1 排序

排序算法大体可分为两种：一种是**比较排序**，时间复杂度 O(nlogn) ~ O(n^2)，主要有：**冒泡排序**，**选择排序**，**插入排序**，**归并排序**，**堆排序**，**快速排序**等。另一种是**非比较排序**，时间复杂度可以达到 O(n)，主要有：**计数排序**，**基数排序**，**桶排序**等。

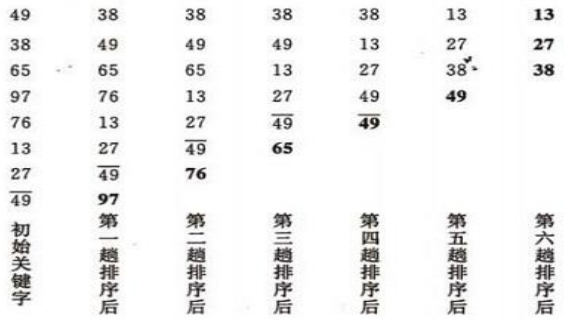
下表给出了常见比较排序算法的性能：

表 9-10-1

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log n) \sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n) \sim O(n)$	不稳定

1.1 冒泡排序

思想：**比较相邻的元素，如果前一个比后一个大，就把它们两个调换位置。**




```
public class BubbleSort {
    // 分类 -------------- 内部比较排序
    // 数据结构 ---------- 数组
    // 最差时间复杂度 ---- O(n^2)
    // 最优时间复杂度 ---- 如果能在内部循环第一次运行时,把最优时间复杂度降低到 O(n)
    // 平均时间复杂度 ---- O(n^2)
    // 所需辅助空间 ------ O(1)
    // 稳定性 ------------ 稳定
    // 排序前:      { 6, 5, 3, 1, 8, 7, 2, 4 }
    // 第一遍排序: { 5, 3, 1, 6, 7, 2, 4, 8 }  前后两个比较，大的放后面，最大的在最后
    // 第二遍排序: { 3, 1, 5, 6, 2, 4, 7, 8 }

    void Swap(int A[], int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
```

```
    }
    void BubbleSort(int A[], int n) {
       for (int j = 0; j < n - 1; j++) {      // 每次最大元素就像气泡一样"浮"到数组的最后
          for (int i = 0; i < n - 1 - j; i++)  {   //比较相邻的两个元素,使较大的那个向后
             if (A[i] > A[i + 1])   {  // 如果条件改成 A[i] >= A[i + 1],则变为不稳定
                  Swap(A, i, i + 1);
              }
          }
       }
    }
}
```

1.2 选择排序(Selection Sort)

思想：初始时在序列中找到最小（大）元素，放到序列的起始位置作为已排序序列；然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。选择排序每遍历一次都记住了当前最小（大）元素的位置，最后**仅需一次交换操作**即可将其放到合适的位置。

```
初始值：3  1  5  7  2  4  9  6

第1趟 : 1  3  5  7  2  4  9  6
第2趟 : 1  2  5  7  3  4  9  6
第3趟 : 1  2  3  7  5  4  9  6
第4趟 : 1  2  3  4  5  7  9  6
第5趟 : 1  2  3  4  5  7  9  6
第6趟 : 1  2  3  4  5  6  9  7
第7趟 : 1  2  3  4  5  6  7  9
第8趟 : 1  2  3  4  5  6  7  9
```

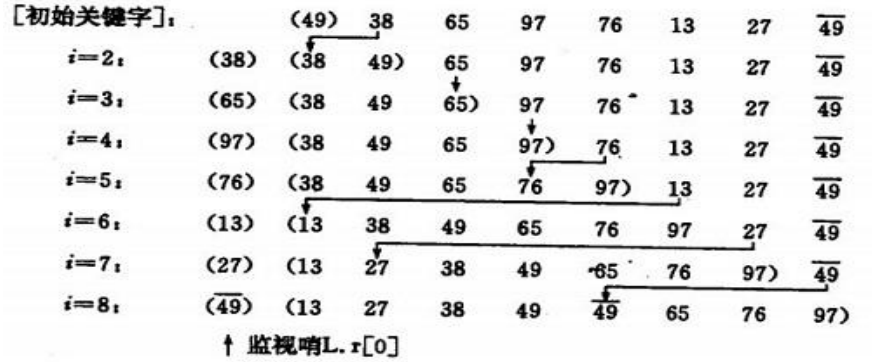
```
public class SelectionSort {
    // 分类 -------------- 内部比较排序
    // 数据结构 ---------- 数组
    // 最差时间复杂度 ---- O(n^2)
    // 最优时间复杂度 ---- O(n^2)
    // 平均时间复杂度 ---- O(n^2)
    // 所需辅助空间 ------ O(1)
    // 稳定性 ------------ 不稳定
    // 排序前：      { 6, 5, 3, 1, 8, 7, 2, 4 }
    // 第一遍排序：  { 1, 5, 3, 6, 8, 7, 2, 4 }  找出最小的，和第一位置上交换
    // 第二遍排序：  { 1, 2, 3, 6, 8, 7, 5, 4 }  找出第二小的，和第二个位置上交换

    void Swap(int A[], int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
    void SelectionSort(int A[], int n)  {
        for (int i = 0; i < n - 1; i++)  {       // i 为已排序序列的末尾
            int min = i;
```

```
            for (int j = i + 1; j < n; j++)  {    // 未排序序列
                if (A[j] < A[min])   {           // 找出未排序序列中的最小值
                    min = j;
                }
            }
            if (min != i)  {
                Swap(A, min, i);    // 放到已排序序列的末尾，该操作很有可能把稳定性打乱
            }
        }
    }
}
```

1.3 插入排序(Insertion Sort)

思想：对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。



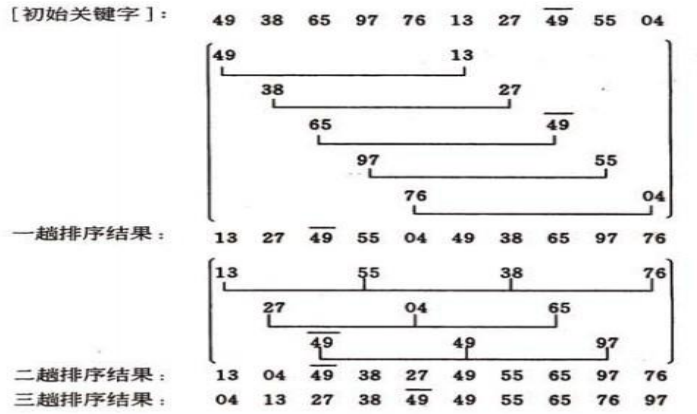

```
public class InsertionSort {
    // 分类 ------------- 内部比较排序
    // 数据结构 ---------- 数组
    // 最差时间复杂度 ---- 最坏情况为输入序列是降序排列的,此时时间复杂度 O(n^2)
    // 最优时间复杂度 ---- 最好情况为输入序列是升序排列的,此时时间复杂度 O(n)
    // 平均时间复杂度 ---- O(n^2)
    // 所需辅助空间 ------ O(1)
    // 稳定性 ------------ 稳定
    // 排序前：      { 6, 5, 3, 1, 8, 7, 2, 4 }
    // 第一遍排序：  { 5, 6, 3, 1, 8, 7, 2, 4 }  找到第两个数的位置，先将前两个排好序
    // 第二遍排序：  { 3, 5, 6, 1, 8, 7, 2, 4 }  找到第三个数的位置，排好前三个数的顺序
    void InsertionSort(int A[], int n) {
        for (int i = 1; i < n; i++) {        // 类似抓扑克牌排序
            int get = A[i];                  // 右手抓到一张扑克牌
            int j = i - 1;                   // 拿在左手上的牌总是排序好的
            while (j >= 0 && A[j] > get) {   // 将抓到的牌与手牌从右向左进行比较
                A[j + 1] = A[j];             // 如果该手牌比抓到的牌大，就将其右移
                j--;
            }
            A[j + 1] = get;
        }
    }
```

插入排序的改进：二分插入排序

```
void InsertionSortDichotomy(int A[], int n) {
        for (int i = 1; i < n; i++)  {
            int get = A[i];
            int left = 0;    // 采用二分法定位新牌的位置
            int right = i - 1;
            while (left <= right)   {
                int mid = (left + right) / 2;
                if (A[mid] > get)
                    right = mid - 1;
                else
                    left = mid + 1;
            }
            for (int j = i - 1; j >= left; j--) {   // 将欲插入位置右边的整体向右移动
                A[j + 1] = A[j];
            }
            A[left] = get;
        }
    }
```

1.4 希尔排序(Shell Sort)

插入排序的更高效改进：希尔排序(Shell Sort)。希尔排序，也叫递减增量排序，是插入排序的一种更高效的改进版本。希尔排序是不稳定的排序算法。希尔排序通过将比较的全部元素分为几个区域来提升插入排序的性能，然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序。




```
public class ShellSort {
    // 分类 -------------- 内部比较排序
    // 数据结构 ---------- 数组
    // 最差时间复杂度 ---- 根据步长序列的不同而不同。已知最好的为O(n(logn)^2)
    // 最优时间复杂度 ---- O(n)
    // 平均时间复杂度 ---- 根据步长序列的不同而不同。
    // 所需辅助空间 ------ O(1)
    // 稳定性 ------------ 不稳定
    // 排序前：      { 6, 5, 3, 1, 8, 7, 2, 4 }
    // 第一遍排序：  { 6, 5, 3, 1, 8, 7, 2, 4 }  找到第两个数的位置，先将前两个排好序  h=13
```

```
    // 第二遍排序： { 6, 5, 2, 1, 8, 7, 3, 4 }  找到第三个数的位置，排好前三个数的顺序 h=4
    // 第三遍排序： h=1
        // { 5, 6, 2, 1, 8, 7, 3, 4 }
        // { 2, 5, 6, 1, 8, 7, 3, 4 }
        // { 1, 2, 5, 6, 8, 7, 3, 4 }
        // { 1, 2, 3, 5, 6, 8, 7, 4 }
        // { 1, 2, 3, 4, 5, 6, 8, 7 }
        // { 1, 2, 3, 4, 5, 6, 7, 8 }
    void ShellSort(int A[], int n) {
        int h = (n-1) / 3 ;
        while (h >= 1) {
            for (int i = h; i < n; i++) {  // i 是后一半
                int j = i - h;    //j 是后一半
                int get = A[i];   //
                while (j >= 0 && A[j] > get) {
                    A[j + h] = A[j];
                    j = j - h;
                }
                A[j + h] = get;
            }
            h = (n-1) / 3;                    // 递减增量
        }
    }
```

1.5 归并排序(Merge Sort)

归并排序的实现分为递归实现与非递归(迭代)实现。递归实现的归并排序是算法设计中分治策略的典型应用，我们将一个大问题分割成小问题分别解决，然后用所有小问题的答案来解决整个大问题。

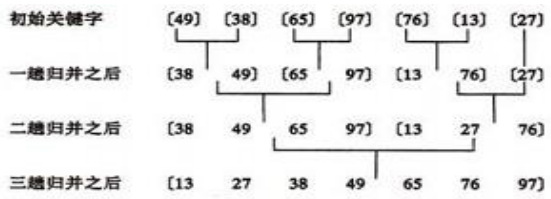
步骤：①申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列；②设定两个指针，最初位置分别为两个已经排序序列的起始位置；③比较两个指针所指向的元素，选择相对小的元素放到合并空间，并移动指针到下一位置；④重复步骤 3 直到某一指针到达序列尾；⑤将另一序列剩下的所有元素直接复制到合并序列尾。

```
public class MergeSort {
    // 分类 -------------- 内部比较排序
    // 数据结构 ---------- 数组
    // 最差时间复杂度 ---- O(nlogn)
    // 最优时间复杂度 ---- O(nlogn)
    // 平均时间复杂度 ---- O(nlogn)
    // 所需辅助空间 ------ O(n)
    // 稳定性 ------------ 稳定
// 合并两个已排好序的数组 A[left...mid]和 A[mid+1...right]
    void Merge(int A[], int left, int mid, int right) {
        int len = right - left + 1;
        int[] temp = new int[len];       // 辅助空间 O(n)
        int index = 0;
        int i = left;                   // 前一数组的起始元素
```

```
    int j = mid + 1;                // 后一数组的起始元素
    while (i <= mid && j <= right) {     // 把较小的数先移到新数组中
        temp[index++] = A[i] <= A[j] ? A[i++] : A[j++];  // 带等号保证稳定性
    }
    while (i <= mid) {    // 把左边剩余的数移入数组
        temp[index++] = A[i++];
    }
    while (j <= right) {      // 把右边边剩余的数移入数组
        temp[index++] = A[j++];
    }
    for (int k = 0; k < len; k++) {      // 把新数组中的数覆盖 nums 数组
        A[left++] = temp[k];
    }
}

void MergeSortRecursion(int A[], int left, int right) {   // 递归实现的归并排序
    if (left == right)    // 当待排序的序列长度为 1 时，递归开始回溯，进行 merge 操作
        return;
    int mid = (left + right) / 2;
    MergeSortRecursion(A, left, mid);  // 左边
    MergeSortRecursion(A, mid + 1, right); // 右边
    Merge(A, left, mid, right); // 左右归并
}

void MergeSortIteration(int A[], int len)  {    // 非递归(迭代)实现的归并排序
    int left, mid, right;
    for (int i = 1; i < len; i *= 2)  {       // 子数组的大小 i 初始为 1，每轮翻倍
        left = 0;
        while (left + i < len)  {            // 后一个子数组存在(需要归并)
            mid = left + i - 1;
            right = mid + i < len ? mid + i : len - 1;// 后一个子数组大小可能不够
            Merge(A, left, mid, right);
            left = right + 1;              // 前一个子数组索引向后移动
        }
    }
}
```



1.6 堆排序(Heap Sort)

堆排序是指利用堆这种数据结构所设计的一种选择排序算法。堆是一种近似完全二叉树的结构（通常堆是通过一维数组来实现的），并满足性质：以最大堆（也叫大根堆、大顶堆）为例，其中父结点的值总是大于它的孩子节点。

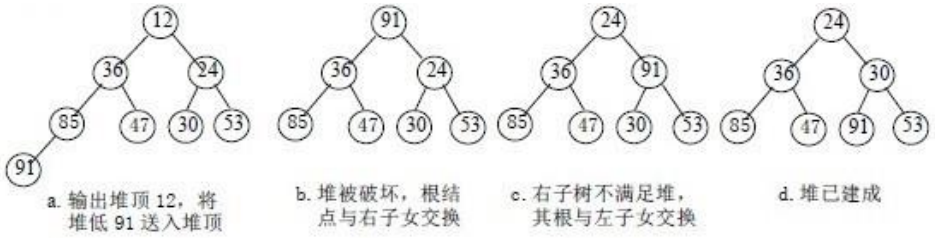
我们可以很容易的定义堆排序的过程：

①由输入的无序数组构造一个最大堆，作为初始的无序区；

②把堆顶元素（最大值）和堆尾元素互换；

③把堆（无序区）的尺寸缩小 1，并调用 heapify(A, 0)从新的堆顶元素开始进行堆调整；

④重复步骤 2，直到堆的尺寸为 1。




```
public class HeapSort {
    // 分类 -------------- 内部比较排序
    // 数据结构 ---------- 数组
    // 最差时间复杂度 ---- O(nlogn)
    // 最优时间复杂度 ---- O(nlogn)
    // 平均时间复杂度 ---- O(nlogn)
    // 所需辅助空间 ------ O(1)
    // 稳定性 ------------ 不稳定
    void Swap(int A[], int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }

    // 从 A[i]向下进行堆调整
    void Heapify(int A[], int i, int size) {
        int left_child = 2 * i + 1;         // 左孩子索引
        int right_child = 2 * i + 2;        // 右孩子索引
        int max = i;                        // 选出当前结点与其左右孩子三者之中的最大值
        if (left_child < size && A[left_child] > A[max])
            max = left_child;
        if (right_child < size && A[right_child] > A[max])
            max = right_child;
        if (max != i) {
            Swap(A, i, max);                // 把当前结点和它的最大(直接)子节点进行交换
            Heapify(A, max, size);          // 递归调用，继续从当前结点向下进行堆调整
        }
    }
```

```
// 建堆，时间复杂度 O(n)
int BuildHeap(int A[], int n) {
    int heap_size = n;
    // 从每一个非叶结点开始向下进行堆调整
    for (int i = heap_size / 2 - 1; i >= 0; i--)
        Heapify(A, i, heap_size);
    return heap_size;
}

void HeapSort(int A[], int n) {
    // 建立一个最大堆
    int heap_size = BuildHeap(A, n);
    while (heap_size > 1) {   // 堆（无序区）元素个数大于 1，未完成排序
        // 将堆顶元素与堆的最后一个元素互换，并从堆中去掉最后一个元素
        // 此处交换操作很有可能把后面元素的稳定性打乱，所以堆排序是不稳定的排序算法
        Swap(A, 0, --heap_size);
        Heapify(A, 0, heap_size); // 从堆顶元素开始向下进行堆调整，时间复杂度 O(logn)
    }
}
```

1.7 快速排序(Quick Sort)

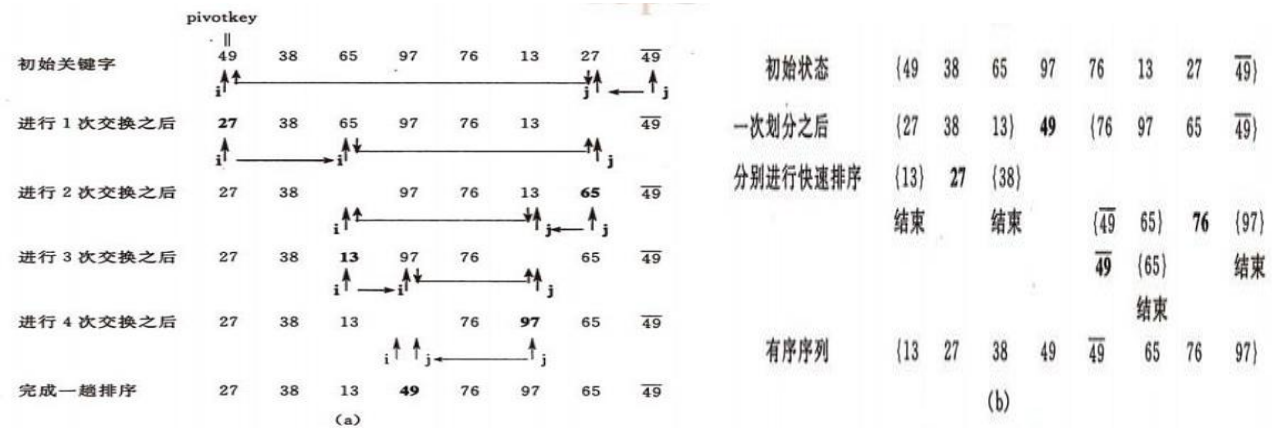
快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 n 个元素要 O(nlogn)次比较。在最坏状况下则需要 O(n^2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他 O(nlogn)算法更快。

快速排序使用分治策略(Divide and Conquer)来把一个序列分为两个子序列。步骤为：

①从序列中挑出一个元素，作为"基准"(pivot)；

②把所有比基准值小的元素放在基准前面，所有比基准值大的元素放在基准的后面（相同的数可以到任一边），这个称为分区(partition)操作；

③对每个分区递归地进行步骤 1~2，递归的结束条件是序列的大小是 0 或 1，这时整体已经被排好序了。




```
public class QuickSort {
    // 分类 ------------ 内部比较排序
    // 数据结构 --------- 数组
    // 最差时间复杂度 ----时间复杂度为 O(n^2)
```

```
// 最优时间复杂度 ----时间复杂度为O(nlogn)
// 平均时间复杂度 ---- O(nlogn)
// 所需辅助空间 ------般为O(logn)，最差为O(n)
// 稳定性 ---------- 不稳定

void Swap(int A[], int i, int j) {
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

// 划分函数
int Partition(int A[], int left, int right) {
    int pivot = A[right];              // 这里每次都选择最后一个元素作为基准
    int tail = left - 1;               // tail为小于基准的子数组最后一个元素的索引
    for (int i = left; i < right; i++) {     // 遍历基准以外的其他元素
        if (A[i] <= pivot) {  // 把小于等于基准的元素放到前一个子数组末尾
            Swap(A, ++tail, i);
        }
    }
    Swap(A, tail + 1, right);  //把基准放到前一个子数组的后边，剩下的是大于基准的子数组
    return tail + 1;                   // 返回基准的索引
}

void QuickSort(int A[], int left, int right) {
    if (left >= right)  return;
    int pivot_index = Partition(A, left, right); // 基准的索引
    QuickSort(A, left, pivot_index - 1);
    QuickSort(A, pivot_index + 1, right);
}
```

2 Tree

2.1 遍历

2.1.1 前序遍历

```
    * 递归先序遍历
    */
public static void preOrderRec(Node root){
    if(root!=null){
        System.out.println(root.value);
        preOrderRec(root.left);
        preOrderRec(root.right);
    }
}
```

```
     * 利用栈模拟递归过程实现循环先序遍历二叉树
     * 这种方式具备扩展性，它模拟递归的过程，将左子树点不断的压入栈，直到null，然后处理栈顶节点的右子树
     */
    public static void preOrderStack_2(Node root){
        if(root==null)return;
        Stack<Node> s=new Stack<Node>();
        while(root!=null||!s.isEmpty()){
            while(root!=null){
                System.out.println(root.value);
                s.push(root);//先访问再入栈
                root=root.left;
            }
            root=s.pop();
            root=root.right;//如果是null，出栈并处理右子树
        }
    }
```

2.1.2 中序遍历

```
     * 递归中序遍历
     */
    public static void inOrderRec(Node root){
        if(root!=null){
            preOrderRec(root.left);
            System.out.println(root.value);
            preOrderRec(root.right);
        }
    }
```

```
    * 利用栈模拟递归过程实现循环中序遍历二叉树
    * 思想和上面的preOrderStack_2相同，只是访问的时间是在左子树都处理完直到null的时候出栈并访问。
    */
   public static void inOrderStack(Node root){
       if(root==null)return;
       Stack<Node> s=new Stack<Node>();
       while(root!=null||!s.isEmpty()){
           while(root!=null){
               s.push(root);//先访问再入栈
               root=root.left;
           }
           root=s.pop();
           System.out.println(root.value);
           root=root.right;//如果是null，出栈并处理右子树
       }
   }
```

2.1.3 后序遍历

```
  * 递归后序遍历
  */
 public static void postOrderRec(Node root){
     if(root!=null){
         preOrderRec(root.left);
         preOrderRec(root.right);
         System.out.println(root.value);
     }
 }
```

```
     * 后序遍历不同于先序和中序，它是要先处理完左右子树，然后再处理根(回溯)，所以需要一个记录哪些节点已经被访问的结构(可以在树结构里面加一个标记)，这里可以用map实现
     */
    public static void postOrderStack(Node root){
        if(root==null)return;
        Stack<Node> s=new Stack<Node>();
        Map<Node,Boolean> map=new HashMap<Node,Boolean>();
        s.push(root);
        while(!s.isEmpty()){
            Node temp=s.peek();
            if(temp.left!=null&&!map.containsKey(temp.left)){
                temp=temp.left;
                while(temp!=null){
                    if(map.containsKey(temp))break;
                    else s.push(temp);
                    temp=temp.left;
                }
                continue;
            }
            if(temp.right!=null&&!map.containsKey(temp.right)){
                s.push(temp.right);
                continue;
            }
            Node t=s.pop();
            map.put(t,true);
            System.out.println(t.value);
        }
    }
```

2.1.4 层次遍历

```
public static ArrayList<ArrayList<Integer>>  Print(TreeNode pRoot) {
    ArrayList<Integer> tmp = new ArrayList<>();

    if(pRoot == null)
        return null;
    LinkedList<TreeNode> queue  = new LinkedList<>(); //队列LinkedList完成层序遍历
    ArrayList<ArrayList<Integer>> ret = new ArrayList<>();


    TreeNode current = null; //当前节点

    int now = 1, next = 0;//now 用于记录遍历的个数，next表示每层节点数量

    queue.add(pRoot);//将根节点入队

    while(!queue.isEmpty()) {
        current = queue.poll();//出队队头元素并访问
        now--;

        tmp.add(current.val); //添加到list

        if(current.left != null) { //如果当前节点的左节点不为空入队
            queue.add(current.left);
            next++;
        }
        if(current.right != null) { //如果当前节点的右节点不为空，把右节点入队
            queue.add(current.right);
            next++;
        }
        if(now == 0) {
            ret.add(new ArrayList<Integer>(tmp));
            tmp.clear();
            now = next;
            next = 0;
        }
    }
    return ret;
}
```

2.2 BST

BST(二叉搜索树)特点：左孩子结点值比当前结点值小，右孩子结点值比当前值大当前值；

2.2.1 插入

/**

```
  * 插入
  *      1. 从 root 节点开始，如果 root 为空,root 为插入值，循环：
  *      2. 如果当前节点值大于插入值,找左节点
  *      3. 如果当前节点值小于插入值,找右节点
  */
public TreeNode insert(TreeNode root, int value) {    //插入指定值的结点
    TreeNode node = new TreeNode(value);
    TreeNode current = root;
    TreeNode parent  = null;
    if(root == null){
        root = node;
        return root;
    }
    while (true) {
        parent = current;
        if (value < current.getValue()) {
            current = current.getLeftChild();
            if (current == null) {
                parent.setLeftChild(node);
                return node;
            }
        } else {
            current = current.getRightChild();
            if (current == null) {
                parent.setRightChild(node);
                return node;
            }
        }
    }
}
```

2.2.2 删除

```
/**
 * 删除节点
 *      1.找到删除节点
 *      2.如果删除节点左节点为空 , 右节点也为空;
 *      3.如果删除节点只有一个子节点 右节点 或者 左节点
 *      4.如果删除节点左右子节点都不为空
 */
public TreeNode delete (TreeNode root, int key) {
    TreeNode parent  = root;
    TreeNode current = root;
    boolean isLeftChild = false;

    // 找到删除节点
```

```
while (current.getValue() != key) {
    parent = current;
    if (current.getValue() > key) {
        isLeftChild = true;
        current = current.getLeftChild();
    } else {
        isLeftChild = false;
        current = current.getRightChild();
    }
    if (current == null) {
        System.out.println("Not Found!!");
        return current;
    }
}
TreeNode leftChild = current.getLeftChild();
TreeNode rightChild = current.getRightChild();

// 如果删除节点 没有左、右节点
if (leftChild == null && rightChild == null) {
    if (current == root) {   //为根节点
        root = null;
    }
    if (isLeftChild == true) {   // 在左子树
        parent.setLeftChild(null);
    } else {
        parent.setRightChild(null);
    }
} else if (rightChild == null) { //只有左孩子
    if (current == root) {   //删除的根节点，根节点变为左孩子
        root = leftChild;
    } else if (isLeftChild) {   //删除的左节点，父节点的左孩子变为左孩子
        parent.setLeftChild(leftChild);
    } else {   //删除的右节点，父节点的右孩子变为左孩子
        parent.setRightChild(leftChild);
    }
}  else if (leftChild == null) {  //只有右孩子
    if (current == root) {    //删除的根节点，根节点变为左孩子
        root = rightChild;
    } else if (isLeftChild) {    //删除的左节点，父节点的左孩子变为右孩子
        parent.setLeftChild(rightChild);
    } else {    //删除的右节点，父节点的右孩子变为右孩子
        parent.setRightChild(rightChild);
    }
}  else if (leftChild != null && rightChild != null) {// 如果左右子节点都不为空
```

```
        // 找到删除节点的后继者
        TreeNode successor = getDeleteSuccessor(current);
        if (current == root) {
            root = successor;
        } else if (isLeftChild) {
            parent.setLeftChild(successor);
        } else {
            parent.setRightChild(successor);
        }
        successor.setLeftChild(current.getLeftChild());
    }
    return current;
}
/* 获取删除节点的后继者：删除节点的后继者是在其右节点树种最小的节点
 */
public TreeNode getDeleteSuccessor(TreeNode deleteNode) {
    // 后继者
    TreeNode successor = null;
    TreeNode successorParent = null;
    TreeNode current = deleteNode.getRightChild();
    while (current != null) {
        successorParent = successor;
        successor = current;
        current = current.getLeftChild();
    }
    // 检查后继者(不可能有左节点树)是否有右节点树
    // 如果它有右节点树,则替换后继者位置,加到后继者父亲节点的左节点.
    if (successor != deleteNode.getRightChild()) {
        successorParent.setLeftChild(successor.getRightChild());
        successor.setRightChild(deleteNode.getRightChild());
    }
    return successor;
}
```

2.2.3 搜索

```
public TreeNode search(TreeNode root, int key) {    //查找指定结点方法
    TreeNode current = root;
    while (current != null && key != current.getValue()) {
        if (key < current.getValue() )
            current = current.getLeftChild();
        else
            current = current.getRightChild();
    }
    return current;
}
```

2.3 BTREE

2.3.1 B-Tree

B 树又叫平衡多路查找树。一棵 m 阶的 B 树的特性如下：

①树中每个结点**最多含有 m 个孩子**（m>=2），最多容纳 m-1 个关键字；

②除根结点和叶子结点外，**其它每个结点至少有[ceil(m / 2)]个孩子**（其中 ceil(x)是一个取上限的函数）；

③若**根结点**不是叶子结点，则**至少有 2 个孩子**；

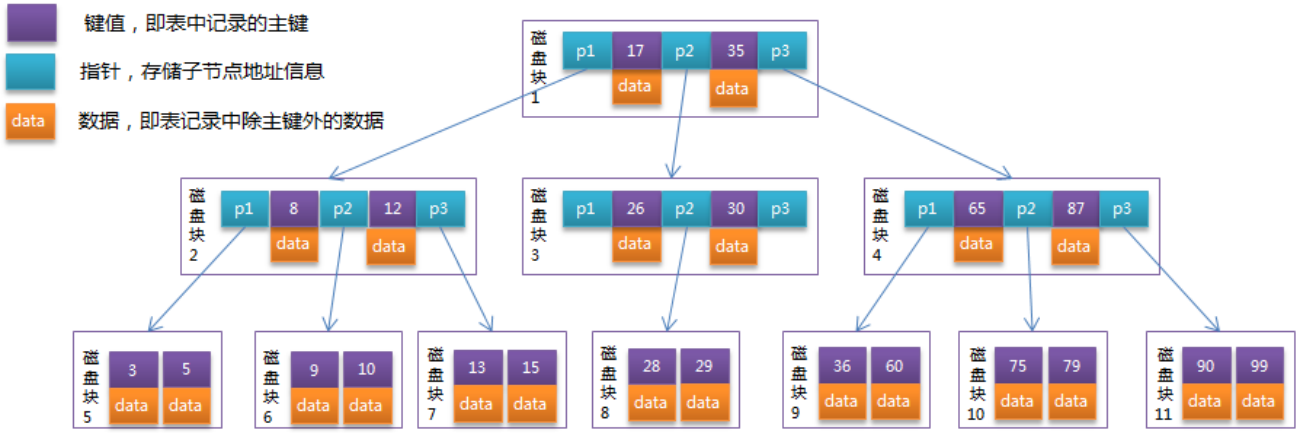
④**所有叶子结点都出现在同一层**，叶子结点不包含任何关键字信息；

⑤每个非终端结点中包含有 n 个关键字信息： (n，P0，K1，P1，K2，P2，……，Kn，Pn)。其中：

a) Ki (i=1...n)为关键字，且关键字按顺序升序排序 K(i-1)< Ki。

b) Pi 为指向子树根的接点，且指针 P(i-1)指向子树中所有结点的关键字均小于 Ki，但都大于 K(i-1)。

c) 关键字的个数 n 必须满足： [ceil(m / 2)-1]<= n <= m-1。

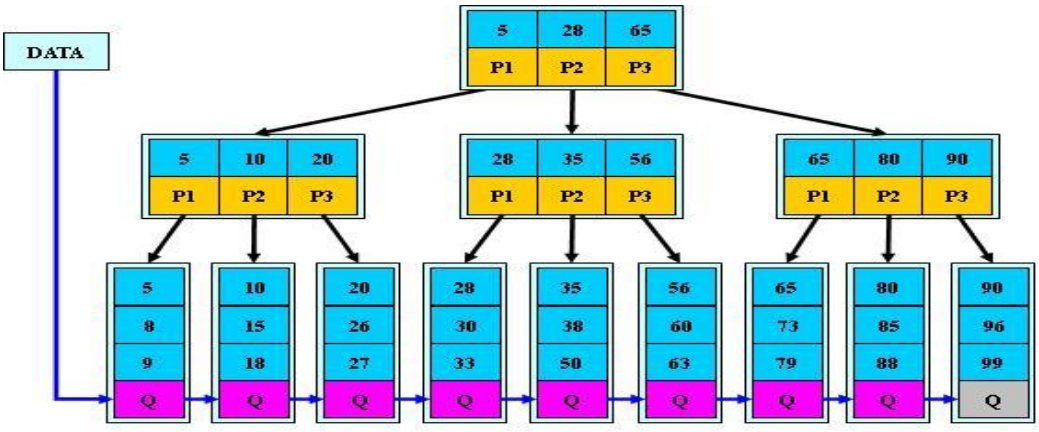


每个节点占用一个盘块的磁盘空间，**一个节点上有两个升序排序的关键字和三个指向子树根节点的指针**，指针存储的是子节点所在磁盘块的地址。**两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域**。以根节点为例，关键字为 17 和 35，P1 指针指向的子树的数据范围为小于 17，P2 指针指向的子树的数据范围为 17~35，P3 指针指向的子树的数据范围为大于 35。

2.3.2 B+Tree

B+Tree 是在 B-Tree 基础上的一种优化，使其更适合实现外存储索引结构，**InnoDB 存储引擎就是用 B+Tree 实现其索引结构。**B+Tree 相对于 B-Tree 有几点不同：

①**非叶子节点只存储键值信息，数据记录都存放在叶子节点中**；②有 K 个子节点的节点一定有 K 个关键码(B-Tree 是 K-1 个关键码)。③所有叶子节点之间都有一个链指针。④非叶子结点的子树指针 P[i]，指向关键字值属于[K[i],K[i+1])的子树（B-树是开区间）；



2.3.3 对比

B-树特点：①关键字集合分布在整棵树中，也就是说每个节点上都有关键字；②每个关键字出现且只会出现在一棵节点上；③查找关键字可能在非叶子节点结束；

B+树：①是 B-树的一种变形；②有 n 棵子树的结点中含有 n 个关键字； (而 B 树是 n 棵子树有 n-1 个关键字)；③所有的关键字肯定出现在叶子节点；④所有的叶子节点之间有个链指针，即所有的叶子节点连起来后是个链表。⑤.所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。 (而 B 树的非终节点也包含需要查找的有效信息)。

2.4 红黑树

红黑树特性：① 每个节点都有颜色，要么是红色，要么是黑色；②**根和叶子节点都是黑色的**；③**不能有连续两个红色的节点**；④从任意节点到它所能到达得叶子节点的所有路径都包含**相同数目的黑色节点**；⑤叶子节点是 null 节点，并非真实的叶子节点。

这些规则保证了红黑色的插入、删除、查找操作的**最坏时间复杂度都为 O(logn)。**

4 链表、栈和队列

4.2 两个队列实现栈

```
/**
 * 两个队列实现栈
 * 思想:
 * 入栈：两个队列哪个不为空，就把元素入队到哪个队列中；
 * 出栈：把不为空的队列中出最后一个元素外的所有元素都移动到两一个队列中，然后出队最后一个元素
 */
public class Queue2Stack {
    private Queue queue1;
    private Queue queue2;
    private int maxLenth;

    public Queue2Stack(int maxLenth){
        this.maxLenth = maxLenth;
        this.queue1 = new ArrayBlockingQueue(maxLenth);
        this.queue2 = new ArrayBlockingQueue(maxLenth);
    }

    public boolean push(int item){
        if(size() == maxLenth){
            return false;
        }
        if(queue2.isEmpty()) {
            queue2.add(item);
        }else {
            queue1.add(item);
        }
        return true;
```

```
    }

    public Object pop(){
        if(size() == 0){
            return null;
        }else{
            if(queue2.isEmpty()){
                while(queue1.size() > 1){
                    queue2.add(queue1.poll());
                }
                return queue1.poll();
            }else{
                while(queue2.size() > 1){
                    queue1.add(queue2.poll());
                }
                return queue2.poll();
            }
        }
    }

    public int size(){
        return queue1.size() + queue2.size();
    }
}
```

4.3 两个栈实现队列

```
/**
 * 用两个栈实现队列
 * 思想：入队都在 stack1 进行，出队都在 stack2 进行
 *  入队：直接把元素亚茹 stack1 中
 *  出队：如果 stack2 不为空，则直接弹出 stack2 中的元素；
 *       如果 Stack2 为空,则将 stack1 中的所有元素倒入 stack2 中,然后弹出 stack2 中的栈顶元素。
 *       若都为空，出队失败
 */
public class Stack2Queue {
    private Stack stack1;
    private Stack stack2;
    private int maxLength;

    public Stack2Queue(int maxLength) {
        this.maxLength = maxLength;
        this.stack1 = new Stack();
        this.stack2 = new Stack();
    }
```

```
    public boolean push(int item){
        if(stack1.size() == maxLength){
            return false;
        }
        stack1.push(item);
        return true;
    }

    public Object  poll(){
        if(stack2.isEmpty() && stack1.isEmpty())
            return null;
        if(!stack2.isEmpty()){
            return stack2.pop();
        } else {
            while (!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
            return stack2.pop();
        }
    }
    public int size(){
        return stack1.size() + stack2.size();
    }
}
```

5 查找

5.1 杨氏矩阵查找

```
/**
 * 杨氏矩阵查找
 * 思想:
 *      先定位第一行最后一个元素，
 *      遇到的数比要找的数大时，向左移动，遇到的数比要找的数小时，向下移动
 */
public class YoungSearch {
    public boolean search(int[][] array, int target){
        int i = 0, j = array[0].length -1;
        int temp = array[i][j];
        while(true){
            if(temp == target) {
                System.out.println("x=" + i + ", y=" + j);
                return true;
            } else if(temp < target && i < array.length - 1){
                temp = array[++i][j];
```

```
            }else if(temp > target && j > 0){
                temp = array[i][--j];
            } else {
                System.out.println("Not found");
                return false;
            }
        }
    }
}
```

6 算法思想

6.1 分治法

分治法：把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后问题可以被简单地直接求解，原问题的解即子问题的解和合并。

例如：Fork/Join 框架，快速排序、二分查找

6.2 动态规划

动态规划：将带求解的问题分解为若干个子问题，按顺序求解子问题，前一个子问题的解为后一个问题的求解提供了有用的信息。

与分治法区别：分解后的子问题往往不是相互独立的；

例如：最短距离

6.3 回溯法

7 其他算法

7.1 数组

7.1.1 调整数组顺序使奇数位于偶数前面

题目：输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

```
public void reOrderArray(int [] array) {
        HashMap<Integer,Integer> map = new HashMap<Integer,Integer>();
        int length = array.length;
        int[] result = array;
        int index = 0;
        int arr = 0;
        for (int i = 0; i < length; i++) {
            if((result[i] & 0x1) == 1){
                array[index++] = result[i];
            }else{
                map.put(arr++, result[i]);
```

```
            }
        }
        for (int i = 0; i < arr; i++) {
            array[index++] = map.get(i);
}
```

7.1.2 顺时针打印二维数组

```
public ArrayList<Integer> printMatrix(int [][] matrix) {
        ArrayList<Integer> result = new ArrayList<>();
        int row = matrix.length;
        int col = matrix[0].length;
        if (row == 0 || col == 0)  return result;
        // 定义四个变量，表示左上和右下的打印范围
        int left = 0, top = 0, right = col - 1, bottom = row - 1;
        while (left <= right && top <= bottom) {
            for (int i = left; i <= right; ++i)    // left to right
                result.add(matrix[top][i]);
            for (int i = top + 1; i <= bottom; ++i)   // top to bottom
                result.add(matrix[i][right]);
            if (top != bottom) {  // right to left
                for (int i = right - 1; i >= left; --i)
                    result.add(matrix[bottom][i]);
            }
            if (left != right) {  // bottom to top
                for (int i = bottom - 1; i > top; --i)
                    result.add(matrix[i][left]);
            }
            left++;top++;right--;bottom--;
        }
        return result;
}
```

7.1.3 数组中出现次数超过一半的数字

题目：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，输出 2。如果不存在则输出 0。

思路：**采用阵地攻守的思想**：①第一个数字作为第一个士兵，守阵地：count = 1；②遇到相同元素，count++;遇到不同元素，即为敌人，count--；③当遇到 count 为 0 的情况，又以新的 i 值作为守阵地的士兵，继续下去，到最后还留下的，有可能是答案。④再一次循环，记录这个士兵的个数看是否大于数组一般即可。

```
public class MoreThanHalfNum {
    //时间复杂度：O(n)
    public int MoreThanHalfNum_Solution(int [] array) {
        int length = array.length;
        if(array == null || length <= 0)  return 0;
        int result = array[0];
        int times = 1;
```

```
        for (int i = 1; i < length; i++) {
            if(times == 0){
                result = array[i];
                times = 1;
            }else if(array[i] == result) times++;
            else  times--;
        }
        if(!CheckMoreThanHalf(array, length, result))  result = 0;
        return result;
    }
    boolean CheckMoreThanHalf(int[] numbers, int length, int number){
        int times = 0;
        for (int i = 0; i < length; i++){
            if(numbers[i] == number)
                times++;
        }
        boolean flag = true;
        if(times * 2 <= length){
            flag = false;
        }
        return flag;
    }
}
```

7.1.4 连续子数组的最大和

思路：使用动态规划，f(i)表示第 i 个数字结尾的子数组的最大和，那么

$$\mathbf{f(i)} = \begin{cases} \boldsymbol{pData[i]} & \boldsymbol{i = 0 \text{ 或者} f(i-1) \ll 0} \\ \boldsymbol{f(i-1) + pData[i]} & \boldsymbol{i \neq 0 \text{并且} f(i-1) > 0} \end{cases}$$

```
public int findGreatestSumOfSubArray(int[] array) {
        if (array==null || array.length==0)
            return 0;
        int curSum = array[0], greatestSum = array[0]; //注意初始值防止只有负数
        for (int i = 1; i < array.length; i++) {
            if (curSum<=0) {
                curSum=array[i]; //记录当前最大值
            }else {
                curSum+=array[i]; //当 array[i]为正数时，加上之前的最大值并更新最大值。
            }
            if (curSum>greatestSum) {
                greatestSum=curSum;
            }
        }
        return greatestSum;
    }
```

7.1.5 数组中的逆序对

题目：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。

思路：【归并排序的改进】①把数据分成前后两个数组(递归分到每个数组仅有一个数据项)，合并数组,合并时,出现前面的数组值 array[i]大于后面数组值 array[j]时;则前面数组 array[i]~array[mid]都是大于 array[j]的，count += mid+1 – i。②还有就是测试用例输出结果比较大，对每次返回求余。

```
public class Solution {
       public int InversePairs(int [] array) {
            if(array==null||array.length==0)  return 0;
            int[] copy = new int[array.length];
            for(int i=0;i<array.length;i++) {
                copy[i] = array[i];
            }
            int count = InversePairsCore(array,copy,0,array.length-1);//数值过大求余
            copy = null;
            return count;

        }
        private int InversePairsCore(int[] array,int[] copy,int low,int high) {
            if(low==high) return 0;
            int mid = (low+high)>>1;
            int leftCount = InversePairsCore(array,copy,low,mid)%1000000007;
            int rightCount = InversePairsCore(array,copy,mid+1,high)%1000000007;
            int count = 0;
            int i=mid;
            int j=high;
            int locCopy = high;
            while(i>=low&&j>mid) {
                if(array[i]>array[j]) {
                    count += j-mid;
                    copy[locCopy--] = array[i--];
                    if(count>=1000000007)  count%=1000000007;
                } else
                    copy[locCopy--] = array[j--];
            }
            for(;i>=low;i--)
                copy[locCopy--]=array[i];
            for(;j>mid;j--)
                copy[locCopy--]=array[j];
            for(int s=low;s<=high;s++)
                array[s] = copy[s];
            return (leftCount+rightCount+count)%1000000007;
        }
    }
```

7.1.6 数字在排序数组中出现的次数

思路：由于已经排好序，则直接使用二分法。

```
public class GetNumberOfK {
    public int GetNumberOfK(int [] array , int k) {
        int number = 0;
        if(array != null && array.length > 0){
            int length = array.length;
            int firstK = getFirstK(array, k, 0, length-1);
            int lastK = getLastK(array, k, 0, length-1);
            if(firstK != -1 && lastK != -1)
              number =  lastK - firstK + 1;
        }
        return number;
    }
    private int getFirstK(int [] array , int k, int start, int end){ //递归写
        if(start > end)  return -1;
        int mid = (start + end) >> 1;
        if(array[mid] > k){ //如果中间的数字比 K 大，那么 k 只能出现在前半段
            return getFirstK(array, k, start, mid-1);
        }else if (array[mid] < k){  //如果中间的数字比 K 小，那么 K 只能出现在后半段
            return getFirstK(array, k, mid+1, end);
        }//如果中间的数字=K，那么比较 mid-1 的数字：如果相等，则在前半段
        else if(mid-1 >=0 && array[mid-1] == k){
            return getFirstK(array, k, start, mid-1);
        }else{  //如果小于，mid 就为第一个位置
            return mid;
        }
    }
    private int getLastK(int [] array , int k, int start, int end){ //循环写法
        if(start > end)  return -1;
        int length = array.length;
        int mid = (start + end) >> 1;
        while(start <= end){
            if(array[mid] > k)    end = mid-1;
            else if(array[mid] < k)    start = mid+1;
            else if(mid+1 < length && array[mid+1] == k)  start = mid+1;
            else  return mid;
            mid = (start + end) >> 1;
        }
        return -1;
    }
}
```

7.1.7 数组中只出现一次的数字

题目：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。要求时间复杂度为：O(n)，空间复杂度为 O(1)。

思路：位运算中异或的性质：两个相同数字异或=0，1 个数和 0 异或还是它本身。

① 首先从头到尾异或数组中的每个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。

② 然后在结果数字钟找到第一个为 1 的位的位置，即为第 n 位，将第 n 为是不是 1 分为两个子数组；

③ 然后在异或每个数组，得到结果即为只出现一次的数字

```
public class FindNumsAppearOnce {
    public void FindNumsAppearOnce(int[] array, int num1[], int num2[]) {
        if (array == null || array.length < 2)  return;
        int resultExclusiveOR = 0;
        for (int i = 0; i < array.length; i++)
            resultExclusiveOR ^= array[i];
        int indexOf1 = FindFirstBitIs1(resultExclusiveOR);
        for (int i = 0; i < array.length; i++) {
            if (isBit(array[i], indexOf1))
                num1[0] ^= array[i];
            else
                num2[0] ^= array[i];
        }
    }
    public int FindFirstBitIs1(int num) {
        int indexBit = 0;
        while (((num & 1) == 0) && (indexBit) < 8 * 4) {
            num = num >> 1;
            ++indexBit;
        }
        return indexBit;
    }
    public boolean isBit(int num, int indexBit) {
        num = num >> indexBit;
        return (num & 1) == 1;
    }
}
```

举一反三：数组中除一个数字只出现一次之外，其他数字都出现了三次。

思路：将二进制表示的每一位都加起来，如果某一位的和能被 3 整除，那么那个只出现一次的数字二进制中对应的哪一位是 0，其他为 1。

```
public int   FindNumsAppearOnce(int[] array) {
        if (array == null || array.length < 1)  return -1;
        int[] bitSum = new int[32];
        for (int i = 0; i < array.length; i++) {
            int bitMask = 1;
            for (int j = 31; j >= 0 ; j--) {
                int bit = array[i] & bitMask;
```

```
                if(bit != 0)
                    bitSum[j] += 1;
                bitMask = bitMask << 1;
            }
        }
        int result = 0;
        for (int i = 0; i < 32; i++) {
            result = result << 1;
            result += bitSum[i] % 3;
        }
        return result;
}
```

7.1.8 数组中重复的数字

题目描述：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。

解题思路：题目里写了数组里数字的范围保证在 0 ~ n-1 之间，所以可以利用现有数组设置标志，当一个数字被访问过后，可以设置对应位上的数 + n，之后再遇到相同的数时，会发现对应位上的数已经大于等于 n 了，那么直接返回这个数即可。

解法 1：

```
public boolean duplicate(int numbers[],int length,int [] duplication) {
        boolean flag = false;
        int index;
        for ( int i= 0 ; i<length; i++) {
            index = numbers[i];
            if (index >= length) {  index -= length; }
            if (numbers[index] >= length) {
                duplication[0] =  index;
                flag = true;
                break;
            }
            numbers[index] = numbers[index] + length;
        }
        return flag;
    }
```

解法 2：

```
public boolean duplicate(int numbers[],int length,int [] duplication) {
        if(numbers == null || numbers.length <= 0)  return false;
        int flag = 0;
        for (int i = 0; i < numbers.length; i++) {
            int number = numbers[i];
            if (((flag >> number) & 1) == 1) {
                duplication[0] = number;
                return true;
            }
            flag |= (1 << number);  //用二进制位来判断是否有数字重复
        }
```

```
            return false;
        }
```

7.1.9 构建乘积数组

题目描述： 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。

解题思路：

```
 /*  第一步：b[0] = 1;
 *  第二步：b[1] = b[0] * a[0] = a[0]
 *  第三步：b[2] = b[1] * a[1] = a[0] * a[1];
 *  第四步：b[3] = b[2] * a[2] = a[0] * a[1] * a[2];
 *  第五步：b[4] = b[3] * a[3] = a[0] * a[1] * a[2] * a[3];
 * 然后对于第二个 for 循环
 *  第一步
 *      temp *= a[4] = a[4];
 *      b[3] = b[3] * temp = a[0] * a[1] * a[2] * a[4];
 *  第二步
 *      temp *= a[3] = a[4] * a[3];
 *      b[2] = b[2] * temp = a[0] * a[1] * a[4] * a[3];
 *  第三步
 *      temp *= a[2] = a[4] * a[3] * a[2];
 *      b[1] = b[1] * temp = a[0] * a[4] * a[3] * a[2];
 *  第四步
 *      temp *= a[1] = a[4] * a[3] * a[2] * a[1];
 *      b[0] = b[0] * temp = a[4] * a[3] * a[2] * a[1];
 */
    public int[] multiply(int[] A) {
        int length = A.length;
        int[] result = new int[length];
        if (length > 1) {
            result[0] = 1;
            for (int i = 1; i < length; ++i) {
                result[i] = result[i - 1] * A[i - 1];
            }
            double temp = 1;
            for (int i = length - 2; i >= 0; --i) {
                temp *= A[i + 1];
                result[i] *= temp;
            }
        }
        return result;
    }
```

7.1.10 机器人的运动范围

题目描述：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？

思路：回溯法

```
public class Solution {
    public int movingCount(int threshold, int rows, int cols) {
        boolean[] visited = new boolean[rows * cols]; //记录是否已经走过
        return movingCountCore(threshold, rows, cols, 0, 0, visited);
    }
    private int movingCountCore(int threshold, int rows, int cols,
                                int row, int col, boolean[] visited) {
        if (row < 0 || row >= rows || col < 0 || col >= cols) return 0;
        int i = row * cols + col;
        if (visited[i] || !checkSum(threshold, row, col)) return 0;
        visited[i] = true;
        return 1 + movingCountCore(threshold, rows, cols, row, col + 1, visited)
               + movingCountCore(threshold, rows, cols, row, col - 1, visited)
               + movingCountCore(threshold, rows, cols, row + 1, col, visited)
               + movingCountCore(threshold, rows, cols, row - 1, col, visited);
    }
    private boolean checkSum(int threshold, int row, int col) {
        int sum = 0;
        while (row != 0) {
            sum += row % 10;
            row = row / 10;
        }
        while (col != 0) {
            sum += col % 10;
            col = col / 10;
        }
        if (sum > threshold) return false;
        return true;
    }
}
```

7.1.11 矩阵中的路径

题目描述：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。

思路：回溯法

```
public class Solution {
    public boolean hasPath(char[] matrix, int rows, int cols, char[] str) {
        if (matrix == null || matrix.length == 0
                || str == null || str.length == 0 || matrix.length != rows * cols
                || rows <= 0 || cols <= 0 || rows * cols < str.length) {
```

```
            return false;
        }
        boolean[] visited = new boolean[rows * cols]; //记录是否已经走过
        int pathLength = 0;
        for (int i = 0; i <= rows - 1; i++) {
            for (int j = 0; j <= cols - 1; j++) {
                if (hasPathCore(matrix, rows, cols, str, i, j, visited, pathLength)) {
                    return true;
                }
            }
        }
        return false;
    }
    public boolean hasPathCore(char[] matrix, int rows, int cols, char[] str,
                               int row, int col, boolean[] visited, int pathLength) {
        boolean flag = false;
        int i = row * cols + col;
        if (row >= 0 && row < rows && col >= 0 && col < cols
                && !visited[i] && matrix[i] == str[pathLength]) {
            pathLength++;
            visited[i] = true;
            if (pathLength == str.length) {
                return true;
            }
            flag = hasPathCore(matrix, rows, cols, str, row, col + 1, visited, pathLength)
              ||hasPathCore(matrix, rows, cols, str, row + 1, col, visited, pathLength)
              ||hasPathCore(matrix, rows, cols, str, row, col - 1, visited, pathLength)
              ||hasPathCore(matrix, rows, cols, str, row - 1, col, visited, pathLength);
            if (!flag) {
                pathLength--;
                visited[i] = false;
            }
        }
        return flag;
    }
}
```

7.2 字符串

7.2.1 字符串的排序

题目：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

思路：回溯法

```
public class Solution {
    public ArrayList<String> Permutation(String str) {
```

```
        ArrayList<String> res = new ArrayList<>();
        if (str != null && str.length() > 0) {
            PermutationHelper(str.toCharArray(), 0, res);
            Collections.sort(res);
        }
        return res;
    }
    public void PermutationHelper(char[] cs, int i, ArrayList list) {
        if(i == cs.length - 1) {
            String val = String.valueOf(cs);
            if (!list.contains(val))
                list.add(val);
        } else {
            for(int j = i; j < cs.length; ++j) {
                swap(cs, i, j);
                PermutationHelper(cs, i + 1, list);
                //最后交换完成入 List 之后再交换回来，回到初始状态
                swap(cs, i, j);
            }
        }
    }
}
```

7.2.2 第一个只出现一次的字符

题目：在一个字符串中找到第一个只出现一次的字符,并返回它的位置。

```
public int FirstNotRepeatingChar(String str) {
     LinkedHashMap <Character, Integer> map = new LinkedHashMap<>();
     for(int i=0;i<str.length();i++){
            if(map.containsKey(str.charAt(i))){
                int time = map.get(str.charAt(i));
                map.put(str.charAt(i), ++time);
            } else
               map.put(str.charAt(i), 1);
     }
     for(int i=0;i<str.length();i++)
            if (map.get(str.charAt(i)) == 1)
                return i;
     return -1;
   }
```

7.2.3 反转字符串

(1) 反转单词顺序

题目：输入一个英文句子，反转句子中单词的顺序，但单词内字符的顺序不变。例如，“student. a am I”。正确的句子应该是“I am a student.”。

```
public String ReverseSentence(String str) {
```

```
        if (str == null || str.length() == 0) return str;
        int len = str.length();
        char[] result = str.toCharArray();
        Reverse(result, 0, len - 1); //反转整个句子
        int start = 0, end = 0;
        while (start != len) {//若起始字符为空格，则 begin 和 end 都自加
            if (result[start] == ' ') {
                start++;
                end++;
            } else if (result[end] == ' ') {  //遍历到终止字符为空格，就进行翻转
                Reverse(result, start, --end);
                start = ++end;
            } else if (end == result.length - 1) {//若遍历结束，就进行翻转
                Reverse(result, start, end);
                start = ++end;
            } else {//没有遍历到空格或者遍历结束，则单独对 end 自减
                end++;
            }
        }
        return String.valueOf(result);
    }
    void Reverse(char[] list, int start, int end) {
        char temp;
        while (start < end) {
            temp = list[start];
            list[start] = list[end];
            list[end] = temp;
            start++;
            end--;
        }
    }
```

(2) 左旋转字符串

题目：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。

原理：$YX = (X^T Y^T)^T$

解法一：

```
    public String LeftRotateString(String str, int n) {
        if (str == null || str.length() == 0 || n <= 0) return str;
        int len = str.length();
        int kn = n % len;
        StringBuffer sb = new StringBuffer(str.substring(0, kn));
        StringBuffer sb1 = new StringBuffer(str.substring(kn, len));
        sb1.append(sb);
        return sb1.toString();
```

```
    }
```

解法二：更灵活

```
    String LeftRotateString1(String str, int n) {
        if (str == null || str.length() == 0 || n <= 0) return str;
        int len = str.length();
        int kn = n % len;
        char[] result = str.toCharArray();
        Reverse(result, 0, kn - 1);
        Reverse(result, kn, len - 1);
        Reverse(result, 0, len - 1);
        return String.valueOf(result);
    }
```

7.2.4 正则表达式匹配

题目描述：**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)。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) {
        if (str == null || pattern == null)  return false;
        return matchCore(str, 0, pattern, 0);
    }
    public boolean matchCore(char[] str, int s, char[] pattern, int p) {
        if (str.length <= s && pattern.length <= p)  return true;  //都匹配完了
        if (str.length > s && pattern.length <= p) return false;  //模式完了，字符串还有
        //模式串 a*a 没结束，匹配串可结束可不结束
        if (p + 1 < pattern.length && pattern[p + 1] == '*') {
           //当前 pattern 的下一个是*号时
           if (s >= str.length)  //字符串完了
                return matchCore(str, s, pattern, p + 2);
            else {
                if (pattern[p] == str[s] || pattern[p] == '.') {
                    //当前位置匹配完成，移动到下一个模式串
                    return matchCore(str, s + 1, pattern, p + 2)
                           || matchCore(str, s + 1, pattern, p)
                           || matchCore(str, s, pattern, p + 2);
                } else
                    return matchCore(str, s, pattern, p + 2);
            }
        }
        //当前 pattern 的下一个不是*时候
        if (s >= str.length) return false;
        else {
            if (str[s] == pattern[p] || pattern[p] == '.')
                return matchCore(str, s + 1, pattern, p + 1);
        }
```

```
        return false;
    }
}
```

7.2.5 表示数值的字符串

题目描述：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。

```
public class Solution {
    private int inx;
    public boolean isNumeric(char[] str) {
        if (str == null || str.length == 0) {
            return false;
        }
        inx = 0;
        boolean flag = scanInteger(str);
        //判断小数部分
        // 如果出现'.'，接下来是数字的小数部分
        if (inx < str.length && str[inx] == '.') {
            inx++;
            // 下面一行代码用||的原因：
            // 1. 小数可以没有整数部分，例如.123 等于 0.123；
            // 2. 小数点后面可以没有数字，例如 233.等于 233.0；
            // 3. 当然小数点前面和后面可以有数字，例如 233.666
            flag = scanUInteger(str) || flag;     //解释 a,见代码下方
        }
        //如果出现'e'或者'E'，接下来跟着的是数字的指数部分
        if (inx < str.length && (str[inx] == 'e' || str[inx] == 'E')) {
            inx++;
            // 下面一行代码用&&的原因：
            // 1. 当 e 或 E 前面没有数字时，整个字符串不能表示数字，例如.e1、e1；
            // 2. 当 e 或 E 后面没有整数时，整个字符串不能表示数字，例如 12e、12e+5.4
            flag = flag && scanInteger(str);
        }
        return flag && inx >= str.length;
    }
    //判断是否是整数
    //整数的格式可以用[+|-]B 表示，其中 B 为无符号整数
    public boolean scanInteger(char[] str) {
        if (inx < str.length && (str[inx] == '+' || str[inx] == '-')) {
            inx = inx + 1;
        }
        return scanUInteger(str);
    }
    public boolean scanUInteger(char[] str) {   //判断是否是无符号整数
        int inx1 = inx;
```

```
        while (inx < str.length && str[inx] >= '0' && str[inx] <= '9') {
            inx = inx + 1;
        }
        // 当 str 中存在若干 0-9 的数字时，返回 true
        return inx > inx1;
    }
}
```

7.2.6 字符流中第一个不重复的字符

题目描述：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"。

```
public class Solution {
    private int[] occurence = new int[256];
    private int index;
    public Solution() {
        for (int i = 0; i < 256; i++) {
            occurence[i] = -1;
        }
        index = 0;
    }
    //Insert one char from stringstream
    public void Insert(char ch)  {
        if (occurence[ch] == -1)
            occurence[ch] = index;
        else if (occurence[ch] >= 0)
            occurence[ch] = -2;
        index++;
    }
    public char FirstAppearingOnce() {
        char ch = '\0';
        int minIndex = Integer.MAX_VALUE;
        for (int i = 0; i < 256; i++) {
            if (occurence[i] >= 0 && occurence[i] < minIndex) {
                ch = (char) i;
                minIndex = occurence[i];
            }
        }
        if (ch == '\0')
            return '#';
        return ch;
    }
}
```

7.2.7 大数运算

相乘：

```
public static String multiply(String num1, String num2) {
        int m = 0,n = 0, x = 0,y = 0;
        if (num1.charAt(0) == '-') {   //去掉前面的符号位
            num1 = num1.substring(1);
            x++;
        }
        if (num2.charAt(0) == '-') {  //去掉前面的符号位
            num2 = num2.substring(1);
            y++;
        }
        while (num1.charAt(m) == '0' && m < num1.length()-1) { //去掉大数前面的无效数字
            m++;
        }
        num1 = num1.substring(m);
        while (num2.charAt(n) == '0' && n < num2.length()-1) {
            n++;
        }
        num2 = num2.substring(n);

        int length1 = num1.length();
        int length2 = num2.length();
        int[] num = new int[length1 + length2];  //用来存储结果的数组
        for (int i = 0; i < length1; i++) {    //第一个数按位循环
            int n1 = num1.charAt(length1 - 1 - i) - '0';  //得到最低位的数字
            int tmp = 0; //保存进位
            for (int j = 0; j < length2; j++) {  //第二个数按位循环
                int n2 = num2.charAt(length2 - 1 - j) - '0';
                //拿出此时的结果数组里存的数+现在计算的结果数+上一个进位数
                tmp = tmp + num[i + j] + n1 * n2;
                num[i + j] = tmp % 10;   //得到此时结果位的值
                tmp /= 10;   //此时的进位
            }
            num[i + length2] = tmp;  //第一轮结束后，如果有进位，将其放入到更高位
        }
        int i = length1 + length2 - 1;
        while (i > 0 && num[i] == 0) {   //计算最终结果值到底是几位数
            i--;
        }
        StringBuilder result = new StringBuilder();
        while (i >= 0) {   //数组保存的是：12345 ,但其表达的是 54321，五万四千三百二十一。
            result.append(num[i--]);
        }
```

```
        if (x+y == 1) {
           return  "-"+ result.toString();
        } else {
             return result.toString();
        }
    }
```

相加：

```
public static String add1(String num1, String num2){
        int m = 0, n = 0, x = 0, y = 0;
        if (num1.charAt(0) == '-') {   //去掉前面的符号位
            num1 = num1.substring(1);
            x++;
        }
        if (num2.charAt(0) == '-') {
            num2 = num2.substring(1);
            y++;
        }
        while (num1.charAt(m) == '0' && m < num1.length() - 1) {//去掉大数前面的无效数字
            m++;
        }
        num1 = num1.substring(m);
        while (num2.charAt(n) == '0' && n < num2.length() - 1) {
            n++;
        }
        num2 = num2.substring(n);
        if (x + y == 2) { //负数减负数
            return "-"+ add(num1, num2);
        } else if( x + y == 1){ //一正一负
            int isBigger = compare(num1, num2);
            if(x+isBigger == 2){ //大负，小正=负(大-小)
                return "-"+ subtract(num1,num2);
            } else if(x+isBigger ==0){ //小负，大正=(大-小)
                return subtract(num2,num1);
            } else if(y+isBigger==0) //小正，大负= 负(大-小)
                return "-"+ subtract(num2,num1);
            else //大正，小负= (大-小)
                return subtract(num1, num2);
        } else {
            return add(num1, num2);
        }
    }
    //比较两个大正数字符串值得大小
    public static int compare(String data1,String data2){
            if (data1.length() < data2.length()) {
```

```
            return -1;
        }else if (data1.length() > data2.length()) {
            return 1;
        }else{
            if (data1.compareTo(data2) > 0) {
                return 1;
            }else if(data1.compareTo(data2) < 0){
                return -1;
            }
        }
        return 1;
}
    /**
     * 两个正数相加
     * @param num1
     * @param num2
     * @return
     */
    public static String add(String num1, String num2) {
        int length1 = num1.length();
        int length2 = num2.length();
        int len = Math.max(length1, length2);
        int[] num = new int[len + 1]; //用来存储结果的数组
        for (int i = 0; i < len; i++) {  //第一个数按位循环
            int tmp = 0; //保存进位
            int n1 = 0 , n2 = 0;
            if(i < length1)
                n1= num1.charAt(length1 - 1 - i) - '0';  //得到最低位的数字
            if(i < length2)
                n2= num2.charAt(length2 - 1 - i) - '0';  //得到最低位的数字
           //拿出此时的结果数组里存的数+现在计算的结果数+上一个进
            tmp = tmp +  n1 + n2;
            num [i] = tmp % 10;   //得到此时结果位的值
            tmp /= 10;   //此时的进位
        }
        StringBuilder result = new StringBuilder();
        while(len > 0 && num[len] == 0)  len -= 1;
        for (int i = len; i >= 0 ; i--) {
            result.append(num[i]);
        }
        return result.toString();
    }
    /**
     * 两个正数相减，且是大数减小数
```

```
     * @param num1
     * @param num2
     * @return
     */
    public static String subtract(String num1, String num2) {
        int length1 = num1.length();
        int length2 = num2.length();
        int len = Math.max(length1, length2);

        int[] num = new int[len]; //用来存储结果的数组
        for (int i = 0; i < len; i++) {  //第一个数按位循环
            int tmp = 0; //保存进位
            int n1 = 0 , n2 = 0;
            if(i < length1)
                n1= num1.charAt(length1 - 1 - i) - '0';  //得到最低位的数字
            if(i < length2)
                n2= num2.charAt(length2 - 1 - i) - '0';  //得到最低位的数字
            if(tmp + n1 < n2){
                tmp = tmp +  n1 + 10 - n2;
                num [i] = tmp % 10;   //得到此时结果位的值
                tmp = -1;   //此时的进位
            }else{
                tmp = tmp + n1 - n2;
                num [i] = tmp % 10;   //得到此时结果位的值
                tmp = 0;   //此时的进位
            }
        }
        StringBuilder result = new StringBuilder();
        while(len > 0 && num[len-1] == 0)  len -= 1;
        for (int i = len - 1; i >= 0 ; i--) {
            result.append(num[i]);
        }
        return result.toString();
    }
```

7.3 链表

7.3.1 倒序打印链表

题目：输入一个链表，从尾到头打印链表每个节点的值。

```
public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
      Stack<Integer> stack = new Stack<>();
      while (listNode != null) {
            stack.push(listNode.val) ;
            listNode = listNode.next;
      }
```

```
        ArrayList<Integer> list = new ArrayList<>();
        while (!stack.isEmpty()) {
            list.add(stack.pop());
        }
        return list;
    }
```

7.3.2 反转链表

思路：当前节点是 pNode，pPrev 为当前节点的前一节点，pNext 为当前节点的下一节点；
让当前节点从 pPrev->pNode->pNext 变成 pPrev<-pNode pNext

注意：反转之后如果不用 next 节点保存 next1 节点的话，此单链表就此断开了

```
public ListNode ReverseList(ListNode head) {
        if(head==null) return null;
        ListNode pNode = head;
        ListNode newHead = null;
        ListNode pPrev = null, pNext = null;

        while(pNode!=null){
            ListNode pNext = pNode.next;
            if(pNext==null) newHead = pNode;
            pNode.next = pPrev;
            pPrev = pNode;
            pNode = pNext;
        }
        return newHead;
}
```

7.3.3 链表中倒数第 k 个结点

思路：两个指针，先让第一个指针和第二个指针都指向头结点，然后再让第一个到达第 k 个节点。然后两个指针同时往后移动，当第一个结点到达末尾的时候，第二个结点所在位置就是倒数第 k 个节点了。

```
public ListNode FindKthToTail(ListNode head,int k) {
        if(k <= 0 || head == null) return null;
        ListNode result = head;
        int i = 0;
        while(head != null){
            i++;
            head = head.next;
            if(i > k){
                result = result.next;
            }
        }
        if(k > i) return null;
        return result;
    }
```

7.3.4 合并两个排序的链表

```
public ListNode Merge(ListNode list1,ListNode list2) {
        if(list1 == null)  return list2;
        if(list2 == null)  return list1;

       /*  递归版本
       if(list1.val <= list2.val){
            list1.next = Merge(list1.next, list2);
            return list1;
        }else {
            list2.next = Merge(list1, list2.next);
            return list2;
        }
        */

        ListNode mergeHead = null;
        ListNode current = null;
        while(list1!=null && list2!=null){
            if(list1.val <= list2.val){
                if(mergeHead == null){ //第一次进来
                    mergeHead = current = list1;
                }else{
                    current.next = list1;
                    current = current.next;
                }
                list1 = list1.next;
            }else{
                if(mergeHead == null){
                    mergeHead = current = list2;
                }else{
                    current.next = list2;
                    current = current.next;
                }
                list2 = list2.next;
            }
        }
        if(list1 == null) current.next = list2;
        if(list2 == null) current.next = list1;
        return mergeHead;
    }
```

7.3.5 复杂链表的复制

题目：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。

```
public class Solution {
    public RandomListNode Clone(RandomListNode pHead) {
        if(pHead == null) return null;
        RandomListNode currNode = pHead;
        //复制 next 如原来是 A->B->C 变成 A->A'->B->B'->C->C'
        while(currNode != null){
            RandomListNode node = new RandomListNode(currNode.label);
            node.next = currNode.next;
            currNode.next = node;
            currNode = node.next;
        }
        currNode = pHead;
        //复制 random pCur 是原来链表的结点 pCur.next 是复制 pCur 的结点
        while(currNode != null){
            if(currNode.random != null){
                currNode.next.random = currNode.random.next;
            }
            currNode = currNode.next.next;
        }
        //拆分
        RandomListNode pCloneHead = pHead.next;
        RandomListNode tmp;
        pCur = pHead;
        while(pCur.next !=null){
            tmp = pCur.next;
            pCur.next =tmp.next;
            pCur = tmp;
        }
        return pCloneHead;
    }
}
```

7.3.6 两个链表的第一个公共结点

思路：由于是单向链表，从第一个公共节点开始，之后他们所有的节点都是重合的。
相当于 p1 后面接了 pHead2，p2 后面接了 pHead1，如果相同肯定在最后，没有没有相同的，到最后一个节点的 p1=p1.next= null, p2=p2.next= null，相同，返回 null.

```
public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
        ListNode p1 = pHead1;
        ListNode p2 = pHead2;
        while(p1!=p2){
            p1 = (p1==null ? pHead2 : p1.next);
            p2 = (p2==null ? pHead1 : p2.next);
        }
        return p1;
```

```
}
```

7.3.7 链表中环的入口结点

```
public ListNode EntryNodeOfLoop(ListNode pHead) {
        if(pHead == null || pHead.next == null)
            return null;
        ListNode p1 = pHead;
        ListNode p2 = pHead;
        while(p2 != null && p2.next != null ){
            p1 = p1.next;
            p2 = p2.next.next;
            if(p1 == p2){
                p2 = pHead;
                while(p1 != p2){
                    p1 = p1.next;
                    p2 = p2.next;
                }
                if(p1 == p2)
                    return p1;
            }
        }
        return null;
}
```

7.3.8 删除链表中重复的结点

题目描述：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 ListNode deleteDuplication(ListNode pHead) {
        if (pHead == null)  return null;
        ListNode first = new ListNode(-1);//设置一个 trick
        first.next = pHead;
        ListNode pNode = pHead;
        ListNode last = first;
        while (pNode != null && pNode.next != null) {
            if (pNode.val == pNode.next.val) {
                int val = pNode.val;
                while (pNode != null && pNode.val == val)
                    pNode = pNode.next;
                last.next = pNode;
            } else {
                last = pNode;
                pNode = pNode.next;
            }
        }
        return first.next;
```

```
}
```

7.4 树

7.4.1 重建二叉树

题目：输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

```
public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        if(pre==null || in==null || pre.length != in.length || pre.length<=0 || in.length <=0)
            return null;
        TreeNode root = reConstructBinaryTree(pre,0,pre.length-1,in,0,in.length-1);
        return root;
    }

    public TreeNode reConstructBinaryTree(int [] pre,int startPre, int endPre, int [] in, int startIn, int endIn){
        if(startPre > endPre){
            return null;
        }
        //在前序遍历序列中第一个数字是树的根节点的值
        TreeNode root = new TreeNode(pre[startPre]);
        for(int index = startIn; index <= endIn; index++){ //遍历中序序列
            if(in[index] == pre[startPre]){  //找到根节点在中序序列中的位置，递归实现左右子树
              root.left = reConstructBinaryTree(pre, startPre + 1, startPre + index - startIn, in, startIn, index-1);
              root.right = reConstructBinaryTree(pre, startPre + index - startIn + 1, endPre, in, index + 1, endIn);
            }
        }
        return root;
    }
}
```

7.4.2 树的子结构

```
public class Solution {
    public boolean HasSubtree(TreeNode root1,TreeNode root2) {
        boolean result = false;
        //当 Tree1 和 Tree2 都不为 0 的时候，才进行比较。否则直接返回 false
        if (root2 != null && root1 != null) {
            if(root1.val == root2.val){ //如果找到了对应 Tree2 的根节点的点
                result = isSUbTree(root1,root2); //以根节点为起点判断是否包含 Tree2
            }
            if (!result) {  //如果找不到，那么就再去 root 的左儿子当作起点
                result = HasSubtree(root1.left,root2);
```

```
            }
            if (!result) {  //如果还找不到，那么就再去 root 的右儿子当作起点
                result = HasSubtree(root1.right,root2);
            }
        }
        return result;
    }

    private boolean isSUbTree(TreeNode root1, TreeNode root2) {
        if(root2 == null) return true;  // Tree2 已经遍历完了都能对应的上，返回 true
        if(root1 == null) return false;   // Tree2 还没有遍历完，Tree1 却遍历完了。返回 false
       //如果其中有一个点没有对应上，返回 false
       if(root1.val != root2.val) return false;
       //如果根节点对应的上，那么就分别去子节点里面匹配
       return isSUbTree(root1.left, root2.left) &&isSUbTree(root1.right, root2.right);
    }
}
```

7.4.3 二叉树的镜像

思路：先前序遍历这棵树的每个结点，如果遍历到的结点有子结点，就交换它的两个子节点，当交换完所有的非叶子结点的左右子结点之后，就得到了树的镜像。

```
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;
        if(root.left != null)  Mirror(root.left);
        if(root.right != null)   Mirror(root.right);
}
```

7.4.11 对称的二叉树

题目描述：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。

```
boolean isSymmetrical(TreeNode pRoot) {
        return isSymmetrical(pRoot, pRoot);
}
private boolean isSymmetrical(TreeNode pRoot1, TreeNode pRoot2) {
        if(pRoot1 == null && pRoot2 == null)  return true;
        if(pRoot1 == null || pRoot2 == null)  return false;
        if(pRoot1.val != pRoot2.val)   return false;
        return isSymmetrical(pRoot1.left, pRoot2.right)
                && isSymmetrical(pRoot1.right, pRoot2.left);
}
```

7.4.4 层次遍历

```
// depth 为数的深度
public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
        //int depth = 0, count = 0, nextCount = 1;
        ArrayList<Integer> list = new ArrayList<>();
        ArrayList<TreeNode> queue = new LinkedList<>();
        if (root == null){
            return list;
        }
        queue.add(root);
        while (queue.size() != 0) {
            TreeNode temp = queue.remove(0);
            //count++;
            if (temp.left != null){
                queue.add(temp.left);
            }
            if (temp.right != null) {
                queue.add(temp.right);
            }
            /*if(count == nextCount){
                nextCount = queue.size();
                count = 0;
                depth++;
            }*/
            list.add(temp.val);
        }
        //return depth;
        return list;
    }
```

7.4.12 按之字形顺序打印二叉树

```
public ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
    ArrayList<Integer> tmp = new ArrayList<>();
    ArrayList<ArrayList<Integer>> ret = new ArrayList<>();
    if (pRoot == null)     return ret;
    //队列 LinkedList 完成层序遍历
    Stack<TreeNode>[] level = new Stack[] {new Stack<>(), new Stack<>()};
    TreeNode node = null; //当前节点
    int current = 0, next = 1;//now 用于记录遍历的个数，next 表示每层节点数量
    level[current].add(pRoot);//将根节点入队
    while (!level[0].isEmpty() || !level[1].isEmpty()) {
            node = level[current].pop();//出队队头元素并访问
            tmp.add(node.val); //添加到 list
            if(current == 0){
```

```
                if (node.left != null) { //如果当前节点的左节点不为空入队
                    level[next].push(node.left);
                }
                if (node.right != null) { //如果当前节点的右节点不为空，把右节点入队
                    level[next].push(node.right);
                }
            } else {
                if (node.right != null) { //如果当前节点的右节点不为空，把右节点入队
                    level[next].push(node.right);
                }
                if (node.left != null) { //如果当前节点的左节点不为空入队
                    level[next].push(node.left);
                }
            }
            if (level[current].isEmpty() ) {
                ret.add(new ArrayList<Integer>(tmp));
                tmp.clear();
                current = 1 - current;
                next = 1 - next;
            }
        }
        return ret;
    }
```

7.4.13 把二叉树打印成多行

题目描述：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。

```
ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
         ArrayList<Integer> tmp = new ArrayList<>();
         ArrayList<ArrayList<Integer>> ret = new ArrayList<>();
         if(pRoot == null)  return ret;
        LinkedList<TreeNode> queue  = new LinkedList<>(); //队列 LinkedList 完成层序遍历
        TreeNode current = null; //当前节点
        int now = 1, next = 0;//now 用于记录遍历的个数，next 表示每层节点数量
        queue.add(pRoot);//将根节点入队
        while(!queue.isEmpty()) {
            current = queue.poll();//出队队头元素并访问
            now--;
            tmp.add(current.val); //添加到 list
            if(current.left != null) { //如果当前节点的左节点不为空入队
                queue.add(current.left);
                next++;
            }
            if(current.right != null) { //如果当前节点的右节点不为空，把右节点入队
                queue.add(current.right);
                next++;
```

```
            }
            if(now == 0) {
                ret.add(new ArrayList<Integer>(tmp));
                tmp.clear();
                now = next;
                next = 0;
            }
        }
        return ret;
}
```

7.4.5 二叉搜索树的后序遍历序列

题目：输入一个数组，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。
思路：BST 的后序序列的合法序列是：最后一个元素根 x；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，且都是合法的后序序列。

```
public class VerifySquenceOfBST {
    public boolean verifySquenceOfBST(int [] sequence) {
        if(sequence.length == 0) return false;
        return judge(sequence, 0, sequence.length - 1);
    }
    boolean judge(int[] a, int left, int right){
        if(left >= right)   return true;
        int i = left;
        //(left,i)：小于根节点的；[i,right):大于根节点
        for (; i < right; i++) {   //左边都小于根节点：a[right]
            if (a[i] > a[right]) break;
        }
        for(int j = i; j < right; j++) {    //右边都大于根节点：a[right]
            if (a[j] < a[right]) return false;
        }
        return judge(a, left, i - 1) && (judge(a, i, right - 1));
    }
}
```

7.4.14 序列化二叉树

题目描述：请实现两个函数，分别用来序列化和反序列化二叉树

```
public class Solution {
    private final String NULLSTR = "$";
    private final String SPILE = ",";
    int index = -1;
    String Serialize(TreeNode root) {
        if(root == null)   return "";
        StringBuilder sb = new StringBuilder();
        Serialize2(root, sb);
        return sb.toString();
```

```
    }
    private void Serialize2(TreeNode root, StringBuilder sb) {
        if(root == null) {
            sb.append(NULLSTR).append(SPILE);
            return;
        }
        sb.append(root.val);
        sb.append(',');
        Serialize2(root.left, sb);
        Serialize2(root.right, sb);
    }
    TreeNode Deserialize(String str) {
        if(str == null || str.length() == 0)  return null;
        String[] strs = str.split(SPILE);
        return Deserialize2(strs);
    }
    private TreeNode Deserialize2(String[] strs) {
        index++;
        if(!strs[index].equals(NULLSTR)) {
            TreeNode root = new TreeNode(0);
            root.val = Integer.parseInt(strs[index]);
            root.left = Deserialize2(strs);
            root.right = Deserialize2(strs);
            return root;
        }
        return null;
    }
}
```

7.4.6 二叉树中和为某一值的路径

题目：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。

```
public class Solution {
    private ArrayList<ArrayList<Integer>> listAll = new ArrayList<>();
    private ArrayList<Integer> list = new ArrayList<>();

    public ArrayList<ArrayList<Integer>> FindPath(TreeNode root, int target) {
        if (root == null) return listAll;
        list.add(root.val);
        target -= root.val;
        if (target == 0 && root.left == null && root.right == null)
            listAll.add(new ArrayList<Integer>(list));
            //因为 add 添加的是引用，如果不 new 一个的话，后面的操作会更改这个 list
        FindPath(root.left, target);
        FindPath(root.right, target);
        list.remove(list.size() - 1);  //移除最后一个元素啊，深度遍历完一条路径后要回退
```

```
        return listAll;
    }
}
```

7.4.7 二叉搜索树与双向链表

```
public class Convert {
    protected TreeNode pLastNodeInList = null;
    public TreeNode Convert1(TreeNode pRootOfTree){
        TreeNode pLastNodeInList = null; //指向双向链表的为节点
        pLastNodeInList = ConvertNode(pRootOfTree);
        TreeNode pHeadOfList = pLastNodeInList;
        while (pHeadOfList != null && pHeadOfList.left != null){    // 返回头节点
            pHeadOfList = pHeadOfList.left;
        }
        return pHeadOfList;
    }
    private TreeNode ConvertNode(TreeNode pRootOfTree) {
        if(pRootOfTree==null) return null;
        TreeNode pCurrent = pRootOfTree;
        if(pCurrent.left != null)
            pLastNodeInList =  ConvertNode(pCurrent.left);
        pCurrent.left = pLastNodeInList;
        if(pLastNodeInList != null)
            pLastNodeInList.right = pCurrent;
        pLastNodeInList = pCurrent;
        if(pCurrent.right != null)
            pLastNodeInList = ConvertNode(pCurrent.right);
        return pLastNodeInList;
    }
}
```

7.4.8 二叉树的深度

```
// 非递归的见 7.4.4
public int TreeDepth(TreeNode root) {
        if (root == null) return 0;
        int left = TreeDepth(root.left);
        int right = TreeDepth(root.right);
        return left > right ? (left + 1) : (right + 1);
}
```

7.4.9 平衡二叉树

思路：后续遍历时，遍历到一个节点，其左右子树已经遍历依次自底向上判断，每个节点只需要遍历一次。

```
public class Solution {
    private boolean isBalanced=true;
    public boolean IsBalanced_Solution(TreeNode root) {
        IsBalanced(root);
```

```
        return isBalanced;
    }
    public int IsBalanced(TreeNode root){
        if(root == null)  return 0;
        int left = IsBalanced(root.left);
        int right = IsBalanced(root.right);
        if(Math.abs(left-right)>1)    isBalanced=false;
        return right>left ?right+1:left+1;
    }
}
```

7.4.10 二叉树的下一个结点

题目描述：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 TreeLinkNode GetNext(TreeLinkNode pNode) {
        if (pNode == null)  return null;
        TreeLinkNode pNext = null;
        if (pNode.right != null) { //如果有右子树，则找右子树的最左节点
            pNext = pNode.right;
            while (pNext.left != null) {
                pNext = pNext.left;
            }
            return pNext;
        }
        while (pNode.next != null) {  //没右子树，则找第一个当前节点是父节点左孩子的节点
            if (pNode.next.left == pNode) //当前结点是其父节点的左孩子位置，返回父节点
                return pNode.next;
            pNode = pNode.next; //找他的父节点的父节点的父节点
        }
        return null;  //退到了根节点仍没找到，则返回 null
}
```

7.4.15 二叉搜索树的第 k 个结点

题目描述：给定一颗二叉搜索树，请找出其中的第 k 大的结点。

```
public class Solution {
    int index = 0; //计数器
    TreeNode KthNode(TreeNode root, int k) {
        TreeNode node = null;
        if (root != null && k > 0) {
            node = KthNode(root.left, k);
            if (node != null)  return node;
            if (++index == k)  return root;
            node = KthNode(root.right, k);
            if (node != null)   return node;
        }
```

```
        return null;
    }
}
```

7.4.16 数据流中的中位数

题目描述：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。

解题思路：最大/小堆：插入时间复杂度：O(logn)；得到中位数的时间复杂度：O(1)

Java 的 PriorityQueue 是从 JDK1.5 开始提供的新的数据结构接口，默认内部是自然排序，结果为小顶堆，也可以自定义排序器，比如下面反转比较，完成大顶堆。

```
import java.util.Comparator;
import java.util.PriorityQueue;
public class Solution {
    private int count = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 11;
    private PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    private PriorityQueue<Integer> maxHeap =
       new PriorityQueue<Integer>(DEFAULT_INITIAL_CAPACITY, new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2 - o1;
        }
    });
    public void Insert(Integer num) {
        if (count %2 == 0) { //当数据总数为偶数时，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
            maxHeap.offer(num);
            int filteredMaxNum = maxHeap.poll();
            //2.筛选后的【大根堆中的最大元素】进入小根堆
            minHeap.offer(filteredMaxNum);
        } else {
            //当数据总数为奇数时，新加入的元素，应当进入大根堆
            //（注意不是直接进入大根堆，而是经小根堆筛选后取小根堆中最大元素进入大根堆）
            //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.offer(num);
            int filteredMinNum = minHeap.poll();
            //2.筛选后的【小根堆中的最小元素】进入大根堆
            maxHeap.offer(filteredMinNum);
        }
        count++;
    }
    public Double GetMedian() {
        if (count %2 == 0) {
```

```
            return new Double((minHeap.peek() + maxHeap.peek())) / 2;
        } else {
            return new Double(minHeap.peek());
        }
    }
}
```

7.4.17 树中两个节点的最低公共祖先

1 如果是二叉查找树

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null||root==p||root==q)    return root;
        if(root.val>p.val&&root.val>q.val)
            return lowestCommonAncestor(root.left,p,q);
        if(root.val<p.val&&root.val<q.val)
            return lowestCommonAncestor(root.right,p,q);
        else    return root;
    }
}
```

2 如果是普通的树，但是树有父节点：使用链表的公共节点

3 如果只是普通的树

```
public class Solution {
    public TreeNode lowestCommonAncestor2(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || p == null || q == null)  return null;
        List<TreeNode> pathp = new ArrayList<>();
        List<TreeNode> pathq = new ArrayList<>();
        getPath(root, p, pathp);
        getPath(root, q, pathq);
        TreeNode lca = null;
        for (int i = 0; i < pathp.size() && i < pathq.size(); i++) {
            if (pathp.get(i) == pathq.get(i))
                lca = pathp.get(i);
            else
                break;
        }
        return lca;
    }
    private boolean getPath(TreeNode pRoot, TreeNode pNode, List<TreeNode> path) {
        if (pRoot == pNode) return true;
        path.add(pRoot);
        boolean found = false;
        if (pRoot.left != null) {
            if (getPath(pRoot.left, pNode, path))
                found = true;
        }
```

```
        if (pRoot.right != null) {
            if (getPath(pRoot.right, pNode, path))
                found = true;
        }
        if(!found)
            path.remove(path.size() - 1);
        return found;
    }
```

7.5 栈和队列

7.5.1 两个栈实现队列

题目：用两个栈实现队列
思路：入队都在 stack1 进行，出队都在 stack2 进行
入队：直接把元素亚茹 stack1 中
出队：①如果 stack2 不为空，则直接弹出 stack2 中的元素；②如果 Stack2 为空，则将 stack1 中的所有元素倒入 stack2 中，然后弹出 stack2 中的栈顶元素。③若都为空，出队失败

```
public class Stack2Queue {
    Stack<Integer> stack1 = new Stack<Integer>();
    Stack<Integer> stack2 = new Stack<Integer>();
    public void push(int node) {
        stack1.push(node);
    }
    public int pop() {
        if(stack1.empty() && stack2.empty())
            throw new RuntimeException("Queue is empty");
        if(stack2.empty()){
            while(!stack1.empty()){
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }
}
```

7.5.2 包含 min 函数的栈

题目：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。
思路：使用两个栈，一个保存数据，另外一个保存依次入栈最小的数
每次入栈的时候，如果入栈的元素比 min 中的栈顶元素小或等于则入栈，否则不如栈。
每次出栈的时候，如果出栈的元素与 min 中的栈顶元素等于则都出栈，否则不出栈。

```
public class Solution {
    Stack<Integer> data = new Stack<Integer>();
    Stack<Integer> min = new Stack<Integer>();
    Integer temp = null;

    public void push(int node) {
```

```
        if(temp == null || node <= temp) {
            temp = node;
            min.push(node);
        }
        data.push(node);
    }
    public void pop() {
        int num = data.pop();
        int num2 = min.peek();
        if(num == num2)
            min.pop();
    }
    public int top() {
        int num = data.peek();
        return num;
    }
    public int min() {
        int num = min.peek();
        return num;
    }
}
```

7.5.3 栈的压入、弹出序列

题目：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。
思路：借用一个辅助的栈，遍历压栈顺序。先讲第一个放入栈中，然后判断栈顶元素是不是出栈顺序的第一个元素：如果不相等，则继续压栈，直到相等以后开始出栈，出栈一个元素，则将出栈顺序向后移动一位，直到不相等。这样循环等压栈顺序遍历完成，如果辅助栈还不为空，说明弹出序列不是该栈的弹出顺序。

```
public boolean isPopOrder(int [] pushA,int [] popA) {
        if(pushA.length == 0 || popA.length == 0 || pushA.length != popA.length)
            return false;
        Stack<Integer> stack = new Stack<>();
        for(int i = 0,j = 0 ;i < pushA.length;i++){
            stack.push(pushA[i]);
            while(j < popA.length && stack.peek() == popA[j]){
                stack.pop();    //出栈
                j++;   //弹出序列向后一位
            }
        }
        return stack.empty();
}
```

7.5.4 滑动窗口的最大值

题目描述：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]}。

```
import java.util.ArrayDeque;
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> maxInWindows(int [] num, int size) {
        ArrayList<Integer> res = new ArrayList<>();
        if (size == 0) return res;
        int begin;
        ArrayDeque<Integer> q = new ArrayDeque<>();
        for (int i = 0; i < num.length; i++) { //遍历所有元素
            begin = i - size + 1; //找到开始比较的位置
            if (q.isEmpty()) //为空，添加
                q.add(i);
            else if (begin > q.peekFirst()) //first 保留最大的数，判断最大的数是否在范围内
                q.pollFirst();   //不在范围内，删除最大的数
            //q 不为空，且新比较的数较大，删除原有的 last
            while ((!q.isEmpty()) && num[q.peekLast()] <= num[i])
                q.pollLast();
            q.add(i); //添加第二大的数
            if (begin >= 0) //有效结果
                res.add(num[q.peekFirst()]);
        }
        return res;
    }
}
```

7.6 查找

7.6.1 二维数组中的查找

题目：在二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

思路：

先定位第一行最后一个元素，

遇到的数比要找的数大时，向左移动，遇到的数比要找的数小时，向下移动

```
public class Solution {
    public boolean Find(int target, int [][] array) {
        boolean flag = false;
        for(int i = 0, j = array[0].length - 1; i < array.length  && j >= 0; ){
            if(target == array[i][j]) {
                flag = true;
                break;
            } else if(target > array[i][j]){
                ++i;
            }else if(target < array[i][j]){
```

```
                --j;
            }
        }
        return flag;
    }
}
```

7.6.2 旋转数组的最小数字

题目：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。NOTE：给出的所有元素都大于 0，若数组大小为 0，请返回 0。

思路：旋转之后的数组实际上可以划分成两个有序的子数组：前面子数组的大小都大于后面子数组中的元素。注意到实际上最小的元素就是两个子数组的分界线。

思路：

（1）我们用两个指针 left,right 分别指向数组的第一个元素和最后一个元素。按照题目的旋转的规则，**第一个元素应该是大于最后一个元素的（没有重复的元素）**。

（2）找到数组的中间元素。

中间元素大于第一个元素，则中间元素位于前面的递增子数组，此时最小元素位于中间元素的后面。让第一个指针 left 指向中间元素。中间元素小于第一个元素，则中间元素位于后面的递增子数组，此时最小元素位于中间元素的前面。我们可以让**第二个指针 right 指向中间元素**。

（3）按照以上思路，**第一个指针 left 总是指向前面递增数组的元素，第二个指针 right 总是指向后面递增的数组元素。最终第一个指针将指向前面数组的最后一个元素，第二个指针指向后面数组中的第一个元素。也就是说他们将指向两个相邻的元素，而第二个指针指向的刚好是最小的元素，这就是循环的结束条件。**

到目前为止以上思路很耗的解决了没有重复数字的情况。

```
public class MinNumberInRotateArray {
    public int minNumberInRotateArray(int [] array) {
        if(array == null || array.length <= 0)
            return 0;
        int start = 0;
        int end = array.length - 1;
        int mid = start;
        while(array[start] >= array[end]){
            if(end - start == 1){
                mid = end;
                break;
            }
            mid = (start + end) / 2;
            if(array[start] == array[end] && array[mid] == array[start])
                return minNumberInRotateArray(array, start, end );
            if(array[mid] >= array[start])
                start = mid;
            else if(array[mid] <= array[end])
                end = mid;
        }
        return array[mid];
    }
```

```
        private int minNumberInRotateArray(int[] array, int start, int end) {
            int result = array[start];
            for (int i = start + 1; i <= end; i++) {
                if(result > array[i])
                    result = array[i];
            }
            return result;
        }
    }
```

7.7 排序

7.7.1 最小的 K 个数

//解法一：java 中的优先队列是基于堆实现的。

```
public ArrayList<Integer> GetLeastNumbers_Solution(int[] input, int k) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        int length = input.length;
        if(k > length || k == 0){
            return result;
        }
         PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(k, new Comparator<Integer>() {
             @Override
             public int compare(Integer o1, Integer o2) {
                 return o2.compareTo(o1);
             }
         });
         for (int i = 0; i < length; i++) {
             if (maxHeap.size() != k) {
                 maxHeap.offer(input[i]);
             } else if (maxHeap.peek() > input[i]) {
                 Integer temp = maxHeap.poll();
                 temp = null;
                 maxHeap.offer(input[i]);
             }
         }
         for (Integer integer : maxHeap) {
             result.add(integer);
         }
         return result;
}
```

//解法二：O(n)的算法，只有当我们可以修改输入的数组时可用，利用快速排序中的获取分割点位置

```
public int partition(int[] arr, int left, int right) {
    int result = arr[left];
    if (left > right)  return -1;
    while (left < right) {
```

```
            while (left < right && arr[right] >= result) {
                  right--;
            }
            arr[left] = arr[right];
            while (left < right && arr[left] < result) {
                  left++;
            }
            arr[right] = arr[left];
      }
      arr[left] = result;
      return left;
}
public int[] getLeastNumbers(int[] input,int k){
     if(input.length == 0 || k<= 0)  return null;
     int[] output = new int[k];
     int start = 0;
     int end = input.length-1;
     int index = partition(input,start,end);
     while(index != k-1){
        if(index > k-1){
             end = index -1;
             index = partition(input,start ,end);
        } else{
             start = index+1;
             index = partition(input,start ,end);
        }
     }
     for(int i = 0;i<k;i++){
        output[i] = input[i];
     }
        return output;
}
//解法三：自己建堆
public ArrayList<Integer> GetLeastNumbers_Solution(int [] input, int k) {
      ArrayList<Integer> list=new ArrayList<Integer>();
      if(input==null || input.length<=0 || input.length<k)  //检查输入的特殊情况
         return list;
      //构建最大堆
      for(int len=k/2-1; len>=0; len--){
            adjustMaxHeapSort(input,len,k-1);
      }
      //从第 k 个元素开始分别与最大堆的最大值做比较，如果比最大值小，则替换并调整堆。
      int tmp;
      for(int i=k; i<input.length; i++){
```

```
            if(input[i]<input[0]){
                tmp=input[0];
                input[0]=input[i];
                input[i]=tmp;
                adjustMaxHeapSort(input,0,k-1);
            }
        }
        for(int j=0; j<k; j++){
            list.add(input[j]);
        }
        return list;
    }
    public void adjustMaxHeapSort(int[] input, int pos, int length){
        int temp;
        int child;
        for(temp=input[pos]; 2*pos+1<=length; pos=child){
            child=2*pos+1;
            if(child<length && input[child]<input[child+1]){
                child++;
            }
            if(input[child]>temp){
                input[pos]=input[child];
            }else{
                break;
            }
        }
        input[pos]=temp;
    }
```

7.8 递归和循环

7.8.1 斐波那契数列

```
public int Fibonacci(int n) {
        int fibonacciRuselt = 0;
        int fibonacciOne = 0;
        int fibonacciTwo = 1;
        if(n == 0 || n == 1)
            return n;
       for (int i = 2; i <= n; i++) {
           fibonacciRuselt = fibonacciOne + fibonacciTwo;
           fibonacciOne = fibonacciTwo;
           fibonacciTwo = fibonacciRuselt;
       }
       return fibonacciRuselt;
}
```

7.8.2 跳台阶

题目：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。
思路：假定第一次跳的是一阶，那么剩下的是 n-1 个台阶，跳法是 f(n-1);假定第一次跳的是 2 阶，那么剩下的是 n-2 个台阶，跳法是 f(n-2)。可以得出总跳法为: f(n) = f(n-1) + f(n-2) 。

```
public int JumpFloor(int target) {
        int result = 0;
        int one = 1;
        int two = 2;
        if(target == 0 || target == 1 || target == 2)
            return target;
        for (int i = 3; i <= target; i++) {
           result = one + two;
           one = two;
           two = result;
       }
        return result;
}
```

7.8.3 变态跳台阶

题目：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。
思路：
(1) n = 1 时，只有 1 种跳法，f(1) = 1；
(2) n = 2 时，会有两个跳得方式，一次 1 阶或者 2 阶，f(2) = f(2-1) + f(2-2) ；
(3) n = 3 时，第一次跳出 1 阶后面剩下：f(3-1);第一次跳出 2 阶，剩下 f(3-2)；第一次 3 阶，那么剩下 f(3-3)
因此结论是 f(3) = f(3-1)+f(3-2)+f(3-3)
(4) n = n 时，得出结论： f(n) = f(n-1)+f(n-2)+...+f(n-(n-1)) + f(n-n) => f(0) + f(1) + f(2) + f(3) + ... + f(n-1)
(5) 由以上已经是一种结论，但是为了简单，我们可以继续简化：
f(n-1) = f(0) + f(1)+f(2)+f(3) + ... + f((n-1)-1) = f(0) + f(1) + f(2) + f(3) + ... + f(n-2)
f(n) = f(0) + f(1) + f(2) + f(3) + ... + f(n-2) + f(n-1) = f(n-1) + f(n-1)
可以得出： **f(n) = 2*f(n-1)**
(6) 得出最终结论,在 n 阶台阶，一次有 1、2、...n 阶的跳的方式时，总得跳法为：

$$f(n) = \begin{cases} 1, & (n=0) \\ 1, & (n=1) \\ 2*f(n-1), & (n>=2) \end{cases}$$

```
public int JumpFloorII(int target) {
     if(target == 0) return 0;
     return 1<<(target-1);
}
```

7.8.4 矩形覆盖

题目：我们可以用 2*1 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 2*1 的小矩形无重叠地覆盖一个 2*n 的大矩形，总共有多少种方法？
思路：① target <= 0 大矩形为<= 2*0,直接 return 0；② target = 1 大矩形为 2*1，只有一种摆放方法；③ target

= 2 大矩形为 2*2，有两种摆放方法；④ target = n 分为两步考虑：第一次摆放一块 2*1 的小矩阵，则摆放方法总共为 f(target - 1)；第一次摆放一块 1*2 的小矩阵，则摆放方法总共为 f(target-2)

7.9 位运算

7.9.1 二进制中 1 的个数

题目：输入一个整数，输出该数二进制表示中 1 的个数。其中负数用补码表示。

思路：①如果一个整数不为 0，那么这个整数二进制表示中至少有一位是 1。②如果这个整数减 1，则把最右边的 1 变成 0，如果最右边的 1 后还有 0 的话， 0 都变成 1，而它左边所有位都保持不变。③接下来把一个整数和它减去 1 的结果做位与运算，相当于把它最右边的 1 编程 0。④那么一个整数的二进制有多少个 1，就可以进行多少次这样的操作。

```
public int NumberOf1(int n) {
    int count = 0;
    while(n != 0){
        ++count;
        n = n & (n-1);
    }
    return count;
}
```

7.9.2 扑克牌顺子

题目：输入一个数组，判断是否能组成为顺子，注意：0 可以替代任何数字。

思路：思路： 必须满足两个条件

①. 除 0 外没有重复的数；②. max - min < 5

```
public boolean isContinuous(int[] numbers) {
        if (numbers.length != 5) return false;
        int min = 14;
        int max = -1;
        int flag = 0;
        for (int i = 0; i < numbers.length; i++) {
            int number = numbers[i];
            if (number < 0 || number > 13) return false;
            if (number == 0) continue;
            if (((flag >> number) & 1) == 1) return false;
            flag |= (1 << number);  //用二进制位来判断是否有数字重复
            if (number > max) max = number;
            if (number < min) min = number;
            if (max - min >= 5) return false;
        }
        return true;
}
```

7.10 数学

7.10.1 数值的整数次方

题目：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。

思路：举例:10^13 = 10^1101 = 10^0001*10^0100*10^1000。通过&1 和>>1 来逐位读取 1101，为 1 时将该位代表的乘数累乘到最终结果。

```
public class Solution {
    public double Power(double base, int exponent) {
        int exponentAbs =  Math.abs(exponent);
        if(Double.compare(base, 0) == 0 && exponent < 0) {
            throw new RuntimeException("has error"); //base=0，指数为负数，需要对 0 求倒
        }
        double result = PowerWithUnsignedExponent(base, exponentAbs);
        if(exponent < 0) //指数为负数，求倒
            result = 1.0 / result;
        return result;
    }
    public double PowerWithUnsignedExponent(double base, int exponent) {
      //  if(exponent == 0) return 1;
      //  if(exponent == 1) return base;
      //  double result = PowerWithUnsignedExponent(base, exponent>>1);
      //  result *= result;
      //  if((exponent & 0x1) == 1)
          //  result *= base;
      //  return result;
      double result = 1.0;
      while(exponent!=0){
          if((exponent&1)==1)
              result *= base;
          base *= base;  // 翻倍 5^4 = 5^2^(4>>1)
          exponent>>=1;  // 右移一位
      }
      return result;
  }
}
```

7.10.2 1~n 整数中 1 出现的次数

思路：**当 X = 0 时**，最高位中永远是不会包含 0 的，因此，从个位累加到左起第二位就要结束。

例如 n=2593 x=5

①首先是个位。从 1 至 2590 中，包含了 259 个 10，因此任意的 X 都出现了 259 次。最后剩余的三个数 2591，2592 和 2593，因为它们最大的个位数字 3 < X，因此不会包含任何 5。

②然后是十位。从 1 至 2500 中，包含了 25 个 100，因此任意的 X 都出现了 25×10=250 次。剩下的数字是从 2501 至 2593，它们最大的十位数字 9 > X，因此会包含全部 10 个 5。最后总计 250 + 10 = 260。

③接下来是百位。从 1 至 2000 中，包含了 2 个 1000，因此任意的 X 都出现了 2×100=2002×100=200 次。剩下的数字是从 2001 至 2593，它们最大的百位数字 5 == X，这时情况就略微复杂，它们的百位肯定是包含 5 的，但不会包含全部 100 个。如果把百位是 5 的数字列出来，是从 2500 至 2593，数字的个数与百位和十位数字相关，是 93+1 = 94。最后总计 200 + 94 = 294。

④最后是千位。现在已经没有更高位，因此直接看最大的千位数字 2 < X，所以不会包含任何 5。到此为

止，已经计算出全部数字 5 的出现次数。
总结一下以上的算法，可以看到，当计算右数第 ii 位包含的 X 的个数时：

1. 取第 i 位左边（高位）的数字，乘以 10^{i-1}，得到**基础值** a。
2. 取第 i 位数字，计算**修正值**：
    1. 如果大于 X，则结果为 $a+10^{i-1}$。
    2. 如果小于 X，则结果为 a。
    3. 如果等 X，则取第 i 位右边（低位）数字，设为 b，最后结果为 $a+b+1$。

```
public class Solution {
    public int NumberOf1Between1AndN_Solution(int n) {
        return count(n, 1);
    }
    public int count(int n, int x) {
        int count = 0;
        for (int i = 1; i <= n;i *= 10) {
            int high = (n / i) / 10;  // 高位的数字。
            int cur =  (n / i) % 10;  // 当前位的数字。
            if (x == 0) {
                if (high != 0) {
                    high--;
                } else {
                    break;
                }
            }
            count += high * i;
            if (cur > x) {
                count += i;
            } else if (cur == x) {
                count += n -  (n / i) * i + 1;
            }
        }
        return count;
    }
}
```

7.10.3 把数组排成最小的数

题目：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。
思路： 先将整型数组转换成 String 数组，然后将 String 数组排序，最后将排好序的字符串数组拼接出来。关键就是制定排序规则。排序规则如下：
(1)自反性：若有 aa=bb,则 a = b；
(2)对称性：若 a < b，则 ab < ba ；所以若 ab < ba 则 a < b；
(3)传递性

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

```
        int n = numbers.length;
        StringBuilder sb = new StringBuilder();
        ArrayList<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(numbers[i]);
        }
        Collections.sort(list, new Comparator<Integer>() {
            public int compare(Integer str1, Integer str2) {
                String s1 = str1 + "" + str2;
                String s2 = str2 + "" + str1;
                return s1.compareTo(s2);
            }
        });
        for (int j : list) {
            sb.append(j);
        }
        return sb.toString();
    }
```

7.10.4 丑数

题目：把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。习惯上把 1 当做是第一个丑数。求按从小到大的顺序的第 N 个丑数。

思路：把找到的丑数都存起来，然后每次都从丑数中去寻找下一个丑数。

如果 p 是丑数，那么 p=2^x * 3^y * 5^z，那么只要赋予 x,y,z 不同的值就能得到不同的丑数。

对于任何丑数 p：

①那么 2*p,3*p,5*p 都是丑数，并且 2*p<3*p<5*p；②如果 p<q，那么 2*p<2*q,3*p<3*q,5*p<5*q

其实每次我们只用比较 3 个数：用于乘 2 的最小的数、用于乘 3 的最小的数，用于乘 5 的最小的数。

```
int GetUglyNumber_Solution(int index) {
        if(index < 0 )        return 0;
        else  if (index < 7)  return index;
        int[] res = new int[index];
        res[0] = 1;
        int t2 = 0, t3 = 0, t5 = 0;
        for (int i = 1; i < index; ++i){
            res[i] = Math.min(res[t2] * 2,  Math.min(res[t3] * 3, res[t5] * 5));
            if (res[i] == res[t2] * 2) t2++;
            if (res[i] == res[t3] * 3) t3++;
            if (res[i] == res[t5] * 5) t5++;
        }
        return res[index - 1];
}
```

7.10.5 和为 S 的数字

题目：找出所有和为 S 的连续正数序列

思路：考虑使用 small/big 表示序列中最小/大的数.

①首先初始化为：small=1，big=2；②small 到 big 序列和小于 S，big++；大于 S，删除 small,然后 small++；③知道 small = (1+s)/2 结束。

```
public ArrayList<ArrayList<Integer> > FindContinuousSequence(int sum) {
       ArrayList<ArrayList<Integer>> res = new ArrayList<>();
        if (sum < 3)    return res;
        int small = 1, big = 2;
        int middle = (1 + sum) / 2;
        int curSum = small + big;
        while(small < middle ){
            while(curSum > sum && small < middle){
                curSum -= small;
                small++;
            }
            if(curSum == sum) {
                ArrayList<Integer> list = new ArrayList<Integer>();
                for (int i = small; i <= big ; i++) {
                    list.add(i);
                }
                res.add(list);
            }
            big++;
            curSum += big;
        }
        return res;
    }
```

7.10.6 和为 S 的两个数字

题目：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，是的他们的和正好是 S，如果有多对数字的和等于 S，输出两个数的乘积最小的。

```
public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) {
        ArrayList<Integer> res = new ArrayList<>();
        if(array == null || array.length < 1)  return res;
        int start = 0, end = array.length - 1;
        while(end > start){
            int curSum = array[start] + array[end];
            if(curSum == sum){
                res.add(array[start]);
                res.add(array[end]);
                return  res;
            }else if(curSum > sum){
                end--;
            }else
                start++;
        }
        return res;
```

}

7.10.7 N 个骰子的点数

题目：把 N 个骰子扔在堵上，所有骰子抄上一面的点数之和的概率。

思路：N 个，最小为 n,最大为 6N，所以开辟 index=6N 的数组

* 我们需要将中间值存起来以减少递归过程中的重复计算问题，可以考虑我们用两个数组 AB，

* A 在 B 之上得到，B 又在 A 之上再次得到，这样 AB 互相作为对方的中间值

* 我们用一个 flag 来实现数组 AB 的轮换，由于要轮转，我们最好声明一个二维数组，

* 这样的话，如果 flag=0 时，1-flag 用的就是数组 1，如果 flag=1 时，1-flag 用的就是数组 0,

思路：

```
public class PrintProbability {
    private static final int g_maxValue = 6;
    //基于循环求骰子点数
    public void PrintProbability_Solution(int n) {
        if (n < 1) return;
        int probability = g_maxValue * n;
        int[][] pProbabilities = new int[2][probability + 1];
        for (int i = 0; i <= probability; i++) { //初始化数组
            pProbabilities[0][i] = 0;
            pProbabilities[1][i] = 0;
        }
        int flag = 0;
        //当第一次抛掷骰子时，有 6 种可能，每种可能出现一次
        for (int i = 1; i <= g_maxValue; i++) {
            pProbabilities[flag][i] = 1;
        }
        //从第二次开始掷骰子，假设第一个数组中的第 n 个数字表示骰子和为 n 出现的次数，
        //在下一循环中，我们加上一个新骰子，此时和为 n 的骰子出现次数应该等于上一次循环中骰子
        //点数和为 n-1,n-2,n-3,n-4,n-5, n-6 的次数总和，
        //所以我们把另一个数组的第 n 个数字设为前一个数组对应的 n-1,n-2,n-3,n-4,n-5，n-6 之和
        for (int k = 2; k <= n; k++) {
            for (int i = 0; i < k; i++) {//第 k 次掷骰子，小于 k 的情况是不可能发生的!
                pProbabilities[1 - flag][i] = 0;
            }
            for (int i = k; i <= g_maxValue * k; i++) {//和最小为 k，最大为 g_maxValue*k
                pProbabilities[1 - flag][i] = 0;//初始化，因为这个数组要重复使用，清 0
                for (int j = 1; j <= i && j <= g_maxValue; j++) {
                    pProbabilities[1 - flag][i] += pProbabilities[flag][i - j];
                }
            }
            flag = 1 - flag;
        }
        double total = Math.pow(g_maxValue, n);
        for (int i = n; i <= g_maxValue * n; i++) {
            String ratio = "" + pProbabilities[flag][i] + "/" + total;
```

```
            System.out.println("sum: " + i + " ratio: " + ratio);
        }
    }
  }
```

7.10.8 圆圈中最后剩下的数字

问题描述：n 个人（编号 0~(n-1)），从 0 开始报数，报到(m-1)的退出，剩下的人继续从 0 开始报数。求胜利者的编号。

解题思路：约瑟夫环问题。F(n,m)表示，n 个数字中删除第 m 个数字最后剩下的数字。

第一次:在 n 个数字中，被删除的数字是： (m-1)%n，即为 k；

第二次:删除 K 之后剩下的数字，从 K+1 开始，即 K+1->0;K+2->1,…,K-1->n-2;

第 n-1 最后胜利者是编号 X，在 n 也为胜利者编号为：P(x)= (x+K+1)%n = (x+m)%n。

$$\mathbf{F(n,m)}=\begin{cases}\mathbf{0} & \boldsymbol{n=1}\\ \mathbf{p\big(F(n-1,m)\big)=\ (F(n-1,m)+\ K+1)\%n\ =\ (F(n-1,m)+m\ )\%n} & \mathbf{n\ >\ 1}\end{cases}$$

```
public int LastRemaining_Solution(int n, int m) {
        if (n < 1 || m < 1)    return -1;
        int last = 0;
        for (int i = 2; i <= n; i++)    last = (last + m) % i;
        return last;
}
```

7.10.9 求 1+2+3+…+n

题目：求 1+2+3+...+n，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）。

思路：利用&&（逻辑与），的**短路特点**，前面为假，后面不计算。

```
public int Sum_Solution(int n) {
        int ans = n;
        boolean flag = ((ans != 0) && ((ans += Sum_Solution(n - 1)) != 0));
        return ans;
}
```

7.10.12 打印 1~n 位最大数

```
Public class Solution{
    public void printToMax2(int n){
        if(n <= 0) return;
        char[] number = new char[n];
        Arrays.fill(number, '0');
        printOrder(number,n,0);
    }
    public void printOrder(char[] number, int n, int index){
        if(index == n){
            PrintNumber(number);
            return;
        }
        for(int i = 0; i <= 9; i++){
            number[index] = (char)('0' + i);
```

```
            printOrder(number,n,index + 1);
        }
    }
    public void PrintNumber(char[] number){
        boolean isBeginning0 = true;
        int length = number.length;
        for (int i = 0; i < length; i++) {
            if(isBeginning0&&number[i]!='0')
                isBeginning0=false;
            if(!isBeginning0){
                System.out.print(number[i]);
            }
        }
        System.out.println();
    }
}
```

7.10.10 不用加减乘除做加法

题目描述：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。

思路： 首先看十进制是如何做的： 5+17=22，三步走

第一步：相加各位的值，不算进位，得到 12。

第二步：计算进位值，得到 10. 如果这一步的进位值为 0，那么第一步得到的值就是最终结果。

第三步：重复上述两步，只是相加的值变成上述两步的得到的结果 12 和 10，得到 22。

同样我们可以用三步走的方式计算二进制值相加： 5-101，17-10001

第一步：相加各位的值，得到 10100，二进制每位相加就相当于各位做异或操作，101^10001。

第二步：计算进位值，得到 001，再向左移一位得到 0010，(101&10001)<<1。

第三步重复上述两步， 各位相加 010^10100=10110，进位值为 0=(010&10100)<<1。

跳出循环，10110 为最终结果 22。

```
public int Add(int num1,int num2) {
        int sum, carry;
        while( num2 != 0){
            sum = num1 ^ num2;
            carry = (num1 & num2) << 1; // num1
            num1 = sum;
            num2 = carry;
        }
        return num1;
}
```

举一反三：交换两个数

```
public void swap(int a, int b){
        a = a + b;
        b = a - b; //a
        a = a - b;
}
```

```
public void swap(int a, int b){
    a = a ^ b;
    b = a ^ b; //a
    a = a ^ b;
}
```

7.10.11 把字符串转换成整数

题目描述：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。 数值为 0 或者字符串不是一个合法的数值则返回 0。

思路：代码中将所有数据当做负数（正数）来处理，最后处理符号问题；

```
public int StrToInt(String str) {
    return parseInt(str, 10);
}
public int parseInt(String s, int radix) throws NumberFormatException {
        if (s == null) throw new NumberFormatException("null");
        if (radix < Character.MIN_RADIX) {   //2
            throw new NumberFormatException("radix less than Character.MIN_RADIX");
        }
        if (radix > Character.MAX_RADIX) {  //36
            throw new NumberFormatException("radix greater than Character.MAX_RADIX");
        }
        int result = 0;
        boolean negative = false;
        int i = 0, len = s.length();
        int limit = -Integer.MAX_VALUE;
        int multmin;
        int digit;
        if (len > 0) {
            char firstChar = s.charAt(0);
            if (firstChar < '0') { // Possible leading "+" or "-"
                if (firstChar == '-') {
                    negative = true;
                    limit = Integer.MIN_VALUE;
                } else if (firstChar != '+')
                    throw new NumberFormatException("For input string: \"" + s + "\"");
                if (len == 1) // Cannot have lone "+" or "-"
                    throw new NumberFormatException("For input string: \"" + s + "\"");
                i++;
            }
            multmin = limit / radix;
            while (i < len) {
                //根据基数返回当前字符的值的十进制
                digit = Character.digit(s.charAt(i++),radix);
                if (digit < 0)
                    throw new NumberFormatException("For input string: \"" + s + "\"");
                if (result < multmin)
                    throw new NumberFormatException("For input string: \"" + s + "\"");
                result *= radix;
                if (result < limit + digit)
                    throw new NumberFormatException("For input string: \"" + s + "\"");
```

```
                result -= digit;
            }
        } else
            throw new NumberFormatException("For input string: \"" + s + "\"");
        return negative ? result : -result;
}
```

注意：**Java 中 Integer 的 valueOf 方法，-128 到 127 的整数将被缓存。**

```
System.out.println(Integer.valueOf("127")==Integer.valueOf("127"));  // true
System.out.println(Integer.valueOf("128")==Integer.valueOf("128"));  // false
System.out.println(Integer.parseInt("128")==Integer.valueOf("128")); // true
```

7.10.12 末尾 0 的个数

题目描述：输入一个正整数 n,求 n!(即阶乘)末尾有多少个 0？ 比如: n = 10; n! = 3628800,所以答案为 2。

解题思路：**两个大数字相乘，都可以拆分成多个质数相乘，而质数相乘结果尾数为 0 的，只可能是 2*5**。两个数相乘尾数 0 的个数其实就是依赖于 2 和 5 因子的个数。又因为每两个连续数字就会有一个因子 2，个数非常充足，所以此时只需要关心 5 因子的个数就行了。

令 n! = (5*K) * (5*(K-1)) * (5*(K-2)) * ... * 5 * A，其中 A 就是不含 5 因子的数相乘结果，n = 5*K + r（0<= r <= 4）。

假设 f(n!)是计算阶乘 n!尾数 0 的个数，而 g(n!)是计算 n!中 5 因子的个数，那么就会有如下公式：

f(n!) = g(n!) = g(5^K * K! * A) = K + g(K!) = K + f(K!)，其中 K=n / 5（取整数）。

很显然，当 0 <= n <= 4 时，f(n!)=0。结合这两个公式，就搞定了这个问题了。

例如：f(1000!) = 200 + f(200!) = 200 + 40 + f(40!) = 240 + 8 + f(8!) = 248 + 1 + f(1) =249

```
public static int numOf0(int n){
        int result = 0;
        while(n >= 5){
            result += n / 5;
            n = n / 5;
        }
        return result;
}
```

7.10.13 两个大数相乘

注意事项：

2、对公司近况、项目情况有所了解，准备好合适的问题问面试官。

3、项目经验：简短的项目背景（项目的规模，功能，目标用户等）、**完成的任务**（负责：项目的总体框架设计、核心算法、团队合作）、为了完成任务做了哪些工作，怎么做的（基于什么工具什么平台下应用了哪些技术）、**自己的贡献**（完成了多少功能，性能优化提高的百分比）。

Winforms 是微软.NET 中的一个成熟的 UI 平台（**Situation**）。本人的工作是在添加少量新功能之外主要负责维护已有的功能（**Task**）。新的功能主要是让 Winforms 的控件的风格和 Vista、Windows 7 的风格保持一致。在维护方面，对于较难的问题我用 WinDbg 等工具进行调试（**Action**）。在过去两年中我总共修改了超过 200 个 Bug（**Result**）。

项目中碰到的最大的问题是什么，怎么解决的；学到了什么；解决冲突。

4、参与：加入某一个开发团队写了几行代码；
负责：项目的总体框架设计、核心算法、团队合作
了解：只是上过课或者看过书，没有做过实际的项目。【建议少用】
熟悉：在实际项目中使用已经有较长时间，通过查阅相关的文档可以独立解决大部分问题。
精通：轻松回答这个领域里的绝大数问题。
5、为什么跳槽：

笔者在面试的时候，通常给出的答案是：现在的工作做了一段时间，已经没有太多的激情了，因此希望寻找一份更有挑战的工作。然后具体论述为什么有些厌倦现在的职位，以及面试的职位我为什么会有兴趣。

1、**编码前将分析过程，自己的思路，明白要做的是什么，怎么做，可以采用举例子、画图**；
2、位于运算（&1）比求约判断奇偶效率高；
3、(n-1) & n 得到的结果相当于把 n 的二进制表示中的最右边一个 1 变成 0。
4、float、double 不能直接==比较大小
5、大数据可以使用字符串模拟，以免溢出，打印时去掉前面的 0.
6、链表 随时考虑 null 和只有一个节点情况。
7、扩展性的考虑