例题分析一

LeetCode 第 03 题：给定一个字符串，请你找出其中不含有重复字符的最长子串的长度。

示例 1

输入："abcabcbb"

输出：3

解释：因为无重复字符的最长子串是"abc"，其长度为 3。

示例 2

输入："bbbbb"

输出：1

解释：因为无重复字符的最长子串是 "b"，其长度为 1。

示例 3

输入："pwwkew"

输出：3

解释：因为无重复字符的最长子串是 "wke"，其长度为 3。

注意：答案必须是子串的长度，"pwke" 是一个子序列，不是子串。

解题思路一：暴力法

3. 无重复字符的最长子串

▸ **解法一：暴力法**

- 假设字符串长度为 n
- 非空子串为 $\frac{n(n+1)}{2}$ 个

找出所有的子串，然后一个一个地去判断每个子串里是否包含有重复的字符。

假设字符串的长度为 n，那么有 n×(n + 1) / 2 个非空子串。计算过程如下。

长度为 1 的子串，有 n 个

长度为 2 的子串，每两个每两个相邻地取，一共有 n - 1 个

长度为 3 的子串，每三个每三个相邻地取，一共有 n - 2 个

……

以此类推，长度为 k 的子串，有 n - k + 1 个。

当 k 等于 n 的时候，n - k + 1=1，即长度为 n 的子串有 1 个。

所有情况相加，得到所有子串的长度为：

$$n + (n - 1) + (n - 2) + (n - 3) + \ldots + 2 + 1 = n \times (n + 1) / 2$$

算上空字符串，那么就一共有 $n \times (n + 1) / 2 + 1$ 个。

拓展一下，对于一个长度为 n 的字符串，一共有多少个子序列呢？和子串不一样，子序列里的元素不需要相互挨着。

同理分析，长度为 1 的子序列有 n 个，即 Cn^1，长度为 2 的子序列个数为 Cn^2，以此类推，长度为 k 的子序列有 Cn^k，那么所有子序列的个数（包括空序列）是 $Cn^0 + Cn^1 + Cn^2 + \ldots Cn^n = 2^n$

注意：对于统计子串和子序列个数的方法和结果，大家务必记下来，对于在分析各种问题时会有很大帮助。

回到本来问题，如果对所有的子串进行判断，从每个子串里寻找出最长的那个并且没有重复字符的，那么复杂度就是：$O(n \times (n + 1)/2 \times n) = O(n^3)$。

解题思路二：线性法

例题 1：给定的字符串里有一段是没有重复字符的，如下，能不能把下一个字符 a 加进来？

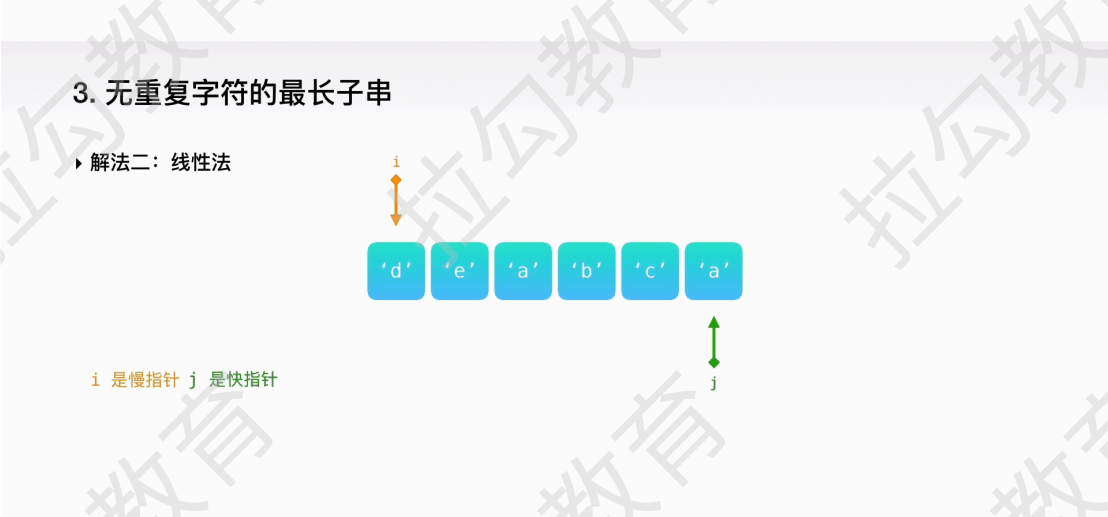
要看当前的子串”abc”是否已经包含了字符 a。

1. 扫描一遍“abc”，当发现某个字符与 a 相同，可以得出结论。
2. 把“abc“三个字符放入到一个哈希集合里，那么就能在 O(1) 的时间里作出判断，提高速度。

使用定义一个哈希集合 set 的方法，从给定字符串的头开始，每次检查一下当前字符是不是在集合里边，如果不在，说明这个字符不会造成重复和冲突，把它加入到集合里，并统计一下当前集合的长度，可能它就是最长的那个子串。

例题 2：如果发现新的字符已经在集合里已经出现了，怎么办？

deabc 是目前为止没有重复字符的最长子串，当我们遇到下一个字符 a 的时候，以这个字符结尾的没有重复的子串是“bca”，而此时集合里的字符有：d，e，a，b，c。首先，必须把 a 删除，因为这样才能把新的 a 加入到集合里，那么如何判断要把 d 和 e 也都删除呢？



1. 可以定义两个指针 i 和 j。
2. i 是慢指针，j 是快指针，当 j 遇到了一个重复出现的字符时，从慢指针开始一个一个地将 i 指针指向的字符从集合里删除，然后判断一下是否可以把新字符加入到集合里而不会产生重复。

3. 把字符 d 删除后，i 指针向前移动一步，此时集合里还剩下：e, a, b, c，很明显，字符 a 还在集合里，仍然要继续删除。
4. 把字符 e 删除后，集合里还剩 a，b，c，字符 a 还在集合里，继续删除慢指针 i 指向的字符 a。
5. 集合里剩 b，c，可以放心地把新的字符 a 放入到集合里，然后快指针 j 往前移动一步。

通过这样不断尝试，每当新的字符加入到集合里的时候，统计一下当前集合里的元素个数，最后记录下最长的那个。

时间复杂度

由于采用的是快慢指针的策略，字符串最多被遍历两次，快指针遇到的字符会被添加到哈希集合，而慢指针遇到的字符会从哈希集合里删除，对哈希集合的操作都是 O(1) 的时间复杂度，因此，整个算法的时间复杂度就是 n×O(1) + n×O(1) = O(n)。

空间复杂度

由于用到了一个哈希集合，在最坏的情况下，给定的字符串没有任何重复的字符，需要把每个字符都加入到哈希集合里，因此空间复杂度是 O(n)。

代码实现

```
// 定义一个哈希集合 set，初始化结果 max 为 0
int lengthOfLongestSubstring(String s) {
      Set<Character> set = new HashSet<>();
      int max = 0;
```

```
        // 用快慢指针 i 和 j 扫描一遍字符串，如果快指针所指向的字符已经出现在哈希集合里，不断地尝试将慢指针所指向的字符从哈希集合里删除
        for (int i = 0, j = 0; j < s.length(); j++) {
            while (set.contains(s.charAt(j))) {
                set.remove(s.charAt(i));
                i++;
            }

            // 当快指针的字符加入到哈希集合后，更新一下结果 max
            set.add(s.charAt(j));
            max = Math.max(max, set.size());
        }
        return max;
    }
```

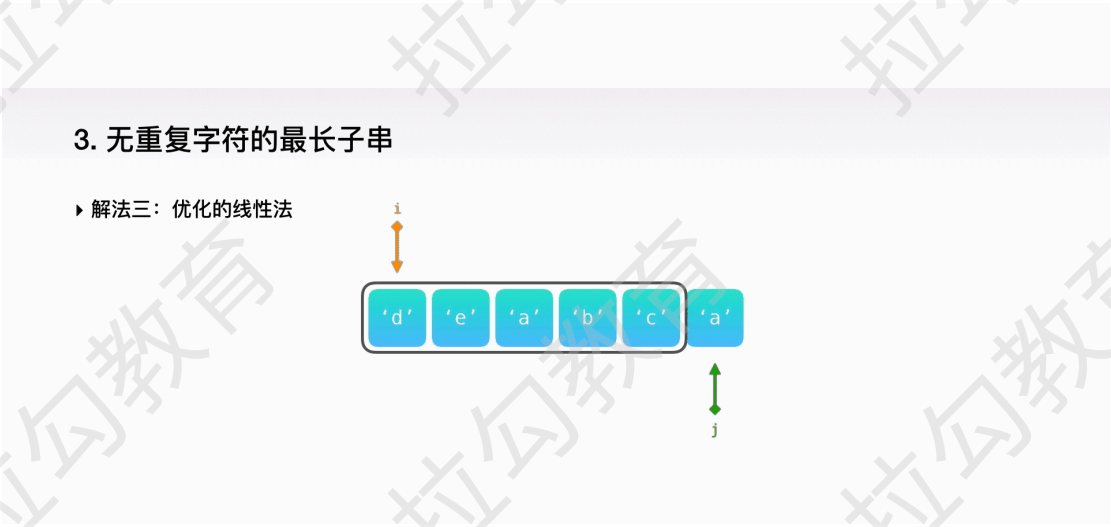
解题思路三：优化的线性法

在上述例题中，能否让慢指针不再一步一步地挪动，而是迅速地跳到字符 b 的位置？

可以用哈希表来记录每个字符以及它出现的位置，当遇到了字符 a 的时候，就知道跟它重复的前一个字符出现的位置，只需要让慢指针指向那个位置的下一个即可。（如果题目说所有字符都是字母的话，也可以用一个数组去记录。）

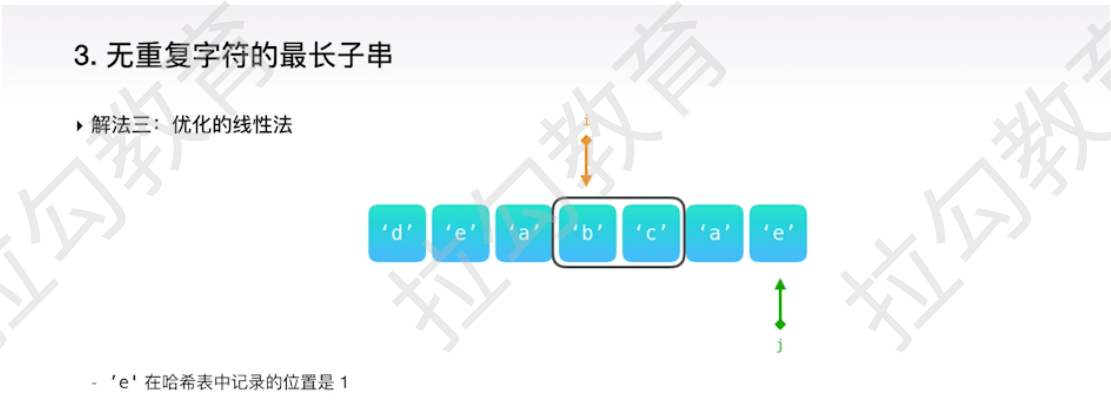
遇到字符 a，此时哈希表的记录 {d: 0, e: 1, a: 2, b: 3: c: 4}，a 的位置是 2，把 2 加上 1 等于 3，就能让慢指针 i 指向下标为 3 的位置，即 b 字符的地

方。



注意：在运用这个算法的时候，不能去数哈希集合的元素个数来作为子串的长度，所以得额外维护一个变量来保存最后的结果。

但是在一些情况下，我们不能简单地将取出来的重复位置加 1，如下：快指针 j 指向的字符是 e，而 e 在哈希表里记录的位置是 1。



在这种情况下，没有必要让 i 重新指向 e 后面的 a。此时，i 应该保留在原地不动。因此，i 被移动到的新位置应该等于 max(i，重复字符出现位置 + 1)。

代码实现

```
//  定义一个哈希表用来记录上一次某个字符出现的位置，并初始化结果 max 为 0
int lengthOfLongestSubstring(String s) {
    Map<Character, Integer> map = new HashMap<>();
    int max = 0;

        // 用快慢指针 i 和 j 扫描一遍字符串，若快指针所对应的字符已经出现过，则慢指针跳跃
        for (int i = 0, j = 0; j < s.length(); j++) {
        if (map.containsKey(s.charAt(j))) {
            i = Math.max(i, map.get(s.charAt(j)) + 1);
        }

        map.put(s.charAt(j), j);
           max = Math.max(max, j - i + 1);
    }

    return max;
}
```

例题分析二

LeetCode 第 04 题：给定两个大小为 m 和 n 的有序数组 nums1 和 nums2。请你找出这两个有序数组的中位数，并且要求算法的时间复杂度为 O(log(m+n))。你可以假设 nums1 和 nums2 不会同时为空。

示例 1

nums1 = [1, 3]

nums2 = [2]

则中位数是 2.0

示例 2

nums1 = [1, 2]

nums2 = [3, 4]

则中位数是 (2 + 3)/2 = 2.5

解题思路一：暴力法

因为两个数组都是排好序的，可以利用归并排序将它们合并成一个长度为 m+n 的有序数组，合并的时间复杂度是 m+n，然后从中选取中位数，整体的时间复杂度就是 O(m+n)。

这是比较直观的解法，但是比题目要求的 O(log(m+n)) 慢了许多，并不适合。

解题思路二：切分法

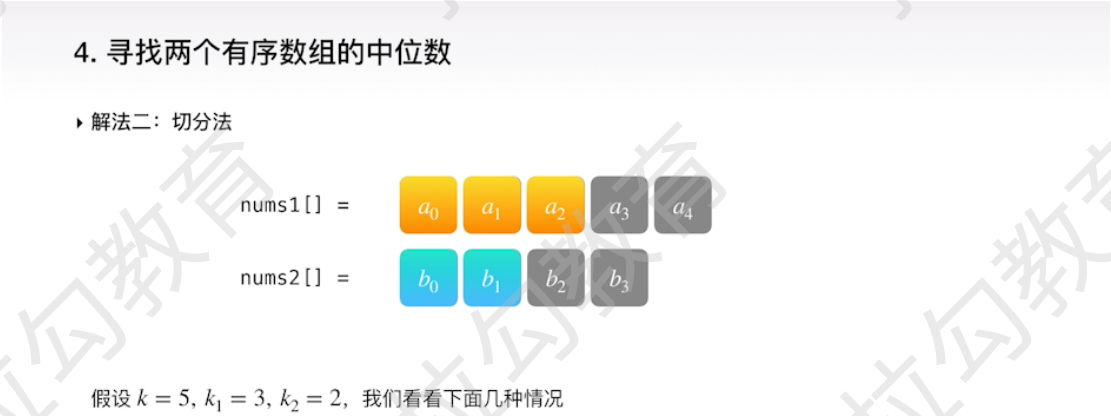
假设 m+n = L，若 L 为奇数，即两个数组的元素总个数为奇数，那么它们的中位数就是第 int(L / 2) + 1 小的数。例如，数组 { 1, 2, 3 } 的中位数是 2，2 就是第二小的数 2 = int(3/2) + 1。

如果 L 是偶数，那么中位数就是第 int(L/2) 小与第 int(L/2)+1 小的数的和的平均值。例如，数组 {1, 2, 3, 4} 的中位数是 (2 + 3) / 2 = 2.5，其中，2 = int(4/2)，3 = int(4/2) + 1。

因此这个问题就转变为在两个有序数组中寻找第 k 小的数 f(k)，当 L 是奇数的时候，另 k = L/2，结果为 f(k + 1)；而当 L 是偶数的时候，结果为 f(k) + f(k + 1) / 2。

如何从两个排好序的数组里找出第 k 小的数？

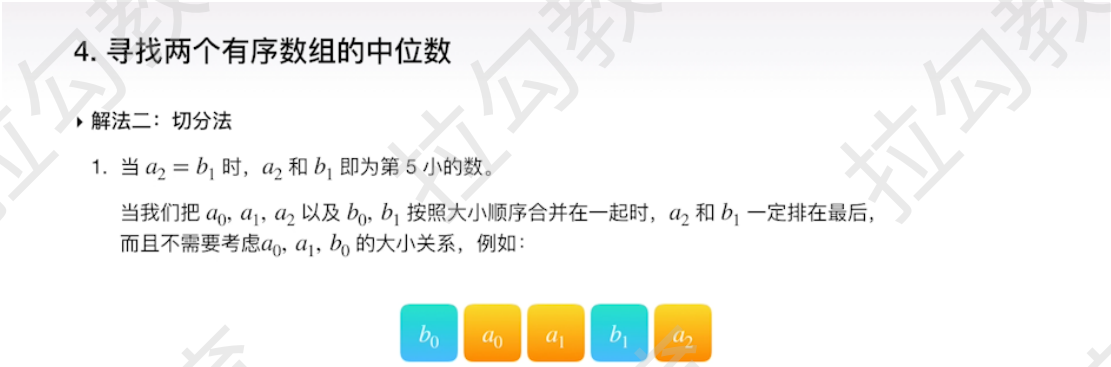
假设我们从第一个数组里前面 k_1 个数，从第二个数组里取出前面 k_2 个数，如下图。



假设 k = 5，k_1 = 3，k_2 = 2，有下面几种情况。

1. 当 $a_2 = b_1$ 时，可以肯定 a_2 和 b_1 就是第 5 小的数。

因为当把 a_0、a_1、a_2 以及 b_0、b_1 按照大小顺序合并在一起的时候，a_2 和 b_1 一定排在最后面，完全不需要考虑 a_0、a_1 和 b_0 的大小关系。其中一种可能的排列如下。



2. 当 $a_2 < b_1$ 的时候，无法肯定 a_2 和 b_1 是不是第 5 小的数。举例如下。

而最终第 5 小的数是 a_3 5 这个数。因此，在这种情况下，我们不能得出第 5 小的数是哪个。

但是，在这种情况下，至少我们可以肯定的是，我们要找的结果肯定不会在 a_0，a_1，a_2 之间，即不会出现在 nums1 数组的前半段里。为什么呢？很简单，因为如果第 5 小的数是 a_0，a_1，a_2 其中一个的话，意味着 k_1+k_2 必然大于 5，这就跟我们的假设不符了。

那么结果会不会在 nums2 的后半段呢？不可能，加入第 5 小的数在 nums2 的后半段，那么意味着，这个数要大于 b_1（即 7），也会大于 a_2（即 3），但是 $k_1 + k_2$ 已经等于 5 了，所以就和假设冲突了。

4. 寻找两个有序数组的中位数

▸ 解法二：切分法

在这样的情况下，我们可以把搜索的范围缩小，从 nums1 的后半段以及 nums2 的前半段中继续寻找。

1. 当 $a_2 > b_1$ 的时候，无法肯定 a_2 和 b_1 是不是第 5 小的数。举例如下。

nums1[] = {5, 6, **7**, 8, 9}

nums2[] = {1, **2**, 3, 4}

$a_2 = 7$，$b_1 = 2$

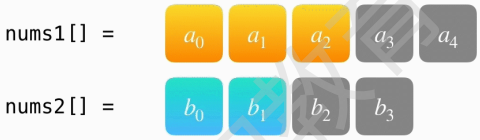
而最终第 5 小的数是 a_0 5 这个数。因此，在这种情况下，我们也不能得出第 5 小的数是哪个。

但是，在这种情况下，至少我们可以肯定的是，我们要找的结果肯定不会是 b0，或者
nums2 数组的前半段里。为什么呢？因为如果第 5 小的数是 b0 的话，意味着 k1+k2 必然大
于 5，这也跟我们的假设不符了。同样的，结果也不可能在 nums1 的后半段里。

在这样的情况下，我们可以把搜索的范围缩小，从 nums2 的后半段以及 nums1 中继续寻找。

4. 寻找两个有序数组的中位数

▸ 解法二：切分法



代码实现

```
double findMedianSortedArrays(int nums1[], int nums2[]) {
    int m = nums1.length;
    int n = nums2.length;

    int k = (m + n) / 2;

    if ((m + n) % 2 == 1) {
        return findKth(nums1, 0, m - 1, nums2, 0, n - 1, k
  + 1);
     } else {
        return (
            findKth(nums1, 0, m - 1, nums2, 0, n - 1, k) +

                        findKth(nums1, 0, m - 1, nums2, 0, n
-  1,  k  +  1)
        ) / 2.0;
    }
}

double findKth(int[] nums1, int l1, int h1, int[] nums2, int
  l2, int h2, int k) {
    int m = h1 - l1 + 1;
    int n = h2 - l2 + 1;

    if (m > n) {
        return findKth(nums2, l2, h2, nums1, l1, h1, k);
```

```
        }

        if (m == 0) {
            return nums2[l2 + k - 1];
        }

        if (k == 1) {
            return Math.min(nums1[l1], nums2[l2]);
        }

        int na = Math.min(k/2, m);
        int nb = k - na;
        int va = nums1[l1 + na - 1];
        int vb = nums2[l2 + nb - 1];

        if (va == vb) {
            return va;
        } else if (va < vb) {
            return findKth(nums1, l1 + na, h1, nums2, l2, l2 + nb - 1, k - na);
        } else {
            return findKth(nums1, l1, l1 + na - 1, nums2, l2 + nb, h2, k - nb);
        }

    }
```

1. 主体函数其实就是根据两个字符串长度的总和进行判断，看看如何调用递归函数以及返回结果。当总长度是奇数的时候，返回正中间的那个数；当总长度是偶数的时候，返回中间两个数的平均值。
2. 进入 findkth 函数，这个函数的目的是寻找第 k 小的元素。
3. 如果 nums1 数组的长度大于 nums2 数组的长度，我们将它们互换一下，这样可以让程序结束得快一些。
4. 当 nums1 的长度为 0 时，直接返回 nums2 数组里第 k 小的数。当 k 等于 1 的时候，返回两个数组中的最小值。

5. 接下来，分别选两个数组的中间数。
6. 比较一下两者的大小，如果相等，表明我们找到了中位数，返回它；如果不等的话，我们进行剪枝处理。

算法分析

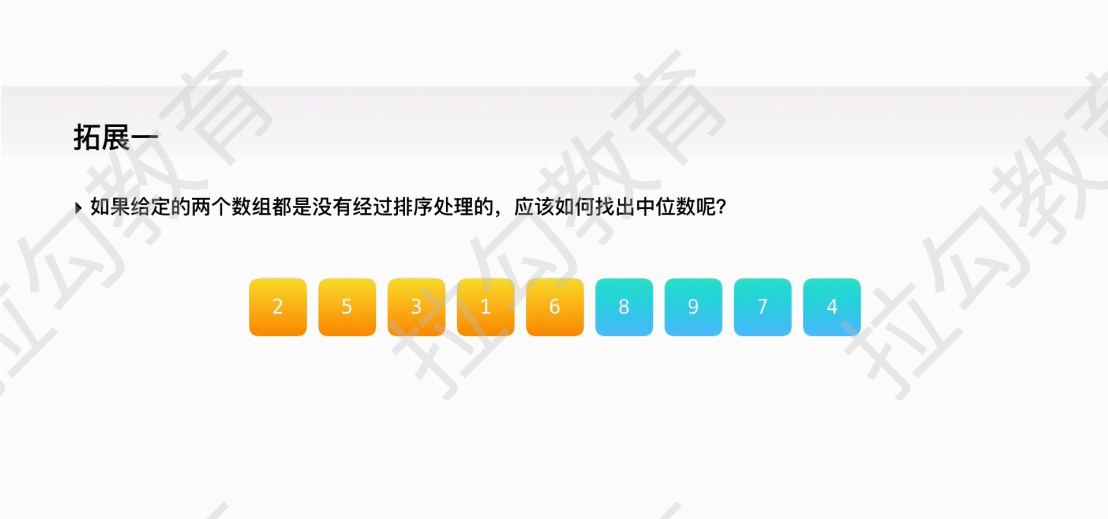
由于要求中位数，即 k = (m+n) / 2，k1 = k / 2，k2 = k / 2，每次都能将一半的数排除，即问题的规模减小一半，因此，算法复杂度就类似二分搜索，复杂度就是 log(k)，即 O(log((m+n) / 2))。

扩展一

例题：如果给定的两个数组是没有经过排序处理的，应该怎么找出中位数呢？

解法 1：直观方法

先将两个数组合并在一起，然后排序，再选出中位数。时间复杂度是：O((m+n)× og(m+n))。



解法 2：快速选择算法

快速选择算法，可以在 O(n) 的时间内从长度为 n 的没有排序的数组中取出第 k 小的数，运用了快速排序的思想。

假如将 nums1[] 与 nums2[] 数组组合成一个数组变成 nums[]：{2, 5, 3, 1, 6, 8, 9, 7, 4}，那么如何在这个没有排好序的数组中找到第 k 小的数呢？

1. 随机地从数组中选择一个数作为基准值，比如 7。一般而言，随机地选择基准值可以避免最坏的情况出现。

拓展一

1. 随机地从数组中选择一个数作为基准值。

2. 将数组排列成两个部分，以基准值作为分界点，左边的数都小于基准值，右边的都大于基准值。

拓展一

1. 随机地从数组中选择一个数作为基准值。
   一般而言，随机选择基准值可以避免最快的情况出现。

3. 判断一下基准值所在位置 p：

a. 如果 p 刚好等于 k，那么基准值就是所求数，直接返回。

b. 如果 k < p，即基准值太大，搜索的范围应该缩小到基准值的左边。

c. 如果 k > p，即基准值太小，搜索的范围应该缩小到基准值的右边。此时需要找的应该是第 k - p 小的数，因为前 p 个数被淘汰。

拓展一

3. 判断基准值所在位置 p

4. 重复第一步，直到基准值的位置 p 刚好就是要找的 k。

代码实现

```
public int findKthLargest(int[] nums, int k) {
    return quickSelect(nums, 0, nums.length - 1, k);
}

// 随机取一个基准值，这里取最后一个数作为基准值
int quickSelect(int[] nums, int low, int high, int k) {
    int pivot = low;

        // 比基准值小的数放左边，把比基准值大的数放右边
        for (int j = low; j < high; j++) {
      if (nums[j] <= nums[high]) {
          swap(nums, pivot++, j);
      }
    }
    swap(nums, pivot, high);
```

```
        //  判断基准值的位置是不是第  k  大的元素
        int  count  =  high  -  pivot  +  1;
    //  如果是，就返回结果。
        if  (count  ==  k)  return  nums[pivot];
      //  如果发现基准值小了，继续往右边搜索
        if  (count  >  k)  return  quickSelect(nums,  pivot  +  1,  h
igh,  k);
    //  如果发现基准值大了，就往左边搜索
        return  quickSelect(nums,  low,  pivot  -  1,  k  -  count);

}
```

时间复杂度

时间复杂度为什么是 O(n)。分析如下。

为了方便推算，假设每次都选择中间的那个数作为基准值。

1. 设函数的时间执行函数为 T(n)，第一次运行的时候，把基准值和所有的 n 个元素进行比较，然后将输入规模减半并递归，所以 T(n) = T(n/2) + n。
2. 当规模减半后，新的基准值只和 n/2 个元素进行比较，因此 T(n/2) = T(n/4) + n/2。
3. 以此类推：

$$T(n/4) = T(n/8) + n/4$$

...

$$T(2) = T(1) + 2$$

$$T(1) = 1$$

将上面的公式逐个代入后得到 T(n) = 1 + 2 + … + n/8 + n/4 + n/2 + n = 2×n，所以 O(T(n)) = O(n)。

空间复杂度

如果不考虑递归对栈的开销，那么算法并没有使用额外的空间，swap 操作都是直接在数组里完成，因此空间复杂度为 O(1)。

解法 3：数组“组合”

把这两个数组“虚拟”地组合在一起，即它们是分开的，但是在访问它们的元素时，把它们看成是一个数组。那么就能运用快速选择的算法。

代码实现

```
double findMedianArrays(int[] nums1, int[] nums2) {
    int m = nums1.length;
    int n = nums2.length;

    int k = (m + n) / 2;

    return (m + n) % 2 == 1 ?
        findKthLargest(nums1, nums2, k + 1) :
        (findKthLargest(nums1, nums2, k) + findKthLargest(nums1,
  nums2, k + 1)) / 2.0;
}

double findKthLargest(int[] nums1, int[] nums2, int k) {
    return quickSelect(nums1, nums2, 0, nums1.length + nums2.le
ngth - 1, k);
}
```

```
double quickSelect(int[] nums1, int[] nums2, int low, int high, int k) {
    int pivot = low;

        // use quick sort's idea
    // put nums that are <= pivot to the left
    // put nums that are  > pivot to the right
    for (int j = low; j < high; j++) {
        if (getNum(nums1, nums2, j) <= getNum(nums1, nums2, high)) {
            swap(nums1, nums2, pivot++, j);
        }
    }
    swap(nums1, nums2, pivot, high);

    // count the nums that are > pivot from high
    int count = high - pivot + 1;
    // pivot is the one!
    if (count == k) return getNum(nums1, nums2, pivot);
    // pivot is too small, so it must be on the right
    if (count > k) return quickSelect(nums1, nums2, pivot + 1, high, k);
    // pivot is too big, so it must be on the left
    return quickSelect(nums1, nums2, low, pivot - 1, k - count);
}

int getNum(int[] nums1, int[] nums2, int index) {
    return (index < nums1.length) ? nums1[index] : nums2[index - nums1.length];
}

void swap(int[] nums1, int[] nums2, int i, int j) {
    int m = nums1.length;

    if (i < m && j < m) {
        swap(nums1, i, j);
    } else if (i >= m && j >= m) {
        swap(nums2, i - m, j - m);
    } else if (i < m && j >= m) {
        int temp = nums1[i];
        nums1[i] = nums2[j - m];
        nums2[j - m] = temp;
    }
```

```
}

void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
```

因为这道题的解法与之前讲的快速选择算法非常类似，差别在于将两个数组合在一起考虑。因此大家可以自己分析一下代码。

时间复杂度是 O(m+n)，空间复杂度 O(1)。

扩展二

例题：有一万个服务器，每个服务器上存储了十亿个没有排好序的数，现在要找所有数当中的中位数，怎么找？

对于分布式地大数据处理，应当考虑两个方面的限制：

1. 每台服务器进行算法计算的复杂度限制，包括时间和空间复杂度
2. 服务器与服务器之间进行通信时的网络带宽限制

限制 1：空间复杂度

假设存储的数都是 32 位整型，即 4 个字节，那么 10 亿个数需占用 40 亿字节，大约 4GB

- 归并排序至少得需要 4GB 的内存

- 快速排序的空间复杂度为 log(n)，即大约 30 次堆栈压入

用非递归的方法去实现快速排序，代码如下。

```
//  每次只需将数组中的某个起始点和终点，即一个范围，压入堆栈中，压入 30 个范围的大小约为 30×2×4=240 字节
class Range {
    public int low;
    public int high;

    public Range(int low, int high) {
        this.low = low;
        this.high = high;
    }
}

//  不使用递归写法，压入堆栈的还包括程序中的其他变量等，假设需要 100 字节，总共需要 30×100=3K 字节
void quickSort(int[] nums) {
    Stack<Range> stack = new Stack<>();

        Range range = new Range(0, nums.length - 1);
    stack.push(range);

    while (!stack.isEmpty()) {
        range = stack.pop();

        int pivot = partition(nums, range.low, range.high);

        if (pivot - 1 > range.low) {
            stack.push(new Range(range.low, pivot - 1));
        }

        if (pivot + 1 < range.high) {
            stack.push(new Range(pivot + 1, range.high));
        }
    }
}

//  快速排序对内存的开销非常小
int partition(int[] nums, int low, int high) {
        int pivot = randRange(low, high), i = low;
```

```
        swap(nums,  pivot,  high);

    for  (int  j  =  low;  j  <  high;  j++)  {
        if  (nums[j]  <=  nums[high])  {
            swap(nums,  i++,  j);
        }
    }

    swap(nums,  i,  high);

    return  i;
}
```

如上，利用一个栈 stack 来记录每次进行快速排序时的范围。一旦发现基准值左边还有未处理完的数，就将左边的范围区间压入到栈里；如果发现基准值右边还有未处理完的数，就将右边的范围区间压入到栈里。其中，处理基准值的 partition 函数非常重要，之前已经介绍过。

限制 2：网络带宽

在实际应用中，这是最重要的考量因素，很多大型的云服务器都是按照流量来进行收费，如何有效地限制流量，避免过多的服务器之间的通信，就是要考量的重点，并且，实际上它与算法的时间复杂度有很大的关系。

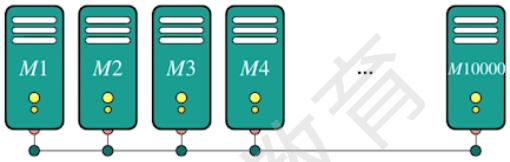
解决方案

借助扩展一的思路。

1. 从 1 万 个服务器中选择一个作为主机（master server）。这台主机将扮演主导快速选择算法的角色。

拓展二

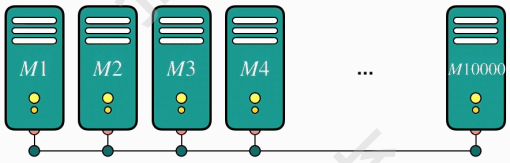
1. 从 10000 个服务器中选择一个作为主机 (master server)。这台主机将扮演主导快速选择算法的角色。



2. 在主机上随机选择一个基准值，然后广播到其他各个服务器上。

拓展二

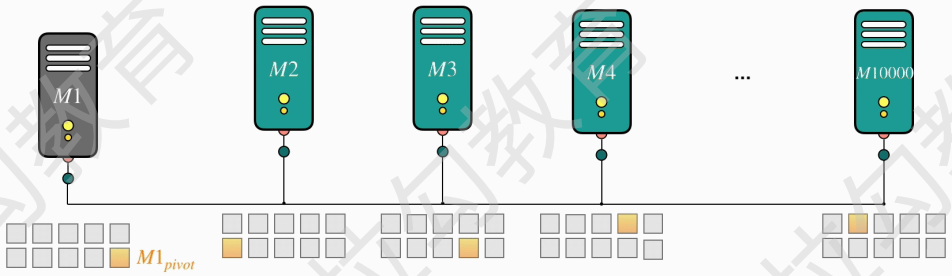
1. 从 10000 个服务器中选择一个作为主机 (master server)。这台主机将扮演主导快速选择算法的角色。



3. 每台服务器都必须记录下最后小于、等于或大于基准值数字的数量：less count，equal count，greater count。

拓展二

3. 每台服务器开始执行快速选择算法的操作，小于基准值的数放在数组左边，反之放在右边。



4. 每台服务器将 less count，equal count 以及 greater count 发送回主机。

5. 主机统计所有的 less count，equal count 以及 greater count，得出所有比基准值小的数的总和 total less count，等于基准值的总和 total equal count，以及大于基准值的总和 total greater count。进行如下判断。

a. 如果 total less count >= total count / 2，表明基准值太大。

b. 如果 total less count + total equal count >= total count / 2，表明基准值即为所求结果。

c. 否则，total less count + total equal count < total count / 2 表明基准值太小。

6. 后面两种情况，主机会把新的基准值广播给各个服务器，服务器根据新的基准值的大小判断往左半边或者右半边继续进行快速选择。直到最后找到中位数。

时间复杂度

整体的时间复杂度是 O(nlog(n))，主机和各个其他服务器之间的通信总共也需要 nlog(n)次，每次通信需要传递一个基准值以及三个计数值。

如果用一些组播网络（Multicast Network)，可以有效地节省更多的带宽。

例题分析三

LeetCode 第 23 题：合并 k 个排好序的链表，返回合并后的排序链表。分析和描述算法的复杂度。

示例

输入：

```
[
 1 -> 4 -> 5,
 1 -> 3 -> 4,
  2 -> 6
]
```

输出：1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6

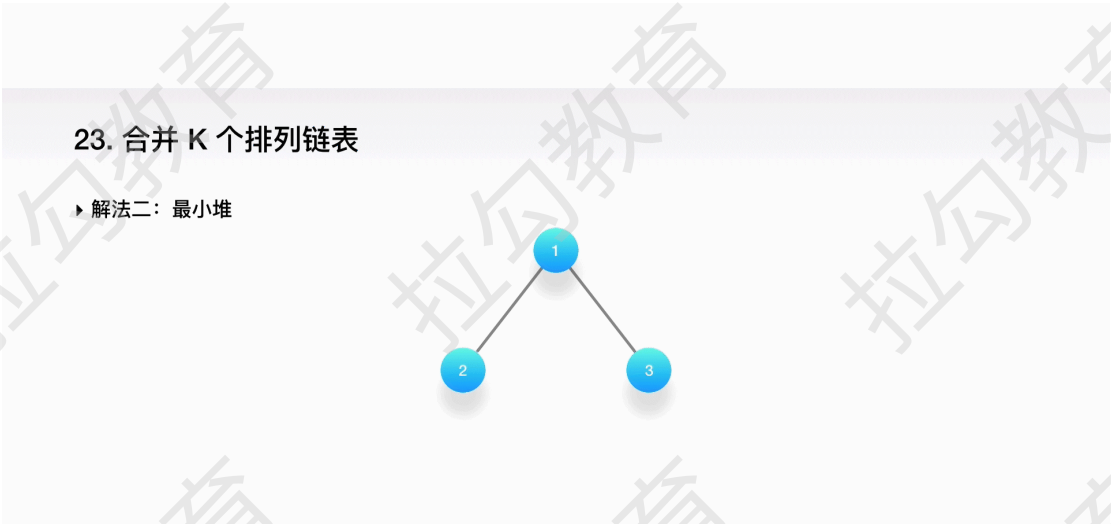
解题思路一：暴力法

用一个数组保存所有链表中的数，然后对这个数组进行排序，再从头到尾将数组遍历一遍，生成一个排好序的链表。假设每个链表的平均长度为 n，整体的时间复杂度就是 O(nk×log(nk))。

解题思路二：最小堆法

面对 k 个排好序的链表时，最小的那个数肯定是从这 k 个链表的头里面选出来。

那么，第二小的如何选择？例如，有下面 k 个链表。



1. 把最小的 1 从所有的 k 个链表头里选出来之后，把 1 从链表里删掉。
2. 下一个最小的数，还是从所有的 k 个链表头里选出来。
3. 以此类推，每一轮都比较 k 个新的链表头的大小，得出最后的结果。

上述操作的时间复杂度是 O(k)。而针对找出最小的数，可以使用最小堆来提高效率。时间复杂度计算如下。

1. 对 k 个链表头创建一个大小为 k 的最小堆，在第 2 课中提到创建一个大小为 k 的最小堆所需的时间是 O(k)；
2. 从堆里取出最小的数，都是 O(lg(k))；
3. 若每个链表的平均长度为 n，一共有 nk 个元素，即用大小为 k 的最小堆去过滤 nk 个元素；
4. 整体的时间复杂度就是 O(nk×log(k))。

维护这个大小为 k 的最小堆，直到遍历完所有 k 个链表里的所有元素。

代码实现

```
public ListNode mergeKLists(ListNode[] lists) {
    //利用一个空的链表头方便插入节点。
        ListNode fakeHead = new ListNode(0), p = fakeHead;
    int k = lists.length;

    // 定义一个最小堆来保存 k 个链表节点；将 k 个链表的头放入到最小堆里。
        PriorityQueue<ListNode> heap =
```

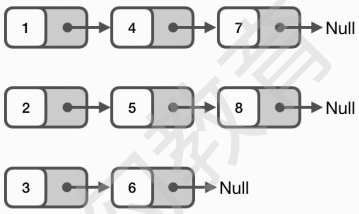
```
            new PriorityQueue<>(k, new Comparator<ListNode>()
{
            public int compare(ListNode a, ListNode b) {
                return a.val - b.val;
            }
        });

        // 从最小堆里将当前最小的节点取出，插入到结果链表中。
        for (int i = 0; i < k; i++) {
        if (lists[i] != null) {
            heap.offer(lists[i]);
        }
    }

        while (!heap.isEmpty()) {
        ListNode node = heap.poll();

        p.next = node;
        p = p.next;

        // 如果发现该节点后面还有后续节点，将后续节点加入到最小堆
里。
            if (node.next != null) {
                heap.offer(node.next);
        }
    }
    return fakeHead.next;

}
```

解题思路三：分治法

当 k=1 的时候，直接返回结果；当 k=2 的时候，把这两个链表归并。当 k=3 的时候，我们可以把它们分成两组，分别归并完毕后再进行最后的归并操作，如下。

23. 合并 K 个排列链表

▸ 解法三：分治法



上述做法运用了典型的分治思想，非常类似归并排序操作。

代码实现

```
public ListNode mergeKLists(ListNode[] lists, int low, int high) {
    if (low == high) return lists[low];

    int middle = low + (high - low) / 2; // 从中间切一刀

        return mergeTwoLists(
        mergeKLists(lists, low, middle),
        mergeKLists(lists, middle + 1, high)
    ); // 递归地处理左边和右边的链表，最后合并
}

public ListNode mergeTwoLists(ListNode a, ListNode b) {
    if (a == null) return b;
    if (b == null) return a;

    if (a.val <= b.val) {
        a.next = mergeTwoLists(a.next, b);
        return a;
```

```
        }

        b.next  =  mergeTwoLists(a,  b.next);
        return  b;
    }
```

合并两个排好序的链表非常简单，此处使用递归函数，可以尝试非递归写法。

时间复杂度：O(nk×log(k))。

空间复杂度：O(1)。因为不像最小堆解法那样需要维护一个额外的数据结构。

提示：因为这道题针对的是链表，所以很多操作都直接在链表上进行。