数据结构

二叉树

相关概念和知识点

二叉树常规遍历

前序遍历：先**访问根节点**，再前序遍历左子树，再前序遍历右子树；

中序遍历：先中序遍历左子树，再**访问根节点**，再中序遍历右子树

后序遍历：先后序遍历左子树，再后序遍历右子树，再**访问根节点**

注意点

- 以根节点访问顺序决定是什么遍历
- 左子树都是优先于右子树

树结构

```
public class TreeNode
{
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode( int x ) { val = x };
}
```

递归遍历

```
private void Traversal( TreeNode root )
{
    if( root == null )
        return;
    System.out.println( root.val );     // 前序
    preOrderTraverse( root.left );
    // System.out.println( root.val );   中序
    preOrderTraverse( root.right );
    // System.out.println( root.val );   后序
}
```

前序非递归

```
// v1
private List<Integer> preOrderTraversal( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    if( root == null )
        return res;

    Deque<TreeNode> stack = new LinkedList<>();
    while( root != null || !stack.isEmpty() )
```

```
    {
        while( root != null )
        {
            // 前序遍历，先保存结果
            res.add( root.val );
            stack.addLast( root );
            root = root.left;
        }
        // pop
        TreeNode node = stack.removeLast();
        root = node.right;
    }
    return res;
}

// v2
private List<Integer> preOrderTraversal( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    if( root == null )
        return res;

    Deque<TreeNode> stack = new LinkedList<>();
    stack.addLast( root );
    while( !stack.isEmpty() )
    {
        TreeNode node = stack.removeLast();
        res.add( node.val );
        if( node.right != null )
            stack.addLast( node.right );
        if( node.left != null )
            stack.addLast( node.left );
    }
    return res;
}
```

中序非递归

```
private List<Integer> inOrderTraversal( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    if( root == null )
        return res;

    Deque<TreeNode> stack = new LinkedList<>();
    while( !stack.isEmpty() || root != null )
    {
        while( root != null )
        {
            stack.addLast( root );
            root = root.left;
        }
        // pop
        TreeNode node = stack.removeLast();
        res.add( node.val );
        root = node.right;
    }
```

```
    return res;
}
```

后序非递归

```
private List<Integer> postOrderTraversal( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    if( root == null )
        return res;

    Deque<TreeNode> stack = new LinkedList<>();
    TreeNode lastVisit = null;
    while( root != null || !stack.isEmpty() )
    {
        while( root != null )
        {
            stack.addLast( root );
            root = root.left;
        }

        TreeNode node = stack.peekLast();
        if( node.right == null || node.right == lastVisit )
        {
            // 根节点必须在其右子节点弹出之后再弹出
            stack.removeLast();
            res.add( node.val );
            lastVisit = node;
        }
        else
            root = node.right;
    }
    return res;
}
```

DFS 深度搜索

UP-TO-DOWN

```
public List<Integer> preOrderTraversal( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    dfs( root, res );
    return res;
}

// v1:深度遍历
private void dfs( TreeNode root, LinkedList<Integer> res )
{
    if( root == null )
        return;
    res.add( root.val );
    dfs( root.left, res );
    dfs( root.right, res );
}
```

BOTTOMUP(DIVIDE AND CONQUER)

```java
public List<Integer> preOrderTraversal( TreeNode root )
{
    List<Integer> res = divideAndConquer( root );
    return res;
}

// 分治法
private List<Integer> divideAndConquer( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    if( root == null )
        return res;

    // divide
    List<Integer> left = divideAndConquer( root.left );
    List<Integer> right = divideAndConquer( root.right );

    // conquer
    res.add( root.val );
    res.addAll( new LinkedList( left ) );
    res.addAll( new LinkedList( right ) );
    return res;
}
```

注意点

DFS 深度搜索（从上到下）和分治法区别：前者一般将最终结果通过指针参数传入，后者一般递归返回结果最后合并

BFS层次遍历

```java
private List<List<Integer>> levelOrder( TreeNode root )
{
    List<List<Integer>> res = new LinkedList<>();
    if( root == null )
        return res;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer( root );
    while( !queue.isEmpty() )
    {
        List<Integer> runner = new LinkedList<>();
        int len = queue.size();
        while( len > 0 )
        {
            TreeNode node = queue.poll();
            runner.add( node.val );
            if( node.left != null )
                queue.offer( node.left );
            if( node.right != null )
                queue.offer( node.right );
            len--;
        }
        res.add( new LinkedList( runner ) );
    }
```

```
    return res;
}
```

分治法应用

适用场景

- 快速排序
- 归并排序
- 二叉树相关问题

分治法模板

- 递归返回条件
- 分段处理
- 合并结果

```
func traversal( root *TreeNode ) ResultType {
    // nil or leaf
    if root == nil {
        // do something and return
    }

    // Divide
    ResultType left = traversal( root.left )
    ResultType right = traversal( root.right )

    // Conquer
    ResultType result = Merge from left to right

    return result
}
```

典型示例1：通过分治法遍历二叉树

```
public List<Integer> preOrderTraversal( TreeNode root )
{
    List<Integer> res = divideAndConquer( root );
    return res;
}

// 分治法
private List<Integer> divideAndConquer( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    if( root == null )
        return res;
    List<Integer> left = divideAndConquer( root.left );
    List<Integer> right = divideAndConquer( root.right );
    res.add( root.val );
    res.addAll( new LinkedList( left ) );
    res.addAll( new LinkedList( right ) );
    return res;
}
```

典型示例2：归并排序

```java
public int[] MergeSort( int[] nums )
{
    return mergeSort( nums, 0, nums.length - 1 );
}

private int[] mergeSort( int[] nums, int start, int end )
{
    if( nums.length <= 1 )
        return nums;

    // divide
    int mid = start + ( end - start ) / 2;
    int[] left = mergeSort( nums, start, mid );
    int[] right = mergeSort( nums, mid + 1, end );
    int[] res = merge( left, right );
    return res;
}

private int[] merge( int[] left, int[] right )
{
    int[] res = new int[left.length + right.length];
    int leftIndex = 0, rightIndex = 0;
    int index = 0;
    while( leftIndex < left.length && rightIndex < right.length )
    {
        if( left[leftIndex] <= right[rightIndex] )
        {
            res[index++] = left[leftIndex];
            leftIndex++;
        }
        else
        {
            res[index++] = right[rightIndex];
            rightIndex++;
        }
    }

    while( leftIndex < left.length )
        res[index++] = left[leftIndex++];
    while( rightIndex < right.length )
        res[index++] = right[rightIndex++]
    return res;
}
```

这里给出官方的 go 代码参考，上面的 java 代码未经过测试

```go
func MergeSort( nums []int ) []int {
    return mergeSort( nums )
}

func mergeSort( nums []int ) []int {
    if( len(nums) < 1 ){
        return nums
    }

```

```
    // 分治法
    mid := len(nums) / 2
    left := mergeSort( nums[:mid] )
    right := mergeSort( nums[mid:] )
    // 合并两段数据
    result := merge( left, right )
    return result
}

func merge( left, right []int ) ( result []int ) {
    // 游标
    l := 0
    r := 0
    // 注意不能越界
    for l < len(left) && r < len(right) {
        if left[l] > right[r] {
            result = append( result, right[r] )
            r++
        } else {
            result = append( result, left[l] )
            l++
        }
    }
    // 剩余部分合并
    result = append( result, left[l:]... )
    result = append( result, right[r:]... )
    return
}
```

典型示例3：快速排序

```
public int[] QuickSort( int[] nums )
{
    quickSort( nums, 0, nums.length - 1 );
    return nums;
}

private void quickSort( int[] nums, int start, int end )
{
    if( start < end )
    {
        int pivot = partition( nums, start, end );
        quickSort( nums, 0, pivot - 1 );
        quickSort( nums, pivot + 1, end );
    }
}

// 单边循环法实现partition()方法
private int partition( int[] nums, int start, int end )
{
    int pivot = nums[start];
    int mark = start;
    for( int i = start + 1; i <= end; i++ )
    {
        if( nums[i] < pivot )
        {
            mark++;
```

```
            int temp = nums[i];
            num[i] = nums[mark];
            nums[mark] = temp;
        }
    }

    nums[start] = nums[mark];
    nums[mark] = pivot;
    return mark;
}
```

题目示例1 `leetcode 104 :二叉树的最大深度`

```
private int maxDepth( TreeNode root )
{
    if( root == null )
        return 0;

    // divide
    int left = maxDepth( root.left );
    int right = maxDepth( root.right );
    return Math.max( left, right ) + 1;
}
```

题目示例2 `leetcode 110 平衡二叉树`

```
public boolean isBalanced( TreeNode root )
{
    if( root == null )
        return true;

    int left = maxDepth( root.left );
    int right = maxDepth( root.right );
    int absDiff = Math.abs( left - right );
    if( absDiff > 1 )
        return false;

    return isBalanced( root.left ) && isBalanced( root.right );
}

private int maxDepth( TreeNode root )
{
    if( root == null )
        return 0;

    // divide
    int left = maxDepth( root.left );
    int right = maxDepth( root.right );
    return Math.max( left, right ) + 1;
}
```

题目示例3: `leetcode 124:二叉树中的最大路径和`

思路：分治法，分为三种情况

- 左子树最大路径和最大

- 右子树最大路径和最大
- 左右子树单向路径加上根节点值之和最大

需要保存两个变量：

- 第一个变量保存子树最大路径和
- 第二个变量保存左右子树单向路径和加上根节点值

  比较两个变量取最大值

```
class Solution
{
    int res = Integer.MIN_VALUE;
    public int maxPathSum( TreeNode root )
    {
        oneSideMax( root );
        return res;
    }

    private int oneSideMax( TreeNode root )
    {
        if( root == null )
            return 0;

        // divide
        int left = oneSideMax( root.left );
        int right = oneSideMax( root.right );

        // conquer
        res = Math.max( res, left + right + root.val );
        return Math.max( left, right ) + root.val;
    }
}
```

题目示例4 `leetcode 236 二叉树的最近公共祖先`

```
private TreeNode lowestCommonAncestor( TreeNode root, TreeNode p, TreeNode
q)
{
    if( root == null || p == root || q == root )
        return root;
    // divide
    TreeNode left = lowestCommonAncestor( root.left, p, q );
    TreeNode right = lowestCommonAncestor( root.right, p, q );
    // conquer
    return left == null? right: right == null ? left:root;
}
```

BFS层次应用

题目示例1： `leetcode 102 二叉树的层序遍历`

```
private List<List<Integer>> levelOrder( TreeNode root )
```

```
{
    List<List<Integer>> res = new LinkedList<>();
    if( root == null )
        return res;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer( root );
    while( !queue.isEmpty() )
    {
        List<Integer> runner = new LinkedList<>();
        int len = queue.size();
        while( len > 0 )
        {
            TreeNode node = queue.poll();
            runner.add( node.val );
            if( node.left != null )
                queue.offer( node.left );
            if( node.right != null )
                queue.offer( node.right );
            len--;
        }
        res.add( new LinkedList( runner ) );
    }
    return res;
}
```

题目示例2： `leetcode 107 二叉树的层次遍历II`

```
private List<List<Integer>> levelOrderBottom( TreeNode root )
{
    List<List<Integer>> res = new LinkedList<>();
    if( root == null )
        return res;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer( root );
    while( !queue.isEmpty() )
    {
        List<Integer> runner = new LinkedList<>();
        int len = queue.size();
        while( len > 0 )
        {
            TreeNode node = queue.poll();
            runner.add( node.val );
            if( node.left != null )
                queue.offer( node.left );
            if( node.right != null )
                queue.offer( node.right );
            len--;
        }
        res.add( 0, new LinkedList( runner ) );
    }
    return res;
}
```

题目示例3： `leetcode 103 二叉树的锯齿形层次遍历`

```java
private List<List<Integer>> zigzagLevelOrder( TreeNode root )
{
    List<List<Integer>> res = new LinkedList<>();
    if( root == null )
        return res;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer( root );
    boolean toggle = false;
    while( !queue.isEmpty() )
    {
        List<Integer> runner = new LinkedList<>();
        int levelLen = queue.size();
        while( levelLen > 0 )
        {
            TreeNode node = queue.poll();
            if( toggle )
                runner.add( 0, node.val );
            else
                runner.add( node.val );

            if( node.left != null )
                queue.offer( node.left );
            if( node.right != null )
                queue.offer( node.right );

            levelLen--;
        }
        res.add( new LinkedList( runner ) );
        toggle = !toggle;
    }
    return res;
}
```

二叉搜索树

题目示例1 `leetcode 98 验证二叉搜索树`

思路1：中序遍历，检查结果列表是否有序

```java
List<Integer> res = new LinkedList<>();

public boolean isValidBST( TreeNode root )
{
    inOrderTraversal( root );
    for( int i = 0; i < res.size() - 1; i++ )
        if( res.get(i) >= res.get(i+1) )
            return false;
    return true;
}

private void inOrderTraversal( TreeNode root )
```

```
{
    if( root == null )
        return;

    inOrderTraversal( root.left );
    res.add( root.val );
    inOrderTraversal( root.right );
}
```

思路2：分治法，判断左MAX < 根 < 右MIN

```
public boolean isValidBST( TreeNode root )
{
    return isValidBST( root, null, null );
}

private boolean isValidBST( TreeNode root, TreeNode min, TreeNode max )
{
    if( root == null )
        return true;
    if( min != null && root.val <= min.val )
        return false;
    if( max != null && root.val >= max.val )
        return false;

    return isValidBST( root.left, min, root ) && isValidBST( root.right,
root, max );
}
```

题目示例2 `leetcode 701 二叉搜索树中的插入操作`

```
public TreeNode insertIntoBST(TreeNode root, int val)
{
    if( root == null )
        return new TreeNode( val );
    else if( val < root.val )
        root.left = insertIntoBST( root.left, val );
    else if( val > root.val )
        root.right = insertIntoBST( root.right, val );
    else
        root.val = val;
    return root;
}
```

题目示例3 `leetcode450 删除二叉搜索树`

```
public TreeNode deleteNode( TreeNode node, int key )
{
    if( root == null )
        return root;
    if( key < root.val )
        root.left = deleteNode( root.left, key );
    else if( key > root.val )
        root.right = deleteNode( root.right, key );
```

```
    else if( key == root.val )
    {
        if( root.left == null )
            return root.right;
        else if( root.right == null )
            return root.left;
        else
        {
            TreeNode t = root;
            root = min( t.right );
            root.right = deleteMin( t.right );
            root.left = t.left;
        }
    }
    return root;
}

private TreeNode min( TreeNode root )
{
    if( root == null || root.left == null )
        return root;
    return min( root.left );
}

private TreeNode deleteMin(  TreeNode root )
{
    if( root == null )
        return root;
    if( root.left == null )
        return root.right;
    root.left = deleteMin( root.left );
    return root;
}
```

题目示例4 `leetcode 669 修剪二叉搜索树`

```
private TreeNote trimBST( TreeNode root, int L, int R )
{
    if( root == null )      return null;
    if( root.val < L )      return trimBST( root.right, L, R );
    if( root.val > R )      return trimBST( root.left, L, R );
    root.left = trimBST( root.left, L, R );
    root.right = trimBST( root.right, L, R );
    return root;
}
```

题目示例5 `leetcode 230 二叉搜索树中第K小的元素`

```
public int kthSmallest( TreeNode root, int k )
{
    return select( root, k );
}

private int select( TreeNode root, int k )
```

```
{
    if( root == null )  return 0;
    int t = size( root.left );
    if( t >= k )                    return select( root.left, k );
    else if( t < k - 1 )             return select( root.right, k - t - 1
);
    else                            return root.val;
}
private int size( TreeNode root )
{
    if( root == null )
        return 0;

    return size( root.left ) + size( root.right ) + 1;
}
```

题目示例6 `leetcode 538 把二叉搜索树转换为累加树`

```

```

题目示例7 `leetcode 235 二叉查找树的最近公共祖先`

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

题目示例8 `leetcode 108 将有序数组转换为二叉搜索树`

```
public TreeNode sortedArrayToBST(int[] nums)
{
    return constructTreee( nums, 0, nums.length - 1 );
}

private TreeNode constructTreee( int[] nums, int start, int end )
{
    if( nums == null || nums.length == 0 )
        return null;

    int mid = start + ( end - start ) / 2;
    TreeNode root = new TreeNode( nums[mid] );
    if( mid - 1 >= start )
        root.left = constructTreee( nums, start, mid - 1 );
    if( mid + 1 <= end )
        root.right = constructTreee( nums, mid + 1, end );
    return root;
}
```

题目示例9 `leetcode 109 有序链表转换二叉搜索树`

```
public TreeNode sortedListToBST(ListNode head)
{
    if( head == null )      return null;
    if( head.next == null )  return new TreeNode( head.val );
    ListNode preMid = preMid( head );
    ListNode mid = preMid.next;
    preMid.next = null;
    TreeNode root = new TreeNode( mid.val );
    root.left = sortedListToBST( head );
    root.right = sortedListToBST( mid.next );
    return root;
}

private ListNode preMid( ListNode head )
{
    ListNode slow = head;
    ListNode fast = head.next;
    ListNode pre = head;
    while( fast != null && fast.next != null )
    {
        pre = slow;
        slow = slow.next;
        fast = fast.next.next;
    }
    return pre;
}
```

题目示例10 `leetcode 653两数之和IV 输入BST`

也有前缀和的思想

```
public boolean findTarget( TreeNode root, int k )
{
    return find( root, new HashSet<Integer>(), k );
}

private boolean find( TreeNode root, Set<Integer> set, int k )
{
    if( root == null )
        return false;
    if( set.contains( k - root.val ) )
        return true;

    set.add( root.val );
    return find( root.left, set, k ) || find( root.right, set, k );
}
```

题目示例 11 `leetcode 530 二叉搜索树的最小绝对差`

```
int minDiff = Integer.MAX_VALUE;
TreeNode pre = null;
```

```java
public int getMinimumDifference(TreeNode root)
{
    traversal( root );
    return minDiff;
}

private void traversal( TreeNode root )
{
    if( root == null )
        return;
    traversal( root.left );
    if( pre != null )
        minDiff = Math.min( minDiff, root.val - pre.val );
    pre = root;
    traversal( root.right );
}
```

链表

基本技能

链表相关的核心点

- null/nil异常处理
- dummy node 哑巴节点
- 快慢指针
- 插入一个节点到排序链表
- 从一个链表中移除一个节点
- 翻转链表
- 合并两个链表
- 找到链表的中间节点

常见题型

题目示例1 `leetcode 83 删除排序链表中的重复元素`

给定一个排序链表，删除所有重复元素，使得每个元素只出现一次

```java
private ListNode deleteDuplicates( ListNode head )
{
    ListNode current = head;
    while( current != null )
    {
        while( current.next != null && current.val == current.next.val )
            current.next = current.next.next;
        current = current.next;
    }
    return head;
}
```

题目示例2 `leetcode 82 删除排序链表中的重复元素II`

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现的数字。

```
private ListNode deleteDuplicates( ListNode head )
{
    if( head == null )
        return head;

    ListNode dummy = new ListNode( 0 );
    dummy.next = head;
    head = dummy;

    // head指向的结点意义：满足题目条件的链表（的一部分）的尾结点
    int removeVal;
    while( head.next != null && head.next.next != null )
    {
        if( head.next.val == head.next.next.val )
        {
            removeVal = head.next.val;
            while( head.next != null && head.next.val == removeVal )
                head.next = head.next.next;
        }
        else
            head = head.next;
    }
    return dummy.next;
}
```

题目示例3 `leetcode 206 反转链表`

```
private ListNode reverseList( ListNode head )
{
    ListNode pre = null;
    ListNode cur = head;
    while( cur != null )
    {
        ListNode next = cur.next;
        cur.next = pre;
        pre = cur;
        cur = next;
    }
    return pre;
}
```

题目示例4 `leetcode 92 反转链表II`

递归版本

```

```

```
private ListNode reverseBetween( ListNode head, int m, int n )
{
    if( head == null )
```

```
        return head;

    ListNode dummy = new ListNode( 0 );
    dummy.next = head;
    head = dummy;
    ListNode pre = null;                          // pre固定指向被反转部分最左侧边界外第一个结点
    for( int i = 0; i < m; i++ )
    {
        pre = head;
        head = head.next;
    }

    ListNode next = null;
    ListNode mid = head;                          // mid固定指向被反转部分最左侧结点
    for( int j = i; head != null && j <= n ; j++ )
    {
        ListNode temp = head.next;
        head.next = next;
        next = head;
        head = temp;                              // head固定指向已被反转部分最右侧边界外第一个结点
    }
    pre.next = next;
    mid.next = head;
    return dummy.next;
}
```

题目示例 5 `leetcode 21 合并两个有序链表`

思路：通过dummy node，连接各个元素

```
private ListNode mergeTwoLists( ListNode l1, ListNode l2 )
{
    ListNode dummy = new ListNode( 0 );
    ListNode runner = dummy;
    while( l1 != null && l2 != null )
    {
        if( l1.val < l2.val )
        {
            runner.next = l1;
            l1 = l1.next;
        }
        else
        {
            runner.next = l2;
            l2 = l2.next;
        }
        runner = runner.next;
    }

    if( l1 != null )
        runner.next = l1;
    if( l2 != null )
        runner.next = l2;
    return dummy.next;
```

```
}
```

题目示例 6 leetcode 86 分隔链表

思路：将大于 x 的节点，放到另外一个链表，最后连接这两个链表

```
private ListNode partition( ListNode head, int x )
{
    if( head == null )
        return head;

    ListNode headDummy = new ListNode( 0 );
    ListNode tailDummy = new ListNode( 0 );
    ListNode tail = tailDummy;
    headDummy.next = head;
    head = headDummy;

    while( head.next != null )
    {
        if( head.next.val < x )
            head = head.next;
        else
        {
            ListNode node = head.next;
            head.next = head.next.next;
            tail.next = node;
            tail = tail.next;
        }
    }
    tail.next = null;
    head.next = tailDummy.next;
    return headDummy.next;
}
```

哑巴结点使用场景：当头结点不确定的时候，使用哑巴结点

题目示例6 `leetcode 148 排序链表`

思路：归并排序，找中点和合并操作

```
public ListNode sortList( ListNode head )
{
    return mergeSort( head );
}

private ListNode findMiddle( ListNode head )
{
    ListNode slow = head;
    ListNode fast = head.next;
    while( fast != null && fast.next != null )
    {
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow;
}

```

```
private ListNode mergeTwoLists( ListNode l1, ListNode l2 )
{
    ListNode dummy = new ListNode( 0 );
    ListNode head = dummy;
    while( l1 != null && l2 != null )
    {
        if( l1.val < l2.val )
        {
            head.next = l1;
            l1 = l1.next;
        }
        else
        {
            head.next = l2;
            l2 = l2.next;
        }
        head = head.next;
    }

    if( l1 != null )
        head.next = l1;
    if( l2 != null )
        head.next = l2;
    return dummy.next;
}

private ListNode mergeSort( ListNode head )
{
    if( head == null || head.next == null )
        return head;

    // find middle
    ListNode middle = findMiddle( head );
    // 断开中间结点
    ListNode tail = middle.next;
    middle.next = null;

    ListNode left = mergeSort( head );
    ListNode right = mergeSort( tail );
    ListNode res = mergeTwoLists( left, right );
    return res;
}
```

题目示例7 `leetcode 143 重排链表`

思路：找到中点断开，翻转后面部分，然后合并两个部分

```
public void reorderList( ListNode head )
{
    if( head == null )
        return;
    ListNode middle = findMiddle( head );
    ListNode tail = reverseList( middle.next );
    middle.next = null;
    head = mergeTwoLists( head, tail );
}

```

```
private ListNode findMiddle( ListNode head )
{
    ListNode slow = head;
    ListNode fast = head.next;
    while( fast != null && fast.next != null )
    {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}

private ListNode mergeTwoLists( ListNode l1, ListNode l2 )
{
    ListNode dummy = new ListNode( 0 );
    ListNode head = dummy;
    boolean toggle = true;
    while( l1 != null && l2 != null )
    {
        if( toggle )
        {
            head.next = l1;
            l1 = l1.next;
        }
        else
        {
            head.next = l2;
            l2 = l2.next;
        }
        toggle = !toggle;
        head = head.next;
    }

    if( l1 != null )
        head.next = l1;
    if( l2 != null )
        head.next = l2;
    return dummy.next;
}

private ListNode reverseList( ListNode head )
{
    ListNode pre = null;
    while( head != null )
    {
        ListNode next = head.next;
        head.next = pre;
        pre = head;
        head = next;
    }
    return pre;
}
```

题目示例8 `leetcode 141 环形链表`

快慢指针

```
```

```
private boolean hasCycle( ListNode head )
{
    if( head == null )
        return false;

    ListNode slow = head;
    ListNode fast = head.next;
    while( fast != null && fast.next.next != null )
    {
        if( fast == slow )
            return true
        slow = slow.next;
        fast = fast.next.next;
    }
    return false;
}
```

题目示例9 `leetcode 142 环形链表II`

思路：快慢指针，相遇之后，慢指针回到链表头部，快慢指针以同样步调移动，相遇点即为入环第一个结点

```
private ListNode detectCycle( ListNode head )
{
    if( head == null )
        return head;

    ListNode slow = head;
    ListNode fast = head;
    while( fast != null && fast.next != null )
    {
        if( fast == slow )
        {
            slow = head;
            fast = fast.next;
            while( fast != slow )
            {
                fast = fast.next;
                slow = slow.next;
            }
            return slow;
        }
        fast = fast.next.next;
        slow = slow.next;
    }
    return null;
}
```

题目示例 10 `leetcode 234 回文链表`

```
private boolean isPalindrome( ListNode head )
{
    if( head == null )
        return true;

    ListNode slow = head;
```

```
    // fast 如果初始化为head.next,则中点在slow.next
    // fast 初始化为head,则中点在slow
    ListNode fast = head.next;
    while( fast != null && fast.next != null )
    {
        slow = slow.next;
        fast = fast.next.next;
    }

    ListNode tail = reverseList( slow.next );
    slow.next = null;
    while( head != null && tail != null )
    {
        if( head.val != tail.val )
            return false;
        head = head.next;
        tail = tail.next;
    }
    return true;
}

private ListNode reverseList( ListNode head )
{
    if( head == null )
        return head;

    ListNode pre = null;
    while( head != null )
    {
        ListNode next = head.next;
        head.next = pre;
        pre = head;
        head = next;
    }
    return pre;
}
```

题目示例11 `leetcode 138 复制带随机指针的链表`

```
private Node copyRandomList( Node head )
{
    if( head == null )
        return head;

    // 复制节点，紧挨在原节点之后
    Node cur = head;
    while( cur != null )
    {
        Node node = new Node( cur.val );
        node.next = cur.next;
        cur.next = node;
        cur = node.next;
    }

    // 处理random指针
    cur = head;
```

```
    while( cur != null )
    {
        if( cur.random != null )
            cur.next.random = cur.random.next;
        cur = cur.next.next;
    }

    // 分离两个链表
    cur = head;
    Node cloneHead = cur.next;
    while( cur != null && cur.next != null )
    {
        Node temp = cur.next;
        cur.next = cur.next.next;
        cur = temp;
    }
    return cloneHead;
}
```

题目示例 12 `leetcode 876 链表的中间结点`

```
private ListNode middleNode( ListNode head )
{
    if( head == null || head.next == null )
        return head;

    ListNode slow = head, fast = head;
    while( fast != null && fast.next != null )
    {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```

栈和队列

栈

题目示例1 `leetcode 155 最小栈`

```
class MinStack
{
    Stack<Integer> dataS;
    Stack<Integer> minS;
    /** initialize your data structure here. */
    public MinStack()
    {
        dataS = new Stack<>();
        minS = new Stack<>();
    }

    public void push(int x)
```

```
    {
        dataS.push( x );
        if( minS.size() == 0 )
            minS.push( x );
        else
        {
            int min = minS.peek() < x ? minS.peek():x;
            minS.push( min );
        }
    }

    public void pop()
    {
        dataS.pop();
        minS.pop();
    }

    public int top()
    {
        return dataS.peek();
    }

    public int getMin()
    {
        return minS.peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

题目示例2 `leetcode 150 逆波兰表达式求值`

```
private int evalRPN( String[] tokens )
{
    if( tokens == null || tokens.length == 0 )
        return 0;

    Stack<Integer> stack = new Stack<>();
    for( int i = 0; i < tokens.length; i++ )
    {
        switch( tokens[i] )
        {
            case "+", "-", "*", "/":
                {
                    if( stack.size() < 2 )
                        return 0;

                    int b = stack.pop();
                    int a = stack.pop();
                    int res = 0;
```

```
                    switch( tokens[i] )
                    {
                        case "+":
                            res = a + b;
                            break;
                        case "-":
                            res = a - b;
                            break;
                        case "*":
                            res = a * b;
                            break;
                        case "/":
                            res = a / b;
                            break;
                    }
                    stack.push( res );
                    break;
                }
            default:
                {
                    int val = Integer.parseInt( tokens[i] );
                    stack.push( val );
                    break;
                }
        }
    }
    return stack.peek();
}
```

题目示例3 `leetcode 394 字符串解码`

```
public String decodeString(String s)
{
    StringBuilder res = new StringBuilder();
    int multi = 0;
    LinkedList<Integer> stack_multi = new LinkedList<>();
    LinkedList<String> stack_res = new LinkedList<>();
    for(Character c : s.toCharArray())
    {
        if(c == '[')
        {
            stack_multi.addLast(multi);
            stack_res.addLast(res.toString());
            multi = 0;
            res = new StringBuilder();
        }
        else if(c == ']')
        {
            StringBuilder tmp = new StringBuilder();
            int cur_multi = stack_multi.removeLast();
            for(int i = 0; i < cur_multi; i++) tmp.append(res);
            res = new StringBuilder(stack_res.removeLast() + tmp);
        }
        else if(c >= '0' && c <= '9') multi = multi * 10 +
Integer.parseInt(c + "");
        else res.append(c);
    }
```

```
    return res.toString();
}
```

栈和队列的特殊应用：单调栈/单调队列

单调栈：栈中存放的数据都是有序的，元素的分布从栈底到栈顶具有单调性，分为单调递增栈和单调递减栈两种

1. **单调递增栈就是元素的值由栈底到栈顶大小单调递增**
2. **单调递减栈就是元素的值由栈底到栈顶大小单调递减**

单调栈里可以保存元素的值或者数组下标

某些场景下栈底也需要维护，此时可能需要借助队列或双端队列实现，此时称为单调队列

单调栈主要回答的几种问题

- 比当前元素更大的下一个元素
- 比当前元素更大的前一个元素
- 比当前元素更小的下一个元素
- 比当前元素更小的前一个元素

根据题目大小变化元素的遍历顺序和不等号的方向即可

```
// 一个简单的单调栈模板
Deque<Integer> stack = new LinkedList<>();
for( int i = 0; i < nums.length; i++ )
{
    while( !stack.isEmpty() && nums[i] <= nums[stack.peekLast()] ) // 单调递增栈
        // 单调递减栈 nums[i] >= nums[stack.peekLast()]
        stack.removeLast();
    stack.addLast( i );
}
```

题目示例 1 `leetcode 496 下一个更大元素 I`

寻找比当前元素更大的下一个元素

```
// v1，从右往左构建一个单调递减栈
private int[] nextGreaterElement( int[] nums1, int[] nums2 )
{
    int[] res = new int[nums1.length];
    int[] temp = new int[nums2.length];
    Deque<Integer> stack = new LinkedList<>();
    for( int i = nums2.length - 1; i >= 0; i-- )
    {
        while( !stack.isEmpty() && nums2[i] >= stack.peekLast() )
            stack.removeLast();
        temp[i] = stack.isEmpty()? -1:stack.peekLast();
        stack.addLast( nums2[i] );
    }
```

```
14
15      for( int i = 0; i < nums1.length; i++ )
16          for( int j = 0; j < nums2.length; j++ )
17              if( nums2[j] == nums1[i] )
18                  res[i] = temp[j];
19      return res;
20  }
21
22
23  // v2，从左往右构建一个单调递减栈
24  private int[] nextGreaterElement( int[] nums1, int[] nums2 )
25  {
26      Deque<Integer> stack = new LinkedList<>();
27      HashMap<Integer, Integer> hashmap = new HashMap<>();
28      int[] res = new int[nums1.length];
29      for( int i = 0; i < nums2.length; i++ )
30      {
31          while( !stack.isEmpty() && nums2[i] > stack.peekLast() )
32              hashmap.put( stack.removeLast(), nums2[i] );
33          stack.addLast( nums2[i] );
34      }
35      while( !stack.isEmpty() )
36          hashmap.put( stack.removeLast(), - 1 );
37      for( int i = 0; i < nums1.length; i++ )
38          res[i] = hashmap.get( nums1[i] );
39      return res;
40  }
```

题目示例2 `leetcode 503 下一个更大元素II`

```
1   // 从右往左构建一个单调递减栈
2   private int[] nextGreaterElements( int[] nums )
3   {
4       int n = nums.length;
5       Deque<Integer> stack = new LinkedList<>();
6       int[] res = new int[n];
7       for( int i = 2 * n - 1; i >= 0; i-- )
8       {
9           while( !stack.isEmpty() && nums[i%n] >= s.peekLast() )
10              stack.removeLast();
11          res[i%n] = stack.isEmpty()? -1:stack.peekLast();
12          stack.addLast( nums[i%n] );
13      }
14      return res;
15  }
```

题目示例3 `leetcode 739 每日温度`

```
1   // 从右往左构建一个单调递减栈
2   private int[] dailyTemperatures( int[] T )
3   {
4       int[] res = new int[T.length];
5       Deque<Integer> stack = new LinkedList<>();
6       for( int i = T.length - 1; i >= 0; i-- )
```

```
    {
        while( !stack.isEmpty() && T[i] >= T[stack.peekLast()] )
            stack.removeLast();
        res[i] = stack.isEmpty() ? 0:stack.peekLast() - i;
        stack.addLast( i );
    }
    return res;
}
```

题目示例 4 `leetcode 962 最大宽度坡`

```
// 从左往右构建一个单调递减栈,这个单调递减栈是一个全局性的单调递减栈
// 其最终存储的结果是从数组整体来看的一个全局性的递减序列
private int maxWidthGap( int[] A )
{
    Deque<Integer> stack = new LinkedList<>();
    stack.addLast(0);
    // 构建单调栈的过程不做其他操作，因为要获得全局的一个单调结果
    for( int i = 0; i < A.length; i++ )
        if( A[i] <= A[stack.peekLast()] )
            stack.addLast( i );
    // 贪心策略，从最远的地方开始往回找
    int maxGap = 0;
    for( int i = A.length - 1; i >= 0; i-- )
        while( !stack.isEmpty() && A[i] >= A[stack.peekLast()] )
            maxGap = Math.max( maxGap, i - stack.removeLast() );
    return maxGap;
}
```

题目示例5 `leetcode 42 接雨水`

```
private int trap( int[] height )
{
    if( height == null || height.length == 0 )
        return 0;

    Deque<Integer> stack = new LinkedList<>();
    int res = 0;

    for( int i = 0; i < height.length; i++ )
    {
        while( !stack.isEmpty() && height[stack.peekLast()] < height[i] )
        {
            int bottomIndex = stack.removeLast();
            // 栈顶元素与bottom相等时应该pop出栈，因为无法形成蓄水的凹槽
            while( !stack.isEmpty() && height[stack.peekLast()] ==
height[bottomIndex] )
                stack.removeLast();
            if( !stack.isEmpty() )
            {
                // leftEdge指向蓄水凹槽的左侧边界
                // 蓄水凹槽的有边界即为i
                int leftEdge = stack.peekLast();
```

```
                res += ( Math.min( height[leftEdge], height[i] ) -
height[bottomIndex] ) * ( i - leftEdge - 1 );
            }
        }
        stack.addLast( i );
    }
    return res;
}
```

题目示例 6 `leetcode 84 柱状图中最大的矩形`

以当前遍历到的柱子`i`的高度height作为矩形的高，矩形的宽度边界为向左找到第一个高度小于当前柱体`i`的柱体`left_i`，向右找到第一个高度小于当前柱体`i`的柱体`right_i`,矩形面积可以表示为`height * ( right_i - left_i - 1 )`

从左到右构建一个单调递增的栈，对于一个栈顶元素而言，下一个可以入栈的元素就是它右边第一个小于它的元素，在栈中栈顶元素的下一个元素就是它左边第一个小于它的元素

```
private int largestRectangleArea( int[] heights )
{
    int[] temp = new int[heights.length+2];
    System.arraycopy( heights, 0, temp, 1, heights.length );

    Deque<Integer> stack = new LinkedList<>();
    int maxArea = 0;
    for( int i = 0; i < temp.length; i++ )
    {
        while( !stack.isEmpty() && temp[i] < temp[stack.peekLast()] )
        {
            int height = temp[stack.removeLast()];
            maxArea = Math.max( maxArea, height * ( i - stack.peekLast() -
1 ) );
        }
        stack.addLast( i );
    }
    return maxArea;
}
```

题目示例7 `leetcode 239 滑动窗口最大值`

题目示例8 `leetcode 85 最大矩形`

题目示例9 `leetcode 402 移掉K位数字`

题目示例10 `leetcode 768 最多能完成排序的块II`

题目示例11 `leetcode 901 股票价格跨度`

题目示例12 `leetcode 1019 链表的下一个更大结点`

题目示例13 `leetcode 1124 表现良好的最长时间段`

```
private int longestWPI( int[] hours )
{
    // 计算前缀和
```

```
    int[] preSum = new int[hours.length+1];
    for( int i = 0; i < hours.length; i++ )
    {
        if( housr[i] > 8 )  preSum[i+1] = preSum[i] + 1;
        else                preSum[i+1] = preSum[i] - 1;
    }

    // 构建单调栈
    Deque<Integer> stack = new LinkedList<>();
    stack.addLast( 0 );
    for( int i = 1; i < preSum.length; i++ )
        if( preSum[i] < preSum[stack.peekLast()] )
            stack.addLast( i );

    // 从右向左利用贪心策略求最大跨度
    int maxL = 0;
    for( int i = preSum.length - 1; i >= 0; i-- )
        while( !stack.isEmpty() && preSum[i] > preSum[stack.peekLast()] )
            maxL = Math.max( maxL, i - stack.removeLast() );
    return maxL;
}
```

题目示例14 `leetcode 907 子数组的最小值之和`

利用栈进行DFS递归搜索模板

```
boolean DFS( Node root, int target )
{
    Set<Node> visited;
    Stack<Node> s;
    add root to s;
    while( s is not empty )
    {
        Node cur = the top element in s;
        return true if cur is target;
        for( Node next:the neighbors of cur )
        {
            if( next is not visited )
            {
                add next to s;
                add next to visited
            }
        }
        remove cur from s;
    }
    return false;
}
```

题目示例5 `leetcode 94 二叉树的中序遍历`

```java
private List<Integer> inorderTraversal( TreeNode root )
{
    List<Integer> res = new LinkedList<>();
    if( root == null )
        return res;

    Stack<TreeNode> stack = new Stack<>();
    while( !stack.isEmpty() || root != null )
    {
        while( root != null )
        {
            stack.push( root );
            root = root.left;
        }
        TreeNode node = stack.pop();
        res.add( node.val );
        root = node.right;
    }
    return res;
}
```

题目示例6 `leetcode 133 克隆图`

```java
public Node cloneGraph( Node node )
{
    HashMap<Node, Node> visited = new HashMap<>();
    return clone( node, visited );
}

private Node clone( Node node, HashMap<Node, Node> visited )
{
    if( node == null )
        return null;

    if( visited.get( node ) != null )
        return visited.get( node );

    Node newNode = new Node( node.val, new ArrayList<>(
node.neighbors.size() ) );
    visited.put( node, newNode );
    for( int i = 0; i < node.neighbors.size(); i++ )
        newNode.neighbors.add( i, clone( node.neighbors.get(i), visited ) );
    return newNode;
}
```

题目示例7 `leetcode 200 岛屿数量`

```java
public int numIslands( char[][] grid )
{
    int counter = 0;
    for( int i = 0; i < grid.length; i++ )
        for( int j = 0; j < grid[0].length; j++ )
            if( grid[i][j] == '1' && dfs( grid, i, j ) >= 1 )
                counter++;
    return counter
}
```

```
10
11  private int dfs( char[][] grid, int i, int j )
12  {
13      if( i < 0 || i >= grid.length || j < 0 || j >= grid[0].length )
14          return 0;
15      if( grid[i][j] == '1' )
16      {
17          grid[i][j] = '0';
18          return dfs( grid, i - 1, j ) + dfs( grid, i, j - 1 ) + dfs( grid, i
    + 1, j ) + dfs( grid, i, j + 1 ) + 1;
19      }
20      return 0;
21  }
```

队列

题目示例8 `leetcode 232 用栈实现队列`

```
1   class MyQueue
2   {
3       Stack<Integer> stack1;
4       Stack<Integer> stack2;
5       /** Initialize your data structure here. */
6       public MyQueue()
7       {
8           stack1 = new Stack<>();
9           stack2 = new Stack<>();
10      }
11
12      /** Push element x to the back of queue. */
13      public void push(int x)
14      {
15          stack1.push( x );
16      }
17
18      /** Removes the element from in front of queue and returns that element.
    */
19      public int pop()
20      {
21          if( !stack2.isEmpty() )
22              return stack2.pop();
23          while( !stack1.isEmpty() )
24          {
25              int t = stack1.pop();
26              stack2.push( t );
27          }
28          return stack2.pop();
29      }
30
31      /** Get the front element. */
32      public int peek()
33      {
34          if( !stack2.isEmpty() )
35              return stack2.peek();
36          while( !stack1.isEmpty() )
37          {
38              int t = stack1.pop();
```

```
            stack2.push( t );
        }
        return stack2.peek();
    }

    /** Returns whether the queue is empty. */
    public boolean empty()
    {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue obj = new MyQueue();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * boolean param_4 = obj.empty();
 */
```

题目示例9 `leetcode 542 01矩阵`

// 暂存

```
private int[][] updateMatrix( int[][] matrix )
{

}
```

基础算法

排序

二分搜索

二分搜索模板

零、二分查找框架

```
private int binarySearch( int[] nums, int target )
{
    int left = 0, right = ...;

    while( ... )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            ...;
        else if( nums[mid] < target )
            left = ...;
        else if( nums[mid] > target )
```

```
            right = ...;
        }
        return ...;
    }

    // 关键点一：分析二分查找算法时，不要出现else,而是把所有情况都用else if写清楚，这样可以清楚的展现所有细节
    // 关键点二：为了防止计算mid时发生溢出，应使用 mid = left + ( right - left ) / 2来代替mid = (right + left ) / 2
```

一、寻找一个数（基本的二分搜索）

搜索一个数，如果存在，返回其索引，否则返回-1

```
private int binarySearch( int[] nums, int target )
{
    int left = 0, right = nums.length - 1;
    while( left <= right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            return mid;
        else if( nums[mid] < target )
            left = mid + 1;
        else if( nums[mid] > target )
            right = left - 1;
    }
    return -1;
}
```

1、while循环中条件为left <= right ,而不是left < right 的原因

因为初始化时`right`赋值为`nums.length - 1`,则每次搜索的区间为闭区间`[left, right]`,循环终止有两个可能:

- 找到目标值，即`nums[mid] == target`
- 搜索区间为空，`while( left <= right )`终止条件为`left == right + 1`,表示闭区间`[right + 1, right ]`,此时区间为空，`while`循环正确终止

2、left = mid + 1, right = mid - 1的变化规律

因为这个算法搜索区间为闭区间`[left, right]`,当发现`mid`对应位置不是目标值时，应该将其从搜索区间中去除，搜索区间变为`[left, mid - 1]`或`[mid + 1, right]`

3、算法的缺陷

无法有效进行边界搜索

二、寻找左侧边界的二分搜索

```
private int leftBound( int[] nums, int target )
{
    if( nums.length == 0 )
        return -1;
    int left = 0;
    int right = nums.length;
```

```
7
8       while( left < right )
9       {
10          int mid = left + ( right - left ) / 2;
11          if( nums[mid] == target )
12              right = mid;
13          else if( nums[mid] < target )
14              left = mid + 1;
15          else if( nums[mid] > target )
16              right = mid;
17      }
18      return left;
19  }
```

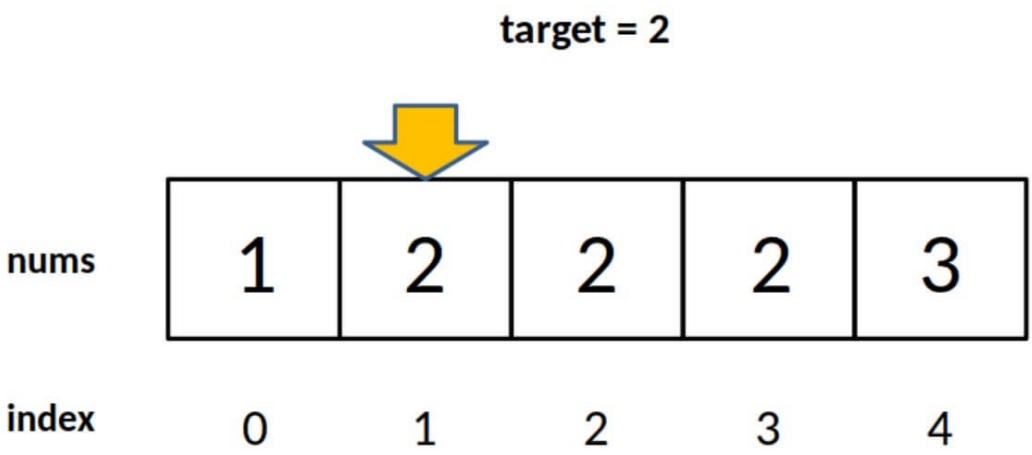
1、while中条件为 left < right 而不是 left <= right的原因

因为初始化时 `right` 赋值为 `right = nums.length` 而不是 `nums.length - 1`,其搜索区间为 `[left, right)` 的左闭右开的区间

`while( left < right )` 的终止条件为 `left == right`,此时搜索区间 `[left, left)` 为空，循环可以正确终止

2、算法在数组中不存在target值的情况下返回结果的含义

左侧边界的含义：



对于上图的数组，算法返回值为1，其含义可以理解为：`nums` 中小于2的元素有1个

再比如有序数组 `nums = [2, 3, 5, 7]`,`target = 1`,算法返回值为0，表示： `nums` 中小于1的元素有0个

再比如有序数组 `nums = [2, 3, 5, 7],target = 8` 算法返回值为4,表示： `nums` 中小于8的元素有4个

可以看出，函数的返回值（即 `left` 变量的值）取值范围为 `[0, nums.length]`,可以通过添加简单的代码来处理数组中不存在目标值的情况

```
while( left < right )
{
    // ...
}

if( left == nums.length )
    return -1;
return nums[left] == target? left:-1;
```

3、`left = mid + 1, right = mid`的变化规律

因为算法搜索区间为`[left, right)`的半开半闭区间，当发现`mid`对应位置不是目标值时，搜索区间应该为`mid`分割的两个子区间`[left, mid )`或`[mid + 1, right)`

4、算法搜索左侧边界的原理

在`nums[mid] == target`时，算法的处理方式为：

```
if( nums[mid] == target )
    right = mid;
```

通过不断缩小搜索区间的上界，使得搜索区间不断向左收缩，达到锁定左侧边界的目的

5、返回值设置为left的原因

其实也可以设置为`right`,因为循环的终止条件为`left == right`

三、寻找右侧边界的二分查找

```
private int rightBound( int[] nums, int target )
{
    if( nums.length == 0 )
        return -1;

    int left = 0, right = nums.length;
    while( left < right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            left = mid + 1;
        else if( nums[mid] < target )
            left = mid + 1;
        else if( nums[mid] > target )
            right = mid;
    }
    return left - 1;
}
```

1、算法搜索右侧边界的原理

```
if( nums[mid] == target )
    left = mid + 1;
```

通过不断增大搜索区间的下界`left`,使得搜索区间不断向右收缩，达到锁定右侧边界的目的

2、返回值设置为left - 1的原因

`while`循环终止的条件为`left == right`,所以也可以返回`right - 1`

由于在搜索右侧边界时有:

```
if( nums[mid] == target )
    left = mid + 1;
```

所以有`mid = left - 1`

3、在数组中不存在目标值时的返回结果设置

与左侧边界搜索相同，因为`while`的终止条件为`left == right`,也就是 left 的取值范围为`[0, nums.length]`,可以添加如下代码处理边界条件

```
while( left < right )
{
    //...
}
if( left == 0 )
    return -1;
return nums[left - 1] == target ? (left - 1):-1;
```

四、逻辑统一

在之前的分析中，普通的二分搜索与左右边界的二分 搜索在形式上有所区别，在这里对其进行统一，规定使用两端封闭的搜索区间来实现

```
// 基本的二分搜索模板
private int binarySearch( int[] nums, int target )
{
    int left = 0, right = nums.length - 1;
    while( left <= right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            return mid;
        else if( nums[mid] < target )
            left = mid + 1;
        else if( nums[mid] > target )
            right = mid - 1;
    }
    return -1;
}

// 搜索左侧边界的二分搜索模板
private int leftBound( int[] nums, int target )
{
    int left = 0, right = nums.length - 1;
    while( left <= right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            right = mid - 1;
        else if( nums[mid] < target )
            left = mid + 1;
```

```
        else if( nums[mid] > target )
            right = mid - 1;
    }

    // 检查left 越界情况
    if( left >= nums.length || nums[left] != target )
        return -1;
    return left;
}

// 搜索右侧边界的二分搜索模板
private int rightBound( int[] nums, int target )
{
    int left = 0, right = nums.length - 1;
    while( left <= right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            left = mid + 1;
        else if( nums[mid] < target )
            left = mid + 1;
        else if( nums[mid] > target )
            right = mid - 1;
    }

    if( right < 0 || nums[right] != target )
        return -1;
    return right;
}
```

典型题目

题目示例1 `leetcode 35 插入位置`

```
// 寻找左侧边界值的二分搜索问题
private int searchInsert( int[] nums, int target )
{
    int left = 0, right = nums.length - 1;
    while( left <= right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            right = mid - 1;
        else if( nums[mid] < target )
            left = mid + 1;
        else if( nums[mid] > target )
            right = mid - 1;
    }
    return left;
}
```

题目示例2 `leetcode 74 搜索二维矩阵`

```
// 关键：将二维矩阵上的搜索转化为一维矩阵上的搜索
private boolean searchMatrix( int[][] matrix, int target )
```

```
{
    if( matrix == null || matrix.length == 0 || matrix[0].length == 0 )
        return false;

    int rowLen = matrix.length, colLen = matrix[0].length;
    int start = 0, end = rowLen * colLen - 1;
    while( start <= end )
    {
        int mid = start + ( end - start ) / 2;
        int curVal = matrix[mid / colLen][mid % colLen];
        if( curVal == target )
            return true;
        else if( curVal < target )
            start = mid + 1;
        else if( curVal > target )
            end = mid - 1;
    }
    return false;
}
```

题目示例3 `leetcode 278 第一个错误的版本`

```
// 寻找左侧边界的二分搜索问题
private int firstBadVersion( int n )
{
    int start = 1, end = n;
    while( start < end )
    {
        int mid = start + ( end - start ) / 2;
        if( isBadVersion( mid ) )
            end = mid;
        else if( !isBadVersion( mid ) )
            left = mid + 1;
    }
    return left;
}
```

题目示例4 `leetcode 153 寻找旋转排序数组中的最小值`

无重复元素,搜索数组中的最小值

```
// v1
private int findMin( int[] nums )
{
    if( nums == null || nums.length <= 0 )
        return -1;

    int left = 0, right = nums.length - 1;
    while( left < right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] <= nums[right] )
            right = mid;
        else if( nums[mid] > nums[right] )
            left = mid + 1;
    }
```

```
    return nums[left];
}

// v2
private int findMin( int[] nums )
{
    int left = 0, right = nums.length - 1;
    while( left <= right )
    {
        if( nums[left] <= nums[right] )
            return nums[left];

        int mid = left + ( right - left ) / 2;
        if( nums[left] <= nums[mid] )
            left = mid + 1;
        else
            right = mid;
    }
    return -1;
}
```

题目示例 5 `leetcode 154 寻找旋转排序数组的最小值II`

存在重复元素，搜索数组中的最小值

```
public int findMin(int[] nums)
{
    if( nums == null || nums.length <= 0 )
        return -1;

    int left = 0, right = nums.length - 1;
    while( left < right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] < nums[right] )
            right = mid;
        else if( nums[mid] > nums[right] )
            left = mid + 1;
        else
            right--;
    }
    return nums[left];
}
```

题目示例5 `leetcode 33 搜索旋转排序数组`

无重复元素，寻找特定值

```
private int search( int[] nums, int target )
{
    if( nums == null || nums.length == 0 )
        return -1;
    if( nums.length == 1 )
        return nums[0] == target ? 0:-1;

    int left = 0, right = nums.length - 1;
    while( left <= right )
```

```
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            return mid;
        if( nums[0] <= nums[mid] )
        {
            if( nums[0] <= target && target < nums[mid] )
                right = mid - 1;
            else
                left = mid + 1;
        }
        else
        {
            if( nums[mid] < target && target <= nums[nums.length - 1] )
                left = mid + 1;
            else
                right = mid - 1;
        }
    }
    return -1;
}
```

题目示例 6 `leetcode 81搜索旋转排序数组II`

```
private boolean search( int[] nums, int target )
{
    if( nums == null || nums.length == 0 )
        return false;

    int left = 0, right = nums.length - 1;
    while( left <= right )
    {
        int mid = left + ( right - left ) / 2;
        if( nums[mid] == target )
            return true;
        if( nums[left] == nums[mid] )
        {
            left ++;
            continue;
        }

        if( nums[0] < nums[mid] )
        {
            if( nums[0] <= target && target < nums[mid] )
                right = mid - 1;
            else
                left = mid + 1;
        }
        else
        {
            if( target > nums[mid] && target <= nums[nums.length - 1] )
                left = mid + 1;
            else
                right = mid - 1;
        }
    }
    return false;
```

```
}
```

动态规划

典型题目

矩阵类型(10%)

题目示例1 `leetcode 64 最小路径和`

```
// dp[i][j] 表示从起点走到（i,j）的最短路径长度
// dp[i][j] = min( dp[i-1][j], dp[i][j-1] ) + grid[i][j]
// base case:dp[0][0] = grid[0][0], dp[i][0] = sum( 0, 0 -> i, 0 ), dp[0][i] = sum( 0, 0 -> 0, i )
// return dp[rowLen-1][colLen-1]

// v1
private int minPathSum( int[][] grid )
{
    if( grid == null || grid.length == 0 || grid[0].length == 0 )
        return 0;

    int m = grid.length, n = grid[0].length;
    int[][] dp = new int[m][n];
    dp[0][0] = grid[0][0];
    for( int i = 1; i < m; i++ )
        dp[i][0] = grid[i][0] + dp[i-1][0];
    for( int i = 1; i < n; i++ )
        dp[0][i] = grid[0][i] + dp[0][i-1];

    for( int i = 1; i < m; i++ )
        for( int j = 1; j < n; j++ )
            dp[i][j] = Math.min( dp[i-1][j], dp[i][j-1] ) + grid[i][j];
    return dp[m-1][n-1];
}

// 根据v1的解法可以知道，dp[i][j]的值只依赖其左侧，上侧的值和当前所在位置的值
// 可以压缩使用的空间，得到如下解法
// v2
private int minPathSum( int[][] grid )
{
    if( grid == null || grid.length == 0 || grid[0].length == 0 )
        return 0;

    int m = grid.length, n = grid[0].length;
    int[] dp = new int[n];
    for( int i = 0; i < m; i++ )
    {
        for( int j = 0; j < n; j++ )
        {
            if( j == 0 )
                dp[j] = dp[j];       // 只能从上侧走到该位置
            else if( i == 0 )
                dp[j] = dp[j-1];     // 只能从左侧走到该位置
            else
```

```
                dp[j] = Math.min( dp[j], dp[j-1] );
            dp[j] += grid[i][j];
        }
    }
    return dp[n-1];
}
```

题目示例2 `leetcode 62 不同路径`

```
// dp[i][j]表示从起点（0，0）走到当前位置（i,j）的路径总数
// 这里使用了空间压缩，原理与示例一相同
private int uniquePaths( int m, int n )
{
    if( m <= 0 || n <= 0 )
        return 0;

    int[] dp = new int[n];
    Arrays.fill( dp, 1 );
    for( int i = 1; i < m; i++ )
        for( int j = 1; j < n; j++ )
            dp[j] = dp[j] + dp[j-1];
    return dp[n-1];
}
```

题目示例2 `leetcode 63不同路径II`

```
// v1
private int uniquePathWithObstacles( int[][] obstacleGrid )
{
    int m = obstacleGrid.length, n = obstacleGrid[0].length;
    if( m == 0 || n == 0 )
        return 0;

    int[][] dp = new int[m][n];
    dp[0][0] = obstacleGrid[0][0] == 0? 1: 0;
    if( dp[0][0] == 0 )
        return 0;                                          // 起点就是障碍物时无法做任何移动
    for( int i = 1; i < m; i++ )
        if( obstacleGrid[i][0] != 1 )
            dp[i][0] = dp[i-1][0];
    for( int i = 1; i < n; i++ )
        if( obstacleGrid[0][i] != 1 )
            dp[0][i] = dp[0][i-1];

    for( int i = 1; i < m; i++ )
        for( int j = 1; j < n; j++ )
            if( obstacleGrid[i][j] != 1 )
               dp[i][j] = dp[i-1][j] + dp[i][j-1];
    return dp[m-1][n-1];
}

// v2 空间压缩版本
private int uniquePathWithObstacles( int[][] obstacleGrid )
{
    int m = obstacleGrid.length, n = obstacleGrid[0].length;
```

```
    if( m == 0 || n == 0 )
        return 0;

    int[] dp = new int[n];
    dp[0] = 1;
    for( int i = 0; i < m; i++ )
    {
        for( int j = 0; j < n; j++ )
            if( obstacleGrid[i][j] == 1 )
                dp[j] = 0;
        else if( j > 0 )
            dp[j] += dp[j-1];
    }
    return dp[n-1];
}
```

序列类型（40%）

题目示例1 `leetcode 70 爬楼梯`

```
private int climbStairs( int n )
{
    if( n < 3 )
        return n;

    int[] dp = new int[n+1];
    dp[1] = 1;
    dp[2] = 2;
    for( int i = 3; i <= n; i++ )
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

题目示例2 `leetcode 55跳跃游戏`

```
// dp[i]表示是否能从0跳到i
// base case:dp[0] = true
// return dp[nums.length - 1]
private boolean canJump( int[] nums )
{
    if( nums.length == 0 )
        return true;

    boolean[] dp = new boolean[nums.length - 1];
    dp[0] = true;
    for( int i = 1; i < nums.length; i++ )
        for( int j = 0; j < i; j++ )
            if( dp[j] && nums[j] + j >= i )
                dp[i] = true;
    return dp[nums.length - 1];
}
```

题目示例3 `leetcode 45跳跃游戏II`

```java
// dp[i]表示从0跳到i的最小次数
// base case: dp[0] = 0
// return dp[nums.length-1]
// v1 出现了超时
private int jump( int[] nums )
{
    int[] dp = new int[nums.length];
    dp[0] = 0;
    for( int i = 1; i < nums.length; i++ )
    {
        dp[i] = i;          // 最大值为i,相当于从0开始每次跳一步到达当前位置
        for( int j = 0; j < i; j++ )
            if( j + nums[j] >= i )
                dp[i] = Math.min( dp[j] + 1, dp[i] );
    }
    return dp[nums.length-1];
}

// v2
private int jump( int[] nums )
{
    int res = 0, end = 0, maxPos = 0;
    for( int i = 0; i < nums.length - 1; i++ )
    {
        maxPos = Math.max( maxPos, nums[i] + i );
        if( i == end )
        {
            end = maxPos;
            res++;
        }
    }
    return  res;
}
```

题目示例4 `leetcode 132 分割回文串`

```java
// dp[i]表示字符串前i个字符组成的子字符串需要的最少分割次数
// base case : dp[0] = -1;
// return: dp[s.length()-1]
private int minCut( String s )
{

}
```

题目示例5 `leetcode 300最长上升子序列`

```java
private int lengthOfLIS( int[] nums )
{
    if( nums == null || nums.length == 0 )
        return 0;

    int[] dp = new int[nums.length];
    Arrays.fill( dp, 1 );
    for( int i = 0; i < nums.length; i++ )
        for( int j = 0; j < i; j++ )
            if( nums[i] > nums[j] )
```

```
                dp[i] = Math.max( dp[i], dp[j] + 1 );
    int res = 0;
    for( int re:dp )
        res = Math.max( res, re );
    return res;
}
```

题目示例6 `leetcode 139 单词拆分`

```
private boolean wordBreak( String s, List<String> wordDict )
{

}
```

双序列（字符串）DP类型 （40%）

0-1背包问题 （10%）

题目示例1 `leetcode 416分割等和子集`

```
// dp[i][j] = var 表示，对于前i个物品，当背包容量为j时，若var = true，表示恰好将背包装满，反之表示装不满
// base case 1: dp[...][0] = true,表示背包容量为0时相当于装满了
// base case 2: dp[0][...] = false,表示没有物品可以选择的时候，无论如何无法装满背包
// return: dp[N][sum/2]
// v1
private boolean canPartition( int[] nums )
{
    int sum = 0;
    for( int num:nums )
        sum += num;
    if( sum % 2 != 0 )
        return false;

    int n = nums.length;
    sum = sum / 2;
    boolean[][] dp = new boolean[n+1][sum+1];
    for( int i = 0; i <= n; i++ )
        dp[i][0] = true;

    for( int i = 1; i <= n; i++ )
        for( int j = 1; j <= sum; j++ )
            if( j - nums[i-1] < 0 )
                dp[i][j] = dp[i-1][j];
            else
                dp[i][j] = dp[i-1][j] | dp[i-1][j-nums[i-1]];
    return dp[n][sum];
}

// v2 状态压缩
private boolean canPartition( int[] nums )
{
    int sum = 0;
    for( int num:nums )
        sum += num;
```

```
    if( sum % 2 != 0 )
        return false;

    int n = nums.length;
    sum /= 2;
    boolean[] dp = new boolean[sum+1];
    dp[0] = true;

    for( int i = 1; i <= n; i++ )
        for( int j = sum; j >= 0; j-- )
            if( j - nums[i-1] >= 0 )
                dp[j] = dp[j] | dp[j-nums[i-1]];
    return dp[sum];
}
```

题目示例2 `leetcode 322零钱兑换`

```
// dp[i]定义：当目标金额为i时，最少需要的硬币数量为dp[i]
private int coinChange( int[] coins, int amount )
{
    int[] dp = new int[amount+1];
    Arrays.fill( dp, amount+1 );
    dp[0] = 0;
    for( int i = 0; i < dp.length; i++ )
        for( int coin:coins )
            if( i - coin >= 0 )
                dp[i] = Math.min( dp[i], dp[i-coin] + 1 );
    return dp[amount] == amount + 1? -1:dp[amount];
}
```

题目示例3 `leetcode 518 零钱兑换II`

```
private int change( int amount, int[] coins )
{
    int n = coins.length;
    int[] dp = new int[amount+1];
    dp[0] = 1;
    for( int i = 0; i < n; i++ )
        for( int j = 1; j <= amount; j++ )
            if( j - coins[i] >= 0 )
                dp[j] = dp[j] + dp[j-coins[i]];
    return dp[amount];
}
```

`leetcode 股票买卖系列问题`

状态：天数、允许交易的最大次数、股票的持有状态

选择：买入股票、卖出股票、无操作

```
/**
* 构建如下的dp数组
* 0 <= n <= n - 1, 1 <= k <= K
* n为天数,K为最大可交易次数
* 题目最多有n X K X 2种状态，可以全部穷举
* 0和1表示持有状态，0表示不持有股票，1表示持有股票
*/
dp[i][k][0 or 1];
for( int i = 0; i < n; i++ )
    for( int k = 1; k <= K; k++ )
        for( int s :{0, 1} )
            dp[i][k][s] = Math.max( buy, sell, rest );
// 最终需要求解得到的答案是:dp[n-1][K][0]

// 根据分析得到如下的状态转移方程
/**
* 解释：今天不持有股票(s = 0 )有两种原因：
* 1.昨天就未持有股票，今天选择rest(不参与购买股票),今天仍然未持有股票
* 2.昨天就持有股票
*/
dp[i][k][0] = Math.max( dp[i-1][k][0], dp[i-1][k][1] + prices[i] );
            = Math.max(  选择rest     ,          选择sell          );

/**
* 解释：今天持有股票( s = 1 )有两种原因：
* 1.昨天就持有股票，今天选择rest(不参与售出股票),今天仍然持有股票
* 2.昨天未持有股票，今天选择buy(购入股票)
* 这个状态转移方程中出现了k-1的原因是：把一次购入股票和售出股票的操作作为一次完整的交易，
* 以购入股票为标志代表使用了一次交易机会
*/
dp[i][k][1] = Math.max( dp[i-1][k][1], dp[i-1][k-1][0] - prices[i] );
           = Math.max(  选择rest     ,          选择buy             );

/**
* 定义base case
*/
dp[-1][k][0] = 0;                   // i从0开始，故i < 0意味着从未进行任何交易，利润为0
dp[-1][k][1] = Integer.MIN_VALUE;    // 未进行交易时不可能持有任何股票，用负无穷表示
dp[i][0][0] = 0;                    // k从1开始，k < 1代表不允许进行任何交易，利润为0
dp[i][0][1] = Integer.MIN_VALUE;     // 不允许进行任何交易的情况下不可能持有股票，用负无穷表示


/**
* 状态转移方程总结
*/
// base case
dp[-1][k][0] = dp[i][0][0] = 0;
dp[-1][k][1] = dp[i][0][1] = Integer.MIN_VALUE;
// 状态转移
dp[i][k][0] = Math.max( dp[i-1][k][0], dp[i-1][k][1] + prices[i] );
dp[i][k][1] = Math.max( dp[i-1][k][1], dp[i-1][k-1][0] - prices[i] );
```

题目示例1 `leetcode 121 买卖股票的最佳时机`

分析：K = 1，可以不考虑其影响

```
class Solution
{
    public int maxProfit(int[] prices)
    {
        int n = prices.length;
        int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
        for( int i = 0; i < n; i++ )
        {
            dpI0 = Math.max( dpI0, dpI1 + prices[i] );
            dpI1 = Math.max( dpI1, -prices[i] );
        }
        return dpI0;
    }
}
```

题目示例2 `leetcode 122 买卖股票的最佳时机II`

分析：K = infinity,可以不考虑其影响

```
class Solution
{
    public int maxProfit(int[] prices)
    {
        int n = prices.length;
        int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
        for( int i = 0; i < n; i++ )
        {
            int temp = dpI0;
            dpI0 = Math.max( dpI0, dpI1 + prices[i] );
            dpI1 = Math.max( dpI1, temp - prices[i] );
        }
        return dpI0;
    }
}
```

题目示例3 `leetcode 123 买卖股票的最佳时机III`

分析：K = 2，遍历所有的K值

```
class Solution
{
    public int maxProfit(int[] prices)
    {
        if( prices == null || prices.length < 2 )
            return 0;
        int maxK = 2;
        int n = prices.length;
        int[][][] dp = new int[n][maxK+1][2];
        for( int i = 0; i < n; i++ )
            for( int k = 1; k <= maxK; k++ )
                if( i == 0 )
                {
                    dp[i][k][0] = 0;
                    dp[i][k][1] = -prices[i];
```

```
                }
                else
                {
                    dp[i][k][0] = Math.max( dp[i-1][k][0], dp[i-1][k][1] + prices[i] );
                    dp[i][k][1] = Math.max( dp[i-1][k][1], dp[i-1][k-1][0] - prices[i] );
                }
        return dp[n-1][maxK][0];
    }
}
```

题目示例4 `leetcode 124 买卖股票的最佳时机IV`

分析：K = infinity,一次完整的交易至少需要两天时间，所以n天之内最多可以完成n / 2次交易，如果K > n / 2,解决方案与 K = infinity时一致，惨开 `leetcode 122`

```
class Solution
{
    public int maxProfit(int k, int[] prices)
    {
        if( prices == null || prices.length < 2 || k < 1 )
            return 0;
        int n = prices.length;
        if( k > n / 2 )
            return maxProfitWithInfiniteK( prices );
        else
            return maxProfitWithLimitedK( k, prices );
    }

    private int maxProfitWithInfiniteK( int[] prices )
    {
        int n = prices.length;
        int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
        for( int i = 0; i < n; i++ )
        {
            int temp = dpI0;
            dpI0 = Math.max( dpI0, dpI1 + prices[i] );
            dpI1 = Math.max( dpI1, temp - prices[i] );
        }
        return dpI0;
    }

    private int maxProfitWithLimitedK( int K, int[] prices )
    {
        int n = prices.length;
        int[][][] dp = new int[n][K+1][2];
        for( int i = 0; i < n; i++ )
            for( int k = 1; k <= K; k++ )
                if( i == 0 )
                {
                    dp[i][k][0] = 0;
                    dp[i][k][1] = -prices[i];
                }
                else
                {
```

```
                    dp[i][k][0] = Math.max( dp[i-1][k][0], dp[i-1][k][1] +
prices[i] );
                    dp[i][k][1] = Math.max( dp[i-1][k][1], dp[i-1][k-1][0]
- prices[i] );
                }
        return dp[n-1][K][0];
    }
}
```

题目示例5 `leetcode 309 买卖股票的最佳时机含冷冻期`

分析：每次sell操作完成之后要隔一天才能进行下一次交易

```
class Solution
{
    public int maxProfit(int[] prices)
    {
        int n = prices.length;
        int preSell = 0;
        int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
        for( int i = 0; i < n; i++ )
        {
            int temp = dpI0;
            dpI0 = Math.max( dpI0, dpI1 + prices[i] );
            dpI1 = Math.max( dpI1, preSell - prices[i] );
            preSell = temp;
        }
        return dpI0;
    }
}
```

题目示例6 `leetcode 714 买卖股票的最佳时机含手续费`

```
class Solution
{
    public int maxProfit(int[] prices, int fee)
    {
        int n = prices.length;
        int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
        for( int i = 0; i < n; i++ )
        {
            int temp = dpI0;
            dpI0 = Math.max( dpI0, dpI1 + prices[i] );
            dpI1 = Math.max( dpI1, temp - prices[i] - fee );
        }
        return dpI0;
    }
}
```

`leetcode 打家劫舍系列问题`

题目示例1 `leetcode 198 打家劫舍`

```
// v1
/**
* dp[i] = x表示:
```

```java
* 从第i间房子开始抢劫，最多能抢到的钱为 x
* base case: dp[n] = 0
*/
class Solution
{
    public int rob(int[] nums)
    {
        int n = nums.length;
        int[] dp = new int[n+2];
        for( int i = n - 1; i >= 0; i-- )
            dp[i] = Math.max( dp[i+1], dp[i+2] + nums[i] );
        return dp[0];
    }
}

// v2
class Solution
{
    public int rob(int[] nums)
    {
        int n = nums.length;
        int dpTwo = 0, dpOne = 0;
        int dpCur = 0;
        for( int i = n - 1; i >= 0; i-- )
        {
            dpCur = Math.max( dpOne, dpTwo + nums[i] );
            dpTwo = dpOne;
            dpOne = dpCur;
        }
        return dpCur;
    }
}
```

题目示例2 `leetcode 213 打家劫舍II`

```java
class Solution
{
    public int rob(int[] nums)
    {
        if( nums.length == 1 )
            return nums[0];
        int res1 = robInRange( nums, 0 , nums.length - 2 );
        int res2 = robInRange( nums, 1, nums.length - 1 );
        return res1 > res2 ? res1:res2;
    }

    private int robInRange( int[] nums, int start, int end )
    {
        int dpOne = 0, dpTwo = 0;
        int dpCur = 0;
        for( int i = end; i >= start; i-- )
        {
            dpCur = Math.max( dpOne, nums[i] + dpTwo );
            dpTwo = dpOne;
            dpOne = dpCur;
        }
        return dpCur;
```

```
    }
}
```

题目示例2 `leetcode 337 打家劫舍III`

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution
{
    public int rob(TreeNode root)
    {
        int[] res = robInTree( root );
        return Math.max( res[0], res[1] );
    }

    /**
    * 返回一个大小为2的数组arr
    * arr[0]表示不抢当前root的话，得到最大钱数
    * arr[1]表示抢当前root的话，得到最大钱数
    */
    private int[] robInTree( TreeNode root )
    {
        if( root == null )
            return new int[]{ 0, 0 };

        int[] leftSub = robInTree( root.left );
        int[] rightSub = robInTree( root.right );

        int notRob = Math.max( leftSub[0], leftSub[1] ) + Math.max(
rightSub[0], rightSub[1] );
        int rob = root.val + leftSub[0] + rightSub[0];
        return new int[]{ notRob, rob };
    }
}
```

算法思维

回溯法

简单的回溯法模板

```
result := []
func backTrack( 选择列表,  路径 ) {
    if 满足结束条件 {
        result.add( 路径 )
        return
    }

    for 选择 in 选择列表 {
        做选择
        backTrack( 选择列表,  路径 )
        撤销选择
    }
}
```

典型题目

题目示例1 `leetcode78 子集`

```
List<List<Integer>> res = new LinkedList<>();
public List<List<Integer>> subsets( int[] nums )
{
    if( nums == null || nums.length == 0 )
        return res;

    backTracking( nums, 0, new LinkedList<Integer>() );
    return res;
}

private void backTracking( int[] nums, int start, LinkedList<Integer> runner
)
{
    res.add( new LinkedList( runner ) );

    for( int i = start; i < nums.length; i++ )
    {
        // 做选择
        runner.add( nums[i] );
        // 进入下一层决策树
        backTracking( nums, i + 1, runner );
        // 撤销选择
        runner.removeLast();
    }
}
```

题目示例2 `leetcode 90 子集II`

```
List<List<Integer>> res = new LinkedList<>();
public List<List<Integer>> subsetsWithDup( int[] nums )
{
    if( nums == null || nums.length == 0 )
        return res;

    Arrays.sort( nums );
    backTracking( nums, 0 , new LinkedList<Integer>() );
    return res;
}
```

```
11
12  private void backTracking( int[] nums, int start, LinkedList<Integer> runner
    )
13  {
14      res.add( new LinkedList( runner ) );
15
16      for( int i = start; i < nums.length; i++ )
17      {
18          if( i > start && nums[i] == nums[i-1] )
19              continue;
20          // 做选择
21          runner.add( nums[i] );
22          // 进入下一层决策树
23          backTracking( nums, i + 1, runner );
24          // 撤销选择
25          runner.removeLast();
26      }
27  }
```

题目示例3 `leetcode 46 全排列`

```
1   List<List<Integer>> res = new LinkedList<>();
2   public List<List<Integer>> permute( int[] nums )
3   {
4       if( nums == null || nums.length == 0 )
5           return res;
6
7       backTracking( nums, new LinkedList<Integer>() );
8       return res;
9   }
10
11  private void backTracking( int[] nums, LinkedList<Integer> runner )
12  {
13      if( runner.size() == nums.length )
14      {
15          res.add( new LinkedList( runner ) );
16          return;
17      }
18
19      for( int i = 0; i < nums.length; i++ )
20      {
21          if( runner.contains( nums[i] ) )
22              continue;
23
24          // 做选择
25          runner.add( nums[i] );
26          // 进入下一层决策树
27          backTracking( nums, runner );
28          // 撤销选择
29          runner.removeLast();
30      }
31  }
```

题目示例4 `leetcode 47 全排列II`

```
1   List<List<Integer>> res = new LinkedList<>();
```

```java
public List<List<Integer>> permuteUnique( int[] nums )
{
    if( nums == null || nums.length == 0 )
        return res;

    boolean[] used = new boolean[nums.length];
    Arrays.sort( nums );
    backTracking( nums, used, new LinkedList<Integer>() );
    return res;
}

private void backTracking( int[] nums, boolean[] used, LinkedList<Integer>
runner )
{
    if( runner.size() == nums.length )
    {
        res.add( new LinkedList( runner ) );
        return;
    }

    for( int i = 0; i < nums.length; i++ )
    {
        if( used[i] )
            continue;
        if( i > 0 && nums[i] == nums[i-1] && !used[i-1] )
            continue;

        // 做选择
        runner.add( nums[i] );
        used[i] = true;
        // 进入下一层决策树
        backTracking( nums, used, runner );
        // 撤销选择
        used[i] = false;
        runner.removeLast();
    }
}
```

题目示例5 `leetcode 77 组合`

```java
List<List<Integer>> res = new LinkedList<>();
public List<List<Integer>> combine( int n, int k )
{
    if( n < 1 || n < k )
        return res;

    backTracking( n, k, 1, new LinkedList<Integer>() );
    return res;
}

private void backTracking( int n, int k, int start, LinkedList<Integer>
runner )
{
    if( runner.size() == k )
    {
        res.add( new LinkedList( runner ) );
        return;
```

```
    }

    for( int i = start; i <= n; i++ )
    {
        // 做选择
        runner.add( i );
        // 进入下一层决策树
        backTracking( n, k, i + 1, runner );
        // 撤销选择
        runner.removeLast();
    }
}
```

题目示例6 `leetcode 39 组合总和`

```
List<List<Integer>> res = new LinkedList<>();
public List<List<Integer>> combinationSum( int[] candidates, int target )
{
    if( candidates == null || candidates.length == 0 )
        return res;

    Arrays.sort( candidates );
    backTracking( candidates, target, 0, new LinkedList<Integer>() );
    return res;
}

private void backTracking( int[] candidates, int target, int start,
LinkedList<Integer> runner )
{
    if( target == 0 )
    {
        res.add( new LinkedList( runner ) );
        return;
    }

    for( int i = start; i < candidates.length; i++ )
    {
        if( target - candidates[i] < 0 )
            break;

        // 做选择、
        runner.add( candidates[i] );
        // 进入下一层决策树
        backTracking( candidates, target - candidates[i], i, runner );
        // 撤销选择
        runner.removeLast();
    }
}
```

题目示例7 `leetcode 40 组合总和II`

```
List<List<Integer>> res = new LinkedList<>();
public List<List<Integer>> combinationSum2( int[] candidates, int target )
{
    if( candidates == null || candidates.length == 0 )
        return res;
```

```
    Arrays.sort( candidates );
    backTracking( candidates, target, 0, new LinkedList<Integer>() );
    return res;
}

private void backTracking( int[] candidates, int target, int start,
LinkedList<Integer> runner )
{
    if( target == 0 )
    {
        res.add( new LinkedList( runner ) );
        return;
    }

    for( int i = start; i < candidates.length; i++ )
    {
        if( target - candidates[i] < 0 )
            break;
        if( i > start && candidates[i] == candidates[i-1] )
            continue;

        // 做选择
        runner.add( candidates[i] );
        // 进入下一层决策树
        backTracking( candidates, target - candidates[i], i + 1, runner );
        // 撤销选择
        runner.removeLast();
    }
}
```

题目示例8 `leetcode 216 组合总和III`

```
List<List<Integer>> res = new LinkedList<>();
public List<List<Integer>> combinationSum3( int k, int n )
{
    if( n <= 0 || k <= 0 )
        return res;

    backTracking( k, n, 1, new LinkedList<Integer>() );
    return res;
}

private void backTracking( int k, int n, int start, LinkedList<Integer>
runner)
{
    // 终止条件
    if( k == 0 )
    {
        if( n == 0 )
            res.add( new LinkedList( runner ) );
        return;
    }


    for( int i = start; i < 10; i++ )
    {
```

```
        // 做选择
        runner.add( i );
        // 进入下一层决策树
        backTracking( k - 1, n - i, i + 1, runner );
        // 撤销选择
        runner.removeLast();
    }
}
```

题目示例9 `leetcode37 解数独`

```
public void solveSudoku( char[][] board )
{
    backTracking( board, 0, 0 );
}

private boolean backTracking( char[][] board, int i, int j )
{
    int row = 9, col = 9;

    // 穷举到最后一列，进入下一行重新开始
    if( j == col )
        return backTracking( board, i + 1, 0 );

    // 找到一个可行解，触发base case
    if( i == row )
        return true;

    // 当前位置已经有数字，不再穷举数字
    if( board[i][j] != '.' )
        return backTracking( board, i, j + 1 );
    for( char c = '1'; c <= '9'; c++ )
    {
        if( !isValid( board, i, j, c ) )
            continue;

        // 做选择
        board[i][j] = c;
        // 进入下一层决策树
        if( backTracking( board, i, j + 1 ) )
            return true;
        board[i][j] = '.';
    }
    return false;
}

private boolean isValid( char[][] board, int row, int col, char c )
{
    for( int i = 0; i < 9; i++ )
    {
        // 判断行是否有重复
        if( board[row][i] == c )    return false;
        // 判断列是否有重复
        if( board[i][col] == c )    return false;
        // 判断3x3方框是否存在重复
        if( board[(row/3)*3 + i/3][(col/3)*3 + i%3] == c )
            return false
```

```
    }
    return true;
}
```

题目示例10 `leetcode22 括号生成`

```
// 这个题目有两个关键性质
// 1.一个合法的括号组合的左括号数量一定等于右括号数量
// 2.对于一个“合法”的括号字符串组合p,必然对于任何 0 <= i < p.length(),都有：子串p[0..i]中左括号的数量都大于等于右括号的数量
List<String> res = new LinkedList<>();
public List<String> generateParenthesis( int n )
{
    if( n <= 0 )
        return res;

    backTracking( n, n, new StringBuffer() );
    return res;
}

private void backTracking( int left, int right, StringBuffer s )
{
    // 剩下的左括号更多，说明不合法
    if( left > right )
        return;
    // 数量小于0，不合法
    if( left < 0 || right < 0 )
        return;
    // 所有括号都能用完，得到一个合法的括号组合
    if( left == 0 && right == 0 )
    {
        res.add( s.toString() );
        return;
    }

    // 尝试放置一个左括号
    // 选择
    s.append( '(' );
    // 进入下一层决策树
    backTracking( left - 1, right, s );
    // 撤销选择
    s.deleteCharAt( s.length() - 1 );

    // 尝试放置一个右括号
    // 选择
    s.append( ')' );
    // 进入下一层决策树
    backTracking( left, right - 1, s );
    // 撤销选择
    s.deleteCharAt( s.length() - 1 );
}
```

题目示例11 `leetcode93 复原IP地址`

题目示例12 `leetcode17 电话号码的字母组合`

题目示例13 `leetcode131 分割回文串`

```java
List<List<String>> res = new LinkedList<>();
public List<List<String>> partition( String s )
{
    backTracking( s, 0, new LinkedList<String> runner );
    return res;
}

private void backTracking( String s, int start, LinkedList<String> runner )
{
    if( start == s.length() )
    {
        res.add( new LinkedList( runner ) );
        return;
    }

    for( int i = start; i < s.length(); i++ )
    {
        if( !isPalindrome( s, start, i ) )
            continue;

        // 选择
        runner.add( s.substring( start, i + 1 ) );
        // 进入下一层决策树
        backTracking( s, i + 1, runner );
        // 撤销选择
        runner.removeLast();
    }
}

private boolean isPalindrome( String s, int left, int right )
{
    while( left < right )
    {
        if( s.charAt( left ) != s.charAt( right ) )
            return false;
        left++;
        right--;
    }
    return true;
}
```

滑动窗口技巧

简单的滑动窗口模板

```java
private void slidingWindow( String s, String t )
{
    HashMap<Character, Integer> need = new HashMap<>();
    HashMap<Character, Integer> window = new HashMap<>();

    for( char c: t.toCharArray() )
        need.put( c, need.getOrDefault( c, 0 ) + 1 );

    int left = 0, right = 0;
```

```
    int valid = 0;
    while( right < s.length() )
    {
        // c是移入窗口的字符
        char c = s.charAt( right  );
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...;

        /*** debug的输出位置 ***/
        System.out.println( "window:[%d, %d ]", left, right );

        // 判断左侧窗口是否需要收缩
        while( window needs shrink )
        {
            // d是将被移出窗口的字符
            char d = s.charAt( left );
            // 窗口左侧右移
            left++;
            // 进行窗口内数据的一系列更新
            ...;
        }
    }
}
```

典型题目

题目示例1 `leetcode 76 最小覆盖子串`

```
private String minWindow( String s, String t )
{
    if( s == null || t == null || s.length() < t.length() )
        return "";

    HashMap<Character, Integer> need = new HashMap<>();
    HashMap<Character, Integer> window= new HashMap<>();
    for( char c : t.toCharArray() )
        need.put( c, need.getOrDefault( c, 0 ) + 1 );

    int left = 0, right = 0;
    int valid = 0;
    // 记录最小覆盖子串的起始索引及长度
    int start = 0, len = Integer.MAX_VALUE;
    while( right < s.length() )
    {
        // c是移入窗口的字符
        char c = s.charAt( right );
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        if( need.containsKey( c ) )
        {
            window.put( c, window.getOrDefault( c, 0 ) + 1 );
            if( window.get( c ).equals( need.get( c ) ) )
                valid++;
        }
```

```
        // 判断左侧窗口是否要收缩
        while( valid == need.size() )
        {
            // 更新最小覆盖子串
            if( right - left < len )
            {
                start = left;
                len = right - left;
            }
            // d是将被移出窗口的字符
            char d = s.charAt( left );
            // 右移窗口左侧
            left++;
            // 进行窗口内数据的一系列更新
            if( need.containsKey( d ) )
            {
                if( window.get( d ).equals( need.get( d ) ) )
                    valid--;
                window.put( d, window.get(d) - 1 );
            }
        }
    }
    return len == Integer.MAX_VALUE? "":s.substring( start, start + len );
}
```

题目示例2 `leetcode 567 字符串的排列`

```
private boolean checkInclusion( String s1, String s2 )
{
    if( s2.length() < s1.length )
        return false;

    HashMap<Character, Integer> need = new HashMap<>();
    HashMap<Character, Integer> window = new HashMap<>();
    for( char c:s1.toCharArray() )
        need.put( c, need.getOrDefault( c, 0 ) + 1 );

    int left = 0, right = 0;
    int valid = 0;
    while( right < s2.length() )
    {
        char c = s2.charAt( right );
        right++;
        if( need.containsKey(c) )
        {
            window.put( c, window.getOrDefault( c, 0 ) + 1 );
            if( window.get( c ).equals( need.get( c ) ) )
                valid++;
        }

        while( (right - left) >= s1.length() )
        {
            // 在这里判断是否找到了合法的子串
            if( valid == need.size() )
                return true;

```

```
            char d = s2.charAt( left );
            left++;
            if( need.containsKey( d ) )
            {
                if( window.get( d ).equals( need.get( d ) ) )
                    valid--;
                window.put( d, window.get(d) - 1 );
            }
        }
    }
    return false;
}
```

题目示例3 `leetcode 438找到所有的字母异位词`

```
private List<Integer> findAnagrams( String s, String p )
{
    List<Integer> res = new LinkedList<>();
    if( s.length() < p.length() )
        return res;

    HashMap<Character, Integer> need = new HashMap<>();
    HashMap<Character, Integer> window = new HashMap<>();
    for( char c:p.toCharArray() )
        need.put( c, need.getOrDefault( c, 0 ) + 1 );

    int left = 0, right = 0;
    int valid = 0;
    while( right < s.length() )
    {
        char c = s.charAt( right );
        right++;
        if( need.containsKey( c ) )
        {
            window.put( c, window.getOrDefault( c, 0 ) + 1 );
            if( window.get(c).equals( need.get(c) ) )
                valid++;
        }

        while( right - left >= p.length() )
        {
            if( valid == need.size() )
                res.add( left );
            char d = s.charAt( left );
            left++;
            if( need.containsKey( d ) )
            {
                if( window.get(d).equals( need.get(d) ) )
                    valid--;
                window.put( d, window.get(d) - 1 );
            }
        }
    }
    return res;
}
```

题目示例4 `leetcode 3 无重复字符的最长子串`

```java
private int lengthOfLongestSubstring( String s )
{
    if( s == null || s.length() == 0 )
        return 0;

    HashMap<Character, Integer> window = new HashMap<>();
    int left = 0, right = 0;
    int res = 0;
    while( right < s.length() )
    {
        char c = s.charAt( right );
        right ++;
        window.put( c, window.getOrDefault( c, 0 ) + 1 );
        while( window.get(c) > 1 )
        {
            char d = s.charAt( left );
            left++;
            window.put( d, window.get(d) - 1 );
        }
        res = Math.max( res, right - left );
    }
    return res;
}
```

前缀和技巧

前缀和简单定义

```java
int n = nums.length;
int[] preSum = new int[n+1];
preSum[0] = 0;
for( int i = 0; i <n; i++ )
    preSum[i+1] = preSum[i] + nums[i];
// preSum[i]表示nums[0..i-1]的和
// nums[i..j]的和可以表示为preSum[j+1] - preSum[i]
```

题目示例1 `leetcode 1 两数之和`

```java
private int[] towSum( int[] nums, int target )
{
    int n = nums.length;
    HashMap<Integer, Integer> hashmap = new HashMap<>();

    for( int i = 0; i < n; i++ )
    {
        int cur = nums[i];
        if( hashmap.containsKey( target - cur ) )
            return new int[]{ hashmap.get( target - cur ), i };
        hashmap.put( nums[i], i );
    }
```

```
    return new int[2];
}
```

题目示例2 leetcode 560 和为K的子数组

```
private int subarraySum( int[] nums, int k )
{
    int n = nums.length;
    int res = 0;
    HashMap<Integer, Integer> preSum = new HashMap<>();
    preSum.put( 0, 1 );
    int curSum = 0;

    for( int i = 0; i < n; i++ )
    {
        curSum += nums[i];
        if( preSum.containsKey( curSum - k ) )
            res += preSum.get( curSum - k );
        preSum.put( curSum, preSum.getOrDefault( curSum, 0 ) + 1 );
    }
    return res;
}
```

题目示例3 leetcode 1248 统计优美子数组

```
private int numberOfSubArrays( int[] nums, int k )
{
    int n = nums.length;
    int[] preSum = new int[n+1];
    preSum[0] = 1;
    int res = 0, curSum = 0;

    for( int num:nums )
    {
        curSum += num & 1;
        preSum[curSum]++;
        if( curSum >= k )
            res += preSum[curSum-k];
    }
    return res;
}
```

题目示例4 leetcode 974 和可被K整除的子数组

```
private int subarrayDivByK( int[] A, int K )
{

}
```