12 | 排序（下）：如何用快排思想在O(n)内查找第K大元素?

2018-10-17 王争

12 | 排序（下）：如何用快排思想在O(n)内查找第K大元素?
朗读人：修阳　21′57″ | 8.81M

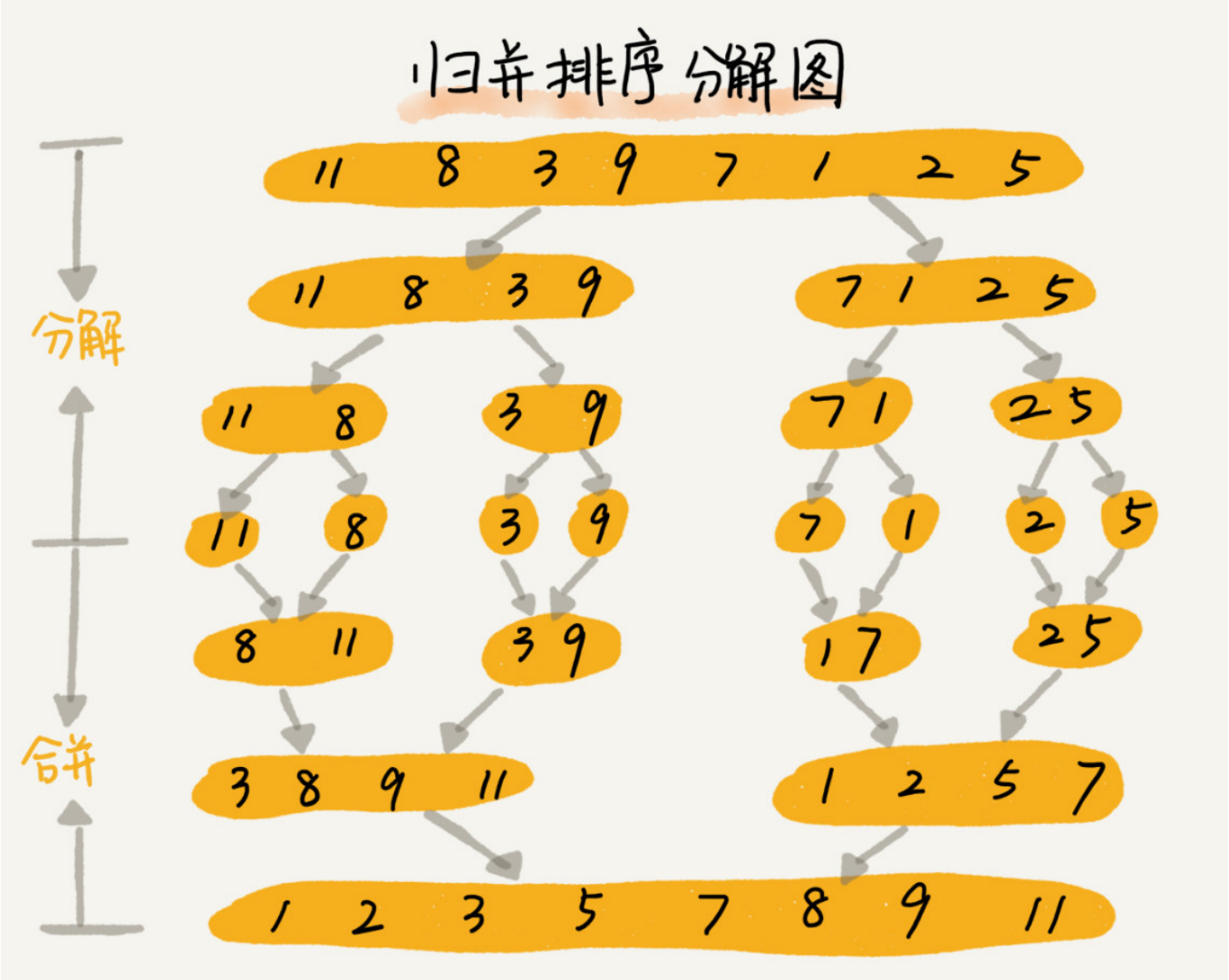
上一节我讲了冒泡排序、插入排序、选择排序这三种排序算法，它们的时间复杂度都是 $O(n^2)$，比较高，适合小规模数据的排序。今天，我讲两种时间复杂度为 O(nlogn) 的排序算法，**归并排序**和**快速排序**。这两种排序算法适合大规模的数据排序，比上一节讲的那三种排序算法要更常用。

归并排序和快速排序都用到了分治思想，非常巧妙。我们可以借鉴这个思想，来解决非排序的问题，比如：**如何在 O(n) 的时间复杂度内查找一个无序数组中的第 K 大元素？** 这就要用到我们今天要讲的内容。

归并排序的原理

我们先来看**归并排序**（Merge Sort）。

归并排序的核心思想还是蛮简单的。如果要排序一个数组，我们先把数组从中间分成前后两部分，然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了。



归并排序使用的就是**分治思想**。分治，顾名思义，就是分而治之，将一个大问题分解成小的子问题来解决。小的子问题解决了，大问题也就解决了。

从我刚才的描述，你有没有感觉到，分治思想跟我们前面讲的递归思想很像。是的，分治算法一般都是用递归来实现的。**分治是一种解决问题的处理思想，递归是一种编程技巧**，这两者并不冲突。分治算法的思想我后面会有专门的一节来讲，现在不展开讨论，我们今天的重点还是排序算法。

前面我通过举例让你对归并有了一个感性的认识，又告诉你，归并排序用的是分治思想，可以用递归来实现。我们现在就来看看**如何用递归代码来实现归并排序**。

我在第 10 节讲的递归代码的编写技巧你还记得吗？写递归代码的技巧就是，分析得出递推公式，然后找到终止条件，最后将递推公式翻译成递归代码。所以，要想写出归并排序的代码，我们先写出归并排序的递推公式。

复制代码

```
递推公式：
merge_sort(p…r) = merge(merge_sort(p…q), merge_sort(q+1…r))

终止条件：
p >= r 不用再继续分解
```

我来解释一下这个递推公式。

merge_sort(p…r) 表示，给下标从 p 到 r 之间的数组排序。我们将这个排序问题转化为了两个子问题，merge_sort(p…q) 和 merge_sort(q+1…r)，其中下标 q 等于 p 和 r 的中间位置，也就是 (p+r)/2。当下标从 p 到 q 和从 q+1 到 r 这两个子数组都排好序之后，我们再将两个有序的子数组合并在一起，这样下标从 p 到 r 之间的数据就也排好序了。

有了递推公式，转化成代码就简单多了。为了阅读方便，我这里只给出伪代码，你可以翻译成你熟悉的编程语言。

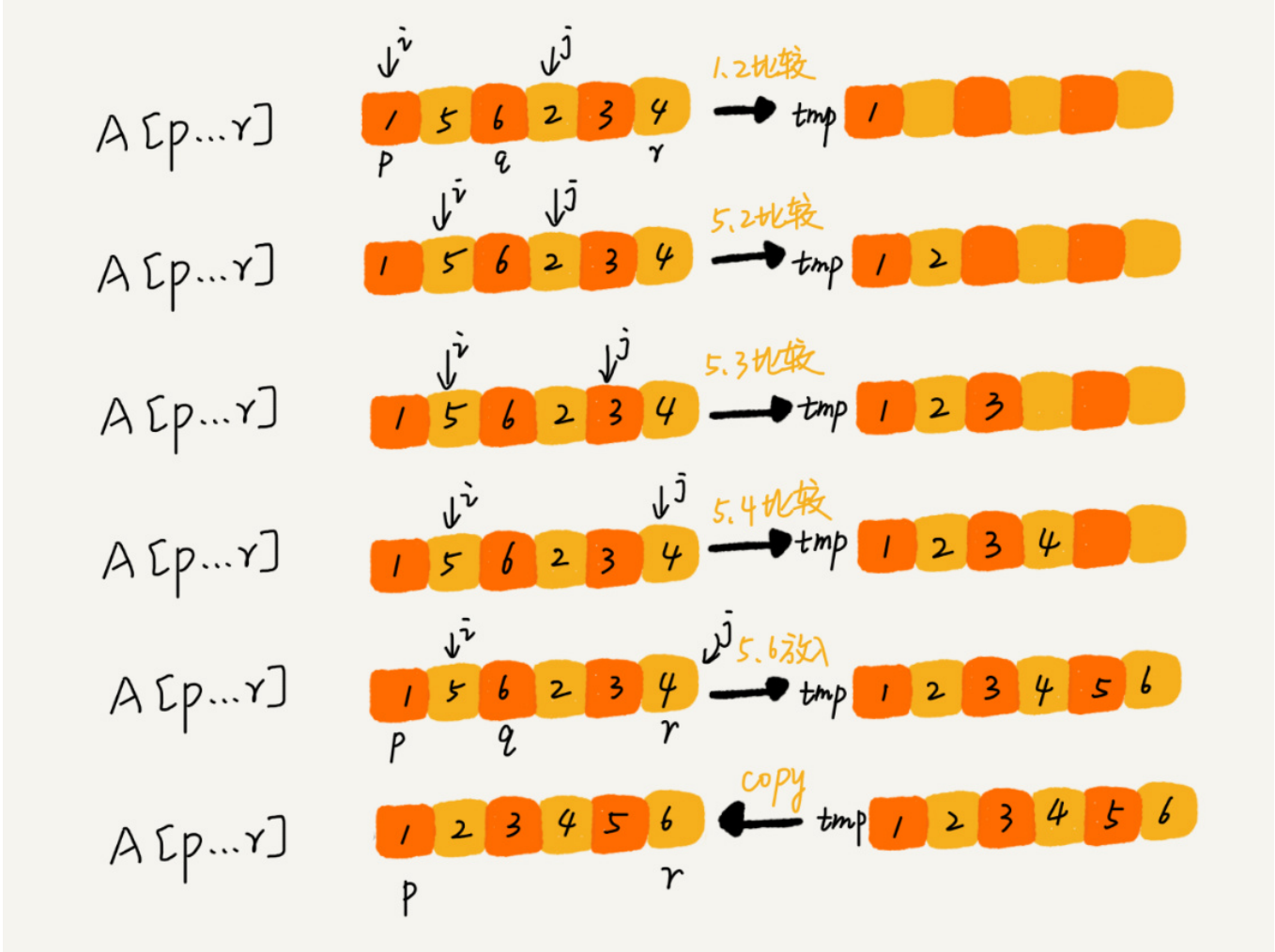
复制代码

```
// 归并排序算法, A 是数组，n 表示数组大小
merge_sort(A, n) {
  merge_sort_c(A, 0, n-1)
}

// 递归调用函数
merge_sort_c(A, p, r) {
  // 递归终止条件
  if p >= r  then return

  // 取 p 到 r 之间的中间位置 q
  q = (p+r) / 2
  // 分治递归
  merge_sort_c(A, p, q)
  merge_sort_c(A, q+1, r)
  // 将 A[p...q] 和 A[q+1...r] 合并为 A[p...r]
  merge(A[p...r], A[p...q], A[q+1...r])
}
```

你可能已经发现了，merge(A[p…r], A[p…q], A[q+1…r]) 这个函数的作用就是，将已经有序的 A[p…q] 和 A[q+1…r] 合并成一个有序的数组，并且放入 A[p…r]。那这个过程具体该如何做呢?

如图所示，我们申请一个临时数组 tmp，大小与 A[p…r] 相同。我们用两个游标 i 和 j，分别指向 A[p…q] 和 A[q+1…r] 的第一个元素。比较这两个元素 A[i] 和 A[j]，如果 A[i]<=A[j]，我们就把 A[i] 放入到临时数组 tmp，并且 i 后移一位，否则将 A[j] 放入到数组 tmp，j 后移一位。

继续上述比较过程，直到其中一个子数组中的所有数据都放入临时数组中，再把另一个数组中的数据依次加入到临时数组的末尾，这个时候，临时数组中存储的就是两个子数组合并之后的结果了。最后再把临时数组 tmp 中的数据拷贝到原数组 A[p…r] 中。



我们把 merge() 函数写成伪代码，就是下面这样：

复制代码

```
merge(A[p...r], A[p...q], A[q+1...r]) {
  var i := p, j := q+1, k := 0 // 初始化变量 i, j, k
  var tmp := new array[0...r-p] // 申请一个大小跟 A[p...r] 一样的临时数组
  while i<=q AND j<=r do {
    if A[i] <= A[j] {
      tmp[k++] = A[i++] // i++ 等于 i:=i+1
    } else {
      tmp[k++] = A[j++]
    }
  }

  // 判断哪个子数组中有剩余的数据
  var start := i, end := q
  if j<=r then start := j, end:=r

  // 将剩余的数据拷贝到临时数组 tmp
  while start <= end do {
    tmp[k++] = A[start++]
  }

  // 将 tmp 中的数组拷贝回 A[p...r]
  for i:=0 to r-p do {
    A[p+i] = tmp[i]
```

```
24    }
25  }
```

你还记得第 7 讲讲过的利用哨兵简化编程的处理技巧吗？merge() 合并函数如果借助哨兵，代码就会简洁很多，这个问题留给你思考。

归并排序的性能分析

这样跟着我一步一步分析，归并排序是不是没那么难啦？还记得上节课我们分析排序算法的三个问题吗？接下来，我们来看归并排序的三个问题。

第一，归并排序是稳定的排序算法吗？

结合我前面画的那张图和归并排序的伪代码，你应该能发现，归并排序稳不稳定关键要看 merge() 函数，也就是两个有序子数组合并成一个有序数组的那部分代码。

在合并的过程中，如果 A[p...q] 和 A[q+1...r] 之间有值相同的元素，那我们可以像伪代码中那样，先把 A[p...q] 中的元素放入 tmp 数组。这样就保证了值相同的元素，在合并前后的先后顺序不变。所以，归并排序是一个稳定的排序算法。

第二，归并排序的时间复杂度是多少？

归并排序涉及递归，时间复杂度的分析稍微有点复杂。我们正好借此机会来学习一下，如何分析递归代码的时间复杂度。

在递归那一节我们讲过，递归的适用场景是，一个问题 a 可以分解为多个子问题 b、c，那求解问题 a 就可以分解为求解问题 b、c。问题 b、c 解决之后，我们再把 b、c 的结果合并成 a 的结果。

如果我们定义求解问题 a 的时间是 T(a)，求解问题 b、c 的时间分别是 T(b) 和 T(c)，那我们就可以得到这样的递推关系式：

```
1  T(a) = T(b) + T(c) + K
```

复制代码

其中 K 等于将两个子问题 b、c 的结果合并成问题 a 的结果所消耗的时间。

从刚刚的分析，我们可以得到一个重要的结论：**不仅递归求解的问题可以写成递推公式，递归代码的时间复杂度也可以写成递推公式。**

套用这个公式，我们来分析一下归并排序的时间复杂度。

我们假设对 n 个元素进行归并排序需要的时间是 T(n)，那分解成两个子数组排序的时间都是 T(n/2)。我们知道，merge() 函数合并两个有序子数组的时间复杂度是 O(n)。所以，套用前面的

公式，归并排序的时间复杂度的计算公式就是：

```
T(1) = C;    n=1 时，只需要常量级的执行时间，所以表示为 C。
T(n) = 2*T(n/2) + n;  n>1
```

通过这个公式，如何来求解 T(n) 呢？还不够直观？那我们再进一步分解一下计算过程。

```
T(n) = 2*T(n/2) + n
     = 2*(2*T(n/4) + n/2) + n = 4*T(n/4) + 2*n
     = 4*(2*T(n/8) + n/4) + 2*n = 8*T(n/8) + 3*n
     = 8*(2*T(n/16) + n/8) + 3*n = 16*T(n/16) + 4*n
     ......
     = 2^k * T(n/2^k) + k * n
     ......
```

通过这样一步一步分解推导，我们可以得到 T(n) = 2^kT(n/2^k)+kn。当 T(n/2^k)=T(1) 时，也就是 n/2^k=1，我们得到 k=$\log_2 n$ 。我们将 k 值代入上面的公式，得到 T(n)=Cn+n$\log_2 n$ 。如果我们用大 O 标记法来表示的话，T(n) 就等于 O(nlogn)。所以归并排序的时间复杂度是 O(nlogn)。

从我们的原理分析和伪代码可以看出，归并排序的执行效率与要排序的原始数组的有序程度无关，所以其时间复杂度是非常稳定的，不管是最好情况、最坏情况，还是平均情况，时间复杂度都是 O(nlogn)。

第三，归并排序的空间复杂度是多少？

归并排序的时间复杂度任何情况下都是 O(nlogn)，看起来非常优秀。（待会儿你会发现，即便是快速排序，最坏情况下，时间复杂度也是 O(n^2)。）但是，归并排序并没有像快排那样，应用广泛，这是为什么呢？因为它有一个致命的“弱点”，那就是归并排序不是原地排序算法。

这是因为归并排序的合并函数，在合并两个有序数组为一个有序数组时，需要借助额外的存储空间。这一点你应该很容易理解。那我现在问你，归并排序的空间复杂度到底是多少呢？是 O(n)，还是 O(nlogn)，应该如何分析呢？

如果我们继续按照分析递归时间复杂度的方法，通过递推公式来求解，那整个归并过程需要的空间复杂度就是 O(nlogn)。不过，类似分析时间复杂度那样来分析空间复杂度，这个思路对吗？

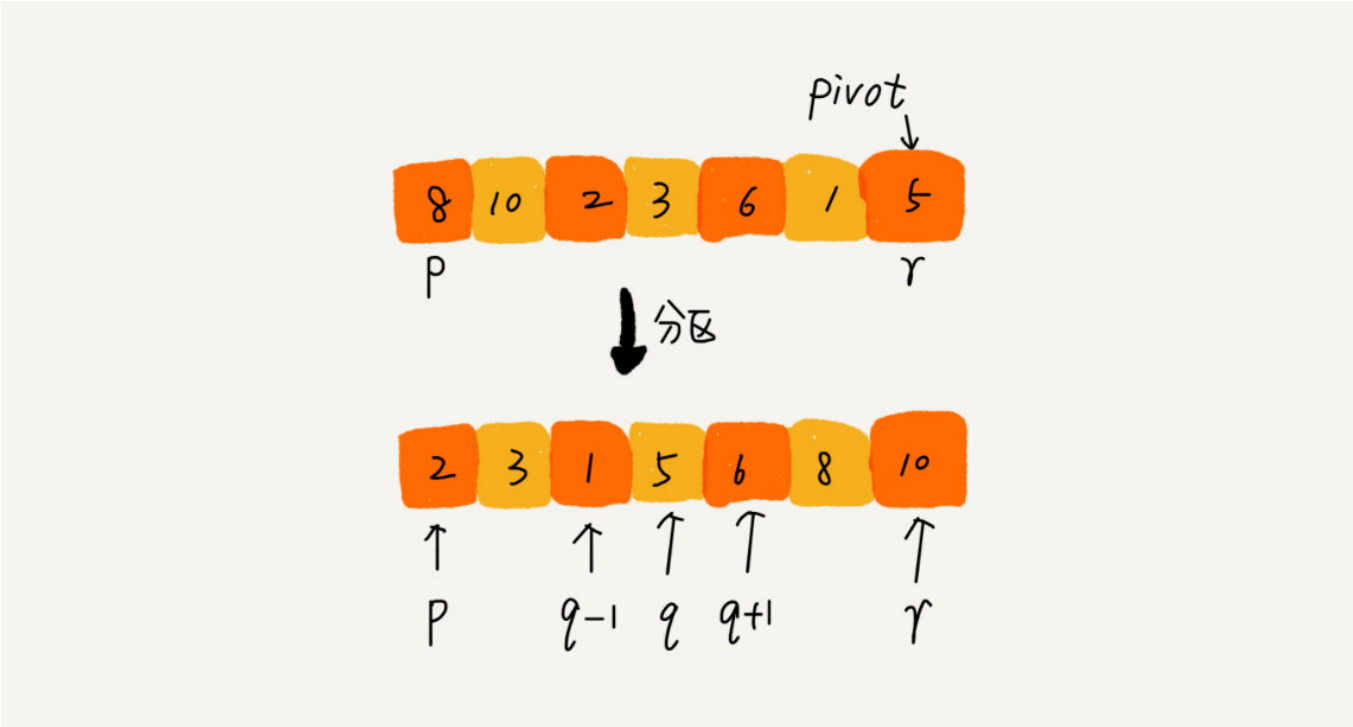
实际上，递归代码的空间复杂度并不能像时间复杂度那样累加。刚刚我们忘记了最重要的一点，那就是，尽管每次合并操作都需要申请额外的内存空间，但在合并完成之后，临时开辟的内存空间就被释放掉了。在任意时刻，CPU 只会有一个函数在执行，也就只会有一个临时的内存空间在使用。临时内存空间最大也不会超过 n 个数据的大小，所以空间复杂度是 O(n)。

快速排序的原理

我们再来看快速排序算法（Quicksort），我们习惯性把它简称为“快排”。快排利用的也是分治思想。乍看起来，它有点像归并排序，但是思路其实完全不一样。我们待会会讲两者的区别。现在，我们先来看下快排的核心思想。

快排的思想是这样的：如果要排序数组中下标从 p 到 r 之间的一组数据，我们选择 p 到 r 之间的任意一个数据作为 pivot（分区点）。

我们遍历 p 到 r 之间的数据，将小于 pivot 的放到左边，将大于 pivot 的放到右边，将 pivot 放到中间。经过这一步骤之后，数组 p 到 r 之间的数据就被分成了三个部分，前面 p 到 q-1 之间都是小于 pivot 的，中间是 pivot，后面的 q+1 到 r 之间是大于 pivot 的。



根据分治、递归的处理思想，我们可以用递归排序下标从 p 到 q-1 之间的数据和下标从 q+1 到 r 之间的数据，直到区间缩小为 1，就说明所有的数据都有序了。

如果我们用递推公式来将上面的过程写出来的话，就是这样：

复制代码

```
递推公式：
quick_sort(p…r) = quick_sort(p…q-1) + quick_sort(q+1, r)

终止条件：
p >= r
```

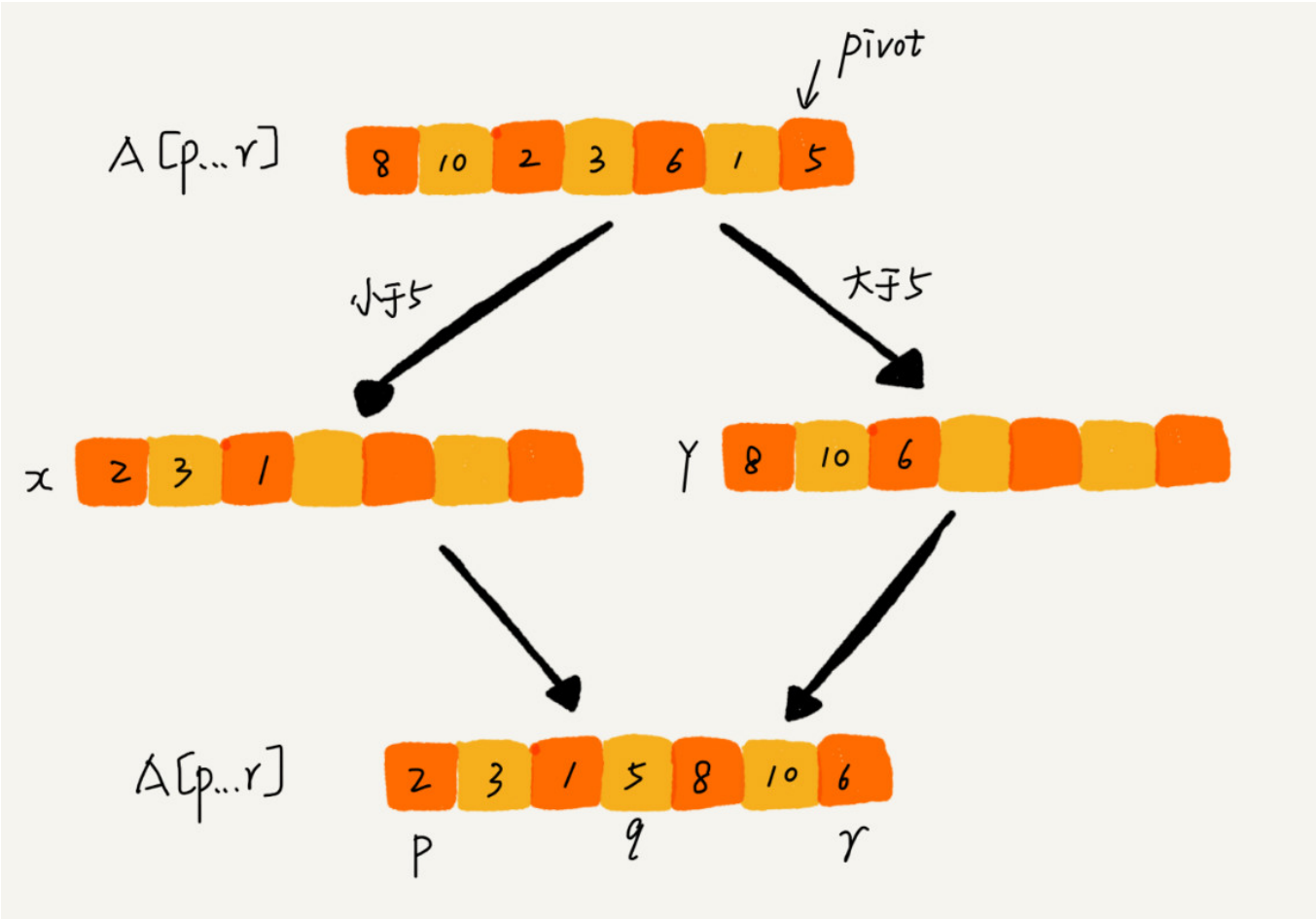
我将递推公式转化成递归代码。跟归并排序一样，我还是用伪代码来实现，你可以翻译成你熟悉的任何语言。

复制代码

```
// 快速排序，A 是数组，n 表示数组的大小
quick_sort(A, n) {
  quick_sort_c(A, 0, n-1)
}
// 快速排序递归函数，p,r 为下标
quick_sort_c(A, p, r) {
  if p >= r then return

  q = partition(A, p, r) // 获取分区点
  quick_sort_c(A, p, q-1)
  quick_sort_c(A, q+1, r)
}
```

归并排序中有一个 merge() 合并函数，我们这里有一个 partition() 分区函数。partition() 分区函数实际上我们前面已经讲过了，就是随机选择一个元素作为 pivot（一般情况下，可以选择 p 到 r 区间的最后一个元素），然后对 A[p...r] 分区，函数返回 pivot 的下标。

如果我们不考虑空间消耗的话，partition() 分区函数可以写得非常简单。我们申请两个临时数组 X 和 Y，遍历 A[p...r]，将小于 pivot 的元素都拷贝到临时数组 X，将大于 pivot 的元素都拷贝到临时数组 Y，最后再将数组 X 和数组 Y 中数据顺序拷贝到 A[p...r]。



但是，如果按照这种思路实现的话，partition() 函数就需要很多额外的内存空间，所以快排就不是原地排序算法了。如果我们希望快排是原地排序算法，那它的空间复杂度得是 O(1)，那

partition() 分区函数就不能占用太多额外的内存空间，我们就需要在 A[p...r] 的原地完成分区操作。

原地分区函数的实现思路非常巧妙，我写成了伪代码，我们一起来看一下。

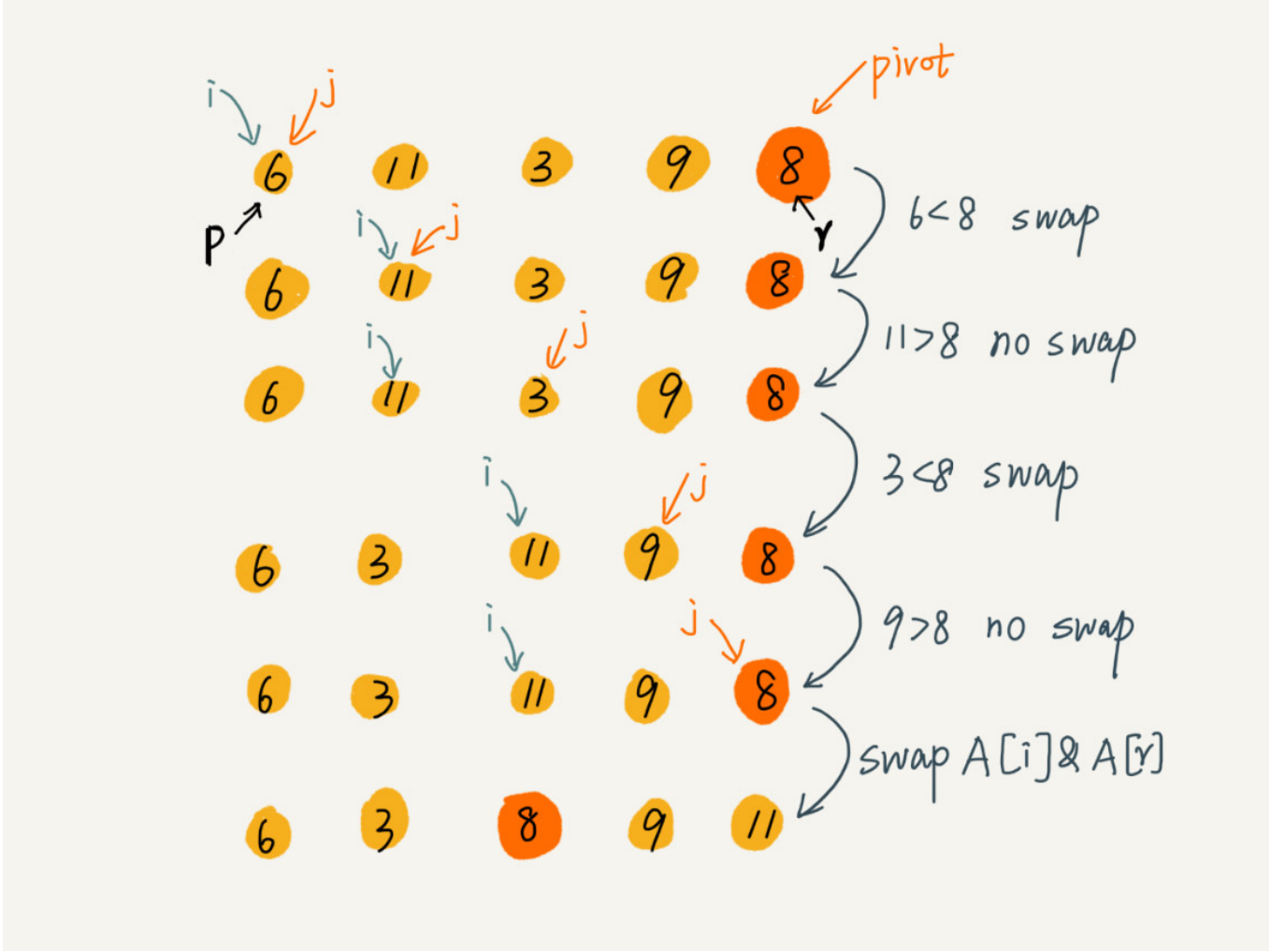
复制代码

```
partition(A, p, r) {
  pivot := A[r]
  i := p
  for j := p to r-1 do {
    if A[j] < pivot {
      swap A[i] with A[j]
      i := i+1
    }
  }
  swap A[i] with A[r]
  return i

```

这里的处理有点类似选择排序。我们通过游标 i 把 A[p...r-1] 分成两部分。A[p...i-1] 的元素都是小于 pivot 的，我们暂且叫它“已处理区间”，A[i...r-1] 是“未处理区间”。我们每次都从未处理的区间 A[i...r-1] 中取一个元素 A[j]，与 pivot 对比，如果小于 pivot，则将其加入到已处理区间的尾部，也就是 A[i] 的位置。

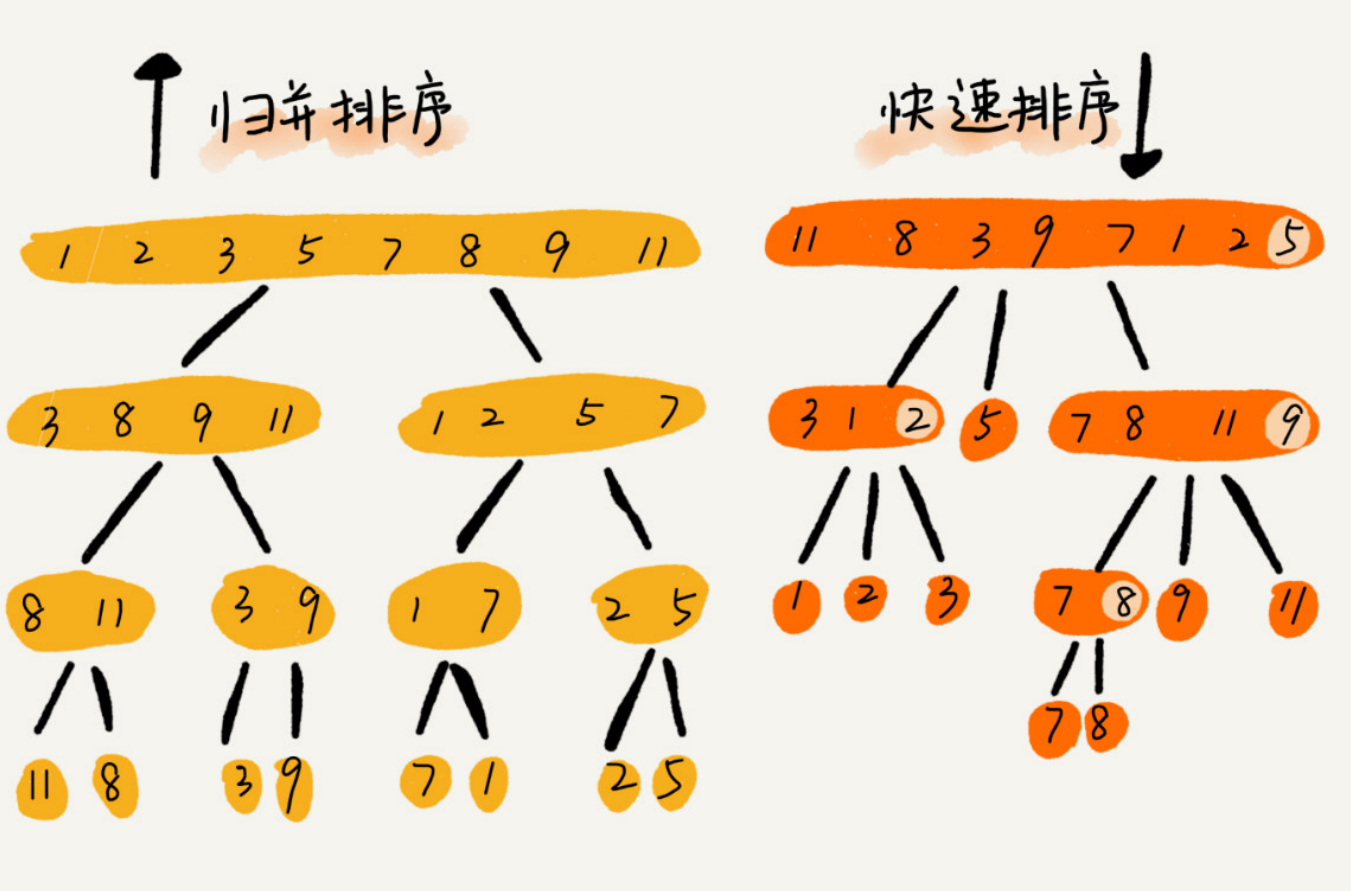
数组的插入操作还记得吗？在数组某个位置插入元素，需要搬移数据，非常耗时。当时我们也讲了一种处理技巧，就是交换，在 O(1) 的时间复杂度内完成插入操作。这里我们也借助这个思想，只需要将 A[i] 与 A[j] 交换，就可以在 O(1) 时间复杂度内将 A[j] 放到下标为 i 的位置。

文字不如图直观，所以我画了一张图来展示分区的整个过程。



因为分区的过程涉及交换操作，如果数组中有两个 8，其中一个是 pivot，经过分区处理之后，后面的 8 就有可能被放到了另一个 8 的前面，先后顺序就颠倒了。所以，快速排序并不是一个稳定的排序算法。

到此，快速排序的原理你应该也掌握了。现在，我再来看另外一个问题：快排和归并用的都是分治思想，递推公式和递归代码也非常相似，那它们的区别在哪里呢？



可以发现，归并排序的处理过程是**由下到上**的，先处理子问题，然后再合并。而快排正好相反，它的处理过程是**由上到下**的，先分区，然后再处理子问题。归并排序虽然是稳定的、时间复杂度为 O(nlogn) 的排序算法，但是它是非原地排序算法。我们前面讲过，归并之所以是非原地排序算法，主要原因是合并函数无法在原地执行。快速排序通过设计巧妙的原地分区函数，可以实现原地排序，解决了归并排序占用太多内存的问题。

快速排序的性能分析

现在，我们来分析一下快速排序的性能。我在讲解快排的实现原理的时候，已经分析了稳定性和空间复杂度。快排是一种原地、不稳定的排序算法。现在，我们集中精力来看快排的时间复杂度。

快排也是用递归来实现的。对于递归代码的时间复杂度，我前面总结的公式，这里也还是适用的。如果每次分区操作，都能正好把数组分成大小接近相等的两个小区间，那快排的时间复杂度递推求解公式跟归并是相同的。所以，快排的时间复杂度也是 O(nlogn)。

复制代码

```
T(1) = C;    n=1 时，只需要常量级的执行时间，所以表示为 C。
T(n) = 2*T(n/2) + n;  n>1
```

但是，公式成立的前提是每次分区操作，我们选择的 pivot 都很合适，正好能将大区间对等地一分为二。但实际上这种情况是很难实现的。

我举一个比较极端的例子。如果数组中的数据原来已经是有序的了，比如 1，3，5，6，8。如果我们每次选择最后一个元素作为 pivot，那每次分区得到的两个区间都是不均等的。我们需要进行大约 n 次分区操作，才能完成快排的整个过程。每次分区我们平均要扫描大约 n/2 个元素，这种情况下，快排的时间复杂度就从 O(nlogn) 退化成了 $O(n^2)$。

我们刚刚讲了两个极端情况下的时间复杂度，一个是分区极其均衡，一个是分区极其不均衡。它们分别对应快排的最好情况时间复杂度和最坏情况时间复杂度。那快排的平均情况时间复杂度是多少呢？

我们假设每次分区操作都将区间分成大小为 9:1 的两个小区间。我们继续套用递归时间复杂度的递推公式，就会变成这样：

复制代码

```
T(1) = C;    n=1 时，只需要常量级的执行时间，所以表示为 C。

T(n) = T(n/10) + T(9*n/10) + n;  n>1
```

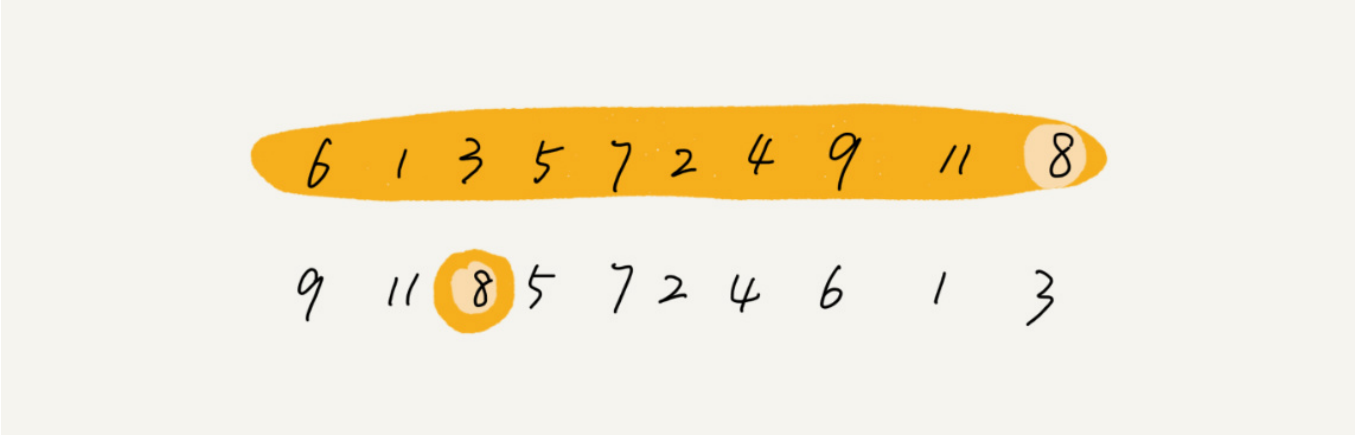
这个公式的递推求解的过程非常复杂，虽然可以求解，但我不推荐用这种方法。实际上，递归的时间复杂度的求解方法除了递推公式之外，还有递归树，在树那一节我再讲，这里暂时不说。我这里直接给你结论：T(n) 在大部分情况下的时间复杂度都可以做到 O(nlogn)，只有在极端情况下，才会退化到 $O(n^2)$。而且，我们也有很多方法将这个概率降到很低，如何来做？我们下节再讲。

解答开篇

快排核心思想就是**分治**和**分区**，我们可以利用分区的思想，来解答开篇的问题：O(n) 时间复杂度内求无序数组中的第 K 大元素。比如，4， 2， 5， 12， 3 这样一组数据，第 3 大元素就是 4。

我们选择数组区间 A[0…n–1] 的最后一个元素 A[n–1] 作为 pivot，对数组 A[0…n–1] 原地分区，这样数组就分成了三部分，A[0…p–1]、A[p]、A[p+1…n–1]。

如果 p+1=K，那 A[p] 就是要求解的元素；如果 K>p+1, 说明第 K 大元素出现在 A[p+1…n–1] 区间，我们再按照上面的思路递归地在 A[p+1…n–1] 这个区间内查找。同理，如果 K<p+1，那我们就在 A[0…p–1] 区间查找。



我们再来看，为什么上述解决思路的时间复杂度是 O(n)？

第一次分区查找，我们需要对大小为 n 的数组执行分区操作，需要遍历 n 个元素。第二次分区查找，我们只需要对大小为 n/2 的数组执行分区操作，需要遍历 n/2 个元素。依次类推，分区遍历元素的个数分别为、n/2、n/4、n/8、n/16.....直到区间缩小为 1。

如果我们把每次分区遍历的元素个数加起来，就是：n+n/2+n/4+n/8+...+1。这是一个等比数列求和，最后的和等于 2n–1。所以，上述解决思路的时间复杂度就为 O(n)。

你可能会说，我有个很笨的办法，每次取数组中的最小值，将其移动到数组的最前面，然后在剩下的数组中继续找最小值，以此类推，执行 K 次，找到的数据不就是第 K 大元素了吗？

不过，时间复杂度就并不是 O(n) 了，而是 O(K * n)。你可能会说，时间复杂度前面的系数不是可以忽略吗？O(K * n) 不就等于 O(n) 吗？

这个可不能这么简单地划等号。当 K 是比较小的常量时，比如 1、2，那最好时间复杂度确实是 O(n)；但当 K 等于 n/2 或者 n 时，这种坏情下的况时间复杂度就是 $O(n^2)$ 了。

内容小结

归并排序和快速排序是两种稍微复杂的排序算法，它们用的都是分治的思想，代码都通过递归来实现，过程非常相似。理解归并排序的重点是理解递推公式和 merge() 合并函数。同理，理解快排的重点也是理解递推公式，还有 partition() 分区函数。

归并排序算法是一种在任何情况下时间复杂度都比较稳定的排序算法，这也使它存在致命的缺点，即归并排序不是原地排序算法，空间复杂度比较高，是 O(n)。正因为此，它也没有快排应用广泛。

快速排序算法虽然最坏情况下的时间复杂度是 $O(n^2)$，但是平均情况下时间复杂度都是 O(nlogn)。不仅如此，快速排序算法时间复杂度退化到 $O(n^2)$ 的概率非常小，我们可以通过合理地选择 pivot 来避免这种情况。

课后思考

现在你有 10 个接口访问日志文件，每个日志文件大小约 300MB，每个文件里的日志都是按照时间戳从小到大排序的。你希望将这 10 个较小的日志文件，合并为 1 个日志文件，合并之后的日志仍然按照时间戳从小到大排列。如果处理上述排序任务的机器内存只有 1GB，你有什么好的解决思路，能“快速”地将这 10 个日志文件合并吗？

欢迎留言和我分享，我会第一时间给你反馈。

我已将本节内容相关的详细代码更新到 Github，戳此即可查看。

写留言

精选留言

王先统 7

可以为每个文件分配一个40M的数组，再另外分配一个400M的数组储存归并结果，每个文件每次读取40M，对十个数组做归并排序直到其中某个数组的数据被处理完，这时将归并结果写入磁盘，处理完的数组继续读入40M继续参与归并，以此类推，直到所有文件都处理完

2018-10-17

刘远通 3

老师 这个地方我没看懂...
“第一次分区查找，我们需要对大小为 n 的数组执行分区操作，需要遍历 n 个元素。第二次分区查找，我们只需要对大小为 n/2 的数组执行分区操作，需要遍历 n/2 个元素。依次类推，分区遍历元素的个数分别为、n/2、n/4、n/8、n/16.....直到区间缩小为 1。”
这里讨论的是最平均的情况吗？ 最坏的情况可以n,n-1,...,1啊

2018-10-17

侯金彪 3

老师，有个问题没懂，在一个数组中找第k大的数这个问题中，为什么如果p+1=k，a[p]就是要查找的结果呢?

2018-10-17

yaya 3

以前写快排的时候总是喜欢用第一个元素作为k值，partion喜欢用找到左右不符合规则的元素再对调，第k大总是纠结于，下次递归应该是第几大，😂这些其实只要把最后一个元素作为key就可以简化代码了。这门课让我懂了以前好多不懂得小细节。应该说曾经认为自己懂了吧，尤其是排序这里。

思考题，由于本来十个部分就是有序的，利用十个index，把这十个读入内存比较，在里面选择最小的一个index增1，如果一个部分放完了，就继续放剩下的九个部分，直到只剩一个部分，再复制。

2018-10-17

陈华应 2

坚持初衷，死磕就行，不退缩，不放弃！

2018-10-17

冷笑的花猫 2

老师您好，最后的思考题思路基本都是先拆成小文件，然后取出topK，最后再合并。疑惑的是内存不够，怎么读到内存中。如果不读到内存中该怎么实现，读不读到内存中的标准是什么？如果每个文件都是3g，内存只有1g，思路类似。老师能提供下详细的代码吗？相信绝大部分同学都有疑惑，谢谢。

2018-10-17

朱月俊 2

假设这十个文件都在本地，同时打开这十个文件，每个文件加一个offset，每次比较十个文件中每个文件最新的一条日志，按照时间戳获取时间戳最小的一条日志，然后对应文件offset加1，按照这种方法可以把在内存很小的情况下读完文件，积累一定大小记录在内存后flush到磁盘。

2018-10-17

趁风卷 1

快排的partition函数的空间复杂度是O(1)，但整个排序过程是递归的，最多会有O(logn)个函数放在栈里。这一部分不算空间复杂度么?

2018-10-17

包身工 1

考虑三路归并排序，如果由于空间限制导致速度较慢，使用两路归并最后即可合并成一个大文件！

2018-10-17

城 1

快排的伪代码写的过于简单，能否按照面试级别给我们写个实例代码呢

2018-10-17

峰 1

每次从各个文件中取一条数据，在内存中根据数据时间戳构建一个最小堆，然后每次把最小值给写入新文件，同时将最小值来自的那个文件再出来一个数据，加入到最小堆中。这个空间复杂度为常数，但没能很好利用1g内存，而且磁盘单个读取比较慢，所以考虑每次读取一批数据，没了再从磁盘中取，时间复杂度还是一样O(n)。

2018-10-17

Light Lin 0

伪代码反而看得费劲，可能还是对代码不够敏感吧

2018-10-17

作者回复

那我以后还是写代码吧

2018-10-17

等风来 0

快排partition函数按搬移数据实现的话,可实现稳定排序且时间复杂度和空间复杂度保持不变;
归并排序Merge函数通过将数组的后半部分插入到前半部分可实现原地排序,时间复杂度会增加.
当然这两种实现方式显然都不是最佳实践.
思考题:
1.从10个文件中每次取50M进行排序,合并成6个500M的文件;均分成3组;
2.组中取每个文件前250M合并成一个文件A,后250M合并成B;
3.将A的前250M写入一个单独文件C;
4.A剩下250M和B的500M的前250M合并D,将D的前250M追加到C中;
5.将B剩下250M和D剩下的250合并写入C中,以此合并成3个1000M的有序文件E,F,G;
6.将E,F分别取250M合并,并将前250M写入H中,后250M写入I;
7.分别从E,F取250M合并写入J;
8.将J前250M和I合并后,取前250M追加到H,剩下回写I;
9.合并I,J写入I;
10.以此类推,得到2000M的H和1000M的G;
11.用类似的方式合并H,G即可.

2018-10-17

豆泥丸 0

编辑大大可不可以也开一个画图的专栏，图好漂亮啊

2018-10-17

柠檬C

0

这个伪代码像pascal风格啊

2018-10-17

作者回复

能看懂就行了

2018-10-17

lovedebug

0

建议为github上代码示例都加上最好/最坏/平均时间复杂度，方便自己分析

2018-10-17

DADDYHINS

0

把文件分小组，每组1g，在内存中进行归并排序，排序完了写进目标文件，以此类推

2018-10-17

walor

0

课后思考

1.申请 10 个数组，每个数组 30M。再申请个临时数组，大小 300M。

2.每个文件每次读取 30M 数据分别存入 30M 的数组中。

3.比较 10 个数组的最大时间戳，获取它们中的最小值

4.合并排序 10 个数组。排序完成后，取出数组前 30M 存入磁盘（硬盘上的最终日志文件），后 270M 存入文件

5.循环至 300M 文件读取结束，生成 10 个新的 270M 文件

6.重复步骤 1。

7.终止条件：生成的新文件大小 0M。

2018-10-17

NeverMore

0

充分利用1G的内存空间。分三次，将300M的文件分成3份，每份100M，10个文件一共1G，内存内对这1G文件进行归并排序。然后将排序好的1G文件写入到磁盘。这样最终会有3份1G的文件落到磁盘上，然后通过读取每份文件的时间戳，对这3份文件进行排序并直接写入到磁盘内（外排）。

比较依赖磁盘的I/O性能。

2018-10-17

Smallfly

0

并归排序 vs. 快速排序

并归排序跟数据有序度无关，时间复杂度稳定在 O(nlogn); 快速排序复杂度受有序度的影响，最坏情况复杂度是 O(n^2)，平均复杂度是 O(nlogn)。

并归排序的处理是由下到上的，所以需要额外的空间，将有序数组进行重组，空间复杂度为 O(n)。快速排序的处理过程是由上而下的，分区的过程中排序，所以它是原地排序。

归并排序合并时可以保证相同元素保持原来的顺序，所以是稳定的排序；而快速排序是不稳定排序。

2018-10-17

夏天🔆 0

每个文件切成100M的区块，对比每个区块的起止时间，区块间按照起止时间排序有时间交集的load进内存进行快排成一个大文件，记录成一个大区块，最后没有时间交集的区块按照时间顺序用脚本拼接

2018-10-17

Yeni 0

正在准备面试，看了老师的课很多以前不理解的问题现在都能想通了，课讲得非常好

2018-10-17

cricket1981 0

分三批处理，每批读入每个文件100MB，10个文件是1G，在内存中归并排序后写入目标文件，完了再下一批处理

2018-10-17

leixingzhi7 0

难度慢慢提升了，要坚持看下去啊都

2018-10-17

jon 0

归并时间复杂度推导时当 T(n/2^k)=T(1) 时。 这里没看懂能解释一下吗

2018-10-17

徐凯 0

思考题是不是通过将文件放在外存 多路归并排序

2018-10-17