**判断一个字符串是否回文**

**利用js数组实现**

js的数组是一个很强大的数据结构，我们可以活用其已实现的原生方法做很多事，比如，这个例子中，判断一个字符串是否是回文。

**步骤：**

* 将字符串拆分成数组  
  将字符串拆分成数组其实也也有多种方法：
  + split()方法

|  |  |
| --- | --- |
| 1  2  3 | let str\_to\_array = function(str){  return str.split('');  } |

* + Array.prototype.map()

|  |  |
| --- | --- |
| 1  2  3 | let str\_to\_array = function (str) {  return Array.prototype.map.call(str,function(x){return x});  } |

* 数组反转 reverse()
* 拼接成字符串 join()

|  |  |
| --- | --- |
| 1  2  3 | let checkPalindrom = function(str){  return str\_to\_array(str).reverse().join('');  } |

**活用数组的reduceRight()方法**

我们可以直接在字符串上调用数组的reduceRight()方法将字符串逆转

|  |  |
| --- | --- |
| 1  2  3 | let rs = Array.prototype.reduceRight.apply('abc',[function(pre,current){  return pre + current;  },'']); |

|  |  |
| --- | --- |
| 1  2  3 | let checkPalindrom = function(str){  return str == rs(str);  } |

**使用栈数据结构**

我们在学习栈这个数据结构的时候，老师讲的最生动的一个例子就是，判断回文有木有。  
先将字符串中字符依次入栈，然后出栈组成新的字符串，即为逆转的字符串，然后做比较。

**去掉一些整型数组中重复的值**

**直接使用es6的Set**

|  |  |
| --- | --- |
| 1  2  3 | let unique = function(array){  return [...new Set(array)];  } |

**使用Object**

我们知道Object对象的键是唯一的，可以利用这个特性为数组去重。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | let unique = function (array) {  let ro = {};  let ra = [];  array.forEach(item=>{  if(!ro[item]){  ro[item] = item;  ra.push(item);  }  });    return ra;  } |

**统计一个字符串出现最多的字母**

**首先我们要先统计字符串中各个字符出现的次数，我们可以使用最笨的遍历方法进行统计：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | let countChar = function countChar(str){  let ro = {};  for(let c of str){  if(!ro[c]){  ro[c] = 1;  }else{  ro[c] ++;  }  }  return ro;  } |

当然，也使用数组的reduce()方法进行统计，因为这个方法就适合进行统计计算。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | let countChar = function countChar(str){  return Array.prototype.reduce.call(str,function(pre,current){  if(pre[current]){  pre[current] ++;  }else{  pre[current ] = 1;  }  return pre;  },{});  } |

**然后，从刚才统计出的数中查找出出现次数最多的字符**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | let findMaxDuplicateChar = function (str){  let chars = countChar(str);  let max = 0;  let char = null;  for(let c in chars){  if(chars[c] > max){  max = chars[c];  char = c;  }  }  return char;  } |

**不用临时变量，交换两个变量的值**

原文中呢，作者教大家要合理运用+,-运算，最后给出如下答案：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function swap(a , b) {  b = b - a;  a = a + b;  b = a - b;  return [a,b];  }  module.exports = swap; |

很巧妙，对吧，确实是合理运用了+,-运算，但是为什么呢？合理运用\*,/运算呢？

|  |  |
| --- | --- |
| 1  2  3 | a = a \* b;  b = a / b;  a = a / b; |

合理运用\*,/好像也可以啊，对吧。  
其实解决问题，我们应该从根上去解决，不能简单的说’合理运用’就敷衍过去了。  
题目是，不用临时变量，临时变量是干嘛的呢？当然是存储临时值用的了，对吧。  
那么，不用临时变量，我们可以把临时值存储到当前现有的变量中，对吧。  
就好像是创造了一个临时变量一样。上面两个例子中，都是用两个变量的差或两个变量的积作为临时值，然后存储到其中一个变量，再由相应的运算交换两个变量的值。  
明白了这个道理后，我们再合理一下嘛，对吧，利用两个变量的和作为临时值：

|  |  |
| --- | --- |
| 1  2  3 | a = a + b;  b = a - b;  a = a - b; |

注意：这里慎用两个变量的商作为临时值，因为如果两个变量除不尽，由于js中除法运算会舍掉余数，则会发生问题。

我们除了使用+,-,\*,/四则运算创造‘临时变量’外，还可以使用位运算

|  |  |
| --- | --- |
| 1  2  3 | a = a ^ b；  b = b ^ a；  a = a ^ b； |

这个比上面的四则运算就要稍难理解了，这里运用了位运算中的异或运算。  
对于异或运算的说明，还有不明白的可以回去翻翻大学的课本。  
第一行 a = a ^ b;即创造了一个临时值存储在a中。

b = b ^ a

相当于

b = b ^ (a ^ b) = b ^ b ^ a = a

同理，

a = a ^ b

相当于

a = (a ^ b) ^ (b ^ (a ^ b)) = a^b^b^a^b = a^a^b^b^b = 0^0^b = b

**找出数组中最大差值**

可以直接遍历数组，找出最大值和最小值，然后做差。但是呢，那样就没意思了，对吧，我们可以直接使用数组的reduce()方法找出最大值和最小值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | let getMaxProfit = function getMaxProfit(arr) {  let max\_min = arr.reduce(function(pre,current){  if(pre.min > current){  pre.min = current;  }  if(pre.max < current){  pre.max = current;  }  return pre;  },{min:arr[0],max:arr[0]});    return max\_min.max - max\_min.min;  } |

当然，使用reduce()貌似还是有点麻烦啊，js的Math对象不是有max()和min()方法嘛，直接用这两个方法找出最大值和最小值就好了啊：

|  |  |
| --- | --- |
| 1  2  3  4  5 | let getMaxProfit = function getMaxProfit(arr){  let max = Math.max.apply(Math,arr);  let min = Math.min.apply(Math,arr);  return max - min;  } |

这里使用了apply方法直接调用max()和min()来获取最大值和最小值。  
我们都知道js中apply和call两个方法是功能相同的两个方法，只是传参方式不同。call方法传递参数列表，而apply传递参数对象或数组。  
在es6中新添加了一个...操作符，用于将数组展开成列表，具体可参见[MDN上的文档](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator" \t "_blank)。  
因此，我们这里可以使用该操作符，直接调用max()和min()方法：

|  |  |
| --- | --- |
| 1  2  3  4  5 | let getMaxProfit = function getMaxProfit(arr){  let max = Math.max(...arr);  let min = Math.min(...arr);  return max - min;  } |

**位操作**

20世纪70年代末到80年代末，Digital Equipment的VAX计算机是一种非常流行的机型。它没有布尔运算AND和OR指令，只有bis(位设置)和bic(位清除)这两个指令。两种指令的输入都是一个数据字x和一个掩码字m。他们生成一个结果z，z是由根据掩码m的位来修改x的位得到的。使用bis指令，就是在m为1的每个位置上，将z对应的位设置为1.使用bic指令，在m为1的每个位置，将z对应的位设置为0。  
只使用bis和bic指令，完成按位|和^运算。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //bis和bic声明  int bis(int x,int m);  int bic(int x,int m);    //完成如下 | 运算 和 ^ 运算  int bool\_or(int x,int y){  int result = \_\_\_\_\_\_ ;  return result;  }    int bool\_xor(int x,int y){  int result = \_\_\_\_\_\_ ;  return result;  } |

这其实是一道逻辑题目，由已知的bis运算逻辑和bic运算逻辑，用这两种运算去实现其他的运算。  
bis运算就是在掩码位为1的位设置为1，其他位不变。而bic运算是在掩码位为1的位置设置为0，其他不变。  
举个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | x 11010100  m 10100101  bis --------  11110101      x 11010100  m 10100101  bic --------  01010000 |

好了，那怎么利用这两种运算指令实现按位 | 和 ^ 运算呢？  
仔细分析一下 bis 运算，所有掩码位为1的位，结果都是1，其他为0的位，还是按照原来的位，这不就是按位 | 运算么？  
于是，第一个有了：

|  |  |
| --- | --- |
| 1  2  3  4 | int bool\_or(int x,int y){  int result = bis(x,y) ;  return result;  } |

那 ^ 运算呢？  
我们都知道，异或运算运算法则为：a⊕b=(¬a∧b)∨(a∧¬b),a⊕b=(¬a∨b)∧(a∨¬b)。至于为什么，可以列真值表验证。  
这里我们采用第一个运算法则：a⊕b=(¬a∧b)∨(a∧¬b)。  
bic运算是怎么来的呢？bic(a,b)是将a上对应b为1的位变为0，其他不变。那么，不就是相当于先将b取反，然后和a进行按位与&运算么，就是：

|  |  |
| --- | --- |
| 1 | bic(a,b) = a & (~b) |

于是，再利用异或第一个运算法则：a⊕b=(¬a∧b)∨(a∧¬b)  
我们可以得出：

|  |  |
| --- | --- |
| 1  2  3  4 | int bool\_xor(int x,int y){  int result = bis(bic(x,y),bic(y,x));  return result;  } |

**1.排序**

一般都是给个数组然后排序，有的从小到大，有的从大到小。一定要看清楚。以下都是从小到大的排序算法。

**冒泡法**

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var arr = [3,6,1,2,5];  var temp;  for(var i= 0;i<arr.length;i++){  for(var j=i+1;j<arr.length;j++){  if(arr[i] > arr[j]){  temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  }  console.log(arr); |

**快速排序法**

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | function quicksort (arr){   if(arr.length<=1){   return arr;   }   var left = [];   var right = [];   var middle = arr[0];   for(var i=1;i<arr.length;i++){   if(arr[i]<middle){   left.push(arr[i]);   }else{   right.push(arr[i]);   }   }   return quicksort(left).concat([middle],quicksort(right));  } |

**注意：可以用快速就不要用冒泡。实在没记住才用冒泡。（因为快速排序设计到递归，面试官更多是想考察你递归算法）**

**2.数组去重**

这题考察的是你会不会存储数组元素的出现次数来解决去重问题。当然解法也有很多，下面是其中一种解法。

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Array.prototype.unique = function(){   var res = [];   var json = {};   for(var i = 0; i < this.length; i++){   if(!json[this[i]]){   res.push(this[i]);   json[this[i]] = 1;   }   }   return res;  }  var arr = [112,112,34,'你好',112,112,34,'你好','str','str1'];  alert(arr.unique()); |

**3.js的拷贝**

这题涉及到的就是你能不能清楚的分辨深拷贝和浅拷贝。

var a = {name:'Tom'};  var b = a;  b.name = 'Peter';

请问a.name = ？

正确答案是Peter，如果你的答案是Tom的话，那么你要好好去看看js的深拷贝。

如果要被拷贝的是数组：

slice和concat都可以直接让数组进行深拷贝

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2 | arr.slice();  arr.concat(); |

下面是解法。当然肯定有比我写得更好的。

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function deepCopy(source){   var result = {};   for(var i in source){   if(typeof source[i] === "object"){   result[i] = deepCopy(source[i]);   }else{   result[i] = source[i];   }   }   return result;  } |

**4.获取字符串里出现子串的位置**

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function appear(str,str\_target){   var n = 0;   var result = [];   while(str.indexOf(str\_target,n)!=-1 && n < str.length){   result.push(str.indexOf(str\_target,n));   n = str.indexOf(str\_target,n) + str\_target.length;   }   return result;  }  var arr = appear('abascbascbabasbascbascascbab','ab');  console.log(arr); |

**5.不确定数量的数组遍历组合算法**

好吧，解释下这题。这题在现实中确实会用到。尤其是做商城网站时，sku的算法真的经常会遇到。

这题的意思就是说。相当于说[1,2,3],[4,5]。。。。的不确定个数的数组进行遍历组合，组成[[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]这样。然后数组越多，组出来就肯定越多。

那怎么做的，我上网查了一些相关算法都没找到好的，然后我就自己写。可能还是会有点毛病，大家将就看。

有写的更好的欢迎评论教我一下。

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | function group(arr,re){   if(arr.length <=0){   return re;   }   if(!re){   var arr = arr.slice();   var re = arr.shift();   return group(arr,re);   }else{   var now = arr.shift();   var newre = [];   for(var j=0;j<now.length;j++){   for(var k=0;k<re.length;k++){   var temp = [];   if(re[k] instanceof Array){    temp = re[k];   }else{    temp.push(re[k]);   }   newre.push(temp.concat(now[j]));   }   }   return group(arr,newre);   }  }  var arr = [['a','b','c'],['e','d','f'],['h','i'],['j','k','l','m']];  // var arr = [['a','b','c'],['e','d','f'],['h','i']];  // console.log(arr);  var result = group(arr);  console.log(result); |

**6.lazyMan(这道题考察了很多内容)**

这道题自行百度吧。。只要百度lazyMan 面试题，应该是可以搜出来的

**7.编写一个函数fn(Number n),将数字转为大写输出，如输入123，输出一百二十三。**

好吧，这道题我是忘了我看的本站的哪位的文章了，觉得确实有点意思。

下面是他的代码。

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | function fn(n){   if(!/^([1-9]\d\*)/.test(n)){    return '非法数据';   }   var unit = '千百十亿千百十万千百十个';   if(n.length > unit.length){    return '数据过长';   }   var newStr = '';   var nlength = n.length;   unit = unit.substr(unit.length - nlength);   for(var i = 0; i < nlength; i++){    newStr += '零一二三四五六七八九'.charAt(n[i]) + unit.charAt(i);   }   newStr = newStr.substr(0,newStr.length-1);   newStr = newStr.replace(/零(千|百|十)/g,'零').replace(/(零)+/g,'零').replace(/零(亿|万)/g,'$1');   return newStr;  }  console.log(fn('205402002103')); |

**8.如何将浮点数左边的数每三位添加逗号**

如1200000.11转成12，000，000.11

[?](http://www.jb51.net/article/109269.htm)

|  |  |
| --- | --- |
| 1  2  3 | result = num && num.toString().replace(/(\d)(?=(\d{3})+\.)/g,function($1,$2){   return $2 + ',';  }) |

上面的解法是用正则，当然你也可以用别的方法。

二叉树

|  |
| --- |
| 'use strict'; |
|  |  |
|  | class Node { |
|  | constructor(data, left, right) { |
|  | this.left = left; |
|  | this.right = right; |
|  | this.data = data; |
|  | } |
|  | getData() { |
|  | return this.data; |
|  | } |
|  | } |
|  |  |
|  | const min = function min(node) { |
|  | let currentNode = node; |
|  | while (currentNode && currentNode.left) { |
|  | currentNode = currentNode.left; |
|  | } |
|  | return currentNode && currentNode.getData(); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* 查找以node为父节点的子树的最大值 |
|  | \* @param {[type]} node [父节点] |
|  | \* @return {[type]} [子树的最大值] |
|  | \*/ |
|  | const max = function max(node) { |
|  | let currentNode = node; |
|  | while (currentNode && currentNode.right) { |
|  | currentNode = currentNode.right; |
|  | } |
|  | return currentNode && currentNode.getData(); |
|  | } |
|  |  |
|  |  |
|  | /\*\* |
|  | \* 从二叉搜索树删除节点 |
|  | \* 1.判断当前节点是否包含待删除的数 |
|  | \* 2.如果包含，则删除节点，如果不包含，则比较当前节点上的数和待删除数据，如果待删除数据小于当前节点上的数，则移至当前节点的左子节点继节点续比较。如果待删除数据大于当前节点上的数，则移至当前节点的右子节点继续比较。 |
|  | \* 删除判断： |
|  | \* 1.如果当前节点是叶子节点（没有左子节点和右子节点），那么只需要将父节点指向它的链接指向null。 |
|  | \* 2.如果待删除的节点只包含一个子节点，那么原本指向它的节点指向它的子节点 |
|  | \* 3.如果待删除的节点包含两个子节点，那么，有两种做法：1>查找删除节点左子树上的最大值 2>查找其右子树上的最小值 |
|  | \*/ |
|  | const removeNode = function removeNode(node, data) { |
|  | if (node == null) { |
|  | return null; |
|  | } |
|  | if (data == node.getData()) { |
|  | //没有子节点的节点 |
|  | if (node.left == null && node.right == null) { |
|  | return null; |
|  | } |
|  | //没有左子节点的节点 |
|  | if (node.left == null) { |
|  | return node.right; |
|  | } |
|  | //没有右子节点的节点 |
|  | if (node.right == null) { |
|  | return node.left; |
|  | } |
|  | //有两个子节点的节点 |
|  | let smallest = min(node.right); |
|  | node.data = smallest; |
|  | node.right = removeNode(node.right, smallest); |
|  | return node; |
|  | } else if (data < node.getData()) { |
|  | node.left = removeNode(node.left, data); |
|  | return node; |
|  | } else { |
|  | node.right = removeNode(node.right, data); |
|  | return node; |
|  | } |
|  | } |
|  |  |
|  | /\*\* |
|  | \* 搜索二叉树 |
|  | \* 定义：若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。 |
|  | \*/ |
|  | class BSTree { |
|  | constructor() { |
|  | this.root = null; |
|  | } |
|  | /\*\* |
|  | \* 搜索二叉树插入规则： |
|  | \* 1.先检查是否有根节点，如果没有，则根节点就是要插入的新节点。 |
|  | \* 2.如果有根节点 |
|  | \* 1》设根节点为当前节点 |
|  | \* 2》如果待插入节点保存的数据小于当前节点,则设新的当前节点为原节点的左节点;反之,执行第 4 步。 |
|  | \* 3》如果当前节点的左节点为null，就将新节点插入这个位置。反之，继续执行下一次循环。 |
|  | \* 4》设新的当前节点为原节点的右节点 |
|  | \* 5》如果当前节点的右节点为null，就将新节点插入这个位置。反之，继续执行下一次循环。 |
|  | \*/ |
|  | insert(element) { |
|  | let newNode = new Node(element); |
|  | //如果根节点为空，则直接将新节点插入到根节点 |
|  | if (this.root == null) { |
|  | this.root = newNode; |
|  | } else { |
|  | let currentNode = this.root; |
|  | while (true) { |
|  | if (newNode.getData() < currentNode.getData()) { |
|  | if (currentNode.left == null) { |
|  | currentNode.left = newNode; |
|  | break; |
|  | } |
|  | currentNode = currentNode.left; |
|  | } else { |
|  | if (currentNode.right == null) { |
|  | currentNode.right = newNode; |
|  | break; |
|  | } |
|  | currentNode = currentNode.right; |
|  | } |
|  | } |
|  |  |
|  | } |
|  | } |
|  |  |
|  | /\*\* |
|  | \* 根据指定元素查找值，如果找到，返回，如果未找到，返回null |
|  | \*/ |
|  | find(element) { |
|  | let currentNode = this.root; |
|  | let value = null; |
|  | while (currentNode) { |
|  | if (element > currentNode.getData()) { |
|  | currentNode = currentNode.right; |
|  | } else if (element < currentNode.getData()) { |
|  | currentNode = currentNode.left; |
|  | } else if (element == currentNode.getData()) { |
|  | value = currentNode.getData(); |
|  | break; |
|  | } |
|  | } |
|  | return value; |
|  | } |
|  |  |
|  | /\*\* |
|  | \* 查找最小值 |
|  | \*/ |
|  | min() { |
|  | return min(this.root); |
|  | } |
|  |  |
|  | /\*\* |
|  | \* 查找最大值 |
|  | \*/ |
|  | max() { |
|  | return max(this.root); |
|  | } |
|  |  |
|  |  |
|  | remove(data) { |
|  | let root = removeNode(this.root, data); |
|  | console.log(JSON.stringify(root)); |
|  | // this.root = BSTree.removeNode(this.root,data); |
|  | } |
|  |  |
|  | } |
|  |  |
|  | /\*\* |
|  | \* [inOrder 中序遍历二叉树] |
|  | \* @param {[Node]} node [遍历树的根节点] |
|  | \* @return {[type]} [description] |
|  | \*/ |
|  | const inOrder = function inOrder(node, result) { |
|  | if (!(node == null)) { |
|  | inOrder(node.left, result); |
|  | result.push(node.getData()); |
|  | inOrder(node.right, result); |
|  | } |
|  | }; |
|  | const preOrder = function preOrder(node, result) { |
|  | if (!(node == null)) { |
|  | result.push(node.getData()); |
|  | preOrder(node.left, result); |
|  | preOrder(node.right, result); |
|  | } |
|  | } |
|  |  |
|  | let a = [87,234,5,21,65,23,98,98,90]; |
|  | let bst = new BSTree(); |
|  | for(let item of a){ |
|  | bst.insert(item); |
|  | } |
|  |  |
|  | let result = []; |
|  | inOrder(bst.root,result); |
|  | console.log(result); |
|  | console.log(bst.min()); |
|  | console.log(bst.max()); |
|  | let value = 98; |
|  | console.log(bst.find(value)); |
|  |  |
|  | // let array = [56, 22, 10, 30, 81, 77, 92]; |
|  | // let bst = new BSTree(); |
|  | // for (let a of array) { |
|  | // bst.insert(a); |
|  | // } |
|  | // |
|  | // console.log(JSON.stringify(bst)); |
|  | // bst.remove(22); |
|  | // console.log(JSON.stringify(bst)); |