# 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 )
2
3
      if( root == null )
4
           return;
5
      System.out.println(root.val); // 前序
       preOrderTraverse( root.left );
6
7
       // System.out.println( root.val ); 中序
       preOrderTraverse( root.right );
8
9
       // System.out.println(root.val); 后序
10 }
```

## 前序非递归

```
1  // v1
2  private List<Integer> preOrderTraversal( TreeNode root )
3  {
4     List<Integer> res = new LinkedList<>();
5     if( root == null )
6         return res;
7
8     Deque<TreeNode> stack = new LinkedList<>();
9     while( root != null || !stack.isEmpty() )
```

```
10
11
            while( root != null )
12
            {
                // 前序遍历, 先保存结果
13
14
                res.add( root.val );
15
                stack.addLast( root );
16
                root = root.left;
17
            }
18
            // pop
19
            TreeNode node = stack.removeLast();
            root = node.right;
20
21
        }
22
        return res;
23
    }
24
25
    // v2
26
    private List<Integer> preOrderTraversal( TreeNode root )
27
28
        List<Integer> res = new LinkedList<>();
29
        if( root == null )
30
            return res;
31
32
        Deque<TreeNode> stack = new LinkedList<>();
        stack.addLast( root );
33
34
        while( !stack.isEmpty() )
35
36
            TreeNode node = stack.removeLast();
37
            res.add( node.val );
38
            if( node.right != null )
39
                stack.addLast( node.right );
            if( node.left != null )
40
41
                stack.addLast( node.left );
42
        }
43
        return res;
44
    }
```

#### 中序非递归

```
private List<Integer> inOrderTraversal( TreeNode root )
1
 2
 3
        List<Integer> res = new LinkedList<>();
        if( root == null )
4
 5
            return res;
6
 7
        Deque<TreeNode> stack = new LinkedList<>();
        while( !stack.isEmpty() || root != null )
8
9
10
            while( root != null )
11
            {
12
                stack.addLast( root );
13
                root = root.left;
14
            }
15
            // pop
16
            TreeNode node = stack.removeLast();
17
            res.add( node.val );
18
            root = node.right;
19
        }
```

```
20 return res;
21 }
```

#### 后序非递归

```
private List<Integer> postOrderTraversal( TreeNode root )
 1
 2
 3
        List<Integer> res = new LinkedList<>();
 4
        if( root == null )
 5
            return res;
 6
 7
        Deque<TreeNode> stack = new LinkedList<>();
 8
        TreeNode lastVisit = null;
 9
        while( root != null || !stack.isEmpty() )
10
11
            while( root != null )
12
            {
13
                stack.addLast( root );
14
                root = root.left;
15
            }
16
17
            TreeNode node = stack.peekLast();
18
            if( node.right == null || node.right == lastVisit )
19
20
                // 根节点必须在其右子节点弹出之后再弹出
21
                stack.removeLast();
22
                res.add( node.val );
23
                lastVisit = node;
24
25
            else
26
                root = node.right;
27
        }
28
        return res;
29
   }
```

#### DFS 深度搜索

**UP-TO-DOWN** 

```
1
    public List<Integer> preOrderTraversal( TreeNode root )
 2
    {
 3
        List<Integer> res = new LinkedList<>();
 4
        dfs( root, res );
 5
        return res;
    }
 6
 7
 8
    // v1:深度遍历
    private void dfs( TreeNode root, LinkedList<Integer> res )
9
10
        if( root == null )
11
12
            return;
13
        res.add( root.val );
14
        dfs( root.left, res );
15
        dfs( root.right, res );
16 }
```

```
1
    public List<Integer> preOrderTraversal( TreeNode root )
 2
 3
        List<Integer> res = divideAndConquer( root );
 4
        return res;
 5
    }
 6
    // 分治法
 7
    private List<Integer> divideAndConquer( TreeNode root )
 9
        List<Integer> res = new LinkedList<>();
10
11
        if( root == null )
12
            return res;
13
        // divide
14
        List<Integer> left = divideAndConquer( root.left );
15
        List<Integer> right = divideAndConquer( root.right );
16
17
        // conquer
18
19
        res.add( root.val );
20
        res.addAll( new LinkedList( left ) );
21
        res.addAll( new LinkedList( right ) );
22
        return res;
23
   }
```

注意点

<u>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</u> 返回结果最后合并

BFS层次遍历

```
1
    private List<List<Integer>> levelOrder( TreeNode root )
 2
 3
        List<List<Integer>> res = new LinkedList<>();
 4
        if( root == null )
 5
             return res;
 6
        Queue<TreeNode> queue = new LinkedList<>();
 8
        queue.offer( root );
 9
        while( !queue.isEmpty() )
10
        {
11
            List<Integer> runner = new LinkedList<>();
            int len = queue.size();
12
            while( len > 0 )
13
14
15
                 TreeNode node = queue.poll();
16
                 runner.add( node.val );
                 if( node.left != null )
17
18
                     queue.offer( node.left );
                 if( node.right != null )
19
                     queue.offer( node.right );
20
21
                 len--;
22
23
             res.add( new LinkedList( runner ) );
24
        }
```

```
25 return res;
26 }
```

# 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 {
 1
 2
        // nil or leaf
 3
        if root == nil {
            // do something and return
 5
        }
 6
        // Divide
 7
 8
        ResultType left = traversal( root.left )
 9
        ResultType right = traversal( root.right )
10
11
        // Conquer
12
        ResultType result = Merge from left to right
13
        return result
14
15 }
```

#### 典型示例1:通过分治法遍历二叉树

```
1
    public List<Integer> preOrderTraversal( TreeNode root )
 2
 3
        List<Integer> res = divideAndConquer( root );
 4
        return res;
 5
    }
 6
 7
    // 分治法
 8
    private List<Integer> divideAndConquer( TreeNode root )
9
10
        List<Integer> res = new LinkedList<>();
        if( root == null )
11
12
            return res;
13
        List<Integer> left = divideAndConquer( root.left );
        List<Integer> right = divideAndConquer( root.right );
14
15
        res.add( root.val );
        res.addAll( new LinkedList( left ) );
16
17
        res.addAll( new LinkedList( right ) );
18
        return res;
19 }
```

```
public int[] MergeSort( int[] nums )
 2
 3
        return mergeSort( nums, 0, nums.length - 1 );
 4
    }
 5
 6
    private int[] mergeSort( int[] nums, int start, int end )
 7
 8
        if( nums.length <= 1 )</pre>
 9
             return nums;
10
        // divide
11
12
        int mid = start + ( end - start ) / 2;
13
        int[] left = mergeSort( nums, start, mid );
14
        int[] right = mergeSort( nums, mid + 1, end );
15
        int[] res = merge( left, right );
16
        return res;
    }
17
18
19
    private int[] merge( int[] left, int[] right )
20
21
        int[] res = new int[left.length + right.length];
        int leftIndex = 0, rightIndex = 0;
22
23
        int index = 0;
24
        while( leftIndex < left.length && rightIndex < right.length )</pre>
25
26
             if( left[leftIndex] <= right[rightIndex] )</pre>
27
28
                 res[index++] = left[leftIndex];
29
                 leftIndex++;
30
             }
31
             else
32
33
                 res[index++] = right[rightIndex];
34
                 rightIndex++;
             }
35
36
        }
37
        while( leftIndex < left.length )</pre>
38
39
             res[index++] = left[leftIndex++];
        while( rightIndex < right.length )</pre>
40
41
             res[index++] = right[rightIndex++]
42
         return res;
43
    }
```

这里给出官方的 go 代码参考,上面的 java 代码未经过测试

```
1
   func MergeSort( nums []int ) []int {
2
       return mergeSort( nums )
3
   }
4
5
   func mergeSort( nums []int ) []int {
6
       if( len(nums) < 1 ){
7
            return nums
8
       }
9
```

```
10
        // 分治法
11
        mid := len(nums) / 2
12
        left := mergeSort( nums[:mid] )
13
        right := mergeSort( nums[mid:] )
14
        // 合并两段数据
15
        result := merge( left, right )
16
        return result
17
    }
18
    func merge( left, right []int ) ( result []int ) {
19
20
        // 游标
21
        1 := 0
        r := 0
22
23
        // 注意不能越界
24
        for 1 < len(left) && r < len(right) {</pre>
25
            if left[1] > right[r] {
26
                result = append( result, right[r] )
27
                r++
            } else {
28
29
                result = append( result, left[]] )
30
31
            }
32
        // 剩余部分合并
33
        result = append( result, left[1:]... )
        result = append( result, right[r:]... )
35
36
        return
37 }
```

#### 典型示例3: 快速排序

```
public int[] QuickSort( int[] nums )
1
2
    {
        quickSort( nums, 0, nums.length - 1 );
 3
4
        return nums;
 5
    }
6
 7
    private void quickSort( int[] nums, int start, int end )
8
    {
9
        if( start < end )</pre>
10
            int pivot = partition( nums, start, end );
11
            quickSort( nums, 0, pivot - 1 );
12
13
            quickSort( nums, pivot + 1, end );
14
        }
    }
15
16
    // 单边循环法实现partition()方法
17
    private int partition( int[] nums, int start, int end )
18
19
20
        int pivot = nums[start];
21
        int mark = start;
        for( int i = start + 1; i <= end; i++ )
22
23
        {
24
            if( nums[i] < pivot )</pre>
25
            {
26
                 mark++;
```

```
27
                 int temp = nums[i];
28
                 num[i] = nums[mark];
29
                 nums[mark] = temp;
30
            }
31
        }
32
33
        nums[start] = nums[mark];
34
        nums[mark] = pivot;
35
        return mark;
36 }
```

#### 题目示例1 Teetcode 104 :二叉树的最大深度

```
private int maxDepth( TreeNode root )
1
2
    {
 3
        if( root == null )
4
            return 0;
 5
        // divide
6
7
        int left = maxDepth( root.left );
8
        int right = maxDepth( root.right );
        return Math.max( left, right ) + 1;
9
10 }
```

#### **题目示例2** leetcode 110 平衡二叉树

```
1
    public boolean isBalanced( TreeNode root )
2
 3
        if( root == null )
4
            return true;
5
6
        int left = maxDepth( root.left );
7
        int right = maxDepth( root.right );
8
        int absDiff = Math.abs( left - right );
9
        if( absDiff > 1)
10
            return false;
11
12
        return isBalanced( root.left ) && isBalanced( root.right );
13
    }
14
15 | private int maxDepth( TreeNode root )
16
    {
17
        if( root == null )
18
            return 0;
19
20
        // divide
21
        int left = maxDepth( root.left );
        int right = maxDepth( root.right );
22
23
        return Math.max( left, right ) + 1;
24 }
```

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

```
1
    class Solution
 2
 3
        int res = Integer.MIN_VALUE;
 4
        public int maxPathSum( TreeNode root )
 5
 6
            oneSideMax( root );
 7
            return res;
 8
 9
10
        private int oneSideMax( TreeNode root )
11
12
            if( root == null )
13
                return 0;
14
            // divide
15
            int left = oneSideMax( root.left );
16
17
            int right = oneSideMax( root.right );
18
19
            // conquer
            res = Math.max( res, left + right + root.val );
20
21
            return Math.max( left, right ) + root.val;
22
        }
23 }
```

#### 题目示例4 Teetcode 236 二叉树的最近公共祖先

```
1
    private TreeNode lowestCommonAncestor( TreeNode root, TreeNode p, TreeNode
    q)
2
    {
 3
        if( root == null || p == root || q == root )
            return root;
4
 5
        // divide
        TreeNode left = lowestCommonAncestor( root.left, p, q );
6
 7
        TreeNode right = lowestCommonAncestor( root.right, p, q );
8
        // conquer
        return left == null? right: right == null ? left:root;
9
10 }
```

## BFS层次应用

题目示例1: leetcode 102 二叉树的层序遍历

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

```
3
        List<List<Integer>> res = new LinkedList<>();
4
        if( root == null )
 5
            return res;
 6
 7
        Queue<TreeNode> queue = new LinkedList<>();
8
        queue.offer( root );
9
        while( !queue.isEmpty() )
10
11
            List<Integer> runner = new LinkedList<>();
12
            int len = queue.size();
13
            while( len > 0 )
14
            {
                TreeNode node = queue.poll();
15
16
                 runner.add( node.val );
                if( node.left != null )
17
                     queue.offer( node.left );
18
19
                if( node.right != null )
20
                     queue.offer( node.right );
21
                len--;
            }
22
23
            res.add( new LinkedList( runner ) );
24
25
        return res;
26
    }
```

#### 题目示例2: Teetcode 107 二叉树的层次遍历II

```
private List<List<Integer>> levelOrderBottom( TreeNode root )
 1
 2
    {
 3
        List<List<Integer>> res = new LinkedList<>();
 4
        if( root == null )
 5
             return res;
 6
 7
        Queue<TreeNode> queue = new LinkedList<>();
 8
        queue.offer( root );
 9
        while( !queue.isEmpty() )
10
11
            List<Integer> runner = new LinkedList<>();
12
            int len = queue.size();
13
            while( len > 0 )
14
15
                 TreeNode node = queue.poll();
16
                 runner.add( node.val );
17
                 if( node.left != null )
18
                     queue.offer( node.left );
                 if( node.right != null )
19
20
                     queue.offer( node.right );
21
                 len--;
22
            }
23
             res.add( 0, new LinkedList( runner ) );
24
25
        return res;
26 }
```

题目示例3: leetcode 103 二叉树的锯齿形层次遍历

```
private List<List<Integer>> zigzagLevelOrder( TreeNode root )
 2
    {
        List<List<Integer>> res = new LinkedList<>();
3
4
        if( root == null )
 5
            return res;
 6
 7
        Queue<TreeNode> queue = new LinkedList<>();
8
        queue.offer( root );
9
        boolean toggle = false;
10
        while( !queue.isEmpty() )
11
12
            List<Integer> runner = new LinkedList<>();
13
            int levelLen = queue.size();
14
            while( levelLen > 0 )
15
                TreeNode node = queue.poll();
16
17
                if( toggle )
                     runner.add( 0, node.val );
18
19
                else
20
                     runner.add( node.val );
21
                if( node.left != null )
22
                    queue.offer( node.left );
23
24
                if( node.right != null )
25
                    queue.offer( node.right );
26
27
                levelLen--;
28
29
            res.add( new LinkedList( runner ) );
            toggle = !toggle;
30
31
32
        return res;
33 }
```

# 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

```
1
    List<Integer> res = new LinkedList<>();
2
 3
    public boolean isValidBST( TreeNode root )
4
 5
        inOrderTraversal( root );
        for( int i = 0; i < res.size() - 1; i++ )
6
7
            if( res.get(i) >= res.get(i+1) )
8
                return false;
9
        return true;
10
    }
11
    private void inOrderTraversal( TreeNode root )
12
```

```
13  {
14     if( root == null )
15         return;
16
17     inOrderTraversal( root.left );
18     res.add( root.val );
19     inOrderTraversal( root.right );
20  }
```

## 思路2:分治法,判断左MAX<根<右MIN

```
1
    public boolean isValidBST( TreeNode root )
2
    {
3
        return isValidBST( root, null, null );
   }
4
 5
   private boolean isValidBST( TreeNode root, TreeNode min, TreeNode max )
6
7
8
        if( root == null )
9
            return true;
10
        if( min != null && root.val <= min.val )</pre>
11
            return false;
        if( max != null && root.val >= max.val )
12
13
            return false;
14
        return isValidBST( root.left, min, root ) && isValidBST( root.right,
    root, max );
16
   }
```

## 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)
 1
 2
    {
 3
        if( root == null )
             return new TreeNode( val );
 4
 5
        else if( val < root.val )</pre>
            root.left = insertIntoBST( root.left, val );
 6
 7
        else if( val > root.val )
8
            root.right = insertIntoBST( root.right, val );
9
        else
10
            root.val = val;
11
        return root;
12 }
```

## 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 );
```

```
9
        else if( key == root.val )
10
        {
            if( root.left == null )
11
12
                return root.right;
13
            else if( root.right == null )
14
                return root.left;
15
            else
16
            {
17
                TreeNode t = root;
18
                root = min( t.right );
                root.right = deleteMin( t.right );
19
20
                root.left = t.left;
21
            }
22
        }
23
        return root;
24 }
25
    private TreeNode min( TreeNode root )
26
27
28
        if( root == null || root.left == null )
29
            return root;
30
        return min( root.left );
31
    }
32
33
    private TreeNode deleteMin( TreeNode root )
34
35
        if( root == null )
36
            return root;
       if( root.left == null )
37
38
            return root.right;
39
        root.left = deleteMin( root.left );
40
        return root;
41 }
```

#### 题目示例4 leetcode 669 修剪二叉搜索树

```
private TreeNote trimBST( TreeNode root, int L, int R )
1
2
3
       if( root == null )
                              return null;
4
       if( root.val < L )
                            return trimBST( root.right, L, R );
       if( root.val > R ) return trimBST( root.left, L, R );
5
       root.left = trimBST( root.left, L, R );
6
7
       root.right = trimBST( root.right, L, R );
8
       return root;
9
  }
```

#### 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;
8
9
        int t = size( root.left );
10
       if(t >= k)
                                      return select( root.left, k );
        else if( t < k - 1 )
11
                                       return select( root.right, k - t - 1
    );
12
        else
                                       return root.val;
13
    }
   private int size( TreeNode root )
14
15
       if( root == null )
16
17
           return 0;
18
       return size( root.left ) + size( root.right ) + 1;
19
20 }
```

#### 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)
2
3
       if( root.val < p.val && root.val < q.val )</pre>
4
           return lowestCommonAncestor( root.right, p, q );
5
       else if( root.val > p.val && root.val > q.val )
6
           return lowestCommonAncestor( root.left, p, q );
7
       else
8
           return root;
9
  }
```

### 题目示例8 Teetcode 108 将有序数组转换为二叉搜索树

```
public TreeNode sortedArrayToBST(int[] nums)
 1
 2
    {
 3
        return constructTreee( nums, 0, nums.length - 1 );
    }
 4
 5
    private TreeNode constructTreee( int[] nums, int start, int end )
 6
 7
    {
 8
        if( nums == null || nums.length == 0 )
9
            return null;
10
        int mid = start + ( end - start ) / 2;
11
12
        TreeNode root = new TreeNode( nums[mid] );
13
        if( mid - 1 >= start )
14
            root.left = constructTreee( nums, start, mid - 1 );
15
        if( mid + 1 <= end )
16
            root.right = constructTreee( nums, mid + 1, end );
17
        return root;
18 }
```

#### 题目示例9 leetcode 109 有序链表转换二叉搜索树

```
public TreeNode sortedListToBST(ListNode head)
 2
    {
3
        if( head == null )
                               return null;
        if( head.next == null ) return new TreeNode( head.val );
4
        ListNode preMid = preMid( head );
5
        ListNode mid = preMid.next;
6
 7
        preMid.next = null;
8
        TreeNode root = new TreeNode( mid.val );
9
        root.left = sortedListToBST( head );
        root.right = sortedListToBST( mid.next );
10
11
        return root;
12
    }
13
    private ListNode preMid( ListNode head )
14
15
16
        ListNode slow = head;
17
        ListNode fast = head.next;
18
       ListNode pre = head;
19
        while( fast != null && fast.next != null )
20
21
            pre = slow;
22
            slow = slow.next;
23
           fast = fast.next.next;
24
25
        return pre;
26 }
```

## **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 )
 2
    {
 3
        return find( root, new HashSet<Integer>(), k );
 4
    }
 5
    private boolean find( TreeNode root, Set<Integer> set, int k )
 7
        if( root == null )
8
9
            return false;
        if( set.contains( k - root.val ) )
10
11
            return true;
12
13
        set.add( root.val );
14
        return find( root.left, set, k ) || find( root.right, set, k );
15 }
```

## 题目示例 11 leetcode 530 二叉搜索树的最小绝对差

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

```
3
    public int getMinimumDifference(TreeNode root)
4
5
        traversal( root );
       return minDiff;
6
7
    }
8
9
   private void traversal( TreeNode root )
10
11
       if( root == null )
12
            return;
13
       traversal( root.left );
14
       if( pre != null )
            minDiff = Math.min( minDiff, root.val - pre.val );
15
       pre = root;
16
17
       traversal( root.right );
18 }
```

# 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

```
private ListNode deleteDuplicates( ListNode head )
2
3
        ListNode current = head;
       while( current != null )
4
5
            while( current.next != null && current.val == current.next.val )
6
7
                current.next = current.next.next;
8
            current = current.next;
9
        }
        return head;
10
11 }
```

```
private ListNode deleteDuplicates( ListNode head )
 1
 2
 3
        if( head == null )
4
            return head;
 5
6
        ListNode dummy = new ListNode( 0 );
 7
        dummy.next = head;
8
        head = dummy;
9
10
        // 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
11
        int removeVal;
        while( head.next != null && head.next.next != null )
12
13
            if( head.next.val == head.next.next.val )
14
15
                removeVal = head.next.val;
16
                while( head.next != null && head.next.val == removeval )
17
18
                    head.next = head.next.next;
19
            }
            else
20
21
                head = head.next;
22
23
        return dummy.next;
24 }
```

#### **题目示例3** Teetcode 206 反转链表

```
private ListNode reverseList( ListNode head )
1
2
    {
 3
        ListNode pre = null;
4
        ListNode cur = head;
 5
        while( cur != null )
6
7
            ListNode next = cur.next;
8
            cur.next = pre;
9
            pre = cur;
10
            cur = next;
11
        }
12
        return pre;
13
   }
```

## 题目示例4 leetcode 92 反转链表II

#### 递归版本

```
1 |
```

```
private ListNode reverseBetween( ListNode head, int m, int n)

if( head == null )
```

```
return head;
 5
6
        ListNode dummy = new ListNode( 0 );
 7
        dummy.next = head;
8
        head = dummy;
9
        ListNode pre = null;
                                                   // pre固定指向被反转部分最左侧边
    界外第一个结点
10
        for( int i = 0; i < m; i++)
11
12
            pre = head;
13
            head = head.next;
14
        }
15
16
        ListNode next = null;
                                                   // mid固定指向被反转部分最左侧结
17
        ListNode mid = head;
18
        for( int j = i; head != null && j <= n ; j++ )
19
20
           ListNode temp = head.next;
21
           head.next = next;
22
           next = head;
23
           head = temp;
                                                   // head固定指向已被反转部分最右
    侧边界外第一个结点
       }
24
25
        pre.next = next;
26
        mid.next = head;
27
        return dummy.next;
28 }
```

#### 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 11, ListNode 12 )
2
    {
3
        ListNode dummy = new ListNode( 0 );
4
        ListNode runner = dummy;
 5
        while( 11 != null && 12 != null )
 6
7
            if( 11.val < 12.val )
8
            {
9
                 runner.next = 11;
                11 = 11.next;
10
11
12
            else
13
14
                 runner.next = 12;
15
                12 = 12.next;
16
17
             runner = runner.next;
        }
18
19
        if( 11 != null )
20
21
             runner.next = 11;
        if( 12 != null )
22
23
             runner.next = 12;
24
        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 )
 1
 2
    {
 3
        if( head == null )
             return head;
 4
 5
 6
        ListNode headDummy = new ListNode( 0 );
 7
        ListNode tailDummy = new ListNode( 0 );
 8
        ListNode tail = tailDummy;
 9
        headDummy.next = head;
10
        head = headDummy;
11
12
        while( head.next != null )
13
            if( head.next.val < x )
14
15
                head = head.next;
            else
16
17
            {
18
                 ListNode node = head.next;
19
                head.next = head.next.next;
20
                tail.next = node;
21
                tail = tail.next;
22
            }
23
        }
24
        tail.next = null;
25
        head.next = tailDummy.next;
26
        return headDummy.next;
27 }
```

## 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** Teetcode 148 排序链表

思路: 归并排序, 找中点和合并操作

```
public ListNode sortList( ListNode head )
1
 2
 3
        return mergeSort( head );
4
    }
 5
    private ListNode findMiddle( ListNode head )
 6
 7
        ListNode slow = head;
8
9
        ListNode fast = head.next;
        while( fast != null && fast.next != null )
10
11
            fast = fast.next.next;
12
13
            slow = slow.next;
14
15
        return slow;
    }
16
17
```

```
18
    private ListNode mergeTwoLists( ListNode 11, ListNode 12 )
19
20
        ListNode dummy = new ListNode( 0 );
21
        ListNode head = dummy;
22
        while( 11 != null && 12 != null )
23
24
            if( 11.val < 12.val )
25
             {
26
                 head.next = 11;
27
                 11 = 11.next;
28
             }
29
             else
30
             {
                 head.next = 12;
31
32
                 12 = 12.next;
33
             }
34
             head = head.next;
35
        }
36
37
        if( 11 != null )
             head.next = 11;
38
39
        if( 12 != null )
40
             head.next = 12;
41
        return dummy.next;
42
    }
43
44
    private ListNode mergeSort( ListNode head )
45
    {
        if( head == null || head.next == null )
46
47
             return head;
48
49
        // find middle
        ListNode middle = findMiddle( head );
50
51
        // 断开中间结点
        ListNode tail = middle.next;
53
        middle.next = null;
54
55
        ListNode left = mergeSort( head );
        ListNode right = mergeSort( tail );
56
57
        ListNode res = mergeTwoLists( left, right );
58
         return res;
59
    }
```

#### 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 )
1
 2
    {
 3
        if( head == null )
4
            return;
 5
        ListNode middle = findMiddle( head );
6
        ListNode tail = reverseList( middle.next );
        middle.next = null;
8
        head = mergeTwoLists( head, tail );
9
    }
10
```

```
private ListNode findMiddle( ListNode head )
11
12
13
         ListNode slow = head;
14
        ListNode fast = head.next;
15
        while( fast != null && fast.next != null )
16
17
             slow = slow.next;
18
             fast = fast.next.next;
19
         }
20
        return slow;
21
    }
22
23
    private ListNode mergeTwoLists( ListNode 11, ListNode 12 )
24
25
         ListNode dummy = new ListNode( 0 );
26
         ListNode head = dummy;
27
         boolean toggle = true;
28
        while( 11 != null && 12 != null )
29
30
             if( toggle )
31
32
                 head.next = 11;
33
                 11 = 11.next;
             }
34
35
             else
36
37
                 head.next = 12;
                 12 = 12.next;
38
39
             }
40
             toggle = !toggle;
             head = head.next;
41
42
        }
43
        if( 11 != null )
44
45
             head.next = 11;
46
         if( 12 != null )
47
             head.next = 12;
48
         return dummy.next;
    }
49
50
51
    private ListNode reverseList( ListNode head )
52
    {
53
         ListNode pre = null;
54
        while( head != null )
55
        {
             ListNode next = head.next;
56
57
             head.next = pre;
58
             pre = head;
             head = next;
59
60
         }
61
         return pre;
    }
```

#### **题目示例8** leetcode 141 环形链表

快慢指针

```
private boolean hasCycle( ListNode head )
1
 2
 3
         if( head == null )
 4
             return false;
 5
 6
        ListNode slow = head;
 7
         ListNode fast = head.next;
         while( fast != null && fast.next.next != null )
 8
 9
10
             if( fast == slow )
11
                 return true
12
             slow = slow.next;
13
             fast = fast.next.next;
14
         }
15
         return false;
16 }
```

## 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 )
 2
    {
 3
        if( head == null )
             return head;
 4
 5
 6
        ListNode slow = head;
 7
        ListNode fast = head;
8
        while( fast != null && fast.next != null )
9
            if( fast == slow )
10
            {
11
                slow = head;
12
13
                fast = fast.next;
                while( fast != slow )
14
15
16
                     fast = fast.next;
17
                     slow = slow.next;
18
                }
19
                return slow;
            }
20
21
            fast = fast.next.next;
            slow = slow.next;
22
23
        }
24
        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
 8
9
        ListNode fast = head.next;
        while( fast != null && fast.next != null )
10
11
12
            slow = slow.next;
13
            fast = fast.next.next;
14
        }
15
16
        ListNode tail = reverseList( slow.next );
17
        slow.next = null;
18
        while( head != null && tail != null )
19
20
            if( head.val != tail.val )
21
                return false;
            head = head.next;
22
23
            tail = tail.next;
24
25
        return true;
26
    }
27
28
    private ListNode reverseList( ListNode head )
29
    {
        if( head == null )
30
31
            return head;
32
33
        ListNode pre = null;
34
        while( head != null )
35
        {
36
            ListNode next = head.next;
37
            head.next = pre;
38
            pre = head;
            head = next;
39
40
        }
41
        return pre;
42
   }
```

#### 题目示例11 leetcode 138 复制带随机指针的链表

```
private Node copyRandomList( Node head )
 1
 2
 3
        if( head == null )
 4
            return head;
 5
        // 复制节点,紧挨在原节点之后
 6
 7
        Node cur = head;
        while( cur != null )
 8
9
        {
            Node node = new Node( cur.val );
10
11
            node.next = cur.next;
12
            cur.next = node;
13
            cur = node.next;
14
        }
15
        // 处理random指针
16
17
        cur = head;
```

```
18
        while( cur != null )
19
        {
20
            if( cur.random != null )
21
                cur.next.random = cur.random.next;
22
            cur = cur.next.next;
23
        }
24
25
        // 分离两个链表
26
        cur = head;
27
        Node cloneHead = cur.next;
        while( cur != null && cur.next != null )
28
29
30
            Node temp = cur.next;
31
            cur.next = cur.next.next;
32
            cur = temp;
33
        }
34
        return cloneHead;
35 }
```

#### 题目示例 12 leetcode 876 链表的中间结点

```
private ListNode middleNode( ListNode head )
2
3
        if( head == null || head.next == null )
4
            return head;
        ListNode slow = head, fast = head;
6
7
        while( fast != null && fast.next != null )
8
9
            slow = slow.next;
10
            fast = fast.next.next;
11
12
       return slow;
13 }
```

# 栈和队列

# 栈

**题目示例1** Teetcode 155 最小栈

```
class MinStack
 2
 3
        Stack<Integer> dataS;
4
        Stack<Integer> minS;
5
        /** initialize your data structure here. */
6
        public MinStack()
 7
8
            dataS = new Stack<>();
9
            minS = new Stack<>();
10
        }
11
12
        public void push(int x)
```

```
13
14
             datas.push( x );
15
            if( minS.size() == 0 )
16
                 minS.push( x );
17
            else
18
            {
19
                 int min = mins.peek() < x ? mins.peek():x;</pre>
20
                 minS.push( min );
21
            }
22
        }
23
24
        public void pop()
25
26
             dataS.pop();
27
            minS.pop();
28
        }
29
30
        public int top()
31
32
             return datas.peek();
33
        }
34
35
        public int getMin()
36
37
             return minS.peek();
38
        }
39
    }
40
41 /**
42
    * Your MinStack object will be instantiated and called as such:
    * MinStack obj = new MinStack();
43
44
     * obj.push(x);
45
     * obj.pop();
46
     * int param_3 = obj.top();
47
     * int param_4 = obj.getMin();
48
```

#### 题目示例2 Teetcode 150 逆波兰表达式求值

```
private int evalRPN( String[] tokens )
 1
 2
    {
 3
        if( tokens == null || tokens.length == 0 )
 4
             return 0;
 5
 6
         Stack<Integer> stack = new Stack<>();
        for( int i = 0; i < tokens.length; i++ )</pre>
 7
 8
 9
             switch( tokens[i] )
10
             {
                 case "+", "-", "*", "/":
11
12
                      {
                          if( stack.size() < 2 )</pre>
13
14
                              return 0;
15
16
                          int b = stack.pop();
17
                          int a = stack.pop();
18
                          int res = 0;
```

```
19
                          switch( tokens[i] )
20
                          {
                               case "+":
21
22
                                   res = a + b;
23
                                   break;
                               case "-":
24
25
                                   res = a - b;
26
                                   break;
                               case "*":
27
28
                                   res = a * b;
29
                                   break;
30
                               case "/":
31
                                   res = a / b;
32
                                   break;
33
                          }
34
                          stack.push( res );
35
                          break;
36
                      }
37
                 default:
38
                      {
                          int val = Integer.parseInt( tokens[i] );
39
40
                          stack.push( val );
41
                          break;
                      }
42
             }
43
44
         }
45
         return stack.peek();
46 }
```

#### **题目示例3** Teetcode 394 字符串解码

```
public String decodeString(String s)
1
 2
    {
 3
        StringBuilder res = new StringBuilder();
4
        int multi = 0;
        LinkedList<Integer> stack_multi = new LinkedList<>();
 6
        LinkedList<String> stack_res = new LinkedList<>();
        for(Character c : s.toCharArray())
8
        {
9
            if(c == '[')
10
            {
                stack_multi.addLast(multi);
11
12
                stack_res.addLast(res.toString());
13
                multi = 0;
14
                res = new StringBuilder();
15
            }
            else if(c == ']')
16
17
            {
18
                StringBuilder tmp = new StringBuilder();
19
                int cur_multi = stack_multi.removeLast();
20
                for(int i = 0; i < cur_multi; i++) tmp.append(res);</pre>
21
                 res = new StringBuilder(stack_res.removeLast() + tmp);
22
            else if(c >= '0' && c <= '9') multi = multi * 10 +
23
    Integer.parseInt(c + "");
24
            else res.append(c);
25
        }
```

```
26    return res.toString();
27 }
```

# 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

```
1 // 一个简单的单调栈模板
  Deque<Integer> stack = new LinkedList<>();
3
  for( int i = 0; i < nums.length; i++ )</pre>
4
5
       while(!stack.isEmpty() & nums[i] <= nums[stack.peekLast()] ) // 单调递增
  栈
6
           // 单调递减栈 nums[i] >= nums[stack.peekLast()]
7
           stack.removeLast();
8
       stack.addLast( i );
9
  }
```

题目示例 1 Teetcode 496 下一个更大元素I

#### 寻找比当前元素更大的下一个元素

```
1 // v1, 从右往左构建一个单调递减栈
    private int[] nextGreaterElement( int[] nums1, int[] nums2 )
3
4
        int[] res = new int[nums1.length];
5
        int[] temp = new int[nums2.length];
        Deque<Integer> stack = new LinkedList<>();
6
7
        for( int i = nums2.length - 1; i >= 0; i-- )
8
9
            while( !stack.isEmpty() && nums2[i] >= stack.peekLast() )
10
                stack.removeLast();
11
            temp[i] = stack.isEmpty()? -1:stack.peekLast();
12
            stack.addLast( nums2[i] );
13
        }
```

```
14
15
        for( int i = 0; i < nums1.length; i++ )</pre>
            for( int j = 0; j < nums2.length; j++)
16
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
    private int[] nextGreaterElement( int[] nums1, int[] nums2 )
24
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++ )</pre>
30
            while( !stack.isEmpty() && nums2[i] > stack.peekLast() )
31
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++ )</pre>
38
             res[i] = hashmap.get( nums1[i] );
39
        return res;
40
    }
```

#### 题目示例2 Teetcode 503 下一个更大元素II

```
// 从右往左构建一个单调递减栈
    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();
            res[i%n] = stack.isEmpty()? -1:stack.peekLast();
11
12
            stack.addLast( nums[i%n] );
13
14
        return res;
15 }
```

#### **题目示例3** Teetcode 739 每日温度

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

#### **题目示例 4** leetcode 962 最大宽度坡

```
1 // 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
2
   // 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 )
3
4
5
       Deque<Integer> stack = new LinkedList<>();
6
       stack.addLast(0);
7
       // 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++ )</pre>
8
9
           if( A[i] <= A[stack.peekLast()] )</pre>
              stack.addLast( i );
10
11
       // 贪心策略,从最远的地方开始往回找
12
       int maxGap = 0;
       for( int i = A.length - 1; i >= 0; i--)
13
14
           while( !stack.isEmpty() && A[i] >= A[stack.peekLast()] )
              maxGap = Math.max( maxGap, i - stack.removeLast() );
15
16
       return maxGap;
17 | }
```

#### 题目示例5 leetcode 42 接雨水

```
private int trap( int[] height )
1
 2
    {
 3
        if( height == null || height.length == 0 )
 4
            return 0;
 5
 6
        Deque<Integer> stack = new LinkedList<>();
 7
        int res = 0;
8
        for( int i = 0; i < height.length; i++ )</pre>
9
10
11
            while( !stack.isEmpty() && height[stack.peekLast()] < height[i] )</pre>
12
            {
                int bottomIndex = stack.removeLast();
13
14
                // 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()] ==
15
    height[bottomIndex] )
16
                    stack.removeLast();
17
                if( !stack.isEmpty() )
18
                    // leftEdge指向蓄水凹槽的左侧边界
19
                    // 蓄水凹槽的有边界即为i
20
21
                    int leftEdge = stack.peekLast();
```

题目示例 6 Teetcode 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 )
1
 2
 3
        int[] temp = new int[heights.length+2];
4
        System.arraycopy( heights, 0, temp, 1, heights.length );
 5
6
        Deque<Integer> stack = new LinkedList<>();
 7
        int maxArea = 0;
        for( int i = 0; i < temp.length; i++ )</pre>
8
9
10
            while( !stack.isEmpty() && temp[i] < temp[stack.peekLast()] )</pre>
11
                int height = temp[stack.removeLast()];
12
13
                maxArea = Math.max( maxArea, height * ( i - stack.peekLast() -
    1));
14
15
            stack.addLast( i );
16
17
        return maxArea;
18 }
```

题目示例7 leetcode 239 滑动窗口最大值

题目示例8 Teetcode 85 最大矩形

**题目示例9** Teetcode 402 移掉K位数字

题目示例10 Teetcode 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

```
1 private int longestWPI( int[] hours )
2 {
3 // 计算前缀和
```

```
int[] preSum = new int[hours.length+1];
 5
        for( int i = 0; i < hours.length; i++ )</pre>
 6
            if( housr[i] > 8 ) preSum[i+1] = preSum[i] + 1;
 7
 8
            else
                                preSum[i+1] = preSum[i] - 1;
 9
        }
10
11
        // 构建单调栈
12
        Deque<Integer> stack = new LinkedList<>();
13
        stack.addLast( 0 );
14
        for( int i = 1; i < preSum.length; i++ )</pre>
15
            if( preSum[i] < preSum[stack.peekLast()] )</pre>
16
                stack.addLast( i );
17
        // 从右向左利用贪心策略求最大跨度
18
19
        int maxL = 0;
20
        for( int i = preSum.length - 1; i >= 0; i-- )
            while( !stack.isEmpty() && preSum[i] > preSum[stack.peekLast()] )
21
                maxL = Math.max( maxL, i - stack.removeLast() );
22
23
        return maxL;
24 }
```

题目示例14 Teetcode 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 )
1
2
 3
        Set<Node> visited;
4
        Stack<Node> s;
        add root to s;
 6
        while( s is not empty )
7
8
            Node cur = the top element in s;
9
             return true if cur is target;
10
            for( Node next:the neighbors of cur )
11
             {
                 if( next is not visited )
12
13
                 {
                     add next to s;
14
15
                     add next to visited
16
                 }
            }
17
18
             remove cur from s;
19
20
        return false;
21 }
```

```
private List<Integer> inorderTraversal( TreeNode root )
 2
    {
 3
        List<Integer> res = new LinkedList<>();
 4
        if( root == null )
 5
             return res;
 6
 7
        Stack<TreeNode> stack = new Stack<>();
 8
        while( !stack.isEmpty() || root != null )
 9
10
            while( root != null )
11
12
                 stack.push( root );
13
                 root = root.left;
14
            }
15
            TreeNode node = stack.pop();
             res.add( node.val );
16
            root = node.right;
17
18
        }
19
        return res;
20 }
```

#### 题目示例6 Teetcode 133 克隆图

```
public Node cloneGraph( Node node )
1
 2
 3
        HashMap<Node, Node> visited = new HashMap<>();
4
        return clone( node, visited );
 5
    }
6
7
    private Node clone( Node node, HashMap<Node, Node> visited )
8
9
        if( node == null )
10
            return null;
11
        if( visited.get( node ) != null )
12
13
            return visited.get( node );
14
15
        Node newNode = new Node( node.val, new ArrayList<>(
    node.neighbors.size() );
        visited.put( node, newNode );
16
17
        for( int i = 0; i < node.neighbors.size(); i++ )</pre>
18
            newNode.neighbors.add( i, clone( node.neighbors.get(i), visited ) );
19
        return newNode;
20 }
```

#### **题目示例7** leetcode 200 岛屿数量

```
public int numIslands( char[][] grid )
1
2
3
        int counter = 0;
4
        for( int i = 0; i < grid.length; i++ )</pre>
            for( int j = 0; j < grid[0].length; <math>j++)
5
                 if( grid[i][j] == '1' \&\& dfs( grid, i, j ) >= 1 )
6
7
                     counter++;
8
        return counter
9
   }
```

```
10
11
    private int dfs( char[][] grid, int i, int j )
12
        if( i < 0 \mid \mid i >= grid.length \mid \mid j < 0 \mid \mid j >= grid[0].length )
13
14
             return 0;
15
        if( grid[i][j] == '1' )
16
17
             grid[i][j] = '0';
             return dfs( grid, i - 1, j ) + dfs( grid, i, j - 1 ) + dfs( grid, i
18
    +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. */
        public void push(int x)
13
14
15
            stack1.push( x );
16
        }
17
        /** Removes the element from in front of queue and returns that element.
18
    */
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
        /** Get the front element. */
31
        public int peek()
32
33
            if( !stack2.isEmpty() )
34
                return stack2.peek();
35
36
            while( !stack1.isEmpty() )
37
            {
                int t = stack1.pop();
```

```
39
                stack2.push( t );
40
            }
41
            return stack2.peek();
        }
42
43
44
        /** Returns whether the queue is empty. */
45
        public boolean empty()
46
47
            return stack1.isEmpty() && stack2.isEmpty();
48
        }
49
   }
50
   /**
51
52
    * Your MyQueue object will be instantiated and called as such:
    * MyQueue obj = new MyQueue();
53
    * obj.push(x);
54
55
    * int param_2 = obj.pop();
    * int param_3 = obj.peek();
56
57
    * boolean param_4 = obj.empty();
```

## **题目示例9** leetcode 542 01矩阵

// 暂存

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

{
3
4 }
```

# 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 )
1
2
 3
        int left = 0, right = ...;
4
5
        while( ... )
6
            int mid = left + ( right - left ) / 2;
7
8
            if( nums[mid] == target )
9
10
            else if( nums[mid] < target )</pre>
11
                left = ...;
12
            else if( nums[mid] > target )
```

```
      13
      right = ...;

      14
      }

      15
      return ...;

      16
      }

      17
      // 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

      19
      // 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 )
2
3
        int left = 0, right = nums.length - 1;
4
        while( left <= right )</pre>
 5
            int mid = left + ( right - left ) / 2;
 6
7
            if( nums[mid] == target )
                 return mid;
8
9
            else if( nums[mid] < target )</pre>
                 left = mid + 1;
10
11
            else if( nums[mid] > target )
12
                 right = left - 1;
13
        }
14
        return -1;
15 }
```

## 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;
```

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

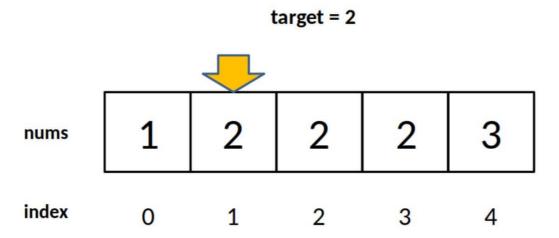
#### 1、while中条件为 left < right 而不是 left <= right的原因

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;</pre>
```

#### 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:

```
1 if( nums[mid] == target )
2 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 )
 2
 3
        if( nums.length == 0 )
 4
            return -1;
 5
        int left = 0, right = nums.length;
 6
 7
        while( left < right )</pre>
8
9
            int mid = left + ( right - left ) / 2;
            if( nums[mid] == target )
10
                left = mid + 1;
11
12
            else if( nums[mid] < target )</pre>
                left = mid + 1;
13
            else if( nums[mid] > target )
14
                right = mid;
15
16
17
        return left - 1;
18 }
```

## 1、算法搜索右侧边界的原理

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

通过不断增大搜索区间的下界 Teft,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:

```
1 | if( nums[mid] == target )
2 | 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;</pre>
```

## 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

```
1 // 基本的二分搜索模板
 2
    private int binarySearch( int[] nums, int target )
 3
        int left = 0, right = nums.length - 1;
 4
 5
        while( left <= right )</pre>
 6
            int mid = left + ( right - left ) / 2;
 7
 8
            if( nums[mid] == target )
 9
                 return mid;
10
            else if( nums[mid] < target )</pre>
                left = mid + 1;
11
            else if( nums[mid] > target )
12
13
                right = mid - 1;
14
        }
15
        return -1;
    }
16
17
    // 搜索左侧边界的二分搜索模板
18
19
    private int leftBound( int[] nums, int target )
20
        int left = 0, right = nums.length - 1;
21
        while( left <= right )</pre>
22
23
            int mid = left + ( right - left ) / 2;
24
25
            if( nums[mid] == target )
26
                 right = mid - 1;
27
            else if( nums[mid] < target )</pre>
                left = mid + 1;
28
```

```
29
            else if( nums[mid] > target )
30
                 right = mid - 1;
31
32
33
        // 检查left 越界情况
34
        if( left >= nums.length || nums[left] != target )
35
             return -1;
36
        return left;
37
    }
38
39
    // 搜索右侧边界的二分搜索模板
40
    private int rightBound( int[] nums, int target )
41
42
        int left = 0, right = nums.length - 1;
43
        while( left <= right )</pre>
44
45
            int mid = left + ( right - left ) / 2;
46
            if( nums[mid] == target )
                left = mid + 1;
47
48
            else if( nums[mid] < target )</pre>
                left = mid + 1;
49
50
            else if( nums[mid] > target )
51
                 right = mid - 1;
        }
52
54
        if( right < 0 || nums[right] != target )</pre>
55
             return -1;
        return right;
56
57 }
```

# 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 )
2
3
4
        int left = 0, right = nums.length - 1;
5
        while( left <= right )</pre>
6
            int mid = left + ( right - left ) / 2;
 7
8
            if( nums[mid] == target )
9
                right = mid - 1;
10
            else if( nums[mid] < target )</pre>
                left = mid + 1;
11
12
            else if( nums[mid] > target )
13
                right = mid - 1;
14
        }
15
        return left;
16 }
```

#### 题目示例2 leetcode 74 搜索二维矩阵

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

```
4
        if( matrix == null || matrix.length == 0 || matrix[0].length == 0 )
 5
             return false:
 6
 7
        int rowLen = matrix.length, colLen = matrix[0].length;
 8
        int start = 0, end = rowLen * colLen - 1;
 9
        while( start <= end )</pre>
10
11
            int mid = start + ( end - start ) / 2;
12
            int curVal = matrix[mid / colLen][mid % colLen];
13
            if( curVal == target )
14
                 return true;
15
            else if( curVal < target )</pre>
                 start = mid + 1;
16
17
            else if( curVal > target )
                 end = mid - 1;
18
19
        }
20
        return false;
21 }
```

# 题目示例3 leetcode 278 第一个错误的版本

```
1 // 寻找左侧边界的二分搜索问题
 2
    private int firstBadVersion( int n )
 3
 4
        int start = 1, end = n;
 5
        while( start < end )</pre>
 6
 7
            int mid = start + ( end - start ) / 2;
           if( isBadVersion( mid ) )
8
9
                end = mid;
10
            else if( !isBadVersion( mid ) )
11
                left = mid + 1;
12
13
        return left;
14 }
```

### 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
 1
 2
    private int findMin( int[] nums )
 3
 4
         if( nums == null || nums.length <= 0 )</pre>
 5
             return -1;
 6
 7
         int left = 0, right = nums.length - 1;
         while( left < right )</pre>
8
 9
             int mid = left + ( right - left ) / 2;
10
             if( nums[mid] <= nums[right] )</pre>
11
12
                 right = mid;
             else if( nums[mid] > nums[right] )
13
14
                 left = mid + 1;
15
         }
```

```
16
    return nums[left];
17
    }
18
19 // v2
20
    private int findMin( int[] nums )
21
22
         int left = 0, right = nums.length - 1;
23
         while( left <= right )</pre>
24
         {
25
             if( nums[left] <= nums[right] )</pre>
26
                 return nums[left];
27
28
             int mid = left + ( right - left ) / 2;
29
             if( nums[left] <= nums[mid] )</pre>
30
                 left = mid + 1;
             else
31
32
                 right = mid;
33
34
         return -1;
35 }
```

#### 题目示例 5 Teetcode 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)
 1
 2
 3
         if( nums == null || nums.length <= 0 )</pre>
 4
             return -1;
 5
         int left = 0, right = nums.length - 1;
 6
 7
         while( left < right )</pre>
 8
         {
 9
             int mid = left + ( right - left ) / 2;
10
             if( nums[mid] < nums[right] )</pre>
11
                 right = mid;
12
             else if( nums[mid] > nums[right] )
13
                 left = mid + 1;
14
             else
15
                 right--;
16
17
         return nums[left];
18 }
```

# 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

```
1
   private int search( int[] nums, int target )
2
3
       if( nums == null || nums.length == 0 )
4
            return -1;
5
       if( nums.length == 1 )
6
            return nums[0] == target ? 0:-1;
7
       int left = 0, right = nums.length - 1;
8
       while( left <= right )</pre>
9
```

```
10
             int mid = left + ( right - left ) / 2;
11
12
             if( nums[mid] == target )
13
                  return mid;
14
             if( nums[0] <= nums[mid] )</pre>
15
16
                  if( nums[0] <= target && target < nums[mid] )</pre>
17
                      right = mid - 1;
18
                  else
19
                      left = mid + 1;
             }
20
21
             else
22
             {
23
                  if( nums[mid] < target && target <= nums[nums.length - 1] )</pre>
24
                      left = mid + 1;
25
                  else
26
                      right = mid - 1;
27
             }
28
         }
29
         return -1;
30 }
```

#### 题目示例 6 leetcode 81搜索旋转排序数组II

```
private boolean search( int[] nums, int target )
 1
 2
    {
 3
         if( nums == null || nums.length == 0 )
 4
             return false;
 5
         int left = 0, right = nums.length - 1;
 6
         while( left <= right )</pre>
 8
 9
             int mid = left + ( right - left ) / 2;
10
             if( nums[mid] == target )
11
                 return true;
12
             if( nums[left] == nums[mid] )
13
14
                 left ++;
15
                 continue;
             }
16
17
18
             if( nums[0] < nums[mid] )</pre>
19
20
                 if( nums[0] <= target && target < nums[mid] )</pre>
21
                      right = mid - 1;
22
                 else
23
                      left = mid + 1;
24
             }
25
             else
26
             {
27
                 if( target > nums[mid] && target <= nums[nums.length - 1] )</pre>
                      left = mid + 1;
28
29
                 else
                      right = mid - 1;
30
31
             }
32
         }
33
         return false;
```

# 动态规划

# 典型题目

# 矩阵类型(10%)

题目示例1 Teetcode 64 最小路径和

```
1 // 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 \rightarrow 0, i )
   // return dp[rowLen-1][colLen-1]
4
6
   // v1
7
    private int minPathSum( int[][] grid )
8
9
        if( grid == null || grid.length == 0 || grid[0].length == 0 )
10
           return 0;
11
12
        int m = grid.length, n = grid[0].length;
13
       int[][] dp = new int[m][n];
       dp[0][0] = grid[0][0];
14
15
        for( int i = 1; i < m; i++)
           dp[i][0] = grid[i][0] + dp[i-1][0];
16
17
        for( int i = 1; i < n; i++)
18
            dp[0][i] = grid[0][i] + dp[0][i-1];
19
20
        for( int i = 1; i < m; i++ )
21
            for( int j = 1; j < n; j++)
22
               dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) + qrid[i][j];
23
        return dp[m-1][n-1];
   }
24
25
26
   // 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
27
28
   // v2
29
    private int minPathSum( int[][] grid )
30
31
        if( grid == null || grid.length == 0 || grid[0].length == 0 )
32
            return 0;
33
        int m = grid.length, n = grid[0].length;
34
35
       int[] dp = new int[n];
        for( int i = 0; i < m; i++)
36
37
        {
38
           for( int j = 0; j < n; j++)
39
               if(j == 0)
40
41
                   dp[j] = dp[j];
                                     // 只能从上侧走到该位置
42
               else if( i == 0 )
43
                   dp[j] = dp[j-1]; // 只能从左侧走到该位置
44
               else
```

#### **题目示例2** Teetcode 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
3
   private int uniquePaths( int m, int n )
4
5
       if( m <= 0 || n <= 0 )
6
           return 0;
 7
       int[] dp = new int[n];
8
9
       Arrays.fill( dp, 1 );
       for( int i = 1; i < m; i++)
10
11
           for( int j = 1; j < n; j++)
12
               dp[j] = dp[j] + dp[j-1];
13
       return dp[n-1];
14 }
```

#### 题目示例2 leetcode 63不同路径II

```
1 // v1
    private int uniquePathWithObstacles( int[][] obstacleGrid )
 2
 3
 4
        int m = obstacleGrid.length, n = obstacleGrid[0].length;
 5
        if( m == 0 || n == 0 )
 6
            return 0;
 7
 8
        int[][] dp = new int[m][n];
 9
        dp[0][0] = obstacleGrid[0][0] == 0? 1: 0;
10
        if(dp[0][0] == 0)
11
            return 0;
                                                            // 起点就是障碍物时无法
    做任何移动
        for( int i = 1; i < m; i++)
12
13
            if( obstacleGrid[i][0] != 1 )
14
                dp[i][0] = dp[i-1][0];
15
        for( int i = 1; i < n; i++)
            if( obstacleGrid[0][i] != 1 )
16
17
                dp[0][i] = dp[0][i-1];
18
19
        for( int i = 1; i < m; i++)
20
            for( int j = 1; j < n; j++)
21
                if( obstacleGrid[i][j] != 1 )
22
                   dp[i][j] = dp[i-1][j] + dp[i][j-1];
23
        return dp[m-1][n-1];
24
    }
25
26
    // v2 空间压缩版本
27
    private int uniquePathWithObstacles( int[][] obstacleGrid )
28
    {
29
        int m = obstacleGrid.length, n = obstacleGrid[0].length;
```

```
if( m == 0 || n == 0 )
30
31
            return 0;
32
33
        int[] dp = new int[n];
34
        dp[0] = 1;
35
        for( int i = 0; i < m; i++)
36
37
            for( int j = 0; j < n; j++)
38
                if( obstacleGrid[i][j] == 1 )
39
                    dp[j] = 0;
            else if(j > 0)
40
41
                dp[j] += dp[j-1];
42
43
        return dp[n-1];
44 }
```

#### 序列类型 (40%)

**题目示例1** leetcode 70 爬楼梯

```
private int climbStairs( int n )
1
2
3
        if(n < 3)
4
            return n;
5
        int[] dp = new int[n+1];
6
7
        dp[1] = 1;
8
        dp[2] = 2;
9
        for( int i = 3; i <= n; i++ )
10
            dp[i] = dp[i-1] + dp[i-2];
        return dp[n];
11
12 }
```

题目示例2 Teetcode 55跳跃游戏

```
1 // dp[i]表示是否能从0跳到i
    // base case:dp[0] = true
 3
   // return dp[nums.length - 1]
    private boolean canJump( int[] nums )
 4
 5
    {
 6
        if( nums.length == 0 )
 7
            return true;
 8
9
        boolean[] dp = new boolean[nums.length - 1];
10
        dp[0] = true;
11
        for( int i = 1; i < nums.length; i++ )</pre>
            for( int j = 0; j < i; j++)
12
                if( dp[j] && nums[j] + j >= i )
13
14
                    dp[i] = true;
15
        return dp[nums.length - 1];
16 }
```

题目示例3 Teetcode 45跳跃游戏II

```
1 // dp[i]表示从0跳到i的最小次数
     // base case: dp[0] = 0
     // return dp[nums.length-1]
  4
     // v1 出现了超时
  5
     private int jump( int[] nums )
  6
  7
         int[] dp = new int[nums.length];
  8
         dp[0] = 0;
  9
         for( int i = 1; i < nums.length; i++ )</pre>
 10
 11
                               // 最大值为i,相当于从0开始每次跳一步到达当前位置
             dp[i] = i;
             for( int j = 0; j < i; j++)
 12
 13
                 if(j + nums[j] >= i)
 14
                     dp[i] = Math.min(dp[j] + 1, dp[i]);
 15
         return dp[nums.length-1];
 16
 17
     }
 18
     // v2
 19
 20
     private int jump( int[] nums )
 21
 22
         int res = 0, end = 0, maxPos = 0;
 23
         for( int i = 0; i < nums.length - 1; i++)
 24
         {
 25
             maxPos = Math.max( maxPos, nums[i] + i );
 26
             if(i == end)
 27
 28
                 end = maxPos;
 29
                 res++;
 30
             }
 31
         }
 32
         return res;
 33 }
```

#### 题目示例4 Teetcode 132 分割回文串

```
1  // 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
2  // base case : dp[0] = -1;
3  // return: dp[s.length()-1]
4  private int minCut( String s )
5  {
6
7  }
```

#### 题目示例5 Teetcode 300最长上升子序列

```
1
    private int lengthOfLIS( int[] nums )
2
 3
        if( nums == null || nums.length == 0 )
4
            return 0;
 5
6
        int[] dp = new int[nums.length];
7
        Arrays.fill( dp, 1 );
8
        for( int i = 0; i < nums.length; i++ )</pre>
            for( int j = 0; j < i; j++)
9
10
                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** Teetcode 139 单词拆分

#### 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

```
1 // 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]
5
   // v1
   private boolean canPartition( int[] nums )
6
7
8
       int sum = 0;
9
       for( int num:nums )
10
           sum += num;
11
       if( sum % 2 != 0 )
12
           return false;
13
14
       int n = nums.length;
15
       sum = sum / 2;
16
       boolean[][] dp = new boolean[n+1][sum+1];
17
       for( int i = 0; i <= n; i++)
18
           dp[i][0] = true;
19
20
       for( int i = 1; i <= n; i++ )
21
           for( int j = 1; j \le sum; j++)
22
               if(j - nums[i-1] < 0)
                   dp[i][j] = dp[i-1][j];
23
24
25
                   dp[i][j] = dp[i-1][j] | dp[i-1][j-nums[i-1]];
26
       return dp[n][sum];
27
   }
28
29
   // v2 状态压缩
   private boolean canPartition( int[] nums )
30
31
   {
32
       int sum = 0;
       for( int num:nums )
33
           sum += num;
```

```
if( sum % 2 != 0 )
35
36
             return false;
37
38
        int n = nums.length;
39
        sum \neq 2;
40
        boolean[] dp = new boolean[sum+1];
41
        dp[0] = true;
42
43
        for( int i = 1; i <= n; i++ )
44
            for( int j = sum; j >= 0; j--)
                if(j - nums[i-1] >= 0)
45
                     dp[j] = dp[j] \mid dp[j-nums[i-1]];
46
47
        return dp[sum];
48 }
```

#### 题目示例2 Teetcode 322零钱兑换

```
1 // dp[i]定义: 当目标金额为i时,最少需要的硬币数量为dp[i]
2
    private int coinChange( int[] coins, int amount )
3
4
       int[] dp = new int[amount+1];
 5
       Arrays.fill( dp, amount+1 );
6
       dp[0] = 0;
7
       for( int i = 0; i < dp.length; i++)
           for( int coin:coins )
8
9
               if(i - coin >= 0)
10
                   dp[i] = Math.min(dp[i], dp[i-coin] + 1);
        return dp[amount] == amount + 1? -1:dp[amount];
11
12 }
```

# 题目示例3 leetcode 518 零钱兑换II

```
1
    private int change( int amount, int[] coins )
2
    {
 3
        int n = coins.length;
4
        int[] dp = new int[amount+1];
5
        dp[0] = 1;
        for( int i = 0; i < n; i++)
6
            for( int j = 1; j \leftarrow amount; j++)
8
                if(j - coins[i] >= 0)
9
                     dp[j] = dp[j] + dp[j-coins[i]];
10
        return dp[amount];
11 | }
```

#### 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

```
1 /**
2 * 构建如下的dp数组
3 \times 0 <= n <= n - 1, 1 <= k <= K
4 * n为天数,K为最大可交易次数
5
   * 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
6
7
   */
8 | dp[i][k][0 or 1];
9
   for( int i = 0; i < n; i++)
10
     for( int k = 1; k \le K; k++)
11
         for( int s :{0, 1} )
12
            dp[i][k][s] = Math.max(buy, sell, rest);
13
   // 最终需要求解得到的答案是:dp[n-1][K][0]
14
15
   // 根据分析得到如下的状态转移方程
16
17
   *解释:今天不持有股票(s = 0 )有两种原因:
18 * 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
19 * 2.昨天就持有股票
20 */
21 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
22
                                                );
23
   /**
24
25 * 解释: 今天持有股票( s = 1 )有两种原因:
26 * 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
27 * 2.昨天未持有股票,今天选择buy(购入股票)
28 * 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,
29 * 以购入股票为标志代表使用了一次交易机会
30 */
31 dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
32
          = Math.max( 选择rest , 选择buy
                                                     );
33
   /**
34
35
   * 定义base case
36
37 | 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
   38
39
   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
40
   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
41
42
43 /**
44
   * 状态转移方程总结
45 */
   // base case
46
47 | dp[-1][k][0] = dp[i][0][0] = 0;
dp[-1][k][1] = dp[i][0][1] = Integer.MIN_VALUE;
   // 状态转移
49
50 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]);
```

```
class Solution
 1
 2
 3
        public int maxProfit(int[] prices)
 4
 5
            int n = prices.length;
 6
            int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
            for( int i = 0; i < n; i++)
 7
 8
            {
                dpI0 = Math.max( dpI0, dpI1 + prices[i] );
9
10
                dpI1 = Math.max( dpI1, -prices[i] );
11
            }
12
            return dpIO;
13
14 }
```

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
 2
    {
 3
        public int maxProfit(int[] prices)
 4
 5
            int n = prices.length;
            int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
 6
 7
            for( int i = 0; i < n; i++)
 8
9
                int temp = dpIO;
                dpI0 = Math.max( dpI0, dpI1 + prices[i] );
10
                dpI1 = Math.max( dpI1, temp - prices[i] );
11
12
            }
13
            return dpIO;
14
        }
15 }
```

题目示例3 Teetcode 123 买卖股票的最佳时机III

分析: K = 2,遍历所有的K值

```
class Solution
 1
 2
 3
        public int maxProfit(int[] prices)
 4
        {
 5
             if( prices == null || prices.length < 2 )</pre>
 6
                 return 0;
 7
             int maxK = 2;
 8
            int n = prices.length;
9
             int[][][] dp = new int[n][maxK+1][2];
             for( int i = 0; i < n; i++)
10
11
                 for( int k = 1; k \le \max k; k++)
                     if(i == 0)
12
13
                     {
14
                         dp[i][k][0] = 0;
15
                         dp[i][k][1] = -prices[i];
```

```
16
17
                     else
18
                     {
19
                        dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] +
    prices[i] );
20
                         dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0]
    - prices[i] );
21
22
            return dp[n-1][maxK][0];
23
        }
   }
24
```

#### 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<math>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 R = 12

```
1
    class Solution
 2
    {
 3
        public int maxProfit(int k, int[] prices)
 4
 5
            if( prices == null || prices.length < 2 || k < 1 )</pre>
 6
                 return 0;
 7
            int n = prices.length;
 8
            if(k > n / 2)
 9
                 return maxProfitWithInfiniteK( prices );
10
            else
11
                 return maxProfitWithLimitedK( k, prices );
12
        }
13
14
        private int maxProfitWithInfiniteK( int[] prices )
15
            int n = prices.length;
16
            int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
17
18
            for( int i = 0; i < n; i++)
19
             {
20
                 int temp = dpIO;
                 dpI0 = Math.max( dpI0, dpI1 + prices[i] );
21
22
                 dpI1 = Math.max( dpI1, temp - prices[i] );
23
            }
24
            return dpIO;
25
        }
26
27
        private int maxProfitWithLimitedK( int K, int[] prices )
28
29
            int n = prices.length;
30
            int[][][] dp = new int[n][K+1][2];
            for( int i = 0; i < n; i++)
31
32
                 for( int k = 1; k \le K; k++)
33
                     if(i == 0)
34
                     {
35
                         dp[i][k][0] = 0;
36
                         dp[i][k][1] = -prices[i];
37
                     }
38
                     else
39
                     {
```

题目示例5 Teetcode 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

```
1
    class Solution
 2
    {
 3
        public int maxProfit(int[] prices)
 4
 5
            int n = prices.length;
 6
            int presell = 0;
 7
            int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
            for( int i = 0; i < n; i++)
 8
 9
10
                int temp = dpIO;
                dpI0 = Math.max( dpI0, dpI1 + prices[i] );
11
                dpI1 = Math.max( dpI1, preSell - prices[i] );
12
13
                preSell = temp;
14
            }
15
            return dpIO;
16
        }
17
   }
```

题目示例6 leetcode 714 买卖股票的最佳时机含手续费

```
1
    class Solution
 2
 3
        public int maxProfit(int[] prices, int fee)
4
        {
 5
            int n = prices.length;
            int dpI0 = 0, dpI1 = Integer.MIN_VALUE;
 6
            for( int i = 0; i < n; i++)
8
            {
9
                int temp = dpIO;
10
                dpI0 = Math.max( dpI0, dpI1 + prices[i] );
                dpI1 = Math.max( dpI1, temp - prices[i] - fee );
11
12
13
            return dpIO;
        }
14
15 }
```

leetcode 打家劫舍系列问题

**题目示例1** leetcode 198 打家劫舍

```
1 // v1
2 /**
3 * dp[i] = x表示:
```

```
* 从第i间房子开始抢劫,最多能抢到的钱为 x
 5
    * base case: dp[n] = 0
 6
    */
 7
    class Solution
8
9
        public int rob(int[] nums)
10
11
            int n = nums.length;
            int[] dp = new int[n+2];
12
13
            for( int i = n - 1; i >= 0; i--)
14
                dp[i] = Math.max(dp[i+1], dp[i+2] + nums[i]);
15
            return dp[0];
16
        }
17
    }
18
19
    // v2
20
    class Solution
21
        public int rob(int[] nums)
22
23
24
            int n = nums.length;
25
            int dpTwo = 0, dpOne = 0;
26
            int dpCur = 0;
            for( int i = n - 1; i >= 0; i--)
27
28
29
                dpCur = Math.max( dpOne, dpTwo + nums[i] );
30
                dpTwo = dpOne;
                dpOne = dpCur;
31
32
            }
33
            return dpCur;
34
        }
35 }
```

#### 题目示例2 leetcode 213 打家劫舍II

```
1
    class Solution
 2
 3
        public int rob(int[] nums)
 4
 5
            if( nums.length == 1 )
 6
                 return nums[0];
 7
            int res1 = robInRange( nums, 0 , nums.length - 2 );
 8
             int res2 = robInRange( nums, 1, nums.length - 1 );
 9
             return res1 > res2 ? res1:res2;
        }
10
11
        private int robInRange( int[] nums, int start, int end )
12
13
14
            int dpOne = 0, dpTwo = 0;
            int dpCur = 0;
15
16
            for( int i = end; i >= start; i-- )
17
18
                 dpCur = Math.max( dpOne, nums[i] + dpTwo );
19
                 dpTwo = dpOne;
20
                 dpOne = dpCur;
21
22
             return dpCur;
```

```
23 | }
24 | }
```

## 题目示例2 leetcode 337 打家劫舍III

```
1 /**
2
    * Definition for a binary tree node.
3
    * public class TreeNode {
4
          int val;
5
          TreeNode left;
6
          TreeNode right;
7
          TreeNode(int x) { val = x; }
    * }
8
     */
9
    class Solution
10
11
12
       public int rob(TreeNode root)
13
14
           int[] res = robInTree( root );
           return Math.max( res[0], res[1] );
15
16
        }
17
        /**
18
19
        * 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
20
21
       * arr[1]表示抢当前root的话,得到最大钱数
       */
22
23
        private int[] robInTree( TreeNode root )
24
           if( root == null )
25
26
               return new int[]{ 0, 0 };
27
28
           int[] leftSub = robInTree( root.left );
29
           int[] rightSub = robInTree( root.right );
30
           int notRob = Math.max( leftSub[0], leftSub[1] ) + Math.max(
    rightSub[0], rightSub[1] );
            int rob = root.val + leftSub[0] + rightSub[0];
33
            return new int[]{ notRob, rob };
       }
34
35 }
```

# 算法思维

# 回溯法

简单的回溯法模板

```
result := []
2
   func backTrack(选择列表,路径){
      if 满足结束条件 {
3
4
          result.add(路径)
5
          return
6
      }
7
      for 选择 in 选择列表 {
8
9
          做选择
          backTrack(选择列表,路径)
10
11
          撤销选择
      }
12
13 }
```

# 典型题目

#### **题目示例1** leetcode78 子集

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

#### 题目示例2 leetcode 90 子集II

```
List<List<Integer>> res = new LinkedList<>();
2
    public List<List<Integer>> subsetsWithDup( int[] nums )
 3
4
        if( nums == null || nums.length == 0 )
 5
            return res;
6
7
        Arrays.sort( nums );
        backTracking( nums, 0 , new LinkedList<Integer>() );
8
9
        return res;
10
    }
```

```
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++ )</pre>
17
            if( i > start && nums[i] == nums[i-1] )
18
19
                continue;
20
            // 做选择
21
            runner.add( nums[i] );
            // 进入下一层决策树
22
23
            backTracking( nums, i + 1, runner );
24
            // 撤销选择
25
            runner.removeLast();
26
        }
27 }
```

#### **题目示例3** Teetcode 46 全排列

```
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
    private void backTracking( int[] nums, LinkedList<Integer> runner )
11
12
13
        if( runner.size() == nums.length )
14
15
            res.add( new LinkedList( runner ) );
16
            return;
17
18
        for( int i = 0; i < nums.length; i++ )</pre>
19
20
        {
            if( runner.contains( nums[i] ) )
21
22
                continue;
23
24
            // 做选择
25
            runner.add( nums[i] );
            // 进入下一层决策树
26
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<>();
```

```
public List<List<Integer>>> permuteUnique( int[] nums )
 3
 4
        if( nums == null || nums.length == 0 )
 5
             return res;
 6
 7
        boolean[] used = new boolean[nums.length];
 8
        Arrays.sort( nums );
        backTracking( nums, used, new LinkedList<Integer>() );
 9
10
        return res;
11
    }
12
13
    private void backTracking( int[] nums, boolean[] used, LinkedList<Integer>
    runner )
14
    {
15
        if( runner.size() == nums.length )
16
17
             res.add( new LinkedList( runner ) );
18
             return;
19
        }
20
        for( int i = 0; i < nums.length; i++ )</pre>
21
22
23
            if( used[i] )
24
                continue;
25
            if( i > 0 && nums[i] == nums[i-1] && !used[i-1] )
26
                 continue;
27
            // 做选择
28
29
             runner.add( nums[i] );
30
            used[i] = true;
31
            // 进入下一层决策树
32
            backTracking( nums, used, runner );
33
            // 撤销选择
34
            used[i] = false;
35
             runner.removeLast();
36
        }
37
    }
```

#### 题目示例5 leetcode 77 组合

```
1
    List<List<Integer>> res = new LinkedList<>();
 2
    public List<List<Integer>> combine( int n, int k )
 3
    {
4
        if( n < 1 || n < k )
 5
            return res;
6
 7
        backTracking( n, k, 1, new LinkedList<Integer>() );
8
        return res;
9
    }
10
11
    private void backTracking( int n, int k, int start, LinkedList<Integer>
    runner )
12
    {
        if( runner.size() == k )
13
14
15
            res.add( new LinkedList( runner ) );
16
            return;
```

```
17
18
19
        for( int i = start; i <= n; i++ )
20
21
            // 做选择
22
            runner.add( i );
23
            // 进入下一层决策树
24
            backTracking( n, k, i + 1, runner );
25
            // 撤销选择
26
            runner.removeLast();
27
        }
28 }
```

#### 题目示例6 leetcode 39 组合总和

```
List<List<Integer>> res = new LinkedList<>();
 2
    public List<List<Integer>> combinationSum( int[] candidates, int target )
 3
 4
        if( candidates == null || candidates.length == 0 )
 5
             return res;
 6
 7
        Arrays.sort( candidates );
 8
        backTracking( candidates, target, 0, new LinkedList<Integer>() );
 9
        return res;
10
    }
11
    private void backTracking( int[] candidates, int target, int start,
12
    LinkedList<Integer> runner )
    {
13
14
        if( target == 0 )
15
        {
16
             res.add( new LinkedList( runner ) );
17
             return;
18
        }
19
20
        for( int i = start; i < candidates.length; i++ )</pre>
21
22
            if( target - candidates[i] < 0 )</pre>
23
                break;
24
25
            // 做选择、
             runner.add( candidates[i] );
26
27
            // 进入下一层决策树
28
            backTracking( candidates, target - candidates[i], i, runner );
29
            // 撤销选择
30
            runner.removeLast();
        }
31
32
   }
```

#### **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;
```

```
6
 7
        Arrays.sort( candidates );
 8
        backTracking( candidates, target, 0, new LinkedList<Integer>() );
9
        return res;
10
    }
11
12
    private void backTracking( int[] candidates, int target, int start,
    LinkedList<Integer> runner )
13
14
        if( target == 0 )
15
16
            res.add( new LinkedList( runner ) );
17
            return;
        }
18
19
20
        for( int i = start; i < candidates.length; i++ )</pre>
21
22
            if( target - candidates[i] < 0 )</pre>
23
                break;
24
            if( i > start && candidates[i] == candidates[i-1] )
25
                continue;
26
27
            // 做选择
            runner.add( candidates[i] );
28
29
            // 进入下一层决策树
            backTracking( candidates, target - candidates[i], i + 1, runner );
30
31
            // 撤销选择
32
            runner.removeLast();
33
        }
34 }
```

#### 题目示例8 leetcode 216 组合总和III

```
List<List<Integer>> res = new LinkedList<>();
 2
    public List<List<Integer>> combinationSum3( int k, int n )
 3
 4
        if( n \le 0 | | k \le 0 )
 5
            return res;
 6
 7
        backTracking( k, n, 1, new LinkedList<Integer>() );
 8
        return res;
 9
    }
10
    private void backTracking( int k, int n, int start, LinkedList<Integer>
11
    runner)
12
    {
        // 终止条件
13
14
        if(k == 0)
15
        {
            if(n == 0)
16
17
                 res.add( new LinkedList( runner ) );
18
            return;
19
        }
20
21
22
        for( int i = start; i < 10; i++ )
23
        {
```

```
      24
      // 做选择

      25
      runner.add(i);

      26
      // 进入下一层决策树

      27
      backTracking(k-1,n-i,i+1,runner);

      28
      // 撤销选择

      29
      runner.removeLast();

      30
      }

      31
      }
```

#### 题目示例9 leetcode37 解数独

```
public void solveSudoku( char[][] board )
1
2
    {
3
        backTracking( board, 0, 0 );
    }
4
 5
    private boolean backTracking( char[][] board, int i, int j )
6
7
8
        int row = 9, col = 9;
9
10
        // 穷举到最后一列,进入下一行重新开始
11
        if(j == col)
12
            return backTracking( board, i + 1, 0 );
13
        // 找到一个可行解, 触发base case
14
15
        if(i == row)
16
            return true;
17
        // 当前位置已经有数字,不再穷举数字
18
        if( board[i][j] != '.' )
19
20
            return backTracking( board, i, j + 1 );
21
        for( char c = '1'; c <= '9'; c++ )
22
23
           if( !isValid( board, i, j, c ) )
24
               continue;
25
26
           // 做选择
27
            board[i][j] = c;
28
           // 进入下一层决策树
29
            if( backTracking( board, i, j + 1 ) )
30
                return true;
31
            board[i][j] = '.';
32
33
        return false;
    }
34
35
    private boolean isValid( char[][] board, int row, int col, char c )
36
37
    {
38
        for( int i = 0; i < 9; i++)
39
40
            // 判断行是否有重复
41
           if( board[row][i] == c )
                                       return false;
            // 判断列是否有重复
42
43
           if( board[i][col] == c )
                                      return false;
            // 判断3x3方框是否存在重复
44
45
           if(board[(row/3)*3 + i/3][(co1/3)*3 + i%3] == c)
46
                return false
```

```
47 | }
48 | return true;
49 |}
```

#### **题目示例10** leetcode22 括号生成

```
1 // 这个题目有两个关键性质
2
   // 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<>();
5
   public List<String> generateParenthesis( int n )
6
7
       if(n \ll 0)
8
          return res;
9
       backTracking( n, n, new StringBuffer() );
10
11
       return res;
12
   }
13
14
   private void backTracking( int left, int right, StringBuffer s )
15
       // 剩下的左括号更多,说明不合法
16
17
       if( left > right )
18
           return;
19
       // 数量小于0,不合法
20
       if( left < 0 || right < 0 )
21
           return;
       // 所有括号都能用完,得到一个合法的括号组合
22
23
       if( left == 0 && right == 0 )
24
       {
25
           res.add( s.toString() );
26
           return;
27
       }
28
29
       // 尝试放置一个左括号
30
       // 选择
       s.append( '(' );
31
32
       // 进入下一层决策树
33
       backTracking( left - 1, right, s );
34
       // 撤销选择
       s.deleteCharAt( s.length() - 1 );
35
36
37
       // 尝试放置一个右括号
38
       // 选择
39
       s.append( ')' );
40
       // 进入下一层决策树
41
       backTracking( left, right - 1, s );
42
       // 撤销选择
43
       s.deleteCharAt( s.length() - 1 );
44 }
```

**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

```
List<List<String>> res = new LinkedList<>();
2
    public List<List<String>> partition( String s )
3
        backTracking( s, 0, new LinkedList<String> runner );
4
 5
        return res;
6
    }
 7
    private void backTracking( String s, int start, LinkedList<String> runner )
8
9
10
        if( start == s.length() )
11
12
             res.add( new LinkedList( runner ) );
13
            return;
14
        }
15
16
        for( int i = start; i < s.length(); i++ )</pre>
17
            if(!isPalindrome(s, start, i))
18
19
                continue;
20
            // 选择
21
22
            runner.add( s.substring( start, i + 1 ) );
23
            // 进入下一层决策树
24
            backTracking( s, i + 1, runner );
25
            // 撤销选择
26
            runner.removeLast();
27
        }
28
    }
29
    private boolean isPalindrome( String s, int left, int right )
30
31
32
        while( left < right )</pre>
33
34
            if( s.charAt( left ) != s.charAt( right ) )
                return false;
35
36
            left++;
37
            right--;
38
        }
39
        return true;
40 | }
```

# 滑动窗口技巧

# 简单的滑动窗口模板

```
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;
```

```
10
        int valid = 0;
11
        while( right < s.length() )</pre>
12
        {
13
           // c是移入窗口的字符
14
           char c = s.charAt( right );
15
           // 右移窗口
16
           right++;
17
           // 进行窗口内数据的一系列更新
18
           . . . ;
19
           /*** debug的输出位置 ***/
20
21
           System.out.println( "window:[%d, %d ]", left, right );
22
23
           // 判断左侧窗口是否需要收缩
24
           while( window needs shrink )
25
26
               // d是将被移出窗口的字符
27
               char d = s.charAt( left );
28
               // 窗口左侧右移
29
               left++;
               // 进行窗口内数据的一系列更新
30
31
               . . . ;
32
           }
33
        }
34
   }
```

# 典型题目

#### 题目示例1 leetcode 76 最小覆盖子串

```
private String minWindow( String s, String t )
 1
 2
 3
        if( s == null || t == null || s.length() < t.length() )</pre>
            return "";
4
 5
        HashMap<Character, Integer> need = new HashMap<>();
 6
        HashMap<Character, Integer> window= new HashMap<>();
 7
 8
        for( char c : t.toCharArray() )
9
            need.put( c, need.getOrDefault( c, 0 ) + 1 );
10
11
        int left = 0, right = 0;
12
        int valid = 0;
13
        // 记录最小覆盖子串的起始索引及长度
        int start = 0, len = Integer.MAX_VALUE;
14
15
        while( right < s.length() )</pre>
16
        {
17
            // c是移入窗口的字符
            char c = s.charAt( right );
18
19
            // 右移窗口
20
            right++;
            // 进行窗口内数据的一系列更新
21
22
            if( need.containsKey( c ) )
23
            {
                window.put( c, window.getOrDefault( c, 0 ) + 1 );
24
25
                if( window.get( c ).equals( need.get( c ) ) )
                    valid++;
26
27
            }
```

```
28
29
            // 判断左侧窗口是否要收缩
30
            while( valid == need.size() )
31
            {
32
                // 更新最小覆盖子串
33
                if( right - left < len )</pre>
34
35
                    start = left;
36
                   len = right - left;
37
                }
38
               // d是将被移出窗口的字符
39
               char d = s.charAt( left );
40
               // 右移窗口左侧
41
               left++;
42
                // 进行窗口内数据的一系列更新
43
               if( need.containsKey( d ) )
44
45
                    if( window.get( d ).equals( need.get( d ) ) )
46
                        valid--;
47
                    window.put( d, window.get(d) - 1 );
48
               }
            }
49
50
        return len == Integer.MAX_VALUE? "":s.substring( start, start + len );
51
52
   }
```

#### **题目示例2** leetcode 567 字符串的排列

```
private boolean checkInclusion( String s1, String s2 )
1
2
    {
 3
        if( s2.length() < s1.length )</pre>
4
            return false;
 5
 6
        HashMap<Character, Integer> need = new HashMap<>();
7
        HashMap<Character, Integer> window = new HashMap<>();
8
        for( char c:s1.toCharArray() )
9
            need.put( c, need.getOrDefault( c, 0 ) + 1 );
10
11
        int left = 0, right = 0;
12
        int valid = 0;
13
        while( right < s2.length() )</pre>
14
15
            char c = s2.charAt( right );
16
            right++;
17
            if( need.containsKey(c) )
18
            {
                window.put( c, window.getOrDefault( c, 0 ) + 1 );
19
20
                if( window.get( c ).equals( need.get( c ) ) )
21
                     valid++;
22
            }
23
            while( (right - left) >= s1.length() )
24
25
            {
                // 在这里判断是否找到了合法的子串
26
27
                if( valid == need.size() )
28
                     return true;
29
```

```
30
                 char d = s2.charAt( left );
31
                 left++;
                 if( need.containsKey( d ) )
32
33
                 {
34
                     if( window.get( d ).equals( need.get( d ) ) )
35
36
                     window.put( d, window.get(d) - 1 );
37
                 }
            }
38
39
        return false;
40
    }
41
```

#### 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 )
 2
 3
        List<Integer> res = new LinkedList<>();
4
        if( s.length() < p.length() )</pre>
 5
             return res;
 6
        HashMap<Character, Integer> need = new HashMap<>();
 8
        HashMap<Character, Integer> window = new HashMap<>();
9
        for( char c:p.toCharArray() )
10
             need.put( c, need.getOrDefault( c, 0 ) + 1 );
11
12
        int left = 0, right = 0;
13
        int valid = 0;
        while( right < s.length() )</pre>
14
15
            char c = s.charAt( right );
16
17
             right++;
            if( need.containsKey( c ) )
18
19
            {
20
                 window.put( c, window.getOrDefault( c, 0 ) + 1 );
                 if( window.get(c).equals( need.get(c) ) )
21
22
                     valid++;
            }
23
24
25
            while( right - left >= p.length() )
26
            {
27
                 if( valid == need.size() )
                     res.add( left );
28
29
                 char d = s.charAt( left );
30
                 left++;
31
                 if( need.containsKey( d ) )
32
33
                     if( window.get(d).equals( need.get(d) ) )
34
                         valid--;
                     window.put( d, window.get(d) - 1 );
35
36
                 }
37
             }
38
39
        return res;
40
    }
```

```
private int lengthOfLongestSubstring( String s )
 2
 3
        if( s == null \mid \mid s.length() == 0)
 4
             return 0;
 6
        HashMap<Character, Integer> window = new HashMap<>();
        int left = 0, right = 0;
 8
        int res = 0;
 9
        while( right < s.length() )</pre>
10
        {
11
             char c = s.charAt( right );
12
             right ++;
13
            window.put( c, window.getOrDefault( c, 0 ) + 1 );
14
            while( window.get(c) > 1 )
15
             {
16
                 char d = s.charAt( left );
17
                 left++;
                 window.put( d, window.get(d) - 1 );
18
19
20
             res = Math.max( res, right - left );
21
        }
22
        return res;
23 }
```

# 前缀和技巧

# 前缀和简单定义

```
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]</pre>
```

# 题目示例1 leetcode 1 两数之和

```
1
    private int[] towSum( int[] nums, int target )
 2
 3
        int n = nums.length;
4
        HashMap<Integer, Integer> hashmap = new HashMap<>();
 5
6
        for( int i = 0; i < n; i++)
 7
        {
8
            int cur = nums[i];
            if( hashmap.containsKey( target - cur ) )
9
                return new int[]{ hashmap.get( target - cur ), i };
10
11
            hashmap.put( nums[i], i );
12
        }
```

```
13 return new int[2];
14 }
```

# 题目示例2 leetcode 560 和为K的子数组

```
private int subarraySum( int[] nums, int k )
 2
 3
        int n = nums.length;
 4
        int res = 0;
        HashMap<Integer, Integer> preSum = new HashMap<>();
 6
        preSum.put( 0, 1 );
 7
        int curSum = 0;
 8
9
        for( int i = 0; i < n; i++)
10
11
            curSum += nums[i];
12
            if( preSum.containsKey( curSum - k ) )
13
                res += preSum.get( curSum - k );
14
            preSum.put( curSum, preSum.getOrDefault( curSum, 0 ) + 1 );
15
16
        return res;
17 | }
```

# 题目示例3 leetcode 1248 统计优美子数组

```
private int numberOfSubArrays( int[] nums, int k )
 1
 2
    {
 3
        int n = nums.length;
 4
        int[] preSum = new int[n+1];
        preSum[0] = 1;
 6
        int res = 0, curSum = 0;
 8
        for( int num:nums )
9
        {
10
            curSum += num & 1;
11
            preSum[curSum]++;
12
            if( curSum >= k )
13
                res += preSum[curSum-k];
14
15
        return res;
16 }
```

# 题目示例4 leetcode 974 和可被K整除的子数组

```
1 private int subarrayDivByK( int[] A, int K )
2 {
3    4 }
```