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：

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

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]
```

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]
```

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]
```

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]
```

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]
```

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]
```

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：

- 分割
- 合并

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

```
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) {
        if(arr1[i] < arr2[j]) {
            res.push(arr1[i])
            i++
        } 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) + 合并两个数组的时间
```

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)
```

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)
```

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

这样不断划分下去，直到每个序列里只有一个数位置。对于规模为 `n` 的数组来说，需要划分的次数为 `log(n)` ，用 `log(n)` 替换掉上述公式中的2的次数，我们就可以得到归并排序的时间复杂度：

```
F(n) = nF(1) + O(nlog(n)) = O(nlog(n))
```

综上所述， 归并排序的时间复杂度是 `O(nlog(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
}

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) {
        if(arr1[i] < arr2[j]) {
            res.push(arr1[i])
            i++
        } 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))
```

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(nlog(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

快速排序在基本思想上和归并排序是一致的，仍然坚持“分而治之”的原则不动摇。区别在于，快速排序并不会把真的数组分割开来再合并到一个新数组中去，而是直接在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        ↑
```

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        ↑
```

继续对比，1比6小，继续右移左指针：

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

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        ↑
           ↑
```

此时由于 6===6，左指针停止移动。开始看右指针：
右指针指向7，7>6，故左移右指针：

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

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
             ↑
```

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
```

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。接着我们以左指针为轴心，划分出两个子数组：

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

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，若子数组只有一个元素，则没有排序必要
  if(arr.length > 1) {
      // 计算当前数组的基准值
      const nextPivot = partition(arr, left, right)
      // 如果左边子数组的长度不小于1，则递归快排这个子数组
```

```
        if(left < nextPivot-1) {
            quickSort(arr, left, nextPivot-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(nextPivot<right) {
            quickSort(arr, nextPivot, right)
        }
    }
    return arr
}
// 寻找基准值的过程
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) {
        // 左指针所指元素若不大于基准值，则右移左指针
        while(arr[i] < pivotValue) {
            i++
        }
        // 右指针所指元素若不小于基准值，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
        if(i<=j) {
            swap(arr, i, j)
            i++
            j--
        }

    }
    // 返回左指针索引作为下一个基准值的索引
```

```
    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]]
  }
```

编码复盘——快速排序的时间复杂度分析

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(nlog(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(nlog(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！