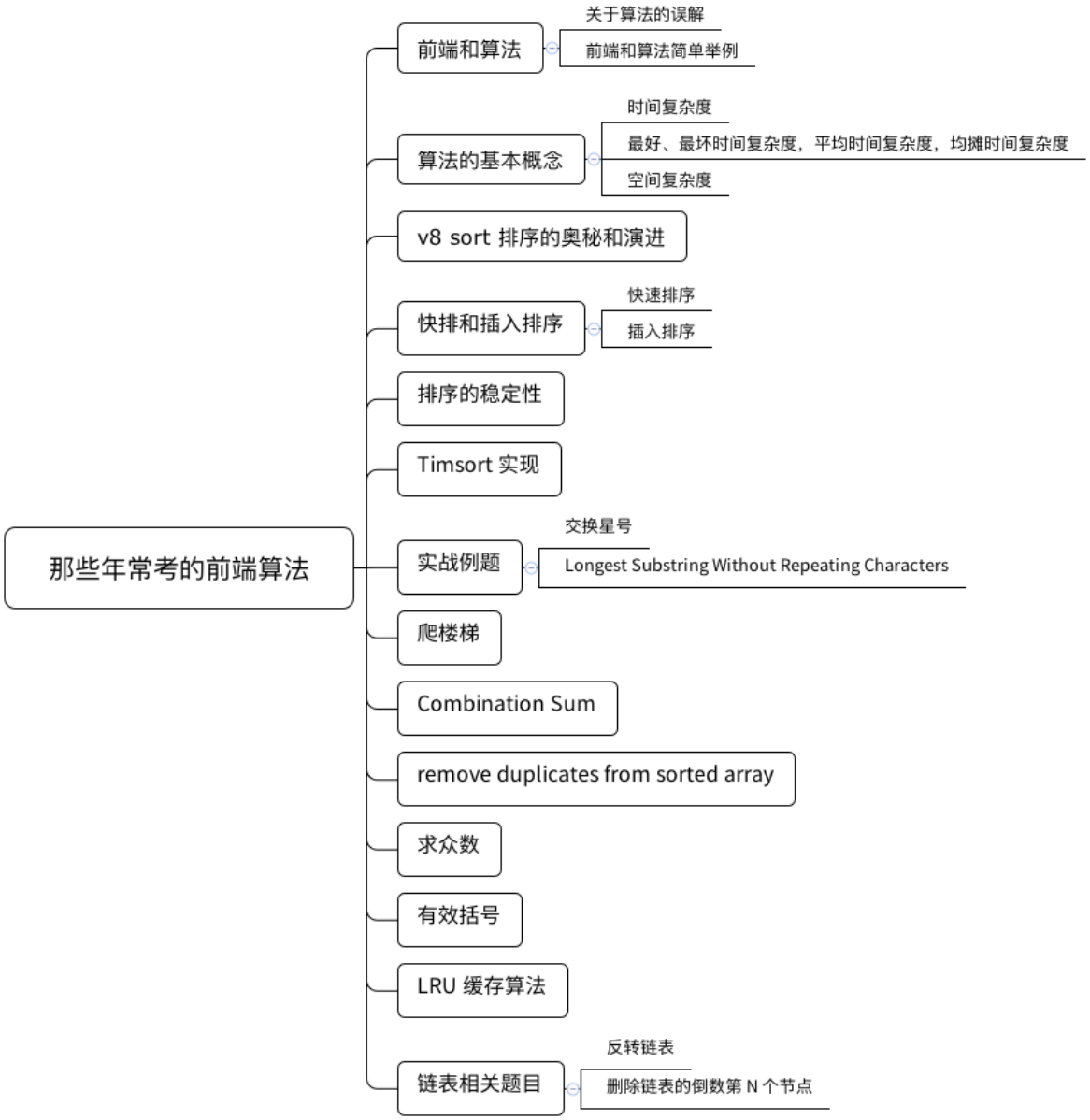
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

那些年常考的前端算法

上一讲我们剖析了算法的一些基本概念。这一讲将围绕 v8 引申出的算法进行分析，同时做一些常见、典型考题。主要内容如下：



v8 sort 排序的奥秘和演进

前一讲，我带大家分析了「如何将一个 JavaScript 数组打乱顺序？」，其中提到了 sort 这个 API，具体有这样的一段描述：

v8 在处理 sort 方法时，使用了插入排序和快排两种方案。当目标数组长度小于 10（不同版本有差别）时，使用插入排序；反之，使用快排。

如果细心的读者可能会到 v8 源代码中找寻相关的算法逻辑，那么你一定会大失所望。因为根本找不到 10 这样的常量，更没有插入排序和快排两种方案的切换，甚至连实现的预言都不是 JavaScript 或者 C++，这是为什么呢?

原来，在新的 v8 版本中（具体 V8 6.9）已经使用了一种名叫 Torque 的开发语言重构，并在 7.0 改进了 sort 算法。也就是说，现在社区上几乎所有的 V8 排序源码分析都已经过时了。

Torque 是 v8 团队专门为了开发 v8 引擎而开发的语言，他的后缀名是 tq。作为一种高级语言，Torque 依靠 CodeStubAssembler 编译器来转换为汇编代码。

在新的版本中，v8 也采用了一种名叫 Timsort 的全新算法，这套算法最开始于 2002 被 Tim Peters 在 Python 语言中使用。

从这个演进过程中，我们分为三大块来看。

快排和插入排序

排序算法多种多样，社区上的分析也比较多。这里我们挑选 v8 sort 排序中「露脸」的快速排序和插入排序进行讲解。

不知道读者是否有这样的困扰：我们看一遍算法，理解了，可是过两天又完全记不得具体讲了什么。针对于此，我们应该结合算法的特点，加以应用，才能深入记忆。排序算法同样如此，对于每一种算法，我们应该先记住其思想，再记住其实现。不过要知道：「排序没有想象中那么简单」。

快速排序

快速排序的特点就是分治。如何体现分治策略呢？我们首先在数组中选取一个基准点，叫做 pivot，根据这个基准点：把比基准点小的数组值放在基准点左边，把比基准点大的数组值放在基准点右边。这样一来，基于基准点，左边分区的值都小于基准点，右边分区的值都大于基准点，然后针对左边分区和右边分区进行同样的操作，直到最后排序完成。

最简单的实现：

```
const quickSort = array => {
 if (array.length < 2) {
   return array.slice()
 }

 // 随机找到 pivot
 let pivot = array[Math.floor(Math.random() *
array.length)]

 let left = []
 let middle = []
 let right = []

 for (let i = 0; i < array.length; i++) {
   var value = array[i]
   if (value < pivot) {
     left.push(value)
   }

   if (value === pivot) {
     middle.push(value)
   }

   if (value > pivot) {
     right.push(value)
   }
 }

 // 递归进行
```

```
 return quickSort(left).concat(middle, quickSort(right))
}
```

这种实现方法有不少优化点，其中之一就是我们可以在原数组上进行操作，而不产生一个新的数组：

```
const quickSort = (array, start, end) => {
 start = start === undefined ? 0 : start
 end = end === undefined ? arr.length - 1 : end;

 if (start >= end) {
   return
 }

 let value = array[start]

 let i = start
 let j = end

 while (i < j) {
   // 找出右边第一个小于参照数的下标并记录
   while (i < j && array[j] >= value) {
     j--
   }

   if (i < j) {
     arr[i++] = arr[j]
   }

   // 找出左边第一个大于参照数的下标，并记录
   while (i < j && array[i] < value) {
     i++
   }

   if (i < j) {
     arr[j--] = arr[i]
   }
```

```
 }

 arr[i] = value

 quickSort(array, start, i - 1)
 quickSort(array, i + 1, end)
}
```

调用方式：

```
let arr = [0, 12, 43, 45, 88, 1, 69]
quickSort(arr, 0, arr.length - 1)
console.log(arr)
```

我们该如何理解 in place 的快排算法呢？

首先使用双指针，指针开始遍历，当右边发现一个小于参照数（即 array[start]）的时候，就将该值赋值给起始位置。赋值完之后，那么右边这个位置就空闲了。这时在左边发现比参照数大的值时，就赋值给这个刚刚空闲出来的右边位置。以此类推，直到 i 不再小于 j。经过这一轮操作之后，所有比参照数小的都到了数组的左边，所有比参照数大的都到了数组右边，而数组中间被赋值为参照数。

我们再来分析另外一个优化点。之前的课程中提到了尾递归调用优化，那么上面的快排能否使用尾递归进行优化呢？

我们进行观察，上面的实现最后两行：

```
quickSort(array, start, i - 1)
quickSort(array, i + 1, end)
```

如果能形成以下的形式：

```
return quickSort()
```

那么就实现了尾递归调用优化。为此，我们需要一个 stack 来进行参数信息的传递：

```
const quickSort = (array, stack) => {
let start = stack[0]
let end = stack[1]

let value = array[start]

let i = start
let j = end

while (i < j) {
  while (i < j && array[j] >= value) {
    j--
  }
  if (i < j) {
    array[i++] = array[j]
  }

  while (i < j && array[i] < value) {
    i++
  }

  if (i < j) {
    array[j--] = array[i]
  }
}

arr[i] = value

// 移除已经使用完的下标
stack.shift()
stack.shift()

// 存入新的下标
if (i + 1 < end) {
```

```
    stack.unshift(i + 1, end)
  }
  if (start < i - 1) {
    stack.unshift(start, i - 1)
  }

  if (stack.length == 0) {
    return;
  }

  return quickSort(array, stack)
}
```

最后，关于快速排序的优化点还有一个最重要的方向就是对 pivot 元素的选取。通过上面的分析，我们发现快速排序的算法核心在于选择一个 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。

那么试想，如果我们对一个已经有序的数组进行排序，恰好每次选择 pivot 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 $O(n^2)$。因此 pivot 的选择非常重要。

在早期 v8 使用快速排序时，采用了三数取中（median-of-three）的 pivot 优化方案：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。具体 v8 源代码为：

```
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);
        }
    }
    ......
};
```

由此看出，这所谓的第三个竞争元素产生方式为：

> 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
>
> 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）左右的值，去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

三数取中（median-of-three）当中，最后选取的是三个元素的中位值作为 pivot。

插入排序

插入排序我们还是从特点入手：它先将待排序序列的第一个元素看做一个有序序列，当然了，就一个元素，那么它一定是有序的；而把第二个元素到最后一个元

素当成是未排序序列；对于未排序的序列进行遍历，将扫描到的每个元素插入有序序列的适当位置，保证有序序列依然有序，那么直到所有数据都完成，我们也就完成了排序。

如果待插入的元素与有序序列中的某个元素相等，那么我们统一先将待插入元素插入到相等元素的后面。

我们的实现：

```
const insertsSort = array => {
   const length = arr.length
   let preIndex
   let current

   for (let i = 1; i < length; i++) {
       preIndex = i - 1
       current = array[i]

       while (preIndex >= 0 && array[preIndex] > current)
{
           array[preIndex + 1] = array[preIndex]
           preIndex--
       }

       array[preIndex + 1] = current
   }
   return array
}
```

那么上述实现的插入排序有优化空间吗?

这是一定的，优化空间主要有这么几个方向：

在遍历未排序序列时，将当前元素插入到有序序列过程中，可以使用二分法减少查找次数（因为是向有序序列插入）

使用链表，将有序数组转为链表这种数据结构，那么插入操作的时间复杂度为 O(1)，查找复杂度变为 O(n)

使用排序二叉树，将有序数组转为排序二叉树结构，然后中序遍历该二叉树，不过这种方式需要额外空间。

采用二分法的优化实现：

```
const insertSort = array => array.reduce(insert, [])

const insert = (sortedArray, value) => {
 const length = sortedArray.length

 if (length === 0) {
   sortedArray.push(value)
   return sortedArray
 }

 let i = 0
 let j = length
 let mid

 // 先判断是否为极端值
 if (value < sortedArray[i]) {
   // 直接插入到数组的最头
   return sortedArray.unshift(value), sortedArray
 }
 if (value >= sortedArray[length - 1]) {
     // 直接插入到数组的最尾
   return sortedArray.push(value), sortedArray
 }

 // 开始二分查找
 while (i < j) {
   mid = ((j + i) / 2) | 0

   if (i == mid) {
```

```
      break
    }

    if (value  < sortedArray[mid]) {
      j = mid
    }

    if (value === sortedArray[mid]) {
      i = mid
      break
    }

    if (value > sortedArray[mid]) {
      i = mid
    }
  }

  let midArray = [value]
  let lastArray = sortedArray.slice(i + 1)

  sortedArray = sortedArray
    .slice(0, i + 1)
    .concat(midArray)
    .concat(lastArray)

  return sortedArray
}
```

到此我们介绍完了两种排序方法。事实上，光排序就是一门很深的学问，也涉及到了算法和数据结构的方方面面，我们将继续通过排序，了解更多算法内容。

排序的稳定性

事实上，除了 v8 引擎，其他引擎也有不同的 sort 排序规则。比如 SpiderMoney 早期内部实现了归并排序，Chakra 的数组排序算法实现的也是快速排序。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，Mozilla 开发组内部针对这个问题进行了一系列讨论之后，Firefox3 将归并排序作为了数组排序的新实现。

我们知道，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。什么是排序的稳定性呢?

简单说，就是能保证排序前 2 个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。形式化一下，如果 array[i] = array[j]，array[i] 原来在位置前，排序后 array[i] 还是要在 array[j] 位置前。

在很多情况下，不稳定的排序也不会造成影响。但是在一些场景中，可能就会「有毒」。比如对于一个数组对象，场景是：

某市的机动车牌照拍卖系统，最终中标的规则为：按价格进行倒排序；相同价格则按照竞标顺位（即价格提交时间）进行正排序。

如果采用不稳定排序，那么结果就有可能不符合预期。

那么如果一些浏览器引擎实现的排序采用了不稳定排序算法应该怎么办呢？方案：

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。

示例代码：

```
const HELPER = Symbol('helper')
```

```
const getComparer = compare =>
  (left, right) => {
      let result = compare(left, right)

      return result === 0 ? left[HELPER] - right[HELPER]
: result
   }

const sort = (array, compare) => {
   array = array.map(
      (item, index) => {
          if (typeof item === 'object') {
              item[HELPER] = index
          }

          return item
      }
   );

   return array.sort(getComparer(compare))
}
```

近些年来，随着浏览器计算能力的进一步提升，项目正在往富客户端应用方向转变，前端在项目中扮演的角色也越来越重要。算法意识是一个不得忽视的话题。

Timsort 实现

好了，我们再把话题收回来。那么 v8 采用的 Timsort 算法到底是什么呢？Timsort 结合了归并排序和插入排序，效率更高。Pyhton 自从 2.3 版，Java SE7 和 Android 以来也一直采用 Timsort 算法排序。

我们看一下 JSE7 中对 Timsort 的描述：

A stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when running on partially sorted arrays, while offering performance comparable to a traditional mergesort when run on random arrays. Like all proper mergesorts, this sort is stable and runs O(n log n) time (worst case). In the worst case, this sort requires temporary storage space for n/2 object references; in the best case, it requires only a small constant amount of space.

Timsort 是稳定且自适应的算法。如果需要排序的数组中存在部分已经排序好的区间，它的时间复杂度会小于 nlogn，它的最坏时间复杂度是 O(nlogn）。在最坏情况下，Timsort 算法需要的临时空间是 n/2，在最好情况下，它只需要一个很小的常量存储空间。

Timsort 算法为了减少对升序部分的回溯和对降序部分的性能倒退，将输入按其升序和降序特点进行了分区。

那么具体的过程：排序输入的单位不是一个个单独的数字，而是一个个分区。其中每一个分区叫一个 run。针对这些 run 序列，每次拿一个 run 出来按规则进行合并。每次合并会将两个 run 合并成一个 run。合并的结果保存到栈中。合并直到消耗掉所有的 run，这时将栈上剩余的 run 合并到只剩一个 run 为止。这时这个仅剩的 run 便是排好序的结果。

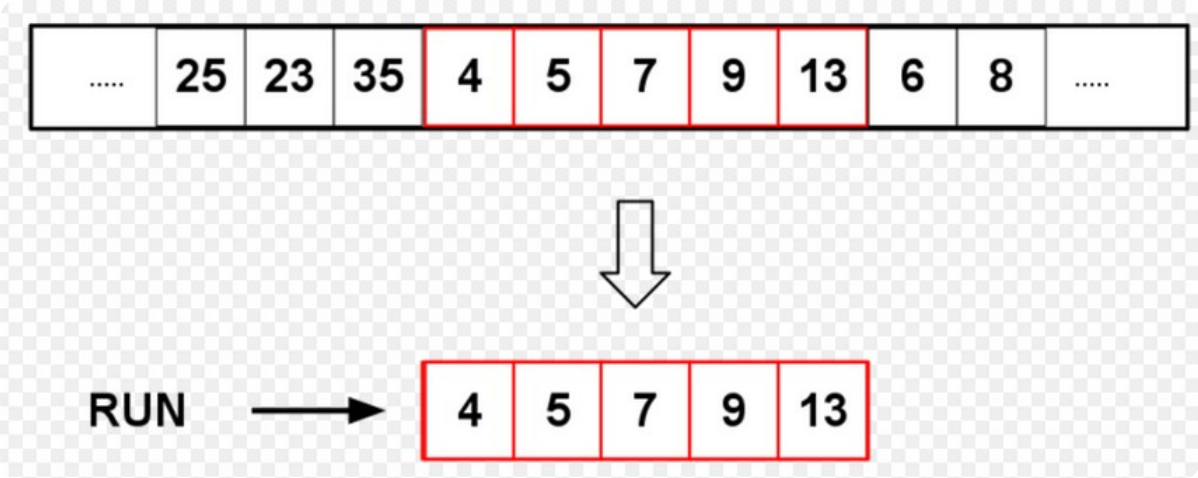
这样一来 Timsort 的具体实施规则就是：

如果数组长度小于某个值，直接用二分插入排序算法

找到各个 run，并入栈

按规则合并 run

理解 run 将会是关键，请看图：



具体实现我参考部分 timsort 内容：

```
Array.prototype.timsort = function(comp) {
   var global_a = this
   var MIN_MERGE = 32;
   var MIN_GALLOP = 7
   var runBase = [];
   var runLen = [];
   var stackSize = 0;
   var compare = comp;
   sort(this, 0, this.length, compare);

   function sort(a, lo, hi, compare) {
       if (typeof compare != "function") {
           throw new Error("Compare is not a function.");
           return
       }
       stackSize = 0;
       runBase = [];
       runLen = [];
       rangeCheck(a.length, lo, hi);
       var nRemaining = hi - lo;
       if (nRemaining < 2) return;
       if (nRemaining < MIN_MERGE) {
           var initRunLen = countRunAndMakeAscending(a,
lo, hi, compare);
           binarySort(a, lo, hi, lo + initRunLen,
```

```
compare);
            return
        }
        var ts = [];
        var minRun = minRunLength(nRemaining);
        do {
            var runLenVar = countRunAndMakeAscending(a, lo,
hi, compare);
            if (runLenVar < minRun) {
                var force = nRemaining <= minRun ?
nRemaining : minRun;
                binarySort(a, lo, lo + force, lo +
runLenVar, compare);
                runLenVar = force
            }
            pushRun(lo, runLenVar);
            mergeCollapse();
            lo += runLenVar;
            nRemaining -= runLenVar
        } while (nRemaining != 0);
        mergeForceCollapse()
    }

    function binarySort(a, lo, hi, start, compare) {
        if (start == lo) start++;
        for (; start < hi; start++) {
            var pivot = a[start];
            var left = lo;
            var right = start;
            while (left < right) {
                var mid = (left + right) >>> 1;
                if (compare(pivot, a[mid]) < 0) right =
mid;
                else left = mid + 1
            }
            var n = start - left;
            switch (n) {
```

```
                case 2:
                    a[left + 2] = a[left + 1];
                case 1:
                    a[left + 1] = a[left];
                    break;
                default:
                    arraycopy(a, left, a, left + 1, n)
            }
            a[left] = pivot
        }
    }

    function countRunAndMakeAscending(a, lo, hi, compare) {
        var runHi = lo + 1;
        if (compare(a[runHi++], a[lo]) < 0) {
            while (runHi < hi && compare(a[runHi], a[runHi
- 1]) < 0) {
                runHi++
            }
            reverseRange(a, lo, runHi)
        } else {
            while (runHi < hi && compare(a[runHi], a[runHi
- 1]) >= 0) {
                runHi++
            }
        }
        return runHi - lo
    }

    function reverseRange(a, lo, hi) {
        hi--;
        while (lo < hi) {
            var t = a[lo];
            a[lo++] = a[hi];
            a[hi--] = t
        }
    }
```

```
    function minRunLength(n) {
        var r = 0;
        return n + 1
    }

    function pushRun(runBaseArg, runLenArg) {
        runBase[stackSize] = runBaseArg;
        runLen[stackSize] = runLenArg;
        stackSize++
    }

    function mergeCollapse() {
        while (stackSize > 1) {
            var n = stackSize - 2;
            if (n > 0 && runLen[n - 1] <= runLen[n] +
runLen[n + 1]) {
                if (runLen[n - 1] < runLen[n + 1]) n--;
                mergeAt(n)
            } else if (runLen[n] <= runLen[n + 1]) {
                mergeAt(n)
            } else {
                break
            }
        }
    }

    function mergeForceCollapse() {
        while (stackSize > 1) {
            var n = stackSize - 2;
            if (n > 0 && runLen[n - 1] < runLen[n + 1]) n-
-;
            mergeAt(n)
        }
    }

    function mergeAt(i) {
```

```
        var base1 = runBase[i];
        var len1 = runLen[i];
        var base2 = runBase[i + 1];
        var len2 = runLen[i + 1];
        runLen[i] = len1 + len2;
        if (i == stackSize - 3) {
            runBase[i + 1] = runBase[i + 2];
            runLen[i + 1] = runLen[i + 2]
        }
        stackSize--;
        var k = gallopRight(global_a[base2], global_a,
base1, len1, 0, compare);
        base1 += k;
        len1 -= k;
        if (len1 == 0) return;
        len2 = gallopLeft(global_a[base1 + len1 - 1],
global_a, base2, len2, len2 - 1, compare);
        if (len2 == 0) return;
        if (len1 <= len2) mergeLo(base1, len1, base2,
len2);
        else mergeHi(base1, len1, base2, len2)
    }

    function gallopLeft(key, a, base, len, hint, compare) {
        var lastOfs = 0;
        var ofs = 1;
        if (compare(key, a[base + hint]) > 0) {
            var maxOfs = len - hint;
            while (ofs < maxOfs && compare(key, a[base +
hint + ofs]) > 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0) ofs = maxOfs
            }
            if (ofs > maxOfs) ofs = maxOfs;
            lastOfs += hint;
            ofs += hint
```

```
        } else {
            var maxOfs = hint + 1;
            while (ofs < maxOfs && compare(key, a[base +
hint - ofs]) <= 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0) ofs = maxOfs
            }
            if (ofs > maxOfs) ofs = maxOfs;
            var tmp = lastOfs;
            lastOfs = hint - ofs;
            ofs = hint - tmp
        }
        lastOfs++;
        while (lastOfs < ofs) {
            var m = lastOfs + ((ofs - lastOfs) >>> 1);
            if (compare(key, a[base + m]) > 0) lastOfs = m
+ 1;
            else ofs = m
        }
        return ofs
    }

    function gallopRight(key, a, base, len, hint, compare)
{
        var ofs = 1;
        var lastOfs = 0;
        if (compare(key, a[base + hint]) < 0) {
            var maxOfs = hint + 1;
            while (ofs < maxOfs && compare(key, a[base +
hint - ofs]) < 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0) ofs = maxOfs
            }
            if (ofs > maxOfs) ofs = maxOfs;
            var tmp = lastOfs;
```

```
            lastOfs = hint - ofs;
            ofs = hint - tmp
        } else {
            var maxOfs = len - hint;
            while (ofs < maxOfs && compare(key, a[base +
hint + ofs]) >= 0) {
                lastOfs = ofs;
                ofs = (ofs << 1) + 1;
                if (ofs <= 0) ofs = maxOfs
            }
            if (ofs > maxOfs) ofs = maxOfs;
            lastOfs += hint;
            ofs += hint
        }
        lastOfs++;
        while (lastOfs < ofs) {
            var m = lastOfs + ((ofs - lastOfs) >>> 1);
            if (compare(key, a[base + m]) < 0) ofs = m;
            else lastOfs = m + 1
        }
        return ofs
    }

    function mergeLo(base1, len1, base2, len2) {
        var a = global_a;
        var tmp = a.slice(base1, base1 + len1);
        var cursor1 = 0;
        var cursor2 = base2;
        var dest = base1;
        a[dest++] = a[cursor2++];
        if (--len2 == 0) {
            arraycopy(tmp, cursor1, a, dest, len1);
            return
        }
        if (len1 == 1) {
            arraycopy(a, cursor2, a, dest, len2);
            a[dest + len2] = tmp[cursor1];
```

```
            return
        }
        var c = compare;
        var minGallop = MIN_GALLOP;
        outer: while (true) {
            var count1 = 0;
            var count2 = 0;
            do {
                if (compare(a[cursor2], tmp[cursor1]) < 0)
{
                    a[dest++] = a[cursor2++];
                    count2++;
                    count1 = 0;
                    if (--len2 == 0) break outer
                } else {
                    a[dest++] = tmp[cursor1++];
                    count1++;
                    count2 = 0;
                    if (--len1 == 1) break outer
                }
            } while ((count1 | count2) < minGallop);
            do {
                count1 = gallopRight(a[cursor2], tmp,
cursor1, len1, 0, c);
                if (count1 != 0) {
                    arraycopy(tmp, cursor1, a, dest,
count1);
                    dest += count1;
                    cursor1 += count1;
                    len1 -= count1;
                    if (len1 <= 1) break outer
                }
                a[dest++] = a[cursor2++];
                if (--len2 == 0) break outer;
                count2 = gallopLeft(tmp[cursor1], a,
cursor2, len2, 0, c);
                if (count2 != 0) {
```

```
                    arraycopy(a, cursor2, a, dest, count2);
                    dest += count2;
                    cursor2 += count2;
                    len2 -= count2;
                    if (len2 == 0) break outer
                }
                a[dest++] = tmp[cursor1++];
                if (--len1 == 1) break outer;
                minGallop--
            } while (count1 >= MIN_GALLOP | count2 >=
MIN_GALLOP);
            if (minGallop < 0) minGallop = 0;
            minGallop += 2
        }
        this.minGallop = minGallop < 1 ? 1 : minGallop;
        if (len1 == 1) {
            arraycopy(a, cursor2, a, dest, len2);
            a[dest + len2] = tmp[cursor1]
        } else if (len1 == 0) {
            throw new Error("IllegalArgumentException.
Comparison method violates its general contract!");
        } else {
            arraycopy(tmp, cursor1, a, dest, len1)
        }
    }

    function mergeHi(base1, len1, base2, len2) {
        var a = global_a;
        var tmp = a.slice(base2, base2 + len2);
        var cursor1 = base1 + len1 - 1;
        var cursor2 = len2 - 1;
        var dest = base2 + len2 - 1;
        a[dest--] = a[cursor1--];
        if (--len1 == 0) {
            arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
            return
        }
```

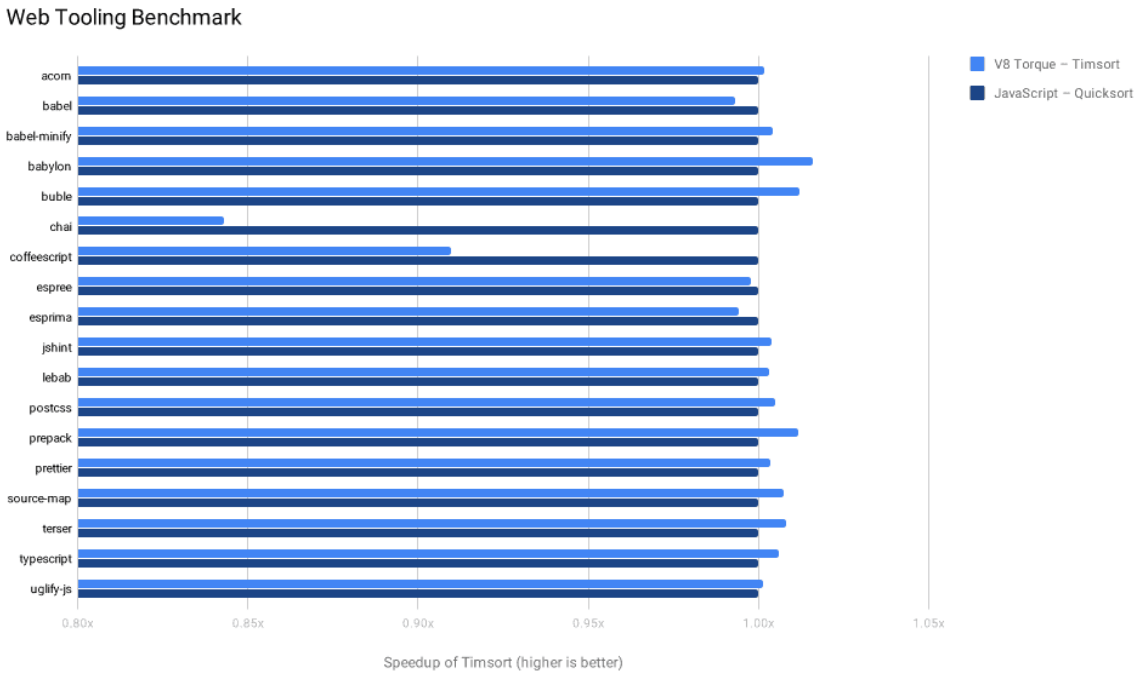
```
        if (len2 == 1) {
            dest -= len1;
            cursor1 -= len1;
            arraycopy(a, cursor1 + 1, a, dest + 1, len1);
            a[dest] = tmp[cursor2];
            return
        }
        var c = compare;
        var minGallop = MIN_GALLOP;
        outer: while (true) {
            var count1 = 0;
            var count2 = 0;
            do {
                if (compare(tmp[cursor2], a[cursor1]) < 0)
{
                    a[dest--] = a[cursor1--];
                    count1++;
                    count2 = 0;
                    if (--len1 == 0) break outer
                } else {
                    a[dest--] = tmp[cursor2--];
                    count2++;
                    count1 = 0;
                    if (--len2 == 1) break outer
                }
            } while ((count1 | count2) < minGallop);
            do {
                count1 = len1 - gallopRight(tmp[cursor2],
a, base1, len1, len1 - 1, c);
                if (count1 != 0) {
                    dest -= count1;
                    cursor1 -= count1;
                    len1 -= count1;
                    arraycopy(a, cursor1 + 1, a, dest + 1,
count1);
                    if (len1 == 0) break outer
                }
```

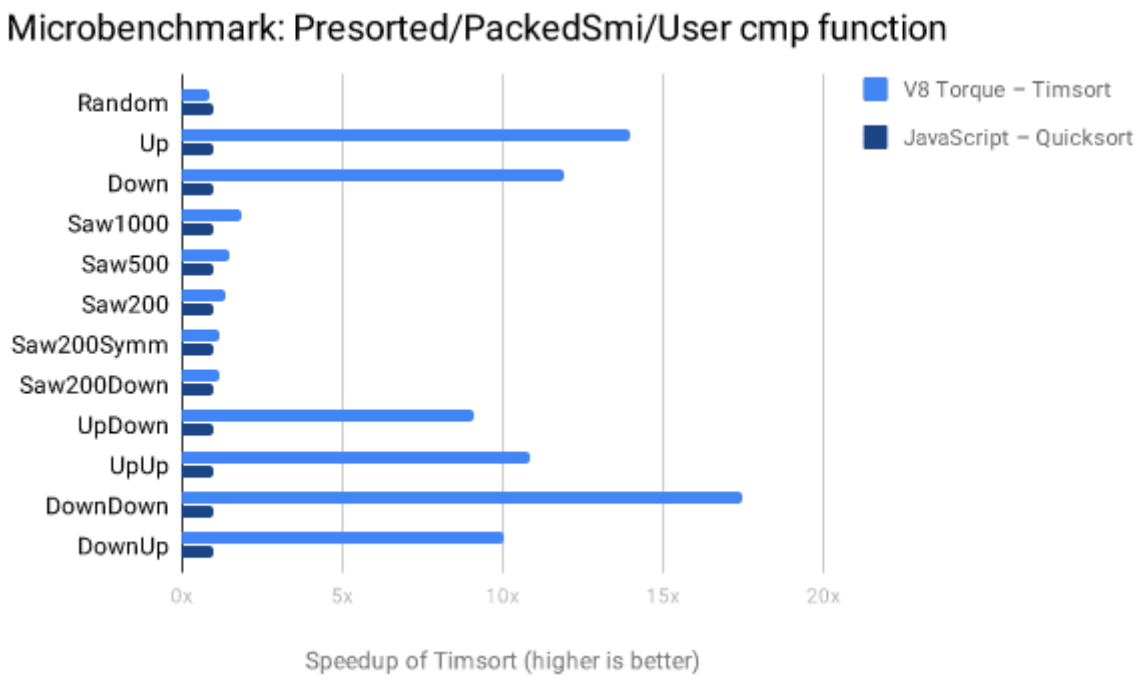
```
                a[dest--] = tmp[cursor2--];
                if (--len2 == 1) break outer;
                count2 = len2 - gallopLeft(a[cursor1], tmp,
0, len2, len2 - 1, c);
                if (count2 != 0) {
                    dest -= count2;
                    cursor2 -= count2;
                    len2 -= count2;
                    arraycopy(tmp, cursor2 + 1, a, dest +
1, count2);
                    if (len2 <= 1) break outer
                }
                a[dest--] = a[cursor1--];
                if (--len1 == 0) break outer;
                minGallop--
            } while (count1 >= MIN_GALLOP | count2 >=
MIN_GALLOP);
            if (minGallop < 0) minGallop = 0;
            minGallop += 2
        }
        this.minGallop = minGallop < 1 ? 1 : minGallop;
        if (len2 == 1) {
            dest -= len1;
            cursor1 -= len1;
            arraycopy(a, cursor1 + 1, a, dest + 1, len1);
            a[dest] = tmp[cursor2]
        } else if (len2 == 0) {
            throw new Error("IllegalArgumentException.
Comparison method violates its general contract!");
        } else {
            arraycopy(tmp, 0, a, dest - (len2 - 1), len2)
        }
    }

    function rangeCheck(arrayLen, fromIndex, toIndex) {
        if (fromIndex > toIndex) throw new
Error("IllegalArgument fromIndex(" + fromIndex + ") >
```

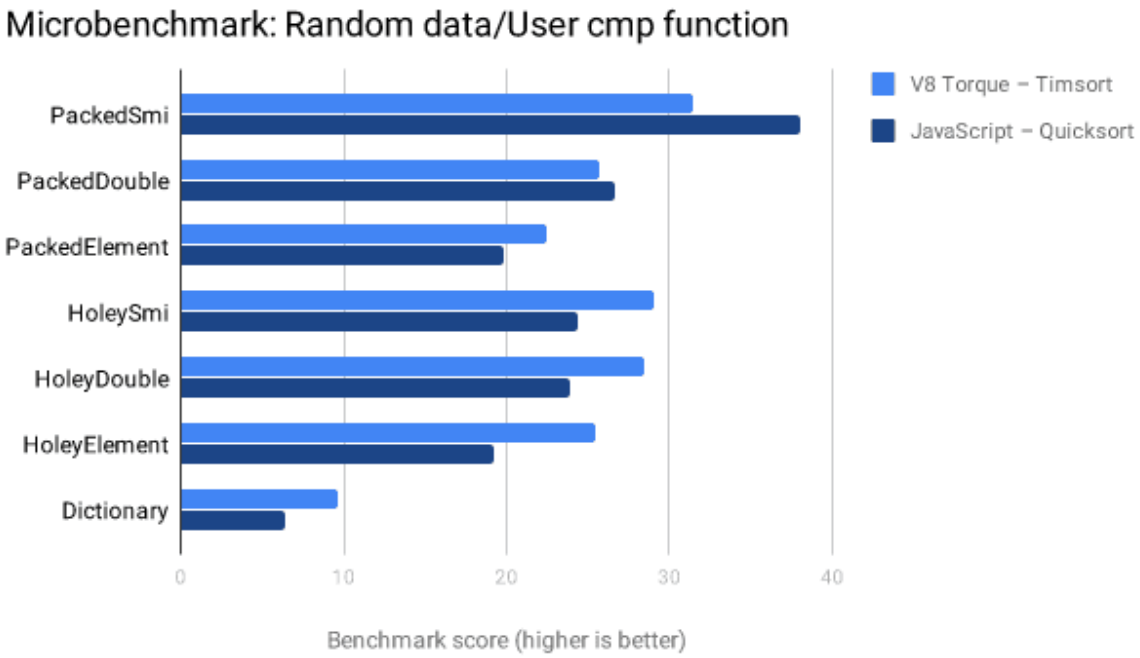
```
toIndex(" + toIndex + ")");
        if (fromIndex < 0) throw new
Error("ArrayIndexOutOfBounds " + fromIndex);
        if (toIndex > arrayLen) throw new
Error("ArrayIndexOutOfBounds " + toIndex);
    }
}
```

具体操作较为复杂，这里建议大家更多的了解这个知识点，而具体实现一般不作要求。

我们来看一下 v8 在采用 Timsort 之后，得到的一些 benchmark：







更多内容，可以参考 v8 官方博客：Getting things sorted in V8

实战例题

从这里开始，我们来「刷」一些实战例题。

交换星号

题目：一个字符串中只包含 *和数字，请把* 号都放开头。

思路：使用两个指针，从后往前扫字符串，遇到数字则赋值给后面的指针，继续往后扫，遇到 * 则不处理。

```
const isNumeric = n => !isNaN(parseFloat(n)) &&
isFinite(n);

/**
* @param {string}
* @return {string}
*/
const solution = s => {
   const n = s.length
   let a = s.split('')
   let j = n - 1

   for (let i = n - 1; i >= 0; --i)
       if (isNumeric(a[i])) a[j--] = a[i]

   for (; j >= 0; --j) a[j] = '*'
   return a.join('')
}
```

这样一来，我们逆序操作数组，遇见数字则向后置，遍历完一遍后，所有的数字都已经在后边了，同时把前边的数组项用 * 填充。

Longest Substring Without Repeating Characters

题意：给定一个字符串，返回它最长的不包含重复的子串长度。例如，输入 abcabcbb 输出 3（对应 abc）。

思路：

暴力枚举起点和终点，并判断重复，时间复杂度是 $O(n^2)$；

通过双指针、滑动窗口，动态维护窗口 [i..j)，使窗口内字符不重复。

我们看第二种思路解法，保证窗口 [i..j) 之间没有重复字符：

首先 i, j 两个指针均指向字符串头部，如果没有重复字符，则 j 不断向右滑动，直到出现重复字符；

如果出现了重复的字符，重复字符出现在第 str[j] 处，这时候开始移动指针 i，找到另一个重复的字读出现在 str[i] 处，那么能保证 [0, i] 以及 [i, j] 子字符串是不重复的，更新临时结果为 Math.max(result, j - i)。

时间复杂度 O(n)

实现：

```
const lengthOfLongestSubstring = str => {
    let result = 0
    let len = str.length

    // 记录当前区间内出现的字符
    let mapping = {}

    for (let i = 0, j = 0; ; ++i) {

        // j 右移的过程
        while (j < len && !mapping[str[j]])
            mapping[str[j++]] = true
        result = Math.max(result, j - i)

        if (j >= len)
            break;

        // 出现了重复字符，i 开始进行右移的过程，同时将移出的字符在 mapping 中重置
        while (str[i] != str[j])
            mapping[str[i++]] = false
        mapping[str[i]] = false
```

```
    }

    return result
};
```

举这个例子的目的是为了展示滑动窗口的思想，通过滑动窗口一般能实现 O(n) 的时间复杂度和 O(1) 的空间复杂度。

总结

这一讲我们主要介绍了几种排序算法和最先进的 Timsort，相信凭借这些内容，在前端排序上你可以「鄙视」面试官了。当然算法的坑还是很深的，我们要保持好的心态。最后部分介绍了两个算法题，算是抛砖引玉，下一讲，让我们针对算法面试，刷一刷算法。

点击查看下一节

那些年常考的前端算法（3）