腾讯50题（按照提交通过率高 -->低排）

链表问题

如果是单链表遍历

```
while(p != null) {
    p = p.next;
}
// 这样会遍历到最后一个节点的下一个节点，即空节点。

while(p.next != null) {
    p = p.next;
}
//这样会遍历到最后一个节点,p 不会指向空节点
```

对于快慢指针指向链表，一定要记住**判断条件**

> 对于快慢指针一起走，一定要记住这个！！！
>
> 开始的时候**相同**
>
> 进入循环后，开始走，走完之后再判断

```
ListNode quick = head;
ListNode slow = head;
while (true) {
    if (quick == null || quick.next == null)
        return null;
    slow = slow.next;
    quick = quick.next.next;
    if (quick == slow) {
        break;
    }
}
```

快排

- 要有递归出口（left < right）
- 判断的时候 `i < j && nums[j] >= pivot`

```
public int[] sortArray(int[] nums) {
        return quickSort(nums, 0, nums.length - 1);
    }
    public int[] quickSort(int[] nums, int left, int right) {
        if (left < right) {     // 注意
            int i = left;
            int j = right;
            int pivot = nums[i];
            while(i < j) {
                 while(i < j && nums[j] >= pivot) { // 注意
                     j--;
                 }
                 nums[i] = nums[j];
```

```
                while(i < j && nums[i] <= pivot) {
                    i++;
                }
                nums[j] = nums[i];
            }
            nums[i] = pivot;
            quickSort(nums, i+1, right);
            quickSort(nums, left, j-1);
        }
        return nums;
    }
```

二分法

二分法有两种形式，一种是 while() { }形式，另外一个就是自身调用自身的形式

> 对于二分法，只要记住，一般都是 `<=` 、 `>=`
>
> 只有在寻找最开始出现目标值左侧的下标、寻找最开始出现目标值右侧的下标 这种情况下才是使用的 `<` `>`不使用等于。

```
// 最普通的二分法查找
private static int myBinarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {
            return mid;
        }else if (nums[mid] > target) {
            right = mid - 1;
        }else if (nums[mid] < target) {
            left = mid + 1;
        }
    }
    // 因为是 <= ，所以会找遍所有的值，都不符合条件，那么跳出循环
    // 直接返回 -1 即可
    return -1;
}
```

```
// 寻找最开始出现目标值左侧的下标
private static int myBinarySearchByLeft(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;

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

```
    // 因为 right = mid  这一行语句的存在，所以不能够 使用 while (left <= right)，否则可能会进入死循环
    // 当跳出循环的时候，就是 left == right，此时再判断 left 下标值是否等于 target 值
    if (nums[left] == target) {
        return left;
    } else {
        return -1;
    }
}
```

栈 队列

> java 中的队列

```
Queue queue = new LinkedList();
// 入队
queue.offer(e);
// 出队
queue.poll();
// 查看队首元素
queue.peek();

Queue queue = new LinkedList();
queue.offer(2);
queue.offer(21);
queue.offer(3);
queue.offer(4);
System.out.println(queue);              // [2, 21, 3, 4]
System.out.println(queue.peek());       // 2

System.out.println(queue.poll());       // 2
System.out.println(queue.peek());       // 21
```

> java 中的栈

```
// 入栈
stack.push(x);
// 出栈
stack.pop();
// 查看栈顶元素
stack.peek();

Deque stack = new LinkedList();
stack.push(1);
stack.push(2);
stack.push(3);

System.out.println(stack);          // [3, 2, 1]
System.out.println(stack.pop());    // 3
System.out.println(stack.peek());   // 2
```

[Deque用法]

子序列和子数组

> 1 子序列 是不要求连续的
>
> 2 子数组和子串一样，是需要连续的

StringBuilder类

```
public static void main(String[] args) {
    StringBuilder sb = new StringBuilder();
    String s = "aaa";

    sb.append(s);
    sb.append("bbb");    // 在末尾添加

    sb.deleteCharAt(sb.length() - 1);    // 删除最后一个字符

    System.out.println(sb.toString());  // 转化为 String 类型
}
```

> 一般使用 StringBuilder 类来对字符串操作， 这样比较方便

```
StringBuffer reverse();                    // 字符串反转
string substring(start,end);               // 返回start至end-1的子串
```

字符串操作

```
String substring(int start,int end);       // 从start开始到end为止。// 包含start
位，不包含end位。
boolean contains(String substring);        // 字符串中包含指定的字符串吗？
boolean equals(string);                    // 覆盖了Object中的方法，判断字符串内容是
否相同。
string substring(start,end);               // 返回start至end-1的子串
s.charAt(i)                                // 查看下标 i 的值
```

java中的字符、字符串及数字之间的转换（转）

```
一、string 和int之间的转换

1、string转换成int  :Integer.valueOf("12")

2、int转换成string : String.valueOf(12)



二、char和int之间的转换
```

```
1、首先将char转换成string

String str=String.valueOf('2')

2、转换

Integer.valueof(str) 或者Integer.PaseInt(str)

Integer.valueof返回的是Integer对象，Integer.paseInt返回的是int
```

59、螺旋矩阵

> 按照从左到右，从上到下，从右到左，从下到上的顺序
>
> 分别看上、右、下、左四条边

46、78. 全排列

> 按照回溯的标准模板来写。
>
> ```
> void backtracking(参数) {
>  if (终止条件) {
>      存放结果;
>      return;
>  }
>
>  for (选择：本层集合中元素（树中节点孩子的数量就是集合的大小）) {
>      处理节点;
>      backtracking(路径，选择列表); // 递归
>      回溯，撤销处理结果
>  }
> }
> ```

344、557. 反转字符串

> 最好将字符串变成字符数组，这样比较好操作
>
> 切记：不可出现下面错误
>
> ```
> char tmp = s.charAt(start);
> s.charAt(start) = s.charAt(end);
> s.charAt(end) = tmp;        // 因为s.charAt(start)、s.charAt(end)都是值类型的，不是变量，不可以这样赋值
> ```
>
> 两个比较重要的转换类型：
>
> ```
> char [] arr = s.toCharArray();  // 字符串转数组
> String.valueOf(arr);    // 数组转字符串
> ```

230、求二叉搜索树中倒数第 k 个元素

```
// 求树节点的总数
```

```
private int deepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int lCount = deepth(root.left);
    int rCount = deepth(root.right);
    return lCount + rCount + 1;
}

// 求树的最大深度
public int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int lHeight = maxDepth(root.left);
    int rHeight = maxDepth(root.right);
    return Math.max(lHeight, rHeight) + 1;
}
```

215、数组中倒数第 k 个最大元素

> 快速排序
>
> 1、首先一个 `if` 判断，递归出口
>
> 2、再循环的时候，需要保证 `i < j` 这个条件
>
> 3、判断的时候是 `>=` 或者 `<=`

```
private int quickSort(int[] nums, int begin, int end, int k) {
    if (begin < end) {
        // 如果区间不止一个数
        int i = begin;
        int j = end;
        int pivot = nums[begin];
        while (i < j) {
            while (i < j && pivot <= nums[j]) {
                j--;
            }
            nums[i] = nums[j];
            while (i < j && pivot >= nums[i]) {
                i++;
            }
            nums[j] = nums[i];
        }
        nums[i] = pivot;
        if (i == nums.length - k) {
            return nums[i];
        } else if (i < nums.length - k) {
            // 在右边
            quickSort(nums, i+1, end, k);
        } else {
            quickSort(nums, begin, i-1, k);
        }
    }
    return nums[nums.length-k];
```

```
    }
```

148、排序链表

23、合并K个升序链表

他们用的方法都是一样的，都是二分法找到两个链表，然后将这两个链表进行排序。

前端

树

递归遍历

```
var inorderTraversal = function(root) {
    var res = []

    function inorder(root) {
        if(root == null) {
            return null
        }
        inorder(root.left)
        res.push(root.val)
        inorder(root.right)
    }
    inorder(root)
    return res
};
```

非递归遍历

先序遍历（144）

```
var preorderTraversal = function(root) {
    var res = []
    var stack = []
    // 先进行判断根是否为空
    if(root != null) {
        stack.push(root)
    }
    while(stack.length > 0) {
        root = stack.pop()
        res.push(root.val)
        // 先右孩子 后左孩子
        if(root.right != null) {
            stack.push(root.right)
        }
        if(root.left != null) {
            stack.push(root.left)
        }
    }
```

```
    return res
};
```

中序遍历（94）

```
var inorderTraversal = function(root) {
    var res = []
    var stack = []
    // 只有一个 while
    while(stack.length > 0 || root != null) {
        if(root != null) {
            stack.push(root)
            root = root.left
        } else {
            root = stack.pop()
            res.push(root.val)
            root = root.right
        }
    }
    return res
};
```

后续遍历

层次遍历（102）

```
var levelOrder = function(root) {
    var res = []
    var queue = []
    // // 先进行判断根是否为空
    if(root != null) {
        queue.push(root)
    }
    while(queue.length > 0) {
        var currentLen = queue.length
        var tmp = []
        while(currentLen > 0) {
            var node = queue.shift()
            tmp.push(node.val)
            if(node.left != null) {
                queue.push(node.left)
            }
            if(node.right != null) {
                queue.push(node.right)
            }
            currentLen--
        }
        res.push(tmp)
    }
    return res
};
```

滑动窗口

滑动窗口

> 滑动窗口题型
>
> - 一般找最大窗口，那么直接找可行解即可（3、寻找无重复的最长子串）
> - 找最小值，分为两步，第一步：找到可行解；第二步：找到最优解

> 包含目标字符串的最小连续子串

```
// 统计的区间是【left,right】
var lengthOfLongestSubstring = function(s) {
    var set = new Set()
    var max = 0
    var left = 0
    var right = -1
    while (right < s.length && left < s.length) {
        // 第一步：找到可行解
        // 第二步：找到最优解（本题不需要）
        // 为什么是 right + 1?
        // 因为right 是已经放进来的，需要看下一个字符是否在里面，当然，他的下标也要小于 n
        while (right + 1 < s.length && !set.has(s.charAt(right + 1))) {
            set.add(s.charAt(right + 1))
            right++
        }
        // 要在删除之前统计
        max = Math.max(max, right - left + 1)

        set.delete(s.charAt(left++))

    }
    return max
};

// 方法二（推荐）
// 统计的区间是 【left,right)
var lengthOfLongestSubstring = function(s) {
    var set = new Set()
    var left = 0
    var right = 0
    var maxLen = 0
    while (right < s.length) {
        // 第一步：找到可行解
        // 第二步：找到最优解（本题不需要）
        while(right < s.length && !set.has(s.charAt(right))) {
            set.add(s.charAt(right))
            right++
        }
        // 开始出现重复的子串了，但是还没有加进去
        // 因为统计的是最长的，所以要先计算
        maxLen = Math.max(maxLen, right - left)
        set.delete(s.charAt(left++))
    }
```

```
    return maxLen
};
```

> 209、长度最小的子数组

```
var minSubArrayLen = function(target, nums) {
    var left = 0
    var right = 0
    var sum = 0
    var min = Infinity
    while(right < nums.length) {
        sum += nums[right]
        right++
        // 注意：这里是 >=
        while(sum >= target) {
            min = Math.min(min, right-left)
            sum -= nums[left]
            left++
        }
    }
    if(min === Infinity) {
        min = 0
    }
    return min
};
```

动态规划

买卖股票的最佳时机

```
// max{之前的最大利润，第 i 天的价格 - 最底价格}

var maxProfit = function(prices) {
    var max = 0
    minPrice = prices[0]
    for(var i = 0; i < prices.length; i++) {
        max = Math.max(max, prices[i] - minPrice)
        minPrice = Math.min(minPrice, prices[i])
    }
    return max
};
```

最大子序和

```
var maxProfit = function(prices) {
    var max = 0
    minPrice = prices[0]
    for(var i = 0; i < prices.length; i++) {
        max = Math.max(max, prices[i] - minPrice)
        minPrice = Math.min(minPrice, prices[i])
    }
    return max
};
```

爬楼梯

```
var climbStairs = function(n) {
    if(n <= 2) {
        return n
    }
    var dp = []
    dp[0] = 0
    dp[1] = 1
    dp[2] = 2
    for(var i = 3; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2]
    }
    return dp[n]
};
```

最长递增子序列

```
 // dp[i] 表示以 nums[i] 结尾的最长上升子序列的长度
int dp[] = new int[nums.length];

// 使用 res 来存储最大的 dp[i]
int res = dp[0];

// 将每个 dp 的值 初始化为 1 ， 因为最小值有一个
for(int i = 0; i < nums.length; i++) {
    dp[i] = 1;
}

for(int i = 0; i < nums.length; i++) {
            for(int j = 0; j < i; j++) {
                if(nums[j] < nums[i]) {
                    // 不判断的话最后dp[i]的值可能是偏小的。
                    // 比如说 当前元素的Index是100。遍历dp[j]的时候（此时nums[i] >
nums[j]）
                    // j =88时，dp[88]是较大值；j=99，dp[99]是较小值。
                    // 如果直接将 dp[i] = dp[j] + 1 的话，那最后取得是dp[99] + 1,而不
是dp【88】，就错了
                    // 遍历前面值的时候，都会记录下来，和之前的值比较，取最大
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
```

```
            res = Math.max(res, dp[i]);
        }
        return res;
    }
```

最长重复子数组

```
if (A.length == 0 || B.length == 0) {
    return 0;
}
int res = 0;
int[][] dp = new int[A.length + 1][B.length + 1];
for (int i = 1; i <= A.length; i++) {
    for (int j = 1; j <= B.length; j++) {
        if (A[i-1] == B[j-1]) {
            dp[i][j] = dp[i-1][j-1] + 1;
            res = Math.max(dp[i][j], res);
        }
    }
}
return res;
```

最小路径和

```
if (grid.length == 0) {
    return 0;
}
int row = grid.length;
int column = grid[0].length;
int[][] dp = new int[row][column];
dp[0][0] = grid[0][0];
for(int i = 1; i < row; i++) {
    // 沿着列走的，取每一个数组的第一个元素
    dp[i][0] = dp[i-1][0] + grid[i][0];
}
for(int i = 1; i < column; i++) {
    dp[0][i] = dp[0][i-1] + grid[0][i];
}

for(int i = 1; i < row; i++) {
    for(int j = 1; j < column; j++) {
        dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
    }
}
return dp[row-1][column-1];
```

不同路径

```
public int uniquePaths(int m, int n) {
    int [][] arr = new int [m][n];
    for(int i = 0; i < m; i++) {
        arr[i][0] = 1;
    }
    for(int i = 0; i < n; i++) {
        arr[0][i] = 1;
    }
    for(int i = 1; i < m; i++) {
        for(int j = 1; j < n; j++) {
            arr[i][j] = arr[i-1][j] + arr[i][j-1];
        }
    }
    return arr[m-1][n-1];
}
```

打家劫舍

```
if (nums.length == 0)
    return 0;
if (nums.length == 1)
    return nums[0];

int[] dp = new int[nums.length];
dp[0] = nums[0];
dp[1] = Math.max(nums[0], nums[1]);

for(int i = 2; i < nums.length; i++) {
    dp[i] = Math.max(nums[i] + dp[i-2], dp[i-1]);
}
return dp[nums.length-1];
```

二分查找

常见的排序算法及其时间复杂度

1、快排

2、堆排序

3、归并排序

4、冒泡排序