



认识“分治”思想

本节我们要学习的两个排序算法都是对“分治”思想的应用。

“分治”，分而治之。其思想就是将一个大问题分解为若干个子问题，针对子问题分别求解后，再将子问题的解整合为大问题的解。

利用分治思想解决问题，我们一般分三步走：

- 分解子问题
- 求解每个子问题
- 合并子问题的解，得出大问题的解

下面我们一起来看看分治思想是如何帮助我们提升排序算法效率的。

归并排序

思路分析

归并排序是对分治思想的典型应用，它按照如下的思路对分治思想“三步走”的框架进行了填充：

- **分解子问题**：将需要被排序的数组从中间分割为两半，然后再将分割出来的每个子数组各分割为两半，重复以上操作，直到单个子数组只有一个元素为止。
- **求解每个子问题**：从粒度最小的子数组开始，两两合并、确保每次合并出来的数组都是有序的。（这里的“子问题”指的就是对每个子数组进行排序）。
- **合并子问题的解，得出大问题的解**：当数组被合并至原有的规模时，就得到了一个完全排序的数组

真实排序过程演示

下面我们基于归并排序的思路，尝试对以下数组进行排序：



首先重复地分割数组，整个分割过程如下：

首次分割，将数组整个对半分：

```
[8, 7, 6, 5, | 4, 3, 2, 1]
```

js

二次分割，将分割出的左右两个子数组各自对半分：

```
[8, 7, | 6, 5, | 4, 3, | 2, 1]
```

js

三次分割，四个子数组各自对半分后，每个子数组内都只有一个元素了：

```
[8, | 7, | 6, | 5, | 4, | 3, | 2, | 1]
```

js

接下来开始尝试解决每个子问题。将规模为1的子数组两两合并为规模为2的子数组，合并时确保有序，我们会得到这样的结果：

```
[7, 8, | 5, 6, | 3, 4, | 1, 2]
```

js

继续将规模为2的按照有序原则合并为规模为4的子数组：

```
[5, 6, 7, 8, | 1, 2, 3, 4]
```

js

最后将规模为4的子数组合并为规模为8的数组：

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

js

整个数组就完全有序了。

编码实现

通过上面的讲解，我们可以总结出归并排序中的两个主要动作：



这两个动作是紧密关联的，分割是将大数组反复分解为一个一个的原子项，合并是将原子项反复地组装回原有的大数组。整个过程符合两个特征：

1. 重复（令人想到递归或迭代）
2. 有去有回（令人想到回溯，进而明确递归这条路）

因此，归并排序在实现上依托的就是递归思想。

除此之外，这里还涉及到另一个小小的知识点——**两个有序数组的合并**。合并有序数组是咱们在第 7 节讲过的一道真题，涉及到双指针法。此处强烈建议印象模糊的同学回头复习一下完整的解题思路。

编码实现

js

```
function mergeSort(arr) {  
    const len = arr.length  
    // 处理边界情况  
    if(len <= 1) {  
        return arr  
    }  
    // 计算分割点  
    const mid = Math.floor(len / 2)  
    // 递归分割左子数组，然后合并为有序数组  
    const leftArr = mergeSort(arr.slice(0, mid))  
    // 递归分割右子数组，然后合并为有序数组  
    const rightArr = mergeSort(arr.slice(mid, len))  
    // 合并左右两个有序数组  
    arr = mergeArr(leftArr, rightArr)  
    // 返回合并后的结果  
    return arr  
}
```

```
function mergeArr(arr1, arr2) {  
    // 初始化两个指针，分别指向 arr1 和 arr2  
    let i = 0, j = 0  
    // 初始化结果数组  
    const res = []  
    // 缓存arr1的长度  
    const len1 = arr1.length  
    // 缓存arr2的长度  
    const len2 = arr2.length  
    // 合并两个子数组  
    while(i < len1 && j < len2) {
```



```
        } else {
            res.push(arr2[j])
            j++
        }
    }
    // 若其中一个子数组首先被合并完全，则直接拼接另一个子数组的剩余部分
    if(i < len1) {
        return res.concat(arr1.slice(i))
    } else {
        return res.concat(arr2.slice(j))
    }
}
```

编码复盘——归并排序的时间复杂度分析

归并排序的时间复杂度的分析，同样是基于分治法。

基于数学计算的分析

我们假设规模为 n 的数组对应的排序的时间复杂度是一个关于 n 的函数 $F(n)$ 。那么它和自己的两个子数组之间就有如下关系：

$$F(n) = F(n/2) + F(n/2) + \text{合并两个数组的时间} \quad \text{js}$$

合并两个数组的过程一共要对 n 个元素进行一轮循环，因此时间复杂度可以目测出来是 $O(n)$ ，代入上面公式：

$$F(n) = F(n/2) + F(n/2) + O(n) = 2^1 * T(n/2) + 2^0 * O(n) \quad \text{js}$$

继续细分，两个子数组被划分为四个子数组，仍然遵循上面公式所描述的关系。代入 $n/4$ 后可以得到四个子数组和大数组之间的关系：

$$F(n/2) = 2 * F(n/4) + O(n) \quad \text{js}$$

$$F(n) = 2 * (2 * F(n/4) + O(n)) + O(n) = 2^2 * F(n/4) + 2^1 * O(n)$$



以得到归并排序的时间复杂度：

$$F(n) = nF(1) + O(n\log(n)) = O(n\log(n))$$

js

综上所述，归并排序的时间复杂度是 $O(n\log(n))$ 。

基于逻辑的分析

如果上面的数学公式让你感到不友好，那么我们通过简单的逻辑估算，也可以得出归并排序的时间复杂度：

逻辑估算的核心思想是“抓主要矛盾”。我们可以回顾一下归并排序的代码：

```
function mergeSort(arr) {  
  const len = arr.length  
  // 处理边界情况  
  if(len <= 1) {  
    return arr  
  }  
  // 计算分割点  
  const mid = Math.floor(len / 2)  
  // 递归分割左子数组，然后合并为有序数组  
  const leftArr = mergeSort(arr.slice(0, mid))  
  // 递归分割右子数组，然后合并为有序数组  
  const rightArr = mergeSort(arr.slice(mid, len))  
  // 合并左右两个有序数组  
  arr = mergeArr(leftArr, rightArr)  
  // 返回合并后的结果  
  return arr  
}
```

js

```
function mergeArr(arr1, arr2) {  
  // 初始化两个指针，分别指向 arr1 和 arr2  
  let i = 0, j = 0  
  // 初始化结果数组  
  const res = []  
  // 缓存arr1的长度  
  const len1 = arr1.length  
  // 缓存arr2的长度  
  const len2 = arr2.length  
  // 合并两个子数组  
  while(i < len1 && j < len2) {
```



```
        } else {
            res.push(arr2[j])
            j++
        }
    }
    // 若其中一个子数组首先被合并完全，则直接拼接另一个子数组的剩余部分
    if(i < len1) {
        return res.concat(arr1.slice(i))
    } else {
        return res.concat(arr2.slice(j))
    }
}
```

我们把每一次切分+归并看做是一轮。对于规模为 n 的数组来说，需要切分 $\log(n)$ 次，因此就有 $\log(n)$ 轮。

每一轮中，切分动作都是小事情，只需要固定的几步：

```
// 计算分割点
const mid = Math.floor(len / 2)
// 递归分割左子数组，然后合并为有序数组
const leftArr = mergeSort(arr.slice(0, mid))
// 递归分割右子数组，然后合并为有序数组
const rightArr = mergeSort(arr.slice(mid, len))
```

js

因此单次切分对应的是常数级别的时间复杂度 $O(1)$ 。

再看合并，单次合并的时间复杂度为 $O(n)$ 。 $O(n)$ 和 $O(1)$ 完全不在一个复杂度量级上，因此本着“抓主要矛盾”的原则，我们可以认为：决定归并排序时间复杂度的操作就是合并操作。

$\log(n)$ 轮对应 $\log(n)$ 次合并操作，因此归并排序的时间复杂度就是 $O(n\log(n))$ 。

以上两种时间复杂度的计算思路，大家理解其中一种即可，不必死磕。

!

快速排序



原有的数组内部进行排序。

思路分析

快速排序会将原始的数组筛选成较小和较大的两个子数组，然后递归地排序两个子数组。

这个描述对初学者来说可能会比较抽象，我们直接通过真实排序的过程来理解它：

真实排序过程演示

首先要做的事情就选取一个基准值。基准值的选择有很多方式，这里我们选取数组中间的值：

```
[5, 1, 3, 6, 2, 0, 7]
  ↑      基准      ↑
```

js

左右指针分别指向数组的两端。接下来我们要做的，就是先移动左指针，直到找到一个不小于基准值的值为止；然后再移动右指针，直到找到一个不大于基准值的值为止。

首先我们来看左指针，5比6小，故左指针右移一位：

```
[5, 1, 3, 6, 2, 0, 7]
  ↑      基准      ↑
```

js

继续对比，1比6小，继续右移左指针：

```
[5, 1, 3, 6, 2, 0, 7]
  ↑      基准      ↑
```

js

继续对比，3比6小，继续右移左指针，左指针最终指向了基准值：

```
[5, 1, 3, 6, 2, 0, 7]
  ↑      基准      ↑
  ↑
```

js



```
[5, 1, 3, 6, 2, 0, 7]
```

```
      基准  ↑  
      ↑
```

js

发现 0 比 6 小，停下来，交换 6 和 0，同时两个指针共同向中间走一步：

```
[5, 1, 3, 0, 2, 6, 7]
```

```
      ↑ 基准  
      ↑
```

js

此时 2 比 6 小，故右指针不动，左指针继续前进：

```
[5, 1, 3, 0, 2, 6, 7]
```

```
      ↑ 基准  
      right↑  
      left
```

js

此时右指针所指的值不大于 6，左指针所指的值不小于 6，故两个指针都不再移动。此时我们会发现，对于左指针所指的数字来说，它左边的所有数字都比它小，右边的所有数字都比它大（这里注意也可能存在相等的情况）。由此我们就能够以左指针为轴心，划分出一左一右、一小一大两个子数组：

```
[5, 1, 3, 0, 2]
```

```
[6, 7]
```

js

针对两个子数组，重复执行以上操作，直到数组完全排序为止。这就是快速排序的整个过程。

编码实现

```
// 快速排序入口
```

```
function quickSort(arr, left = 0, right = arr.length - 1) {
```

```
  // 定义递归边界，若数组只有一个元素，则没有排序必要
```

```
  if(arr.length > 1) {
```

```
    // lineIndex表示下一次划分左右子数组的索引位
```

```
    const lineIndex = partition(arr, left, right)
```

js



```
        quickSort(arr, left, lineIndex-1)
    }
    // 如果右边子数组的长度不小于1，则递归快排这个子数组
    if(lineIndex<right) {
        // 右子数组以 lineIndex 为左边界
        quickSort(arr, lineIndex, right)
    }
}
return arr
}
// 以基准值为轴心，划分左右子数组的过程
function partition(arr, left, right) {
    // 基准值默认取中间位置的元素
    let pivotValue = arr[Math.floor(left + (right-left)/2)]
    // 初始化左右指针
    let i = left
    let j = right
    // 当左右指针不越界时，循环执行以下逻辑
    while(i<=j) {
        // 左指针所指元素若小于基准值，则右移左指针
        while(arr[i] < pivotValue) {
            i++
        }
        // 右指针所指元素大于基准值，则左移右指针
        while(arr[j] > pivotValue) {
            j--
        }

        // 若i<=j，则意味着基准值左边存在较大元素或右边存在较小元素，交换两个元素确保左右两侧有序
        if(i<=j) {
            swap(arr, i, j)
            i++
            j--
        }
    }
    // 返回左指针索引作为下一次划分左右子数组的依据
    return i
}

// 快速排序中使用 swap 的地方比较多，我们提取成一个独立的函数
function swap(arr, i, j) {
    [arr[i], arr[j]] = [arr[j], arr[i]]
}
```



快速排序的时间复杂度的好坏，是由基准值来决定的。

- 最好时间复杂度：它对应的是这种情况——我们每次选择基准值，都刚好是当前子数组的中间数。这时，可以确保每一次分割都能将数组分为两半，进而只需要递归 $\log(n)$ 次。这时，快速排序的时间复杂度分析思路和归并排序相似，最后结果也是 $O(n\log(n))$ 。
- 最坏时间复杂度：每次划分取到的都是当前数组中的最大值/最小值。大家可以尝试把这种情况代入快排的思路中，你会发现此时快排已经退化为了冒泡排序，对应的时间复杂度是 $O(n^2)$ 。
- 平均时间复杂度： $O(n\log(n))$

小结

经过两节的学习，大家已经掌握了前端算法面试中最常考、最关键的5种排序算法。对于已经学过的这些知识，希望大家课下多消化多反思，以“默写”为目标去反复熟悉每一个算法。

排序算法的学习，对于培养大家的时间效率敏感度、提升算法优化思维等方面是大有裨益的。在整个算法知识体系中，还有一些虽然不常考察，但同样有趣的排序算法，比如基数排序、桶排序、堆排序等等，在这里推荐学有余力、时间充裕的同学课下多读多看，在排序算法这个专题下更进一步。

大家加油！

留言

输入评论（Enter换行，⌘ + Enter发送）

发表评论



👍 点赞 🗨️ 1

PINGDIYSL 1月前

而且也没哪个官方说 快排 就必须是原地排序 不能进行拆分数组

👍 点赞 🗨️ 回复

徙倚何依 Lv2 大四学生 2月前

少了一些判断

👍 1 🗨️ 回复



敲代码的小提琴手 Lv2 前端开发练习生 @ JDT 3月前

补充下 不稳定的四种排序方法 快选希堆 🐱 最快的排序方法 快速排序 归并排序 $n \times \log n$

👍 2 🗨️ 1



敲代码的小... Lv2 3月前

另外补充下从大佬那里拿来的图 两个使用分支思想的算法的区别

👍 1 🗨️ 回复



JoeeeeeWu 6月前

好像有点复杂了快速排序，"快速排序"的思想很简单，整个排序过程只需要三步：（1）在数据集之中，选择一个元素作为"基准"（pivot）。（2）所有小于"基准"的元素，都移到"基准"的左边；所有大于"基准"的元素，都移到"基准"的右边。（3）对"基准"左边和右边的两个子集，不断重复第一步和第二步，直到所有子集只剩下一个元素为止。

```
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 = [...
```

[展开](#)

👍 2 🗨️ 1

恍惚之间 Lv1 4月前

这个不对吧，空间复杂度

👍 1 🗨️ 回复



👍 点赞 🗨 回复



nan Lv1 7月前

快排 直接运行代码 超过10个数都会报 Maximum call stack size exceeded 而且运行结果还不对

👍 1 🗨 1



Wailen Lv2 3月前

取中间值的时候得 $\text{left} + (\text{right} - \text{left}) / 2$

👍 点赞 🗨 回复



柠致 前端 @ 广东工业大学 7月前

1、通过“分治”思想，得到一个 $\log(n)$ 的执行量级，降低时间复杂度；2、快速排序与归并排序的不同在于，快速排序在自身数组进行排序，归并排序会把数组分割，再组合到新的数组中合并起来。

👍 点赞 🗨 回复



潘小安 Lv2 @不务正业的程序员 @ ... 8月前

`let pivotValue = arr[Math.floor(left + (right - left) / 2)]` 为什么要这么写 不能直接 `Math.floor((left + right) / 2)` 吗

👍 点赞 🗨 4



柠致 7月前

可以的

👍 点赞 🗨 回复

先疯队同同 7月前

$(\text{left} + \text{right}) >>> 1$

👍 点赞 🗨 回复

查看更多回复 ✓



宇瞬 前端开发 8月前

不小于，不大于，感觉好绕啊，直接说大于和小于不就好了嘛 😊

👍 1 🗨 1

恍惚之间 Lv1 4月前

也是被这个搞晕了



Benbinbin Lv2 1年前

原来快排实际上并不快 😊

👍 2 💬 回复



Winneyliu 前端开发 1年前

如果数组中存在重复的值，快速排序要怎么优化？

👍 点赞 💬 回复

Mr. Y同学 Lv1 1年前

这种快排麻烦了吧 直接这种-----

```
if (arr.length <= 1) { return arr; } var  
pivotIndex = Math.floor(arr.length / 2); // 3 var pivot = arr.splice(pivotIndex, 1)[0]; // 5  
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));
```

👍 5 💬 5

解耦偏执狂 Lv1 1年前

确实你这个更好理解

👍 点赞 💬 回复



Winneyliu 1年前

这个空间复杂度高了吧

👍 3 💬 回复

[查看更多回复](#) ▼

你程弟 Lv1 前端 @ 字节跳动 1年前

你好，快排的代码中第三行备注'// 定义递归边界，若子数组只有一个元素，则没有排序必要'，这里的判断不应该是递归边界吧，因为arr的长度从来没有改变过，arr也不是描述的子数组吧。

👍 点赞 💬 1



修言 (作者) 1年前

是的！“子”字完全多余。感谢校稿~Thanks♪(·ω·)~

👍 点赞 💬 回复



走进科学爱学习 1年前

[2, 7, 5, 6, 3, 4, 9, 1, -2]做快排结果正确，但是第一次nextPivot = 4, arr = [2, -2, 1, 3, 6, 4, 9, 5, 7] 不满足右边的所有数字都比arr[nextPivot]大



这里的 nextPivot 可能语义化不够，它其实指的是下一次快排前，划分左侧数组和右侧数组的依据。我刚刚把它改成了 lineIndex。而“基准值”这个概念其实只有一个，就是 partition 函数里面那个 pivotValue。结合上面的信息，再理解一下这个问题试试看。

👍 点赞 💬 回复



lugusliu 程序员 1年前

“此时由于 $6===6$ ，左指针停止移动。开始看右指针：右指针指向7，76吧

👍 点赞 💬 1



修言 (作者) 1年前

是 $7>6$ ，哈哈，这个笔误一个多月前就fix了。。。评论咋现在才释放/捂脸

👍 1 💬 回复



zjzbz7 1年前

在快排里面，如果左指针左侧的元素都比它小，右侧的元素都比它大，为什么下一次排序从 $l=pivot+1$ 分别进行不对呢

👍 点赞 💬 3



修言 (作者) 1年前

没懂啥意思

👍 点赞 💬 回复



得能莫忘 回复 修言 1年前

问问，为什么明明是按照基准值去划分左右子数组，而下次快排左右数组要按照 i 的下标划分，而不是按照基准值的下标

“没懂啥意思”

👍 点赞 💬 回复

查看更多回复 ✓



Lion Lv3 FE @ Dont Worry Be H... 1年前

“此时右指针所指的值小于 6，左指针所指的值满足大于等于6，故两个指针都不再移动。此时我们会发现，对左指针所指的数字来说，它左边的所有数字都比它小，右边的所有数字都比它大。”这句话的理解可能有误：假设数组是：[5, 1, 3, 6, 8, 0, 7] pivotValue = 6，left = 0，right = 6 6 与 0 交换位置后 [5, 1, 3, 0, 8, 6, 7]，此时左指针 = 4，右指针 = 4 由



1 / 点赞 2



Lion Lv3 1年前

是不是可以理解成左数组小于等于基准值，右数组大于等于基准值。

👍 点赞 💬 回复



修言（作者） 1年前

等于这种情况是存在的。文中由于是针对具体数组来讲的，案例中没有出现等于，因此案例分析中直接表述为“大于”。谢谢你的细心(づ。●。●。づ) 我试了一下用括号把等于标出来，看看有没有好一点？

👍 点赞 💬 回复



小塔塔 前端 1年前

写的很好，求更新~~

👍 1 💬 回复