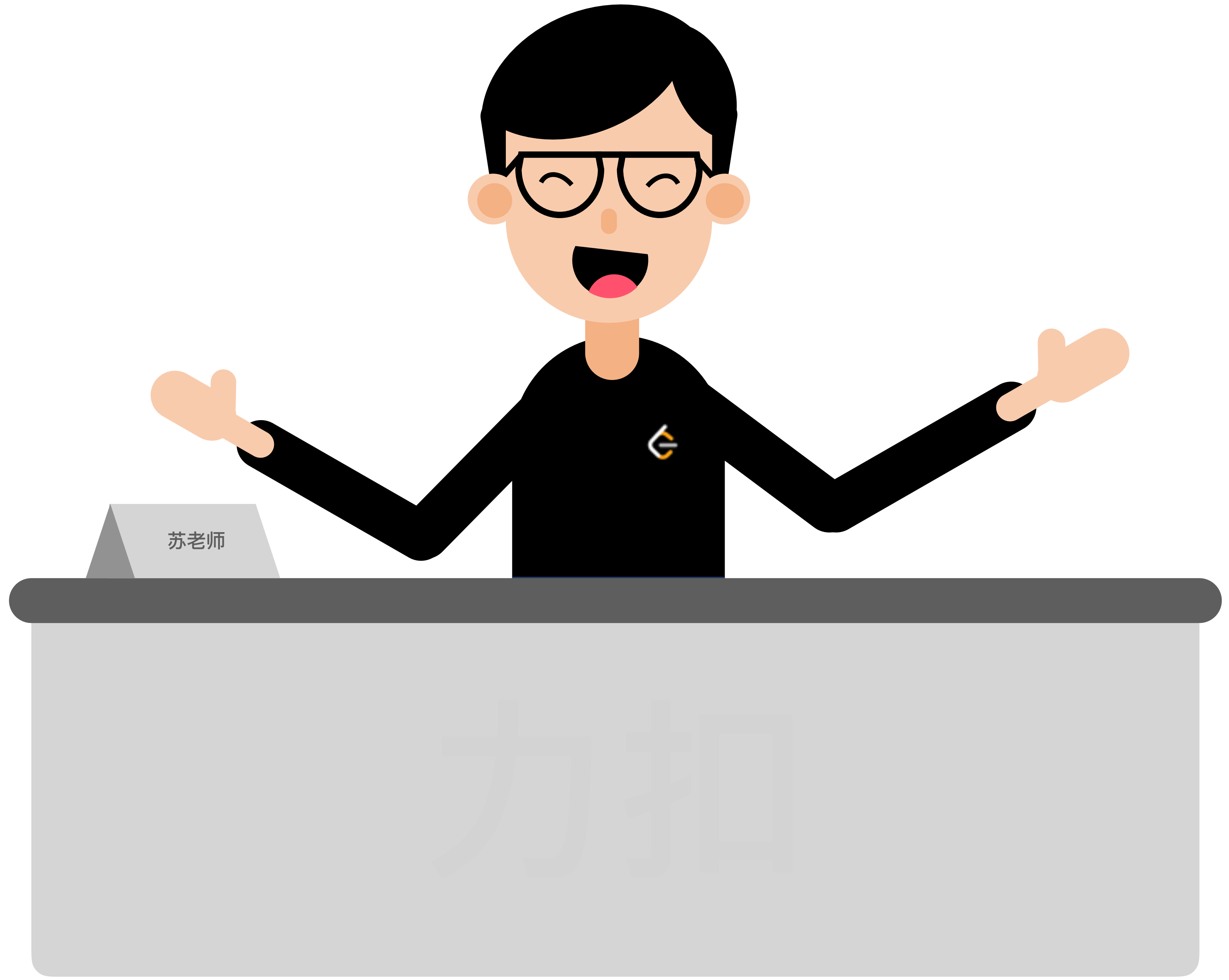
第八课

剖析大厂算法面试真题 - 高频题精讲（一）

8.0 简介 / Introduction

大厂面试高频题精讲

- 问题的剖析能力
- 寻找并分析解决问题的方案
- 代码的书写功底



3. 无重复字符的最长子串

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。
请注意，你的答案必须是 **子串** 的长度，"pwke" 是一个子序列，不是子串。

3. 无重复字符的最长子串

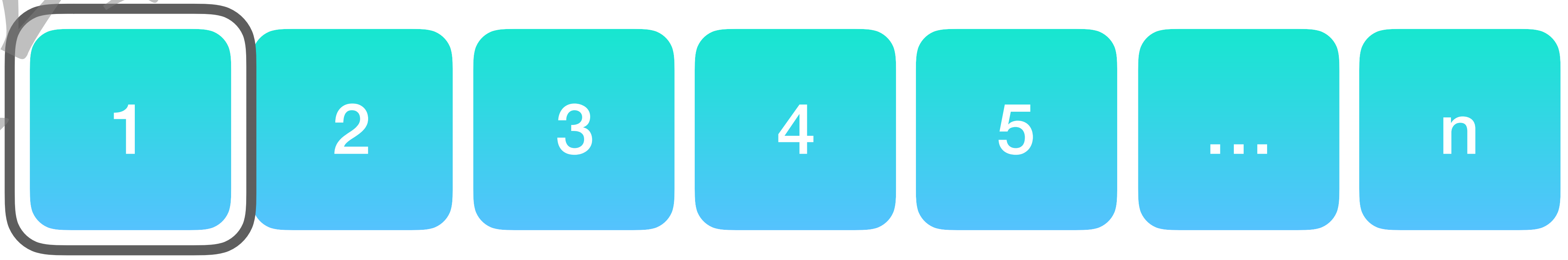
- **解法一：暴力法**

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

3. 无重复字符的最长子串

- **解法一：暴力法**

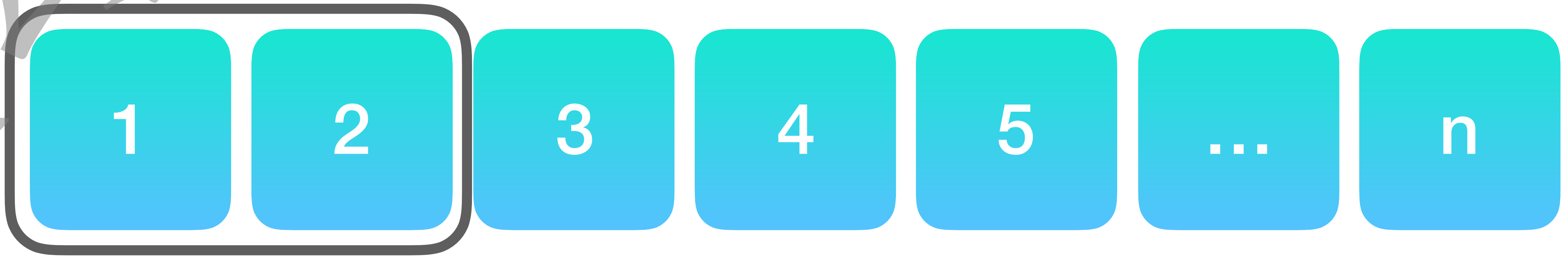
  - 长度为 1 的子串，有 n 个



3. 无重复字符的最长子串

解法一：暴力法

- 长度为 1 的子串，有 n 个
- 长度为 2 的子串，有 $n-1$ 个



3. 无重复字符的最长子串

▸ **解法一：暴力法**

- 长度为 1 的子串，有 n 个
- 长度为 2 的子串，有 $n-1$ 个
- 长度为 3 的子串，有 $n-2$ 个

  …

- 长度为 k 的子串，有 $n-k+1$ 个
- 当 $k=n$ 时，$n-k+1=1$，即长度为 n 的子串就是 1 个

3. 无重复字符的最长子串

▸ **解法一：暴力法**

所有情况全部相加，可得

$$n + (n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = \frac{n(n+1)}{2}$$

算上空字符，共有 $\frac{n(n+1)}{2} + 1$

3. 无重复字符的最长子串

长度为 n 的字符串，一共有多少子序列？

- 子序列不同于子串
- 子序列中的与元素不需要相互挨着

- 长度为 1 的子序列有 n 个，即 C_n^1
- 长度为 2 的子序列个数为 C_n^2

…

- 长度为 k 的序列有 C_n^k
- 所有子序列的个数（包括空序列）为：$C_n^0 + C_n^1 + C_n^2 + \ldots + C_n^n = 2^n$

3. 无重复字符的最长子串

▸ **解法一：暴力法**

如果对所有的子串进行判断，从每个子串里寻找最长且没有重复字符的，复杂度为：

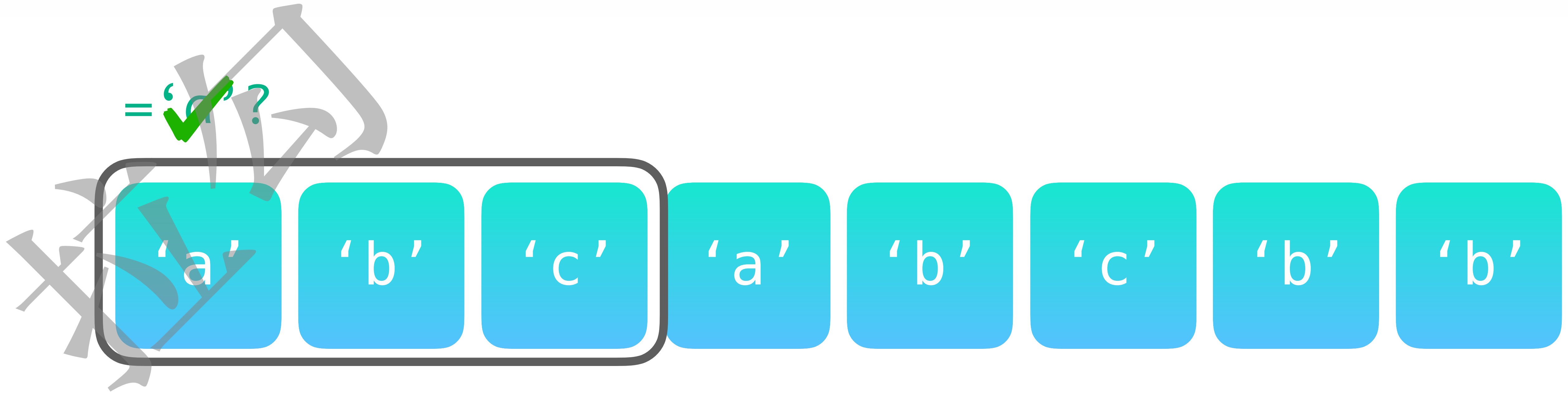
$$O(\frac{n(n+1)}{2} \times n) = O(n^3)$$

3. 无重复字符的最长子串

- **解法二：线性法**

  - 扫描 ‘abc’

  - 将 ‘abc’ 放入哈希集合，复杂度为 $O(1)$



3. 无重复字符的最长子串

- **解法二：线性法**

  - 定义一个哈希集合 set
  - 从给定字符串的头开始，每次检查当前字符是否在集合内
  - 如果不在，说明该字符不会造成重复和冲突
  - 将其加入到集合中，并统计当前集合长度，或许为最长子串

‘d’ ‘e’ ‘a’ ‘b’ ‘c’ ‘a’

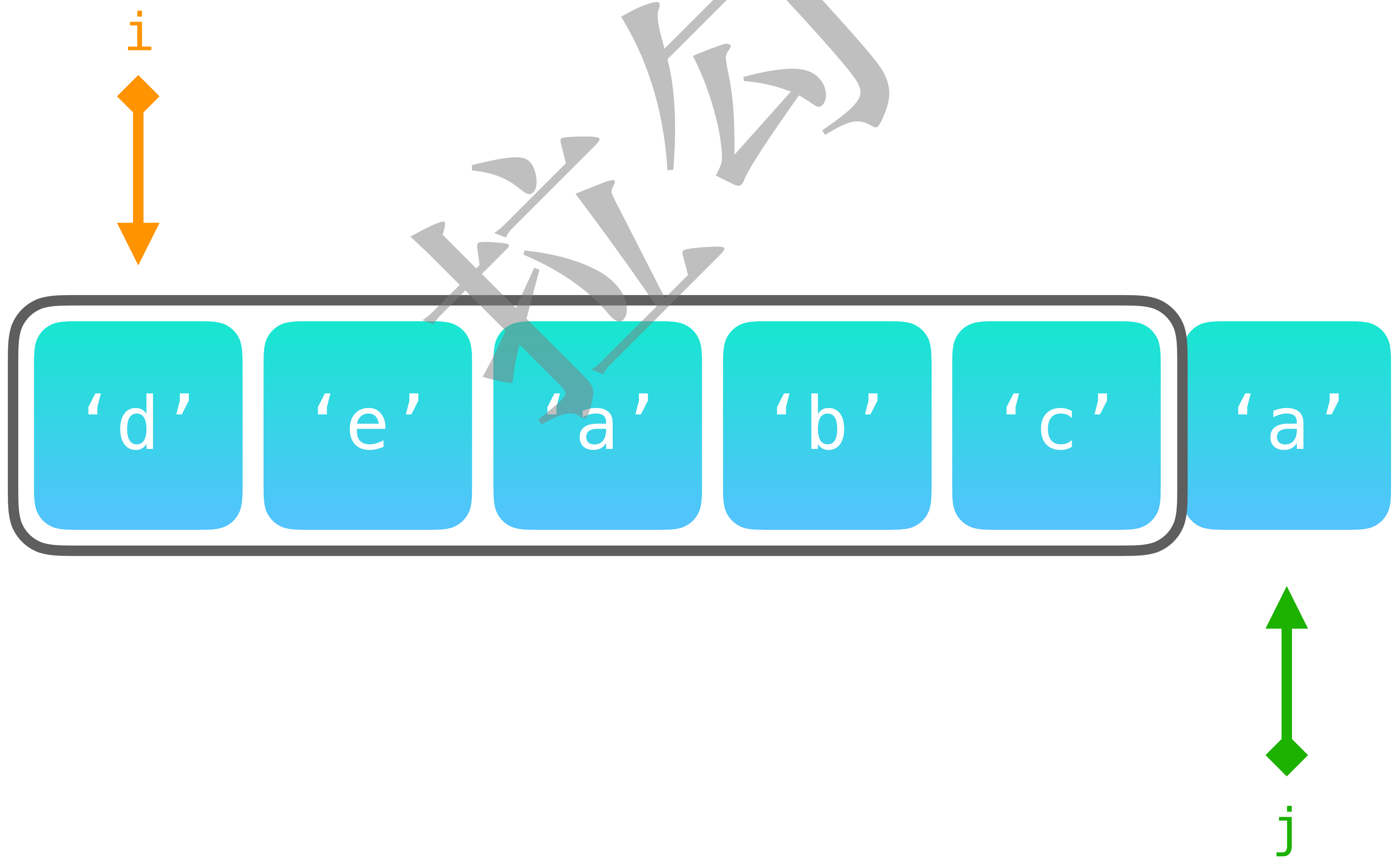
3. 无重复字符的最长子串

- 解法二：线性法

‘d’	‘e’	‘a’	‘b’	‘c’	‘a’

3. 无重复字符的最长子串

- **解法二：线性法**



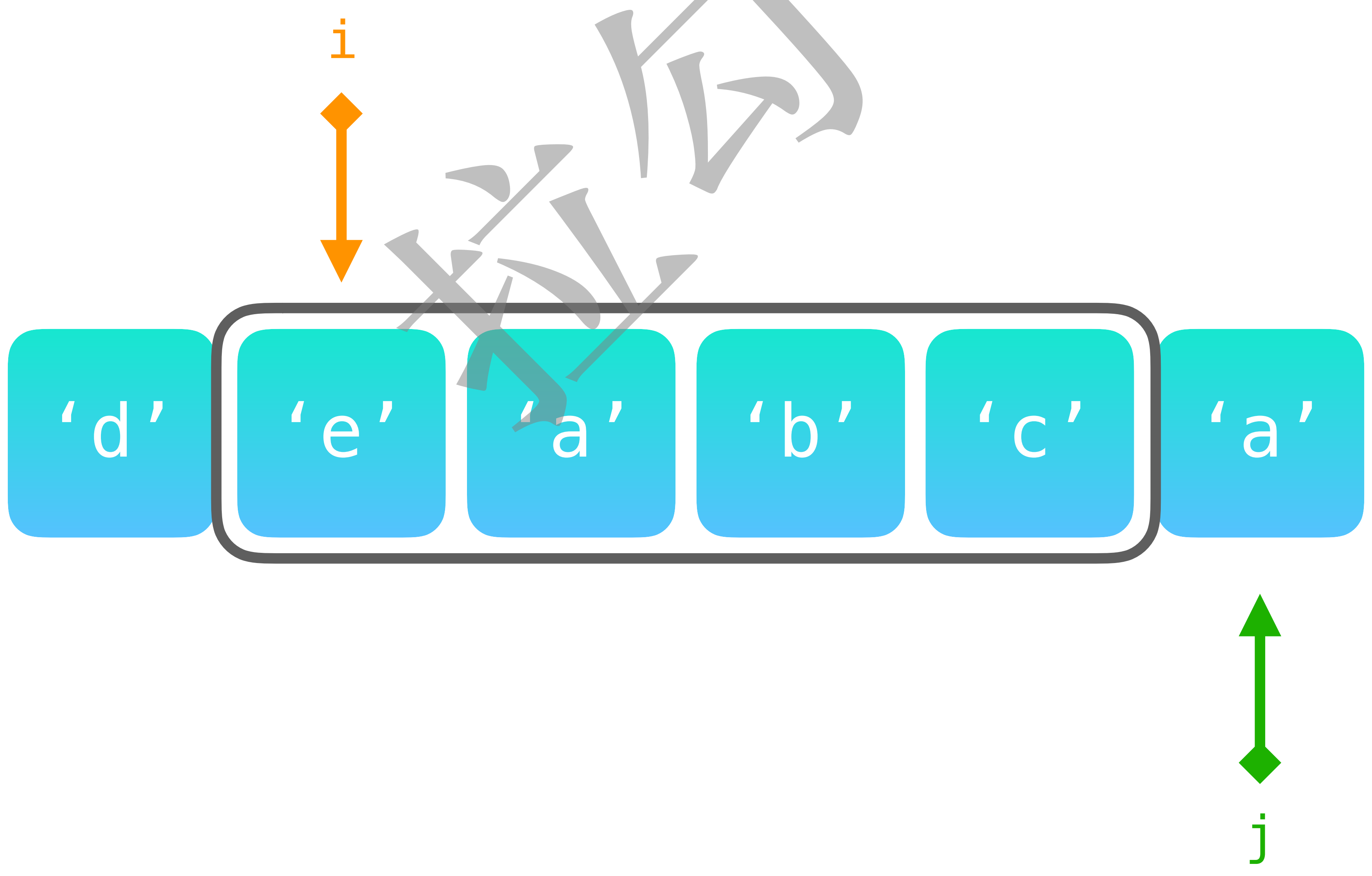
i 是慢指针 j 是快指针

当 j 遇到一个重复出现的字符时，我们从慢指针开始一个一个地将 i 指针指向的字符从集合中删除

然后判断是否可以把新字符加入到集合而不会重复

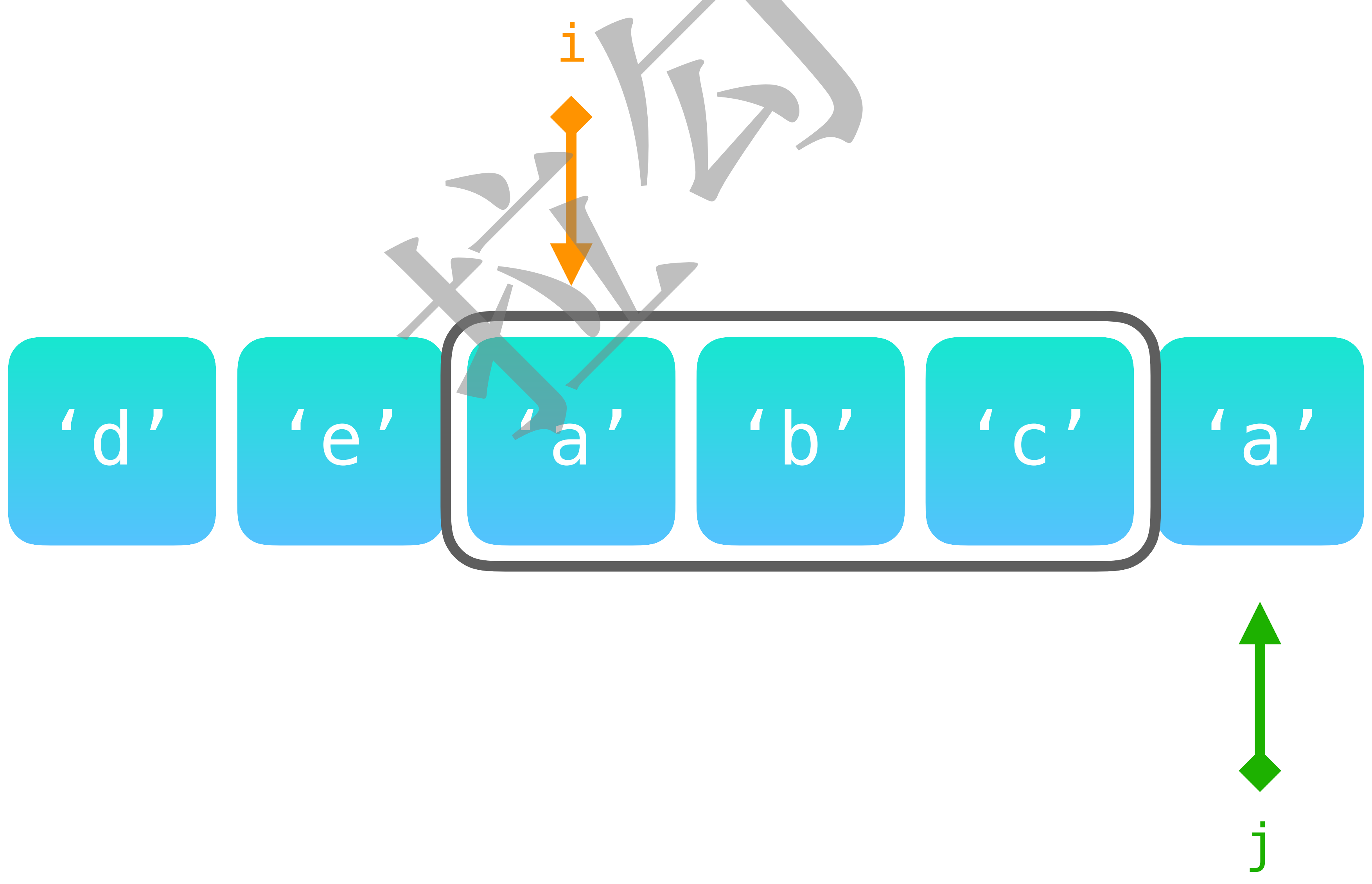
3. 无重复字符的最长子串

- 解法二：线性法



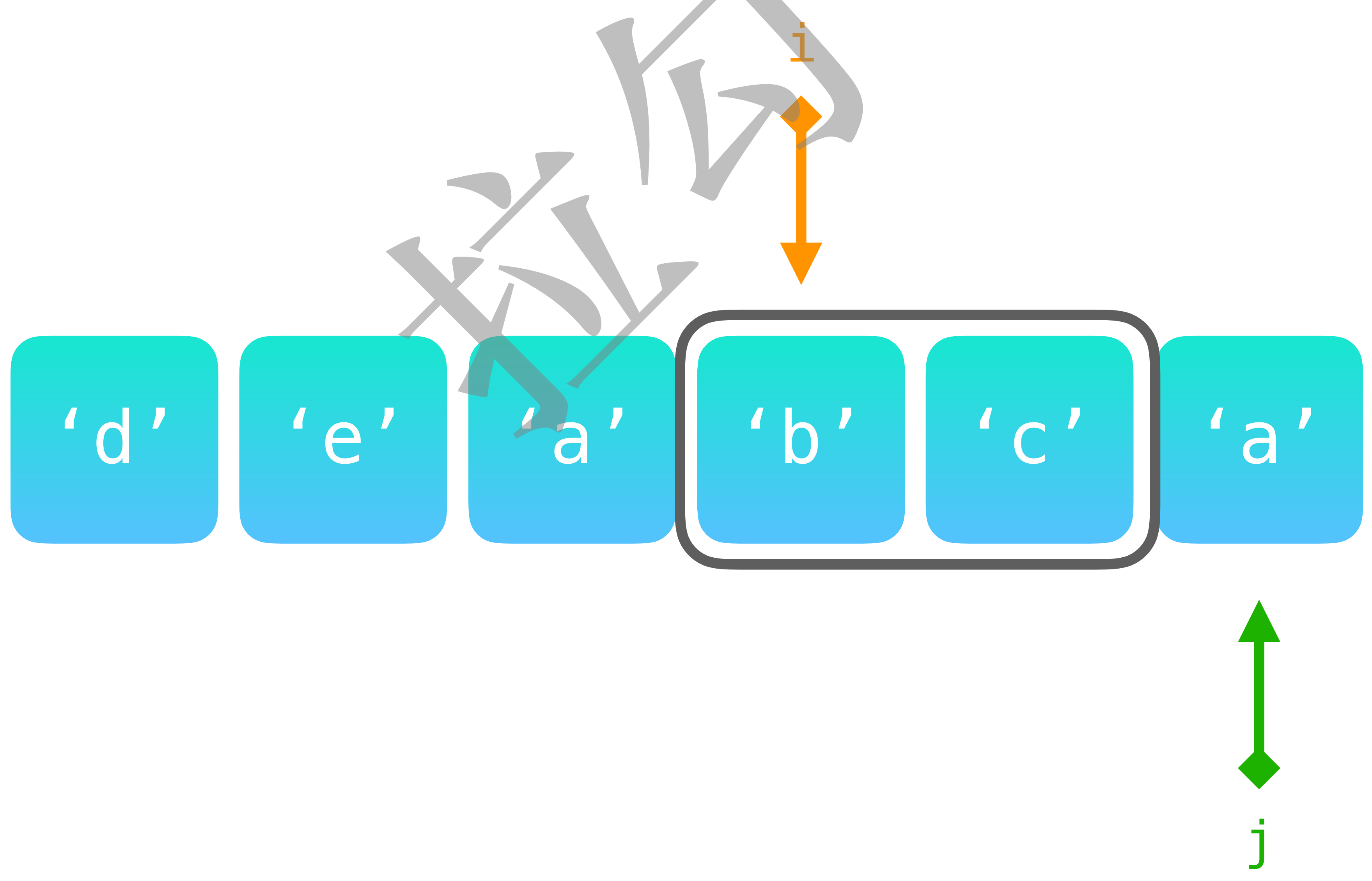
3. 无重复字符的最长子串

▸ 解法二：线性法



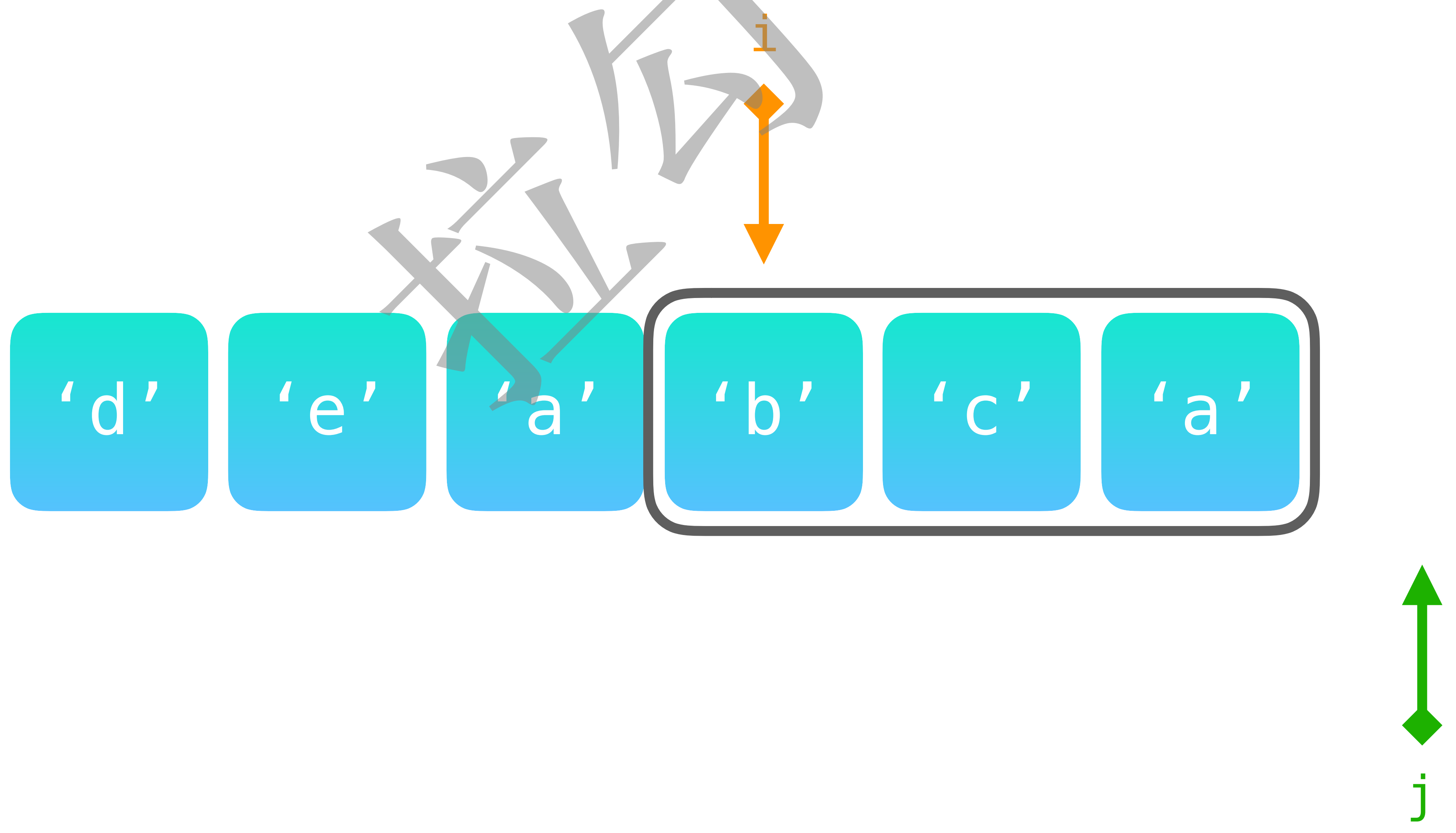
3. 无重复字符的最长子串

- 解法二：线性法



3. 无重复字符的最长子串

- 解法二：线性法



3. 无重复字符的最长子串

‣ 解法二：线性法

- 时间复杂度分析
    - 使用快慢指针策略，字符串最多被遍历两次
    - 快指针会被添加到哈希集合，慢指针遇到的字符会从哈希集合中删除
    - 哈希集合操作时间复杂度为 $O(1)$，因此整个算法复杂度为 $n \times O(1) + n \times O(1) = O(n)$

- 空间复杂度分析
    - 由于使用到哈希集合，最坏的情况下，即给定的字符串没有任何重复的字符，我们需要把每个字符都加入集合
    - 空间复杂度为 $O(n)$

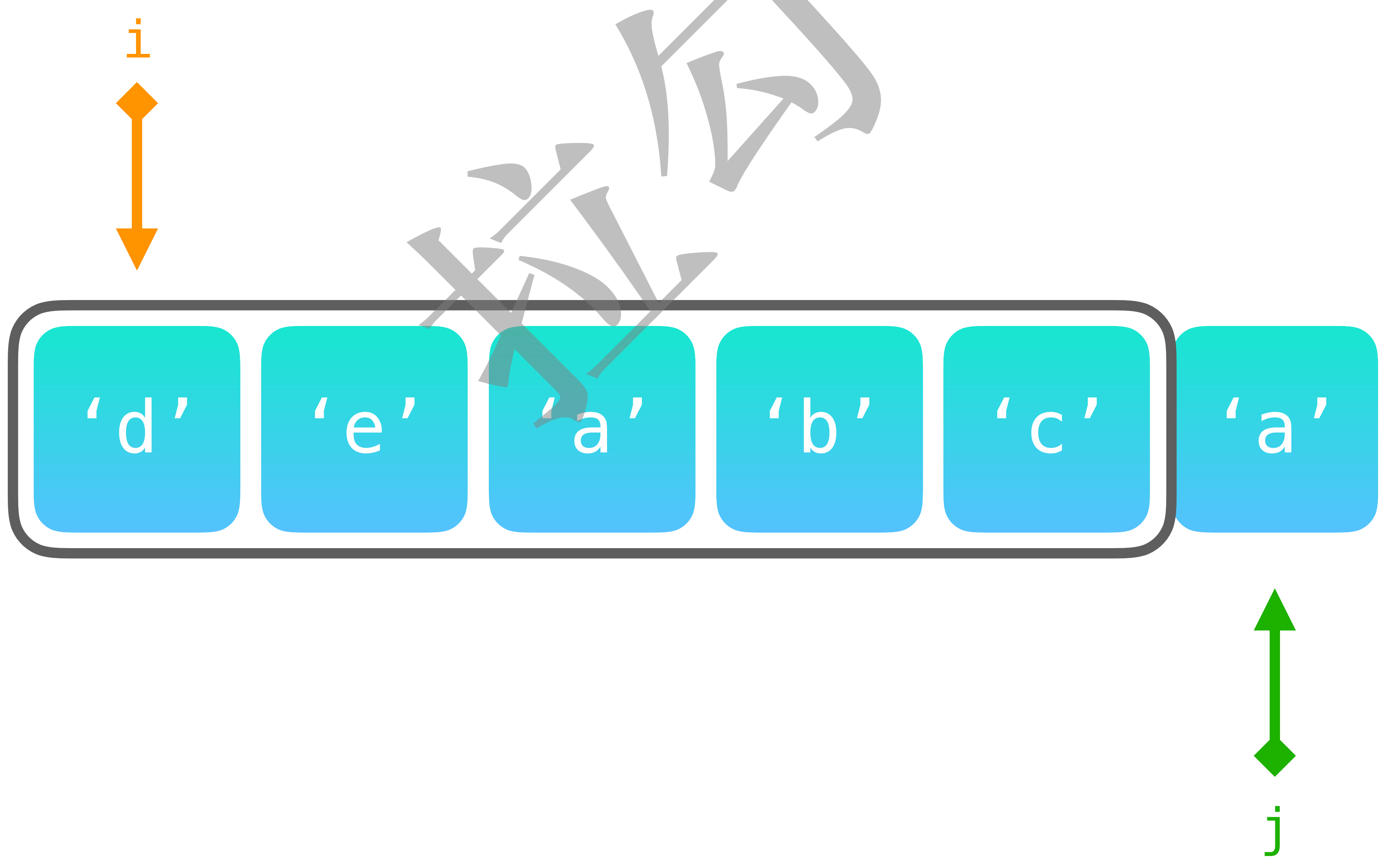
力扣 LeetCode × 拉勾 LAGOU.COM

```
int lengthOfLongestSubstring(String s) {
  Set<Character> set = new HashSet<>();
  int max = 0;

  for (int i = 0, j = 0; j < s.length(); j++) {
    while (set.contains(s.charAt(j))) {
      set.remove(s.charAt(i));
      i++;
    }

    set.add(s.charAt(j));
    max = Math.max(max, set.size());
  }

  return max;
}
```

- 定义一个哈希集合 set，初始化结果 max 为 0
- 利用快慢指针 i 和 j 扫描一遍字符串
- 如果快指针指向的字符已出现在哈希集合，不断尝试将慢指针指向的字符从哈希集合中删除
- 当快指针的字符终于能加入到哈希集合，更新结果 max
- 遍历完毕后，返回结果 max

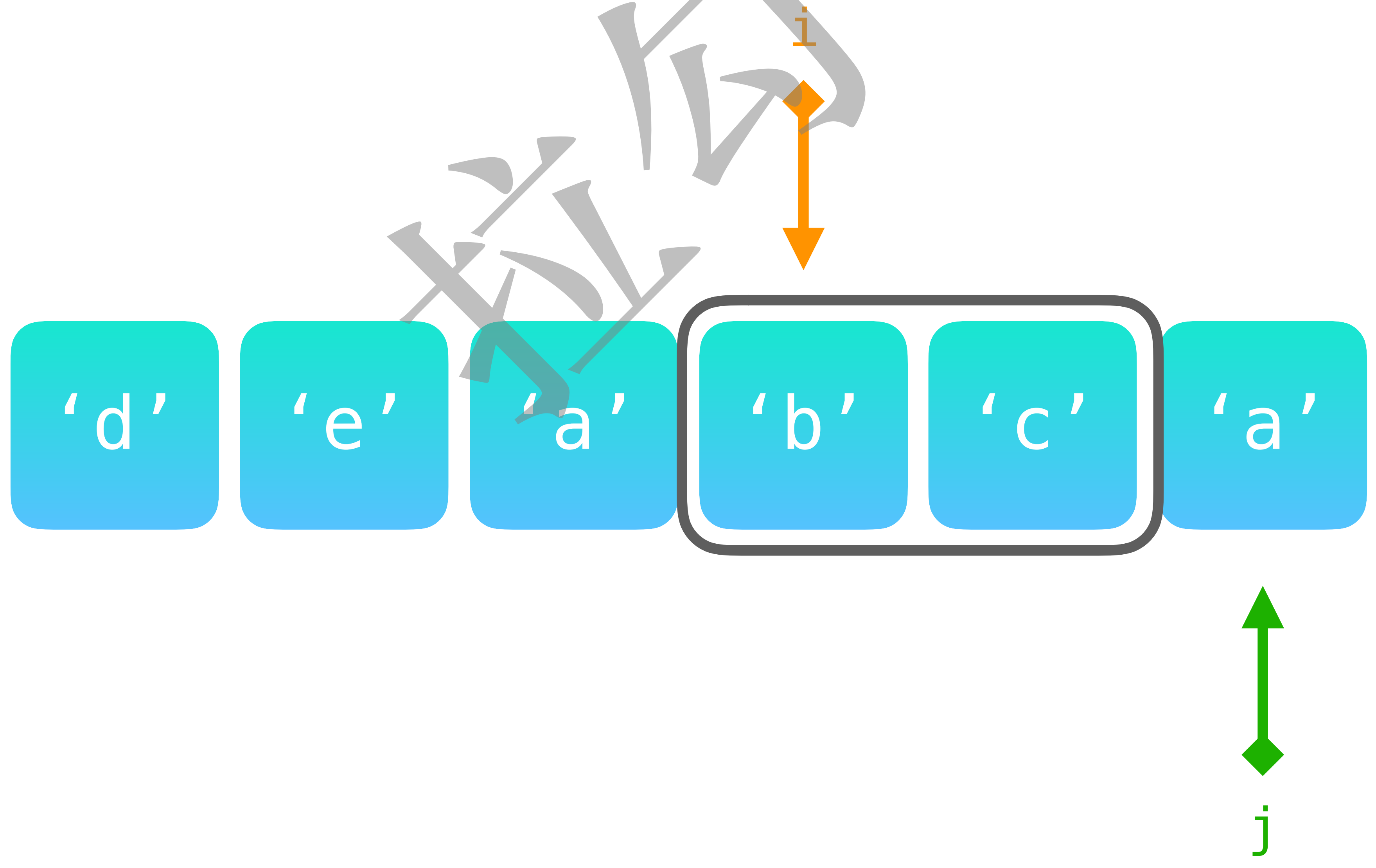
3. 无重复字符的最长子串

- 解法三：优化的线性法



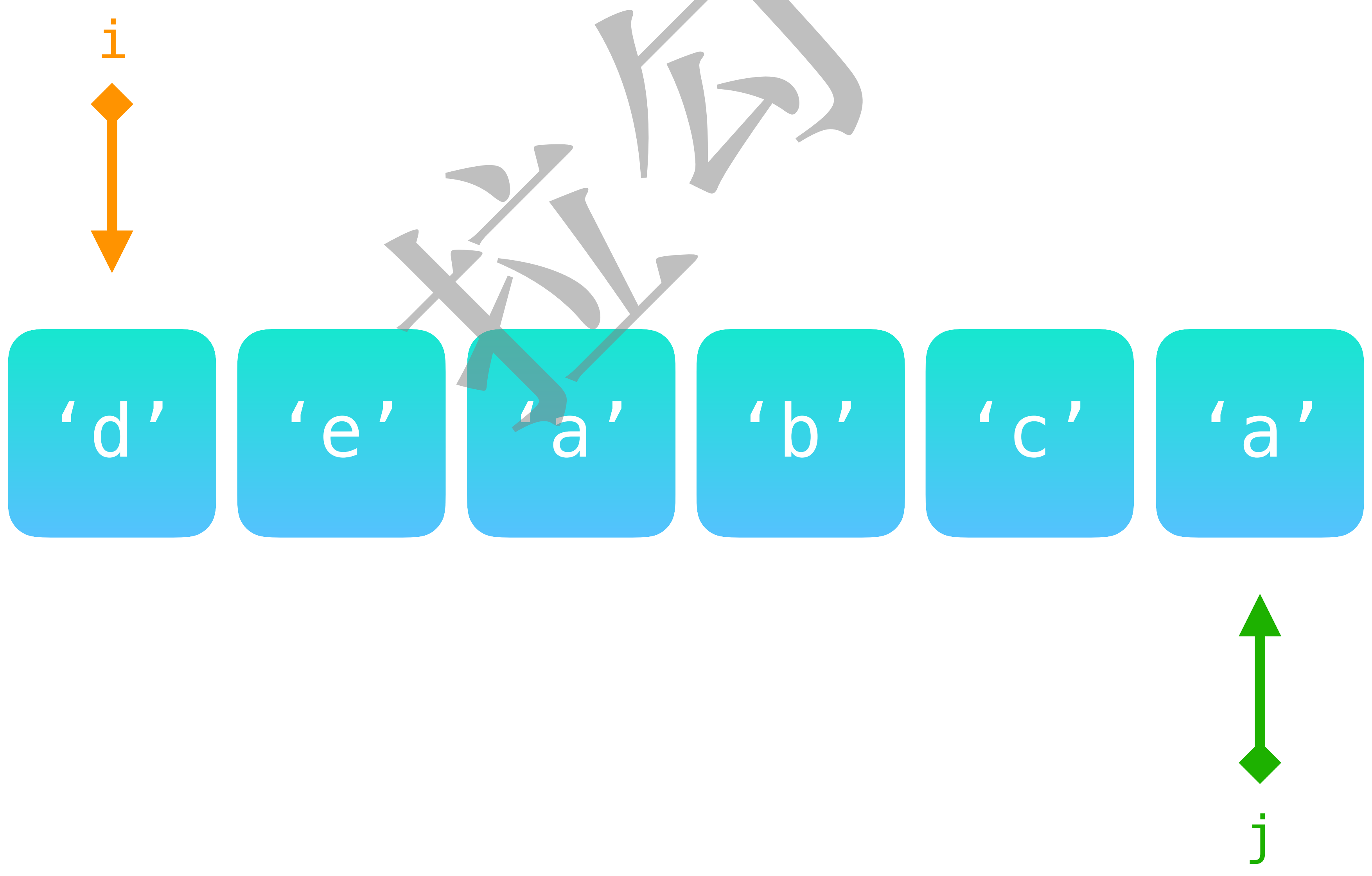
3. 无重复字符的最长子串

- 解法三：优化的线性法



3. 无重复字符的最长子串

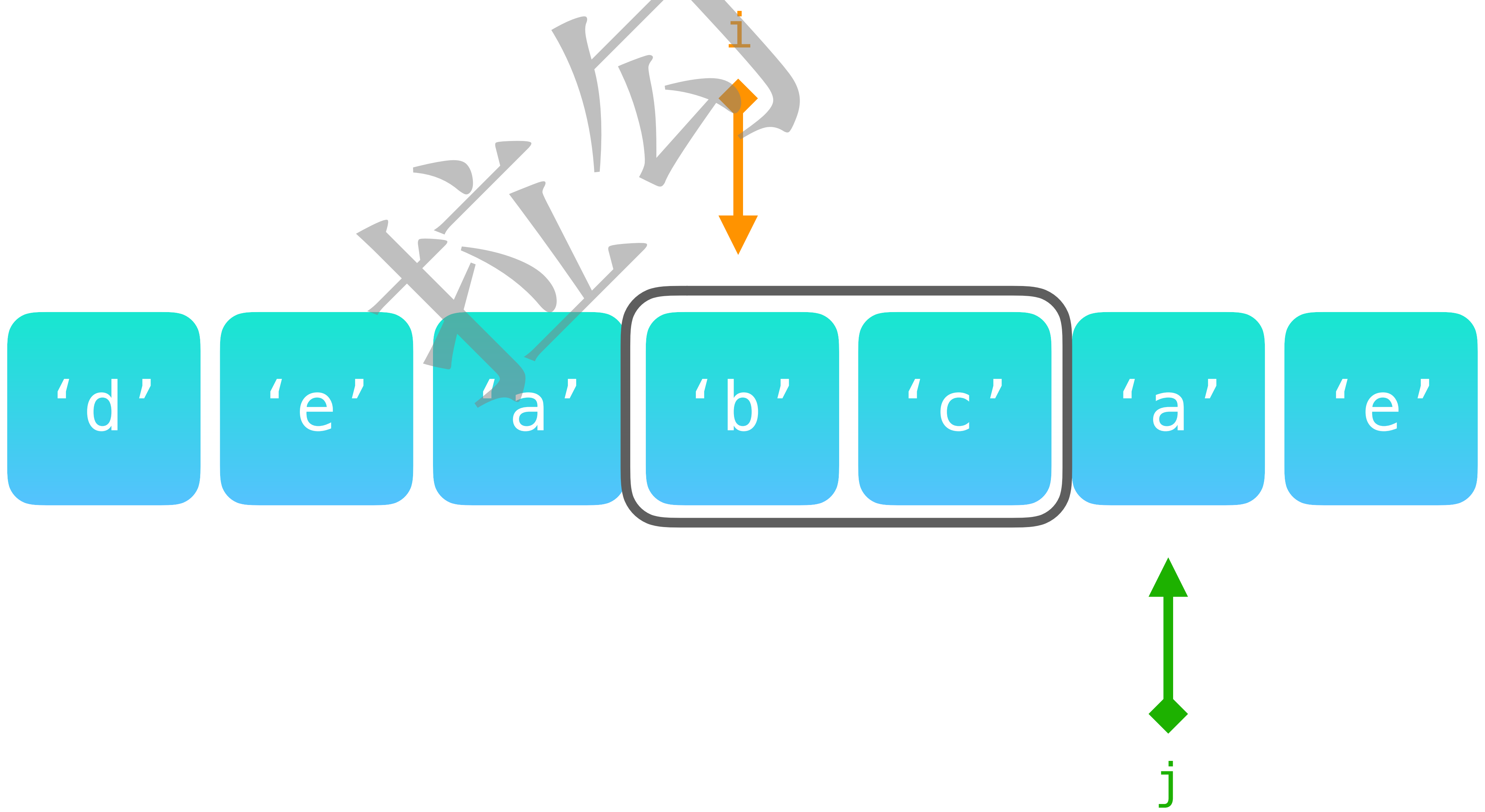
- 解法三：优化的线性法



哈希表记录：{d:0, e:1, a:2, b:3, c:4}

3. 无重复字符的最长子串

- **解法三：优化的线性法**



- ’e' 在哈希表中记录的位置是 1
- i 被移动到的新位置为 max(i, 重复字符出现的位置 + 1)

```java
int lengthOfLongestSubstring(String s) {
  Map<Character, Integer> map = new HashMap<>();
  int max = 0;

  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;
}
```

- 定义一个哈希集合 set 记录上次某字符出现的位置，初始化结果 max 为 0
- 利用快慢指针 i 和 j 扫描一遍字符串
- 如果发现快指针所对应的字符已经出现过，慢指针就进行跳跃
- 把快指针所对应的的字符添加到哈希表中
- 更新结果 max
- 返回结果 max

4. 寻找两个有序数组的中位数

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
```

4. 寻找两个有序数组的中位数

- **解法一：暴力法**

  - 利用归并排序思想将它们合并成一个长度为 $m + n$ 的有序数组

  - 合并的时间复杂度为 $m + n$，从中选取中位数，整体时间复杂度为 $O(m+n) > O(log(m+n))$

4. 寻找两个有序数组的中位数

- **解法二：切分法**

  - 假设 $m + n = L$

    - 如果 L 为奇数，即两个数组元素总个数为奇数，中位数为第 `int(L / 2) + 1` 小的数

      例如数组 [1, 2, 3] 的中位数是 2，2 是第 2 小的数字：`2 = int(3 / 2) + 1`

    - 如果 L 为偶数，中位数为第 `int(L / 2)` 小与 `int(L / 2) + 1` 小的数求和的平均值

      例如数组 [1, 2, 3, 4] 的中位数是 (2 + 3) / 2 = 2.5，其中：`2 = int(3 / 2)`, `3 = int(4 / 2) + 1`

4. 寻找两个有序数组的中位数

解法二：切分法

问题转变为两个有序数组中寻找第 k 小的数 $f(k)$

- 当 L 是奇数时，令 $k = \frac{L}{2}$ ，结果为 $f(k+1)$

- 当 L 是偶数时，结果为 $\frac{f(k)+f(k+1)}{2}$

4. 寻找两个有序数组的中位数

- **解法二：切分法**

nums1[] = a_0 a_1 a_2 a_3 a_4

nums2[] = b_0 b_1 b_2 b_3

假设 $k = 5$，$k_1 = 3$，$k_2 = 2$，我们看看下面几种情况

4. 寻找两个有序数组的中位数

- **解法二：切分法**

1. 当 $a_2 = b_1$ 时，a_2 和 b_1 即为第 5 小的数。

   当我们把 a_0, a_1, a_2 以及 b_0, b_1 按照大小顺序合并在一起时，a_2 和 b_1 一定排在最后，而且不需要考虑a_0, a_1, b_0 的大小关系，例如：

b_0	a_0	a_1	b_1	a_2

4. 寻找两个有序数组的中位数

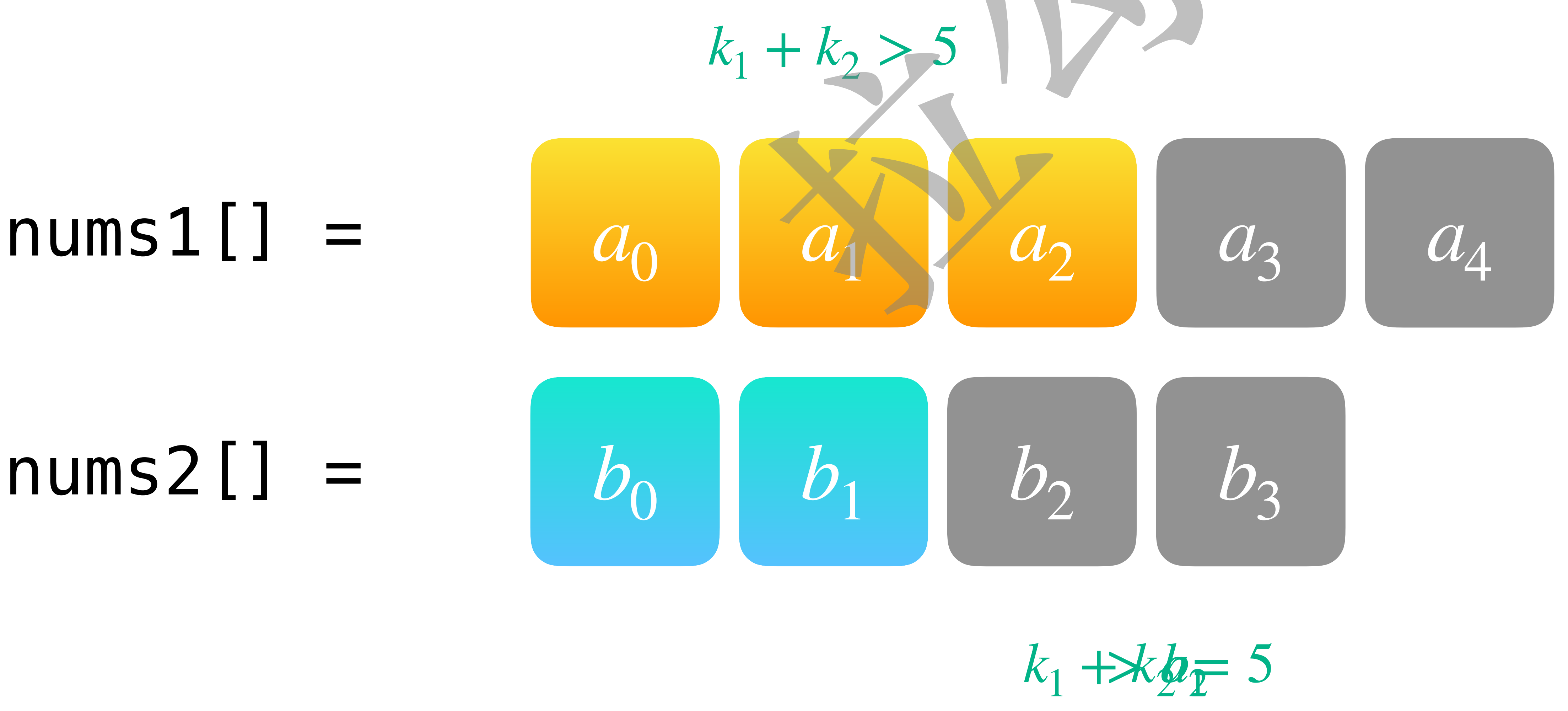
- **解法二：切分法**

2. 当 $a_2 < b_1$ 时，我们无法肯定 a_2 和 b_1 为第 5 小的数。例如：


```
nums1[] =  1  2  3  5  6
nums2[] =  4  7  8  9
```

4. 寻找两个有序数组的中位数

- 解法二：切分法



4. 寻找两个有序数组的中位数

- 解法二：切分法

4. 寻找两个有序数组的中位数

解法二：切分法

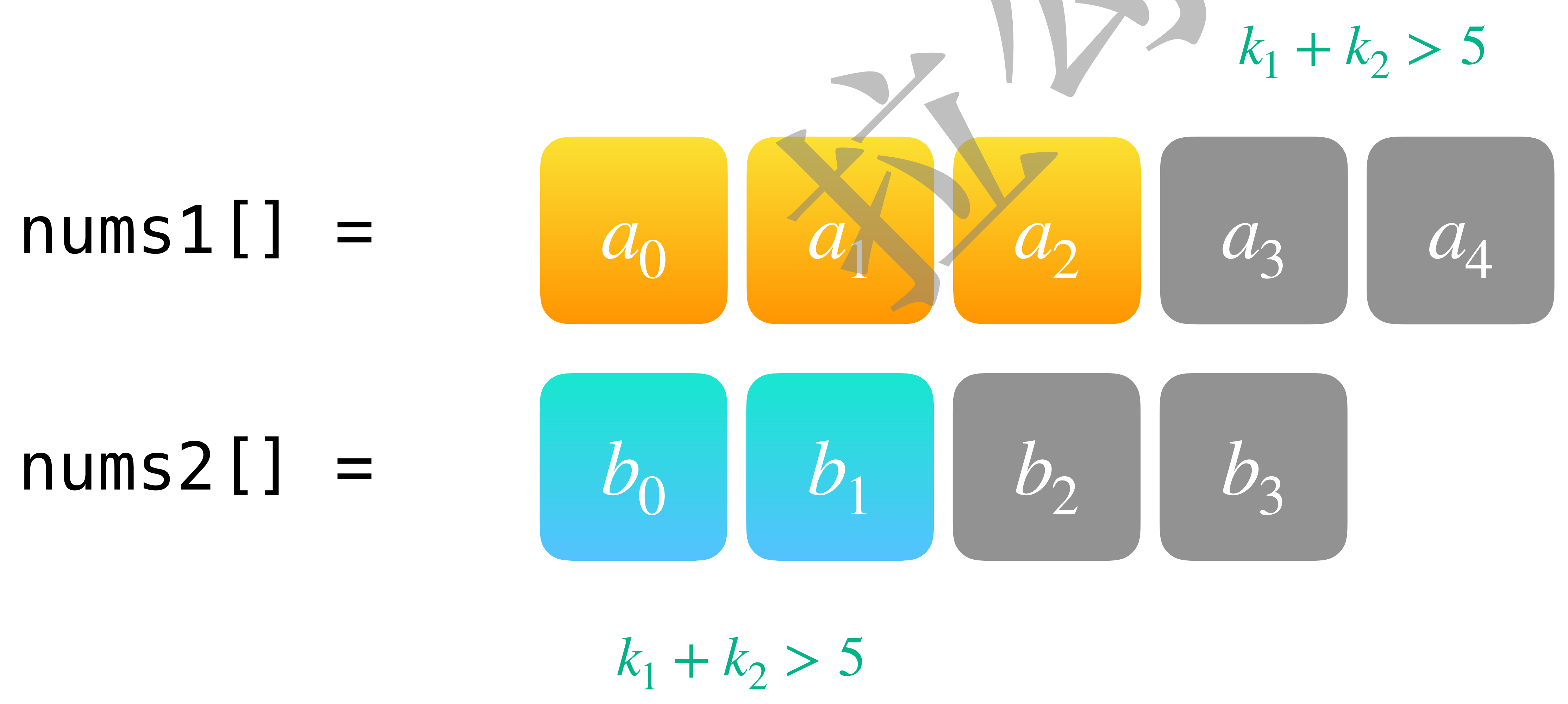
3. 当 $a_2 > b_1$ 时，我们无法肯定 a_2 和 b_1 为第 5 小的数。例如：

nums1[] = 5 6 7 8 9

nums2[] = 1 2 3 4

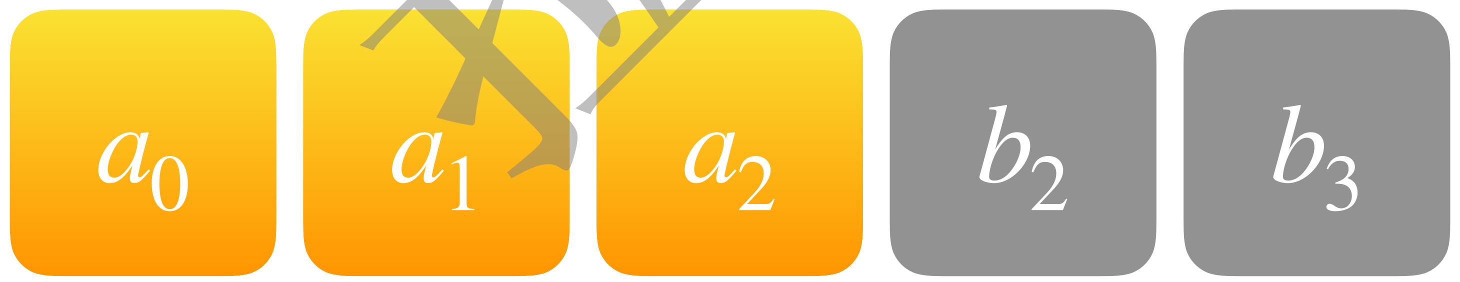
4. 寻找两个有序数组的中位数

- 解法二：切分法



4. 寻找两个有序数组的中位数

- 解法二：切分法



```
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;
  }
}
```

- 主体函数是根据两个字符串长度的总和判断如何调用递归函数以及返回结果。
  - 当总长度为奇数时，返回正中间的数
  - 当总长度为偶数时，返回中间两个数的平均值

```
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];
  }
```

- 进入 findkth，这个函数的目的是寻找第 k 小的元素
- 如果 nums1 数组的长度大于 nums2 数组的长度，将二者互换，加快程序结束
- 如果 nums1 数组的长度为 0 时，直接返回 nums2 数组里第 k 小的数

```
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);
 }
}
```

- 当 k = 1 时，返回两个数组中的最小值
- 分别选两个数组的中间数
- 比较下两者的大小
  - 如果相等，表明中位数已找到，返回该值
  - 如果不等，进行剪枝处理

4. 寻找两个有序数组的中位数

- **解法二：切分法**

  - 时间复杂度分析

求中位数 $k = \frac{(m+n)}{2}$ $k_1 = \frac{k}{2}$ $k_2 = \frac{k}{2}$ 规模减半

算法复杂度类似二分搜索，为 $O(log\frac{(m+n)}{2})$

拓展一

- 如果给定的两个数组都是没有经过排序处理的，应该如何找出中位数呢?

nums1[] = 2 5 3 1 6

nums2[] = 8 9 7 4

拓展一

- 如果给定的两个数组都是没有经过排序处理的，应该如何找出中位数呢?

2 5 3 1 6 8 9 7 4

拓展一

- 如果给定的两个数组都是没有经过排序处理的，应该如何找出中位数呢?

1	2	3	4	5	6	7	8	9

时间复杂度：$O((m+n) \times log(m+n))$

拓展一

「力扣 215」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

快速选择算法

nums1[] =	2	5	3	1	6
nums2[] =	8	9	7	4	

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。

2	5	3	1	6	8	9	7	4

拓展一

2. 将数组排列成两部分：以基准值为分界点，左边的数都小于基准值，右边的数都大于基准值。

2	5	3	1	6	4	7	8	9

拓展一

3. 判断基准值所在位置 p

- 如果 $p = k$，基准值即为所找值，直接返回
- 如果 $k < p$，基准值过大，搜索范围应缩小至基准值左边
- 如果 $k > p$，基准值过小，搜索范围应缩小至基准值右边，此时应寻找第 $k - p$ 小的数，前 p 个数已被淘汰

4. 重复第一步，直到基准值的位置 p 刚好就是要找的 k

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

int quickSelect(int[] nums, 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 (nums[j] <= nums[high]) {
      swap(nums, pivot++, j);
    }
  }
  swap(nums, pivot, high);
```

- 随机取一个基准值，这里取最后一个数作为基准值
- 将比基准值小的数放在左边，比基准值大的数放在右边

8.2 高频题精讲（一） / 寻找两个有序数组的中位数

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

- 判断基准值的位置是不是第 k 大的元素
  - 如果是，返回结果
  - 如果基准值过小，向右搜索
  - 如果基准值过大，向左搜索

- 时间复杂度分析：假设每次都选择了中间的数字作为基准值，设函数的时间执行函数为 $T(n)$

  - 第一次运行，基准值与 n 个元素进行比较，将输入规模减半并递归 $T(n) = T(\frac{n}{2}) + n$
  - 第二次运行，基准值与 $\frac{n}{2}$ 个元素进行比较，将输入规模减半并递归 $T(\frac{n}{2}) = T(\frac{n}{4}) + \frac{n}{2}$
  - $T(\frac{n}{4}) = T(\frac{n}{8}) + \frac{n}{4}$

  $\ldots$

  - $T(2) = T(1) + 2$
  - $T(1) = 1$
  - 逐个带入后得$T(n) = 1 + 2 + \ldots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n = 2 \times n$，所以 $O(T(n)) = O(n)$

- 空间复杂度分析：
  - 如果不考虑递归对栈的开销，则算法没有使用额外的空间，swap 操作都是直接在数组中完成的
  - 因此空间复杂度为 $O(1)$

```
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.length - 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);
```

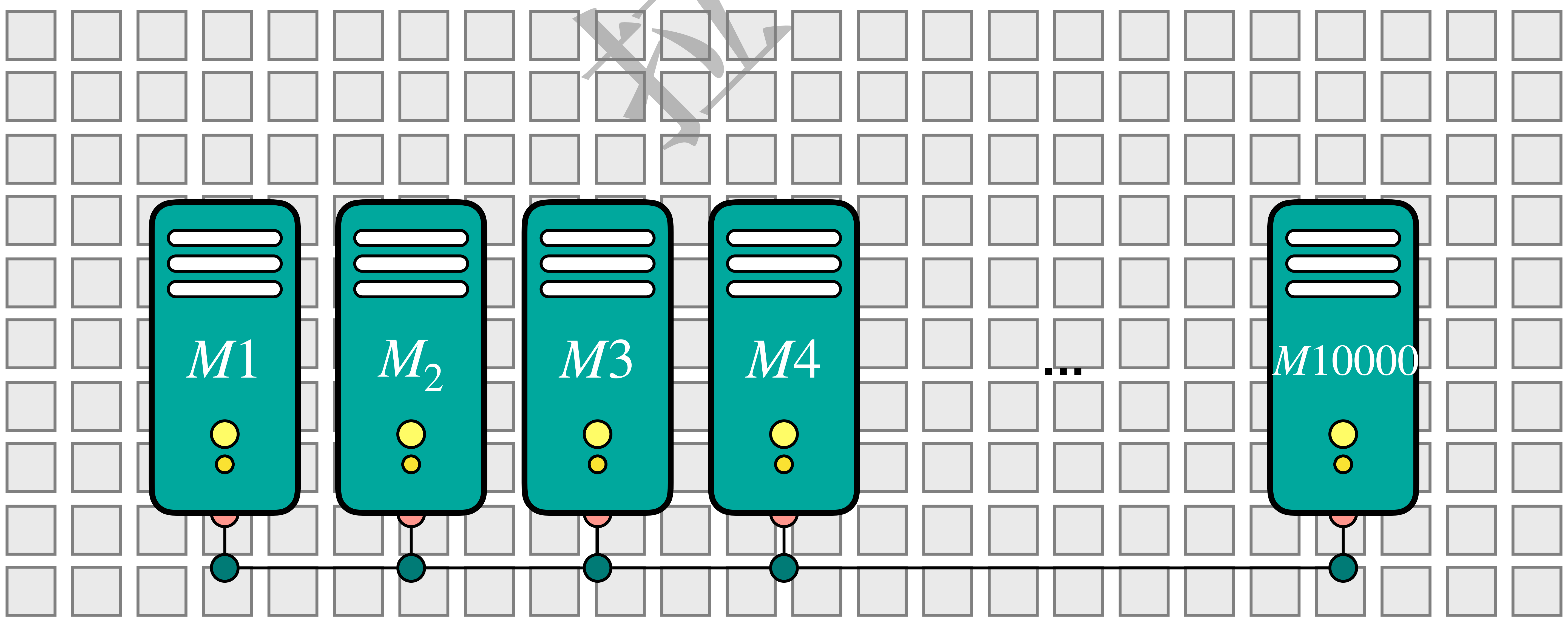
```
  // count the nums that are > pivot from high
  int count = high - pivot + 1;
  // pivot is the one!
  if (count == k) return getNum(nums1, nums2, pivot);
  // pivot is too small, so it must be on the right
  if (count > k) return quickSelect(nums1, nums2, pivot + 1,
high, k);
  // pivot is too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


- 时间复杂度是 $O(m+n)$
- 空间复杂度是 $O(1)$

拓展二

- 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，应该如何找出中位数呢?



拓展二

- 对于分布式的大数据处理，应考虑两方面的限制：
  - 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
    - 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**
    - 而快速排序的空间复杂度为 $log(n)$，即大约 30 次堆栈压入
  - 服务器与服务器之间进行通信时的网络带宽限制

```
class Range {
  public int low;
  public int high;

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

- 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 $30 \times 2 \times 4 = 240$ 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));
    }
  }
}
```

- 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 $30 \times 2 \times 4 = 240$ 字节
- 如果不使用递归写法，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 $30 \times 100 = 3000$ 字节

```
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;
}
```

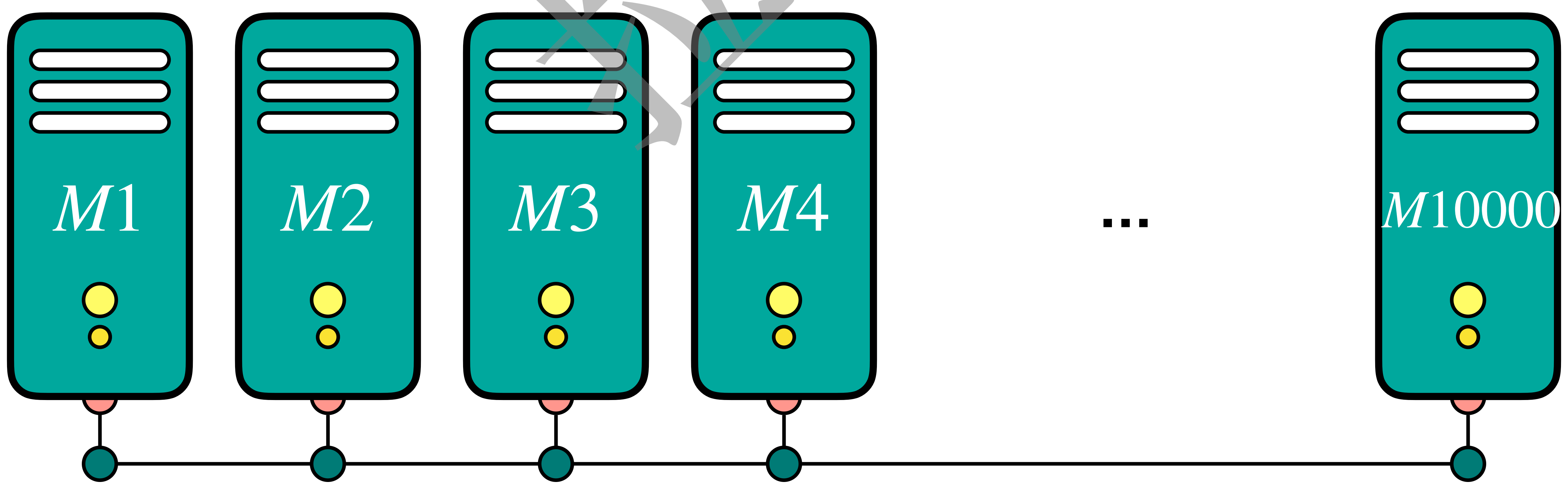

- 每次只需将数组中的某个起始点和终点，即一个范围，压入堆栈中，压入 30 个范围的大小约为 $30 \times 2 \times 4 = 240$ 字节
- 如果不使用递归写法，压入堆栈的还包括程序中的其他变量等，假设需要 100 字节，总共需要 $30 \times 100 = 3000$ 字节
- 快速排序对内存的开销非常小

拓展二

- 对于分布式的大数据处理，应考虑两方面的限制：
  - 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
  - 服务器与服务器之间进行通信时的网络带宽限制
    - 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进行收费的
    - 实际上，它与算法的时间复杂度有很大的关系

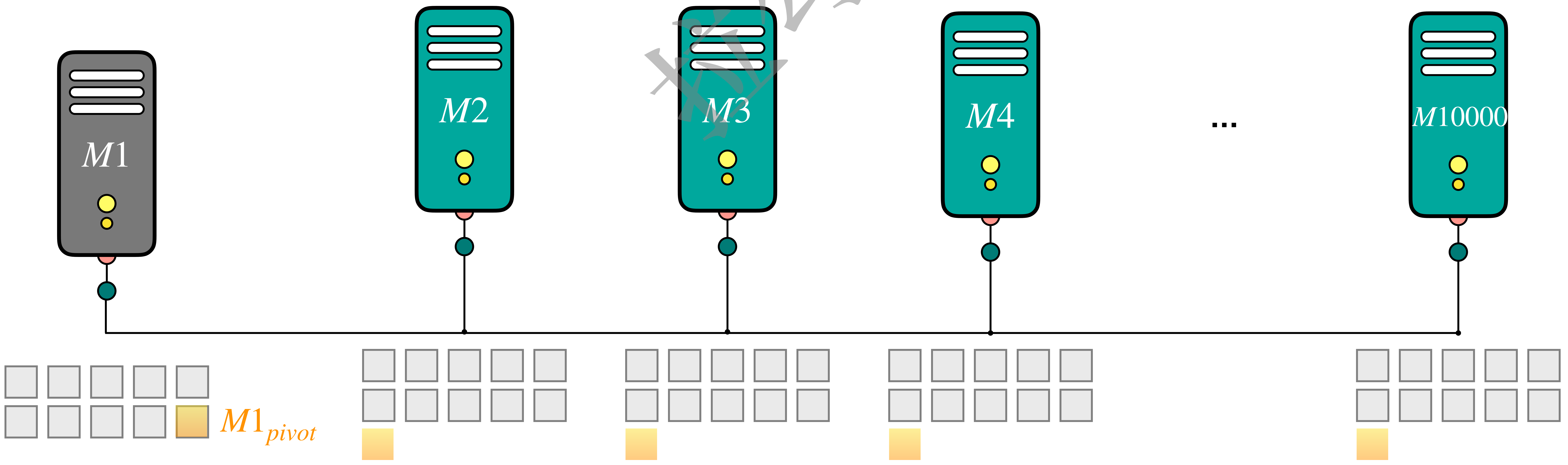
拓展二

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。



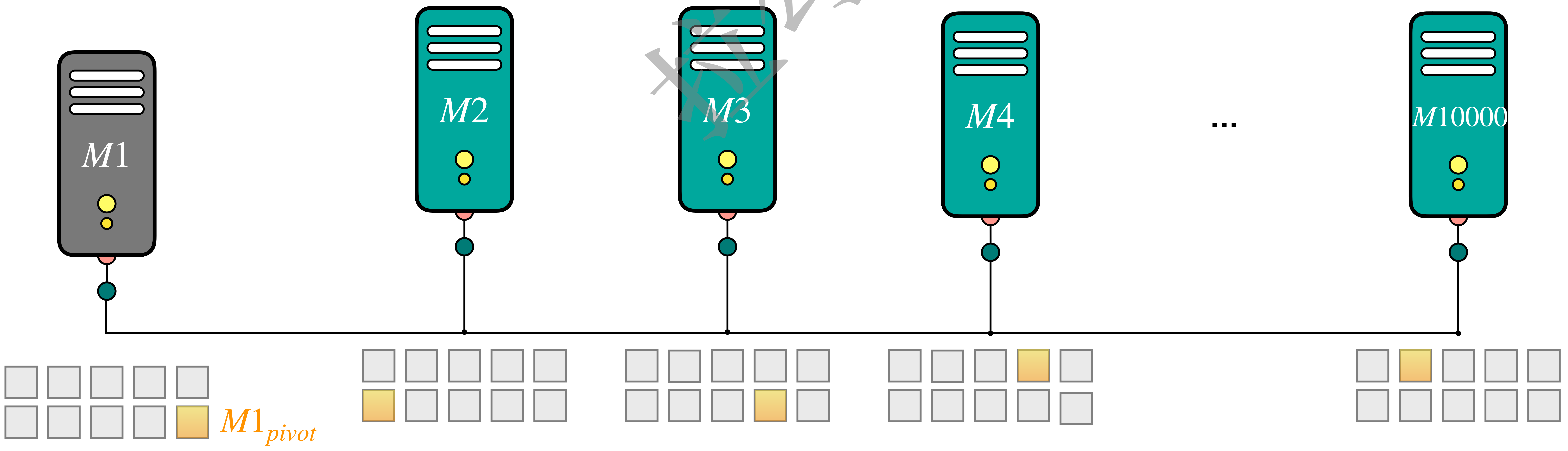
拓展二

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。



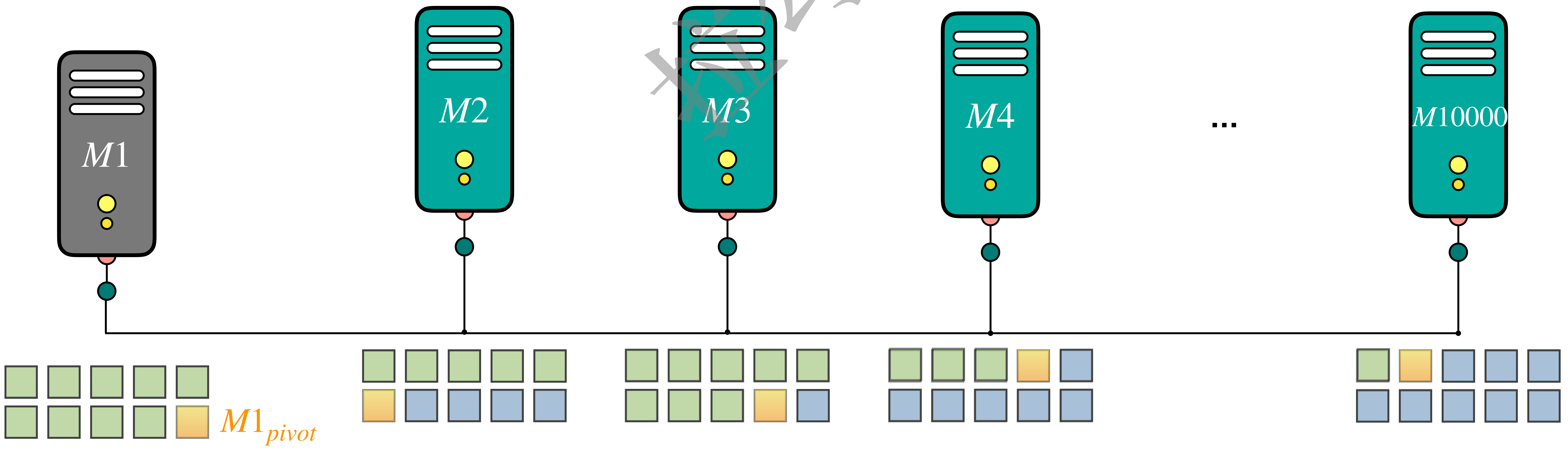
拓展二

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。



拓展二

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

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, greater count，得出各数总和。 接下来进行判断：

- 如果 $total\ less\ count \geq \frac{totalcount}{2}$，表明基准值过大
- 如果 $total\ less\ count + total\ equal\ count \geq \frac{totalcount}{2}$，表明基准值即所寻结果
- 如果 $total\ less\ count + total\ equal\ count < \frac{totalcount}{2}$，表明基准值过小

拓展二

6. 如果为后两种情况，主机会将新基准值广播给各服务器，服务器根据新基准值的大小判断快速选择方向直到最后找到中位数。

- **时间复杂度分析**
  - 整体时间复杂度为 $O(nlogn)$
  - 主机与各服务器之间通信总共需要 $nlogn$ 次，每次通信需要传递一个基准值以及其他三个计数值
  - 如果我们用一些组播网络 (Multicast Network)，可以有效地节省更多带宽

23. 合并 K 个排列链表

合并 k 个排序链表，返回合并后的排序链表。

请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

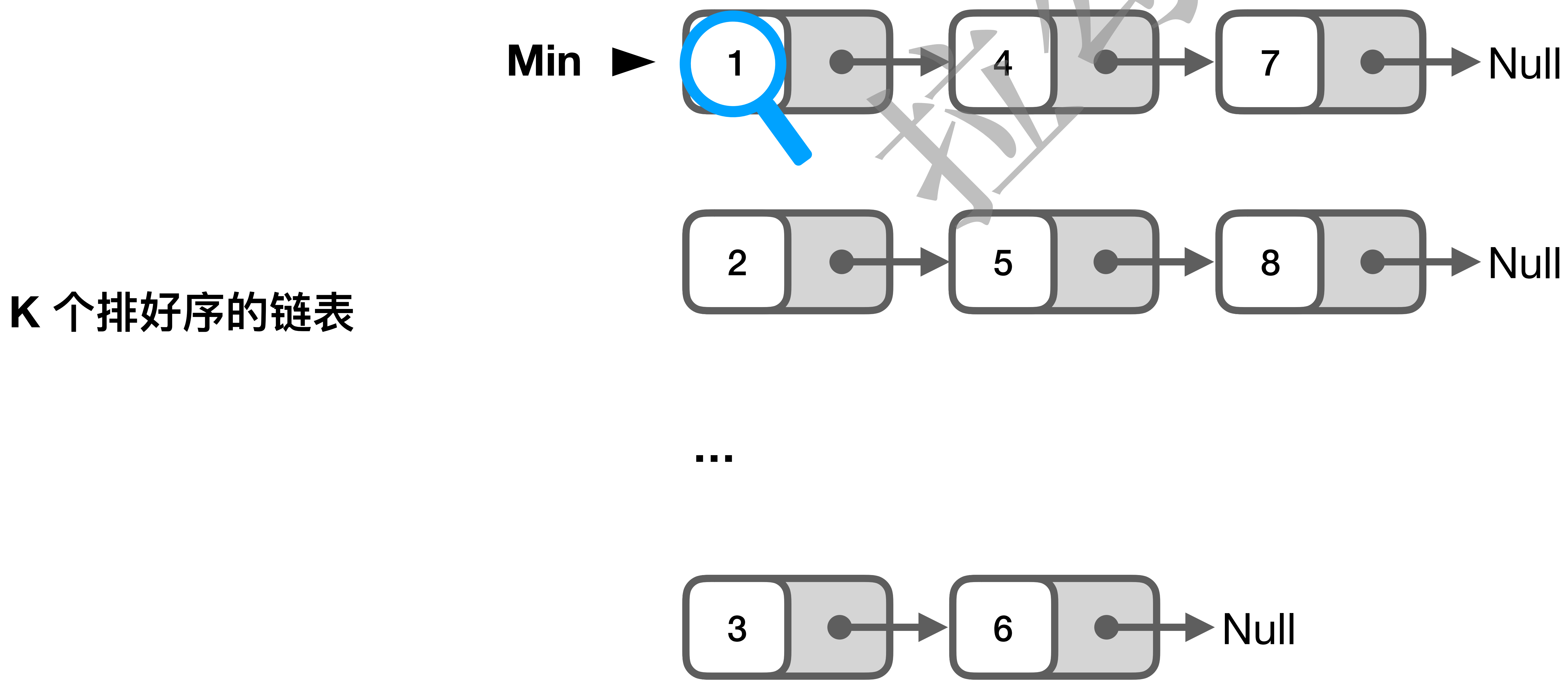
23. 合并 K 个排列链表

- **解法一：暴力法**

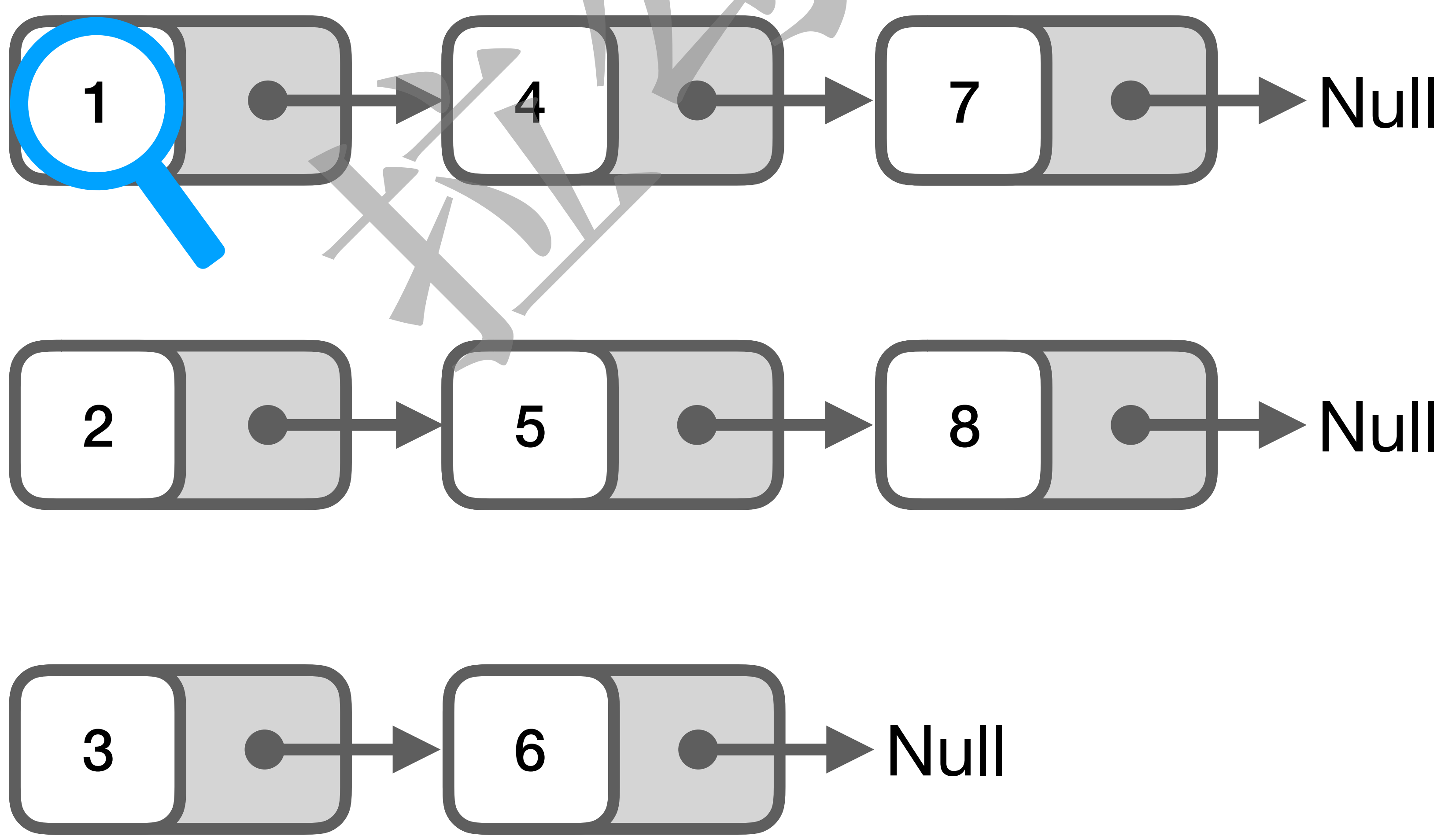
  - 用一个数组保存所有链表中的数后进行排序，再从头到尾将数组遍历，生成一个排好序的链表

  - 假设每个链表平均长度为 n，整体时间复杂度为 $O(nk \times log(nk))$

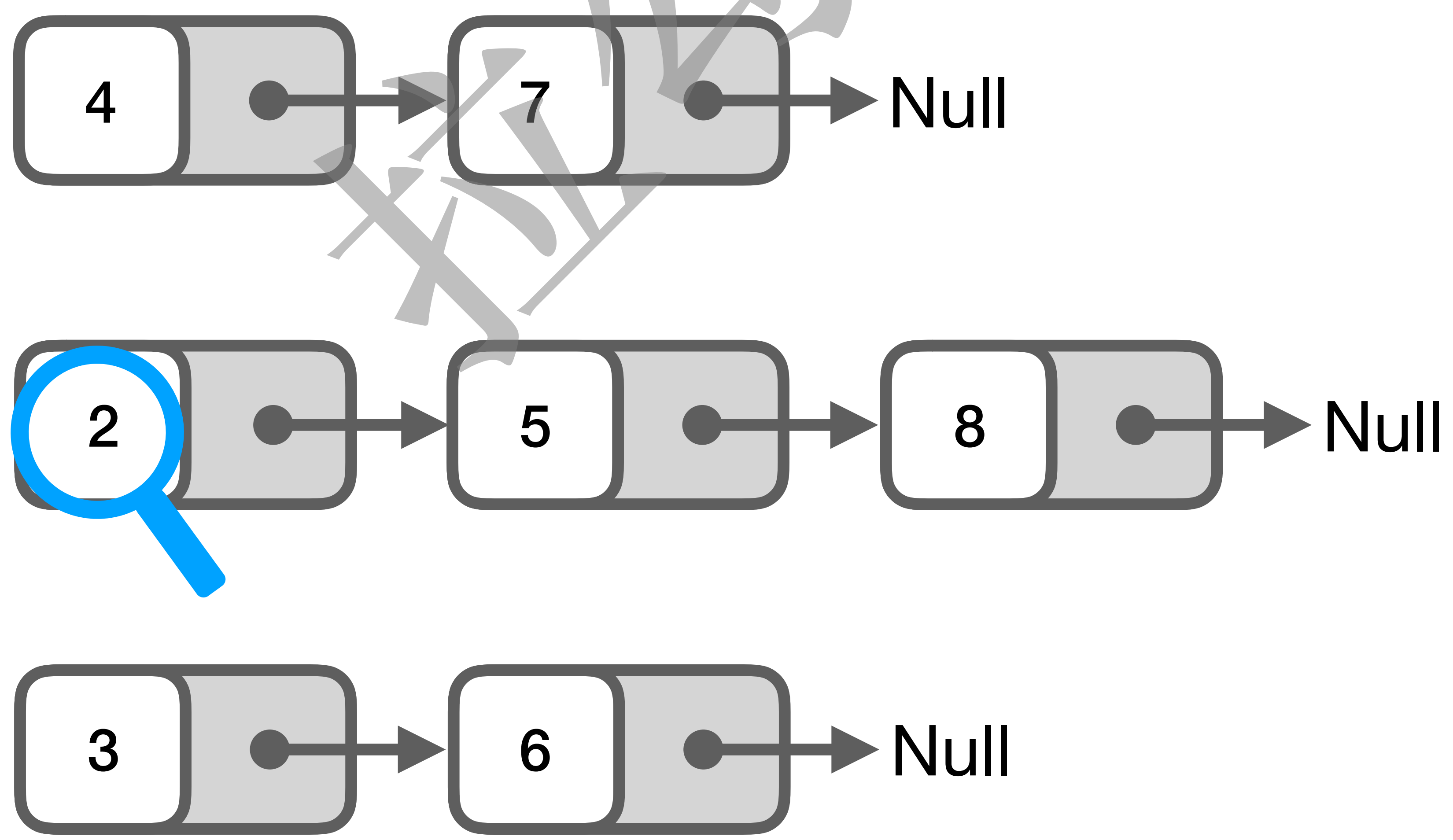
23. 合并 K 个排列链表



23. 合并 K 个排列链表

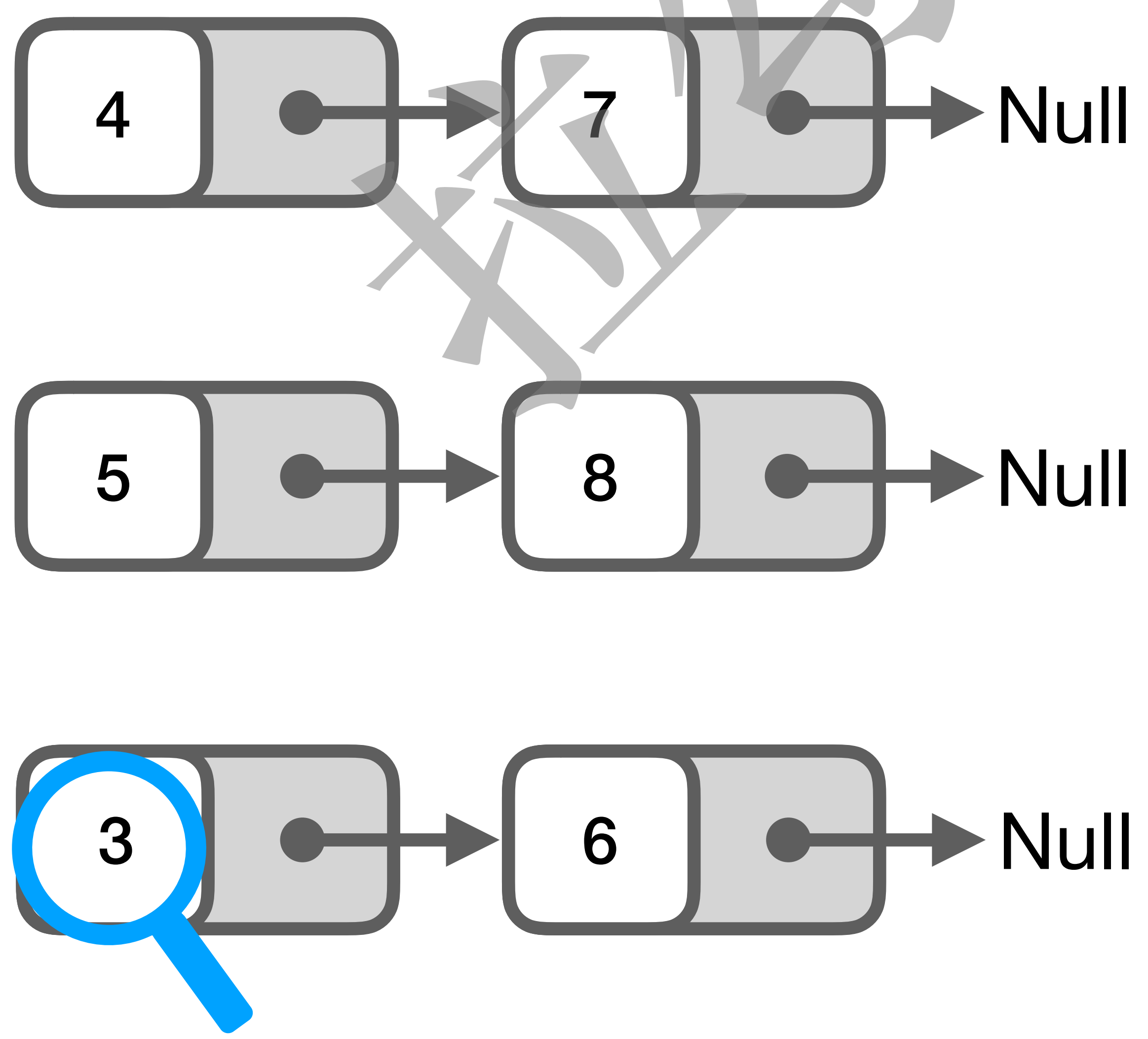


23. 合并 K 个排列链表



输出：1

23. 合并 K 个排列链表



输出：1 → 2

23. 合并 K 个排列链表

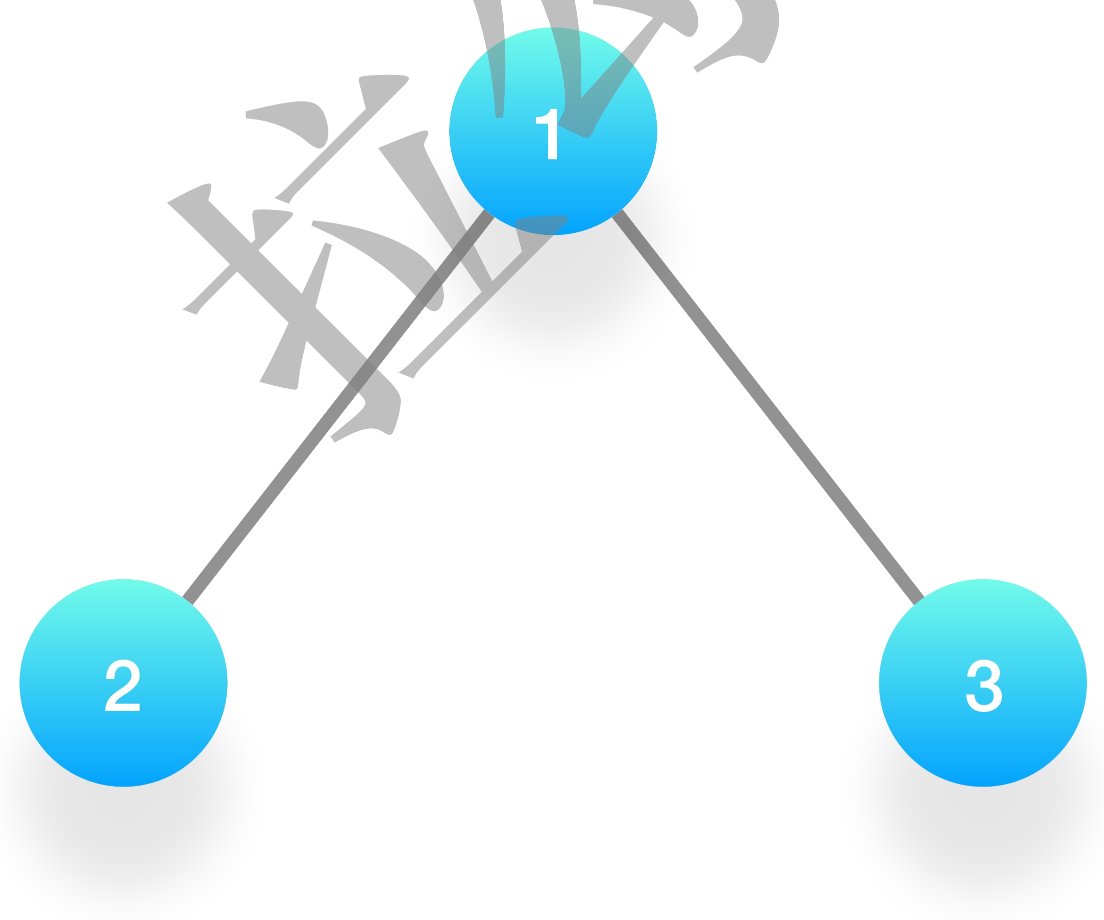
- **解法二：最小堆**

- 每次比较 k 个链表头，时间复杂度为 $O(k)$

- 对 k 个链表头创建一个大小为 k 的最小堆
  - 创建一个大小为 k 的最小堆所需时间为 $O(k)$
  - 从堆中取最小的数，所需时间都是 $O(lg(k))$
  - 如果每个链表平均长度为 n，则共有 nk 个元素，即用大小为 k 的最小堆过滤 nk 个元素
  - 整体时间复杂度为 $O(nk \times log(k))$

23. 合并 K 个排列链表

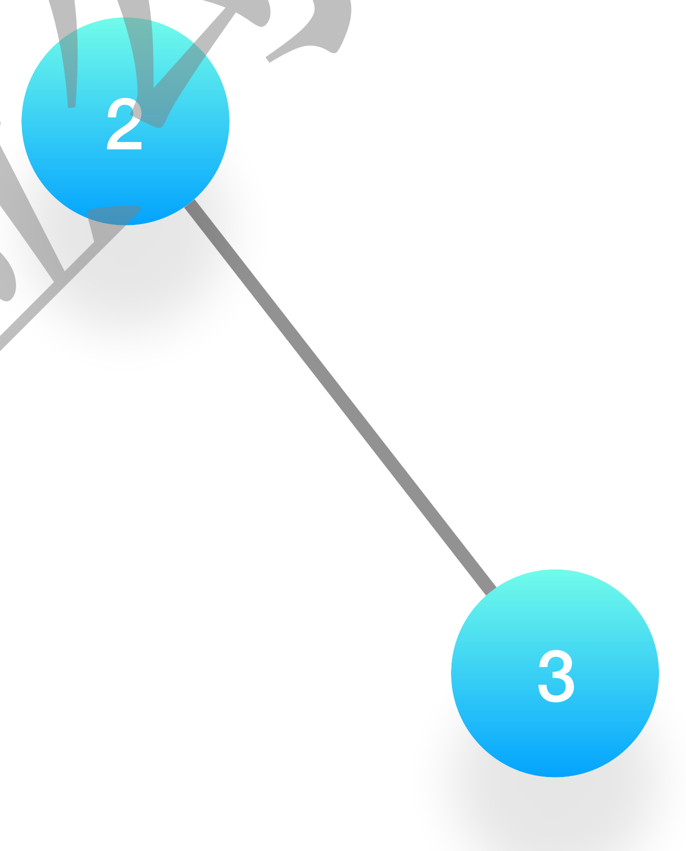
- 解法二：最小堆



LeetCode 力扣 × 拉勾 LAGOU.COM

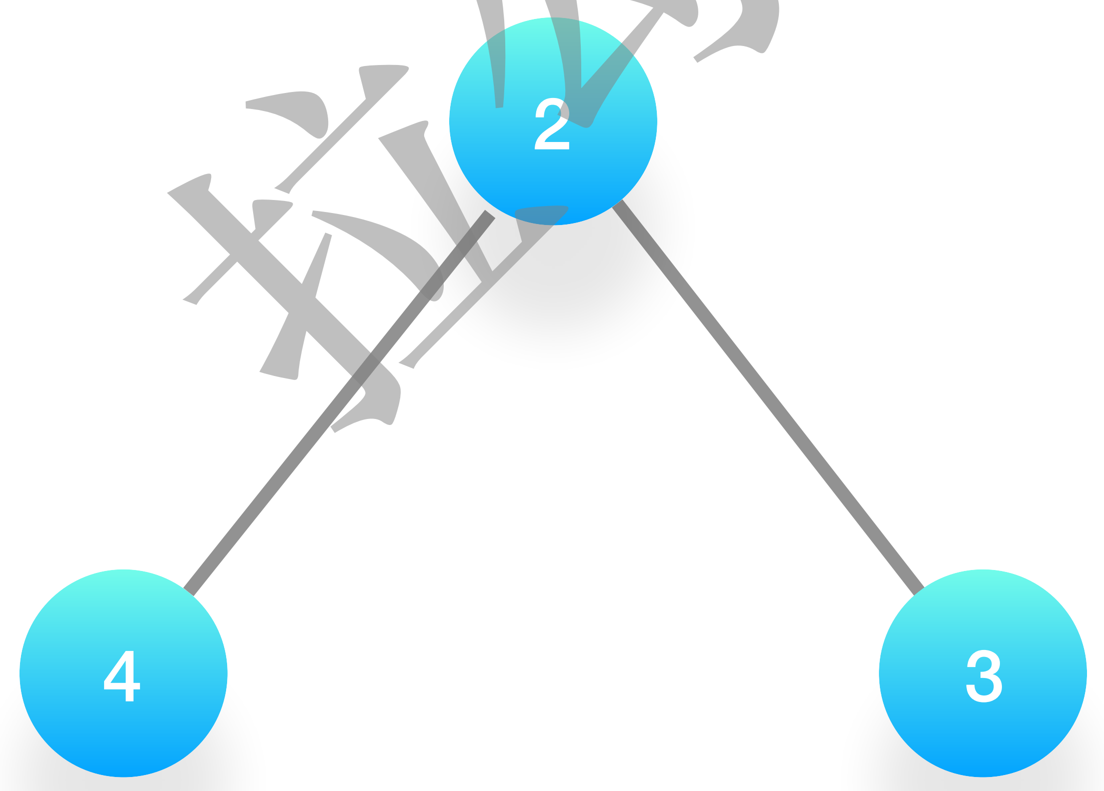
23. 合并 K 个排列链表

- **解法二：最小堆**



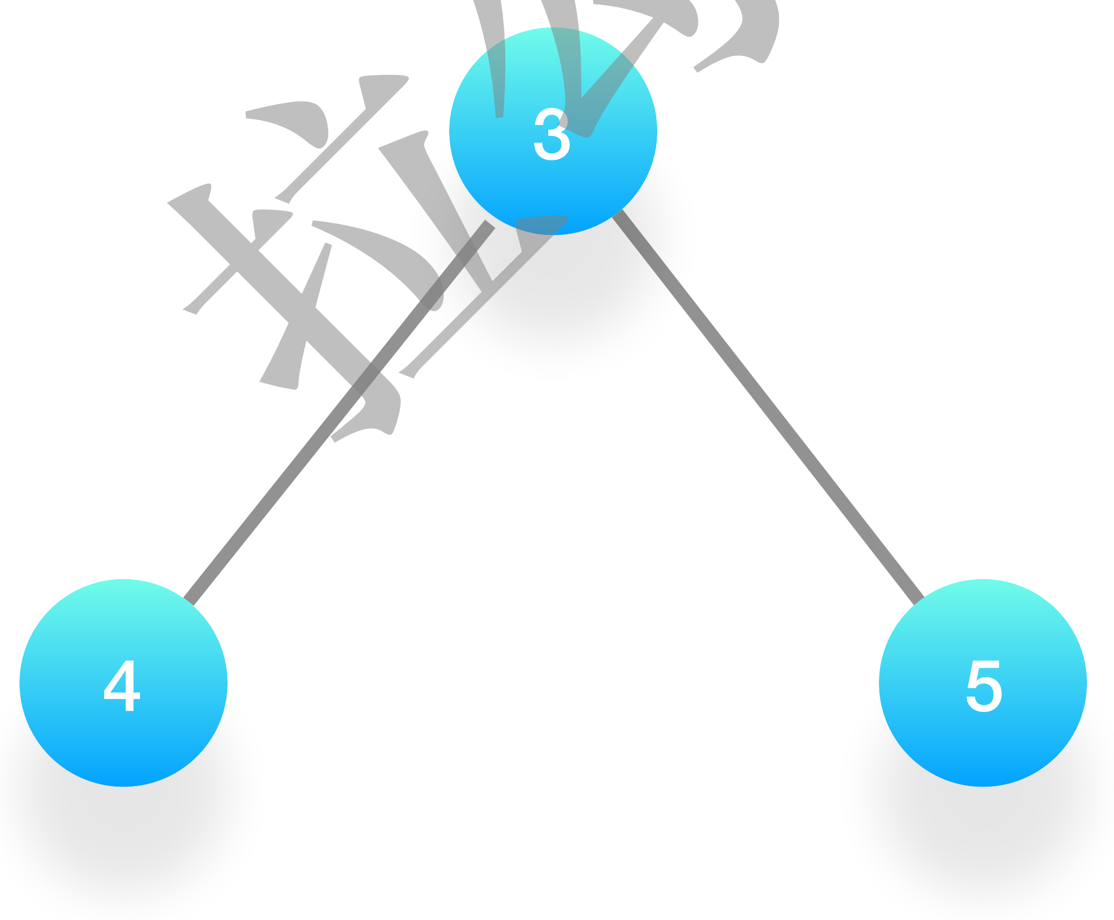
23. 合并 K 个排列链表

- 解法二：最小堆



23. 合并 K 个排列链表

- 解法二：最小堆



```java
public ListNode mergeKLists(ListNode[] lists) {
  ListNode fakeHead = new ListNode(0), p = fakeHead;

  int k = lists.length;

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

  for (int i = 0; i < k; i++) {
   if (lists[i] != null) {
     heap.offer(lists[i]);
   }
  }
```

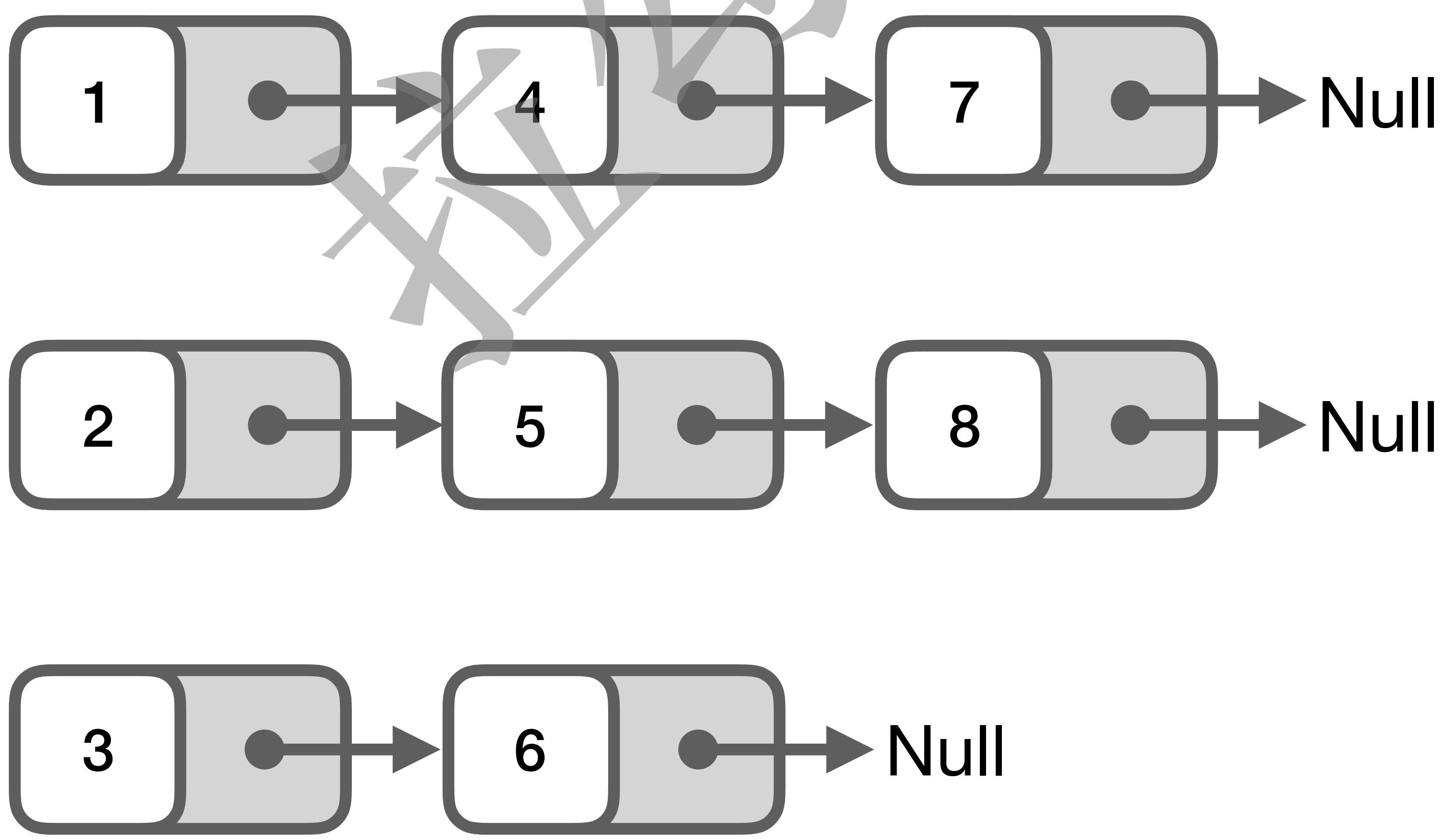
- 利用一个空的链表头方便我们插入节点
- 定义一个最小堆来保存 k 个链表节点。
- 将 k 个链表的头放到最小堆中
- 从最小堆中将当前最小节点取出，插入到结果链表中

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

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

  if (node.next != null) {
    heap.offer(node.next);
  }
}
return fakeHead.next;
}
```

- 如果发现该节点后还有后续节点，将其点加入最小堆中
- 最后返回结果链表

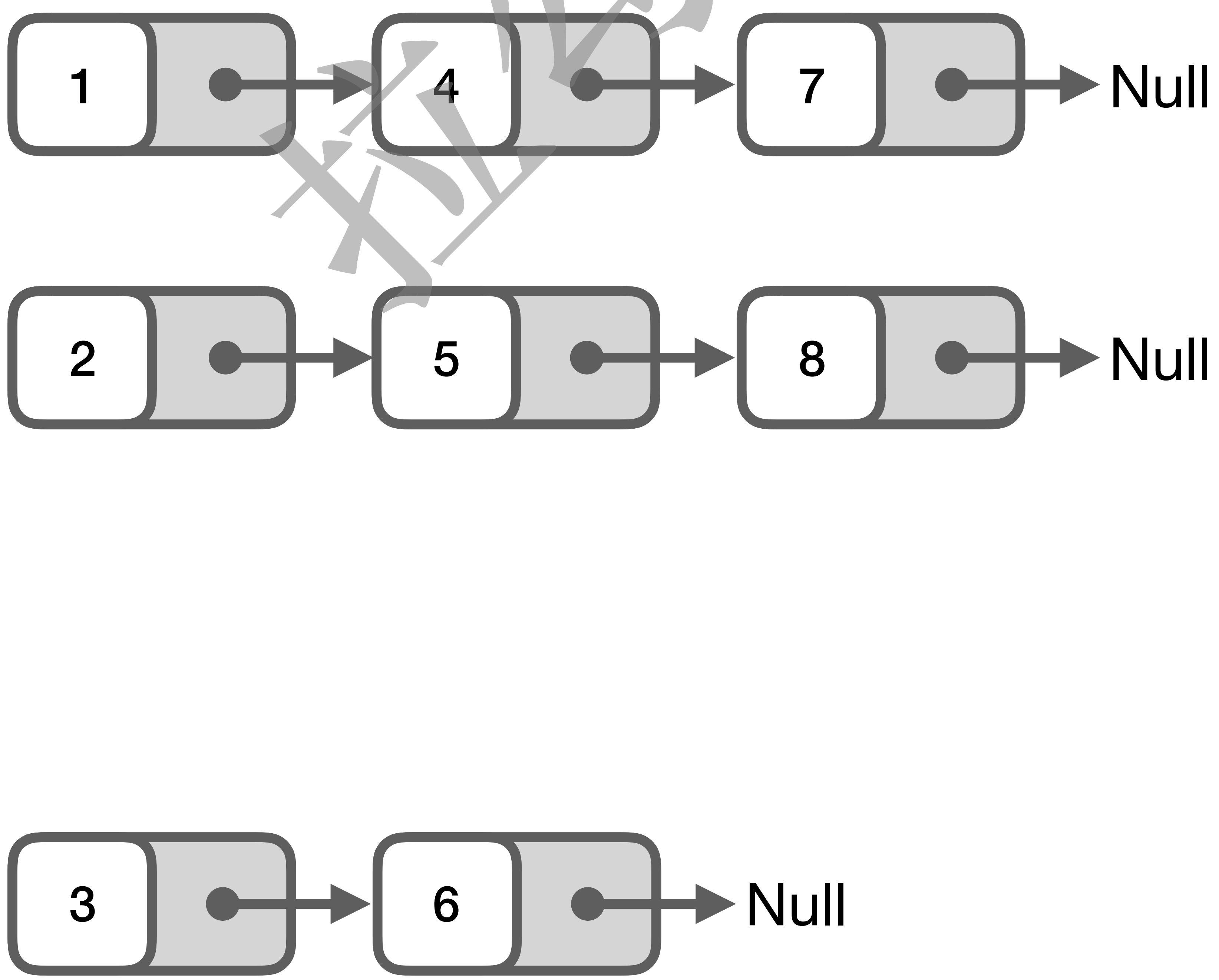
23. 合并 K 个排列链表

- 解法三：分治法



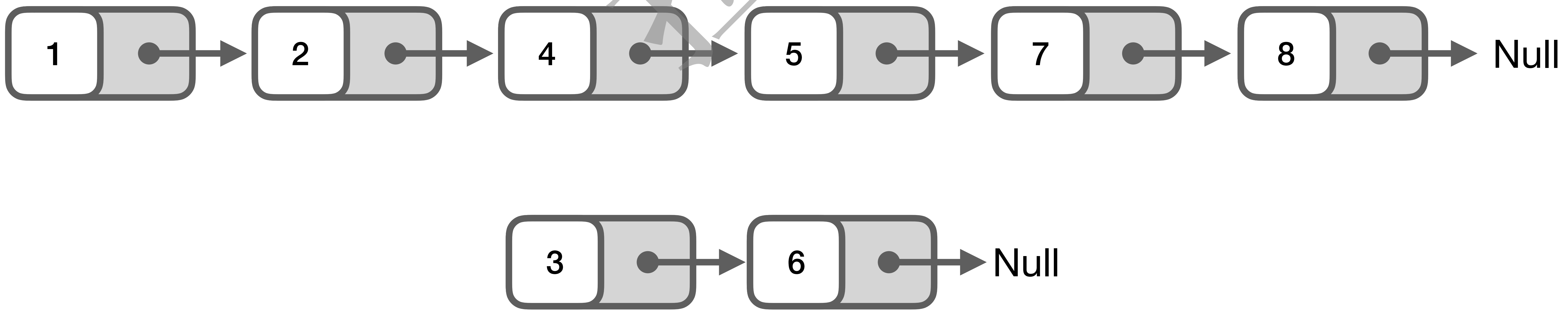
23. 合并 K 个排列链表

- 解法三：分治法



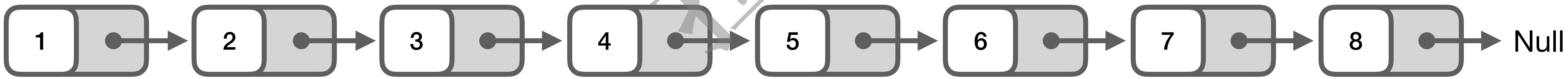
23. 合并 K 个排列链表

- 解法三：分治法



23. 合并 K 个排列链表

- 解法三：分治法



这个做法运用的是典型的分治思想，非常类似归并排序操作

8.3 高频题精讲（一） / 合并 K 个排列链表

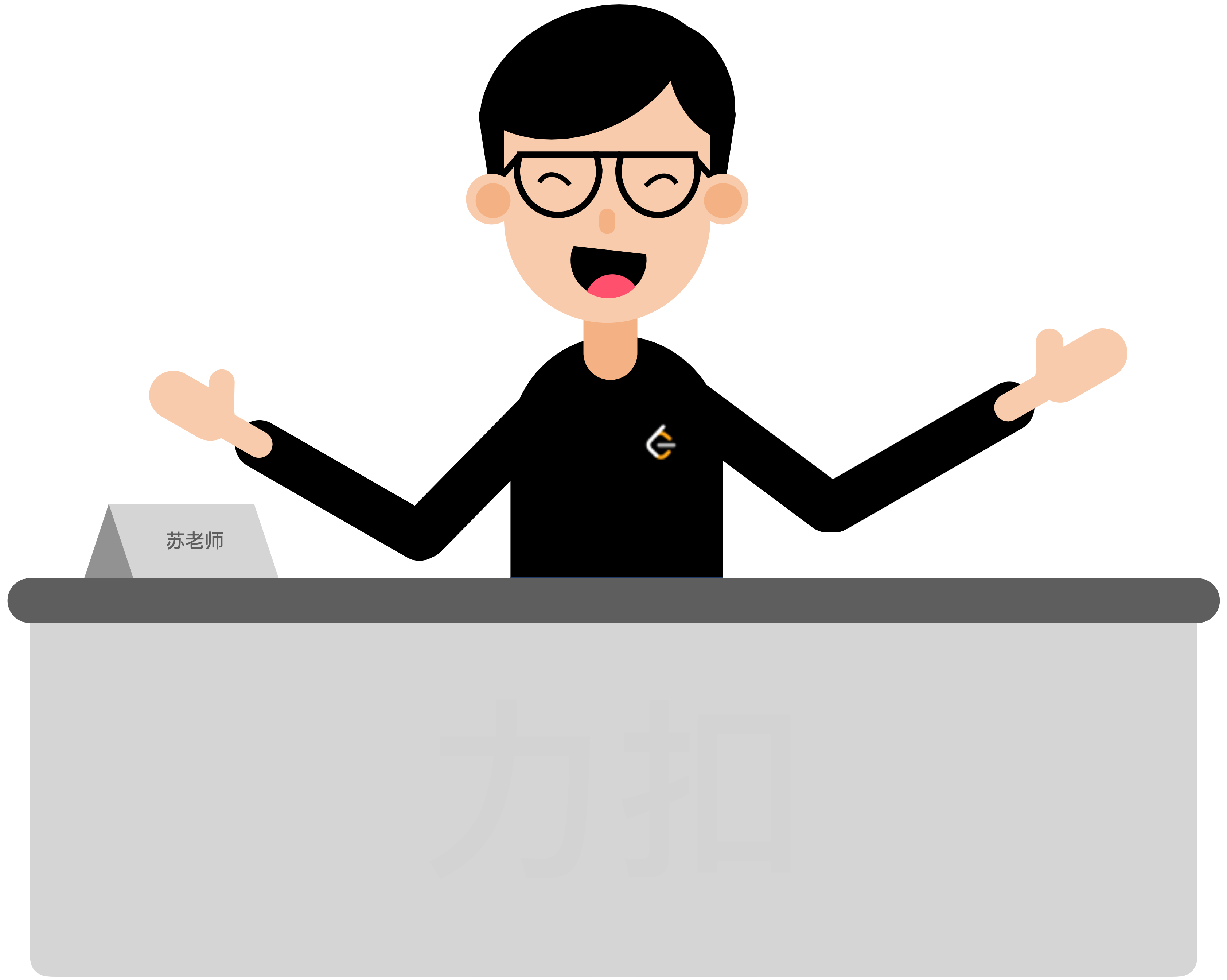
```
public ListNode mergeKLists(ListNode[] lists, int low, int high) {
  if (low == high) return lists[low];

  int middle = low + (high - low) / 2;

  return mergeTwoLists(
    mergeKLists(lists, low, middle),
    mergeKLists(lists, middle + 1, high)
  );
}

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;
}
```

- 主函数时分类似之前介绍过的归并排序
- 从中间切一刀
- 然后递归地处理坐标和右边的列表，最后合并起来

- 时间复杂度是 $O(nk \times log(k))$
- 不像最小堆解法一样需要维护一个额外的数据结构
- 空间复杂度是 $O(1)$

大厂面试高频题精讲

- 力扣 3
- 力扣 4
- 力扣 23



Next：课时 9《大厂算法面试真题 - 高频题精讲（二）》

多加练习，才能更好地巩固知识点。

关注 “拉勾教育”
学习技术干货

关注 “LeetCode力扣”
获得算法技术干货