第六章 二分查找与二叉排序树

文章目录

二分查找与二叉排序树

二分查找基础知识 && 二叉查找（排序）树基础知识

（1）二分查找

二分查找又称折半查找，首先假设表中元素是按升序排列，将表中间位置的关键字与查找关键字比较：

1. 如果两者相等，则查找成功；
2. 否则利用中间位置将表分成前、后两个子表：
   （1）如果中间位置的关键字大于查找关键字，则进一步查找前一子表
   （2）否则进一步查找后一子表
   重复以上过程，直到找到满足条件的记录，即查找成功，或直到子表不存在为止，此时查找不成功。

```
    // 二分查找（递归）
    public boolean binary_search(int[] sort_array, int begin, int end, int target) {
        // 搜索target到返回true,否则返回false
        // begin和end为待搜索区间左端，右端
        if(begin>end)return false;  // 区间不存在
        int mid = (begin+end)/2;    //找中点
        if(target == sort_array[mid])return true;   //找到target
        else if(target < sort_array[mid])
            return binary_search(sort_array,begin,mid-1,target);
        else
            return binary_search(sort_array,mid+1,end,target);
    }

    // 二分查找（循环）
    public boolean binary_search2(int[] sort_array, int target) {
        int begin = 0;
        int end = sort_array.length-1;
        while(begin<=end) {
            int mid = (begin+end)/2;
            if(target == sort_array[mid])return true;
            else if(target<sort_array[mid]) {
                end = mid - 1;
            }else if(target>sort_array[mid]) {
                begin = mid + 1;
            }
        }
        return false;
    }
```

（2）二叉查找树

二叉查找树，它是一棵具有下列性质的二叉树：

1. 若左子树不空，则左子树上所有结点的值均小于或等于它的根节点的值；
2. 若右子树不空，则左子树上所有结点的值均大于或等于它的根节点的值；
3. 左、右子树也分别为二叉排序树；
4. 等于的情况只能出现在左子树或右子树的某一侧；

```
    class TreeNode{
        int value;
        TreeNode left;
        TreeNode right;
        TreeNode(int value){
            this.value = value;
            left = null;
            right = null;
        }
    }
```

二叉查找树插入结点

将某结点(insert_node)插入至以node为根的二叉查找树中：
如果insert_node节点值小于当前node节点值：
如果node有左子树，则递归的将该结点插入至左子树为根二叉排序树中
否则，将node->left赋值为该结点地址

否则（如果insert_node节点值大于等于当前node节点值：）
如果node有左子树，则递归的将该结点插入至左子树为根二叉排序树中
否则，将node->left赋值为该结点地址

```
    // 二叉查找树插入数值
    void BST_insert(TreeNode node, TreeNode insert_node) {
        if(insert_node.value < node.value) {
            if(node.left != null)
                BST_insert(node.left,insert_node); // 左子树不为空，递归将insert_node插入左子树
            else
                node.left = insert_node;    // 左子树为空时，将node左指针与待插入结点相连
        }
        else {
            if(node.right!=null)
                BST_insert(node.right,insert_node); // 右子树不为空，递归将insert_node插入右子树
            else
                node.right = insert_node;// 右子树为空时，将node右指针与待插入结点相连
        }
    }
```

二叉查找树查找数值
查找数值value是否在二叉查找树中出现：
如果value等于当前查看node的节点值，返回真
如果value节点值小于当前node节点值：
如果当前结点有左子树，继续在左子树中查找该值；否则，返回假
否则（如果value节点值大于当前node节点值：）
如果当前结点有右子树，继续在右子树中查找该值；否则，返回假

```
    // 二叉查找树查找数值
    boolean BST_search(TreeNode node,int value) {
        if(node.value == value)return true; // 当前结点就是value，返回真
        if(node.value > value) {
            // 当前node结点值大于value
            if(node.left != null)return BST_search(node.left,value);    // node结点有左子树，继续搜索左子树
            else return false;
        }else {
            if(node.right != null)return BST_search(node.right,value); // node结点有右子树，继续搜索右子树
            else return false;
        }
    }
```

leetCode

例1：插入位置（35）

题目描述
给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。
你可以假设数组中无重复元素。

```
示例 1:

输入: [1,3,5,6], 5
输出: 2
示例 2:

输入: [1,3,5,6], 2
输出: 1
示例 3:

输入: [1,3,5,6], 7
输出: 4
示例 4:

输入: [1,3,5,6], 0
输出: 0
```

算法思路
设元素所在位置（或最终需要插入位置）为index，
在二分查找的过程中：
如果target==nums[mid]：index = mid;
如果target<nums[mid]，且（mid == 0 或 target>nums[mid-1]）：index = mid;
如果target>nums[mid]，且（mid == nums.size()-1 或 target<nums[mid+1]）：index = mid+1;
程序代码

```
    // 35. 搜索插入位置
    // 给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。
    // 如果目标值不存在于数组中，返回它将会被按顺序插入的位置。
    // 你可以假设数组中无重复元素。
    public int searchInsert(int[] nums, int target) {
        // 二分查找（循环）
            int index = -1; //  最终返回的下标，若找到则返回该元素下标，否则为需要插入的位置
            int begin = 0;  // 搜索区间左端点
        int end = nums.length-1;        // 搜索区间右端点
        int mid = (begin+end)/2;
        while(index == -1) {    //若index == -1,说明未找到正确位置，持续二分搜索
            mid = (begin+end)/2;
            if(target == nums[mid]) index = mid;    //  找到元素，下标为该元素下标
            else if(target<nums[mid]) { // 目标值小于当前值
                if(mid == 0 || target>nums[mid-1]) {index = mid;break;} // 结束条件：小于第一个元素 || 位于mid元素之前
                end = mid - 1;  // 继续二分查找
            }else if(target>nums[mid]) {
                if(mid == nums.length-1 || target<nums[mid+1]) {index = mid+1;break;}       // 结束条件：大于最后一个元素 || 位于mid元素之后
                begin = mid + 1;        // 继续二分查找
            }
        }

```

```
        return index;
    }
```

例2：区间查找（34）

题目描述

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。
你的算法时间复杂度必须是 O(log n) 级别。
如果数组中不存在目标值，返回 [-1, -1]。

```
示例 1:

输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]
示例 2:

输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]
```

算法思路

1. 二分查找查找目标值
2. 查找到目标值所在位置后，向前和向后遍历。找到目标值所在区间的起始位置和结束位置

程序代码

```
// 34. 在排序数组中查找元素的第一个和最后一个位置
// 给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。
public int[] searchRange(int[] nums, int target) {
    int begin = 0;
    int end = nums.length-1;
    int startIdx = -1;  // 目标值在数组中开始位置
    int endIdx = -1;        // 目标值在数组中结束位置
    int[] result = {startIdx,endIdx};
    while(begin<=end) {
        int mid = (begin+end)/2;
        if(target == nums[mid]) {
            // 查找到目标值所在位置后，向前和向后遍历。找到目标值所在区间的起始位置和结束位置。
            startIdx = mid;
            endIdx = mid;
            while(startIdx-1 >= begin && nums[startIdx-1] == target)startIdx--;
            while(endIdx+1 <= end && nums[endIdx+1] == target)endIdx++;
            result[0] = startIdx;
            result[1] = endIdx;
            return result;
        }
        // 采用二分查找查找目标值的位置
        else if(target<nums[mid]) {
            end = mid - 1;
        }else if(target>nums[mid]) {
            begin = mid + 1;
        }
    }
    return result;
}
```

例3：旋转数组查找（33）

题目描述

假设按照升序排序的数组在预先未知的某个点上进行了旋转。
(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。
搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。
你可以假设数组中不存在重复的元素。
你的算法时间复杂度必须是 O(log n) 级别。

```
示例 1:

输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4
示例 2:

输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

算法思路

- **当目标target < 中点nums[mid]：**
  如果nums[begin] < nums[mid]:（也即nums[begin]>nums[end]）
  (说明递增区间[begin,mid-1]，旋转区间为[mid+1,end])
  如[7,9,12,15,20,1,3,6]
  如果target(12) >= nums[begin]:
  在递增区间[begin,mid-1]中查找
  否则（1）：
  在旋转区间[mid+1,end]中查找
  如果nums[begin]>nums[mid]:
  (说明递增区间[mid+1,end]，旋转区间为[begin,mid-1])
  如[20,1,3,6,7,9,12,15]
  直接在旋转区间[begin,mid-1]查找
  例如3（不可能在比6大的递增区间）
  如果nums[begin] == nums[mid]：
  （说明目标只可能在[mid+1，end]间）
  如target = 1,数组[6,1]

- **当目标target > 中点nums[mid]：**
  如果nums[begin] < nums[mid]:
  (说明递增区间[begin,mid-1]，旋转区间为[mid+1,end])
  如[7,9,12,15,20,1,3,6]
  直接在旋转区间[mid+1,end]查找
  例如查找target(20)
  如果nums[begin]>nums[mid]:（也即nums[begin]>nums[end]）
  (说明递增区间[mid+1,end]，旋转区间为[begin,mid-1])
  如[15,20,1,3,6,7,9,12]
  如果target>= nums[begin]:
  在递增区间[begin,mid-1]中查找（如查找20）
  否则（1）：
  在旋转区间[mid+1,end]中查找（如查找9）
  如果nums[begin] == nums[mid]：
  （说明目标只可能在[mid+1，end]间）
  如target = 7,数组[6,7]

程序代码

```
// 33. 搜索旋转排序数组
// 假设按照升序排序的数组在预先未知的某个点上进行了旋转。
// ( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。
// 搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。
public int search(int[] nums, int target) {
        // 针对旋转数组的二分查找算法（循环）
    int index = -1; //  最终返回的下标，若找到则返回该元素下标，否则返回-1
    int begin = 0;  // 搜索区间左端点
    int end = nums.length-1;        // 搜索区间右端点
    int mid = (begin+end)/2;
    while(begin <= end && index == -1) {    //若index == -1,说明未找到正确位置，持续二分搜索
        mid = (begin+end)/2;
        if(target == nums[mid]) index = mid;    //  找到元素，下标为该元素下标

        else if(target<nums[mid]) { // 目标值小于当前值
            if(nums[begin]<nums[mid]) { // [begin,mid-1]为递增序列，[mid+1,end]为旋转序列
                if(target >= nums[begin]) end = mid-1;  // 目标值在[begin,mid-1]
                else begin = mid+1; // 目标值在[mid+1,end]
            }else if(nums[begin]>nums[mid]){    // [begin,mid-1]为旋转序列，[mid+1,end]为递增序列
                end = mid-1;        // 因为target<nums[mid],所以不可能大于递增序列[mid+1,end]中任意值
            }else if(nums[begin] == nums[mid]) {
                begin = mid+1;  // target只可能在[mid+1,end]
            }
        }else if(target>nums[mid]) { // 目标值大于当前值
            if(nums[end]<nums[mid]) {   // [begin,mid-1]为递增序列，[mid+1,end]为旋转序列
                begin = mid+1;  // 目标值在[mid+1,end],target>mid,所以target一定大于[begin,mid]
            }else if(nums[end]>nums[mid]){  // [begin,mid-1]为旋转序列，[mid+1,end]为递增序列
                if(target <= nums[end])begin = mid+1;   // 目标值在[mid+1,end]
                else end = mid-1;   // 目标值在[begin,mid-1]
            }else if(nums[end] == nums[mid]) {
                end = mid-1;    // target只可能在[mid+1,end]
            }
        }
    }
    return index;
}
```

例4：二叉查找树编码与解码（449）

题目描述

序列化是将数据结构或对象转换为一系列位的过程，以便它可以存储在文件或内存缓冲区中，或通过网络连接链路传输，以便稍后在同一个或另一个计算机环境中重建。
设计一个算法来序列化和反序列化二叉搜索树。 对序列化/反序列化算法的工作方式没有限制。 您只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。
编码的字符串应尽可能紧凑。
注意：不要使用类成员/全局/静态变量来存储状态。 你的序列化和反序列化算法应该是无状态的。

算法思路

1. 先序遍历，将二叉树转换为字符串
2. 将遍历结果按照顺序重新构造为二叉排序树树

程序代码

```
// 449.序列化和反序列化二叉搜索树
 // 设计一个算法来序列化和反序列化二叉搜索树。 对序列化/反序列化算法的工作方式没有限制。
 // 您只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。
 public String serialize(TreeNode root) {
     // 先序遍历，将二叉树转换为字符串
    StringBuilder sb = new StringBuilder();
    preOrder(root,sb);
    return sb.toString();
 }

  public TreeNode deserialize(String data) {
      // 将遍历结果按照顺序重新构造为二叉排序树树
         if(data == null || data.equals(""))return null;
     String[] values = data.split(",");
     TreeNode root = new TreeNode(Integer.parseInt(values[0]));
     for(int i=1;i<values.length;i++) {
             TreeNode node = new TreeNode(Integer.parseInt(values[i]));
             BST_insert(root,node);
     }
         return root;
  }

 public static class TreeNode{
     int value;
     TreeNode left;
     TreeNode right;
```

```
    public TreeNode(int value){
        this.value = value;
        left = null;
        right = null;
    }
}

 public void preOrder(TreeNode node, StringBuilder sb) {
        if(node==null)return;

        sb.append(node.value+","); // 字符之间用分隔符逗号分隔开
        preOrder(node.left,sb); // 遍历左子树
        preOrder(node.right,sb); // 遍历右子树
 }

// 二叉查找树插入数值
void BST_insert(TreeNode node, TreeNode insert_node) {
    if(insert_node.value < node.value) {
        if(node.left != null)
            BST_insert(node.left,insert_node); // 左子树不为空，递归将insert_node插入左子树
        else
            node.left = insert_node;    // 左子树为空时，将node左指针与待插入结点相连
    }
    else {
        if(node.right!=null)
            BST_insert(node.right,insert_node); // 右子树不为空，递归将insert_node插入右子树
        else
            node.right = insert_node;// 右子树为空时，将node右指针与待插入结点相连
    }
}
```

例5：逆序数（315）

题目描述

给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质： counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

示例:

```
输入: [5,2,6,1]
输出: [2,1,1,0]
解释:
5 的右侧有 2 个更小的元素 (2 和 1).
2 的右侧仅有 1 个更小的元素 (1).
6 的右侧有 1 个更小的元素 (1).
1 的右侧有 0 个更小的元素.
```

算法思路

将元素按照原数组逆置后的顺序插入到二叉查找树中，在插入时候，计算有多少个元素比当前元素小，算法如下：

1. 设置变量count_small = 0,记录在插入过程中，有多少个元素比插入节点值小；
2. 若待插入节点值小于等于当前结点node值，node->count++，递归将该结点插入到当前节点的左子树；
3. 若待插入结点大于当前结点node值，count_mall += node->count+1（当前结点左子树+1）；递归将该结点插入到当前结点右子树

程序代码

```
    Integer count = 0; // 全局变量count（解决： Integer 传值不传引用）
    public List<Integer> countSmaller(int[] nums) {
       // 将元素按照原数组逆置后的顺序插入到二叉查找树中
        // 若该元素插入在右子树上，则加上左子树上所有结点
        List<Integer> result = new ArrayList<Integer>();
        if(nums == null || nums.length == 0)return result; // 数组为空
        // 1. 将原数组逆序排序
        reverse(nums);
        // 2. 将逆序数组插入到二叉查找树
        BinaryTreeNode root = new BinaryTreeNode(nums[0]); //根节点单独处理
        result.add(0);
        for(int i=1;i<nums.length;i++) {
            count = 0; // count 为插入结点右侧小于该结点值的元素的数量
            CS_insert(root,nums[i]);    // 将数组中的值逐一插入二叉排序树
            result.add(count);
        }
        // 倒序
        for(int i=0;i<result.size()/2;i++) {
            Integer temp = result.get(result.size()-i-1);
            result.set(result.size()-i-1, result.get(i));
            result.set(i, temp);
        }

        return result;
    }

    public void reverse(int[] nums) {  // 逆转函数
        for(int i=0;i<nums.length/2;i++) {
            int temp = nums[nums.length-i-1];
            nums[nums.length-i-1] = nums[i];
            nums[i] = temp;
        }
    }

    public class BinaryTreeNode{
        int value;
        int count;  // 可重复结点的数量
        BinaryTreeNode left;
        BinaryTreeNode right;

        public void addCount() {
            this.count++;
        }

```

```
        public BinaryTreeNode(int value) {
            this.value = value;
            this.count = 1;
            left = null;
            right = null;
        }
    }

    public void CS_insert(BinaryTreeNode node,Integer value) {
        if(value == node.value) {
            countNode(node.left);   // 若插入结点与当前结点相同，计算当前结点左子树的值
            node.addCount();            // 将当前结点的数目++
        }
        else if(value < node.value) {   // 当前值较小，插入左子树
        if(node.left != null)
            CS_insert(node.left,value); // 左子树不为空，递归将insert_node插入左子树
        else
            node.left = new BinaryTreeNode(value); // 左子树为空时，将node左指针与待插入结点相连
       }
       else {
            count+=node.count;  // 插入右子树时，元素数目为当前结点的数目+当前结点左子树中元素数目
            countNode(node.left);
            if(node.right!=null)
                CS_insert(node.right,value); // 右子树不为空，递归将insert_node插入右子树
            else
                node.right = new BinaryTreeNode(value);// 右子树为空时，将node右指针与待插入结点相连
       }
    }

    public void countNode(BinaryTreeNode node){
            // 计算子树中所有结点数目（结点可重复）
            if(node != null) {
                count += node.count;

            if(node.left!=null)countNode(node.left);
            if(node.right!=null)countNode(node.right);
            }
    }
```

剑指offer

例1：旋转数组的最小数字（6）

题目描述

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。
输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。
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。
NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

程序代码

```
    public int minNumberInRotateArray(int [] array) {
            if(array == null || array.length == 0)return 0;
        return getMinFromArray(array,0,array.length-1);
    }

    public int getMinFromArray(int[] array,int start,int end) {
            //使用二分查找从旋转数组 array[start..end]中选取最小值
            if(start == end) return array[start];
            int mid = (start + end)/2;
            int left_min = 0;   // 数组左端最小值
            int right_min = 0;  // 数组右端最小值
            // 获取数组左端最小值
            if(array[start]<=array[mid])left_min = array[start]; // array[start..mid]为递增序列，返回最小值array[start]
                else left_min = getMinFromArray(array,start,mid);   // array[start..mid]为旋转数组
            // 获取数组右端最小值
            if(array[mid+1]<=array[end])right_min = array[mid+1]; // array[mid+1..end]为递增序列，返回最小值array[mid+1]
                else right_min = getMinFromArray(array,mid+1,end);  // array[mid+1..end]为旋转数组

            return Integer.min(left_min, right_min);
    }
```

例2：二叉搜索树的后序遍历序列（23）

题目描述

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。

程序代码

```
public boolean VerifySquenceOfBST(int [] sequence) {
        /*基本知识储备：
         * 二叉搜索树：左子树<根<右子树+后续遍历：左右根
         * 思想：BST的后序序列的合法序列是，对于一个序列S，最后一个元素是x （也就是根），如果去掉最后一个元素的序列为T，那么T满足：
         * T可以分成两段，前一段（左子树）小于x，后一段（右子树）大于x，且这两段（子树）都是合法的后序序列。
        */
            if(sequence.length==0)return false;
            return judge(sequence,0,sequence.length-1);
    }

    public boolean judge(int[] sequence,int start,int end) {
        // 判断sequence[start..end]是否为后序遍历序列
        // 只要有一个子树不满足后序遍历条件，即非后序遍历序列
        if(start >= end)return true;

        int key = sequence[end];        // 二叉搜索树根节点
        int i = end - 1;                // 二叉搜索树的遍历指针
        int mid = end-1;                // 二叉搜索树左右子树分隔节点
        for(;i>=0;i--) if(sequence[i]<key) {mid=i;break;}
        for(;i>=0;i--) if(sequence[i]>=key) return false;   // 若左子树值大于根节点，则不是二叉搜索树

        if(!judge(sequence,start,mid))return false;     // 若左子树不满足二叉搜索树，则非二叉搜索树
```

```
        if(!judge(sequence,mid+1,end-1))return false;       // 若右子树不满足二叉搜索树，则非二叉搜索树

        return true;
}
```

例3：数字在排序数组中出现的次数（36）

题目描述

统计一个数字在排序数组中出现的次数。

程序代码

```
    // 36. 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
    public int GetNumberOfK(int [] array , int k) {
        // 1.采用二分查找找到排序数组中数字k的位置
            // 2.向前向后数k出现的次数
            int position = findKInArray(array,0,array.length-1,k);
            if(position != -1)return countNumberOfK(array,k,position);
            return 0;
    }

    public int findKInArray(int[] array,int start,int end,int k) {
            // 二分查找k在数组array[start...end]中的位置
            // 若不存在，则返回-1
            if(start > end)return -1;
            int mid = (start + end)/2;
            if(array[mid] == k)return mid;
            else if(k < array[mid])return findKInArray(array,start,mid-1,k);
            else return findKInArray(array,mid+1,end,k);
    }

    public int countNumberOfK(int[] array,int k,int position) {
            // 求数字K在数组中出现的次数
            int count = 1;
            int idx = position - 1;
            while(idx>=0 && array[idx--]==k) count++;
                // 向前求数字K出现的次数
            idx = position + 1;
            while(idx< array.length && array[idx++]==k) count++;
            return count;
    }
```

例4：二叉搜索树第k个结点（61）

题目描述

给定一棵二叉搜索树，请找出其中的第k小的结点。例如，（5，3，7，2，4，6，8）中，按结点数值大小顺序第三小结点的值为4。

程序代码

```
    // 61. 二叉搜索树第k个节点
    // 给定一棵二叉搜索树，请找出其中的第k小的结点。
    // 例如，（5，3，7，2，4，6，8）中，按结点数值大小顺序第三小结点的值为4。
    ArrayList<TreeNode> sortList = new ArrayList<TreeNode>();
    Integer nodeNumber = 0;
    TreeNode KthNode(TreeNode pRoot, int k)
    {
        // 利用二叉搜索树的性质：
            // 中序遍历结果为从小到大的顺序序列
            constructSortList(pRoot);
            if(k<=0 || k>nodeNumber) return null;   // 不合法
            return sortList.get(k-1);
    }

    public void constructSortList(TreeNode node) {
            if(node!=null) {
                constructSortList(node.left);
                sortList.add(node);
                nodeNumber++;
                constructSortList(node.right);
            }
    }
```

2019 校招

例1：丰收（11）

题目描述

又到了丰收的季节，恰逢小易去牛牛的果园里游玩。

牛牛常说他对整个果园的每个地方都了如指掌，小易不太相信，所以他想考考牛牛。

在果园里有N堆苹果，每堆苹果的数量为ai，小易希望知道从左往右数第x个苹果是属于哪一堆的。

牛牛觉得这个问题太简单，所以希望你来替他回答。

输入描述:

第一行一个数n(1 <= n <= 105)。

第二行n个数ai(1 <= ai <= 1000)，表示从左往右数第i堆有多少苹果

第三行一个数m(1 <= m <= 105)，表示有m次询问。

第四行m个数qi，表示小易希望知道第qi个苹果属于哪一堆。

输出描述:

m行，第i行输出第qi个苹果属于哪一堆。

算法代码

```
    // 10. 丰收
    // 又到了丰收的季节，恰逢小易去牛牛的果园里游玩。
    // 牛牛常说他对整个果园的每个地方都了如指掌，小易不太相信，所以他想考考牛牛。
    // 在果园里有N堆苹果，每堆苹果的数量为ai，小易希望知道从左往右数第x个苹果是属于哪一堆的。
    // 牛牛觉得这个问题太简单，所以希望你来替他回答。
    public void findHeapOfApple(){
        // 1. 定义和数组 sum[] ，表示到从第1堆到第i+1堆中共有sum[i]个苹果
```

```java
        // 2. 则查找第 j 个苹果属于哪一堆时，
        // 对sum数组进行而二分查找（sum一定是按升序排序）
        // 找到第一个大于j的sum[i],则苹果属于第 i+1 堆
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();   // 有n堆苹果
        int[] sum = new int[n]; // 表示从第1堆到第i+1堆共有sum[i]个苹果
        for(int i=0;i<n;i++) {if(i==0)sum[0]=sc.nextInt();else sum[i] = sum[i-1]+sc.nextInt();}
        int m = sc.nextInt();   // 共有m个询问
        while(m-->0) {
            int q = sc.nextInt();
            System.out.println(binarySearch(sum,q,0,n)+1);
        }
    }

    public int binarySearch(int[] array,int target,int start,int end) {
        // 查找数组array，查找值target,起始start,终止end
        // 查找 大于等于target 的 最小值
        if(start == end)return end;
        int mid = (start + end)/2;
        if(target == array[mid])return mid;
        else if(target > array[mid])return binarySearch(array,target,mid+1,end);
        else return binarySearch(array,target,start,mid);
    }
```