前端面试编程题

Javascript 编程题

目的

考查应聘者的编程能力和逻辑思维能力。

Talk is cheap, show me the code.

注意事项

1. 让候选人看完题目后先思考，让他先谈谈解题的思路，如果这个思路靠谱的话再去写程序。#为了让面试者表现得更好#
2. 最少两道编程题，如果时间充足，多面试几道。为节约时间，只需要上机一题即可，其他在白纸上写，甚至都可以不用上机编程。

1. 编写函数 parse，解析 url 的参数成一个数组。

例如

var a = '?token=abc&id=123'; 执行 parse(a) 返回 {'token':'abc', 'id':123}
var b = '?name=john&age'; 执行 parse(b) 返回 {'name':'john', 'age':null}

涉及知识点：

1. split
2. substring
3. parseInt
4. decodeURIComponent
5. Number 类
6. Array 类
7. 强制类型转换，如+number 转成数字
8. 正则表达式

可能的解法：

1. 正则
2. split 拆分然后循环

扩展：

1. var a = 'token[]=a&token[]=b&id=123'; 执行 parse(a) 返回 {'token':['a', 'b'], 'id':123}

2. 给数组原型增加一个方法 strip，用于去掉数组中的重复元素。

例如

var a = ['a', 'b', 'a', 8, 3, 5, 8]; a.strip(); => a 为 ['a', 'b', 8, 3, 5];
var b = ['apple', 'banana', 'apple', 'grape']; b.strip(); => b 为 ['apple', 'banana', 'grape'];

涉及知识点：

1. Array.prototype
2. 数组的常用操作，如 splice, push

可能的解法：

1. 两层循环
2. 一层循环

扩展：

1. 区分数字和字符串
   var a = ['a', 'b', 'a', 8, 3, 5, '8', 8]; a.strip(); => a 为 ['a', 'b', 8, 3, 5, '8'];
2. 区分数字、字符串和对象实例
   function A(){}
   A.prototype.toString = function() {return '8'; };
   var c = new A();
   var a = ['a', 'b', 'a', 8, 3, 5, '8', c, 8]; a.strip(); => a 为 ['a', 'b', 8, 3, 5, '8', c];

3. 编写通用字符串替换函数 replace。

例如

var a = 'hi, my name is {name}, I am {age} years old, my email is {email}.';
var b = {name:'max', age: 12, email: 'max@gmail.com'};
执行 replace(a, b) 返回'hi, my name is max, I am 12 years old, my email is max@gmail.com.'

4. 算出字符串中出现次数最多的字符是什么，出现了多少次。(难度偏小)

5. 求 1000！(难度偏大)

```
/**
 * 大数
 * @class BigNumber
 * @param {Number} num
```

```
 */
function BigNumber(num) {
    this.digits = this._serialize(num);
}
BigNumber.prototype._serialize = function (num) {
    var digits = [];
    while (num > 0) {
        digits.push(num % 10);
        num = Math.floor(num / 10);
    }
    return digits;
};
BigNumber.prototype.multiply = function (num) {
    var digits = this.digits,
        nums = this._serialize(num),
        i, j, len = digits.length, lenNums = nums.length,
        product, carry,
        result = [];

    for (i = 0; i < lenNums; ++i) {
        for (j = 0; j < len; ++j) {
            product = nums[i] * digits[j] + (result[i + j] || 0);
            carry = Math.floor(product / 10);
            result[i + j] = product % 10;
            if (carry > 0) {
                result[i + j + 1] = carry + (result[i + j + 1] || 0);
            }
        }
    }
    this.digits = result;

    return this;
};
BigNumber.prototype.format = function () {
    return this.digits.concat().reverse().join('');
};


/**
 * 求阶乘
 * @method factorial
 * @param {Number} num
 * @return {String}
 */
function factorial(num) {
    var n = new BigNumber(num);
    while (--num > 0) {
        n.multiply(num);
    }
    return n.format();
}


assert.equal(factorial(1000),
'402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601
```

```
14182780946464962910563938874378864873371191810458257836478499770124766328898
35955735432513185323958463075557409114262417474349347553428646576611667797396
66882029120737914385371958824980812686783837455973174613608537953452422158659
32019280908782973084313928444032812315586110369768013573042161687476096758713
48312025478589320767169132448426236131412508780208000261683151027341827977704
78463586817016436502415369139828126481021309276124489635992870511496497541990
93422215668325720808213331861168115536158365469840467089756029009505376164758
47728421889679646244945160765353408198901385442487984959533191017233555566602
13945039973628075013783761530712776192684903435262520001588853514733161170210
39681759215109077880193931781141945452572238655414610628921879602238389714760
88506276862967146674697562911234082439208160153780889893964518263243671616762
17916890977991190375403127462228998800519544441428201218736174599264295658174
66283029555702990243241531816172104658320367869061172601587835207515162842255
40265170483304226143974286933061690897968482590125458327168226458066526769958
65268227280707578139185817888965220816434834482599326604336766017699961283186
07883861502794659551311565520360939881806121385586003014356945272242063446317
97460594682573103790084024432438465657245014402821885252470935190620929023136
49327349756551395872055965422874977401141334696271542284586237738753823048386
56889764619273838149001407673104466402598994902222217659043399018860185665264
85061799702356193897017860040811889729918311021171229845901641921068884387121
85564612496079872290851929681937238864261483965738229112312502418664935314397
01374285319266498753372189406942814341185201580141233448280150513996942901534
83077644569099073152433278288269864602789864321139083506217095002597389863554
27719674282224875758676575234422020757363056949882508796892816275384886339690
99598262809561214509948717012445164612603790293091208890869420285106401821543
99457156805941872748998094254742173582401063677404595741785160829230135358081
84009699637252423056085590370062427124341690900415369010593398383577793941097
00277534720000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000', 'factorial failed');
```

6. 二分查找(难度偏大)

例如

var a = [1, 3, 5, 7, 9, 42, 77, 80, 89]; // 查找 5 返回 2, 查找 42 返回 5，查找 100 返回-1

变种：查找有序数组中比 x 大且最接近 x 的数值项下标，如果没有返回-1

7. 求任意数组中最大的两个数

例如

var a = [1, 31, 5, 27, 9, 42, 89, 80, 77]; // 最大两个数为 89, 80

8. 取 8 个大小不同的数字中第二大数

评判算法优劣的标准是最坏情况下比较次数最少（相同时间复杂度也会有常数项/倍数项的差别）。

1. 排序
   最简单的一种方案，最坏情况下比较次数为（快速排序最坏情况等同冒泡，所以以冒泡排序为标准）：28 次
   如果应聘者连这种方式都不能想出，坚决 pass
2. 遍历两次
   排序改进版。因为只需要取到第二大数，所以可以用两次遍历直接获得。最坏情况下比较次数（也是最好情况）：13 次
   如果应聘者连这种方式都不能想出，应该 pass
3. 改进堆排序
   先建好最大堆（1 + 2 * 3 = 6 次），此时堆顶为最大值，然后最后一个与最大值互换，重新最大化堆（2 * 2 次），堆顶即为第二大数。最坏情况下比较次数（也是最好情况）：10 次
   如果应聘者可以自己想出这种方式，weak hire
4. 两个变量遍历一次
   空间换时间。使用 max、min 两个变量，max 存储前两个数中较大的，min 存储较小的，然后此次用后面的 6 个数和 max 比，如果比 max 大，则 min=max，max 设为当前数，如果比 max 小，再与 min 比。。。最坏情况下比较次数为 13 次，最好情况为 7 次。平均比较次数要比第二种方案好一些
   如果应聘者可以自己想出这种方式，weak hire
5. 分治
   方法比较多，有以下几种：
   1. 划分为两组，各选最大
      划分为两组，每组四个，然后各选出最大的（3＋3＝6 次），两组最大的进行比较（1 次），然后取较小的那个与较大的那个所在组剩余的 3 个进行比较（4 个中选出最大的，3 次），最坏情况下比较次数为（也是最好情况）：10 次
      如果应聘者可以自己想出这种方式，hire。如果在引导下可以想出这种方式，weak hire
   2. 划分为两组，各选前两大
      划分为两组，每组四个，然后各选出前两大的（最坏 5＋5＝10 次，最好 3＋3＝6 次），两组最大的进行比较（1 次），然后取较小的那个与较大的那个组第二大的进行比较（1 次），最坏情况下比较次数为：12 次，最好情况下比较次数为 8 次
      如果应聘者可以自己想出这种方式，hire。如果在引导下可以想出这种方式，weak hire

3. 划分为四组
   划分为四组，每组两个，各选出最大的，这样共选出 4 个，然后这四个再两两分为两组，各选最大的，这样选出 2 个，然后再从这两个中选出最大的，第二大的必然是被第一大的比较下去的，因为一共三轮比较，所以可以通过找到这三个中最大的得到第二大数。最坏情况下比较次数为（也是最好情况）：9 次
   这个方法应该是最优方案。如果应聘者可以很快想出这种方式，strong hire。如果在引导下可以想出这种方式，hire

6. 泛化
   如果应聘者很 NB，可以问一些泛化的问题，例如如何取任意数组的第 i 大数，可以考察这些点：
   1. i 越接近数组中位数优化空间越小，i 越接近两侧则优化空间越大
   2. 考虑各种排序算法和数组自身特征等方面的关系。例如基本有序的数组适合用优化过的冒泡算法，使用快排非常糟糕等

9. 合并两个有序数组

已知两个有序的数组，将它们合并成也是有序的一个数组。举例如下：
输入：A = [1, 2, 4, 7], B = [3, 5, 6, 8]，输出 C = [1, 2, 3, 4, 5, 6, 7, 8]。

最简单解法：A.concat(B).sort()，只能答出这个为不及格。

要求不允许使用数组的任何内置方法，解法五花八门，常见解法如下：

```
function merge(a, b) {
    var c = [],
        i = 0, j = 0,
        lenA = a.length, lenB = b.length;
    while (i < lenA && j < lenB) {
        c.push(a[i] < b[j] ? a[i++] : b[j++]);
    }
    if (i < lenA) {
        c.push.apply(c, a.slice(i));
    }
    if (j < lenB) {
        c.push.apply(c, b.slice(j));
    }
    return c;
}
```

扩展 1：根据这个函数写出归并排序算法，并计算该算法的复杂度。

扩展 2：现在有 K 个数组，A = [1, 2, 4, 7], B = [3, 5, 6, 8]，C = [9, 18, 20, 21], D = [10, 11, 13, 19] .....K = [15, 16, 22, 25, 26]，合并成 X = [1, 2, 3, 4, 5, ...]。

#思路 1：K 个数组从 A 到 K，依次用前面的写出来的函数合并
#思路 2：K 个数组中间 F 从 A，从 G 到 K，依次用前面的写出来的函数合并，最后再合并一次
#思路 3：取出 K 个数组的第一个元素开始建堆，每次从堆中取出最小的值，然后重新建堆。

扩展 3：现在 K 为 20，每个数组大小为 1K，但内存只有 10K，磁盘有 1M，如何处理。

10. 数据格式转化

```
[
  { id: 1, name: 'text1', pid: 0 },
  { id: 2, name: 'text2', pid: 1 },
  { id: 3, name: 'text3', pid: 2 },
  { id: 4, name: 'text4', pid: 8 },
  { id: 5, name: 'text5', pid: 4 }
]
```

转换后得到

```
[
{id: 1, name:'text1', pid: 0, members: [
        {id: 2, name: 'text2', pid: 1, members: [
                {id: 3, name: 'text3', pid: 2, members: [] }
        ]}
]},
{id: 4, name: 'text4', pid: 0, members: [
        {id: 5, name: 'text5', pid: 4, members: [] }
]}
]
```

这个题目里面有很多可以考察的点， 在面试中也很少有候选人能思维缜密的给出清晰的思路， 比较有区分度

1、一般思路：先找根节点，再进行递归（大多数人都能想到）

- 如何找根节点，考察数据结构的灵活运用（大多数的人会想到双重循环）
- 如何递归，考察候选人的思维逻辑， （顺带可以考察一下数据结构中的深度遍历和广度遍历）
- 考察算法的复杂度

2、 将整个数据结构转换为便于查找的 Hash 结构， 再利用引用关系的对象在传参时值的改变会影响原值的特性

- 可以很简单的达到转换的目的，大大降低复杂度；
- 也可以考察对语言特性的掌握情况

11. 设计题目：AutoComplete

考察点：

功能方面：

- 数据的获取（异步或同步），JSON 数据格式（同步意味着搜索时由前端处理匹配）
- 搜索的频率，事件、输入法的兼容
- 拼音匹配
- 数据缓存

设计方面：

- 数据和 DOM 节点的双向绑定
- 扩展设计，如加入选择功能
- 代码组织
- 其它

11. 按层遍历 JSON 数据

问题：获取第 n 层的节点

扩展：顺序获取第 0-n 层的节点，并依次装入数组

```
var s = {
 a1: {
  b1: {c1:1},
  b2: {c2:1}
 }
};
```

则第 0-2 层的节点依次为：

a1

b1、b2

c1、c2

扩展题说明

要求返回结果为

```
[['a1'], ['b1','b2'],['c1','c2']]
```

考察点:

1、树形结构遍历（递归和非递归实现均可考察）

2、函数柯里化（闭包）

参考实现代码

```
//获取第 n 层节点，递归实现
var f = function(s,n) {
    var array = [];
    if (n < 0 || !s) {
        return array;
    }
    if (n === 0) {
        for (var p in s) {
            if (s.hasOwnProperty(p)) {
                array.push(p);
            }
        }
        return array;
    }
    for (var p in s) {
        if (s.hasOwnProperty(p)) {
            array = array.concat(f(s[p], n-1));
        }
    }
    return array;
}
```

扩展题参考编码：

```
//顺序获取第 0-n 层的节点，要求尽可能高效，避免重复遍历
//这是递归实现，思路是直接调用上面的获取第 n 层节点的函数，然后通过柯里化实现缓存提高效率。
//需要将上面的代码修改下:
//array.push(p);改为 array.push({value: s[p], key: p});
var getAllLeafByOrder = (function () {
    var cache = [];
    return function (s, n) {
        var array = [];
        for (var i = 0; i <= n; i++) {
            if (cache[i-1]) {
                var temp = [];
                for (j = 0; j < cache[i-1].length; j++) {
                    var subObj = cache[i-1][j].value;
                    temp = temp.concat(f(subObj, 0));
                }
```

```
                cache[i] = temp;
                array.push(temp);
                continue;
            }
            cache[i] = f(s, i);
            array.push(cache[i]);
        }
        //取 key
        for (var k = 0; k < array.length; k++) {
            var row = array[k];
            for (var l = 0; l < row.length; l++) {
                row[l] = row[l].key;
            }
        }
        return array;
    }
})();
```

如果能用 while 实现，而非递归，建议 strong hire，难度大。

最好先令其作流程图说明思路，再看代码。

如下例所示，思路如下：

建立一个数组 array，从根节点起每次将所有子节点 push 进 array。依次遍历子节点，若其也包含子节点，则继续 push 到 array。如此，array 内的元素就都是按层级的顺序排列了。在遍历下一级子节点之前，收集本级节点所有的 key。

```
//顺序获取第 0-n 层的节点，要求尽可能高效，避免重复遍历
function getLeafs(s, n) {
    var result = [];
    var array = [];
    array.push(s);
    var cur = 0;
    var last = 1;
    while(cur < array.length) {
        last = array.length;
        //收集本级节点的 key
        var temp = [];
        for (var i = cur; i < last; i++) {
            temp = temp.concat(Object.keys(array[i]));
        }
        result.push(temp);
        //将下一级子节点依次平铺到 array
        while(cur < last) {
            for (var p in array[cur]) {
                if (array[cur].hasOwnProperty(p) &&
Object.prototype.toString.call(array[cur][p]) === '[object Object]') {
                    array.push(array[cur][p]);
                }
            }
            cur++;
```

```
        }
    }
    return result;
}
```

12. 统计二进制中 0 和 1 的分布

给一个数字，请输出其二进制表示中 0 和 1 的分布，连续的数字标记为 "起始位-结束位"

比如说给定 2014，其二进制表示是 11111011110，要求输出 '1: 0-4, 0: 5, 1: 6-9, 0: 10'

考察正则表达式的基本知识。

```
function distribution(num) {
        var str = num.toString(2);
        var x = /(0+)|(1+)/g;
        var c = [];

        while (b = x.exec(str)) {
                res = +!b[1] + ": " + b.index;
                if (b[0].length !== 1) res += "-" + (x.lastIndex - 1);

                c.push(res);
        }
        return c.join(",");
}


# 2014 -> '1: 0-4, 0: 5, 1: 6-9, 0: 10'
distribution(2014);
```

遍历字符串的方法也可以接受。

13. 自定义事件

实现自定义事件，要求如下

```
on('lunchReady', function (menu) {
        console.log('I\'m so hungry!');
});
on('lunchReady', function (menu) {
        if (menu.indexOf('Hamburg') !== -1) {
                console.log('Wow! I like Hamburg!');
        }
});
```

```
fire('lunchReady', ['Hamburg', 'Coffee', 'Pizza']);
// output:
// I'm so hungry!
// Wow! I like Hamburg!
```

进阶：

- stopImmediatePropagation
- fire once
- add target

考察点：

- sub/pub 模式
- 对事件内部机制的深度理解