

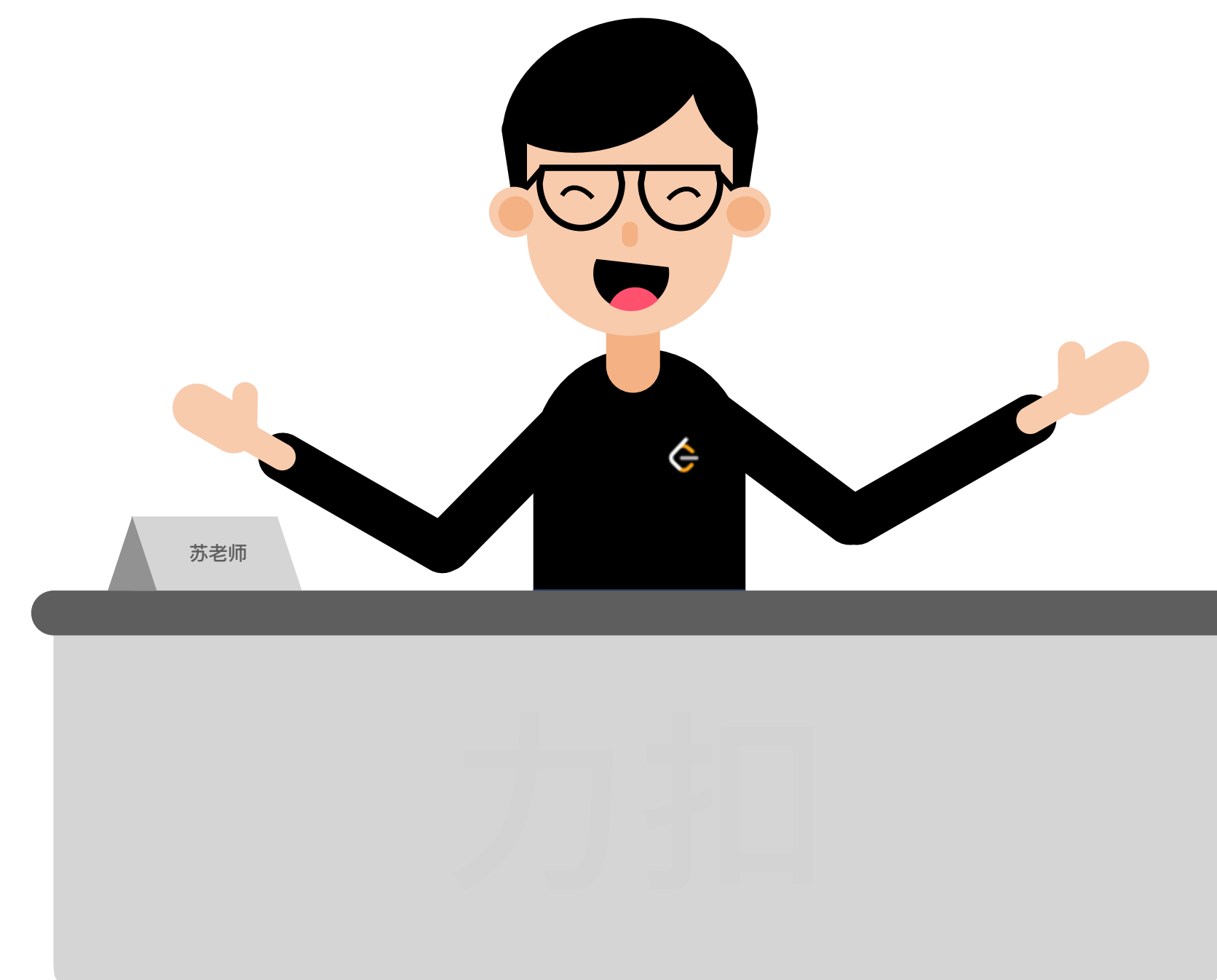
第十课

剖析大厂算法面试真题 - 难题精讲（一）

难题精讲（一）

- 正则表达式匹配
- 柱状图中的最大矩形
- 实现 `strStr()`

拉勾



10. 正则表达式匹配

给你一个字符串 s 和一个字符规律 p ，请你来实现一个支持 ‘.’ 和 ‘*’ 的正则表达式匹配。

‘.’ 匹配任意单个字符

‘*’ 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖整个字符串 s 的，而不是部分字符串。

说明：

- s 可能为空，且只包含从 $a-z$ 的小写字母。
- p 可能为空，且只包含从 $a-z$ 的小写字母，以及字符 ‘.’ 和 ‘*’。

示例 1：

输入：

$s = "aa"$

$p = "a"$

输出：false

解释：“a” 无法匹配 “aa” 整个字符串。

示例 2：

输入：

$s = "aa"$

$p = "a*"$

输出：true

解释：因为 ‘*’ 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 ‘a’。因此，字符串 “aa” 可被视为 ‘a’ 重复了一次。

10. 正则表达式匹配

给你一个字符串 s 和一个字符规律 p ，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

'.' 匹配任意单个字符

'*' 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖整个字符串 s 的，而不是部分字符串。

说明：

- s 可能为空，且只包含从 $a-z$ 的小写字母。
- p 可能为空，且只包含从 $a-z$ 的小写字母，以及字符 '.' 和 '*'。

示例 3：

输入：

$s = "ab"$

$p = ".*"$

输出：true

解释："." 表示可匹配零个或多个（'*'）任意字符（'.'）。

示例 4：

输入：

$s = "aab"$

$p = "c*a*b"$

输出：true

解释：因为 '*' 表示零个或多个，这里 'c' 为 0 个，'a' 被重复一次。因此可以匹配字符串 "aab"。

10. 正则表达式匹配

给你一个字符串 s 和一个字符规律 p ，请你来实现一个支持 ‘.’ 和 ‘*’ 的正则表达式匹配。

‘.’ 匹配任意单个字符

‘*’ 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖整个字符串 s 的，而不是部分字符串。

说明：

- s 可能为空，且只包含从 $a-z$ 的小写字母。
- p 可能为空，且只包含从 $a-z$ 的小写字母，以及字符 ‘.’ 和 ‘*’。

示例 5：

输入：

$s = \text{"mississippi"}$

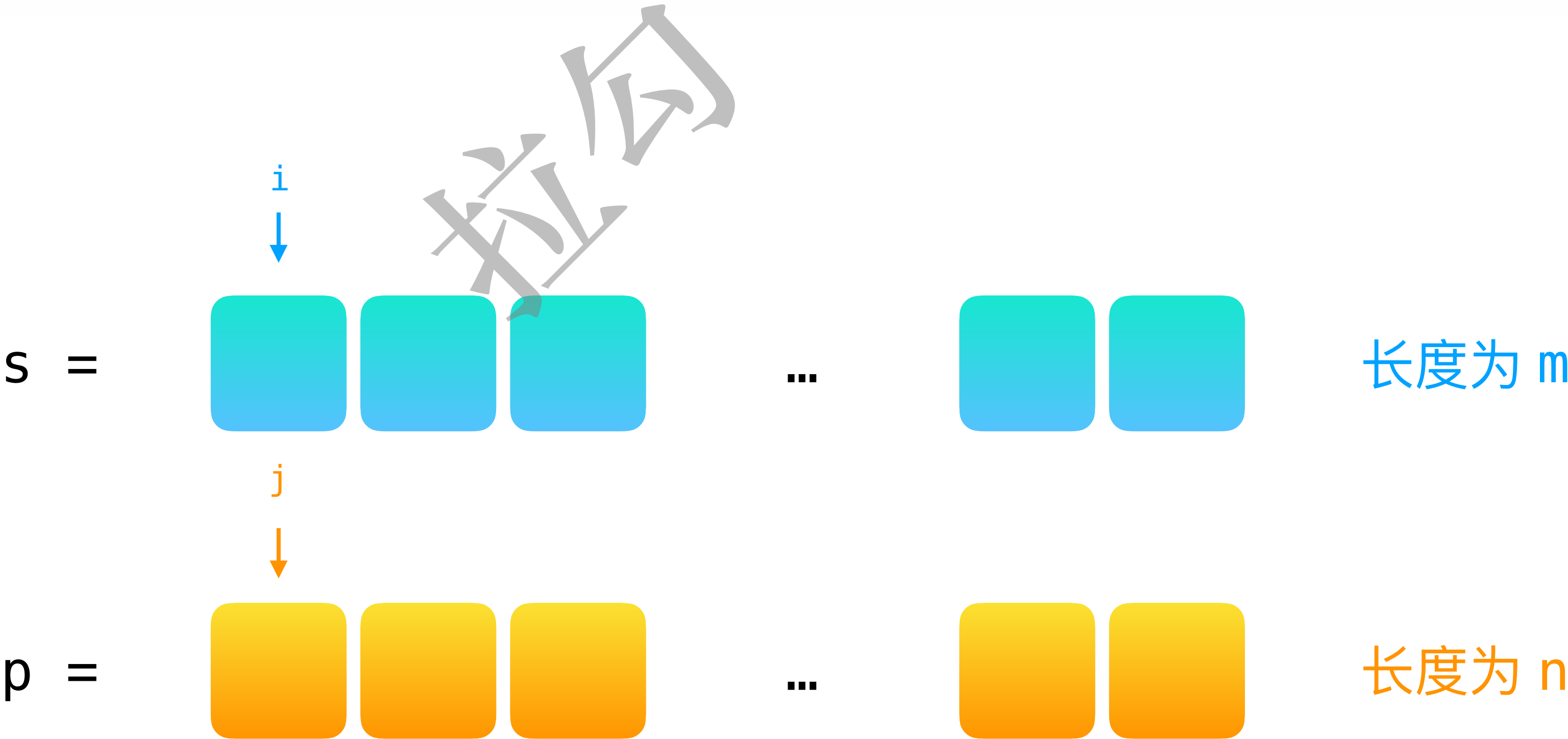
$p = \text{"mis*is*p*."}$

输出：false

10. 正则表达式匹配

解题思路

▸ 判断 s 与 p 是否匹配



10. 正则表达式匹配

解题思路

▸ 判断 s 与 p 是否匹配

i == m

&&

j == n

————→ s 与 p 匹配

10. 正则表达式匹配

解题思路

- ▶ p 字符串中的出现的点匹配符 ‘.’

输入：

```
s = "leetcode"
```

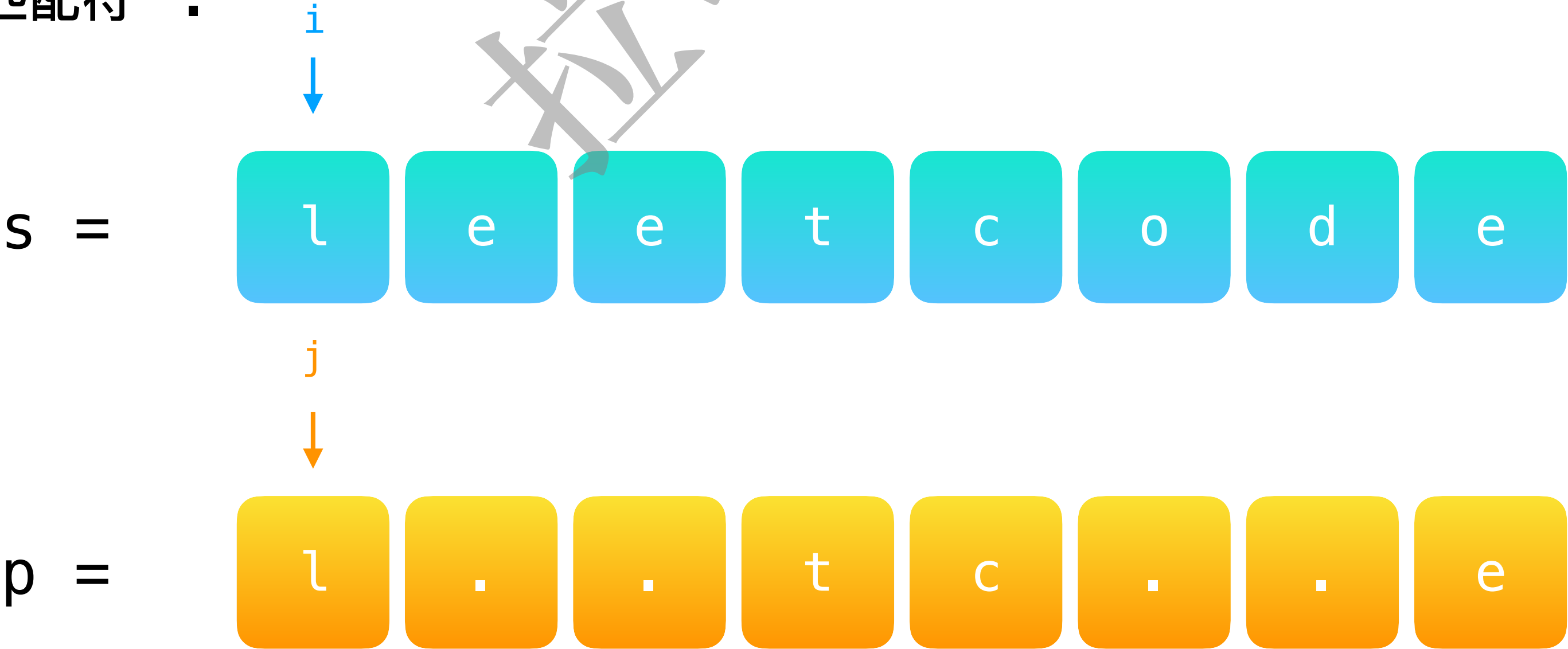
```
p = "l..tc..e"
```

输出： true

10. 正则表达式匹配

解题思路

▸ p 字符串中的出现的点匹配符 ‘.’



10. 正则表达式匹配

解题思路

- ▶ p 字符串中的出现的星匹配符 ‘*’ -> ‘*’ 匹配零个或多个前面的哪一个元素
 - 它匹配的是 p 字符串中，该星号前面的那个字符
 - 它可以匹配零个或多个
 - 星号匹配符前面必须有一个非星的字符

a*

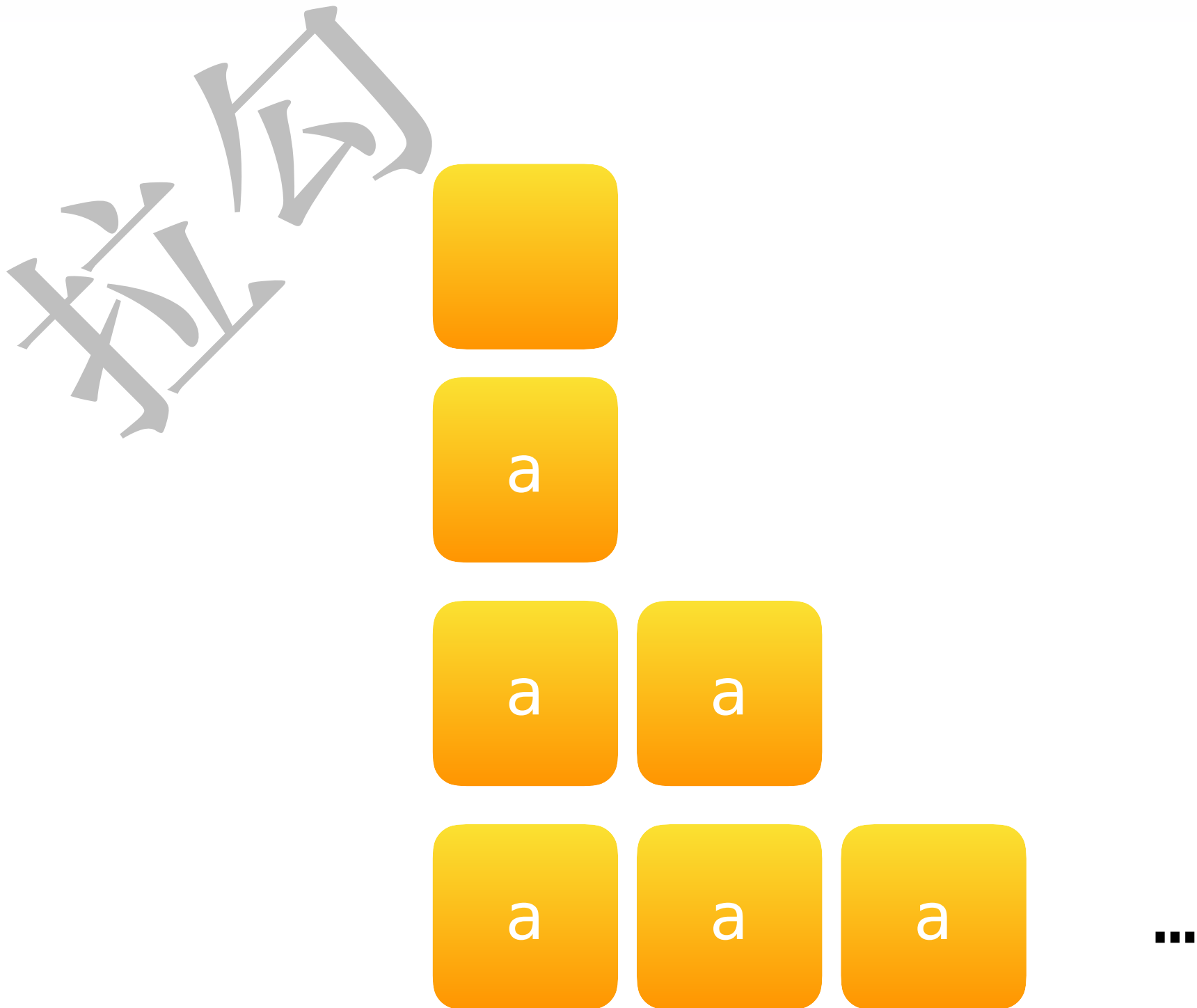
*

10. 正则表达式匹配

解题思路

- p 字符串中的出现的星匹配符 ‘*’

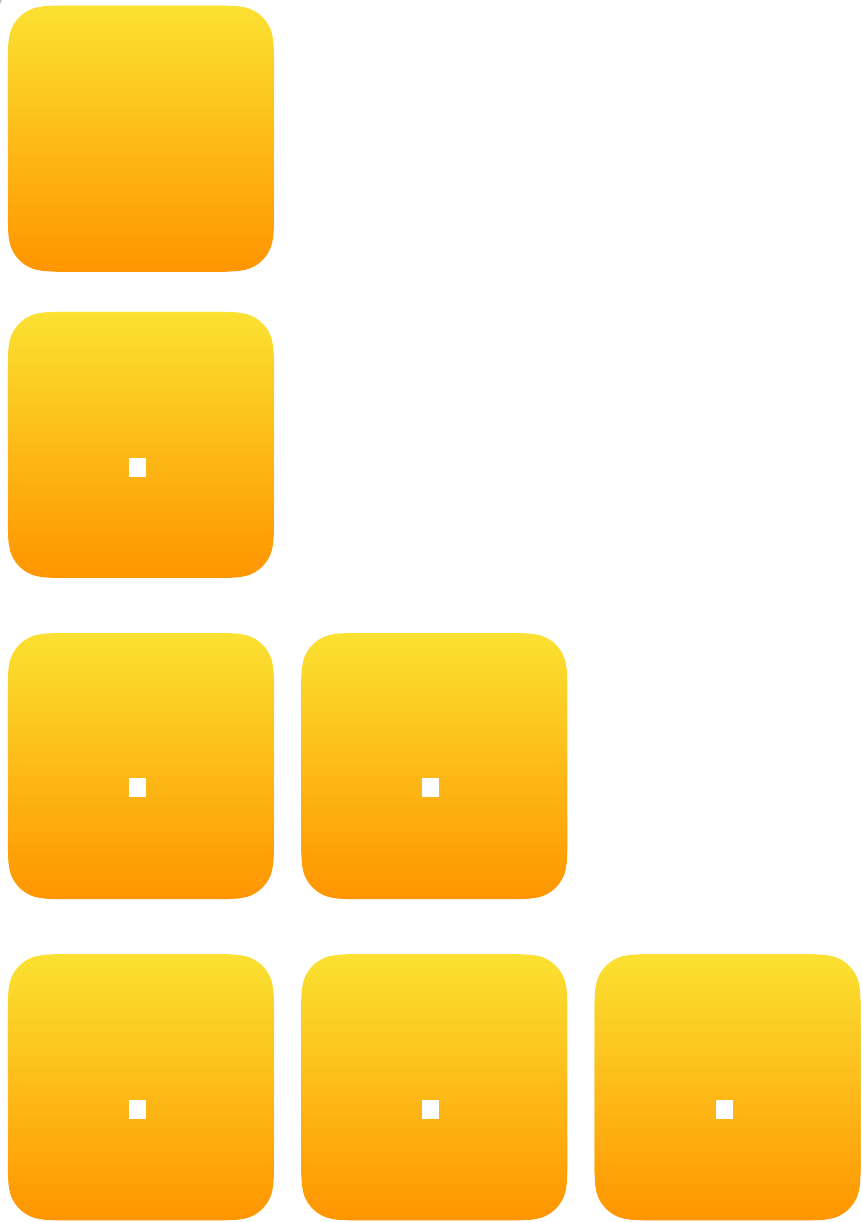
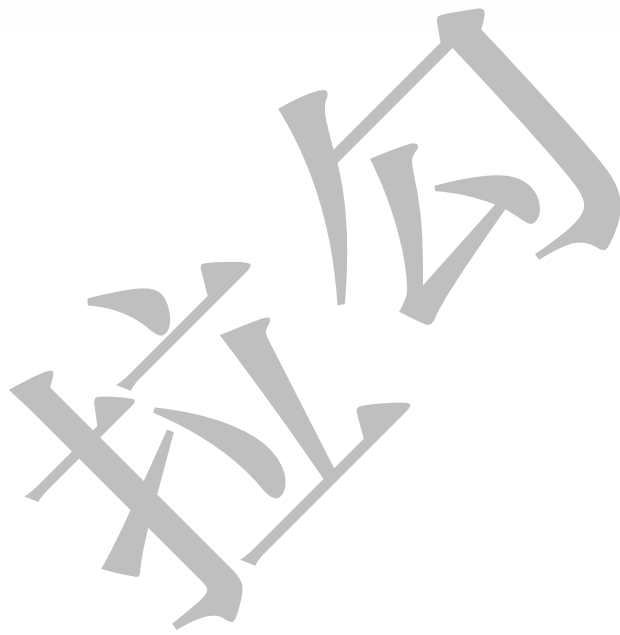
a*



10. 正则表达式匹配

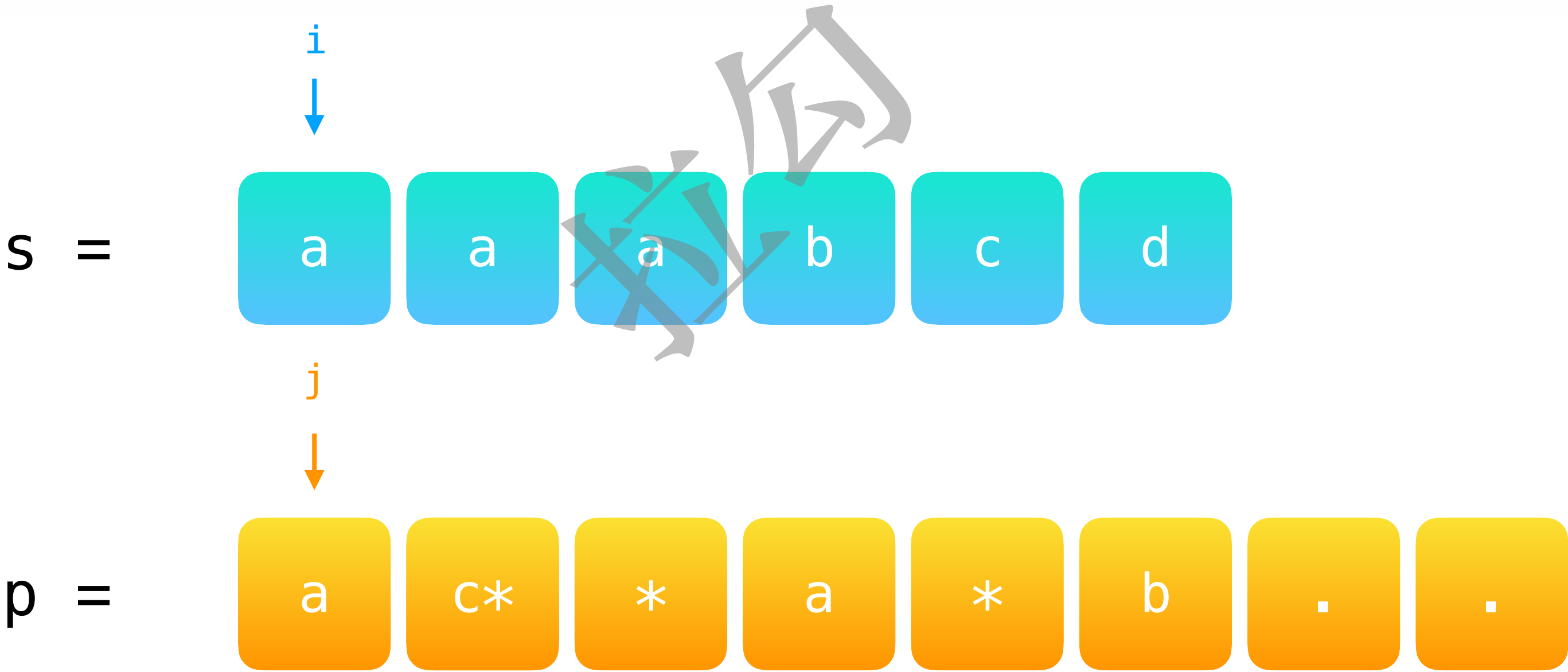
解题思路

- ▶ p 字符串中的出现的星匹配符 ‘*’



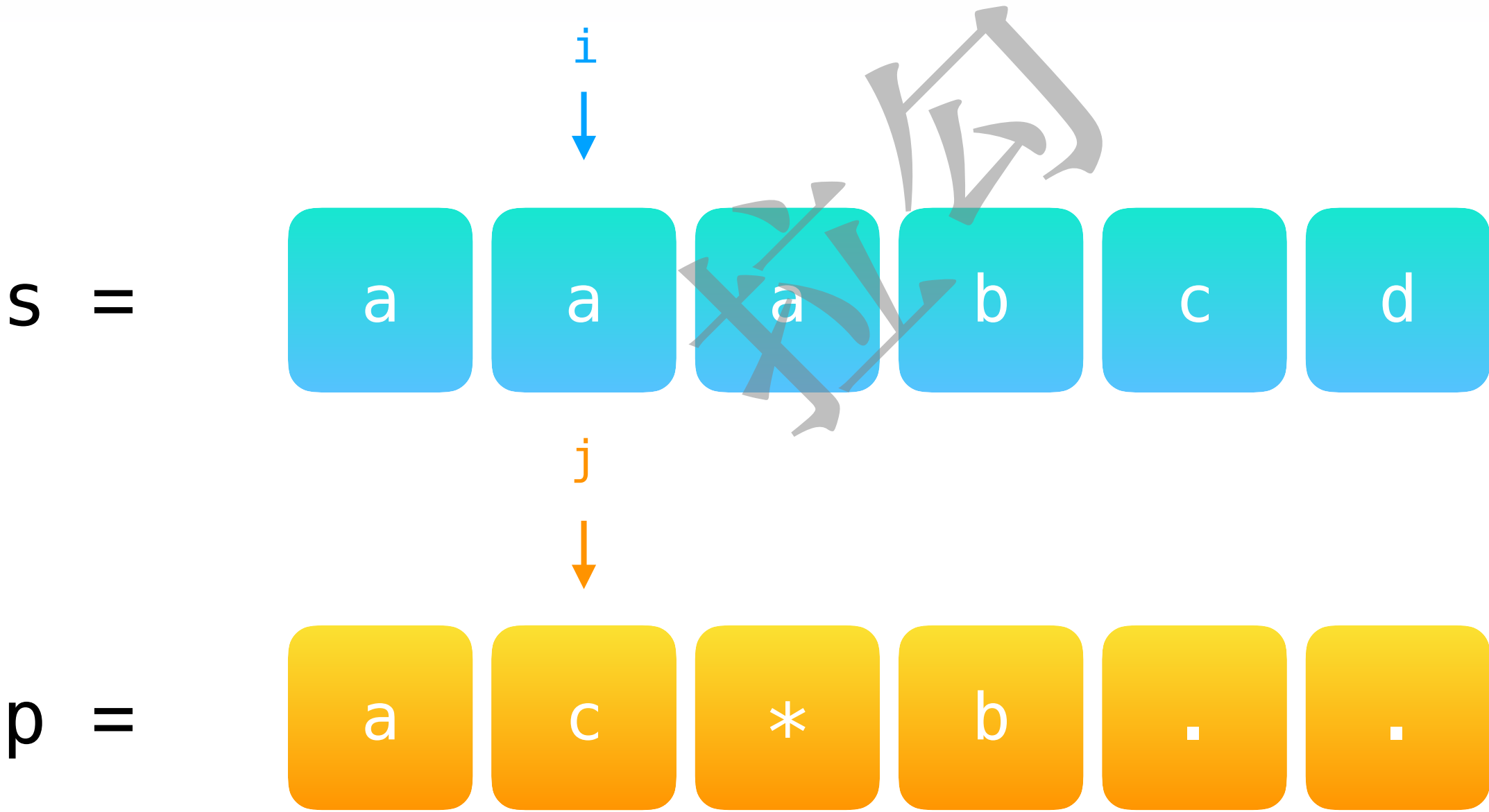
10. 正则表达式匹配

解题思路



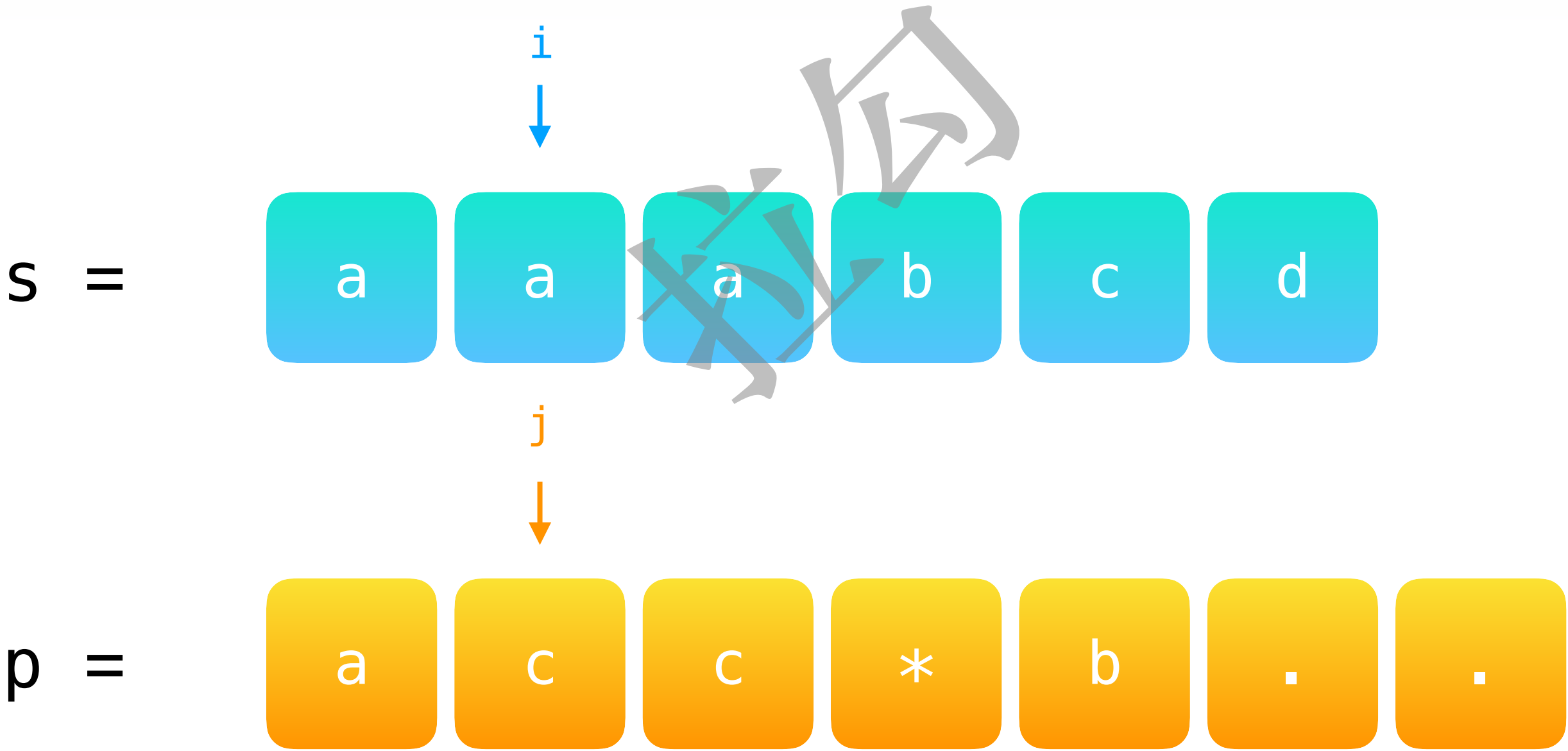
10. 正则表达式匹配

解题思路



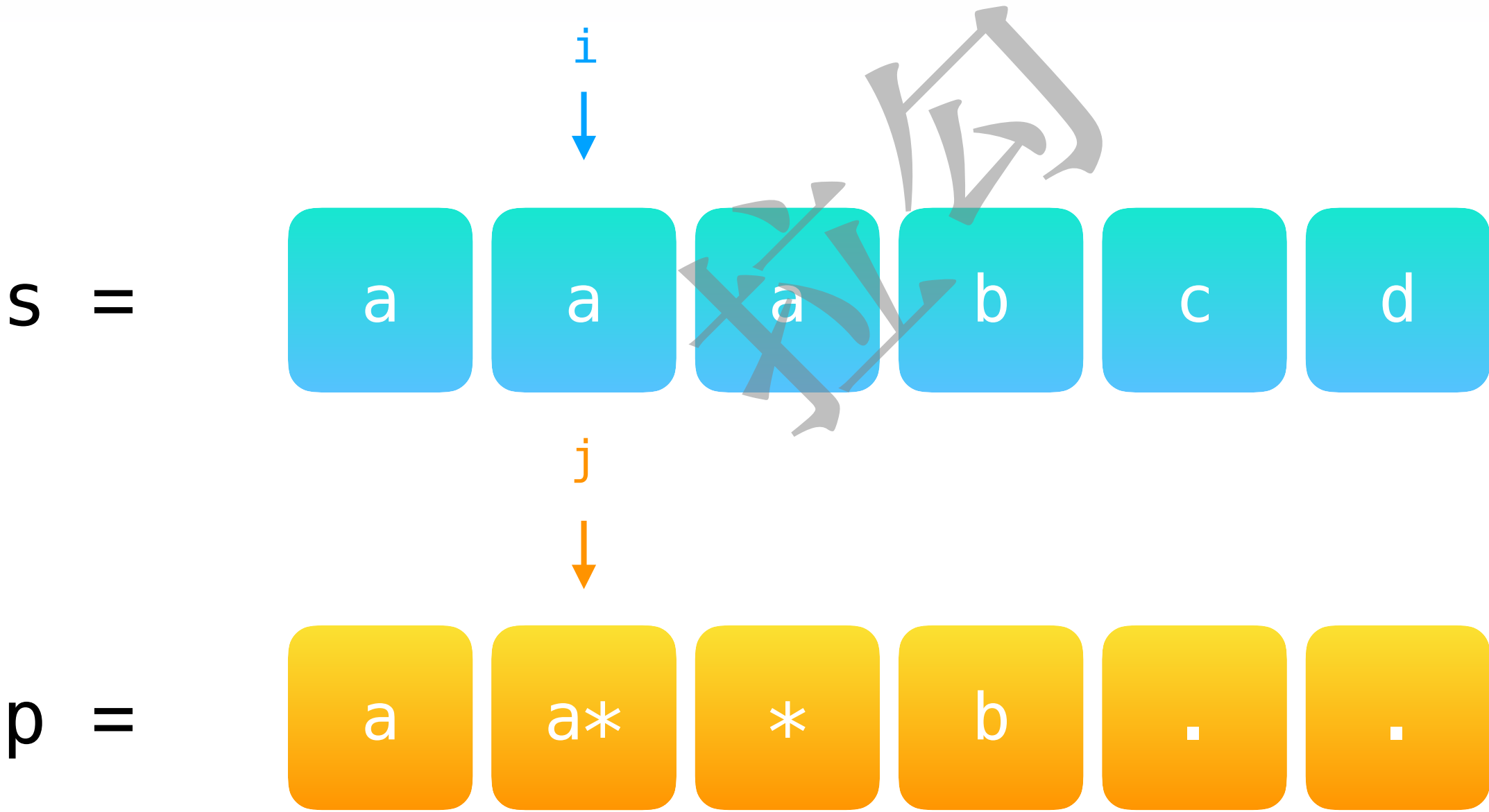
10. 正则表达式匹配

解题思路



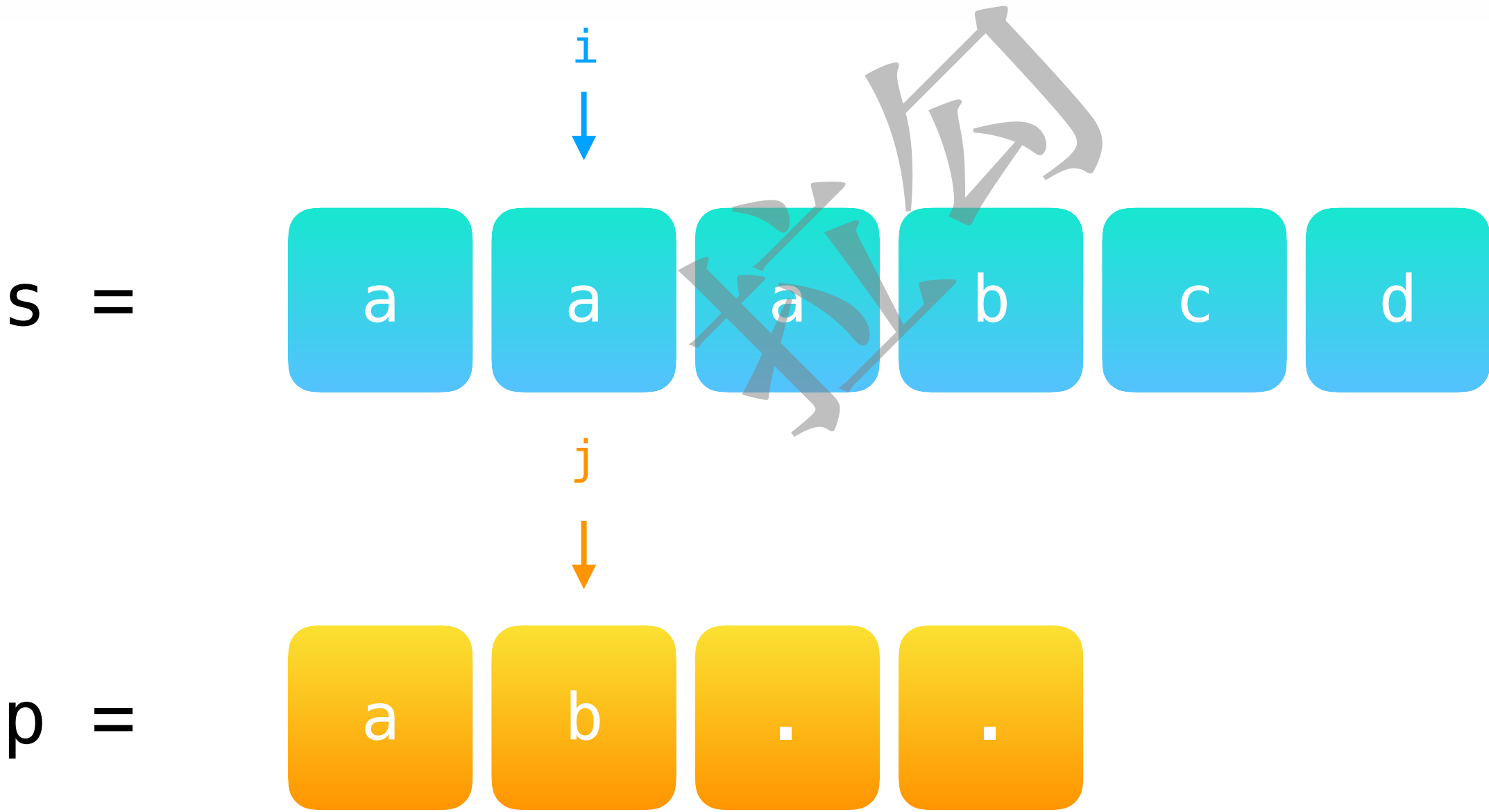
10. 正则表达式匹配

解题思路



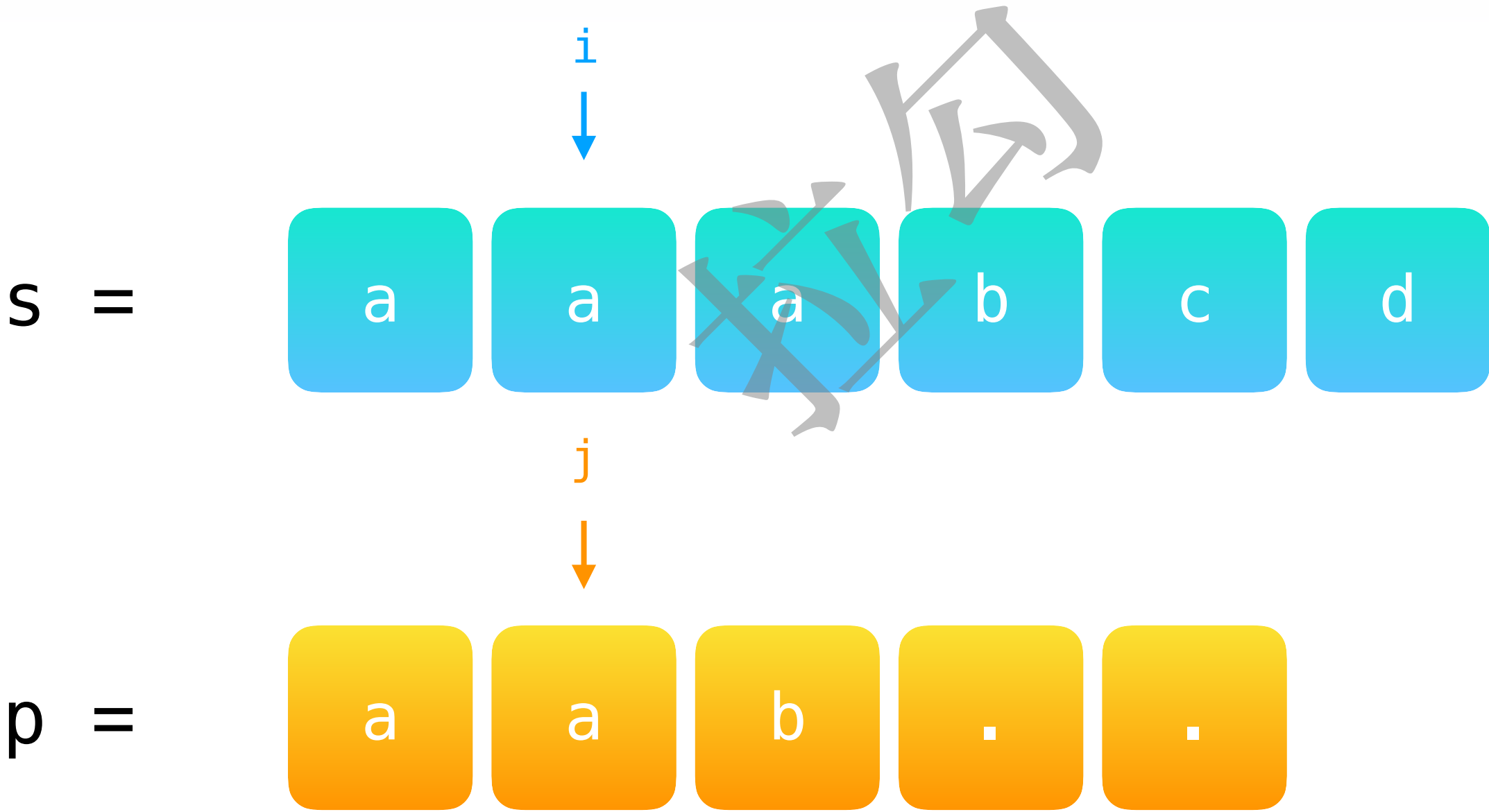
10. 正则表达式匹配

解题思路



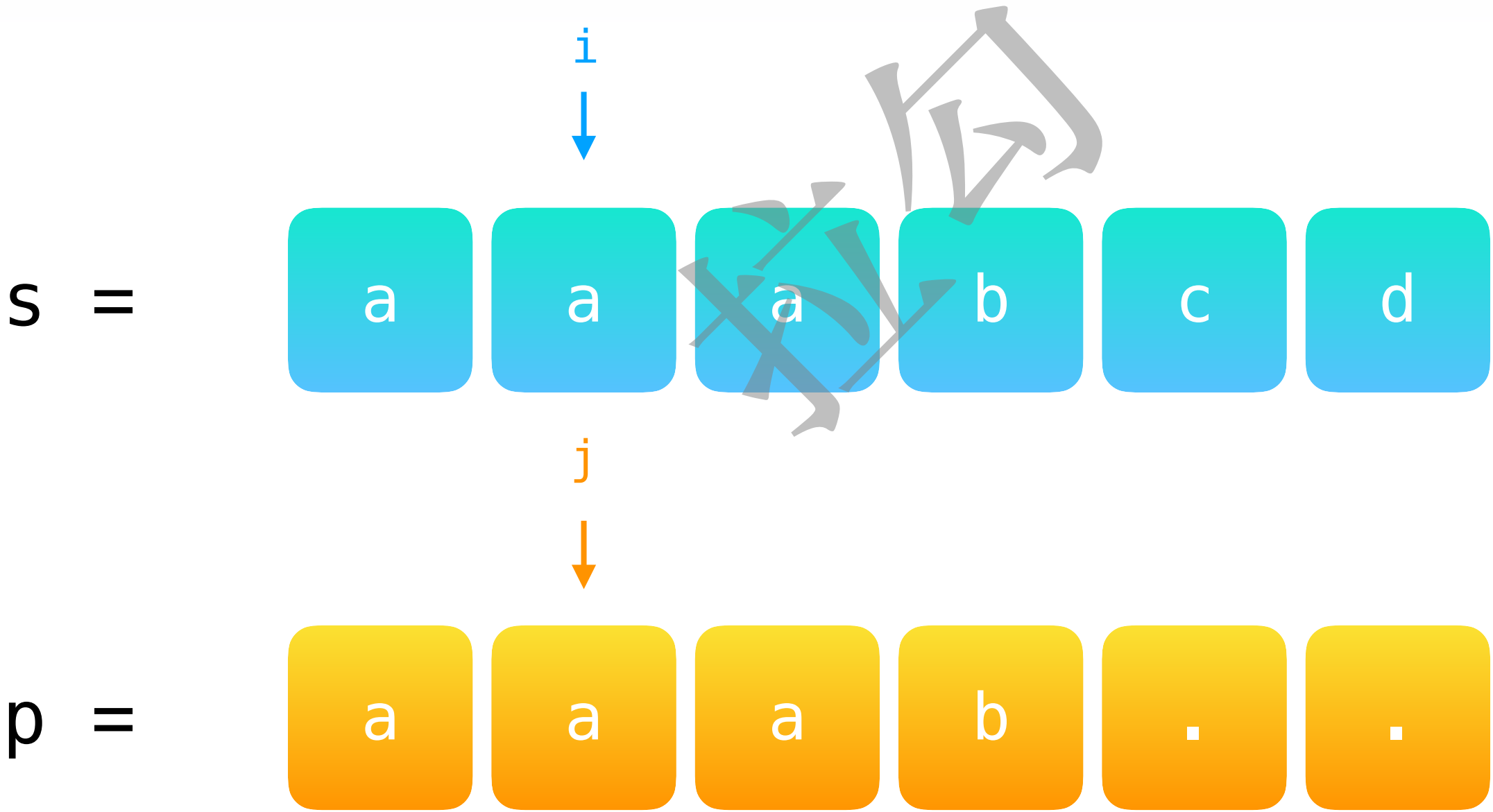
10. 正则表达式匹配

解题思路



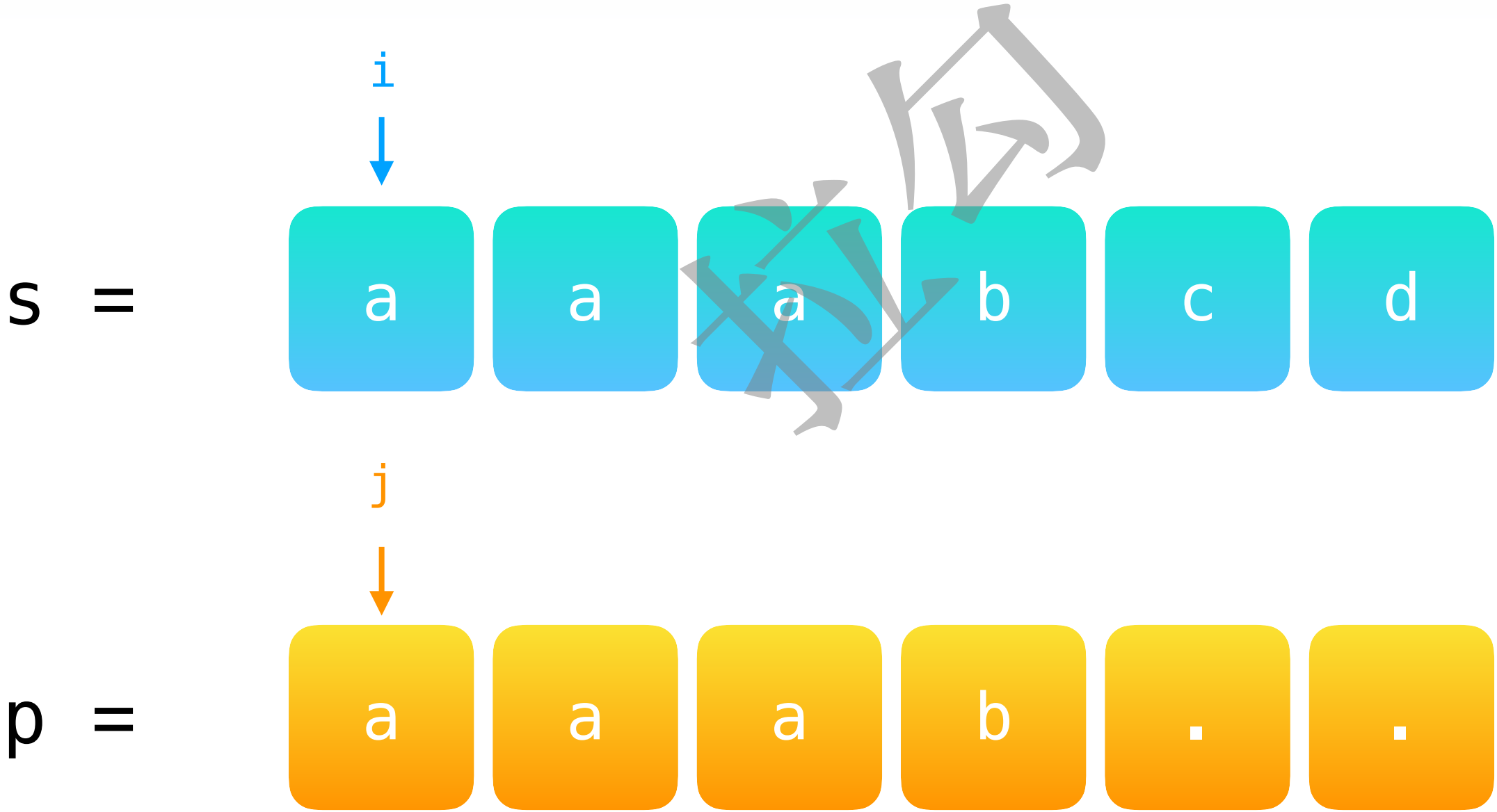
10. 正则表达式匹配

解题思路



10. 正则表达式匹配

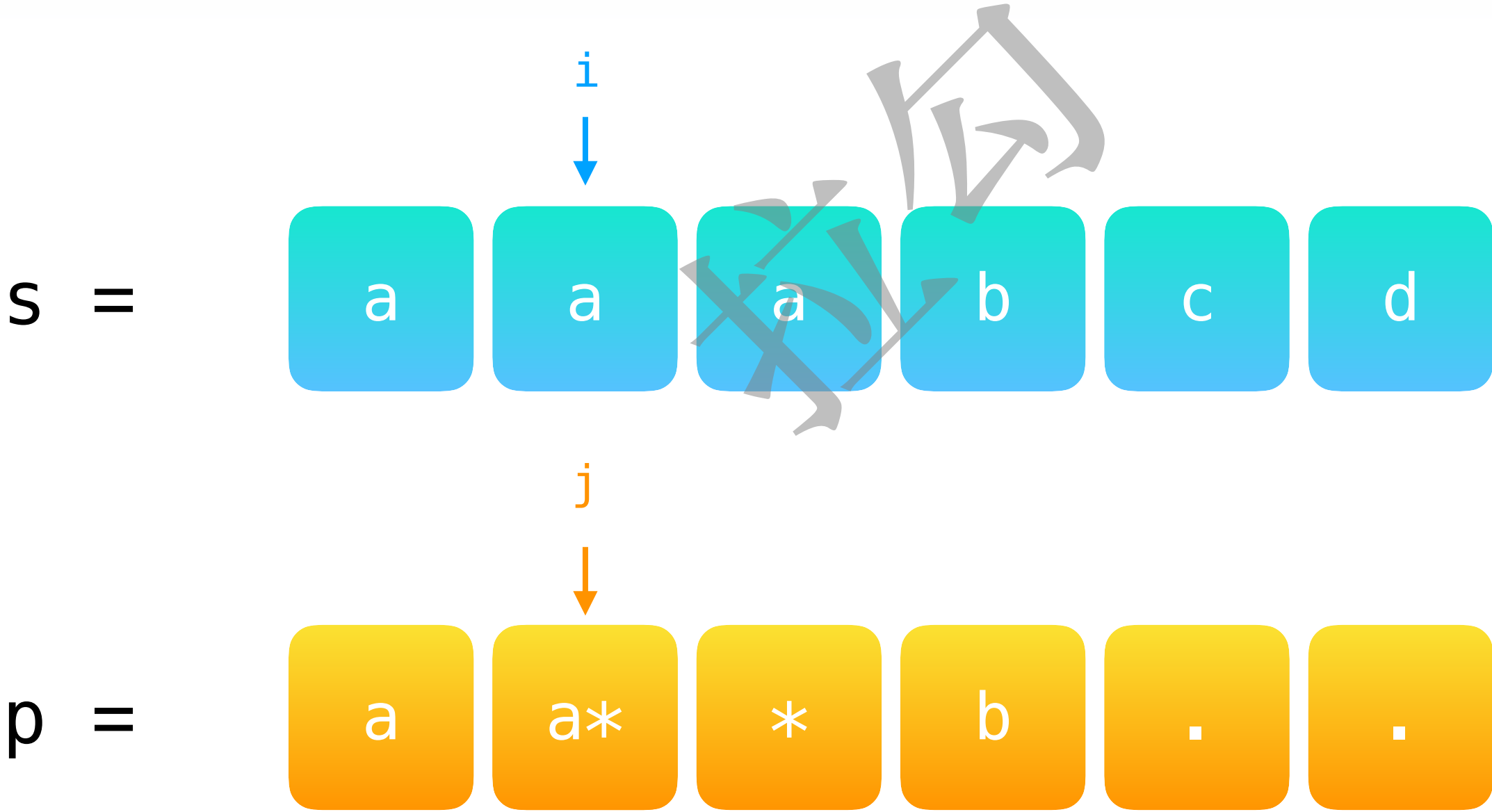
递归解法



```
isMatch(String s, int i, String p, int j)
```

10. 正则表达式匹配

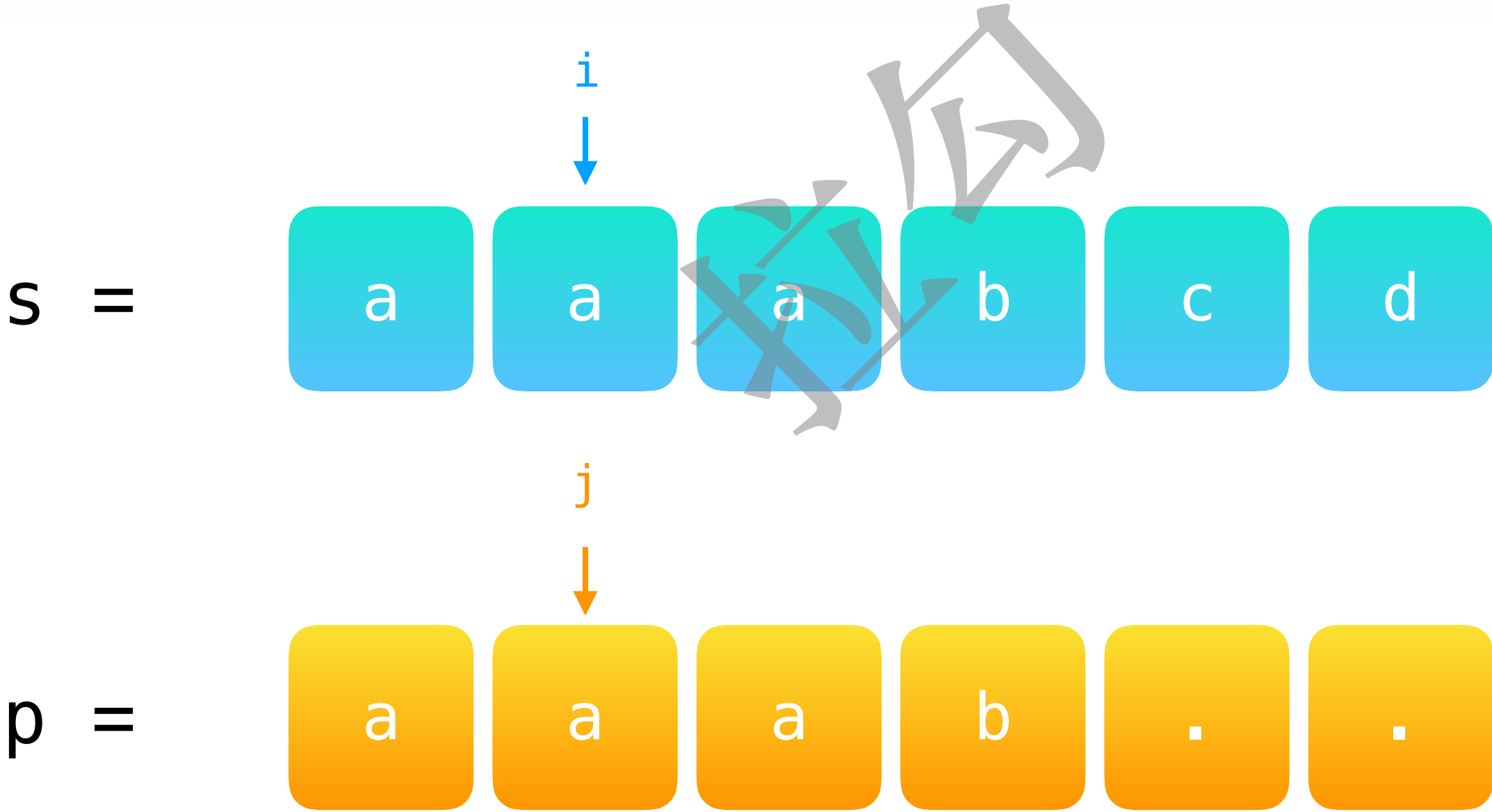
递归解法



`isMatch(s, i, p, j + 2)`

10. 正则表达式匹配

递归解法



```
isMatch(s, i + 1, p, j + 2)
```

```
boolean isMatch(String s, String p) {  
    if (s == null || p == null) {  
        return false;  
    }  
  
    return isMatch(s, 0, p, 0);  
}
```

- 简单判断：只要 s 和 p 有一个为 null，则不匹配
- 调用递归函数：指针 i 和 j 都指向 0 的位置

```
boolean isMatch(String s, int i, String p, int j) {  
    int m = s.length();  
    int n = p.length();
```

```
// 看看pattern和字符串是否都扫描完毕
```

```
if (j == n) {  
    return i == m;  
}
```

- 函数接收四个输入参数：
s 字符串，p 字符串，i 指针，j 指针
- 一开始计算一下 s 字符串和 p 字符串的长度，
分别标记为 m 和 n
- 考虑递归函数结束的时间：
当 j 指针遍历完 p 字符串后，即可跳出递归
而当 i 指针也刚好遍历完，说明 s 和 p 完全匹配


```
// next char is not '*': 必须满足当前字符并递归到下一层
```

```
if (j == n - 1 || p.charAt(j + 1) != '*') {  
    return (i < m) &&  
        (p.charAt(j) == '.' || s.charAt(i) == p.charAt(j)) &&  
        isMatch(s, i + 1, p, j + 1);  
}
```

```
// next char is '*', 如果有连续的s[i]出现并且都等于p[j], 一直  
尝试下去
```

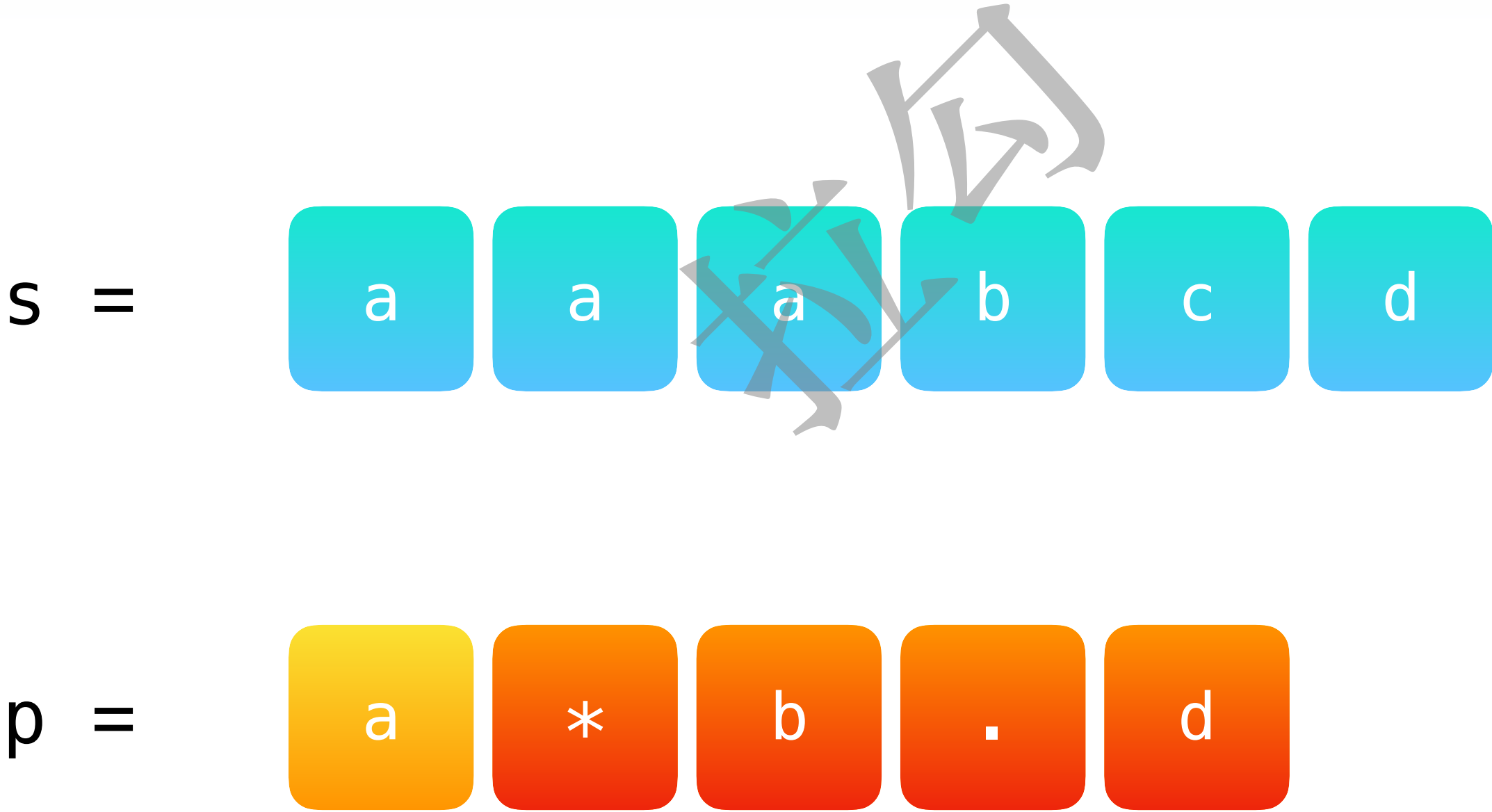
```
if (j < n - 1 && p.charAt(j + 1) == '*') {  
    while ((i < m) && (p.charAt(j) == '.' || s.charAt(i) ==  
p.charAt(j))) {  
        if (isMatch(s, i, p, j + 2)) {  
            return true;  
        }  
        i++;  
    }  
}
```

```
// 接着继续下去  
return isMa
```

- 接下来看看 j 指针的下一个是否为星号，如果不是星号，则递归地调用 isMatch 函数
- 如果 j 指向的字符下一个为星号，则不断地将它和星号作为一个整体，分别表示空字符串，一个 j 指向的字符，两个字符，以此类推。
如果其中一种情况能出现 s 和 p 匹配，则返回 true
- while 循环 - 整个递归算法的核心
 - i 指向的字符必须要能和 j 指向的字符匹配
其中 j 指向的可能是点匹配符
 - 如果无法匹配，则 i++，
表示用星号组合去匹配更长的一段字符串
- 当 i 与 j 指向的字符不相同，或 i 已遍历完 s 字符串，同时 j 指向的字符后跟着一个星号的情况，我们只能用型号组合去表示一个空字符串，然后递归下去

10. 正则表达式匹配

递归解法 二



10. 正则表达式匹配

递归解法 二

s =

a

a

p =

a

*

```
boolean isMatch(String s, String p) {  
    if (s == null || p == null) return false;  
  
    return isMatch(s, s.length(), p, p.length());  
}
```

- 主函数简单判断：
如果 s 和 p 有一个为 null，返回 false
- 调用递归函数

```
boolean isMatch(String s, int i, String p, int j) {  
    if (j == 0) return i == 0;  
  
    if (i == 0) {  
        return j > 1 && p.charAt(j - 1) == '*' && isMatch(s, i, p, j - 2);  
    }  
  
    if (p.charAt(j - 1) != '*') {  
        return isMatch(s.charAt(i - 1), p.charAt(j - 1)) &&  
            isMatch(s, i - 1, p, j - 1);  
    }  
  
    return isMatch(s, i, p, j - 2) || isMatch(s, i - 1, p, j) &&  
        isMatch(s.charAt(i - 1), p.charAt(j - 2));  
}  
  
boolean isMatch(char a, char b) {  
    return a == b || b == '.';  
}
```

- 递归函数四个输入参数：字符串 s ，当前字符串 s 的下标，字符串 p ，字符串 p 的当前下标，由主函数可知，两个字符串的下标都是从最后一位开始
- 如果 p 字符串为空， s 字符串也为空，表示匹配
- 如果 p 字符串不为空，而 s 字符串为空
 - 类似之前的例子：当 s 为空字符串，而 p 为 a^*
 - 只要 p 总是由星号组合构成，则一定满足匹配，否则不行
- 当 p 的当前字符不是星号时，判断当前两个字符是否相等，如果相等，则递归地看前面的字符
- 否则，当 p 的当前字符是星号时，进行两种尝试：
 - 用星号组合表示空字符串，看看是否能匹配
 - 用星号组合表示一个字符，看看是否能匹配

```
boolean isMatch(String s, String p) {  
    int m = s.length(), n = p.length();
```

```
    boolean[][] dp = new boolean[m + 1][n + 1];
```

```
    dp[0][0] = true;
```

```
    for (int j = 1; j <= n; j++) {  
        dp[0][j] = j > 1 && p.charAt(j - 1) == '*' && dp[0][j - 2];  
    }
```

动态规划 - 自底向上

- 分别用 m 和 n 表示 s 字符串和 p 字符串的长度
- 定义一个二维布尔矩阵 dp
- 初始化 $dp[0][0]$ 等于 `true`, 表示当两字符串长度都为 0, 也就是空字符串时, 它们互相匹配
- **【重要】** 初始化二维矩阵第一行的所有值:
 - 当 s 为空字符串时, 对 p 字符串的任一位置, 要使得这个位置的子串能和空字符串匹配, 要求, 这个子串都是由一系列的星号组合构成

```
for (int i = 1; i <= m; i++) {  
    for (int j = 1; j <= n; j++) {  
        if (p.charAt(j - 1) != '*') {  
            dp[i][j] = dp[i - 1][j - 1] &&  
                isMatch(s.charAt(i - 1), p.charAt(j - 1));  
        } else {  
            dp[i][j] = dp[i][j - 2] || dp[i - 1][j] &&  
                isMatch(s.charAt(i - 1), p.charAt(j - 2));  
        }  
    }  
}  
  
return dp[m][n];  
}  
  
boolean isMatch(char a, char b) {  
    return a == b || b == '.';  
}
```

动态规划 - 自底向上

- 对二维矩阵填表：逻辑与递归一样
- p 的当前字符不是星号时，判断当前两字符是否相等，如果相等，则看看 $dp[i-1][j-1]$ 的值，因为它保存了前一个匹配的结果
- 当 p 的当前字符是星号时，进行两种尝试：
 - 用星号组合表示空字符串，看看是否能匹配，即 $dp[i][j - 2]$
 - 用星号组合表示一个字符，看看是否能匹配，即 $dp[i - 1][j]$
- 复杂度分析：运用动态规划，我们把时间复杂度控制在 $O(n^2)$ ，空间复杂度也是 $O(n^2)$

84. 柱状图中最大的矩形

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。
每个柱子彼此相邻，且宽度为 1。

求在该柱状图中，能够勾勒出来的矩形的最大面积。

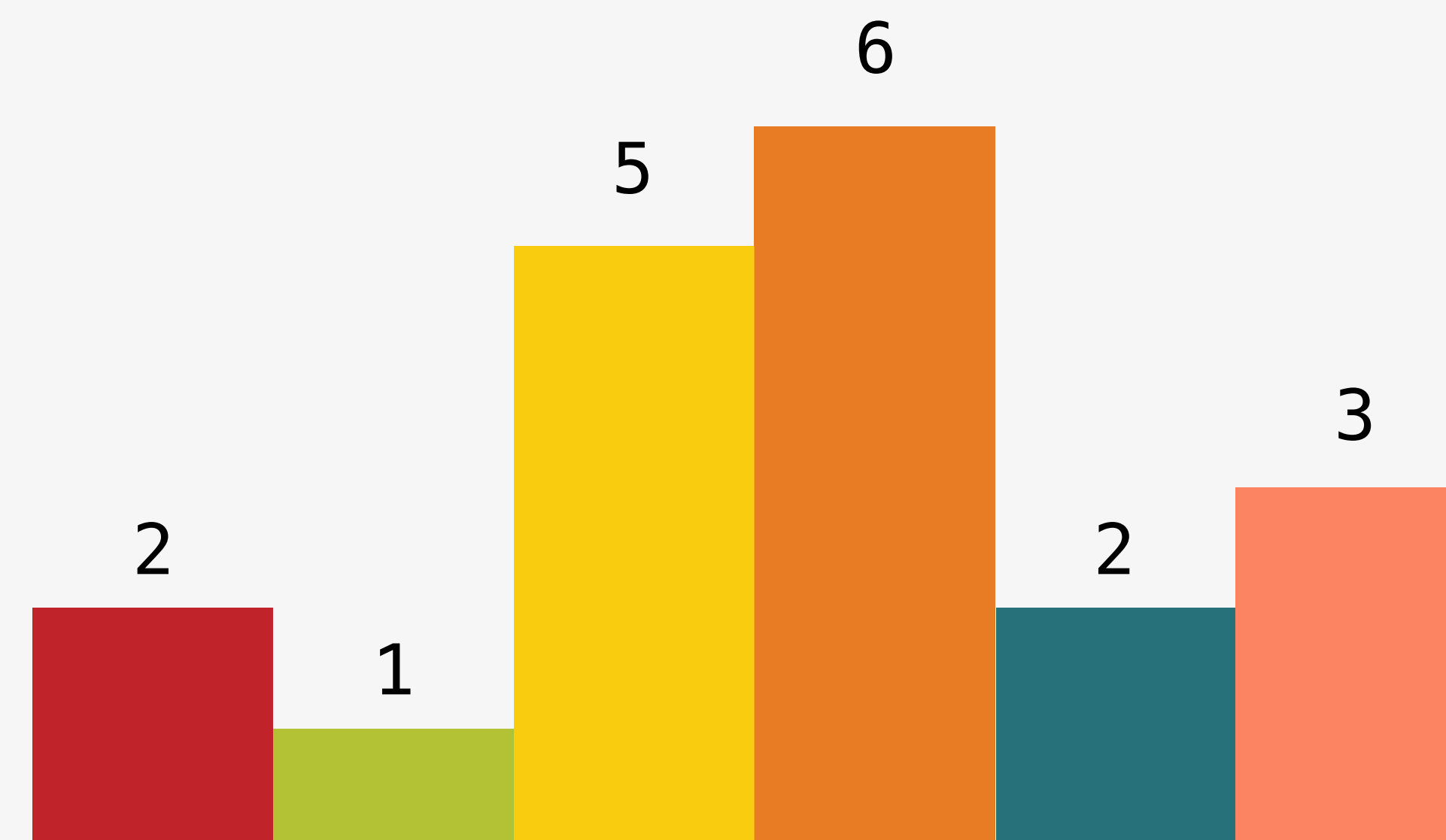
右图是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 $[2, 1, 5, 6, 2, 3]$ 。

图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

示例 1:

输入: $[2, 1, 5, 6, 2, 3]$

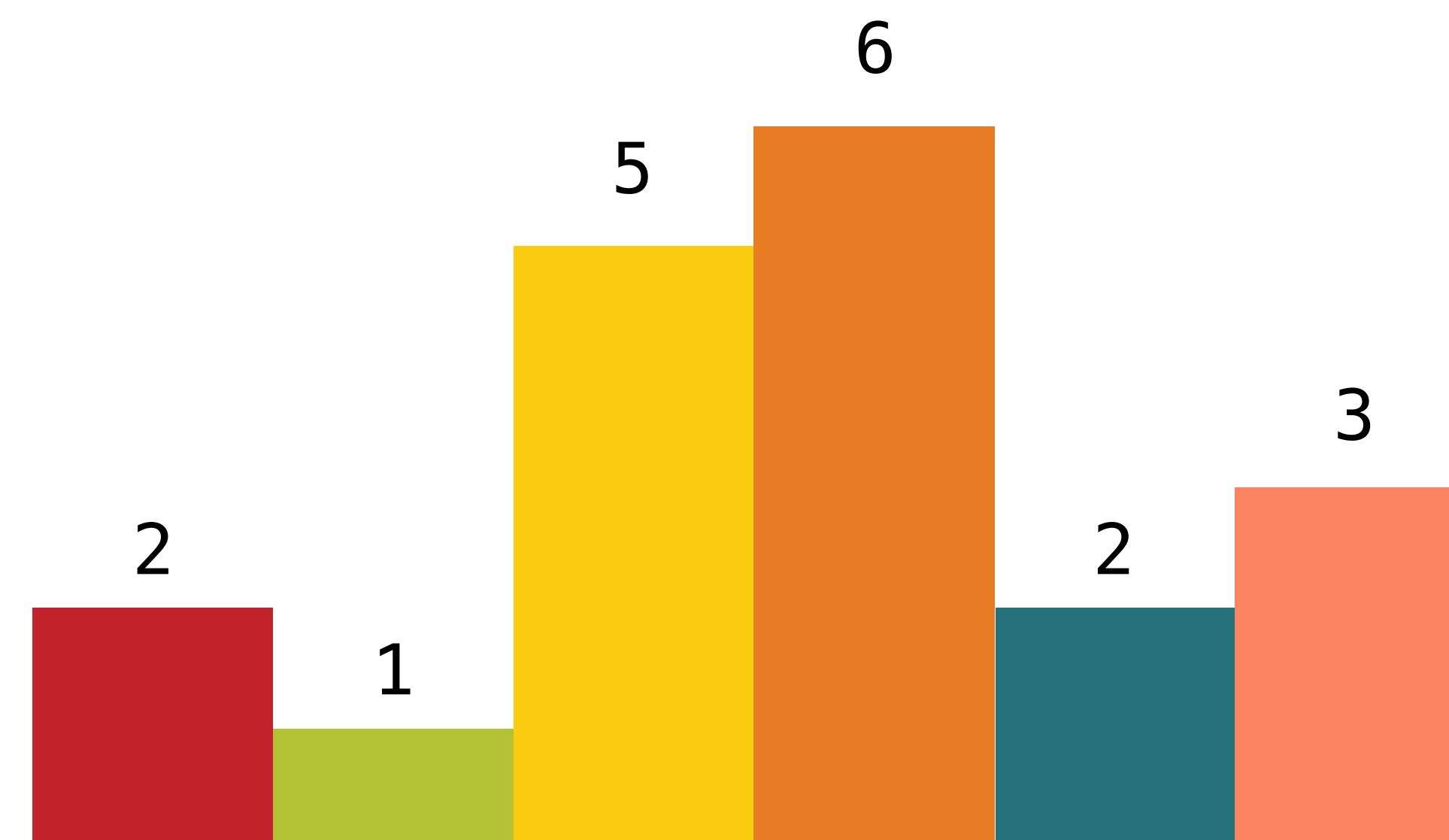
输出: 10



84. 柱状图中最大的矩形

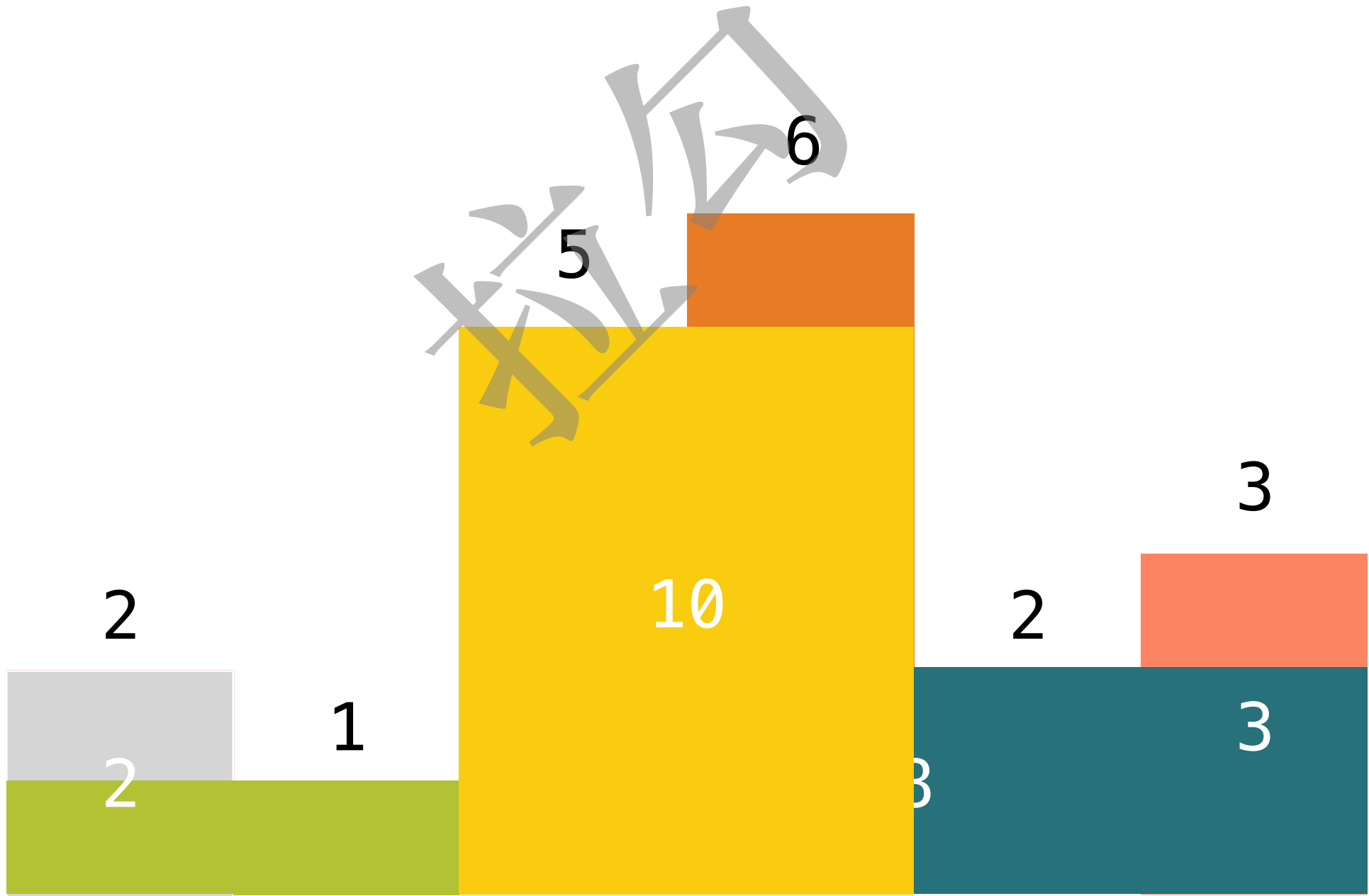
暴力法

- 从左到右扫描输入的数组
- 将每根柱子的高度作为当前矩形的高度
- 矩形的宽度就从当前柱子出发一直延伸到左边和右边
- 一旦遇到低于当前高度的柱子就停止
- 计算面积，最后统计所有面积里的最大值



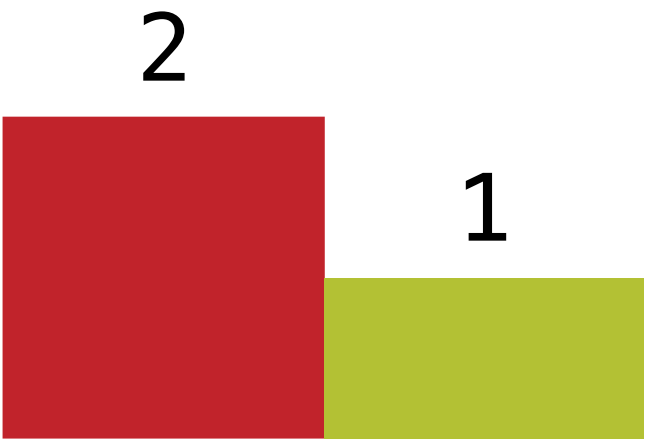
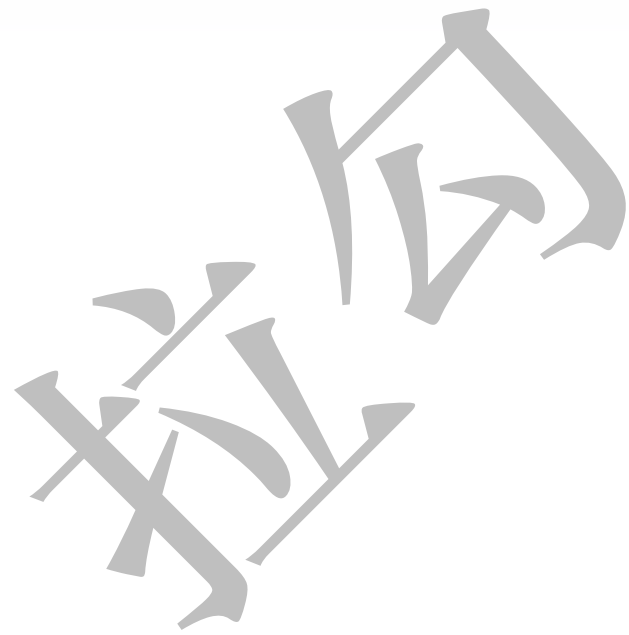
84. 柱状图中最大的矩形

暴力法

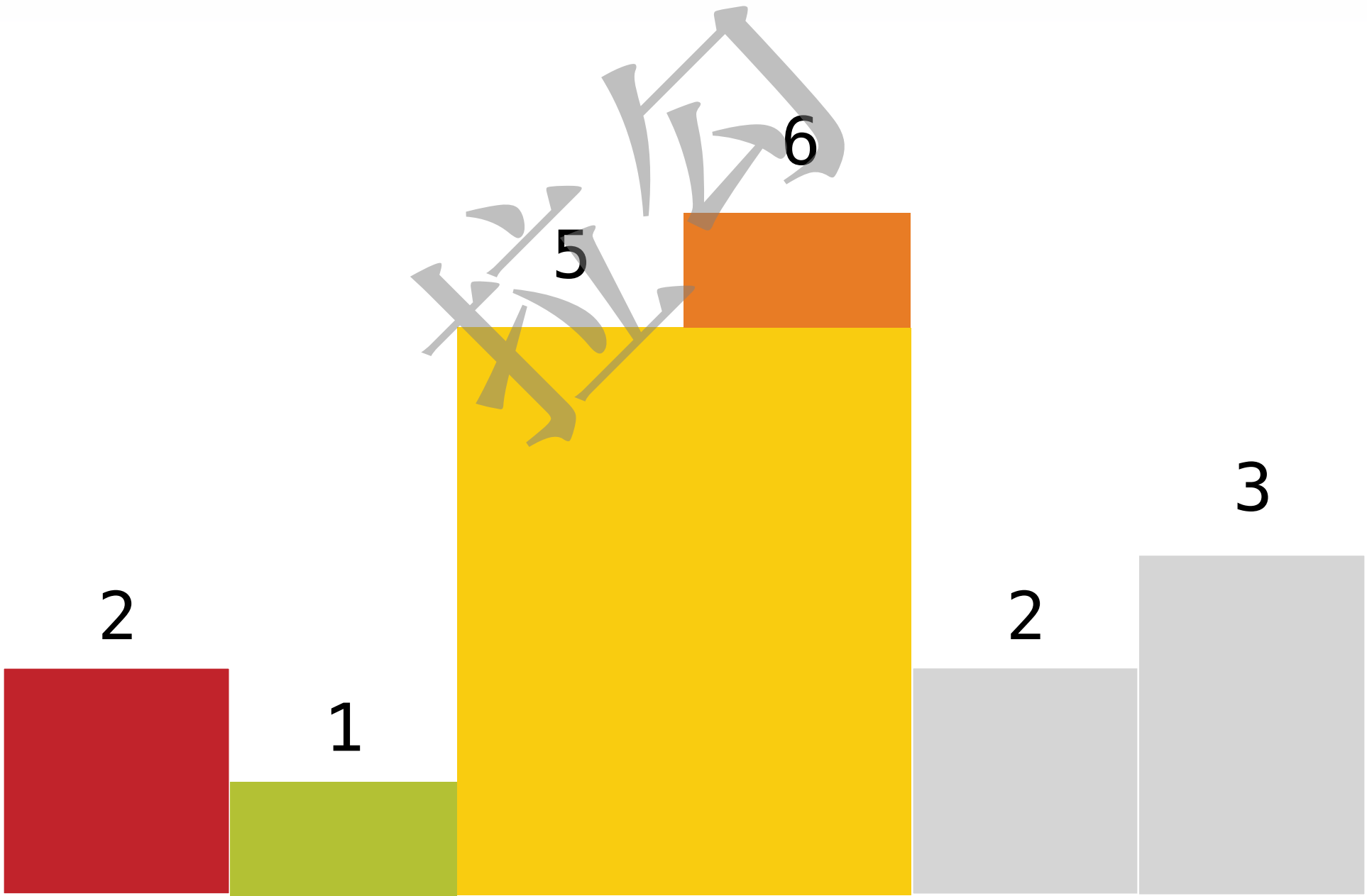


时间复杂度 $O(n^2)$

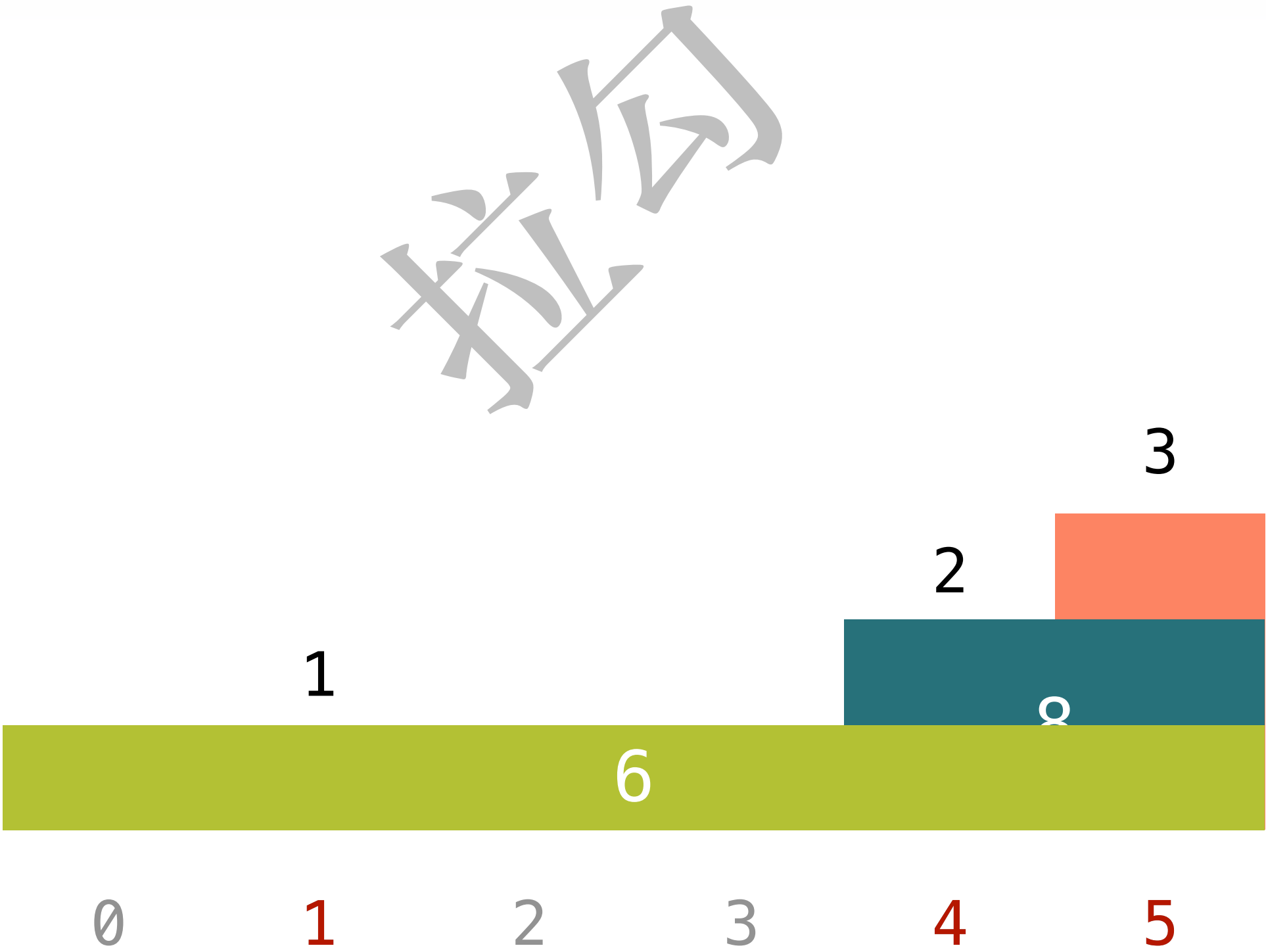
84. 柱状图中最大的矩形



84. 柱状图中最大的矩形



84. 柱状图中最大的矩形



$$\begin{aligned} i &= 1 - 4 \equiv 6 - 1 - 4 \equiv 1 \\ i &= 1 - 1 \equiv 6 - 1 - 1 \equiv 4 \\ S &= 3 \times 1 \equiv 3 \\ S &= 2 \times 4 \equiv 8 \end{aligned}$$

```
int largestRectangleArea(int[] heights) {  
    int n = heights.length, max = 0;  
  
    Stack<Integer> stack = new Stack<>();  
  
    for (int i = 0; i <= n; i++) {  
        while (  
            !stack.isEmpty() &&  
            (i == n || heights[i] < heights[stack.peek()])  
        ) {  
            int height = heights[stack.pop()];  
            int width = stack.isEmpty() ? i : i - 1 - stack.peek();  
  
            max = Math.max(max, width * height);  
        }  
  
        stack.push(i);  
    }  
  
    return max;  
}
```

- 将输入数组的长度记为 n ，初始化最大面积 max 为 0
- 定义一个堆栈 $stack$ 用来辅助计算
- 接下来从头开始扫描输入数组
- 一旦发现当前高度比堆栈顶端所记录的高度要矮，即可开始对堆栈顶端记录的高度计算面积了
 - 此处巧妙处理了当 i 等于 n 的情况
 - 同时判断当前面积是否为最大值
- 如果当前的高度比堆栈顶端所记录的高度要高，则压入堆栈
- 最后返回面积最大值

```
int largestRectangleArea(int[] heights) {  
    int n = heights.length, max = 0;  
  
    Stack<Integer> stack = new Stack<>();  
  
    for (int i = 0; i <= n; i++) {  
        while (  
            !stack.isEmpty() &&  
            (i == n || heights[i] < heights[stack.peek()])  
        ) {  
            int height = heights[stack.pop()];  
            int width = stack.isEmpty() ? i : i - 1 - stack.peek();  
  
            max = Math.max(max, width * height);  
        }  
  
        stack.push(i);  
    }  
  
    return max;  
}
```

► 复杂度分析:

- 时间复杂度是 $O(n)$,
因为从头到尾扫描数组,
每个元素都被压入堆栈一次, 弹出一次
- 空间复杂度是 $O(n)$,
因为使用一个堆栈来保存各元素的下标,
最坏的情况为各高度按照从矮到高的顺序排列,
我们需要将它们都压入堆栈

28. 实现 strStr()

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置（从 0 开始）。如果不存在，则返回 -1。

说明：

当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0。这与 C 语言的 strstr() 以及 Java 的 indexOf() 定义相符。

示例 1：

输入：

```
haystack = "hello",  
needle = "ll"
```

输出：2

示例 2：

输入：

```
haystack = "aaaaa",  
needle = "bba"
```

输出：-1

28. 实现 strStr()

暴力法

输入：

haystack = "iloveleetcode"

needle = "leetcode"

28. 实现 strStr()

暴力法

输入：

haystack = "iloveleetcode"

needle = "leetcode"

28. 实现 strStr()

暴力法

输入：

haystack = "i~~love~~leetcode"

needle = "le~~et~~code"

28. 实现 strStr()

暴力法

输入：

```
haystack = "iloveleetcode"  
needle   = "leetcode"
```

28. 实现 strStr()

暴力法

输入：

haystack = "iloveleetcode"

needle = "leetcode"

28. 实现 strStr()

暴力法

输入：

haystack = "iloveleetcode"

needle = "leetcode"

28. 实现 strStr()

暴力法

输入：

```
haystack = "iloveleetcode"  
needle   = "leetcode"
```

```
int strStr(String haystack, String needle) {  
    for (int i = 0; ; i++) {  
        for (int j = 0; ; j++) {  
            if (j == needle.length()) return i;  
            if (i + j == haystack.length()) return -1;  
            if (needle.charAt(j) != haystack.charAt(i + j))  
                break;  
        }  
    }  
}
```

- 定义一个 `i` 指针，用来遍历 `haystack` 字符串
- 对于每个 `i` 指针，
用另外一个 `j` 指针来扫描 `needle` 字符串
- 如果 `j` 扫描完毕，
表示在 `haystack` 字符串中找到了 `needle` 字符串
- 如果 `i` 扫描完毕，
表示无法在 `haystack` 字符串中找到 `needle` 字符串，
返回 `-1`
- 如果要比较的两个字符不相等，
则跳出内循环，`i` 指针向前挪一个位置，继续刚才的比较


```
int strStr(String haystack, String needle) {  
    for (int i = 0; ; i++) {  
        for (int j = 0; ; j++) {  
            if (j == needle.length()) return i;  
            if (i + j == haystack.length()) return -1;  
            if (needle.charAt(j) != haystack.charAt(i + j))  
                break;  
        }  
    }  
}
```

- 假设 haystack 字符串长度为 m ,
needle 字符串长度为 n ,
暴力法时间复杂度为 $O(m \times n)$
- 更快的方法? KMP 算法!

28. 实现 strStr()

KMP

- ▶ KMP(Knuth-Morris-Pratt) 是由三个人联合发表的算法
- ▶ 目的：为了在字符串 haystack 中找到另一个字符串 needle 出现的所有位置
- ▶ 核心思想：避免暴力法中出现的不必要的比较

```
haystack = "ABC ABCDAB ABCDABCDABDE"
```

```
needle   = "ABCDABD"
```

28. 实现 strStr()

KMP

haystack = "ABC ABCDAB ABCDABCDABDE"
needle = "ABCDABD"

28. 实现 strStr()

KMP

haystack = "ABC ABCDAB ABCDABCDABDE"

needle = "ABCDABD"

28. 实现 strStr()

KMP

```
haystack = "ABC ABCDAB ABCDABCDABDE"  
needle   = "ABCDABD"
```

- ▶ 为什么不需要慢慢挪动来比较，而是可以跳跃式地比较呢？或者说这样跳跃式比较会否错过一些可能性？
- ▶ 为什么知道能直接跳跃到这个位置呢？

28. 实现 strStr()

KMP - 重要数据结构：最长的公共前缀和后缀 (Longest Prefix and Suffix, 简称 LPS)

- ▶ 它是一个数组
- ▶ 记录了字符串从头开始到某个位置结束的一段字符串中，公共前缀和后缀的最大长度
- ▶ 公共前缀和后缀，即字符串的前缀等于后缀，并且，前缀和后缀不能是同一段字符串

```
needle = "ABCDABD"
```

```
LPS    = {0000120}
```

28. 实现 strStr()

KMP

- ▶ $LPS[0] = 0$ ，表示字符串“A”的最长公共前缀和后缀的长度为 0。
 - 注意：前缀和后缀不能是同一段字符串
- ▶ $LPS[1] = 0$ ，表示字符串“AB”的最长公共前缀和后缀的长度为 0。
 - 因为该字符串只有一个前缀 A 和后缀 B
- ▶ $LPS[4] = 1$ ，表示字符串“ABCDA”的最长公共前缀和后缀的长度为 1。
- ▶ $LPS[5] = 2$ ，表示字符串“ABCDAB”的最长公共前缀和后缀的长度为 2。
 - 前缀有：A，AB，ABC，ABCD；后缀有：BCDA，CDA，DA，A
 - 该字符串有很多前缀和后缀
 - 两个相同并且长度最长的是 A，则 LPS 为 1
 - 前缀有：A，AB，ABC，ABCD，ABCD A，后缀有：BCDAB，CDAB，BAD，AB，B
 - 两个相同并且长度最长的是 AB，则 LPS 为 2

needle = “ABCDABD”

LPS = {0000120}

28. 实现 strStr()

KMP

haystack = "ABC **ABCDAB** ABCDABCDABDE"

needle = "A**BCDAB**D"

LPS = 2

28. 实现 strStr()

KMP

haystack = "ABC ABCDAB ABCDABCDABDE"

needle = "ABCDABD"

28. 实现 strStr()

KMP

haystack = "ABC ABCDAB ABCDABCDABDE"

needle = "ABCDABD"

28. 实现 strStr()

KMP

haystack = "ABC ABCDAB ABCDABCDABDE"

needle = "ABCDABD"

28. 实现 strStr()

KMP

haystack = "ABC ABCDAB ABCDABCDABDE"
needle = "ABCDABD"

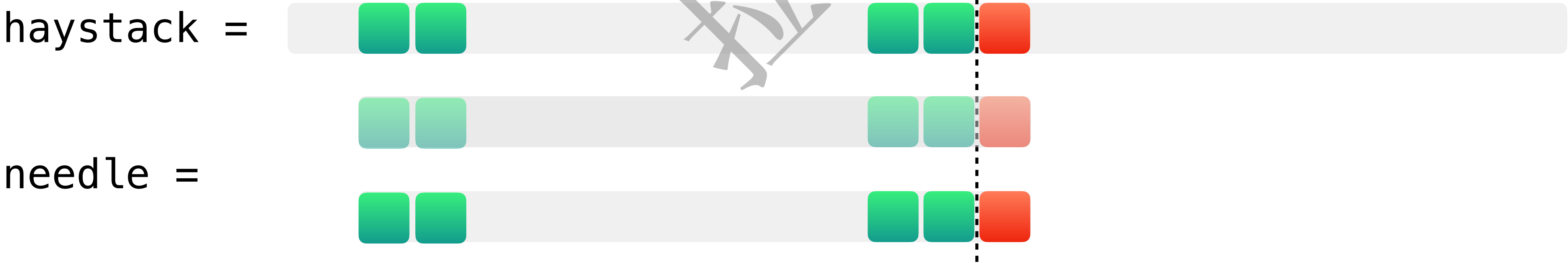
28. 实现 strStr()

KMP



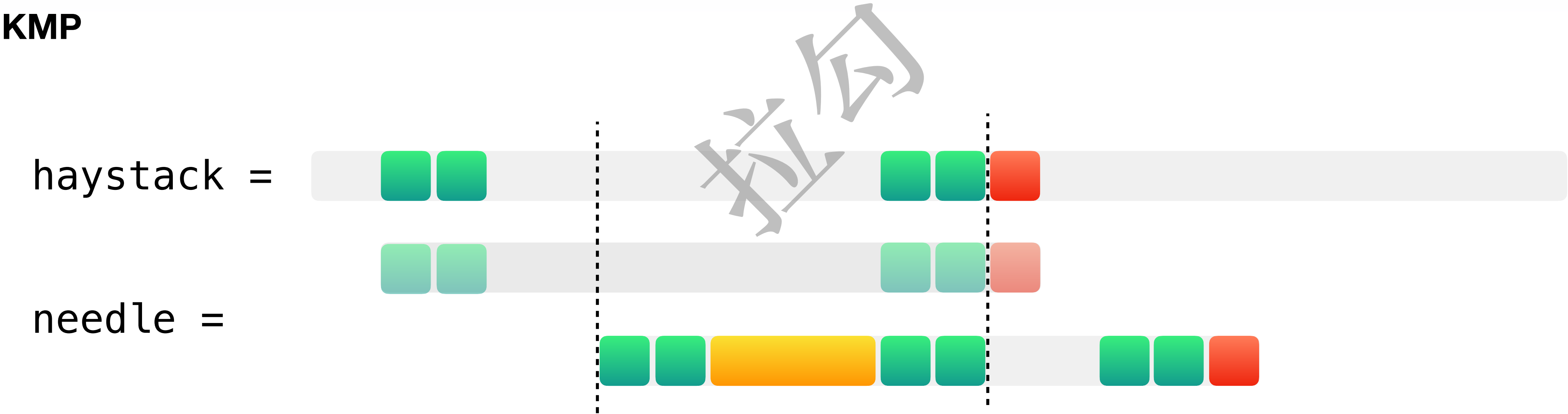
28. 实现 strStr()

KMP



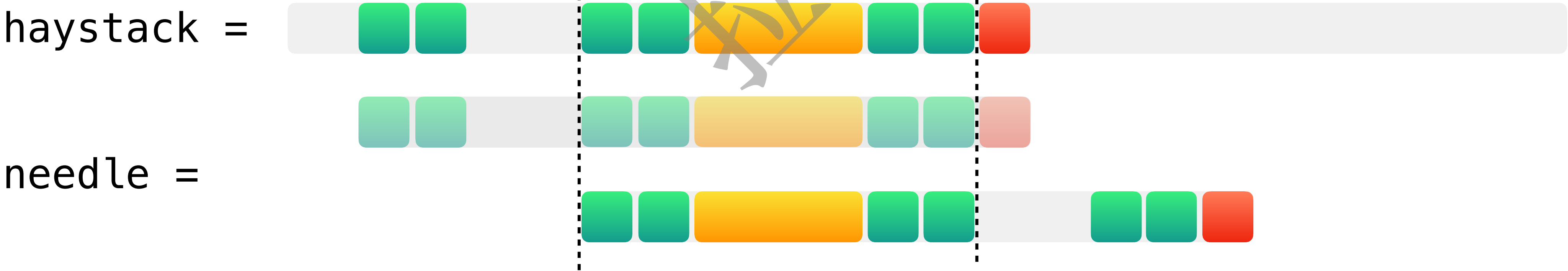
28. 实现 strStr()

KMP



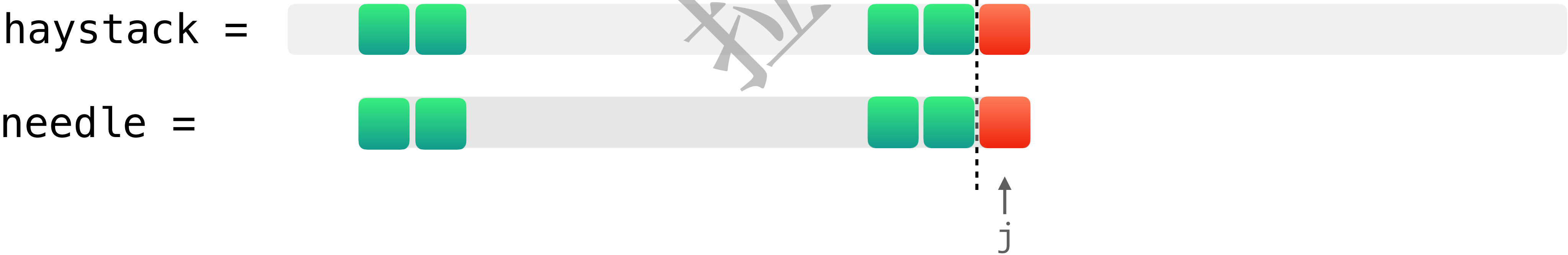
28. 实现 strStr()

KMP



28. 实现 strStr()

KMP



LPS[j - 1] = 0 ... 1 2

j = LPS[j - 1]

28. 实现 strStr()

KMP



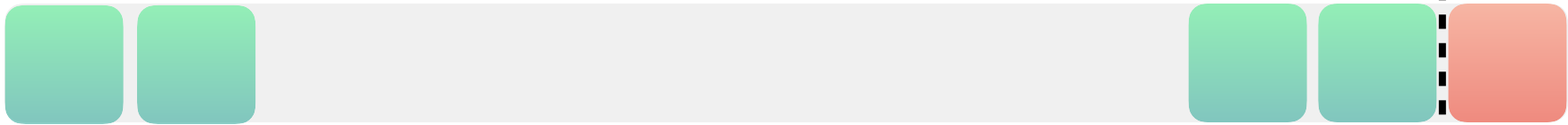
28. 实现 strStr()

KMP

haystack =



needle =



```
int strStr(String haystack, String needle) {
```

```
    int m = haystack.length();
```

```
    int n = needle.length();
```

```
    if (n == 0) {
```

```
        return 0;
```

```
    }
```

```
    int[] lps = getLPS(needle);
```

```
    int i = 0, j = 0;
```

- 用变量 m 记录 haystack 字符串的长度
用变量 n 记录 needle 字符串的长度
- 题目要求：如果 n 等于 0，返回 0
- 求出 needle 的 LPS，即最长的公共前缀和后缀数组
- 定义指针 i 用来扫描 haystack，
定义指针 j 用来扫描 needle

```
while (i < m) {  
    if (haystack.charAt(i) == needle.charAt(j)) {  
        i++; j++;
```

```
        if (j == n) {  
            return i - n;  
        }
```

```
    } else if (j > 0) {  
        j = lps[j - 1];
```

```
    } else {  
        i++;
```

```
    }  
}
```

```
return -1;
```

```
}
```

- 进入循环体，直到 i 扫描完整个 haystack，一旦扫描完还没有发现 needle，则跳出循环
- 在循环体中，当发现 i 指针与 j 指针指向的字符相等时，两个指针一起向前走一步 $i++$ ， $j++$
- 一旦发现 j 已经扫描完 needle 字符串，说明已在 haystack 中找到了 needle，立即返回它在 haystack 中的起始位置
- 在循环体中，当发现 i 指针与 j 指针指向的字符不相同时，尝试进行跳跃操作 $j = \text{LPS}[j - 1]$ ，这里必须判断 j 是否大于 0
- j 等于 0 的情况，表明此时 needle 的第一个字符已不同于 haystack 的字符，尝试对比 haystack 的下一个字符，故 $i++$
- 最终，若未能在 haystack 中找到 needle，返回 -1

28. 实现 strStr()

如何求出 **needle** 字符串的最长公共前缀和后缀数组 - 暴力法

- ▶ 假设此时子串长度为 m
- ▶ 对字符串的每个位置先尝试长度为 $m - 1$ 的前缀和后缀
 - 如果两者一样，则记录下来
 - 如果两者不一样，则尝试长度为 $m - 2$ 的前缀和后缀，以此类推
 - 复杂度为 $O(n^2)$

28. 实现 strStr()

needle =



~~needle[i] != needle[4]~~
needle[i] == needle[4]

28. 实现 strStr()

needle =



假设 $LPS[len - 1] = 3$

28. 实现 strStr()

needle =



- 更新 len = LPS[len - 1]
- 继续比较 needle[i] 和 needle[len]

```
int[] getLPS(String str) {  
    int[] lps = new int[str.length()];  
  
    int i = 1, len = 0;  
  
    while (i < str.length()) {  
        if (str.charAt(i) == str.charAt(len)) {  
            lps[i++] = ++len;  
        } else if (len > 0) {  
            len = lps[len - 1];  
        } else {  
            i++;  
        }  
    }  
  
    return lps;  
}
```

- 初始化一个 LPS 数组用来保存最终的结果
- 由于 LPS 的第一个值一定为 0，即长度为 1 的字符串的最长公共前缀后缀的长度为 0，直接从第二个位置遍历，并初始化当前最长的 LPS 长度为 0，用 len 变量记录一下
- 用指针 i 遍历整个输入字符串
- 如果 i 指针能够延续前缀和后缀，更新 LPS 值为 len + 1
- 否则，判断 len 是否大于 0，然后尝试第二长的前缀和后缀，看看是否能继续延续
- 尝试了所有的前缀和后缀都不行时，则当前的 LPS 为 0，i++
- 最后返回 LPS 数组

28. 实现 strStr()

len = 0, i = 1

i
↓

needle = ADCADB

LPS = 000000

28. 实现 strStr()

len = 0, i = 1

needle = ADCADB

LPS = 000000

lps[1] = 0

28. 实现 strStr()

len = 0, i = 2

i
↓

needle = ADCADB

LPS = 000000

lps[2] = 0

28. 实现 strStr()

len = 0, i = 3

i

needle = ADCADB

LPS = 000100

```
needle[len] = needle[3]
```

```
lps[i++] = ++len
```

28. 实现 strStr()

len = 1, i = 4

i
↓

needle = ADCADB

LPS = 000120

```
needle[len] = needle[i]  
lps[i++] = ++len  
lps[4] = 2
```

28. 实现 strStr()

len = 2, i = 5
↓
needle = ADCAD**B**
LPS = 00012**0**

needle[len] = "C" ≠ needle[i] = "B"
len = lps[len - 1]

28. 实现 strStr()

len = 2, i = 5

i
↓

needle = ADCAD**B**

LPS = 00012**0**

```
len = lps[len - 1]  
needle[len] ≠ needle[i]
```

28. 实现 strStr()

复杂度分析

- 运用了 KMP 算法后，我们需要 $O(n)$ 的时间计算 LPS 数组，需要 $O(m)$ 的时间扫描 haystack 字符串
- 整体时间复杂度为 $O(m + n)$

Next: 课时 11 《大厂算法面试真题 - 难题精讲（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