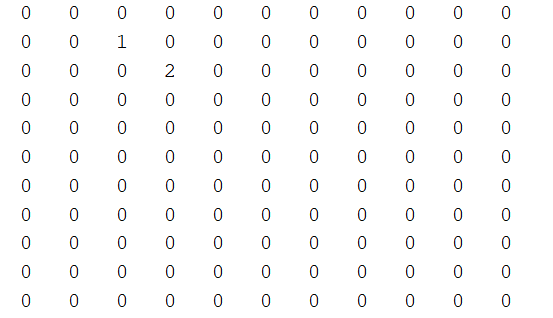
# Java数据结构与算法

### 稀疏数组

#### 稀疏数组概念

稀疏数组可以看做是普通数组的压缩，但是这里说的普通数组是指无效数据量远大于有效数据量的数组

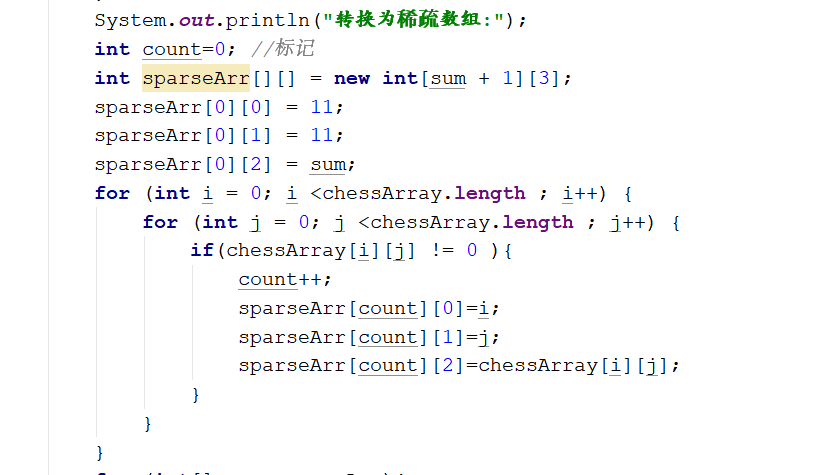
例如:





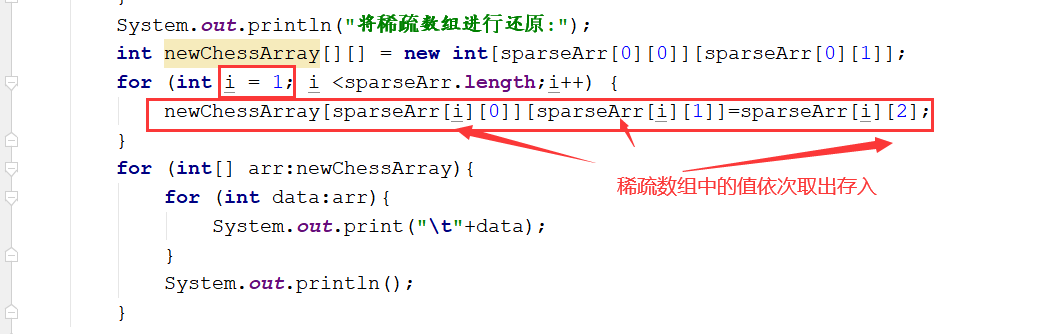
#### 稀疏数组压缩

1. 对原始的数组进行遍历当遇到不为0的值就将标记加1然后统计出原始数组中不为0的值
2. 当一个数组被压缩为稀疏数组时，它只有**N行3列的形式。**然后定义稀疏数组int spare[][]=new int[N][3]
3. 在稀疏数组中 它的第一列代表行 第二列代表列 第三列代表当前的值（**注意：稀疏数组第一行的三个值分别为原始数组的行列和有效值个数**）
4. 然后遍历原始数组 遇到不为0的值就将行列值存入稀疏数组中



#### 稀疏数组还原

1. 创建一个还原的二维数组,初始化大小为稀疏数组中第一行的第一列和第二列的值
2. 遍历稀疏数组，将稀疏数组的行列值存入二维数组中



#### 稀疏数组完整代码实现

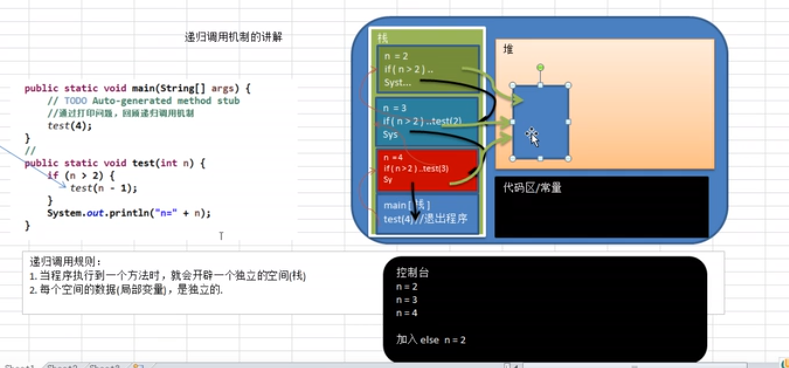
*/\*\*  
 \* 稀疏数组  
 \*/***public class** sparseArray {  
 **public static void** main(String[] args) **throws** Exception {  
 */\*  
 创建一个11\*11的二维数组  
 0代表没有棋子 1代表黑子 2代表白子  
 \*/* **int** chessArray[][] = **new int**[11][11];  
 chessArray[1][2] = 1;  
 chessArray[2][3] = 2;  
 *//输出原始的二位数组* System.***out***.println(**"原始的二位数组:"**);  
 **for** (**int**[] arr : chessArray) {  
 **for** (**int** data : arr) {  
 System.***out***.print(**"\t"** + data);  
 }  
 System.***out***.println();  
 }  
 */\*  
 遍历原始数组 得到稀疏数组大小  
 \*/* **int** sum = 0; *//标记数组不为0的个数* **for** (**int** i = 0; i < chessArray.**length**; i++) {  
 **for** (**int** j = 0; j < chessArray.**length**; j++) {  
 **if** (chessArray[i][j] != 0) {  
 sum++;  
 }  
 }  
 }  
 System.***out***.println(**"转换为稀疏数组:"**);  
 **int** count=0; *//标记* **int** sparseArr[][] = **new int**[sum + 1][3];  
 sparseArr[0][0] = 11;  
 sparseArr[0][1] = 11;  
 sparseArr[0][2] = sum;  
 **for** (**int** i = 0; i <chessArray.**length** ; i++) {  
 **for** (**int** j = 0; j <chessArray.**length** ; j++) {  
 **if**(chessArray[i][j] != 0 ){  
 count++;  
 sparseArr[count][0]=i;  
 sparseArr[count][1]=j;  
 sparseArr[count][2]=chessArray[i][j];  
 }  
 }  
 }  
 **for** (**int**[] arr:sparseArr){  
 **for** (**int** data:arr){  
 System.***out***.print(**"\t"**+data);  
 }  
 System.***out***.println();  
 }  
 System.***out***.println(**"将稀疏数组进行还原:"**);  
 **int** newChessArray[][] = **new int**[sparseArr[0][0]][sparseArr[0][1]];  
 **for** (**int** i = 1; i <sparseArr.**length**;i++) {  
 newChessArray[sparseArr[i][0]][sparseArr[i][1]]=sparseArr[i][2];  
 }  
 **for** (**int**[] arr:newChessArray){  
 **for** (**int** data:arr){  
 System.***out***.print(**"\t"**+data);  
 }  
 System.***out***.println();  
 }  
 }  
}

### 递归算法

#### 递归算法需要遵守的重要规则

1. 执行一个方法时，就创建一个新的受保护的独立空间（栈空间）。
2. 方法的局部变量是独立的，不相互影响，比如n变量。
3. 如果方法中使用的是引用类型变量（比如数组）就会共享该引用类型的数据。
4. 递归必须向退出条件无限逼近，否则就会形成无限递归,出现StackOverflowError。
5. 当一个方法执行完毕，或者遇到return，就会返回，遵守谁调用就将结果返回给谁的原则，同时当方法执行完毕或者返回时，该方法也就会执行完毕。

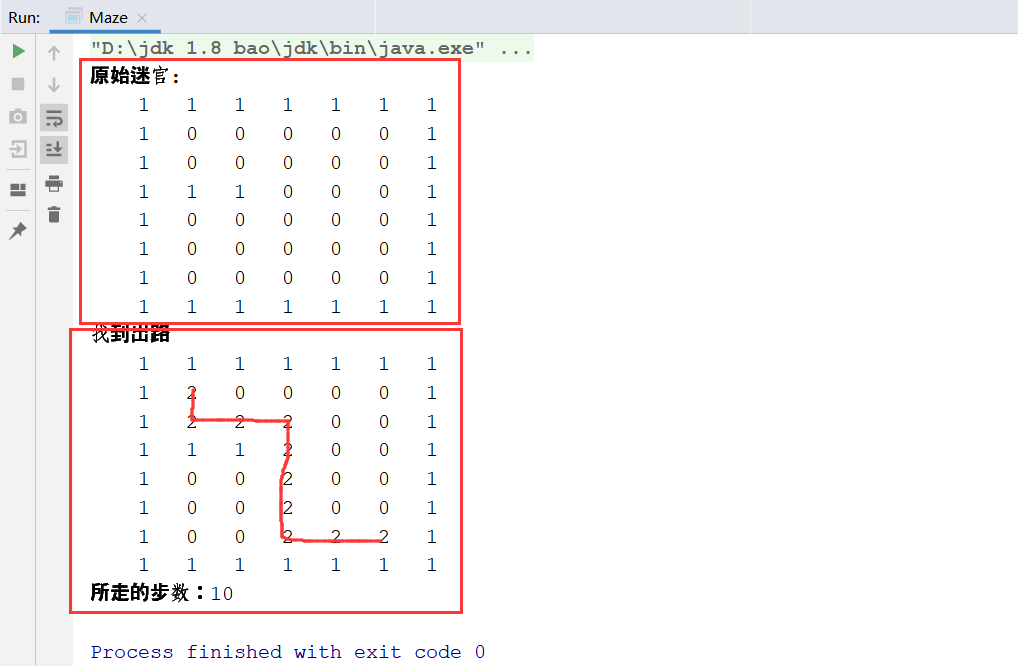
#### 递归调用机制



#### 迷宫问题（递归实现）

*\*\*  
 \* 利用递归算法解决迷宫问题  
 \*/***public class** Maze {  
 **public static void** main(String[] args) {  
 *//先定义一个二位数组 用1表示墙 0表示没有走过的路 2表示当前路可通 3表示不可通* **int**[][] maze = **new int**[8][7];  
 **for** (**int** i = 0; i < 8; i++) {  
 maze[i][0] = 1;  
 maze[i][6] = 1;  
 }  
 **for** (**int** i = 0; i < 7; i++) {  
 maze[0][i] = 1;  
 maze[7][i] = 1;  
 }  
 maze[3][1] = 1;  
 maze[3][2] = 1;  
 System.***out***.println(**"原始迷宫:"**);  
 **for** (**int**[] ma : maze) {  
 **for** (**int** m : ma) {  
 System.***out***.print(**"\t"** + m);  
 }  
 System.***out***.println();  
 }  
 **boolean** way = *getWay*(maze, 1, 1);  
 **int** k=0;  
 **if** (way) {  
 System.***out***.println(**"找到出路"**);  
 **for** (**int**[] ma : maze) {  
 **for** (**int** m : ma) {  
 **if**(m==2){  
 k++;  
 }  
 System.***out***.print(**"\t"** + m);  
 }  
 System.***out***.println();  
 }  
 System.***out***.println(**"所走的步数："**+k);  
 } **else** {  
 System.***out***.println(**"未找到出路"**);  
 }  
 }  
 *//找寻路径的方法(下右上左)* **public static boolean** getWay(**int**[][] maze, **int** i, **int** j) {  
 **if** (maze[6][5] == 2) { *//找到出路* **return true**;  
 } **else** {  
 *//当前路径是是否没走过* **if** (maze[i][j] == 0) {  
 maze[i][j] = 2; *//假设可以走通* **if** (*getWay*(maze, i + 1, j)) {  
 **return true**;  
 } **else if** (*getWay*(maze, i, j + 1)) {  
 **return true**;  
 } **else if** (*getWay*(maze, i - 1, j)) {  
 **return true**;  
 } **else if** (*getWay*(maze, i, j - 1)) {  
 **return true**;  
 } **else** {  
 maze[i][j] = 3;  
 **return false**;  
 }  
 } **else** { *//maze[i][j]!=0 可取 1 2 3* **if** (maze[i][j]==2) *//如果为2 返回true* **return true**;  
 **else** {  
 **return false**; *//其他返回false* }  
 }  
 }  
 }  
}

代码运行截图：



#### 八皇后问题（递归-回溯实现）

八皇后问题

八皇后问题，一个古老而著名的问题，是[回溯算法](https://baike.baidu.com/item/%E5%9B%9E%E6%BA%AF%E7%AE%97%E6%B3%95/9258495" \t "_blank)的典型案例。该问题由国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出：在 8×8 格的[国际象棋](https://baike.baidu.com/item/%E5%9B%BD%E9%99%85%E8%B1%A1%E6%A3%8B/80888" \t "_blank)上摆放八个[皇后](https://baike.baidu.com/item/%E7%9A%87%E5%90%8E/15860305)，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。

八皇后算法问题思路分析

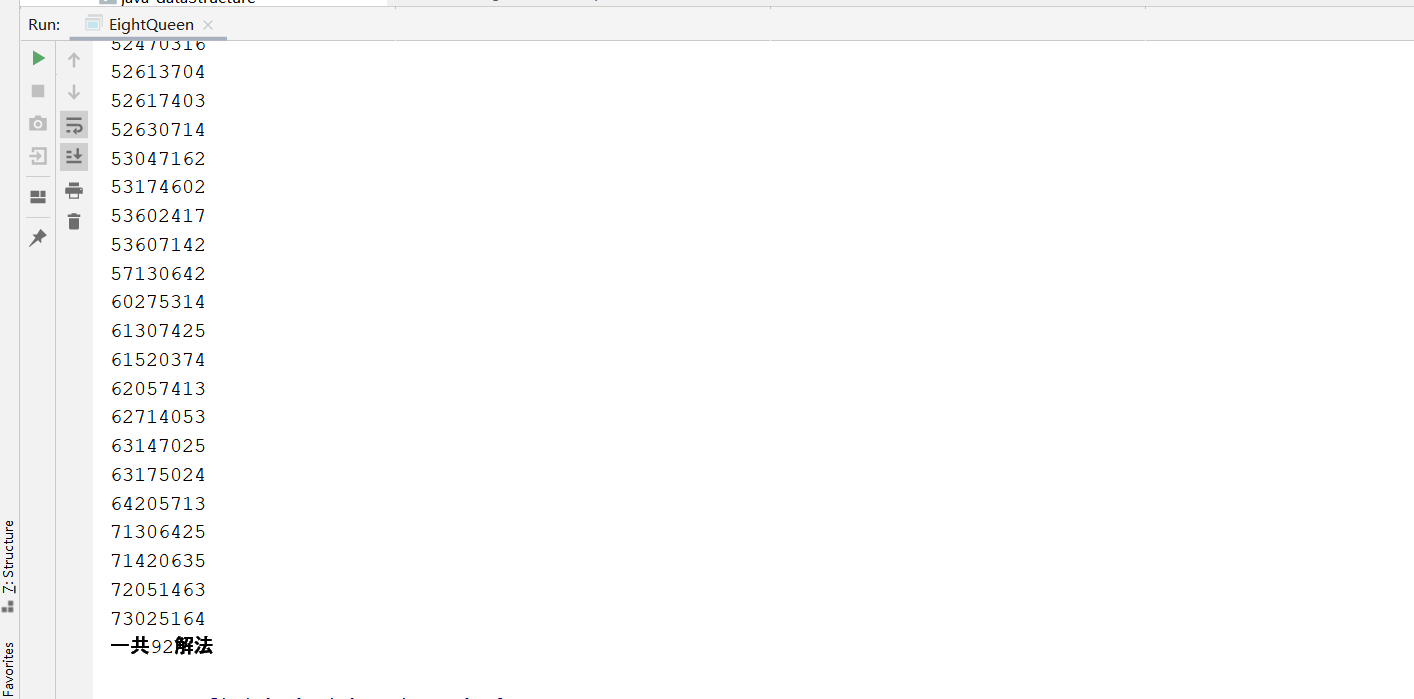
1. 第一个皇后先放在第一行第一列
2. 第二个皇后放在第二行第一列，然后判断是否可行，如果可行，继续放在第二列，第三列，依次把所有列都放完，找到一个合适的
3. 继续放第三个皇后，还是第一列，第二列……直到第8个皇后也能放在一个不冲突的位置，算是找到了一个正确解
4. 当得到一个正确的解时，在栈回退到上一个栈时，就会开始回溯，即将第一个皇后放到第一列的所有解全部得到
5. 然后后头继续第一个皇后放第二列，后面继续循环执行1，2，3，4步骤

代码实现八皇后问题

*/\*\*  
 \* 八皇后问题  
 \*/***public class** EightQueen {  
 **int max** = 8; *//定义皇后的最大个数* **int**[] **array** = **new int**[**max**];  
 **static int** *count*=0; *//统计解法* **public static void** main(String[] args) {  
 */\*使用一个一维数模拟棋盘  
 因为八皇后中放置皇后的顺序为：第一个先放第一行第一列 然后放第二个在第二行...  
 而我们定义的形如array[8]={0，1，7，0，0，0，0，0}刚好满足行列对正  
 array[0]=0代表第一个皇后放在第一行第一列 array[1]=1代表第二个皇后放在第二行第2个位置  
 当然每一个array[]= 可以有0 1 2 3 4 5 6 7 取法  
 就可以将二维数组简化为一个一维数组 int []array = new int[max]  
 \*/* EightQueen eightQueen = **new** EightQueen();  
 eightQueen.check(0);  
 System.***out***.println(**"一共"**+*count*+**"解法"**);  
  
 }  
  
 */\*\*  
 \* 检查当前放置的皇后是否会出现冲突 如果出现则回溯  
 \*  
 \** ***@param n*** *放置第n个皇后  
 \*/* **private void** check(**int** n) {  
 **if** (n == **max**) {*//把第8个皇后放置完成没有出现冲突* printQueen();*//打印* **return**;  
 }  
 *//依次放入皇后检查是否冲突* **for** (**int** i = 0; i < **max**; i++) {  
 *//当前皇后从第01234567开始放置* **array**[n] = i;  
 **if** (judge(n)){ *//不冲突* check(n+1);*//开始检查后一个皇后* }  
 *//如果冲突 则会执行array[n]=i即判断当前皇后是否可以放置本行的下一个位置* }

n--; *//不满足条件 n-- 产生回溯到上一个皇后位置*  
 }  
  
 */\*\*  
 \* 判断放置当前位置是否产生冲突  
 \*  
 \** ***@param n*** *第n个皇后(实际为n-1皇后因为数组是从0开始)  
 \** ***@return*** *\*/* **private boolean** judge(**int** n) {  
 **for** (**int** i = 0; i < n; i++) {  
 */\*  
 array[]={0,1,2,3,4,5,6,7}表示第一个皇后放第一行第一列 第二个皇后放第二行第2列...  
 array[i]== array[n]表示判断第n个皇后和前面的n-1个皇后是否在同一行  
 Math.abs(n-i)==Math.abs(array[n]-array[i])判断第n个皇后是否和第i个在同一斜线  
 例如：n=1(实际为第二个皇后) 放置第二行第某列(具体看数组array[n]的值 可取01234567)  
 \*/* **if** (**array**[i] == **array**[n] || Math.*abs*(n - i) == Math.*abs*(**array**[n] - **array**[i])) {  
 **return false**;  
 }  
 }  
 **return true**;  
 }  
  
 *//打印解法* **private void** printQueen() {  
 *count*++;  
 **for** (**int** i = 0; i < **max**; i++) {  
 System.***out***.print(**array**[i]);  
 }  
 System.***out***.println();  
 }  
  
}

运行结果：



### 队列

#### 普通队列(数组实现)

队列是一个先进先出的数据结构

*/\*\*  
 \* 利用数组实现队列  
 \*/***public class** arrayQueue {  
 **public static void** main(String[] args) {  
 queue q = **new** queue(3);  
 **char** key=**' '**;  
 Scanner scanner = **new** Scanner(System.***in***);  
 **boolean** loop = **true**;  
 **while** (loop){  
 System.***out***.println(**"s(show):显示队列"**);  
 System.***out***.println(**"a(add):添加队列"**);  
 System.***out***.println(**"g(get):从队列取出数据"**);  
 System.***out***.println(**"h(get):队列头数据"**);  
 key=scanner.next().charAt(0);*//接收一个字符* **switch** (key){  
 **case 's'**:  
 q.showQueue();**break**;  
 **case 'a'**:  
 **int** data=scanner.nextInt();  
 q.addQueue(data);  
 **break**;  
 **case 'g'**:  
 **int** queue = q.getQueue();  
 System.***out***.println(**"取出数据"**+queue);  
 **break**;  
 **case 'h'**:  
 **int** headQueue = q.headQueue();  
 System.***out***.println(**"队列头数据"**+headQueue);  
 **break**;  
 **default**:  
 **break**;  
 }  
 }  
 }  
}  
*/\*\*  
 \* 创建一个队列  
 \*/***class** queue {  
 **private int maxSize**; *//标记队列最大容量* **private int front**; *//队列的头* **private int footer**; *//队列的尾* **private int**[] **arr**; *//存储队列的值* **public** queue(**int** maxSize) {  
 **this**.**maxSize** = maxSize;  
 **arr** = **new int**[maxSize];  
 **front** = -1;  
 **footer** = -1;  
 }  
 */\*\*  
 \* 判断是否队空  
 \*  
 \** ***@return*** *\*/* **public boolean** isEmpty() {  
 **return front** == **footer**;  
 }  
  
 */\*\*  
 \* 判断是否队满  
 \*  
 \** ***@return*** *\*/* **public boolean** isFull() {  
 **return front** == (**maxSize** - 1);  
 }  
 */\*\*  
 \* 队列插入数据  
 \*/* **public void** addQueue(**int** data){  
 **if**(isFull()){  
 System.***out***.println(**"队满，无法插入数据"**);  
 **return**;  
 }  
 **footer**++;  
 **arr**[**footer**]=data;  
 }  
 */\*\*  
 \* 数据出队列  
 \** ***@return*** *\*/* **public int** getQueue(){  
 **if**(isEmpty()){  
 **throw new** RuntimeException(**"队列为空"**);  
 }  
 **front**++;  
 **return arr**[**front**];  
 }  
  
 */\*\*  
 \* 取队列头数据  
 \** ***@return*** *\*/* **public int** headQueue(){  
 **if**(isEmpty()){  
 **throw new** RuntimeException(**"队列为空"**);  
 }  
 **return arr**[**front**+1];  
 }  
 */\*\*  
 \* 显示队列数据  
 \*/* **public void** showQueue(){  
 **if**(isEmpty()){  
 System.***out***.println(**"队列为空，没有数据可显示"**);  
 **return**;  
 }  
 **for** (**int** ar:**arr**){  
 System.***out***.println(ar);  
 }  
 }  
}

#### 环形队列(数组实现)

*/\*\*  
 \* 数组模拟环形队列  
 \* 思路分析：  
 \* 1.front变量的含义做调整：  
 \* front就指向队列的第一个元素，也就是说arr[front]就是队列的第一个元素 front的初始值为0  
 \* 2.footer变量的含义做调整：  
 \* footer就指向队列的最后一个元素的后一位 因为需要空出一个空间作为约定 front的初始值为0  
 \* 3.当队列满的条件： front == (footer+1)%maxSize  
 \* 4.当队列为空的条件: front==footer  
 \* 5.队列中有效元素： (footer-front+maxSize)%maxSize  
 \*/***public class** loopQueue {  
 **public static void** main(String[] args) {  
 queue2 q = **new** queue2(3);  
 **char** key=**' '**;  
 Scanner scanner = **new** Scanner(System.***in***);  
 **boolean** loop = **true**;  
 **while** (loop){  
 System.***out***.println(**"s(show):显示队列"**);  
 System.***out***.println(**"a(add):添加队列"**);  
 System.***out***.println(**"g(get):从队列取出数据"**);  
 System.***out***.println(**"h(get):队列头数据"**);  
 key=scanner.next().charAt(0);*//接收一个字符* **switch** (key){  
 **case 's'**:  
 q.showQueue();**break**;  
 **case 'a'**:  
 **int** data=scanner.nextInt();  
 q.addQueue(data);  
 **break**;  
 **case 'g'**:  
 **int** queue = q.getQueue();  
 System.***out***.println(**"取出数据"**+queue);  
 **break**;  
 **case 'h'**:  
 **int** headQueue = q.headQueue();  
 System.***out***.println(**"队列头数据"**+headQueue);  
 **break**;  
 **default**:  
 **break**;  
 }  
 }  
 }  
}  
  
**class** queue2 {  
 **private int front**; *//队首* **private int footer**;*//队尾* **private int maxSize**;  
 **private int**[] **arr**;  
  
 **public** queue2(**int** maxSize) {  
 **this**.**maxSize** = maxSize;  
 **arr** = **new int**[maxSize];  
 **footer** = 0;  
 **front** = 0;  
 }  
  
 */\*\*  
 \* 判断是否队空  
 \*  
 \** ***@return*** *\*/* **public boolean** isEmpty() {  
 **return front** == **footer**;  
 }  
  
 */\*\*  
 \* 判断是否队满  
 \*  
 \** ***@return*** *\*/* **public boolean** isFull() {  
 **return front** == (**footer** + 1) % **maxSize**;  
 }  
  
 */\*\*  
 \* 队列插入数据  
 \*/* **public void** addQueue(**int** data) {  
 **if** (isFull()) {  
 System.***out***.println(**"队满，无法插入数据"**);  
 **return**;  
 }  
 **arr**[**footer**] = data;  
 *//footer指针后移 因为是环形队列 所以使用取模计算后移位置* **footer** = (**footer** + 1) % **maxSize**;  
 }  
 */\*\*  
 \* 数据出队列  
 \** ***@return*** *\*/* **public int** getQueue(){  
 **if**(isEmpty()){  
 **throw new** RuntimeException(**"队列为空"**);  
 }  
 **int** value = **arr**[**front**];  
 **front** = (**front**+1)%**maxSize**;  
 **return** value;  
 }  
  
 */\*\*  
 \* 取队列头数据  
 \** ***@return*** *\*/* **public int** headQueue(){  
 **if**(isEmpty()){  
 **throw new** RuntimeException(**"队列为空"**);  
 }  
 **return arr**[**front**];  
 }  
 */\*\*  
 \* 显示队列数据  
 \*/* **public void** showQueue(){  
 **if**(isEmpty()){  
 System.***out***.println(**"队列为空，没有数据可显示"**);  
 **return**;  
 }  
 **for** (**int** i = **front**; i <(**front**+(**footer**-**front**+**maxSize**)%**maxSize**) ; i++) {  
 System.***out***.println(**arr**[i]);  
 }  
 }  
  
}

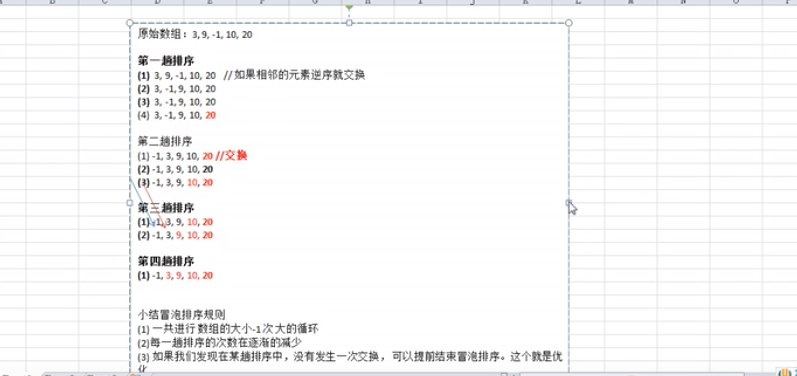
### 八大常用排序

#### 冒泡排序

**时间复杂度O(n^2)**

基本思想：通过对待排序序列从前向后（从下标较小的元素开始）依次比较相邻元素的值，若发逆序则交换，使值较大的元素逐渐从前移向后部。

注意：因为排序过程中，各元素不断的交互自己的位置，如果一趟比较下来没有进行交换，就说明序列有序。因此要在排序过程中设置一个flag判断元素是否进行过交换，从而减少不必要的步数。



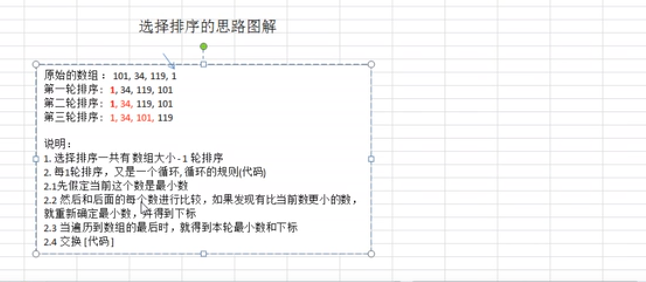
*/\*\*  
 \* 冒泡排序  
 \*/***public class** bubbleSort {  
 **public static void** main(String[] args) {  
 **int** temp=0;**boolean** flag=**false**;  
 **int** arr[]={10,8,11,5,6};  
 **for** (**int** i = 0; i <arr.**length**-1 ; i++) {  
 **for** (**int** j = 0; j <arr.**length**-1-i ; j++) {  
 **if** (arr[j]>arr[j+1]){  
 flag = **true**;  
 temp = arr[j];  
 arr[j] = arr[j+1];  
 arr[j+1] = temp;  
 }  
  
 }  
 **if** (!flag){  
 **break**; *//优化程序 如果一次都没有进入循环则证明数组本身是有序的* }**else** {  
 flag = **false**; *//重置flag* }  
 System.***out***.println(**"第"**+(i+1)+**"趟"**);  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
 }  
}

#### 选择排序

**时间复杂度O(n^2)**

基本思想：第一次从arr[0]—arr[n-1]中选取最小值，与arr[0]交换，第二次从arr[1]—arr[n-1]中选取最小的，与arr[1]交换，第三次从arr[2]—arr[n-1]中选取最小的，与arr[2]交换……

第i次从arr[i-1]—arr[n-1]中选取最小的交换，总共通过n-1次得到一个安排序码从小到大排列的有序序列。



**代码：**

*/\*\*  
 \* 选择排序  
 \*/***public class** SelectSort {  
 **public static void** main(String[] args) {  
 **int** arr[]={8,9,7,11,5};  
 *selectSort*(arr);  
 }  
 **public static void** selectSort(**int** arr[]) {  
 **for** (**int** i = 0; i < arr.**length** - 1; i++) {  
 **int** minIndex = i;  
 **int** min = arr[i];  
 **for** (**int** j = i+1; j <arr.**length** ; j++) {  
 **if** (min>arr[j]){  
 min = arr[j];  
 minIndex = j;  
 }  
 }  
 **if** (minIndex!=i){  
 arr[minIndex] = arr[i];  
 arr[i] = min;  
 }  
 }  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
}

#### 插入排序

**时间复杂度O(n^2)**

基本思想：把n个待排序的元素看成为一个有序列表和一个无序列表，开始时有序列表中只包含一个元素，无序列表中包含n-1个元素，排序过程中每次从无序列表中取出第一个元素，把它的排序码依次与有序表元素的排序码进行比较，将它插入到有序列表的适当位置，使之成为新的有序表。

**代码：**

*/\*\*  
 \* 插入排序  
 \*/***public class** InsertSort {  
 **public static void** main(String[] args) {  
 **int**[] arr = {10, 9, 5, 6, 4};  
 *insertSort*(arr);  
 }  
 **public static void** insertSort(**int** arr[]) {  
 **for** (**int** i = 1; i < arr.**length**; i++) {  
 *//存储要交换的当前值* **int** insertValue = arr[i];  
 *//要进行比较的前一位索引* **int** insertIndex = i - 1;  
 **while** (insertIndex >= 0 && insertValue < arr[insertIndex]) {  
 arr[insertIndex+1] = arr[insertIndex];  
 insertIndex--;  
 }  
 *//退出while循环时即找到插入的位置* arr[insertIndex+1] = insertValue;  
 }  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
}

#### 希尔排序

**时间复杂度O(n^（1.3—2）)**

基本思想：希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键字越来越多，当增量减至1时，整个文件恰被分成一组，算法终止。

也就是将一个数组按照无限分组（每次/2的方式）直到数组长度>1结束；然后再每组中使用插入排序的思想进行对每组数据排序

**代码**

*/\*\*  
 \* 希尔排序  
 \*/***public class** ShellSort {  
 **public static void** main(String[] args) {  
  
 **int** arr[] = {1, 8, 2, 6, 7, 5, 4};  
 *shellSort1(arr);  
 shellSort2*(arr);  
 }  
  
 *//采用交换法* **public static void** shellSort1(**int**[] arr) {  
 **int** temp = 0;  
 **for** (**int** group = arr.**length** / 2; group > 0; group /= 2) {  
 **for** (**int** i = group; i < arr.**length**; i++) {  
 **for** (**int** j = i - group; j >= 0; j -= group) {  
 **if** (arr[j] > arr[j + group]) {  
 temp = arr[j];  
 arr[j] = arr[j + group];  
 arr[j + group] = temp;  
 }  
 }  
 }  
 }  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
  
 *//移位法* **public static void** shellSort2(**int**[] arr) {  
 **for** (**int** group = arr.**length** / 2; group > 0; group /= 2) {  
 **for** (**int** i = group; i < arr.**length**; i++) {  
 **int** j = i;  
 **int** temp = arr[j];  
 **if** (arr[j] < arr[j - group]) {  
 **while** (j - group >= 0 && temp < arr[j - group]) {  
 arr[j] = arr[j - group];  
 j -= group;  
 }  
 arr[j] = temp;  
 }  
 }  
 }  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
}

#### 快速排序

**时间复杂度O(nlog2n)**

基本思想：快速排序是对冒泡排序的一种改进。通过一趟排序将要排序的数据分割成独立的两部分。其中一部分的所有数据都比另一部分的所有数据都要小，然后再按照此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

**代码**

*/\*\*  
 \* 快速排序  
 \*/***public class** QuicklySort {  
 **public static void** main(String[] args) {  
 **int** arr[] = {-9, 11, 5, 0, -8, -9,0,11,5,99,-9, 0};  
 *quicklySort*(arr, 0, arr.**length** - 1);  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
  
 **public static void** quicklySort(**int**[] arr, **int** left, **int** right) {  
 **int** l = left;  
 **int** r = right;  
 **int** mid = arr[(r + l) / 2];  
 **int** temp = 0;  
 **while** (l < r) {  
 **while** (arr[l] < mid) {  
 l++;  
 }  
 **while** (arr[r] > mid) {  
 r--;  
 }  
 temp = arr[l];  
 arr[l] = arr[r];  
 arr[r] = temp;

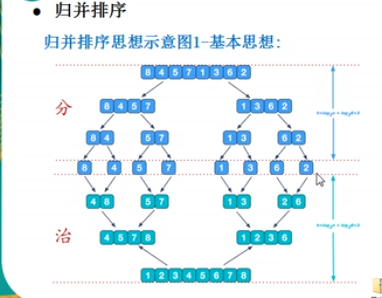
*//说明左边值全部小于等于中间值 右边值已经大于等于中间值*  
 **if** (l>=r){  
 **break**;  
 }

//  
 **if** (arr[l] == mid) {  
 r--;  
 }  
 **if** (arr[r] == mid) {  
 l++;  
 }  
 }  
 *//如果l==r 必须r-- l++否则为栈溢出* **if** (l == r) {  
 r--;  
 l++;  
 }  
 *//左递归* **if** (left < r) {  
 *quicklySort*(arr, left, r);  
 }  
 *//右递归* **if** (right > l) {  
 *quicklySort*(arr, l, right);  
 }  
  
 }  
}

#### 归并排序

**时间复杂度O(nlog2n)**

归并操作的工作原理如下（利用分治思想）：



第一步：申请空间，使其大小为两个已经[排序](https://baike.baidu.com/item/%E6%8E%92%E5%BA%8F)序列之和，该空间用来存放合并后的序列

第二步：设定两个[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "_blank)，最初位置分别为两个已经排序序列的起始位置

第三步：比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置

重复步骤3直到某一指针超出序列尾

将另一序列剩下的所有元素直接复制到合并序列尾

**代码：**

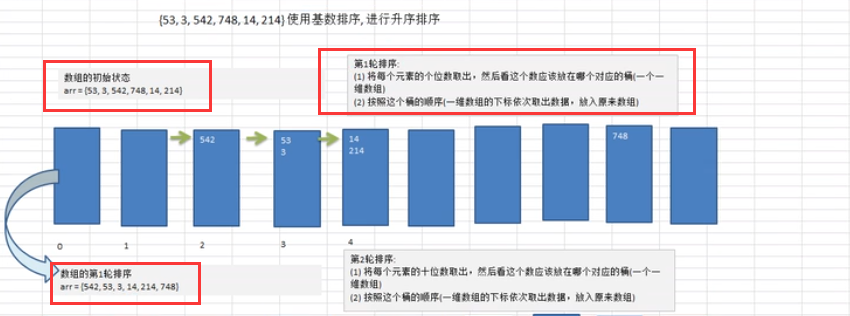
*/\*\*  
 \* 归并排序  
 \*/***public class** MergeSort {  
 **public static void** main(String[] args) {  
 **int** arr[] = {8,4,5,7,1,3,6,2,8,2};  
 **int** tempArr[] = **new int**[arr.**length**];  
 *mergeSort*(arr,0,arr.**length**-1, tempArr);  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
  
 */\*\*  
 \** ***@param arr*** *原始数组  
 \** ***@param left*** *最左边索引  
 \** ***@param right*** *最右边索引  
 \** ***@param tempArr*** *临时数组  
 \*/* **public static void** mergeSort(**int**[] arr, **int** left, **int** right, **int**[] tempArr) {  
 *//中间值 用来区分左右两边的值(左边的小于中间值 右边的大于中间值)* **if** (left < right) {  
 **int** mid = (left + right) / 2;  
 *//左递归分解  
 mergeSort*(arr,left,mid,tempArr);  
 *//右递归分解  
 mergeSort*(arr,mid+1,right, tempArr);  
 *//合并  
 merge*(arr, left,mid, right, tempArr);  
 }  
 }  
 *//合并方法* **public static void** merge(**int**[] arr, **int** left, **int** mid, **int** right, **int**[] tempArr) {  
 **int** t = 0;  
 **int** i = left;  
 **int** j = mid + 1;  
 **while** (i <= mid && j<=right) {  
 **if** (arr[i] <= arr[j]) {  
 tempArr[t] = arr[i];  
 t++;  
 i++;  
 } **else** {  
 tempArr[t] = arr[j];  
 j++;  
 t++;  
 }  
 }  
 *//将左边剩余的值填充到临时数组* **while** (i <= mid) {  
 tempArr[t] = arr[i];  
 i++;  
 t++;  
 }  
 *//将右边剩余的值填充到临时数组* **while** (j <= right) {  
 tempArr[t] = arr[j];  
 j++;  
 t++;  
 }  
 t = 0;  
 **int** tempIndex=left;  
 **while** (tempIndex <= right) {  
 arr[tempIndex] = tempArr[t];  
 t++;  
 tempIndex++;  
 }  
 }  
}

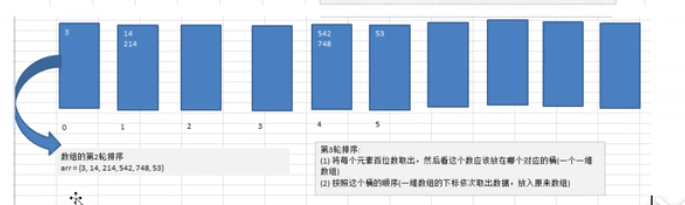
#### 基数排序

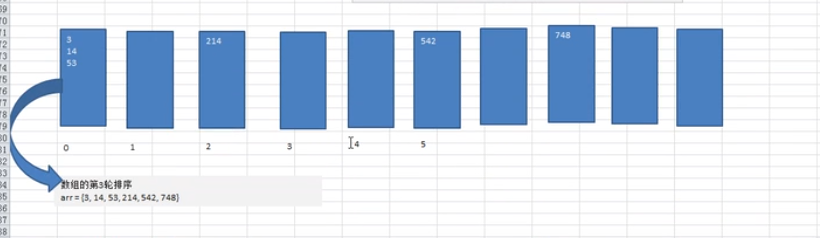
**时间复杂度O(n)**

基数排序介绍：基数排序（radix sort）属于“分配式排序”，又称为“桶子法”或者bin sort属于稳定性排序

基数排序思想：将所有待排序得数值统一为同样的数位长度，数位较短的数前面0.然后，从最低位开始，依次进行一次排序，这样从最低位排序一直到最高位排序完成以后，数列就变成了一个有序序列。







**代码：**

*/\*\*  
 \* 基数排序  
 \*/***public class** RadixSort {  
 **public static void** main(String[] args) {  
 **int** arr[] = {53, 3, 542, 748, 14, 214};  
 *radixSort*(arr);  
 }  
  
 **public static void** radixSort(**int**[] arr) {  
 *//遍历原始数组 找到最大的数 然后获取到其长度 进行遍历长度-1次* **int** max = arr[0];  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 **if** (arr[i] > max)  
 max = arr[i];  
 }  
 **int** maxLength = (max+**""**).length();  
 */\*定义一个二位数组 代表10个桶  
 因为无法知道具体每个桶中存放的元素个数  
 因此定义长度为arr.length采用空间换时间  
 \*/* **int**[][] bucket = **new int**[10][arr.**length**];  
 *//标记每个桶中的元素个数* **int**[] bucketElement = **new int**[10];  
 **for** (**int** m = 0 ,n=1; m <maxLength ; m++,n\*=10) {  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 *//取出每一个元素的个位* **int** digit = arr[i] /n % 10;  
 bucket[digit][bucketElement[digit]] = arr[i];  
 bucketElement[digit]++;  
 }  
 *//按照这个桶的顺序 将数据再次放入原始数组* **int** index = 0;  
 **for** (**int** j = 0; j < bucket.**length**; j++) {  
 **if** (bucketElement[j] != 0) {  
 **for** (**int** l = 0; l < bucketElement[j]; l++) {  
 arr[index] = bucket[j][l];  
 index++;  
 }  
 }  
 *//重新置初值方便后面重新计算* bucketElement[j] = 0;  
 }  
 }  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
}

#### 排序算法的各方面比较



### 查找算法

#### 顺序查找

**代码：**

*/\*\*  
 \* 顺序查找  
 \*/***public class** SequenceSearch {  
 **public static void** main(String[] args) {  
 **int** arr[] = {1,2,3,4,5,6,7};  
 System.***out***.println(*sequenceSearch*(arr,4));  
 }  
 **public static int** sequenceSearch(**int**[]arr,**int** n){  
 **for** (**int** i = 0; i < arr.**length**-1; i++) {  
 **if** (arr[i]==n){  
 **return** i;  
 }  
 }  
 **return** -1;  
 }  
}

#### 二分查找

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。

思路分析:

首先确定该数组的中间下标 mid=(left+right) /2

然后让需要查找的数字和arr[mid]比较

如果大于中间值 则查找的数字在右边

如果小于中间值 则查找的数字在左边

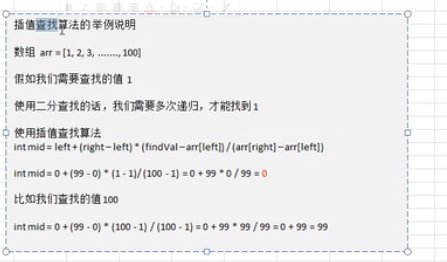
如果相等 则找到查询的数字 返回下标即可

**代码：**

*/\*\*  
 \* 二分查找  
 \*/***public class** BinarySearch {  
 **public static void** main(String[] args) {  
 **int** arr[] = {1, 5, 6, 7, 8,9,9,9,10};  
 System.***out***.println(*binarySearch*(arr, 0, arr.**length** - 1, 9));  
 List<Integer> list = *binarySearch2*(arr, 0, arr.**length** - 1, 9);  
 System.***out***.println(list.toString());  
 }  
  
 *//无法查询相同元素的下标* **public static int** binarySearch(**int**[] arr, **int** left, **int** right, **int** searchNumber) {  
 **int** mid = (left + right) / 2;  
 **int** midValue = arr[mid];  
 *//递归推出条件 当left>right 说明没有找到* **if** (left > right) {  
 **return** -1;  
 }  
 **if** (midValue > searchNumber) {  
 **return** *binarySearch*(arr, left, mid - 1, searchNumber);  
 } **else if** (midValue < searchNumber) {  
 **return** *binarySearch*(arr, mid + 1, right, searchNumber);  
 } **else** {  
 **return** mid;  
 }  
 }  
  
 *//返回所有相同元素的下标* **public static** List<Integer> binarySearch2(**int**[] arr, **int** left, **int** right, **int** searchNumber) {  
 **int** mid = (left + right) / 2;  
 **int** midValue = arr[mid];  
 *//递归推出条件 当left>right 说明没有找到* **if** (left > right) {  
 **return new** ArrayList<>();  
 }  
 **if** (midValue > searchNumber) {  
 **return** *binarySearch2*(arr, left, mid - 1, searchNumber);  
 } **else if** (midValue < searchNumber) {  
 **return** *binarySearch2*(arr, mid + 1, right, searchNumber);  
 } **else** {  
 */\*  
 当查询到mid时 不立刻返回  
 在mid左右两边进行扫描 看是否有相同的元素  
 \*/* **int** temp = mid - 1;*//左扫描* List<Integer> list = **new** ArrayList<>();  
 **while** (**true**) {  
 **if** (temp < 0 || arr[temp] != searchNumber) {  
 **break**;  
 }  
 list.add(temp);  
 temp -= 1;  
 }  
 list.add(mid); *//加入中间值* temp = mid + 1; *//右扫描* **while** (**true**) {  
 **if** (temp > arr.**length** - 1 || arr[temp] != searchNumber) {  
 **break**;  
 }  
 list.add(temp);  
 temp += 1;  
 }  
 **return** list;  
 }  
 }  
}

#### 插值查找

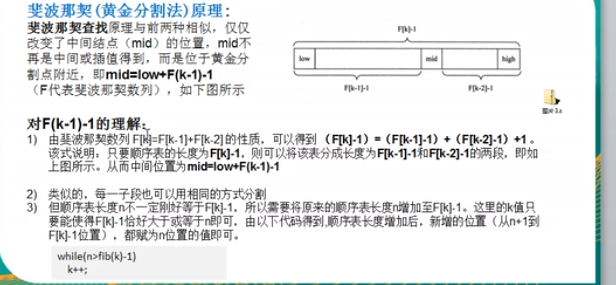




**代码：**

*/\*\*  
 \* 插值查找  
 \*/***public class** InsertValueSearch {  
 **public static void** main(String[] args) {  
 **int**[] arr = **new int**[100];  
 **for** (**int** i = 0; i < 100; i++) {  
 arr[i] = i;  
 }  
 System.***out***.println(*insertValueSearch*(arr, 0, arr.**length** - 1, 80));  
 }  
  
 **public static int** insertValueSearch(**int**[] arr, **int** left, **int** right, **int** findValue) {  
 **if** (left > right || arr[0] > findValue || arr[arr.**length** - 1] < findValue) {  
 **return** -1;  
 }  
 *//插值查找的索引计算公式* **int** mid = left + (right - left) \* (findValue - arr[left]) / (arr[right] - arr[left]);  
 **if** (arr[mid] == findValue) {  
 **return** mid;  
 } **else if** (arr[mid] < findValue) {  
 *//向右递归* **return** *insertValueSearch*(arr, mid + 1, right, findValue);  
 } **else if** (arr[mid] > findValue) {  
 *//左递归* **return** *insertValueSearch*(arr, left, mid - 1, findValue);  
 } **else** {  
 **return** -1;  
 }  
 }  
}

#### 斐波那契（黄金分割法）查找



**代码**

*/\*\*  
 \* 斐波那契查找  
 \*/***public class** FibonacciSearch {  
 **private final static int *maxSize*** = 20;  
 **public static void** main(String[] args) {  
 **int** arr[]={1,2,3,4,5,6,7};  
 System.***out***.println(*feibonacciSearch*(arr, 7));  
 }  
  
 *//先获取到一个斐波那契数列作为查询基准* **public static int**[] feibonacci() {  
 **int**[] feibo = **new int**[***maxSize***];  
 feibo[0] = 1;  
 feibo[1] = 1;  
 **for** (**int** i = 2; i < ***maxSize***; i++) {  
 feibo[i] = feibo[i - 1] + feibo[i - 2];  
 }  
 **return** feibo;  
 }  
  
 *//进行斐波那契查询* **public static int** feibonacciSearch(**int**[] arr, **int** findValue) {  
 **int** low = 0;  
 **int** high = arr.**length** - 1;  
 **int** mid = 0;  
 **int** k = 0; *//记录斐波那契分割点下标* **int** F[] = *feibonacci*();  
 */\*\*  
 \* 例如我们数组长度为6 但是在斐波那契数列中只有F[5]=8-1=7才满足  
 \* 因此我们需要将k的值加到5 也就是斐波那契中F[5]的位置  
 \*/* **while** (high > F[k] - 1) {  
 k++;  
 }  
 *//拷贝元素数组到以斐波那契分割点为长度的数组中* **int** temp[] = Arrays.*copyOf*(arr, F[k]);  
 */\*找到的长度并不一定大小刚好等于原始数组  
 因此空出来的位置利用原始数组最后一个数填充  
 例如：temp={1,2,3,4,5,0,0} =>temp={1,2,3,4,5,5,5}  
 \*/* **for** (**int** i = high + 1; i < temp.**length**; i++) {  
 temp[i] = arr[high];  
 }  
 **while** (low<=high){  
 mid = low+(F[k-1]-1);  
 **if** (findValue<temp[mid]){  
 *//向左查询* high = mid-1;  
 */\*  
 k--的原因：  
 全部元素 = 前半部分 + 后半部分  
 F[k] = F[K-1] + F[K-2]  
 即在F[k-1]还可以继续拆分 F[k-1]=F[K-2]+F[K-3]  
 与前面相差k-- 下次循环mid=F[k-1-1]-1  
 \*/* k--;  
 }**else if** (findValue>temp[mid]){  
 low=mid+1;  
 */\*  
 k-2的原因：  
 全部元素 = 前半部分 + 后半部分  
 F[k] = F[K-1] + F[K-2]  
 即在F[k-2]还可以继续拆分 F[k-2]=F[K-3]+F[K-4]  
 与前面相差k-2 下次循环mid=F[k-1-2]-1  
 \*/* k-=2;  
 }**else** {  
 *//找到* **if** (mid<=high){  
 **return** mid;  
 }**else  
 return** high;  
 }  
 }  
 **return** -1;  
 }  
}

### 哈希表

