1、使用 typeof bar === "object" 判断 bar 是不是一个对象有神马潜在的弊端？如何避免这种弊端？

使用 typeof 的弊端是显而易见的(这种弊端同使用 instanceof)：

```
let obj = {};
let arr = [];

console.log(typeof obj === 'object');  //true
console.log(typeof arr === 'object');  //true
console.log(typeof null === 'object');  //true
```

从上面的输出结果可知，typeof bar === "object" 并不能准确判断 bar 就是一个 Object。可以通过 Object.prototype.toString.call(bar) === "[object Object]" 来避免这种弊端：

```
let obj = {};
let arr = [];

console.log(Object.prototype.toString.call(obj));  //[object Object]
console.log(Object.prototype.toString.call(arr));  //[object Array]
console.log(Object.prototype.toString.call(null));  //[object Null]
```

另外，为了珍爱生命，请远离 ==：

而 [] === false 是返回 false 的。

2、下面的代码会在 console 输出神马？为什么？

```
(function(){
  var a = b = 3;
})();

console.log("a defined? " + (typeof a !== 'undefined'));
console.log("b defined? " + (typeof b !== 'undefined'));
```

这跟变量作用域有关，输出换成下面的：

```
console.log(b); //3
console,log(typeof a); //undefined
```

拆解一下自执行函数中的变量赋值：

```
b = 3;
var a = b;
```

所以 b 成了全局变量，而 a 是自执行函数的一个局部变量。

3、下面的代码会在 console 输出神马？为什么？

```
var myObject = {
  foo: "bar",
  func: function() {
    var self = this;
    console.log("outer func:  this.foo = " + this.foo);
    console.log("outer func:  self.foo = " + self.foo);
    (function() {
      console.log("inner func:  this.foo = " + this.foo);
      console.log("inner func:  self.foo = " + self.foo);
    }());
  }
};
myObject.func();
```

第一个和第二个的输出不难判断，在 ES6 之前，JavaScript 只有函数作用域，所以 func 中的 IIFE 有自己的独立作用域，并且它能访问到外部作用域中的 self，所以第三个输出会报错，因为 this 在可访问到的作用域内是 undefined，第四个输出是 bar。如果你知道闭包，也很容易解决的：

```
(function(test) {
      console.log("inner func:  this.foo = " + test.foo);  //'bar'
      console.log("inner func:  self.foo = " + self.foo);
}(self));
```

如果对闭包不熟悉，可以戳此：从作用域链谈闭包

4、将 JavaScript 代码包含在一个函数块中有神马意思呢？为什么要这么做？

换句话说，为什么要用立即执行函数表达式（Immediately-Invoked Function Expression）。

IIFE 有两个比较经典的使用场景，一是类似于在循环中定时输出数据项，二是类似于 JQuery/Node 的插件和模块开发。

```
for(var i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
```

```
}
```

上面的输出并不是你以为的 0，1，2，3，4，而输出的全部是 5，这时 IIFE 就能有用了：

```
for(var i = 0; i < 5; i++) {
   (function(i) {
     setTimeout(function() {
       console.log(i);
     }, 1000);
   })(i)
}
```

而在 JQuery/Node 的插件和模块开发中，为避免变量污染，也是一个大大的 IIFE：

```
(function($) {
     //代码
 } )(jQuery);
```

5、在严格模式('use strict')下进行 JavaScript 开发有神马好处？

- 消除 Javascript 语法的一些不合理、不严谨之处，减少一些怪异行为;
- 消除代码运行的一些不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的 Javascript 做好铺垫。

6、下面两个函数的返回值是一样的吗？为什么？

```
function foo1()
{
 return {
    bar: "hello"
 };
}

function foo2()
{
 return
 {
    bar: "hello"
```

```
 };
}
```

在编程语言中，基本都是使用分号（;）将语句分隔开，这可以增加代码的可读性和整洁性。而在 JS 中，如若语句各占独立一行，通常可以省略语句间的分号（;），JS 解析器会根据能否正常编译来决定是否自动填充分号：

```
var test = 1 +
2
console.log(test);  //3
```

在上述情况下，为了正确解析代码，就不会自动填充分号了，但是对于 return 、break、continue 等语句，如果后面紧跟换行，解析器一定会自动在后面填充分号(;)，所以上面的第二个函数就变成了这样：

```
function foo2()
{
 return;
 {
    bar: "hello"
 };
}
```

所以第二个函数是返回 undefined。

7、神马是 NaN，它的类型是神马？怎么测试一个值是否等于 NaN?

NaN 是 Not a Number 的缩写，JavaScript 的一种特殊数值，其类型是 Number，可以通过 isNaN(param) 来判断一个值是否是 NaN：

```
console.log(isNaN(NaN)); //true
console.log(isNaN(23)); //false
console.log(isNaN('ds')); //true
console.log(isNaN('32131sdasd')); //true
console.log(NaN === NaN); //false
console.log(NaN === undefined); //false
console.log(undefined === undefined); //false
console.log(typeof NaN); //number
console.log(Object.prototype.toString.call(NaN)); //[object Number]
```

ES6 中，isNaN() 成为了 Number 的静态方法：Number.isNaN().

8、解释一下下面代码的输出

```
console.log(0.1 + 0.2);  //0.30000000000000004
console.log(0.1 + 0.2 == 0.3);  //false
```

JavaScript 中的 number 类型就是浮点型，JavaScript 中的浮点数采用 IEEE-754 格式的规定，这是一种二进制表示法，可以精确地表示分数，比如 1/2，1/8，1/1024，每个浮点数占 64 位。但是，二进制浮点数表示法并不能精确的表示类似 0.1 这样 的简单的数字，会有舍入误差。

由于采用二进制，JavaScript 也不能有限表示 1/10、1/2 等这样的分数。在二进制中，1/10(0.1)被表示为 0.0001100110011001100110011…… 注意 0011 是无限重复的，这是舍入误差造成的，所以对于 0.1 + 0.2 这样的运算，操作数会先被转成二进制，然后再计算：

```
0.1 => 0.0001 1001 1001 1001…（无限循环）
0.2 => 0.0011 0011 0011 0011…（无限循环）
```

双精度浮点数的小数部分最多支持 52 位，所以两者相加之后得到这么一串 0.0100110011001100110011001100110011001100110011…因浮点数小数位的限制而截断的二进制数字，这时候，再把它转换为十进制，就成了 0.30000000000000004。

对于保证浮点数计算的正确性，有两种常见方式。

一是先升幂再降幂：

```
function add(num1, num2){
 let r1, r2, m;
 r1 = (''+num1).split('.')[1].length;
 r2 = (''+num2).split('.')[1].length;

 m = Math.pow(10,Math.max(r1,r2));
 return (num1 * m + num2 * m) / m;
}
console.log(add(0.1,0.2));  //0.3
console.log(add(0.15,0.2256)); //0.3756
```

二是是使用内置的 toPrecision() 和 toFixed() 方法，**注意，方法的返回值字符串。**

```
function add(x, y) {
   return x.toPrecision() + y.toPrecision()
}
console.log(add(0.1,0.2));  //"0.10.2"
```

9、实现函数 isInteger(x) 来判断 x 是否是整数

可以将 x 转换成 10 进制，判断和本身是不是相等即可：

```
function isInteger(x) {
  return parseInt(x, 10) === x;
}
```

ES6 对数值进行了扩展，提供了静态方法 isInteger() 来判断参数是否是整数：

```
Number.isInteger(25) // true
Number.isInteger(25.0) // true
Number.isInteger(25.1) // false
Number.isInteger("15") // false
Number.isInteger(true) // false
```

JavaScript 能够准确表示的整数范围在 -2^53 到 2^53 之间（不含两个端点），超过这个范围，无法精确表示这个值。ES6 引入了 Number.MAX_SAFE_INTEGER 和 Number.MIN_SAFE_INTEGER 这两个常量，用来表示这个范围的上下限，并提供了 Number.isSafeInteger() 来判断整数是否是安全型整数。

10、在下面的代码中，数字 1-4 会以什么顺序输出？为什么会这样输出？

```
(function() {
  console.log(1);
  setTimeout(function(){console.log(2)}, 1000);
  setTimeout(function(){console.log(3)}, 0);
  console.log(4);
})();
```

这个就不多解释了，主要是 JavaScript 的定时机制和时间循环，不要忘了，JavaScript 是单线程的。详解可以参考 从 setTimeout 谈 JavaScript 运行机制。

11、写一个少于 80 字符的函数，判断一个字符串是不是回文字符串

```
function isPalindrome(str) {
  str = str.replace(/\W/g, '').toLowerCase();
  return (str == str.split('').reverse().join(''));
}
```

这个题我在 codewars 上碰到过，并收录了一些不错的解决方式，可以戳这里：Palindrome For Your Dome

12、写一个按照下面方式调用都能正常工作的 sum 方法

```
console.log(sum(2,3));  // Outputs 5
console.log(sum(2)(3));  // Outputs 5
```

针对这个题，可以判断参数个数来实现：

```
function sum() {
 var fir = arguments[0];
 if(arguments.length === 2) {
  return arguments[0] + arguments[1]
 } else {
  return function(sec) {
    return fir + sec;
  }
 }

}
```

13、根据下面的代码片段回答后面的问题

```
for (var i = 0; i < 5; i++) {
 var btn = document.createElement('button');
 btn.appendChild(document.createTextNode('Button ' + i));
 btn.addEventListener('click', function(){ console.log(i); });
 document.body.appendChild(btn);
}
```

1、点击 Button 4，会在控制台输出什么？
2、给出一种符合预期的实现方式
1、点击 5 个按钮中的任意一个，都是输出 5
2、参考 IIFE。

14、下面的代码会输出什么？为什么？

```
var arr1 = "john".split(''); // j o h n
var arr2 = arr1.reverse(); //n h o j
var arr3 = "jones".split(''); //j o n e s
arr2.push(arr3);
console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));
console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1));
```

会输出什么呢？你运行下就知道了，可能会在你的意料之外。

MDN 上对于 reverse() 的描述是酱紫的：

Description

The reverse method transposes the elements of the calling array object in place, mutating the array, and returning a reference to the array.

reverse() 会改变数组本身，并返回原数组的引用。

slice 的用法请参考：slice

15、下面的代码会输出什么？为什么？

```
console.log(1 +  "2" + "2");
console.log(1 +  +"2" + "2");
console.log(1 +  -"1" + "2");
console.log(+"1" +  "1" + "2");
console.log( "A" - "B" + "2");
console.log( "A" - "B" + 2);
```

输出什么，自己去运行吧，需要注意三个点：

- 多个数字和数字字符串混合运算时，跟操作数的位置有关

```
console.log(2 + 1 + '3'); / /‘33’
console.log('3' + 2 + 1); //'321'
```

- 数字字符串之前存在数字中的正负号(+/-)时，会被转换成数字

```
console.log(typeof '3');  // string
console.log(typeof +'3');  //number
```

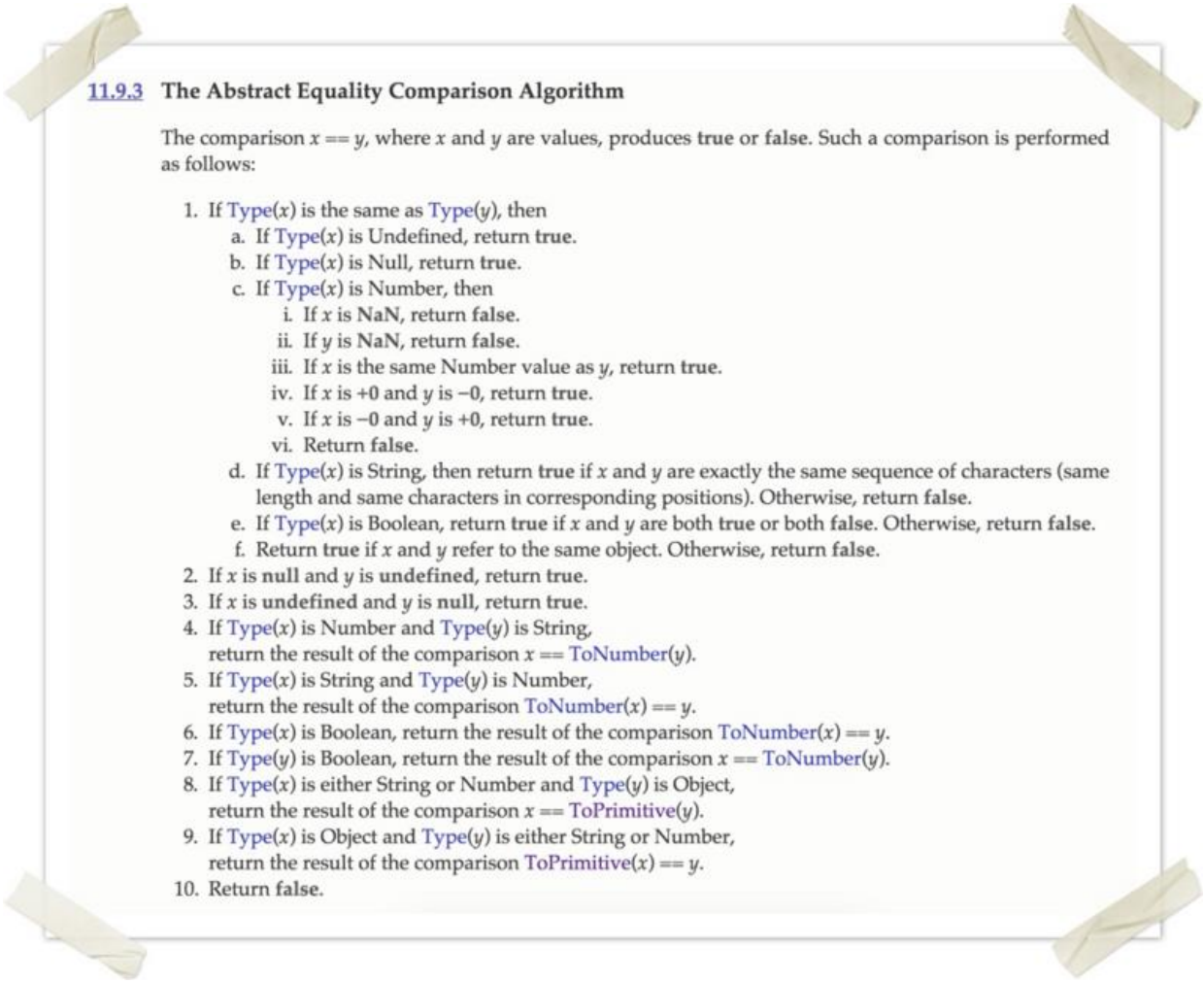
同样，可以在数字前添加 ''，将数字转为字符串

```
console.log(typeof 3);  // number
console.log(typeof (''+3));  //string
```

- 对于运算结果不能转换成数字的，将返回 NaN

```
console.log('a' * 'sd');  //NaN
console.log('A' - 'B');  // NaN
```

这张图是运算转换的规则



11.9.3 The Abstract Equality Comparison Algorithm

The comparison $x == y$, where x and y are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(x) is the same as Type(y), then
   a. If Type(x) is Undefined, return **true**.
   b. If Type(x) is Null, return **true**.
   c. If Type(x) is Number, then
      i. If x is **NaN**, return **false**.
      ii. If y is **NaN**, return **false**.
      iii. If x is the same Number value as y, return **true**.
      iv. If x is +0 and y is −0, return **true**.
      v. If x is −0 and y is +0, return **true**.
      vi. Return **false**.
   d. If Type(x) is String, then return **true** if x and y are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return **false**.
   e. If Type(x) is Boolean, return **true** if x and y are both **true** or both **false**. Otherwise, return **false**.
   f. Return **true** if x and y refer to the same object. Otherwise, return **false**.
2. If x is **null** and y is **undefined**, return **true**.
3. If x is **undefined** and y is **null**, return **true**.
4. If Type(x) is Number and Type(y) is String,
   return the result of the comparison $x ==$ ToNumber(y).
5. If Type(x) is String and Type(y) is Number,
   return the result of the comparison ToNumber(x) $== y$.
6. If Type(x) is Boolean, return the result of the comparison ToNumber(x) $== y$.
7. If Type(y) is Boolean, return the result of the comparison $x ==$ ToNumber(y).
8. If Type(x) is either String or Number and Type(y) is Object,
   return the result of the comparison $x ==$ ToPrimitive(y).
9. If Type(x) is Object and Type(y) is either String or Number,
   return the result of the comparison ToPrimitive(x) $== y$.
10. Return **false**.

16、如果 list 很大，下面的这段递归代码会造成堆栈溢出。如果在不改变递归模式的前提下修善这段代码？

```
var list = readHugeList();

var nextListItem = function() {
   var item = list.pop();

   if (item) {
      // process the list item...
      nextListItem();
   }
};
```

原文上的解决方式是加个定时器：

```
var list = readHugeList();

var nextListItem = function() {
   var item = list.pop();

   if (item) {
      // process the list item...
      setTimeout( nextListItem, 0);
   }
};
```

解决方式的原理请参考第 10 题。

17、什么是闭包？举例说明

可以参考此篇：从作用域链谈闭包

18、下面的代码会输出什么？为啥？

```
for (var i = 0; i < 5; i++) {
  setTimeout(function() { console.log(i); }, i * 1000 );
}
```

请往前面翻，参考第 4 题，解决方式已经在上面了

19、解释下列代码的输出

```
console.log("0 || 1 = "+(0 || 1));
console.log("1 || 2 = "+(1 || 2));
console.log("0 && 1 = "+(0 && 1));
console.log("1 && 2 = "+(1 && 2));
```

逻辑与和逻辑或运算符会返回一个值，并且二者都是短路运算符：

- 逻辑与返回第一个是 false 的操作数 或者 最后一个是 true 的操作数

```
console.log(1 && 2 && 0);  //0
console.log(1 && 0 && 1);  //0
console.log(1 && 2 && 3);  //3
```

如果某个操作数为 false，则该操作数之后的操作数都不会被计算

- 逻辑或返回第一个是 true 的操作数 或者 最后一个是 false 的操作数

```
console.log(1 || 2 || 0); //1
console.log(0 || 2 || 1); //2
console.log(0 || 0 || false); //false
```

如果某个操作数为 true，则该操作数之后的操作数都不会被计算
如果逻辑与和逻辑或作混合运算，则逻辑与的优先级高：

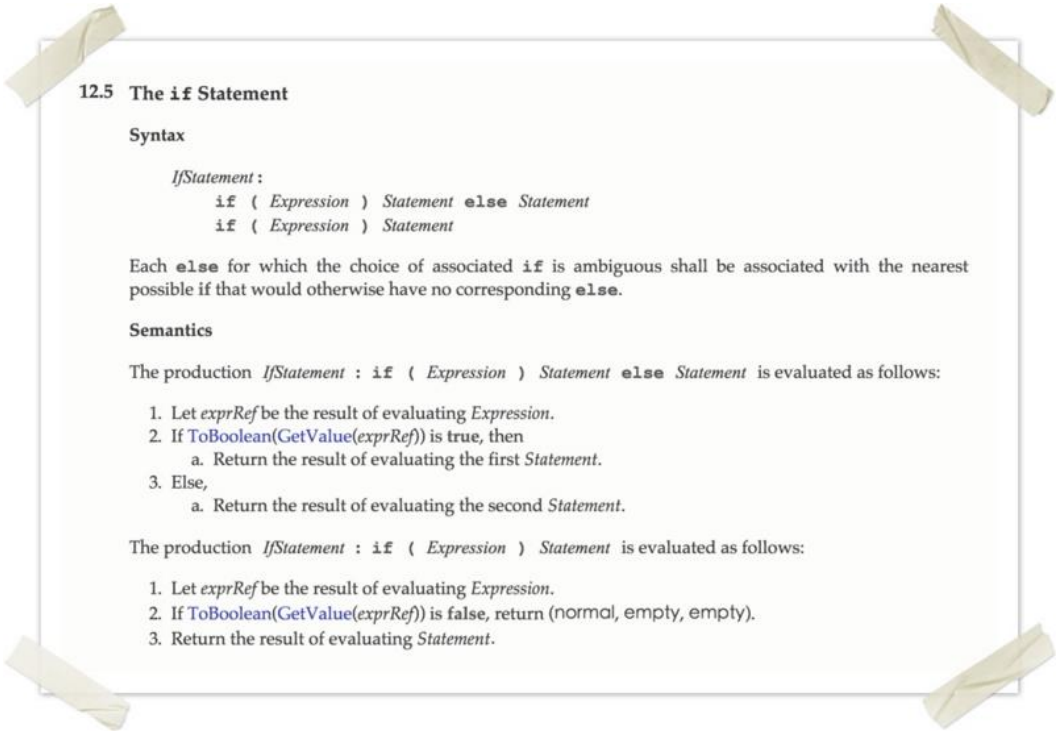
```
console.log(1 && 2 || 0); //2
console.log(0 || 2 && 1); //1
console.log(0 && 2 || 1); //1
```

在 JavaScript，常见的 false 值：

```
0, '0', +0, -0, false, '',null,undefined,null,NaN
```

要注意**空数组([])和空对象({}):**

```
console.log([] == false) //true
console.log({} == false) //false
console.log(Boolean([])) //true
console.log(Boolean({})) //true
```



12.5 The `if` Statement

Syntax

IfStatement :
`if` (*Expression*) *Statement* `else` *Statement*
`if` (*Expression*) *Statement*

Each `else` for which the choice of associated `if` is ambiguous shall be associated with the nearest possible if that would otherwise have no corresponding `else`.

Semantics

The production *IfStatement* : `if` (*Expression*) *Statement* `else` *Statement* is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. If ToBoolean(GetValue(*exprRef*)) is **true**, then
   a. Return the result of evaluating the first *Statement*.
3. Else,
   a. Return the result of evaluating the second *Statement*.

The production *IfStatement* : `if` (*Expression*) *Statement* is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, empty, empty).
3. Return the result of evaluating *Statement*.

所以
在 if 中，
[] 和 {} 都表现为 true：

20、解释下面代码的输出

```
console.log(false == '0')
console.log(false === '0')
```

请参考前面第 14 题运算符转换规则的图。

21、解释下面代码的输出

```
var a={},
   b={key:'b'},
   c={key:'c'};

a[b]=123;
a[c]=456;

console.log(a[b]);
```

输出是 456，参考原文的解释：

The reason for this is as follows: When setting an object property, JavaScript will implicitly stringify the parameter value. In this case, since b and c are both objects, they will both be converted to "[object Object]". As a result, a[b] anda[c] are both equivalent to a["[object Object]"] and can be used interchangeably. Therefore, setting or referencing a[c] is precisely the same as setting or referencing a[b].

22、解释下面代码的输出

```
console.log((function f(n){return ((n > 1) ? n * f(n-1) : n)})(10));
```

结果是 10 的阶乘。这是一个递归调用，为了简化，我初始化 n=5，则调用链和返回链如下：

23、解释下面代码的输出

```
(function(x) {
   return (function(y) {
      console.log(x);
   })(2)
})(1);
```

输出 1，闭包能够访问外部作用域的变量或参数。

24、解释下面代码的输出，并修复存在的问题

```
var hero = {
    _name: 'John Doe',
    getSecretIdentity: function (){
        return this._name;
    }
};

var stoleSecretIdentity = hero.getSecretIdentity;

console.log(stoleSecretIdentity());
console.log(hero.getSecretIdentity());
```

将 getSecretIdentity 赋给 stoleSecretIdentity，等价于定义了 stoleSecretIdentity 函数：

```
var stoleSecretIdentity =  function (){
        return this._name;
}
```

stoleSecretIdentity 的上下文是全局环境，所以第一个输出 undefined。若要输出 John Doe，则要通过 call 、apply 和 bind 等方式改变 stoleSecretIdentity 的 this 指向(hero)。第二个是调用对象的方法，输出 John Doe。

25、给你一个 DOM 元素，创建一个能访问该元素所有子元素的函数，并且要将每个子元素传递给指定的回调函数。

函数接受两个参数：

- DOM
- 指定的回调函数

原文利用 深度优先搜索(Depth-First-Search) 给了一个实现：

```
function Traverse(p_element,p_callback) {
   p_callback(p_element);
   var list = p_element.children;
   for (var i = 0; i < list.length; i++) {
       Traverse(list[i],p_callback);  // recursive call
   }
}
```

文章参考：

25 Essential JavaScript Interview Questions
JavaScript 中的立即执行函数表达式
Javascript 严格模式详解

JavaScript 面试中常见算法问题详解
JavaScript 面试中常见算法问题详解 翻译自 Interview Algorithm Questions in Javascript() {...} 从属于笔者的 Web 前端入门与工程实践。下文提到的很多问题从算法角度并不一定要么困难，不过用 JavaScript 内置的 API 来完成还是需要一番考量的。

JavaScript Specification

阐述下 JavaScript 中的变量提升

所谓提升，顾名思义即是 JavaScript 会将所有的声明提升到当前作用域的顶部。这也就意味着我们可以在某个变量声明前就使用该变量，不过虽然 JavaScript 会将声明提升到顶部，但是并不会执行真的初始化过程。

阐述下 use strict; 的作用

use strict; 顾名思义也就是 JavaScript 会在所谓严格模式下执行，其一个主要的优势在于能够强制开发者避免使用未声明的变量。对于老版本的浏览器或者执行引擎则会自动忽略该指令。

```
// Example of strict mode
"use strict";

catchThemAll();
function catchThemAll() {
  x = 3.14; // Error will be thrown
  return x * x;
}
```

解释下什么是 Event Bubbling 以及如何避免

Event Bubbling 即指某个事件不仅会触发当前元素，还会以嵌套顺序传递到父元素中。直观而言就是对于某个子元素的点击事件同样会被父元素的点击事件处理器捕获。避免 Event Bubbling 的方式可以使用 event.stopPropagation() 或者 IE 9 以下使用 event.cancelBubble。

== 与 === 的区别是什么

=== 也就是所谓的严格比较，关键的区别在于=== 会同时比较类型与值，而不是仅比较值。

```
// Example of comparators
0 == false; // true
0 === false; // false

2 == '2'; // true
2 === '2'; // false
```

解释下 null 与 undefined 的区别

JavaScript 中，null 是一个可以被分配的值，设置为 null 的变量意味着其无值。而 undefined 则代表着某个变量虽然声明了但是尚未进行过任何赋值。

解释下 Prototypal Inheritance 与 Classical Inheritance 的区别

在类继承中，类是不可变的，不同的语言中对于多继承的支持也不一样，有些语言中还支持接口、final、abstract 的概念。而原型继承则更为灵活，原型本身是可以可变的，并且对象可能继承自多个原型。

数组

找出整型数组中乘积最大的三个数

给定一个包含整数的无序数组，要求找出乘积最大的三个数。

```
var unsorted_array = [-10, 7, 29, 30, 5, -10, -70];

computeProduct(unsorted_array); // 21000
```

```
function sortIntegers(a, b) {
  return a - b;
}

// greatest product is either (min1 * min2 * max1 || max1 * max2 * max3)
function computeProduct(unsorted) {
  var sorted_array = unsorted.sort(sortIntegers),
    product1 = 1,
    product2 = 1,
    array_n_element = sorted_array.length - 1;

  // Get the product of three largest integers in sorted array
  for (var x = array_n_element; x > array_n_element - 3; x--) {
     product1 = product1 * sorted_array[x];
  }
  product2 = sorted_array[0] * sorted_array[1] * sorted_array[array_n_element];

  if (product1 > product2) return product1;

  return product2
};
```

寻找连续数组中的缺失数

给定某无序数组，其包含了 n 个连续数字中的 n - 1 个，已知上下边界，要求以 O(n)的复杂度找出缺失的数字。

```
// The output of the function should be 8
var array_of_integers = [2, 5, 1, 4, 9, 6, 3, 7];
var upper_bound = 9;
var lower_bound = 1;

findMissingNumber(array_of_integers, upper_bound, lower_bound); //8

function findMissingNumber(array_of_integers, upper_bound, lower_bound) {
```

```
 // Iterate through array to find the sum of the numbers
 var sum_of_integers = 0;
 for (var i = 0; i < array_of_integers.length; i++) {
   sum_of_integers += array_of_integers[i];
 }

 // 以高斯求和公式计算理论上的数组和
 // Formula: [(N * (N + 1)) / 2] - [(M * (M - 1)) / 2];
 // N is the upper bound and M is the lower bound

 upper_limit_sum = (upper_bound * (upper_bound + 1)) / 2;
 lower_limit_sum = (lower_bound * (lower_bound - 1)) / 2;

 theoretical_sum = upper_limit_sum - lower_limit_sum;

 //
 return (theoretical_sum - sum_of_integers)
}
```

数组去重

给定某无序数组，要求去除数组中的重复数字并且返回新的无重复数组。

```
// ES6 Implementation
var array = [1, 2, 3, 5, 1, 5, 9, 1, 2, 8];

Array.from(new Set(array)); // [1, 2, 3, 5, 9, 8]


// ES5 Implementation
var array = [1, 2, 3, 5, 1, 5, 9, 1, 2, 8];

uniqueArray(array); // [1, 2, 3, 5, 9, 8]

function uniqueArray(array) {
```

```
 var hashmap = {};
 var unique = [];
 for(var i = 0; i < array.length; i++) {
  // If key returns null (unique), it is evaluated as false.
  if(!hashmap.hasOwnProperty([array[i]])) {
    hashmap[array[i]] = 1;
    unique.push(array[i]);
  }
 }
 return unique;
}
```

数组中元素最大差值计算

给定某无序数组，求取任意两个元素之间的最大差值，注意，这里要求差值计算中较小的元素下标必须小于较大元素的下标。譬如[7, 8, 4, 9, 9, 15, 3, 1, 10]这个数组的计算值是 11(15 - 4) 而不是 14(15 - 1)，因为 15 的下标小于 1。

```
var array = [7, 8, 4, 9, 9, 15, 3, 1, 10];
// [7, 8, 4, 9, 9, 15, 3, 1, 10] would return `11` based on the difference between `4` and `15`
// Notice: It is not `14` from the difference between `15` and `1` because 15 comes before 1.

findLargestDifference(array);

function findLargestDifference(array) {

 // 如果数组仅有一个元素，则直接返回 -1

 if (array.length <= 1) return -1;

 // current_min 指向当前的最小值

 var current_min = array[0];
 var current_max_difference = 0;
```

```
 // 遍历整个数组以求取当前最大差值，如果发现某个最大差值，则将新的值覆盖current_max_difference
 // 同时也会追踪当前数组中的最小值，从而保证 `largest value in future` - `smallest value before it`

 for (var i = 1; i < array.length; i++) {
  if (array[i] > current_min && (array[i] - current_min > current_max_difference)) {
   current_max_difference = array[i] - current_min;
  } else if (array[i] <= current_min) {
   current_min = array[i];
  }
 }

 // If negative or 0, there is no largest difference
 if (current_max_difference <= 0) return -1;

 return current_max_difference;
}
```

数组中元素乘积

给定某无序数组，要求返回新数组 output ， 其中 output[i] 为原数组中除了下标为 i 的元素之外的元素乘积，要求以 O(n) 复杂度实现：

```
var firstArray = [2, 2, 4, 1];
var secondArray = [0, 0, 0, 2];
var thirdArray = [-2, -2, -3, 2];

productExceptSelf(firstArray); // [8, 8, 4, 16]
productExceptSelf(secondArray); // [0, 0, 0, 0]
productExceptSelf(thirdArray); // [12, 12, 8, -12]

function productExceptSelf(numArray) {
 var product = 1;
 var size = numArray.length;
 var output = [];
```

```
 // From first array: [1, 2, 4, 16]
 // The last number in this case is already in the right spot (allows for us)
 // to just multiply by 1 in the next step.
 // This step essentially gets the product to the left of the index at index + 1
 for (var x = 0; x < size; x++) {
    output.push(product);
    product = product * numArray[x];
 }

 // From the back, we multiply the current output element (which represents the product
 // on the left of the index, and multiplies it by the product on the right of the element)
 var product = 1;
 for (var i = size - 1; i > -1; i--) {
    output[i] = output[i] * product;
    product = product * numArray[i];
 }

 return output;
}
```

数组交集

给定两个数组，要求求出两个数组的交集，注意，交集中的元素应该是唯一的。

```
var firstArray = [2, 2, 4, 1];
var secondArray = [1, 2, 0, 2];

intersection(firstArray, secondArray); // [2, 1]

function intersection(firstArray, secondArray) {
 // The logic here is to create a hashmap with the elements of the firstArray as the keys.
 // After that, you can use the hashmap's O(1) look up time to check if the element exists in the
hash
 // If it does exist, add that element to the new array.
```

```
 var hashmap = {};
 var intersectionArray = [];

 firstArray.forEach(function(element) {
  hashmap[element] = 1;
 });

 // Since we only want to push unique elements in our case... we can implement a counter to
keep track of what we already added
 secondArray.forEach(function(element) {
  if (hashmap[element] === 1) {
   intersectionArray.push(element);
   hashmap[element]++;
  }
 });

 return intersectionArray;

 // Time complexity O(n), Space complexity O(n)
}
```

字符串

颠倒字符串

给定某个字符串，要求将其中单词倒转之后然后输出，譬如"Welcome to this Javascript Guide!" 应该输出为 "emocleW ot siht tpircsavaJ !ediuG"。

```
var string = "Welcome to this Javascript Guide!";

// Output becomes !ediuG tpircsavaJ siht ot emocleW
var reverseEntireSentence = reverseBySeparator(string, "");

// Output becomes emocleW ot siht tpircsavaJ !ediuG
var reverseEachWord = reverseBySeparator(reverseEntireSentence, " ");
```

```
function reverseBySeparator(string, separator) {
  return string.split(separator).reverse().join(separator);
}
```

乱序同字母字符串

给定两个字符串，判断是否颠倒字母而成的字符串，譬如 Mary 与 Army 就是同字母而顺序颠倒：

```
var firstWord = "Mary";
var secondWord = "Army";

isAnagram(firstWord, secondWord); // true

function isAnagram(first, second) {
  // For case insensitivity, change both words to lowercase.
  var a = first.toLowerCase();
  var b = second.toLowerCase();

  // Sort the strings, and join the resulting array to a string. Compare the results
  a = a.split("").sort().join("");
  b = b.split("").sort().join("");

  return a === b;
}
```

回文字符串

判断某个字符串是否为回文字符串，譬如 racecar 与 race car 都是回文字符串：

```
isPalindrome("racecar"); // true
isPalindrome("race Car"); // true

function isPalindrome(word) {
  // Replace all non-letter chars with "" and change to lowercase
  var lettersOnly = word.toLowerCase().replace(/\s/g, "");
```

```
 // Compare the string with the reversed version of the string
 return lettersOnly === lettersOnly.split("").reverse().join("");
}
```

栈与队列

使用两个栈实现入队与出队

```
var inputStack = []; // First stack
var outputStack = []; // Second stack

// For enqueue, just push the item into the first stack
function enqueue(stackInput, item) {
 return stackInput.push(item);
}

function dequeue(stackInput, stackOutput) {
 // Reverse the stack such that the first element of the output stack is the
 // last element of the input stack. After that, pop the top of the output to
 // get the first element that was ever pushed into the input stack
 if (stackOutput.length <= 0) {
  while(stackInput.length > 0) {
   var elementToOutput = stackInput.pop();
   stackOutput.push(elementToOutput);
  }
 }

 return stackOutput.pop();
}
```

判断大括号是否闭合

创建一个函数来判断给定的表达式中的大括号是否闭合：

```
var expression = "{{}}{}{}"
```

```
var expressionFalse = "{}{{}";

isBalanced(expression); // true
isBalanced(expressionFalse); // false
isBalanced(""); // true

function isBalanced(expression) {
  var checkString = expression;
  var stack = [];

  // If empty, parentheses are technically balanced
  if (checkString.length <= 0) return true;

  for (var i = 0; i < checkString.length; i++) {
    if(checkString[i] === '{') {
      stack.push(checkString[i]);
    } else if (checkString[i] === '}') {
      // Pop on an empty array is undefined
      if (stack.length > 0) {
        stack.pop();
      } else {
        return false;
      }
    }
  }

  // If the array is not empty, it is not balanced
  if (stack.pop()) return false;
  return true;
}
```

递归

二进制转换

通过某个递归函数将输入的数字转化为二进制字符串：

```
decimalToBinary(3); // 11
decimalToBinary(8); // 1000
decimalToBinary(1000); // 1111101000

function decimalToBinary(digit) {
  if(digit >= 1) {
    // If digit is not divisible by 2 then recursively return proceeding
    // binary of the digit minus 1, 1 is added for the leftover 1 digit
    if (digit % 2) {
      return decimalToBinary((digit - 1) / 2) + 1;
    } else {
      // Recursively return proceeding binary digits
      return decimalToBinary(digit / 2) + 0;
    }
  } else {
    // Exit condition
    return '';
  }
}
```

二分查找

```
function recursiveBinarySearch(array, value, leftPosition, rightPosition) {
  // Value DNE
  if (leftPosition > rightPosition) return -1;

  var middlePivot = Math.floor((leftPosition + rightPosition) / 2);
  if (array[middlePivot] === value) {
    return middlePivot;
  } else if (array[middlePivot] > value) {
    return recursiveBinarySearch(array, value, leftPosition, middlePivot - 1);
  } else {
    return recursiveBinarySearch(array, value, middlePivot + 1, rightPosition);
  }
}
```

数字

判断是否为 2 的指数值

```
isPowerOfTwo(4); // true
```

```
isPowerOfTwo(64); // true
isPowerOfTwo(1); // true
isPowerOfTwo(0); // false
isPowerOfTwo(-1); // false

// For the non-zero case:
function isPowerOfTwo(number) {
 // `&` uses the bitwise n.
 // In the case of number = 4; the expression would be identical to:
 // `return (4 & 3 === 0)`
 // In bitwise, 4 is 100, and 3 is 011. Using &, if two values at the same
 // spot is 1, then result is 1, else 0. In this case, it would return 000,
 // and thus, 4 satisfies are expression.
 // In turn, if the expression is `return (5 & 4 === 0)`, it would be false
 // since it returns 101 & 100 = 100 (NOT === 0)

 return number & (number - 1) === 0;
}

// For zero-case:
function isPowerOfTwoZeroCase(number) {
 return (number !== 0) && ((number & (number - 1)) === 0);
}
```

JavaScript 排序算法汇总

前言

关于排序算法的有关文章已经很多了，然而网络上用 Javascript 语言来作为示例并详实介绍的文章貌似还是不太多。这里主要是我来尝试自己针对网上各式的排序算法进行一份详实的个人总结，从而温故知新。

基本概念

算法优劣评价术语

稳定性

- 稳定：如果 a 原本在 b 前面，而 a = b，排序之后 a 仍然在 b 的前面；
- 不稳定：如果 a 原本在 b 的前面，而 a = b，排序之后 a 可能会出现在 b 的后面；

排序方式

- 内排序：所有排序操作都在内存中完成，占用常数内存，不占用额外内存。
- 外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行，占用额外内存。

复杂度

- 时间复杂度: 一个算法执行所耗费的时间。
- 空间复杂度: 运行完一个程序所需内存的大小。

排序算法图片总结

名词解释： n : 数据规模 k : 桶的个数

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡	O(n2)O(n2)	O(n)O(n)	O(n2)O(n2)	O(1)O(1)	内排序	稳定

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
排序						
选择排序	O(n2)O(n2)	O(n2)O(n2)	O(n2)O(n2)	O(1)O(1)	内排序	不稳定
插入排序	O(n2)O(n2)	O(n)O(n)	O(n2)O(n2)	O(1)O(1)	内排序	稳定
希尔排序	O(nlogn)O(nlog□n)	O(n(log22n))O(n(log22□n))	O(n(log22n))O(n(log22□n))	O(1)O(1)	内排序	不稳定
归并排序	O(nlogn)O(nlog□n)	O(nlogn)O(nlog□n)	O(nlogn)O(nlog□n)	O(n)O(n)	外排序	稳定
快速排序	O(nlogn)O(nlog□n)	O(nlogn)O(nlog□n)	O(n2)O(n2)	O(logn)O(log□n)	内排序	不稳定
堆排序	O(nlogn)O(nlog□n)	O(nlogn)O(nlog□n)	O(nlogn)O(nlog□n)	O(1)O(1)	内排序	不稳定

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
计数排序	O(n+k)O(n+k)	O(n+k)O(n+k)	O(n+k)O(n+k)	O(k)O(k)	外排序	稳定
桶排序	O(n+k)O(n+k)	O(n+k)O(n+k)	O(n2)O(n2)	O(n+k)O(n+k)	外排序	稳定
基数排序	O(n×k)O(n×k)	O(n×k)O(n×k)	O(n×k)O(n×k)	O(n+k)O(n+k)	外排序	稳定

从分类上来讲：

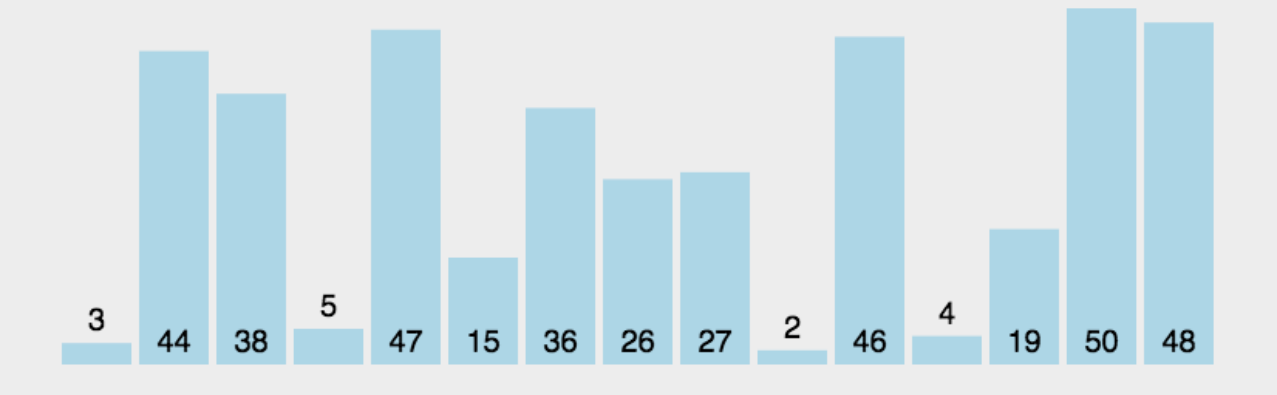
排序算法	**交换排序**	冒泡排序	快速排序	
	选择排序	选择排序	堆排序	
	插入排序	插入排序	希尔排序	
	归并排序	归并排序		
	分布排序	计数排序	桶排序	基数排序

冒泡排序（BUBBLE SORT）[1]

算法原理

冒泡排序（Bubble Sort）是一种简单的排序算法。它重复地走访要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换时，此时该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢"浮"到数列的顶端。

图解如下:



算法描述与实现

- 具体算法描述如下：

1. 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
3. 针对所有的元素重复以上的步骤，除了最后一个；
4. 重复步骤 1 31 3，直到排序完成。

- 代码实现如下：

```
const bubbleSort = (arr) => {
 let len = arr.length;
 for(let i = 0;i < len;i++){
  for(let j = 0;j < len - 1 - i;j++){
   if(arr[j] > arr[j+1]){
    let temp = arr[j+1];
    arr[j+1] = arr[j]
    arr[j] = temp;
   }
  }
 }
```

```
 return arr;
}

let arr = [3,44,38,5,47,15,36,26,27,2,46,4,19,50,48];
console.log(bubbleSort(arr)); //[2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]
```

改进冒泡排序：我们设置一个标志性变量pos，用于记录每趟排序中最后一次进行交换的位置。由于pos位置之后的元素均已交换到位，故在进行下一趟排序时只要扫描到pos位置即可。这样的优化方式可以在最好情况下把复杂度降到O(n)O(n)。

```
const bubbleSort2 = (arr) => {
  let i = arr.length - 1;
  while(i > 0){
    let pos = 0;
    for(let j = 0; j < i; j++){
      if (arr[j] > arr[j+1]){
        pos = j; //记录交换的位置
        let tmp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = tmp;
      }
    }
    i = pos; //为下一趟排序作准备
  }
  return arr;
}
```

另外传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值，我们可以考虑利用在每趟排序中进行正向和反向两遍冒泡的方法来一次得到两个最终值(最大者和最小者)，从而继续优化使排序趟数几乎减少一半。（这就是鸡尾酒排序）

```
const cooktailSort = (arr) => {
  let min = 0;
  let max = arr.length - 1;
  while(min < max){
    for(let j = min;j < max;j++){
      if(arr[j] > arr[j+1]){
```

```
      let tmp = arr[j];
      arr[j] = arr[j+1];
      arr[j+1] = tmp;
     }
    }
    -- max;
    for(let j = max;j > min;j--){
     if(arr[j] < arr[j-1]){
      let tmp = arr[j]
      arr[j] = arr[j-1];
      arr[j-1] = tmp;
     }
    }
    ++ min;
   }
   return arr;
}
```

算法分析

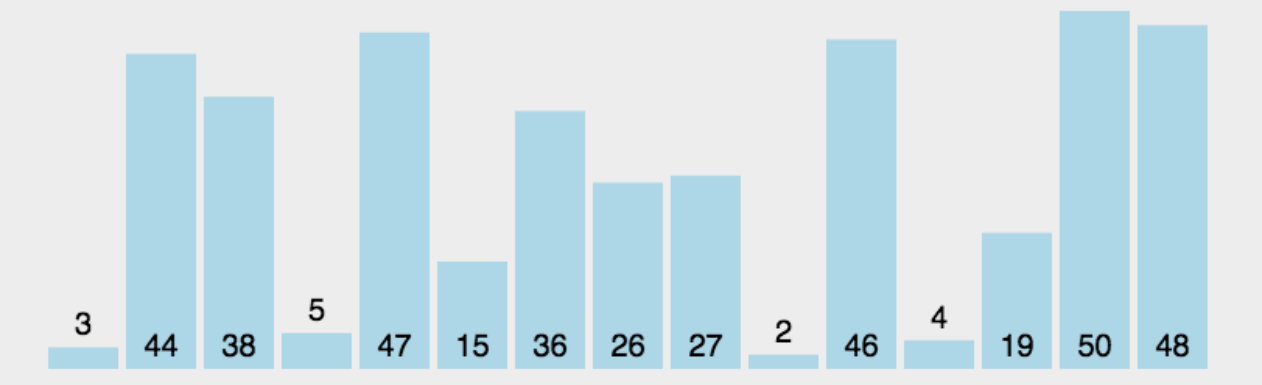
冒泡排序对有 nn 个元素的项目平均需要 O(n2)O(n2) 次比较次数，它可以原地排序，并且是能简单实现的几种排序算法之一，但是它对于少数元素之外的数列排序是很没有效率的。

- 最佳情况： T(n)=O(n)T(n)=O(n)
- 最差情况： T(n)=O(n2)T(n)=O(n2)
- 平均情况： T(n)=O(n2)T(n)=O(n2)

选择排序（SELECTION SORT）[2]

算法原理

选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。



图解如下:

算法描述与实现

- 具体算法描述如下：

nn 个记录的直接选择排序可经过 n−1n−1 趟直接选择排序得到有序结果。具体算法描述如下：

1. 初始状态：无序区为 R[1 ... n]，有序区为空；
2. 第 ii 趟排序(i=1,2,3...n−1)(i=1,2,3...n−1)开始时，当前有序区和无序区分别为 R[1...i−1]R[1...i−1] 和 R(i...n)R(i...n)。该趟排序从当前无序区中选出关键字最小的记录 R[k]R[k]，将它与无序区的第 1 个记录 RR 交换，
使 R[1...i]R[1...i] 和 R[i+1...n]R[i+1...n]分别变为记录个数增加 1 个的新有序区和记录个数减少 1 个的新无序区；
3. n−1n−1 趟结束后，数组有序化。

- 代码实现如下：

```
const SelectionSort = (arr) => {
  let len = arr.length;
  let minIndex, tmp;
```

```
  console.time('选择排序耗时');
  for (let i = 0;i < len - 1; i++){
    minIndex = i;
    for(let j = i + 1;j < len;j++){
      if(arr[minIndex] > arr[j]){
        minIndex = j;
      }
    }
    tmp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = tmp;
  }
  console.timeEnd('选择排序耗时');
  return arr;
}
```

算法分析

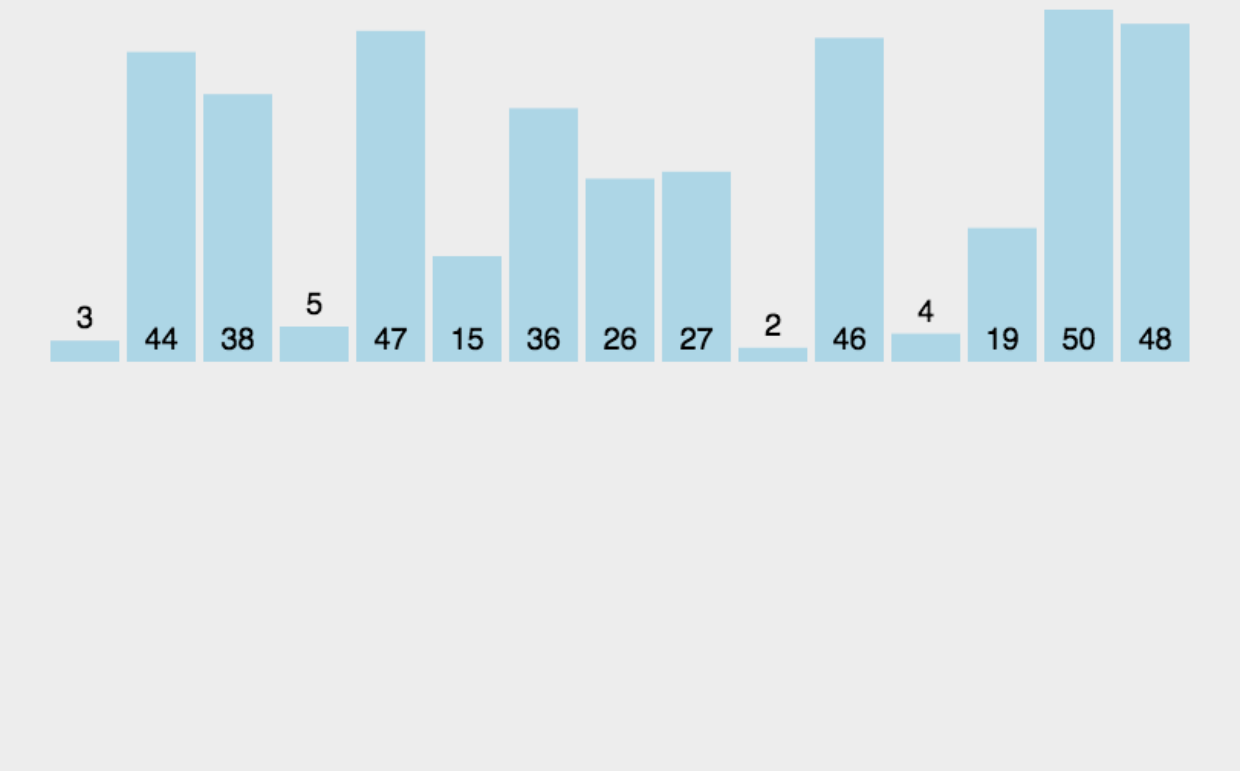
选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对 nn 个元素的表进行排序总共进行至多 n−1n−1 次交换。在所有的***完全依靠交换***去移动元素的排序方法中，选择排序属于非常好的一种。 但原地操作几乎是选择排序的唯一优点，当空间复杂度要求较高时，可以考虑选择排序；实际适用的场合非常罕见。

- 最佳情况： T(n)=O(n2)T(n)=O(n2)
- 最差情况： T(n)=O(n2)T(n)=O(n2)
- 平均情况： T(n)=O(n2)T(n)=O(n2)

插入排序（INSERTION SORT）[3]

算法原理

*插入排序（Insertion Sort）是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前 s 扫描，找到相应位置并插入。插入排序在实现上，通常采用 in-place 排序（即只需用到*O(1)O(1) *的额外空间的排序），因而在从后向前的扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。*



图解如下:

算法描述与实现

- 具体算法描述如下：

一般来说，插入排序都采用 in-place 在数组上实现。具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置
4. 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤 2 52 5

如果*比较操作*的代价比*交换操作大*的话，可以采用*二分查找法*来减少比较操作的数目。该算法可以认为是插入排序的一个变种，称为*二分查找插入排序*。

- 代码实现如下：

```
const insertionSort = (arr) => {
  let len = arr.length;
  console.time('插入排序耗时');
  for (let i = 1;i < len; i++){
    let key = arr[i];
    let j = i - 1;
    while(j >= 0 && arr[j] > key){
      arr[j+1] = arr[j];
      j--;
    }
    arr[j+1] = key;
  }
  console.timeEnd('插入排序耗时');
  return arr;
}
```

改进插入排序：查找插入位置时使用二分查找的方式。

具体思路如下：

1. 在插入第 ii 个元素时，对前面的 0 i−10 i−1 元素进行折半。
2. 与前面的 0 i−10 i−1 个元素中间的元素进行比较，如果小，则对前半再进行折半，否则对后半进行折半。
3. 直到 left > right，再把第 ii 个元素前 1 位和目标位置间的所有元素后移，把第 ii 个元素放在目标位置上。

- 代码实现如下：

```
const binaryInsertionSort = (arr) => {
```

```
  console.time('二分插入排序耗时：');
  for (let i = 1; i < arr.length; i++){
    let key = arr[i], left = 0, right = i - 1;
    while (left <= right){
      let middle = parseInt((left + right) / 2);
      if (key < arr[middle]){
        right = middle - 1;
      }else{
        left = middle + 1;
      }
    }
    for (let j = i - 1; j >= left; j--){
      arr[j + 1] = arr[j];
    }
    arr[left] = key;
  }
  console.timeEnd('二分插入排序耗时：');
  return arr;
}
```

算法分析

- 最佳情况： T(n)=O(n)T(n)=O(n)
- 最差情况： T(n)=O(n2)T(n)=O(n2)
- 平均情况： T(n)=O(n2)T(n)=O(n2)

希尔排序（SHELL SORT）[4]

算法原理

希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。它是非稳定排序算法。

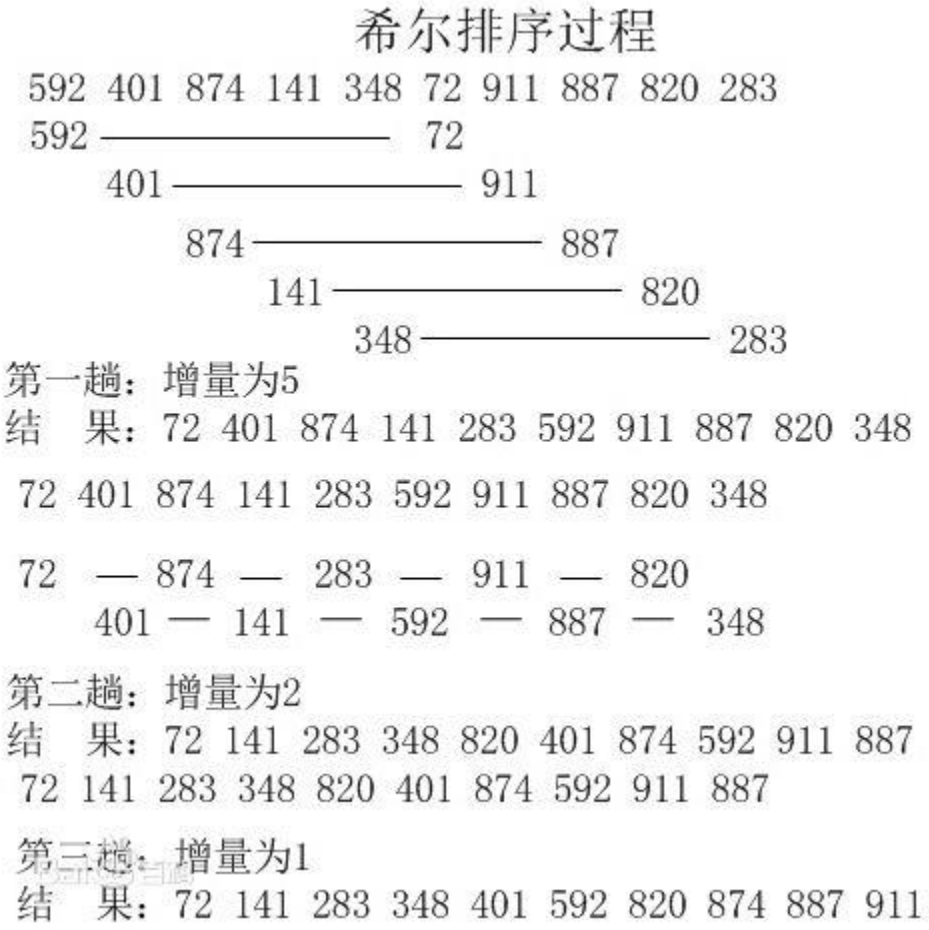
希尔排序基于插入排序的以下两点性质提出了改进方法：

1. 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率。

2. 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。

它与插入排序的不同之处在于，它会优先比较距离较远的元素。因此希尔排序又叫缩小增量排序。

图解如下:



算法描述与实现

- 具体算法描述如下：

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

1. 选择一个增量序列 t1，t2，…，tkt1，t2，…，tk，其中 ti>tj(i>j)ti>tj(i>j)，tk=1tk=1；
2. 按增量序列个数 kk，对序列进行 kk 趟排序；
3. 每趟排序，根据对应的增量 titi，将待排序列分割成若干长度为 mm 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

- 代码实现如下：

```
const shellSort = (arr) => {
 console.time('希尔排序耗时:');
 let len = arr.length,temp,gap = 1;
 while(gap < len / 5) {// 动态定义间隔序列步长为5
    gap = gap * 5 + 1;
 }
 for (gap; gap > 0; gap = Math.floor(gap / 5)) {
  for (let i = gap; i < len; i++) {
    temp = arr[i];
    let j;
    for (j = i - gap; j >= 0 && arr[j] > temp; j -= gap) {
      arr[j + gap] = arr[j];
    }
    arr[j + gap] = temp;
  }
 }
```

```
    console.timeEnd('希尔排序耗时:');
    return arr;
}
```

算法分析

- 最佳情况： T(n)=O(nlog2n)T(n)=O(nlog2n)
- 最差情况： T(n)=O(nlog2n)T(n)=O(nlog2n)
- 平均情况： T(n)=O(nlogn)T(n)=O(nlogn)

归并排序（MERGE SORT）[5]

算法原理

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

图解如下：

算法描述与实现

- 具体算法描述如下：

1. 把长度为 n 的输入序列分成两个长度为 n/2 的子序列；
2. 对这两个子序列分别采用归并排序；
3. 将两个排序好的子序列合并成一个最终的排序序列。

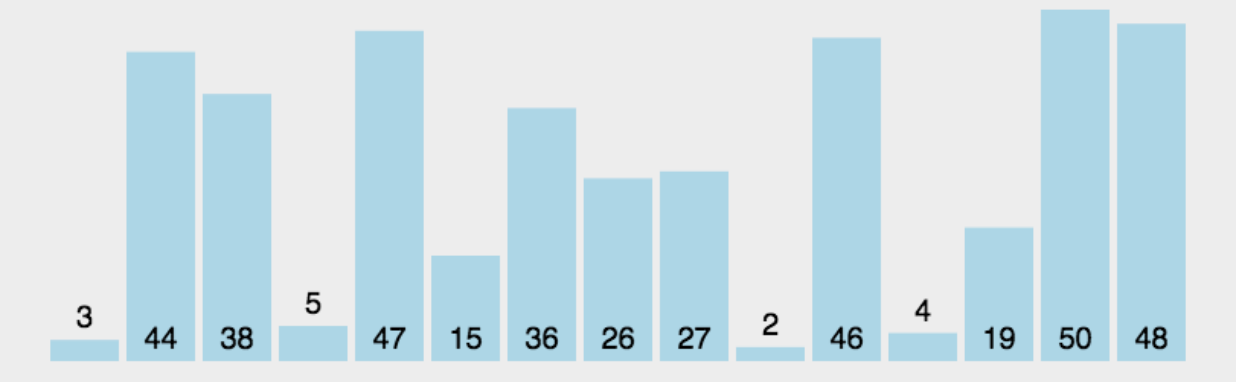
- 代码实现如下：

```
"use strict"
const mergeSort = (arr) =>{ //采用自上而下的递归方法
```

```
    let len = arr.length;
    if(len < 2) {
        return arr;
    }
    let middle = Math.floor(len / 2),
        left = arr.slice(0, middle),
        right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort(right));
}

const merge = (left, right) => {
    let result = [];
    while (left.length && right.length) {
        if (left[0] <= right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }
    while (left.length)
        result.push(left.shift());
    while (right.length)
        result.push(right.shift());
    return result;
}

let arr=[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48];
console.time('归并排序耗时');
console.log(mergeSort(arr));
console.timeEnd('归并排序耗时');
```

算法分析

和选择排序一样，归并排序的性能不受输入数据的影响，但它的表现比选择排序要好，因为它始终都是 O(nlogn)O(nlogn) 的时间复杂度。但代价是需要额外的内存空间。



- 最佳情况： T(n)=O(n)T(n)=O(n)
- 最差情况： T(n)=O(nlogn)T(n)=O(nlog⁡n)
- 平均情况： T(n)=O(nlogn)T(n)=O(nlog⁡n)

快速排序（QUICK SORT）[6]

算法原理

通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

算法描述与实现

- 具体算法描述如下：

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

1. 从数列中挑出一个元素，称为 “基准”（pivot）；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

- 代码实现如下：

```
'use strict'
const quickSort = (arr) => {
    if (arr.length <= 1) { return arr; }
    let pivotIndex = Math.floor(arr.length / 2);
    let pivot = arr.splice(pivotIndex, 1)[0];
    let left = [];
    let right = [];
    for (let i = 0; i < arr.length; i++){
            if (arr[i] < pivot) {
                    left.push(arr[i]);
            } else {
                    right.push(arr[i]);
            }
    }
    return quickSort(left).concat([pivot], quickSort(right));
};
let arr=[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48];
console.time('快速排序耗时');
console.log(quickSort(arr));
console.timeEnd('快速排序耗时');
```

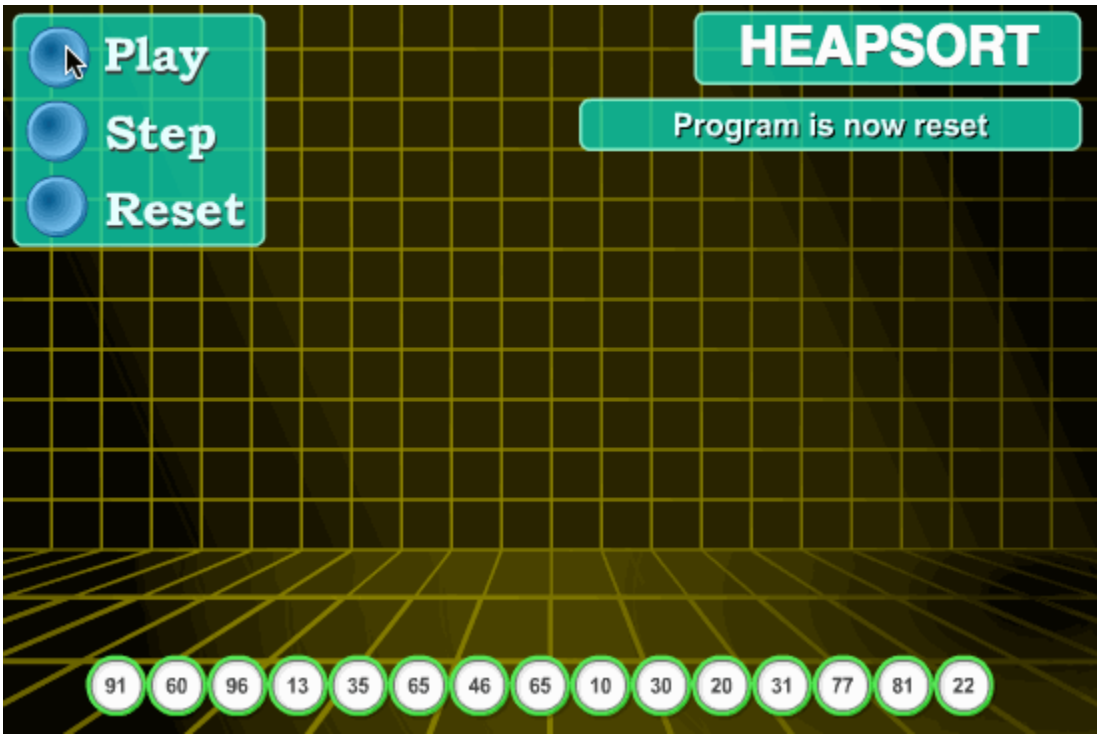
算法分析

- 最佳情况： T(n)=O(nlogn)T(n)=O(nlog n)
- 最差情况： T(n)=O(n2)T(n)=O(n2)
- 平均情况： T(n)=O(nlogn)T(n)=O(nlog n)

堆排序（HEAP SORT）[7]

算法原理

堆排序可以说是一种利用堆的概念来排序的选择排序。它利用堆这种数据结构所设计。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。



- 具体算法描述如下：

1. 将初始待排序关键字序列 (R1,R2…,Rn)(R1,R2…,Rn) 构建成大顶堆，此堆为初始的无序区；
2. 将堆顶元素 R[1]R[1] 与最后一个元素 R[n]R[n] 交换，此时得到新的无序区 (R1,R2,……,Rn−1)(R1,R2,……,Rn−1) 和新的有序区 (Rn)(Rn)，且满足 R[1,2…,n−1]<=R[n]R[1,2…,n−1]<=R[n]；
3. 由于交换后新的堆顶 R[1]R[1] 可能违反堆的性质，因此需要对当前无序区 (R1,R2,……,Rn−1)(R1,R2,……,Rn−1) 调整为新堆，然后再次将 R[1]R[1] 与无序区最后一个元素交换，得到新的无序区 (R1,R2…,Rn−2)(R1,R2…,Rn−2) 和新的有序区 (Rn−1,Rn)(Rn−1,Rn)。不断重复此过

程直到有序区的元素个数为 n-1，则整个排序过程完成。

- 代码实现如下：

```
"use strict"
const heapSort = (array) => {
    console.time('堆排序耗时');
    //建堆
    let heapSize = array.length, temp;
    for (let i = Math.floor(heapSize / 2) - 1; i >= 0; i--) {
        heapify(array, i, heapSize);
    }
    //堆排序
    for (let j = heapSize - 1; j >= 1; j--) {
        temp = array[0];
        array[0] = array[j];
        array[j] = temp;
        heapify(array, 0, --heapSize);
    }
    console.timeEnd('堆排序耗时');
    return array;
}
/*方法说明：维护堆的性质
@param  arr 数组
@param  x   数组下标
@param  len 堆大小*/
const heapify = (arr, x, len) => {
    let l = 2 * x + 1, r = 2 * x + 2, largest = x, temp;
    if (l < len && arr[l] > arr[largest]) {
        largest = l;
    }
    if (r < len && arr[r] > arr[largest]) {
        largest = r;
    }
    if (largest != x) {
```

```
32          temp = arr[x];
33          arr[x] = arr[largest];
34          arr[largest] = temp;
35          heapify(arr, largest, len);
36        }
37    }
38    let arr=[91,60,96,13,35,65,46,65,10,30,20,31,77,81,22];
39    console.log(heapSort(arr));
```

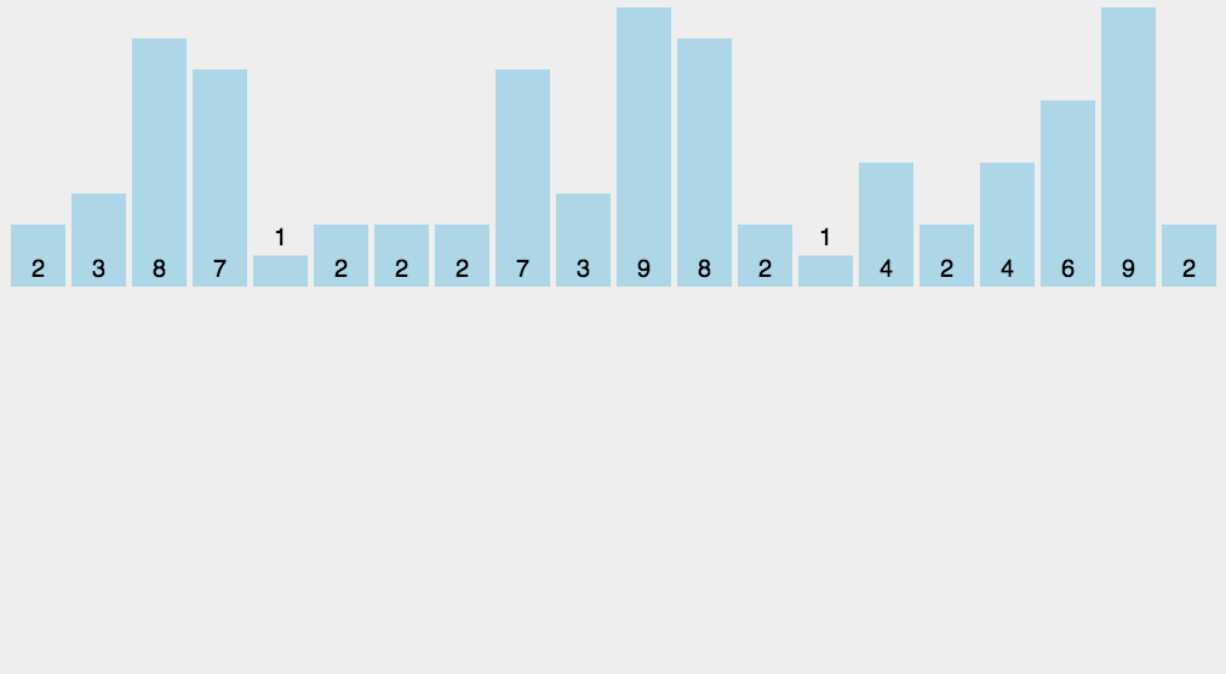
算法分析

- 最佳情况： T(n)=O(nlogn)T(n)=O(nlog⁡n)
- 最差情况： T(n)=O(nlogn)T(n)=O(nlog⁡n)
- 平均情况： T(n)=O(nlogn)T(n)=O(nlog⁡n)

计数排序（COUNTING SORT）[8]

算法原理

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组 C，其中第 i 个元素是待排序数组 A 中值等于 i 的元素的个数。然后根据数组 C 来将 A 中的元素排到正确的位置。它只能对整数进行排序。



算法描述与实现

- 具体算法描述如下：

1. 找出待排序的数组中最大和最小的元素；
2. 统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项；
3. 对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）；
4. 反向填充目标数组：将每个元素 i 放在新数组的第 C(i)C(i) 项，每放一个元素就将 C(i)C(i) 减去 1。

- 代码实现如下：

```
'use strict'
const countingSort = (array) => {
    let len = array.length,
        result = [],
        C = [],
        min,max;
```

```
    min = max = array[0];
  console.time('计数排序耗时');
  for (let i = 0; i < len; i++) {
    min = min <= array[i] ? min : array[i];
    max = max >= array[i] ? max : array[i];
    C[array[i]] = C[array[i]] ? C[array[i]] + 1 : 1;
  }
  for (let j = min; j < max; j++) {
    C[j + 1] = (C[j + 1] || 0) + (C[j] || 0);
  }
  for (let k = len - 1; k >= 0; k--) {
    result[C[array[k]] - 1] = array[k];
    C[array[k]]--;
  }
  console.timeEnd('计数排序耗时');
  return result;
}
let arr = [2, 2, 3, 8, 7, 1, 2, 2, 2, 7, 3, 9, 8, 2, 1, 4, 2, 4, 6, 9, 2];
console.log(countingSort(arr));
```

算法分析

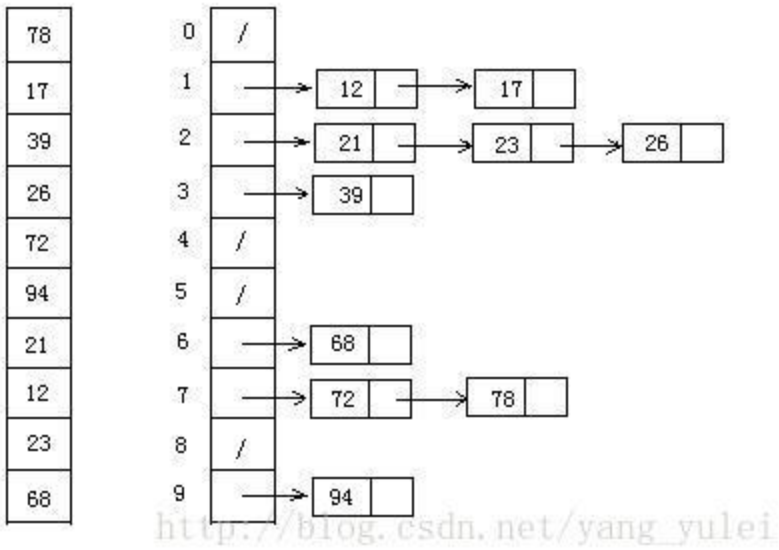
计数排序的核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

当输入的元素是 n 个 0 到 k 之间的整数时，它的运行时间是 O(n + k)。计数排序不是比较排序，排序的速度快于任何比较排序算法。由于用来计数的数组 C 的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上 1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。

- 最佳情况： T(n)=O(n+k)T(n)=O(n+k)
- 最差情况： T(n)=O(n+k)T(n)=O(n+k)
- 平均情况： T(n)=O(n+k)T(n)=O(n+k)

桶排序（BUCKET SORT）[9]

算法原理



工作的原理是将数组分到有限数量的桶里。每个桶再个别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序）

算法描述与实现

- 具体算法描述如下：

1. 设置一个定量的数组当作空桶；
2. 遍历输入数据，并且把数据一个一个放到对应的桶里去；
3. 对每个不是空的桶进行排序；
4. 从不是空的桶里把排好序的数据拼接起来。

- 代码实现如下：

```
"use strict"
const bucketSort = (array, num) => {
    if (array.length <= 1) {
        return array;
    }
    var len = array.length, buckets = [], result = [], min = max = array[0], regex = '/^[1-9]+[0-
9]*$/', space, n = 0;
    num = num || ((num > 1 && regex.test(num)) ? num : 10);
    console.time('桶排序耗时');
    for (var i = 1; i < len; i++) {
```

```
11        min = min <= array[i] ? min : array[i];
12        max = max >= array[i] ? max : array[i];
13      }
14      space = (max - min + 1) / num;
15      for (var j = 0; j < len; j++) {
16        var index = Math.floor((array[j] - min) / space);
17        if (buckets[index]) {  // 非空桶，插入排序
18          var k = buckets[index].length - 1;
19          while (k >= 0 && buckets[index][k] > array[j]) {
20            buckets[index][k + 1] = buckets[index][k];
21            k--;
22          }
 2          buckets[index][k + 1] = array[j];
        } else {   //空桶，初始化
          buckets[index] = [];
          buckets[index].push(array[j]);
        }
      }
      while (n < num) {
        result = result.concat(buckets[n]);
        n++;
      }
      console.timeEnd('桶排序耗时');
      return result;
    }
    var arr=[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48];
    console.log(bucketSort(arr,4));
```

算法分析

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。 桶排序最好情况下使用线性时间 O(n)O(n)，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为 O(n)O(n)。很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

- 最佳情况： T(n)=O(n+k)T(n)=O(n+k)
- 最差情况： T(n)=O(n+k)T(n)=O(n+k)
- 平均情况： T(n)=O(n2)T(n)=O(n2)

基数排序（RADIX SORT）[10]

算法原理

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

算法描述与实现

- 具体算法描述如下：

1. 取得数组中的最大数，并取得位数；
2. arr 为原始数组，从最低位开始取每个位组成 radix 数组；

3. 对 radix 进行计数排序（利用计数排序适用于小范围数的特点）；

- 代码实现如下：

```
'use strict'
const radixSort = (arr, maxDigit) => {
    var mod = 10;
    var dev = 1;
    var counter = [];
    console.time('基数排序耗时');
    for (var i = 0; i < maxDigit; i++, dev *= 10, mod *= 10) {
        for(var j = 0; j < arr.length; j++) {
            var bucket = parseInt((arr[j] % mod) / dev);
            if(counter[bucket]== null) {
                counter[bucket] = [];
            }
            counter[bucket].push(arr[j]);
        }
        var pos = 0;
        for(var j = 0; j < counter.length; j++) {
            var value = null;
            if(counter[j]!=null) {
                while ((value = counter[j].shift()) != null) {
                    arr[pos++] = value;
                }
            }
        }
    }
    console.timeEnd('基数排序耗时');
    return arr;
}
var arr = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48];
console.log(radixSort(arr,2));
```

算法分析

基数排序有两种方法：

MSD 从高位开始进行排序 LSD 从低位开始进行排序

- 最佳情况： T(n)=O(n2)T(n)=O(n2)
- 最差情况： T(n)=O(n2)T(n)=O(n2)
- 平均情况： T(n)=O(n2)T(n)=O(n2)

基数排序 vs 计数排序 vs 桶排序

这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

基数排序：根据键值的每位数字来分配桶 计数排序：每个桶只存储单一键值 桶排序：每个桶存储一定范围的数值

以上代码可访问 GitHub 具体查看

JS 原生函数中的排序

说到前端排序，自然首先会想到 JavaScript 的原生接口 `Array.prototype.sort`

这个接口自 ECMAScript 1st Edition 起就被设计存在。而在规范中关于它的描述是这样的：[11]

`Array.prototype.sort(compareFn)`

The elements of this array are sorted. The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). If comparefn is not undefined, it should be a function that accepts two arguments x and y and returns a negative value if x < y, zero if x = y, or a positive value if x > y.

显然，规范里并没有限定 sort 内部需要用什么排序算法。在这样的背景下，前端排序这件事完全取决于各家浏览器内核的具体实现。

Chrome 的实现

Chrome 的 JavaScript 引擎是 v8。由于它是开源的，所以可以直接看源代码。

整个 array.js 都是用 JavaScript 语言实现的。排序方法部分很明显比通常看到的快速排序要复杂得多，但显然核心算法还是快速排序的思想。算法复杂的原因在于 v8 出于性能考虑进行了很多优化。(后续会展开说)

Firefox 中的实现

暂时无法确定 Firefox 的 JavaScript 引擎即将使用的数组排序算法会是什么。

按照现有的信息，SpiderMoney 内部实现了归并排序。（这里不多做叙述）

Microsoft Edge 中的实现

Microsoft Edge 的 JavaScript 引擎 Chakra 的核心部分代码已经于 2016 年初在 Github 开源。

通过源代码可以发现，Chakra 的数组排序算法也主要是以快速排序为主。并针对其他具体特殊情况进行具体优化。

排序的差异

如上所述，快速排序是一种不稳定的排序算法，而归并排序是一种稳定的排序算法。由于不同引擎在算法选择上可能存在差异，导致前端如果依赖 Array.prototype.sort 接口实现的 JavaScript 代码，在浏览器中实际执行的排序效果是不一致的。

而排序稳定性的差异其实也只是在特定的场景才会体现出问题；在很多情况下，不稳定的排序也并不会造成什么影响。所以假如实际项目开发中，对于数组的排序没有稳定性的需求，那么看到这里就可以了。

但是若项目要求排序必须是稳定的，那么这些差异的存在将无法满足需求。我们需要为此进行一些额外的工作。

举个例子：

某市的机动车牌照拍卖系统，最终中标的规则为：

```
1. 按价格进行倒排序；
2. 相同价格则按照竞标顺位（即价格提交时间）进行正排序。
```

排序若是在前端进行，那么采用快速排序的浏览器中显示的中标者很可能是不符合预期的。

此外例如： MySQL 5.6 的 limit M,N 语句实现 等情况都是不稳定排序的特征。

背后的原因

Chrome 为什么采用快速排序

其实这个情况从一开始便存在。

Chrome 测试版于 2008 年 9 月 2 日发布 (这里附上当时随版本发布的漫画，还是很有意思的 Google on Google Chrome - comic book)，然而发布后不久，就有开发者向 Chromium 开发组提交 #90 Bug V8 doesn't stable sort 反馈 v8 的数组排序实现不是稳定排序的。

这个 Bug ISSUE 讨论的时间跨度很大。时至今日，仍然有开发者对 v8 的数组排序的实现提出评论。

同时我们还注意到，该 ISSUE 曾经已被关闭。但是于 2013 年 6 月被开发组成员重新打开，作为 ECMAScript Next 规范讨论的参考。

而 es-discuss 的最后结论是这样的

It does not change. Stable is a subset of unstable. And vice versa, every unstable algorithm returns a stable result for some inputs. Mark's point is that requiring "always unstable" has no meaning, no matter what language you chose.

这也正如本文前段所引用的已定稿 ECMAScript 2015 规范中的描述一样。

时代特点

IMHO，Chrome 发布之初即被报告出这个问题可能是有其特殊的时代特点。

上文已经说到，Chrome 第一版是 2008 年 9 月发布的。根据 statcounter 的统计数据，那个时期市场占有率最高的两款浏览器分别是 IE(那时候只有 IE6 和 IE7) 和 Firefox，市场占有率分别达到了 67.16% 和 25.77%。也就是说，两个浏览器加起来的市场占有率超过了 90%。

而根据另一份浏览器排序算法稳定性的统计数据显示，这两款超过了 90% 市场占有率的浏览器都采用了稳定的数组排序。所以 Chrome 发布之初被开发者质疑也是合情合理的。

我们从 ISSUE 讨论的过程中，可以大概理解开发组成员对于引擎实现采用快速排序的一些考量。他们认为引擎必须遵守 ECMAScript 规范。由于规范不要求稳定排序的描述，故他们认为 v8 的实现也是完全符合规范的。

性能考虑

另外，他们认为 v8 设计的一个重要考量在于引擎的性能。

快速排序相比较于**归并排序**，在整体性能上表现更好：

- 更高的计算效率。快速排序在实际计算机执行环境中比同等时间复杂度的其他排序算法更快（不命中最差组合的情况下）
- 更低的空间成本。前者仅有 O(logn)O(logn) 的空间复杂度，相比较后者 O(n)O(n) 的空间复杂度在运行时的内存消耗更少

v8 在数组排序算法中的性能优化

既然说 v8 非常看中引擎的性能，那么在数组排序中它做了哪些事呢？

通过阅读源代码，还是粗浅地学习了一些皮毛。

- 混合插入排序 快速排序是分治的思想，将大数组分解，逐层往下递归。但是若递归深度太大，为了维持递归，调用栈的内存资源消耗也会很大。优化不好甚至可能造成栈溢出。

目前 v8 的实现是设定一个阈值，对最下层的 10 个及以下长度的小数组使用插入排序。

根据代码注释以及 Wikipedia 中的描述，虽然插入排序的平均时间复杂度为 O(n2)O(n2) 差于快速排序的 O(nlogn)O(nlogn)。但是在运行环境，小数组使用插入排序的效率反而比快速排序会更高，这里不再展开。

v8 代码示例:

```
var QuickSort = function QuickSort(a, from, to) {
  ......
  while (true) {
    // Insertion sort is faster for short arrays.
    if (to - from <= 10) {
      InsertionSort(a, from, to);
      return;
    }
    ......
  }
  ......
```

```
};
```

- 三数取中

正如已知的，快速排序的阿克琉斯之踵在于，最差数组组合情况下会算法退化。

快速排序的算法核心在于选择一个基准(**pivot**)，将经过比较交换的数组按基准分解为两个数区进行后续递归。试想如果对一个已经有序的数组，每次选择基准元素时总是选择第一个或者最后一个元素，那么每次都会有一个数区是空的，递归的层数将达到 **n**，最后导致算法的时间复杂度退化为 O(n2)O(n2)。因此 pivot 的选择非常重要。
v8 采用的是**三数取中(median-of-three)**的优化：除了头尾两个元素再额外选择一个元素参与基准元素的竞争。

第三个元素的选取策略大致为：

1. 当数组长度小于等于 **1000** 时，选择折半位置的元素作为目标元素。
2. 当数组长度超过 **1000** 时，每隔 **200-215** 个(***非固定，跟着数组长度而变化***)左右选择一个元素来先确定一批候选元素。接着在这批候选元素中进行一次排序，将所得的中位值作为目标元素

最后取三个元素的中位值作为 **pivot**。

v8 代码示例：

```
var GetThirdIndex = function(a, from, to) {
  var t_array = new InternalArray();
  // Use both 'from' and 'to' to determine the pivot candidates.
  var increment = 200 + ((to - from) & 15);
  var j = 0;
  from += 1;
  to -= 1;
  for (var i = from; i < to; i += increment) {
    t_array[j] = [i, a[i]];
    j++;
  }
```

```
    t_array.sort(function(a, b) {
        return comparefn(a[1], b[1]);
    });
    var third_index = t_array[t_array.length >> 1][0];
    return third_index;
};

var QuickSort = function QuickSort(a, from, to) {
    ......
    while (true) {
        ......
        if (to - from > 1000) {
            third_index = GetThirdIndex(a, from, to);
        } else {
            third_index = from + ((to - from) >> 1);
        }
    }
    ......
};
```

- 原地排序

目前，大多数快速排序算法中大部分的代码实现如下所示：

```
var quickSort = function(arr) {
      if (arr.length <= 1) { return arr; }
      var pivotIndex = Math.floor(arr.length / 2);
      var pivot = arr.splice(pivotIndex, 1)[0];
      var left = [];
      var right = [];
      for (var i = 0; i < arr.length; i++){
            if (arr[i] < pivot) {
                  left.push(arr[i]);
            } else {
                  right.push(arr[i]);
```

```
            }
        }
        return quickSort(left).concat([pivot], quickSort(right));
};
```

以上代码存在一个问题在于：利用 left 和 right 两个数区存储递归的子数组，因此它需要 O(n)O(n) 的额外存储空间。这与理论上的平均空间复杂度 O(logn)O(logn) 相比差距较大。
额外的空间开销，同样会影响实际运行时的整体速度。（这也是快速排序在实际运行时的表现可以超过同等时间复杂度级别的其他排序算法的其中一个原因。）所以一般来说，性能较好的快速排序会采用原地(in-place)排序的方式。

v8 源代码中的实现是对原数组进行元素交换。

Firefox 为什么采用归并排序

它的背后也是有故事的。

Firefox 其实在一开始发布的时候对于数组排序的实现并不是采用稳定的排序算法，这块有据可考。

Firefox(Firebird) 最初版本实现的数组排序算法是堆排序，这也是一种不稳定的排序算法。因此，后来有人对此提交了一个 Bug。

Mozilla 开发组内部针对这个问题进行了一系列讨论。

从讨论的过程我们能够得出几点

- 同时期 Mozilla 的竞争对手是 **IE6**，从上文的统计数据可知 IE6 是稳定排序的
- JavaScript 之父 Brendan Eich 觉得 **Stability is good**
- Firefox 在采用**堆排序**之前采用的是**快速排序**

基于开发组成员倾向于实现稳定的排序算法为主要前提，**Firefox3** 将**归并排序**作为了数组排序的新实现。

解决排序稳定性的差异

以上说了这么多，主要是为了讲述各个浏览器对于排序实现的差异，以及解释为什么存在这些差异的一些比较表层的原因。

但是读到这里，读者可能还是会有疑问：如果我的项目就是需要依赖稳定排序，那该怎么办呢？

解决方案

其实解决这个问题的思路比较简单。

浏览器出于不同考虑选择不同排序算法。可能某些偏向于追求极致的性能，某些偏向于提供良好的开发体验，但是有规律可循。

从目前已知的情况来看，所有主流浏览器（包括 IE6，7，8）对于数组排序算法的实现基本可以枚举：

- 归并排序 / Timsort
- 快速排序

所以，我们将快速排序经过定制改造，变成稳定排序的是不是就可以了？

一般来说，针对对象数组使用不稳定排序会影响结果。而其他类型数组本身使用稳定排序或不稳定排序的结果是相等的。因此方案大致如下：

- 将待排序数组进行预处理，为每个待排序的对象增加自然序属性，不与对象的其他属性冲突即可。
- 自定义排序比较方法 compareFn，总是将自然序作为前置判断相等时的第二判断维度。

面对归并排序这类实现时由于算法本身就是稳定的，额外增加的自然序比较并不会改变排序结果，所以方案兼容性比较好。

但是涉及修改待排序数组，而且需要开辟额外空间用于存储自然序属性，可想而知 v8 这类引擎应该不会采用类似手段。不过作为开发者自行定制的排序方案是可行的。

方案代码示例

```
1  'use strict';
2
```

```
3  const INDEX = Symbol('index');
4
5  function getComparer(compare) {
6    return function (left, right) {
7      let result = compare(left, right);
8
9      return result === 0 ? left[INDEX] - right[INDEX] : result;
10   };
11 }
12
13 function sort(array, compare) {
14   array = array.map(
15     (item, index) => {
16       if (typeof item === 'object') {
17         item[INDEX] = index;
18       }
19
20       return item;
21     }
22   );
23
24   return array.sort(getComparer(compare));
25 }
```

以上只是一个简单的满足稳定排序的算法改造示例。

之所以说简单，是因为实际生产环境中作为数组输入的数据结构冗杂，需要根据实际情况判断是否需要进行更多样的排序前类型检测。

选择排序算法的参考方法：

影响排序的因素有很多，平均时间复杂度低的算法并不一定就是最优的。相反，有时平均时间复杂度高的算法可能更适合某些特殊情况。同时，选择算法时还得考虑它的可读性，以利于软件的维护。一般而言，需要考虑的因素有以下四点：

1. 待排序的记录数目 n 的大小；

2. 记录本身数据量的大小，也就是记录中除关键字外的其他信息量的大小；
3. 关键字的结构及其分布情况；
4. 对排序稳定性的要求。

设待排序元素的个数为 n。选择的大致方案如下：

1. 当 n 较大，则应采用时间复杂度为 O(nlogn)O(nlogn) 的排序方法：快速排序、堆排序或归并排序。

- 快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；
- 堆排序：如果内存空间允许且要求稳定性的，
- 归并排序：它有一定数量的数据移动，所以我们可能过与插入排序组合，先获得一定长度的序列，然后再合并，在效率上将有所提高。

2. 当 n 较大，内存空间允许，且要求稳定性则选择归并排序
3. 当 n 较小，可采用直接插入或直接选择排序。

直接插入排序：当元素分布有序，直接插入排序将大大减少比较次数和移动记录的次数。

直接选择排序：元素分布有序，如果不要求稳定性，选择直接选择排序

4. 一般不使用或不直接使用传统的冒泡排序。
5. 基数排序： 它是一种稳定的排序算法，但有一定的局限性，最好满足：

- 关键字可分解。

- 记录的关键字位数较少，如果密集更好
- 如果是数字时，最好是无符号的，否则将增加相应的映射复杂度，可先将其正负分开排序。

参考资料 & 拓展阅读

十大经典排序算法

聊聊前端排序的那些事

Array.prototype.sort 随机排列数组的错误实现

番外

睡排序（Sleep sort)

复杂度：O(WTF)

```
var numbers = [8, 32, 49, 4, 111, 999];

numbers.forEach(item => {
   setTimeout(() => {console.log(item)},item);
})
```

---

1. 冒泡排序概念 ↩
2. 选择排序概念 ↩
3. 插入排序概念 ↩
4. 希尔排序概念 ↩
5. 归并排序概念 ↩
6. 快速排序概念 ↩
7. 堆排序概念 ↩
8. 计数排序概念 ↩

9. 桶排序概念 ↩
10. 基数排序概念 ↩
11. Array.prototype.sort(compareFn) ↩

在 Javascript 中学习数据结构与算法

这是一本 5 万字符（中文约 2w）的小书，可能需要几个小时阅读，需要几天或更多时间去消化。部分代码还不能正确地跑起来，有错别字，有不准确的概念...，但这不妨碍它作为你一个野生前端学习数据结构与算法的启蒙文章，期待你的一针见血、刀刀致命😐

对任何专业技术人员来说，理解数据结构都非常重要。作为软件开发者，我们要能够用编程语言和数据结构来解决问题。编程语言和数据结构是这些问题解决方案中不可或缺的一部分。如果选择了不恰当的数据结构，可能会影响所写程序的性能。因此，了解不同数据结构和它们的适用范围十分重要。

一句话：算法即原力，即正义

本文主要讲述 Javascript 中实现栈、队列、链表、集合、字典、散列表、树、图等数据结构，以及各种排序和搜索算法，包括冒泡排序、选择排序、插入排序、归并排序、快速排序、顺序搜索、二分搜索，最后还介绍了动态规划和贪心算法等常用的高级算法及相关知识。

在阅读之前假设你已了解并可以熟练使用 Javascript 编写应用程序。

概览

数据结构

- 栈：一种遵从先进后出 (LIFO) 原则的有序集合；新添加的或待删除的元素都保存在栈的末尾，称作栈顶，另一端为栈底。在栈里，新元素都靠近栈顶，旧元素都接近栈底。
- 队列：与上相反，一种遵循先进先出 (FIFO / First In First Out) 原则的一组有序的项；队列在尾部添加新元素，并从头部移除元素。最新添加的元素必须排在队列的末尾。
- 链表：存储有序的元素集合，但不同于数组，链表中的元素在内存中并不是连续放置的；每个元素由一个存储元素本身的节点和一个指向下一个元素的引用（指针/链接）组成。

- **集合**：由一组无序且唯一（即不能重复）的项组成；这个数据结构使用了与有限集合相同的数学概念，但应用在计算机科学的数据结构中。
- **字典**：以 [键，值] 对为数据形态的数据结构，其中键名用来查询特定元素，类似于 Javascript 中的 Object。
- **散列**：根据关键码值（Key value）直接进行访问的数据结构；它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度；这个映射函数叫做散列函数，存放记录的数组叫做散列表。
- **树**：由 n（n>=1）个有限节点组成一个具有层次关系的集合；把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的，基本呈一对多关系，树也可以看做是图的特殊形式。
- **图**：图是网络结构的抽象模型；图是一组由边连接的节点（顶点）；任何二元关系都可以用图来表示，常见的比如：道路图、关系图，呈多对多关系。

算法

排序算法

- **冒泡排序**：比较任何两个相邻的项，如果第一个比第二个大，则交换它们；元素项向上移动至正确的顺序，好似气泡上升至表面一般，因此得名。
- **选择排序**：每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，以此循环，直至排序完毕。
- **插入排序**：将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，此算法适用于少量数据的排序，时间复杂度为 O(n^2)。
- **归并排序**：将原始序列切分成较小的序列，只到每个小序列无法再切分，然后执行合并，即将小序列归并成大的序列，合并过程进行比较排序，只到最后只有一个排序完毕的大序列，时间复杂度为 O(n log n)。
- **快速排序**：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行上述递归排序，以此达到整个数据变成有序序列，时间复杂度为 O(n log n)。

搜索算法

- **顺序搜索**：让目标元素与列表中的每一个元素逐个比较，直到找出与给定元素相同的元素为止，缺点是效率低下。
- **二分搜索**：在一个有序列表，以中间值为基准拆分为两个子列表，拿目标元素与中间值作比较从而再在目标的子列表中递归此方法，直至找到目标元素。

其他

贪心算法：在对问题求解时，不考虑全局，总是做出局部最优解的方法。
动态规划：在对问题求解时，由以求出的局部最优解来推导全局最优解。
复杂度概念：一个方法在执行的整个生命周期，所需要占用的资源，主要包括：时间资源、空间资源。

数据结构

栈

栈是一种遵从先进后出 (LIFO) 原则的有序集合；新添加的或待删除的元素都保存在栈的末尾，称作栈顶，另一端为栈底。在栈里，新元素都靠近栈顶，旧元素都接近栈底。

通俗来讲，一摞叠起来的书或盘子都可以看做一个栈，我们想要拿出最底下的书或盘子，一定要现将上面的移走才可以。

栈也被用在编程语言的编译器和内存中保存变变量、方法调用。

在 Javascript 中我们可以使用数组的原生方法实现一个栈/队列的功能，鉴于学习目的，我们使用类来实现一个栈。

```
class Stack {
    constructor() {
        this.items = []
    }

    // 入栈
    push(element) {
        this.items.push(element)
    }

    // 出栈
    pop() {
        return this.items.pop()
```

```
    }

    // 末位
    get peek() {
        return this.items[this.items.length - 1]
    }

    // 是否为空栈
    get isEmpty() {
        return !this.items.length
    }

    // 尺寸
    get size() {
        return this.items.length
    }

    // 清空栈
    clear() {
        this.items = []
    }

    // 打印栈数据
    print() {
        console.log(this.items.toString())
    }
}
```

使用栈类：

```
// 实例化一个栈
const stack = new Stack()
console.log(stack.isEmpty) // true
```

```
// 添加元素
stack.push( 5 )
stack.push( 8 )

// 读取属性再添加
console .log(stack.peek) // 8
stack.push( 11 )
console .log(stack.size) // 3
console .log(stack.isEmpty) // false
```

队列

与栈相反，队列是一种遵循先进先出 (FIFO / First In First Out) 原则的一组有序的项；队列在尾部添加新元素，并从头部移除元素。最新添加的元素必须排在队列的末尾。

在现实中，最常见的例子就是排队，吃饭排队、银行业务排队、公车的前门上后门下机制...，前面的人优先完成自己的事务，完成之后，下一个人才能继续。

在计算机科学中，一个常见的例子就是打印队列。比如说我们需要打印五份文档。我们会打开每个文档，然后点击打印按钮。每个文档都会被发送至打印队列。第一个发送到打印队列的文档会首先被打印，以此类推，直到打印完所有文档。

同样的，我们在 Javascript 中实现一个队列类。

```
class Queue {

    constructor ( items ) {
        this .items = items || []
    }
```

```
    enqueue(element){
        this.items.push(element)
    }

    dequeue(){
        return this.items.shift()
    }

    front(){
        return this.items[0]
    }

    clear(){
        this.items = []
    }

    get size(){
        return this.items.length
    }

    get isEmpty(){
        return !this.items.length
    }

    print() {
        console.log(this.items.toString())
    }
}
```

使用队列类：

```
const queue = new Queue()
console.log(queue.isEmpty) // true
```

```
queue.enqueue( 'John' )
queue.enqueue( 'Jack' )
queue.enqueue( 'Camila' )
console.log(queue.size) // 3
console.log(queue.isEmpty) // false
queue.dequeue()
queue.dequeue()
queue.print() // 'Camila'
```

优先队列

队列大量应用在计算机科学以及我们的生活中，我们在之前话题中实现的默认队列也有一些修改版本。

其中一个修改版就是优先队列。元素的添加和移除是基于优先级的。一个现实的例子就是机场登机的顺序。头等舱和商务舱乘客的优先级要高于经济舱乘客。在有些国家，老年人和孕妇(或 带小孩的妇女)登机时也享有高于其他乘客的优先级。

另一个现实中的例子是医院的(急诊科)候诊室。医生会优先处理病情比较严重的患者。通常，护士会鉴别分类，根据患者病情的严重程度放号。

实现一个优先队列，有两种选项：设置优先级，然后在正确的位置添加元素；或者用入列操作添加元素，然后按照优先级移除它们。在下面示例中，我们将会在正确的位置添加元素，因此可以对它们使用默认的出列操作：

```
class PriorityQueue {
    constructor() {
        this.items = []
    }

    enqueue(element, priority){
        const queueElement = { element, priority }
        if (this.isEmpty) {
```

```
            this.items.push(queueElement)
        } else {
            const preIndex = this.items.findIndex((item) => queueElement.priority < item.priority)
            if (preIndex > -1) {
                this.items.splice(preIndex, 0, queueElement)
            } else {
                this.items.push(queueElement)
            }
        }
    }

    dequeue(){
        return this.items.shift()
    }

    front(){
        return this.items[0]
    }

    clear(){
        this.items = []
    }

    get size(){
        return this.items.length
    }

    get isEmpty(){
        return !this.items.length
    }

    print() {
        console.log(this.items)
    }
```

```
}
```

优先队列的使用：

```
const priorityQueue = new PriorityQueue()

priorityQueue.enqueue('John', 2)
priorityQueue.enqueue('Jack', 1)
priorityQueue.enqueue('Camila', 1)
priorityQueue.enqueue('Surmon', 3)
priorityQueue.enqueue('skyRover', 2)
priorityQueue.enqueue('司马萌', 1)
priorityQueue.print()

console.log(priorityQueue.isEmpty, priorityQueue.size) // false 6
```

循环队列

为充分利用向量空间，克服"假溢出"现象的方法是：将向量空间想象为一个首尾相接的圆环，并称这种向量为循环向量。存储在其中的队列称为循环队列（Circular Queue）。这种循环队列可以以单链表、队列的方式来在实际编程应用中来实现。

下面我们基于首次实现的队列类，简单实现一个循环引用的示例：

```
class LoopQueue extends Queue {

    constructor (items) {
        super (items)
    }

    getIndex(index) {
        const length = this .items.length
```

```
        return index > length ? (index % length) : index
    }

    find(index) {
        return !this.isEmpty ? this.items[this.getIndex(index)] : null
    }
}
```

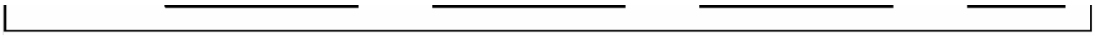
访问一个循环队列：

```
const loopQueue = new LoopQueue(['Surmon'])

loopQueue.enqueue('SkyRover')
loopQueue.enqueue('Even')
loopQueue.enqueue('Alice')
console.log(loopQueue.size, loopQueue.isEmpty) // 4 false

console.log(loopQueue.find(26)) // 'Evan'
console.log(loopQueue.find(87651)) // 'Alice'
```

链表

要存储多个元素，数组（或列表）可能是最常用的数据结构。 每种语言都实现了数组。这种数据结构非常方便，提供了一个便利的 `[]` 语法来访问它的元素。 然而，这种数据结构有一个缺点：在大多数语言中，数组的大小是固定的，从数组的起点或中间插入或移除项的成本很高，因为需要移动元素； 尽管 JavaScript 中的 `Array` 类方法可以帮我们做这些事，但背后的处理机制同样如此。

链表存储有序的元素集合，但不同于数组，链表中的元素在内存中并不是连续放置的。每个 元素由一个存储元素本身的节点和一个指向下一个元素的引用(也称指针或链接)组成。下图展示了链表的结构:



相对于传统的数组，链表的一个好处在于，添加或移除元素的时候不需要移动其他元素。然而，链表需要使用指针，因此实现链表时需要额外注意。

数组的另一个细节是可以直接访问任何位置的任何元素，而要想访问链表中间的一个元素，需要从起点(表头)开始迭代列表直到找到所需的元素。

现实中有许多链表的例子：一列火车是由一系列车厢/车皮组成的，每节车厢/车皮都相互连接，你很容易分离一节车皮，改变它的位置，添加或移除它。下图演示了一列火车，每节车皮都是列表的元素，车皮间的连接就是指针：

下面我们使用 Javascript 创建一个链表类：

```
// 链表节点
class Node {
    constructor ( element ) {
        this .element = element
        this .next = null
    }
}

// 链表
class LinkedList {

    constructor () {
        this .head = null
        this .length = 0
    }

    // 追加元素
    append(element) {
        const node = new Node(element)
        let current = null
        if ( this .head === null ) {
            this .head = node
```

```
        } else {
            current = this.head
            while (current.next) {
                current = current.next
            }
            current.next = node
        }
        this.length++
    }

    // 任意位置插入元素
    insert(position, element) {
        if (position >= 0 && position <= this.length) {
            const node = new Node(element)
            let current = this.head
            let previous = null
            let index = 0
            if (position === 0) {
                this.head = node
            } else {
                while (index++ < position) {
                    previous = current
                    current = current.next
                }
                node.next = current
                previous.next = node
            }
            this.length++
            return true
        }
        return false
    }

    // 移除指定位置元素
    removeAt(position) {
```

```
        // 检查越界值
        if (position > -1 && position < length) {
            let current = this.head
            let previous = null
            let index = 0
            if (position === 0) {
                this.head = current.next
            } else {
                while (index++ < position) {
                    previous = current
                    current = current.next
                }
                previous.next = current.next
            }
            this.length--
            return current.element
        }
        return null
    }

    // 寻找元素下标
    findIndex(element) {
        let current = this.head
        let index = -1
        while (current) {
            if (element === current.element) {
                return index + 1
            }
            index++
            current = current.next
        }
        return -1
    }
```

```
    // 删除指定文档
    remove(element) {
        const index = this.indexOf(element)
        return this.removeAt(index)
    }

    isEmpty() {
        return !this.length
    }

    size() {
        return this.length
    }

    // 转为字符串
    toString() {
        let current = this.head
        let string = ''
        while (current) {
            string += ` ${current.element}`
            current = current.next
        }
        return string
    }
}
```

链表类的使用：

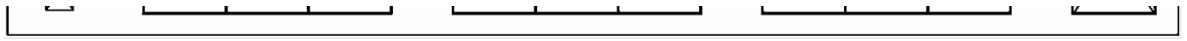
```
const linkedList = new LinkedList()

console.log(linkedList)
linkedList.append(2)
linkedList.append(6)
```

```
linkedList. append ( 24 )
linkedList. append ( 152 )

linkedList.insert( 3 , 18 )
console.log(linkedList)
console.log(linkedList.findIndex( 24 ))
```

双向链表

链表有多种不同的类型，这一节介绍双向链表。双向链表和普通链表的区别在于，在链表中， 一个节点只有链向下一个节点的链接，而在双向链表中，链接是双向的:一个链向下一个元素， 另一个链向前一个元素，如下图所示:

双向链表提供了两种迭代列表的方法:从头到尾，或者反过来。我们也可以访问一个特定节 点的下一个或前一个元素。在单向链表中，如果迭代列表时错过了要找的元素，就需要回到列表 起点，重新开始迭代。这是双向链表的一个优点。



我们继续来实现一个双向链表类：

```
// 链表节点

class Node {
    constructor ( element ) {
        this .element = element
        this .prev = null
        this .next = null
    }
}

// 双向链表
class DoublyLinkedList {

    constructor () {
        this .head = null
        this .tail = null
```

```
        this.length = 0
    }

    // 任意位置插入元素
    insert(position, element) {
        if (position >= 0 && position <= this.length){
            const node = new Node(element)
            let current = this.head
            let previous = null
            let index = 0
            // 首位
            if (position === 0 ) {
                if (!head){
                    this.head = node
                    this.tail = node
                } else {
                    node.next = current
                    this.head = node
                    current.prev = node
                }
            // 末位
            } else if (position === this.length) {
                current = this.tail
                current.next = node
                node.prev = current
                this.tail = node
            // 中位
            } else {
                while (index++ < position) {
                    previous = current
                    current = current.next
                }
                node.next = current
                previous.next = node
                current.prev = node
```

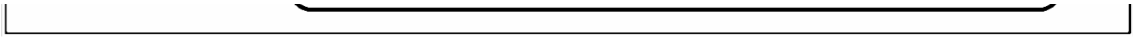
```
            node.prev = previous
        }
        this.length++
        return true
    }
    return false
}

    // 移除指定位置元素
    removeAt(position) {
        if (position > -1 && position < this.length) {
            let current = this.head
            let previous = null
            let index = 0

            // 首位
            if (position === 0) {
                this.head = this.head.next
                this.head.prev = null
                if (this.length === 1) {
                    this.tail = null
                }

            // 末位
            } else if (position === this.length - 1) {
                this.tail = this.tail.prev
                this.tail.next = null

            // 中位
            } else {
                while (index++ < position) {
                    previous = current
                    current = current.next
                }
                previous.next = current.next
```

```
                current.next.prev = previous
            }
            this.length--
            return current.element
        } else {
            return null
        }
    }

    // 其他方法...
}
```

循环链表

循环链表可以像链表一样只有单向引用，也可以像双向链表一样有双向引用。循环链表和链表之间唯一的区别在于，最后一个元素指向下一个元素的指针(tail.next)不是引用null， 而是指向第一个元素(head)，如下图所示。

双向循环链表有指向 head 元素的 tail.next，和指向 tail 元素的 head.prev。



链表相比数组最重要的优点，那就是无需移动链表中的元素，就能轻松地添加和移除元素。因此，当你需要添加和移除很多元素 时，最好的选择就是链表，而非数组