**for** **(**int i**=**1**;** **;** i**++)** **{**

seq**.**add**(**i**);**

tempSum **+=** i**;**

bigNum **=** i**;**//更新序列中的最大值

**if** **(**tempSum **>=** sum**)** **break;**

**}**

//当smallId 为序列的倒数第二个时，说明已经到了sum的一半位置

**while** **(**smallId **<** seq**.**size**()-**1**)** **{**

**if** **(**tempSum **>** sum**)** **{**

//从序列中 移除左边的数值

tempSum **-=** seq**.**get**(**smallId**);**

smallId**++;**

**}** **else** **if** **(**tempSum **<** sum**)** **{**

//向右递进

bigNum**++;**

seq**.**add**(**bigNum**);**

tempSum **+=** bigNum**;**

**}** **else** **{**//equal

//添加符合要求的序列

lists**.**add**(new** ArrayList**<>(**seq**.**subList**(**smallId**,** seq**.**size**())));**

//继续递进增加

bigNum**++;**

seq**.**add**(**bigNum**);**

tempSum **+=** bigNum**;**

**}**

// System.out.println(seq);

**}**

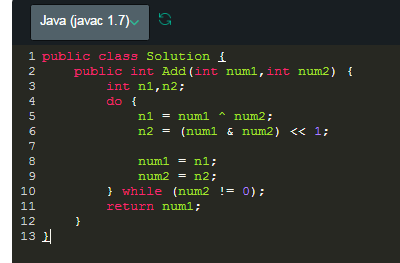
**return** lists**;**

**}**

**}**

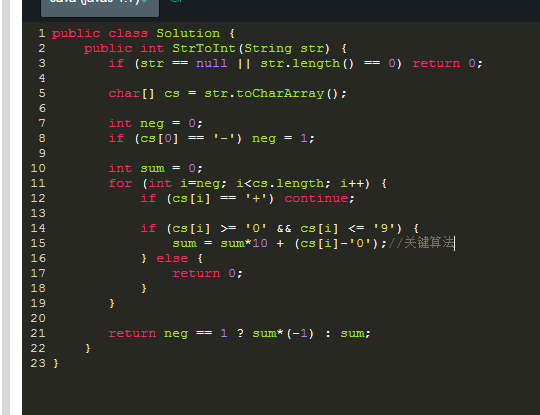
# 不用加减乘除做加法

写一个函数，求两个整数之和，要求在函数体内不得使用+、-、\*、/四则运算符号。



# 把字符串转换成整数

将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。



# 正则表达式匹配

请实现一个函数用来匹配包括'.'和'\*'的正则表达式。模式中的字符'.'表示任意一个字符，而'\*'表示它前面的字符可以出现任意次（包含0次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab\*ac\*a"匹配，但是与"aa.a"和"ab\*a"均不匹配

public class Solution **{**

public boolean match**(**char**[]** str**,** char**[]** pattern**)**

**{**

**return** match2**(**str**,** pattern**);**

**}**

static public boolean match2**(**char**[]** str**,** char**[]** pattern**)** **{**

**if(**str**==null&&**pattern**==null)**

**return** **true;**

**if(**str**==null||**pattern**==null)**

**return** **false;**

**return** helper**(**str**,**0**,**str**.**length**,**pattern**,**0**,**pattern**.**length**);**

**}**

static private boolean helper**(**char**[]** str**,** int i**,** int length**,**

char**[]** pattern**,**int j**,** int length2**)** **{**

**if(**i**==**length **&&** j**==**length2**)**//主串完事，匹配串也完事

**return** **true;**

**if(**i**!=**length **&&** j**==**length2**)**//主串没有完事，匹配串完事

**return** **false;**

**if(**j**+**1**<**length2 **&&** pattern**[**j**+**1**]==**'\*'**){**

**if(**i**<**length **&&** **(**str**[**i**]==**pattern**[**j**]** **||** pattern**[**j**]==**'.'**)){**//主串和模式串当前字符匹配

**return** helper**(**str**,** i**+**1**,** length**,** pattern**,**j**,** length2**)||** //主串向后移动，匹配串不变aaa和a\*

helper**(**str**,** i**+**1**,** length**,** pattern**,**j**+**2**,** length2**)||** //主串向后移动，匹配串跳过a\*

helper**(**str**,** i**,** length**,** pattern**,**j**+**2**,** length2**);** //主串不变，匹配模式串后2个字符，跳过a\*

**}else**

**return** helper**(**str**,** i**,** length**,** pattern**,**j**+**2**,** length2**);**//主串和模式串不匹配，略过c和\*

**}**

**if(**i**<**length**&&(**str**[**i**]==**pattern**[**j**]||(**pattern**[**j**]==**'.'**))){**//主串当前字符不为空，要么和模式串匹配，要么模式串为.

**return** helper**(**str**,** i**+**1**,** length**,** pattern**,**j**+**1**,** length2**);**

**}**

**return** **false;**

**}**

**}**

# 表示数值的字符串

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。

public class Solution **{**

public static boolean isNumeric**(**char**[]** str**)** **{**

// return new String(str).matches("[\\+-]?[0-9]\*");

**return** **new** String**(**str**).**matches**(**"[\\+\\-]?[0-9]\*(\\.[0-9]\*)?([Ee][\\+\\-]?[0-9]+)?"**);**

**}**

**}**

# 求1+2+3+...+n

求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

public class Solution **{**

public int Sum\_Solution**(**int n**)** **{**

// if (n == 0) return 0;

// return n+Sum\_Solution(n-1);

// int sum = n;

// if (n == 0) return 0;

// return sum += Sum\_Solution(n-1);

//看这个即可

// int sum = n;

// boolean ans = (n>0) && (sum += Sum\_Solution(n-1)) > 0;

// return sum;

boolean a **=** **(**n **>** 0**)** **&&** **(**n **+=** Sum\_Solution**(**n**-**1**))** **>** 0**;**

**return** n**;**

**}**

**}**

# 数组中重复的数字

在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字2或者3。

简单的使用hashMap.

不需要额外空间的算法，利用当前数组，根据 所有数字都在0到n-1的范围内 这个特征。

访问数组arr中的arr[k]时，arr[k] = num，令 arr[num] += n; 如果数组中存在另一个 arr[kk]=num， 此时访问arr[num] > n，表示num是重复数值。 如果用数字当下标访问的时候，先判断该数字是否大于等于n，是则减去n

public boolean duplicate**(**int numbers**[],**int length**,**int **[]** duplication**)** **{**

**if** **(**numbers **==** **null** **||** numbers**.**length **==** 0**)** **return** **false;**

**for** **(**int i**=**0**;** i**<**length**;** i**++)** **{**

int num **=** numbers**[**i**];** // 1

**if** **(**num **>=** length**)** num **-=** length**;** // 4

**if** **(**numbers**[**num**]** **>** length**)** **{** // 3

duplication**[**0**]** **=** num**;**

**return** **true;**

**}**

numbers**[**num**]** **+=** length**;** // 2

**}**

**return** **false;**

**}**

**}**

# 字符流中第一个不重复的字符

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。

建立hash表，初始化为-1， hash的key是字符，value是字符的下标。当字符第一次出现时将相应key的value设置为下标位置，大于一次出现设置为-2。

**import** java**.**util**.\*;**

public class Solution **{**

int**[]** book **=** **new** int**[**256**];**

int index **=** **-**1**;**

int minId **=** Integer**.**MAX\_VALUE**;**

char resultChar **=** 0**;**

public Solution**()** **{**

**for** **(**int i**=**0**;** i**<**book**.**length**;** i**++)** **{**

book**[**i**]** **=** **-**1**;**

**}**

**}**

//Insert one char from stringstream

public void Insert**(**char ch**)**

**{**

index**++;**

**if** **(**book**[**ch**]** **==** **-**1**)** **{**

book**[**ch**]** **=** index**;**

**}** **else** **if** **(**book**[**ch**]** **!=** **-**1**)** **{**

book**[**ch**]** **=** **-**2**;**

**}**

**}**

//return the first appearence once char in current stringstream

public char FirstAppearingOnce**()**

**{**

minId **=** Integer**.**MAX\_VALUE**;**

**for** **(**int i**=**0**;** i**<**book**.**length**;** i**++)** **{**

**if** **(**book**[**i**]** **!=** **-**1 **&&** book**[**i**]** **!=** **-**2**)** **{**

**if** **(**minId **>** book**[**i**])** **{**

minId **=** book**[**i**];**

resultChar **=** **(**char**)** i**;**

**}**

**}**

**}**

**if** **(**minId **==** Integer**.**MAX\_VALUE**)** **return** '#'**;**

**return** resultChar**;**

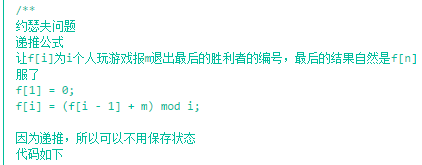
**}**

**}**

# 孩子们的游戏(圆圈中最后剩下的数)

每年六一儿童节,牛客都会准备一些小礼物去看望孤儿院的小朋友,今年亦是如此。HF作为牛客的资深元老,自然也准备了一些小游戏。其中,有个游戏是这样的:首先,让小朋友们围成一个大圈。然后,他随机指定一个数m,让编号为0的小朋友开始报数。每次喊到m-1的那个小朋友要出列唱首歌,然后可以在礼品箱中任意的挑选礼物,并且不再回到圈中,从他的下一个小朋友开始,继续0...m-1报数....这样下去....直到剩下最后一个小朋友,可以不用表演,并且拿到牛客名贵的“名侦探柯南”典藏版(名额有限哦!!^\_^)。请你试着想下,哪个小朋友会得到这份礼品呢？(注：小朋友的编号是从0到n-1)

约瑟夫问题



链表模拟：

注意，报数值从0开始，每次递增 m-1次，

**import** java**.**util**.\*;**

public class Solution **{**

static public int LastRemaining\_Solution**(**int n**,** int m**)** **{**

**if** **(**n **==** 0 **||** m **==** 0**)return** **-**1**;**

LinkedList**<**Integer**>** list **=** **new** LinkedList**<>();**

//初始化链表

**for** **(**int i**=**0**;** i**<**n**;** i**++)** **{**

list**.**add**(**i**);**

**}**

//直到剩下最后一个

int count **=** 0**;**

**while** **(**list**.**size**()** **>** 1**)** **{**

//此处计数 m-1次

**for** **(**int i**=**1**;** i**<**m**;** i**++)** **{**

count**++;**

count **=** count **%** list**.**size**();**

**}**

list**.**remove**(**count**);**

**}**

**return** list**.**get**(**0**);**

**}**

**}**

# 左旋转字符串

汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的字符序列S，请你把其循环左移K位后的序列输出。例如，字符序列S=”abcXYZdef”,要求输出循环左移3位后的结果，即“XYZdefabc”。是不是很简单？OK，搞定它！

/\*解法一：在原字符串上修改

“abcdef”循环左移9位（3位）

前3位逆序，后3位逆序，整体再逆序。

“cbafed”-> “defabc”

public class Solution **{**

public String LeftRotateString**(**String str**,**int n**)** **{**

**if** **(**str **==** **null** **||** str**.**length**()<**n**)** **return** ""**;**

//return str.substring(n,str.length())+str.substring(0,n);

**return** leftRotateString**(**str**,** n**);**

**}**

static String leftRotateString**(**String str**,** int n**)** **{**

char**[]** cs **=** str**.**toCharArray**();**

rev**(**cs**,** 0**,** n**-**1**);**

rev**(**cs**,** n**,** cs**.**length**-**1**);**

rev**(**cs**,** 0**,** cs**.**length**-**1**);**

**return** String**.**valueOf**(**cs**);**

**}**

static void rev**(**char**[]** cs**,** int l**,** int r**)** **{**

**while** **(**l **<** r**)** **{**

char c **=** cs**[**l**];**

cs**[**l**]** **=** cs**[**r**];**

cs**[**r**]** **=** c**;**

l**++;**

r**--;**

**}**

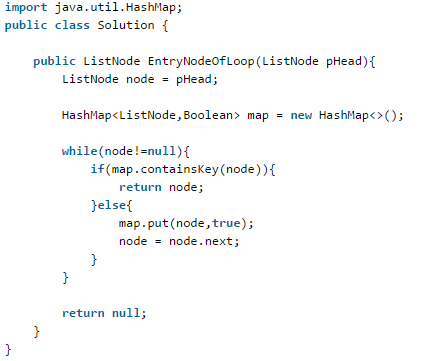
**}**

**}**

# 链表中环的入口结点

一个链表中包含环，请找出该链表的环的入口结点。

或者用HashMap



/\*

public class ListNode {

int val;

ListNode next = null;

ListNode(int val) {

this.val = val;

}

}

\*/

public class Solution **{**

//1.设置两个指针(fast, slow)，初始值都指向头，slow每次前进一步，fast每次前进二步，如果链表存在环，

//则fast必定先进入环，而slow后进入环，两个指针必定在环内相遇。

//2.从链表头、与相遇点分别设一个指针，每次各走一步，两个指针必定相遇，且相遇第一点为环入口点。

public ListNode EntryNodeOfLoop**(**ListNode pHead**)** **{**

**if** **(**pHead **==** **null)** **return** **null;**

ListNode slowN **=** pHead**,**fastN **=** pHead**;**

**while** **(**slowN **!=** **null** **&&** fastN**.**next **!=** **null)** **{**

slowN **=** slowN**.**next**;**

fastN **=** fastN**.**next**.**next**;**

**if** **(**slowN **==** fastN**)** **{**

**break;**

**}**

**}**

//无环链表

**if** **(**slowN **==** **null** **||** fastN**.**next **==** **null)** **return** **null;**

fastN **=** pHead**;**

**while** **(**fastN **!=** slowN**)** **{**

fastN **=** fastN**.**next**;**

slowN **=** slowN**.**next**;**

**}**

**return** fastN**;**

**}**

**}**

# 删除链表中重复的结点

在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5

public class Solution **{**

public ListNode deleteDuplication**(**ListNode pHead**)** **{**

ListNode head **=** getHead**(**pHead**);//头节点也可能会重复**

ListNode cur **=** head**;**

**while** **(**cur **!=** **null)** **{**

cur**.**next **=** getHead**(**cur**.**next**);**

cur **=** cur**.**next**;**

**}**

**return** head**;**

**}**

//得到以该结点起始的无重复结点的起始结点，如果发现重复，则删除（跳过）重复结点

static ListNode getHead**(**ListNode pHead**)** **{**

int count **=** 0**;**

**while** **(**pHead **!=** **null** **&&** pHead**.**next **!=** **null**

**&&** pHead**.**val **==** pHead**.**next**.**val**)** **{**

count**++;**

pHead **=** pHead**.**next**;**

**}**

**if** **(**count **==** 0**)** **{**

**return** pHead**;**

**}** **else** **{**

**return** getHead**(**pHead**.**next**);**

**}**

**}**

**}**

# 构建乘积数组

给定一个数组A[0,1,...,n-1],请构建一个数组B[0,1,...,n-1],其中B中的元素B[i]=A[0]\*A[1]\*...\*A[i-1]\*A[i+1]\*...\*A[n-1]。不能使用除法。

利用两个辅助数组，

第一个数组L依次保存A数组从0~length-1的乘积，

第二个数组h依次保存从length-1~0的乘积，

然后每一个要求的B[i]=L[i-1]\*H[i+1].

**import** java**.**util**.**ArrayList**;**

public class Solution **{**

public int**[]** multiply**(**int**[]** input**)** **{**

**if** **(**input **==** **null)** **return** **null;**

int len **=** input**.**length**;**

int**[]** out **=** **new** int**[**len**];**

int**[]** low **=** **new** int**[**len**];**

int**[]** high **=** **new** int**[**len**];**

**for** **(**int i**=**len**-**1**;** i**>=**0**;** i**--)** **{**

**if** **(**i **==** len **-** 1**)** **{**

high**[**i**]** **=** 1**;**

**}** **else** **{**

high**[**i**]** **=** high**[**i**+**1**]\***input**[**i**+**1**];**

**}**

**}**

**for** **(**int i**=**0**;** i**<**len**;** i**++)** **{**

**if** **(**i **==** 0**)** **{**

low**[**i**]** **=** 1**;**

**}** **else** **{**

low**[**i**]** **=** low**[**i**-**1**]\***input**[**i**-**1**];**

**}**

out**[**i**]** **=** low**[**i**]\***high**[**i**];**

**}**

// System.out.println(Arrays.toString(low));

// System.out.println(Arrays.toString(high));

// System.out.println(Arrays.toString(out));

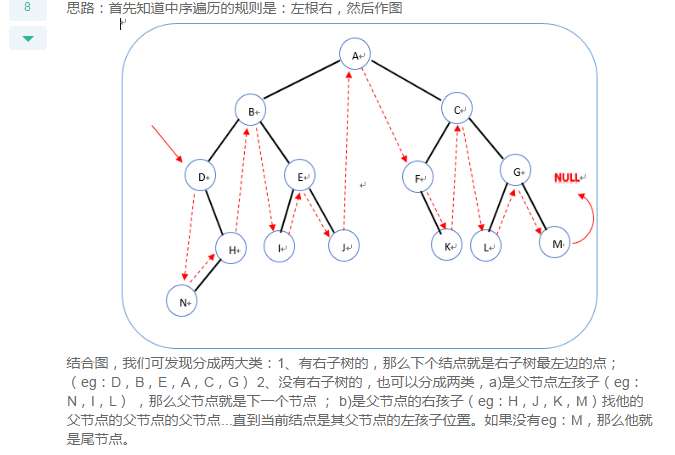
**return** out**;**

**}**

**}**

# 二叉树的下一个结点

给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。



/\*

public class TreeLinkNode {

int val;

TreeLinkNode left = null;

TreeLinkNode right = null;

TreeLinkNode next = null;

TreeLinkNode(int val) {

this.val = val;

}

}

\*/

public class Solution **{**

public TreeLinkNode GetNext**(**TreeLinkNode pNode**)** **{**

**if** **(**pNode **==** **null)** **return** **null;**

//如果有右子树，则下一结点为右子树的最左子结点

**if** **(**pNode**.**right **!=** **null)** **{**

pNode **=** pNode**.**right**;**

**while** **(**pNode**.**left **!=** **null)** pNode **=** pNode**.**left**;**

**return** pNode**;**

**}**

//如果没有右子树，则找到第一个是它父节点的左孩子的节点，返回此节点的父节点

**while** **(**pNode**.**next **!=** **null)** **{**

**if** **(**pNode **==** pNode**.**next**.**left**)** **return** pNode**.**next**;**

pNode **=** pNode**.**next**;**

**}**

//退到了根结点，仍没有找到，返回null

**return** **null;**

**}**

**}**

或者忘了这个规律的话，找到根结点，从根结点进行中序遍历。

**import** java**.**util**.\*;**

public class Solution **{**

public TreeLinkNode GetNext**(**TreeLinkNode pNode**){**

**if(**pNode**==null)**

**return** **null;**

//不是从这个pNode开始中序遍历，而是从树根开始

//所有先找到树根

TreeLinkNode root **=** pNode**;**

**while(**root**.**next**!=null)**

root **=** root**.**next**;**

List**<**TreeLinkNode**>** list **=** **new** ArrayList**<**TreeLinkNode**>();**

//中序遍历

get**(**root**,**list**);**

list**.**add**(null);**

//找到下一个节点

**for(**int i **=** 0**;**i**<**list**.**size**()-**1**;**i**++){**

**if(**list**.**get**(**i**)==**pNode**)**

**return** list**.**get**(**i**+**1**);**

**}**

**return** **null;**

**}**

public void get**(**TreeLinkNode root**,**List**<**TreeLinkNode**>** list**){**

**if(**root**==null)**

**return;**

get**(**root**.**left**,**list**);**

list**.**add**(**root**);**

get**(**root**.**right**,**list**);**

**}**

**}**

# 对称的二叉树

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

/\*

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

\*/

public class Solution **{**

//分析：可以用递归实现，一般一颗空树也是一个对称树，在比较两颗子树的候，

//判断左子树的左分支与右子树的右分支对称，右子树的左分支和左子树的右分支

//对称，那么这颗树就对称，通过上述可以实现递归

boolean isSymmetrical**(**TreeNode root**)**

**{**

**if** **(**root **==** **null)** **return** **true;**

**return** compareTree**(**root**.**left**,** root**.**right**);**

**}**

boolean compareTree**(**TreeNode root1**,** TreeNode root2**)** **{**

//同时为空返回true 不同时为空返回false

**if** **(**root1 **==** **null)** **return** root2 **==** **null;**

**if** **(**root2 **==** **null)** **return** **false;** //root1不为null

**if** **(**root1**.**val **!=** root2**.**val**)** **return** **false;**

**return** compareTree**(**root1**.**left**,** root2**.**right**)**

**&&** compareTree**(**root1**.**right**,** root2**.**left**);**

**}**

# 序列化二叉树

请实现两个函数，分别用来序列化和反序列化二叉树。这里没有规定序列化的方式。

/\*

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}

\*/

public class Solution **{**

//前序遍历，序列化

String Serialize**(**TreeNode root**)** **{**

StringBuilder sb **=** **new** StringBuilder**();**

**if** **(**root **==** **null)** **{**

**return** sb**.**append**(**"#,"**).**toString**();**

**}**

sb**.**append**(**root**.**val**+**","**);**

sb**.**append**(**Serialize**(**root**.**left**));**

sb**.**append**(**Serialize**(**root**.**right**));**

**return** sb**.**toString**();**

**}**

int index **=** **-**1**;**

TreeNode Deserialize**(**String str**)** **{**

String**[]** strs **=** str**.**split**(**","**);**

**return** deserializeDfs**(**strs**);**

**}**

//反序列化

TreeNode deserializeDfs**(**String**[]** strs**)** **{**

index**++;**

TreeNode node **=** **null;**

**if** **(!**strs**[**index**].**equals**(**"#"**))** **{**

node **=** **new** TreeNode**(**Integer**.**valueOf**(**strs**[**index**]));**

node**.**left **=** deserializeDfs**(**strs**);**

node**.**right **=** deserializeDfs**(**strs**);**

**}**

**return** node**;**

**}**

**}**

# 二叉搜索树的第k个结点

给定一颗二叉搜索树，请找出其中的第k大的结点。例如， 5 / \ 3 7 /\ /\ 2 4 6 8 中，按结点数值大小顺序第三个结点的值为4。

**import** java**.**util**.\*;**

public class Solution **{**

TreeNode KthNode**(**TreeNode root**,** int k**)**

**{**

**if** **(**k **==** 0 **||** root **==** **null)** **return** **null;**

midVisit**(**root**);**

**if** **(**k **>** list**.**size**())** **return** **null;**

**return** list**.**get**(**k**-**1**);**

**}**

LinkedList**<**TreeNode**>** list **=** **new** LinkedList**<>();**

void midVisit**(**TreeNode root**)** **{**

**if** **(**root **==** **null)** **{**

**return;**

**}**

midVisit**(**root**.**left**);**

list**.**add**(**root**);**

midVisit**(**root**.**right**);**

**}**

**}**

改进的中序遍历

public class Solution **{**

int cnt **=** 0**;**

TreeNode KthNode**(**TreeNode root**,**int k**)** **{**

**if** **(**root **==** **null)** **return** **null;**

TreeNode target **=** KthNode**(**root**.**left**,** k**);**

**if** **(**target **!=** **null)**

**return** target**;**

**if** **(++**cnt **<** k**)** **{**

**return** KthNode**(**root**.**right**,** k**);**

**}** **else** **if** **(**cnt **==** k**)** **{**

**return** root**;**

**}**

**return** **null;**

**}**

# 把二叉树打印成多行

从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

**import** java**.**util**.\*;**

public class Solution **{**

ArrayList**<**ArrayList**<**Integer**>** **>** Print**(**TreeNode root**)** **{**

ArrayList**<**Integer**>** out **=** **new** ArrayList**<>();**

ArrayList**<**ArrayList**<**Integer**>>** result **=** **new** ArrayList**<>();**

Queue**<**TreeNode**>** queue **=** **new** ArrayDeque**<>();**

**if** **(**root **==** **null)** **return** result**;**

int count **=** 0**,** nextCount **=** 1**;**

queue**.**add**(**root**);**

TreeNode cur**;**

**while** **(!**queue**.**isEmpty**())** **{**

cur **=** queue**.**poll**();**

out**.**add**(**cur**.**val**);**

count**++;**

**if** **(**cur**.**left **!=** **null)** **{**

queue**.**add**(**cur**.**left**);**

**}**

**if** **(**cur**.**right **!=** **null)** **{**

queue**.**add**(**cur**.**right**);**

**}**

**if** **(**count **==** nextCount**)** **{**

nextCount **=** queue**.**size**();**

count **=** 0**;**

result**.**add**(new** ArrayList**<>(**out**));**

out**.**clear**();**

**}**

**}**

**return** result**;**

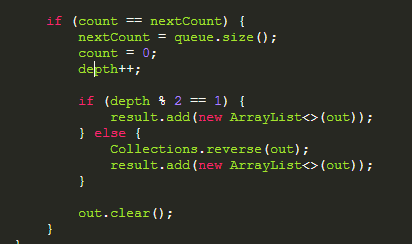
**}**

**}**

按之字形顺序打印二叉树

请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。

可以将上一个改一下，记录树的层次，对偶数层将顺序颠倒即可：



# 数据流中的中位数

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

使用最小堆和最大堆，最小堆的数大于最大最大堆的数，第奇数个数插入最小堆中，偶数个插入最大堆中。

往最大堆中插入数字前，为了保证最大堆的数字总是小于最小堆的，应该先将该数字插入到最小堆中，找到最小值，将该最小值插入到最大堆。即过滤一下，往最小堆插入时也同理需要过滤。

如果最后一次是插入到最小堆，即总共有奇数个数字，输出最小堆顶数字，否则输出2个堆顶的平均数。

**import** java**.**util**.\*;**

public class Solution **{**

PriorityQueue**<**Integer**>** minHeap **=** **new** PriorityQueue**<**Integer**>();**

PriorityQueue**<**Integer**>** maxHeap **=** **new** PriorityQueue**<**Integer**>(**11**,** **new** Comparator**<**Integer**>()** **{**

@Override

public int compare**(**Integer o1**,** Integer o2**)** **{**

// TODO Auto-generated method stub

**return** o2**.**compareTo**(**o1**);**

**}**

**});**

int count **=** 0**;**

public void Insert**(**Integer num**)** **{**

count**++;**

//当数据总数为奇数时，新加入的元素，应当进入小根堆

//（注意不是直接进入小根堆，而是经大根堆筛选后取大根堆中最大元素进入小根堆）

//1.新加入的元素先入到大根堆，由大根堆筛选出堆中最大的元素

**if** **(**count **%** 2 **==** 1**)** **{**

maxHeap**.**add**(**num**);**

int filter **=** maxHeap**.**poll**();**

minHeap**.**add**(**filter**);**

**}** **else** **{**

//1.新加入的元素先入到小根堆，由小根堆筛选出堆中最小的元素

minHeap**.**add**(**num**);**

int filter **=** minHeap**.**poll**();**

maxHeap**.**add**(**filter**);**

**}**

**}**

public Double GetMedian**()** **{**

double res **=** 0**;**

**if** **(**count **%** 2 **==** 1**)** **{**

res **=** minHeap**.**peek**();**

**}** **else** **{**

res **=** **((**double**)**minHeap**.**peek**()+**maxHeap**.**peek**())** **/** 2**;**

**}**

**return** res**;**

**}**

**}**

# 滑动窗口的最大值

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

复杂度

时间 O(NlogK) 空间 O(K)

维护一个大小为K的最大堆，依此维护一个大小为K的窗口，每次读入一个新数，都把堆中窗口最左边的数扔掉，再把新数加入堆中，这样堆顶就是这个窗口内最大的值。

**import** java**.**util**.\*;**

public class Solution **{**

public ArrayList**<**Integer**>** maxInWindows**(**int **[]** nums**,** int size**)**

**{**

ArrayList**<**Integer**>** res **=** **new** ArrayList**<>();**

**if(**nums **==** **null** **||** nums**.**length **==** 0 **||** size **==** 0**)** **return** res**;**

PriorityQueue**<**Integer**>** maxHeap **=** **new** PriorityQueue**<**Integer**>(**11**,** **new** Comparator**<**Integer**>()** **{**

@Override

public int compare**(**Integer o1**,** Integer o2**)** **{**

//PriorityQueue默认是小顶堆，实现大顶堆，需要反转默认排序器

**return** o2**.**compareTo**(**o1**);**

**}**

**});**

**for** **(**int i**=**0**;** i**<**nums**.**length**;** i**++)** **{**

//去掉窗口最左边的数

**if** **(**i **>=** size**)** maxHeap**.**remove**(**nums**[**i**-**size**]);**

// 把新的数加入窗口的堆中

maxHeap**.**add**(**nums**[**i**]);**

// 堆顶就是窗口的最大值

**if** **(**i **+** 1 **>=** size**)** **{**

res**.**add**(**maxHeap**.**peek**());**

**}**

**}**

**return** res**;**

**}**

**}**

复杂度

时间 O(N) 空间 O(K)

思路

我们用双向队列可以在O(N)时间内解决这题。当我们遇到新的数时，将新的数和双向队列的末尾比较，如果末尾比新数小，则把末尾扔掉，直到该队列的末尾比新数大或者队列为空的时候才住手。这样，我们可以保证队列里的元素是从头到尾降序的，由于队列里只有窗口内的数，所以他们其实就是窗口内第一大，第二大，第三大...的数。保持队列里只有窗口内数的方法和上个解法一样，也是每来一个新的把窗口最左边的扔掉，然后把新的加进去。然而由于我们在加新数的时候，已经把很多没用的数给扔了，这样队列头部的数并不一定是窗口最左边的数。这里的技巧是，我们队列中存的是那个数在原数组中的下标，这样我们既可以直到这个数的值，也可以知道该数是不是窗口最左边的数。这里为什么时间复杂度是O(N)呢？因为每个数只可能被操作最多两次，一次是加入队列的时候，一次是因为有别的更大数在后面，所以被扔掉，或者因为出了窗口而被扔掉。

**import** java**.**util**.\*;**

public class Solution **{**

public ArrayList**<**Integer**>** maxInWindows**(**int **[]** nums**,** int size**)**

**{**

ArrayList**<**Integer**>** res **=** **new** ArrayList**<>();**

**if(**nums **==** **null** **||** nums**.**length **==** 0 **||** size **==** 0**)** **return** res**;**

LinkedList**<**Integer**>** deque **=** **new** LinkedList**<>();**

**for** **(**int i**=**0**;** i**<**nums**.**length**;** i**++)** **{**

// 每当新数进来时，如果发现队列头部的数的下标，是窗口最左边数的下标，则扔掉

int leftRemove **=** i **-** size**;**

**if** **(!**deque**.**isEmpty**()** **&&** deque**.**peek**()** **==** leftRemove**)** deque**.**removeFirst**();**

// 把队列尾部所有比新数小的都扔掉，保证队列是降序的

**while** **(!**deque**.**isEmpty**()** **&&** nums**[**i**]** **>** nums**[**deque**.**peekLast**()])** deque**.**removeLast**();**

//加入新数

deque**.**offerLast**(**i**);**

// 队列头部就是该窗口内第一大的

**if** **(**i **+** 1 **>=** size**)** res**.**add**(**nums**[**deque**.**peek**()]);**

**}**

**return** res**;**

**}**

**}**

# 堆的实现

**import** java**.**util**.\*;**

public class Main **{**

static int**[]** arr **=** **{**2**,**2**,**2**,**2**,**3**};**

public static void main**(**String**[]** args**)** **{**

MyHeap myHeap **=** **new** MyHeap**(true);**

**for** **(**int i**=**1**;** i**<=**5**;** i**++)** **{**

myHeap**.**add**(**i**);**

**}**

System**.**out**.**println**(**myHeap**);**

myHeap**.**pop**();**

System**.**out**.**println**(**myHeap**);**

myHeap**.**pop**();**

System**.**out**.**println**(**myHeap**);**

myHeap**.**pop**();**

System**.**out**.**println**(**myHeap**);**

myHeap**.**pop**();**

System**.**out**.**println**(**myHeap**);**

myHeap**.**pop**();**

System**.**out**.**println**(**myHeap**);**

myHeap**.**pop**();**

System**.**out**.**println**(**myHeap**);**

**}**

**}**

class MyHeap **{**

ArrayList**<**Integer**>** table **=** **new** ArrayList**<>();**

boolean type **=** **false;** //false 表示最小堆， true最大堆

public MyHeap**()** **{**

**}**

/\*

\* false 表示最小堆， true最大堆,默认最小堆

\* \*/

public MyHeap**(**boolean type**)** **{**

**this();**

**this.**type **=** type**;**

**}**

//插入一个值，插入到最后位置，然后上浮调整

public void add**(**Integer val**)** **{**

table**.**add**(**val**);**

int insert **=** table**.**size**()** **-** 1**;**

int parent **=** getParentId**(**insert**);**

**while** **(**parent **>=** 0 **&&** insert **!=** 0**)** **{**

**if** **(**compare**(**table**.**get**(**parent**),** table**.**get**(**insert**))** **>=** 0**)** **{**

**break;**

**}**

//需要交换

swap**(**parent**,** insert**);**

insert **=** parent**;**

parent **=** getParentId**(**insert**);**

**}**

**}**

//弹出堆顶元素

public Integer pop**()** **{**

**if** **(**size**()** **==** 0**)** **return** **null;**

int lastIndex **=** table**.**size**()** **-** 1**;**

Integer res **=** table**.**get**(**0**);**

swap**(**lastIndex**,** 0**);**

table**.**remove**(**lastIndex**);**

shiftDown**(**0**);**

**return** res**;**

**}**

//查看堆顶元素

public Integer peek**()** **{**

**if** **(**size**()** **==** 0**)** **return** **null;**

**return** table**.**get**(**0**);**

**}**

public int size**()** **{**

**return** table**.**size**();**

**}**

//向下调整

private void shiftDown**(**int n**)** **{**

int child **;**

**while** **((**child **=** getLeftChild**(**n**))** **<** table**.**size**())** **{**

**if** **(**child**+**1 **<** table**.**size**()** **&&**

compare**(**table**.**get**(**child**+**1**),** table**.**get**(**child**))** **>** 0**)** **{**

child**++;**

**}**

**if** **(**compare**(**table**.**get**(**child**),** table**.**get**(**n**))** **>** 0**)** **{**

//需要交换

swap**(**n**,** child**);**

n **=** child**;**

**}** **else** **{**

**break;**

**}**

**}**

**}**

//false 表示最小堆， true最大堆，默认最小堆

private int compare**(**Integer a**,** Integer b**)** **{**

**if** **(**type**)** **{**

**return** a **-** b**;**

**}** **else** **{**

**return** b **-** a**;**

**}**

**}**

private int getParentId**(**int id**)** **{**

**return** **(**id**-**1**)/**2**;**

**}**

private int getLeftChild**(**int id**)** **{**

**return** id**\***2 **+** 1**;**

**}**

private void swap**(**int id1**,** int id2**)** **{**

int tmp **=** table**.**get**(**id1**);**

table**.**set**(**id1**,** table**.**get**(**id2**));**

table**.**set**(**id2**,** tmp**);**

**}**

@Override

public String toString**()** **{**

**return** "MyHeap [table=" **+** table **+** "]"**;**

**}**

**}**

# 链表的实现

public class Main **{**

public static void main**(**String**[]** args**)** **{**

MyLink link **=** **new** MyLink**();**

link**.**append**(**11**);**

link**.**append**(**22**);**

link**.**append**(**33**);**

link**.**append**(**44**);**

link**.**append**(**55**);**

System**.**out**.**println**(**link**);**

System**.**out**.**println**();**

System**.**out**.**println**(**link**.**get**(**5**));**

**}**

**}**

class MyLink **{**

//头节点指向第一个节点

class Node **{**

Object o**;**

Node next **=** **null;**

**}**

public Node head**;**

public int size**;**

public MyLink**()** **{**

head **=** **new** Node**();**

size **=** 0**;**

**}**

//在链表的最后位置插入一个节点

public void append**(**Object o**)** **{**

Node node **=** **new** Node**();**

node**.**o **=** o**;**

**if** **(**head**.**next **==** **null)** **{**

head**.**next **=** node**;**

**}** **else** **{**

Node iterator **=** head**;**

**while** **(**iterator**.**next **!=** **null)** **{**

iterator **=** iterator**.**next**;**

**}**

iterator**.**next **=** node**;**

**}**

size**++;**

**}**

//获取第i个节点的数据,没有则为null

public Object get**(**int i**)** **{**

Object res **=** **null;**

Node iterator **=** head**;**

int index **=** 0**;**

**while** **(**iterator **!=** **null** **&&** index **<** i**)** **{**

iterator **=** iterator**.**next**;**

index**++;**

**}**

**if** **(**iterator **!=** **null)** **{**

res **=** iterator**.**o**;**

**}**

**return** res**;**

**}**

//在第i个节点处插入 o

//i=0和i=1，都是在第一个节点插入数据

//成功返回true

public boolean insert**(**int i**,** Object o**)** **{**

Node iterator **=** head**;**

int index **=** 0**;**

**while** **(**iterator **!=** **null** **&&** index **<** i**-**1**)** **{**

iterator **=** iterator**.**next**;**

index**++;**

**}**

**if** **(**iterator **!=** **null)** **{**

Node node **=** **new** Node**();**

node**.**o **=** o**;**

node**.**next **=** iterator**.**next**;**

iterator**.**next **=** node**;**

size**++;**

**return** **true;**

**}**

**return** **false;**

**}**

//删除第i个节点的数据，并返回删除的数据，删除失败返回null

public Object delete**(**int i**)** **{**

Object returnObject **=** **null;**

Node iterator **=** head**;**

int index **=** 0**;**

**while** **(**iterator**.**next **!=** **null** **&&** index **<** i**-**1**)** **{**

iterator **=** iterator**.**next**;**

index**++;**

**}**

**if** **(**iterator**.**next **!=** **null)** **{**

returnObject **=** iterator**.**next**.**o**;**

iterator**.**next **=** iterator**.**next**.**next**;**

size**--;**

**}**

**return** returnObject**;**

**}**

public String toString**()** **{**

Node iterator **=** head**;**

//指向第一个节点

StringBuilder sb **=** **new** StringBuilder**();**

sb**.**append**(**"size:"**+**size**+**" {"**);**

**while** **(**iterator**.**next **!=** **null)** **{**

iterator **=** iterator**.**next**;**

sb**.**append**(**iterator**.**o**+**" "**);**

**}**

**return** sb**.**deleteCharAt**(**sb**.**length**()-**1**).**append**(**"}"**).**toString**();**

**}**

**}**

# 栈的实现

# 队列的实现

# 矩阵中的路径

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如 a b c e s f c s a d e e 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

其实就是dfs算法，类似找迷宫。

public class Solution **{**

static public boolean hasPath**(**char**[]** matrix**,** int rows**,** int cols**,** char**[]** str**)**

**{**

**if** **(**matrix **==** **null** **||** str **==** **null** **||** str**.**length **==** 0 **||**

matrix**.**length **==** 0 **||** matrix**.**length **!=** rows**\***cols**)** **return** **false;**

**for** **(**int i**=**0**;** i**<**rows**;** i**++)** **{**

**for** **(**int j**=**0**;** j**<**cols**;** j**++)** **{**

//寻找起始位置

**if** **(**matrix**[**i**\***cols**+**j**]** **==** str**[**0**])** **{**

isHas **=** **false;**

dfs**(**matrix**,** rows**,** cols**,** **new** int**[**rows**\***cols**],** str**,** 0**,** j**,** i**);**

**if** **(**isHas**)**

**return** isHas**;**

**}**

**}**

**}**

**return** **false;**

**}**

static boolean isHas **=** **false;** //找到字符串标志

//book是用来标记一个位置是否被进入过

//id 是遍历str字符串时的下标

//x,y是对应二维矩阵中的坐标位置

static void dfs**(**char**[]** matrix**,** int rows**,** int cols**,**

int**[]** book**,** char**[]** str**,** int id**,** int x**,** int y**)** **{**

//字符串被找到

**if** **(**id **==** str**.**length**)** **{**

isHas **=** **true;**

**return** **;**

**}**

//边界判断

**if** **(**x **<** 0 **||** x **>=** cols **||** y **<** 0 **||** y **>=** rows**)** **return;**

int index **=** x **+** y**\***cols**;**//对应matrix下标

**if** **(**matrix**[**index**]** **==** str**[**id**]** **&&** book**[**index**]** **==** 0**)** **{**

book**[**index**]** **=** 1**;**//标志已经进入

//进入下一批格子

int**[][]** next **=** **{{**0**,**1**},{**1**,**0**},{**0**,-**1**},{-**1**,**0**}};**

**for** **(**int i**=**0**;** i**<**4**;** i**++)** **{**

int tx **=** x **+** next**[**i**][**0**];**

int ty **=** y **+** next**[**i**][**1**];**

dfs**(**matrix**,** rows**,** cols**,** book**,** str**,** id**+**1**,** tx**,** ty**);**

**}**

book**[**index**]** **=** 0**;**//清除标志

**}**

**}**

**}**

改进后：

public class Solution **{**

static public boolean hasPath**(**char**[]** matrix**,** int rows**,** int cols**,** char**[]** str**)**

**{**

**if** **(**matrix **==** **null** **||** str **==** **null** **||** str**.**length **==** 0 **||**

matrix**.**length **==** 0 **||** matrix**.**length **!=** rows**\***cols**)** **return** **false;**

**for** **(**int i**=**0**;** i**<**rows**;** i**++)** **{**

**for** **(**int j**=**0**;** j**<**cols**;** j**++)** **{**

//寻找起始位置

**if** **(**matrix**[**i**\***cols**+**j**]** **==** str**[**0**])** **{**

**if** **(**dfs**(**matrix**,** rows**,** cols**,** **new** int**[**rows**\***cols**],** str**,** 0**,** j**,** i**))** **{**

**return** **true;**

**}**

**}**

**}**

**}**

**return** **false;**

**}**

//book是用来标记一个位置是否被进入过

//id 是遍历str字符串时的下标

//x,y是对应二维矩阵中的坐标位置

static boolean dfs**(**char**[]** matrix**,** int rows**,** int cols**,**

int**[]** book**,** char**[]** str**,** int id**,** int x**,** int y**)** **{**

boolean res **=** **false;**

//字符串被找到

**if** **(**id **==** str**.**length**)** **{**

**return** **true;**

**}**

//边界判断

**if** **(**x **<** 0 **||** x **>=** cols **||** y **<** 0 **||** y **>=** rows**)** **return** **false;**

int index **=** x **+** y**\***cols**;**//对应matrix下标

**if** **(**matrix**[**index**]** **==** str**[**id**]** **&&** book**[**index**]** **==** 0**)** **{**

book**[**index**]** **=** 1**;**//标志已经进入

//进入下一批格子

int**[][]** next **=** **{{**0**,**1**},{**1**,**0**},{**0**,-**1**},{-**1**,**0**}};**

**for** **(**int i**=**0**;** i**<**4**;** i**++)** **{**

int tx **=** x **+** next**[**i**][**0**];**

int ty **=** y **+** next**[**i**][**1**];**

res **=** res **||** dfs**(**matrix**,** rows**,** cols**,** book**,** str**,** id**+**1**,** tx**,** ty**);**

**}**

book**[**index**]** **=** 0**;**//清除标志

**}**

**return** res**;**

**}**

**}**

# 机器人的运动范围

地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于k的格子。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子？

public class Solution **{**

static public boolean hasPath**(**char**[]** matrix**,** int rows**,** int cols**,** char**[]** str**)**

**{**

**if** **(**matrix **==** **null** **||** str **==** **null** **||** str**.**length **==** 0 **||**

matrix**.**length **==** 0 **||** matrix**.**length **!=** rows**\***cols**)** **return** **false;**

**for** **(**int i**=**0**;** i**<**rows**;** i**++)** **{**

**for** **(**int j**=**0**;** j**<**cols**;** j**++)** **{**

//寻找起始位置

**if** **(**matrix**[**i**\***cols**+**j**]** **==** str**[**0**])** **{**

**if** **(**dfs**(**matrix**,** rows**,** cols**,** **new** int**[**rows**\***cols**],** str**,** 0**,** j**,** i**))** **{**

**return** **true;**

**}**

**}**

**}**

**}**

**return** **false;**

**}**

//book是用来标记一个位置是否被进入过

//id 是遍历str字符串时的下标

//x,y是对应二维矩阵中的坐标位置

static boolean dfs**(**char**[]** matrix**,** int rows**,** int cols**,**

int**[]** book**,** char**[]** str**,** int id**,** int x**,** int y**)** **{**

boolean res **=** **false;**

//字符串被找到

**if** **(**id **==** str**.**length**)** **{**

**return** **true;**

**}**

//边界判断

**if** **(**x **<** 0 **||** x **>=** cols **||** y **<** 0 **||** y **>=** rows**)** **return** **false;**

int index **=** x **+** y**\***cols**;**//对应matrix下标

**if** **(**matrix**[**index**]** **==** str**[**id**]** **&&** book**[**index**]** **==** 0**)** **{**

book**[**index**]** **=** 1**;**//标志已经进入

//进入下一批格子

int**[][]** next **=** **{{**0**,**1**},{**1**,**0**},{**0**,-**1**},{-**1**,**0**}};**

**for** **(**int i**=**0**;** i**<**4**;** i**++)** **{**

int tx **=** x **+** next**[**i**][**0**];**

int ty **=** y **+** next**[**i**][**1**];**

res **=** res **||** dfs**(**matrix**,** rows**,** cols**,** book**,** str**,** id**+**1**,** tx**,** ty**);**

**}**

book**[**index**]** **=** 0**;**//清除标志

**}**

**return** res**;**

**}**

**}**

改进版

public class Solution **{**

static public int movingCount**(**int threshold**,** int rows**,** int cols**)**

**{**

int**[][]** book **=** **new** int**[**rows**][**cols**];**

**return** dfs**(**book**,** 0**,** 0**,** threshold**);**

**}**

static public int dfs**(**int**[][]** book**,** int x**,** int y**,** int threshold**)** **{**

int count **=** 0**;**

int**[][]** next **=** **{{**0**,**1**},{**1**,**0**},{**0**,-**1**},{-**1**,**0**}};**

int rows **=** book**.**length**;**

int cols **=** book**[**0**].**length**;**

//边界判断

**if** **(**x **<** 0 **||** x **>=** rows **||** y **<** 0 **||** y **>=** cols**)** **return** 0**;**

**if** **(**book**[**x**][**y**]** **==** 0 **&&** canGo**(**x**,** y**,** threshold**))** **{**

book**[**x**][**y**]** **=** 1**;**

count**++;**//计数

**for** **(**int i**=**0**;** i**<**4**;** i**++)** **{**

int tx **=** x **+** next**[**i**][**0**];**

int ty **=** y **+** next**[**i**][**1**];**

count **+=** dfs**(**book**,** tx**,** ty**,** threshold**);**//计数

**}**

**}**

**return** count**;**

**}**

static private boolean canGo**(**int x**,** int y**,** int threshold**)** **{**

**return** **(**getSum**(**x**)** **+** getSum**(**y**))** **<=** threshold**;**

**}**

static int getSum**(**int x**)** **{**

int sum **=** 0**;**

**while** **(**x **>** 0**)** **{**

sum **+=** x **%** 10**;**

x **/=** 10**;**

**}**

**return** sum**;**

**}**

**}**

# 合唱队

计算最少出列多少位同学，使得剩下的同学排成合唱队形

说明：

N位同学站成一排，音乐老师要请其中的(N-K)位同学出列，使得剩下的K位同学排成合唱队形。   
合唱队形是指这样的一种队形：设K位同学从左到右依次编号为1，2…，K，他们的身高分别为T1，T2，…，TK，   则他们的身高满足存在i（1<=i<=K）使得Ti<T2<......<Ti-1<Ti>Ti+1>......>TK。   
     你的任务是，已知所有N位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

**import** java**.**util**.**Scanner**;**

public class Main **{**

public static void main**(**String**[]** args**)** **{**

// TODO Auto-generated method stub

Scanner in **=** **new** Scanner**(**System**.**in**);**

**while** **(**in**.**hasNext**())** **{**

int n **=** in**.**nextInt**();**

int**[]** nums **=** **new** int**[**n**];**

int**[]** small **=** **new** int**[**n**];**

int**[]** big **=** **new** int**[**n**];**

**for** **(**int i**=**0**;** i**<**n**;** i**++)** **{**

nums**[**i**]** **=** in**.**nextInt**();**

**}**

//动态规划 small[i],表示nums[i]左边递增序列的个数

small**[**0**]** **=** 0**;**

**for** **(**int i**=**1**;** i**<**n**;** i**++)** **{**

int k **=** i**;**

**for** **(**int j**=**i**-**1**;** j**>=**0**;** j**--)** **{**

**if** **(**nums**[**k**]** **>** nums**[**j**])** **{**

small**[**k**]** **=** Math**.**max**(**small**[**j**]+**1**,** small**[**k**]);**

**}**

**}**

**}**

//动态规划 big[i],表示nums[i]右边递减序列的个数

big**[**n**-**1**]** **=** 0**;**

**for** **(**int i**=**n**-**2**;** i**>=**0**;** i**--)** **{**

int k **=** i**;**

**for** **(**int j**=**i**+**1**;** j**<**n**;** j**++)** **{**

**if** **(**nums**[**k**]** **>** nums**[**j**])** **{**

big**[**k**]** **=** Math**.**max**(**big**[**j**]+**1**,** big**[**k**]);**

**}**

**}**

**}**

int max **=** 0**;**

**for** **(**int i**=**0**;** i**<**n**;** i**++)** **{**

max **=** Math**.**max**(**max**,** small**[**i**]+**big**[**i**]);**

**}**

// System.out.println(Arrays.toString(small));

// System.out.println(Arrays.toString(big));

System**.**out**.**println**(**n**-**max**-**1**);**

**}**

**}**

**}**