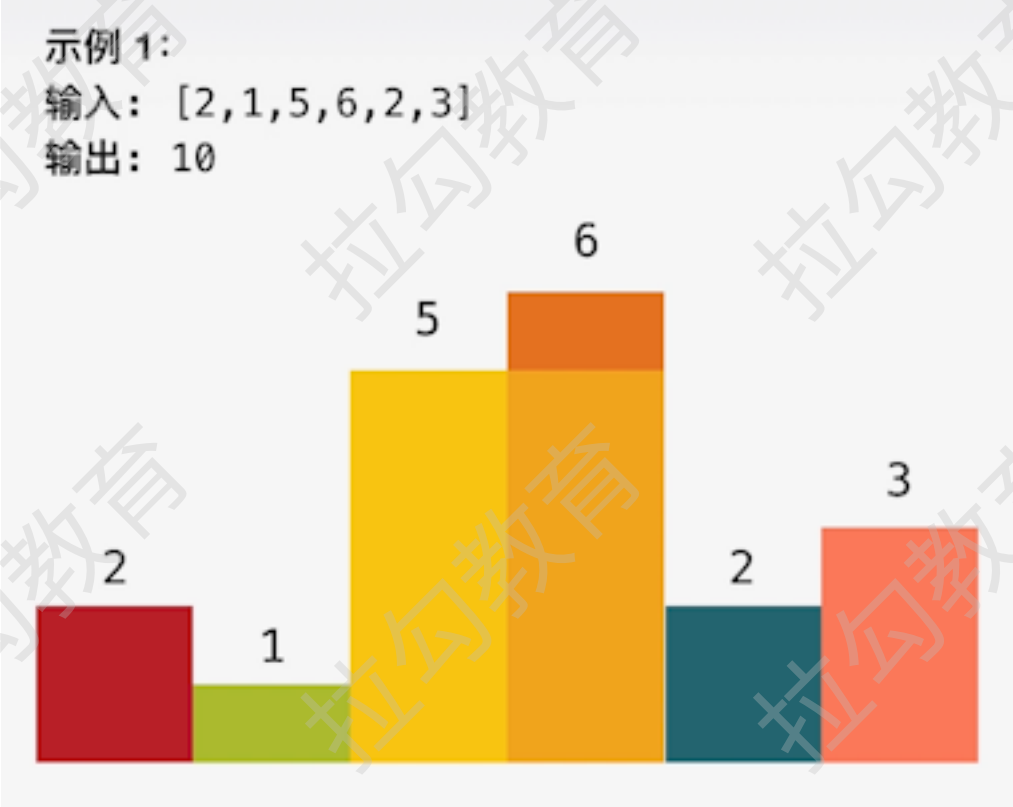
例题分析二

LeetCode 第 84 题 :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。

说明：下图是柱状图的示例，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

输入: [2,1,5,6,2,3]

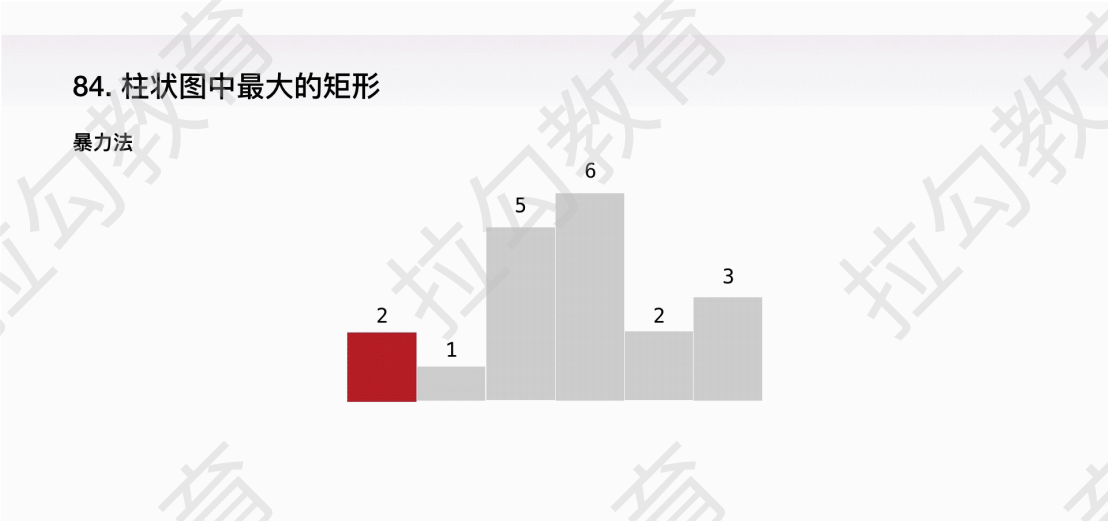
输出: 1

解题思路一：暴力法

从暴力法开始寻找思路。既然要找出最大的面积，就把所有可能的面积都找出来，然后从中比较出最大的那个。如何找出所有的面积呢？

1. 从左到右扫描一遍输入的数组。
2. 遇到每根柱子的时候，以它的高度作为当前矩形的高度。
3. 矩形的宽度从当前柱子出发一直延伸到左边和右边。
4. 一旦遇到了低于当前高度的柱子就停止。
5. 计算面积，统计所有面积里的最大值。

具体的实现步骤如下。



1. 第一根柱子高度是 2，当往右边扩展的时候，发现第二根柱子的高度为 1，要低于当前的高度，于是扩展结束，即以第一根柱子高度作为矩形高度，得到矩形面积是 2。
2. 第二根柱子，它的高度为 1，以它作为高度的矩形面积是 6。
3. 以 5 为高度的矩形面积是 10。
4. 以 6 为高度的矩形面积是 6。
5. 以 2 为高度的矩形面积是 8。
6. 以 3 作为高度的矩形面积为 3。

由此，得到最大的面积是 10。

该算法的时间复杂度是 O(n2)。

解题思路二：解法优化

以两个柱子的情况为例进行分析。

1. 不必急于计算以 2 为高度的矩形面积，把 2 暂时保存起来备用，因为一旦从开始就计算矩形面积的话，就是暴力法。

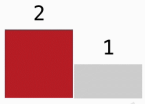
84. 柱状图中最大的矩形

2. 遇到 1 的时候，由于 1 的高度低，造成以 2 为高度的矩形无法延伸到高度为 1 的柱子，即，可以计算高度为 2 的矩形面积。每当遇到一个下降的高度时，就可以开始计算以之前高度作为矩形高度的面积。

84. 柱状图中最大的矩形

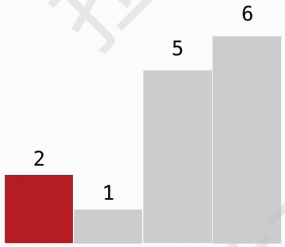
3. 遇到更高的高度时，也不急计算以 1 为高度的矩形面积，因为 5 的下一个是 6，面积还能继续扩大。

84. 柱状图中最大的矩形



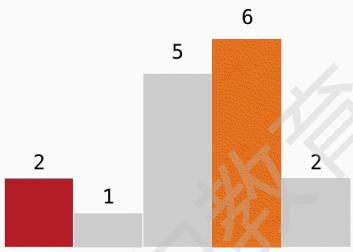
4. 再次遇到 2 时，按照之前的策略，可以计算以 6 为高度的矩形面积。

84. 柱状图中最大的矩形



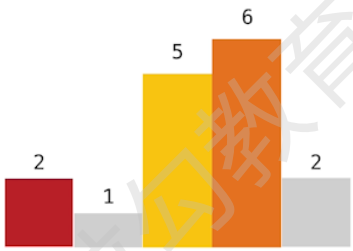
5. 是否要计算以 5 作为高度的矩形面积呢？是的，因为 2 比 5 低，以 5 作为高度的矩形无法包含 2 这个点。该宽度如何计算呢？是不是就是 2 的下标减去 5 的下标就可以呢？

84. 柱状图中最大的矩形



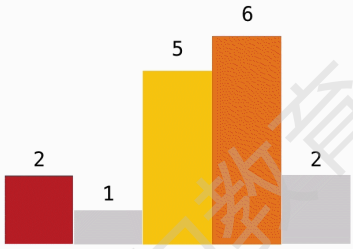
6. 当计算完高度为 6 的矩形面积时，立即知道下一个高度是 5，以及 5 所对应的下标，可以利用一个 stack 来帮助记录。（注意：此处在整个算法里都很重要。）

84. 柱状图中最大的矩形



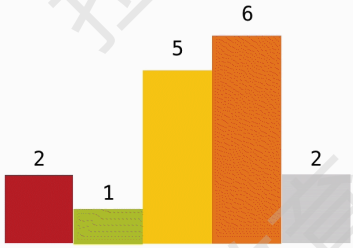
7. 计算完了以 5 作为高度的矩形面积后，还剩下 1，由于 2 比 1 高，表明后面可能还有更高的点，而以 1 为高度的矩形还能扩展。

84. 柱状图中最大的矩形



8. 下一个比 2 还高，于是继续保留它在 stack 里。

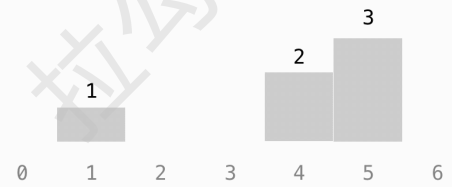
84. 柱状图中最大的矩形



到这里，所有的柱子都遍历完了，如何处理剩下的 3 根柱子呢？

以新的柱子高度为 0，由于 0 低于任何一根柱子的高度，那么对剩下的柱子计算，以它们的高度作为边的矩形的面积。

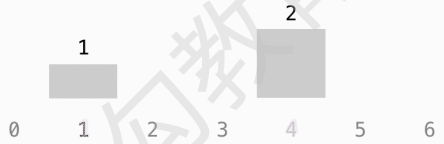
84. 柱状图中最大的矩形



- 指针停留在下标为 6 的地方，堆栈里记录的是三根柱子的下标：5，4，1。
- 跟之前计算其他柱子的情况一样，先将堆栈里的下标弹出，第一个弹出的是 5。
- 然后比它矮的那根柱子的下标一定是堆栈目前顶端的那个，也就是 4。
- 因此以 3 作为高度的矩形的宽度就是：i - 1 - 4 = 6 - 1 - 4 = 1，那么面积就是 3 x 1 = 3。

剩下的 2 根柱子，方法同样，目前 stack 里的值是：4，1。

84. 柱状图中最大的矩形



把下标 4 弹出，得知比这根柱子还要矮的柱子的下标一定是 stack 顶端的值，也就是 1。

那么以高度 2 作为矩形高度的矩形宽度就是：i - 1 - 1 = 6 - 1 - 1 = 4，面积就是 2 x 4 = 8。

最后处理剩下 1 的柱子。

84. 柱状图中最大的矩形

将它弹出，发现此时堆栈为空。那以 1 作为高度的矩形的宽度是多少呢？很简单，就是 i，也就是 6。因为它一定是最矮的那个才会留到最后，那么它的宽度就应该是横跨整个区间。所以求得面积就是 6。

代码实现

1. 一旦我们发现当前的高度要比堆栈顶端所记录的高度要矮，就可以开始对堆栈顶端记录的高度计算面积了。在这里，我们巧妙地处理了当 i 等于 n 时的情况。同时在这一步里，我们判断一下当前的面积是不是最大值。
2. 如果当前的高度比堆栈顶端所记录的高度要高，就压入堆栈。

```
//  将输入数组的长度记为  n，初始化最大面积  max  为  0
int  largestRectangleArea(int[]  heights)  {
    int  n  =  heights.length,  max  =  0;

        //  定义一个堆栈  stack  用来辅助计算
        Stack<Integer>  stack  =  new  Stack<>();

          //  从头开始扫描输入数组
        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);
        }
        // 返回面积最大值
            return max;
}
```

复杂度分析

时间复杂度是 O(n)，因为从头到尾扫描了一遍数组，每个元素都被压入堆栈一次，弹出一次。

空间复杂度是 O(n)，因为用了一个堆栈来保存各个元素的下标，最坏的情况就是各个高度按照从矮到高的顺序排列，需要将它们都压入堆栈。

例题分析三

LeetCode 第 28 题：实现 strStr() 函数。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。

示例 1

输入: haystack = "hello", needle = "ll"

输出: 2

解释：“ll”出现在 haystack 第 2 个位置。

示例 2

输入: haystack = "aaaaa", needle = "bba"

输出: -1

解释：“bba”并不出现在 "aaaaa"里

解题思路一：暴力法

实现 :在一个字符串中找出某个字符串出现的位置 ,用暴力法来做是非常简单的，从头遍历一遍 haystack 字符串，每遍历到一个位置，就扫描一下，看看是不是等于 needle 字符串。举例说明如下。

输入：

haystack = "iloveleetcode"

needle = "leetcode"

28. 实现 strStr()

暴力法

不断移动 needle，来对比是否在 haystack 中，一旦找到就返回它的位置。

注意 :当 needle 是空字符串时 ,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 是空字符串时应当返回 0 。这与 C 语言的 strstr() 以及 Java 的 indexOf() 定义相符。

代码实现

暴力法的代码实现比较简单，如下。

```
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)) bre
ak;
        }
    }
}
```

复杂度分析

假设 haystack 的字符串长度为 m，needle 字符串的长度为 n，那么暴力法的时间复杂度是 O(m×n)。

解题思路二：KMP

KMP(Knuth-Morris-Pratt)是由三人联合发表的一个算法，目的就是为了在一个字符串 haystack 中找出另外一个字符串 needle 出现的所有位置。它的核心思想是避免暴力法当中出现的不必要的比较。

用维基百科中的例题说明。

举例：

haystack = "ABC ABCDAB ABCDABCDABDE"

needle = "ABCDABD"

28. 实现 strStr()

KMP

```
haystack = “ABC ABCDAB ABCDABCDABDE”
needle   =     “ABCDABD”
```

解法 1：暴力法，当比较到上图所示位置的时候，发现 D 和空格不一样。接下来，needle 往前挪动一小步，然后继续和 haystack 比较。

解法 2：KMP，直接让 needle 挪动到如上图所示的位置。

此处有两个常见的问题：

1. 为什么 KMP 无需慢慢移动比较，可以跳跃式比较呢？不会错过一些可能性吗？
2. 如何能知道 needle 跳跃的位置呢？

LPS

为了说明这两个问题，必须先讲解 KMP 里的一个重要数据结构——最长的公共前缀和后缀，英文是 Longest Prefix and Suffix，简称 LPS。

LPS 其实是一个数组，记录了字符串从头开始到某个位置结束的一段字符串当中，公共前缀和后缀的最大长度。所谓公共前缀和后缀，就是说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。

以上题中 needle 字符串，它的 LPS 数组就是：{0, 0, 0, 0, 1, 2, 0}。

needle = "ABCDABD"

LPS = {0000120}

- 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：虽然"A"的前缀和后缀都等于 A，但前缀和后缀不能是同一段字符串，因此，” A” 的 LPS 为 0。

- LPS[1] = 0，表示字符串” AB” 的最长公共前缀和后缀长度为 0。

  因为它只有一个前缀 A 和后缀 B，并且它们不相等，因此 LPS 为 0。

- LPS[4] = 1，表示字符串 ABCDA 的最长公共前缀和后缀的长度为 1。

  该字符串有很多前缀和后缀，前缀有：A，AB，ABC，ABCD，后缀有：BCDA，CDA，DA，A，其中两个相同并且长度最长的就是 A ，所以 LPS 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。

  该字符串有很多前缀和后缀，前缀有：A，AB，ABC，ABCD，ABCDA，后缀有：BCDAB，CDAB，DAB，AB，B，其中两个相同并且长度最长的就是 AB，所以 LPS 为 2。

LPS 实现跳跃比较

那么，LPS 数组如何实现跳跃比较 haystack 和 needle 字符串呢？

28. 实现 strStr()

KMP

```
haystack = “ABC ABCDAB ABCDABCDABDE”
needle   =     “ABCDABD”
```

LPS = 2

1. haystack 里面的空格和 needle 里的 D 不相等时，在 needle 里，D 前面的字符串 ABCDAB 与 haystack 中对应的字符串是相等的。
2. ABCDAB 的 LPS 为 2，即，对于 ABCDAB ，它最后两个字符一定与它最前面两个字符相等。
3. 若把最前面的两个字符挪到最后两个字符的位置，可以保证 AB 位置绝对能和 haystack 配对。

那么，为什么不需要去比较前面的位置？

例如：

KMP

```
haystack = “ABC ABCDAB ABCDABCDABDE”
needle   =      “ABCDABD”
```

例如：

KMP

```
haystack = “ABC ABCDAB ABCDABCDABDE”
needle   =       “ABCDABD”
```

因为没有必要。下面通过反证法来证明。将下图所示情况用抽象成为方块图形来表示。

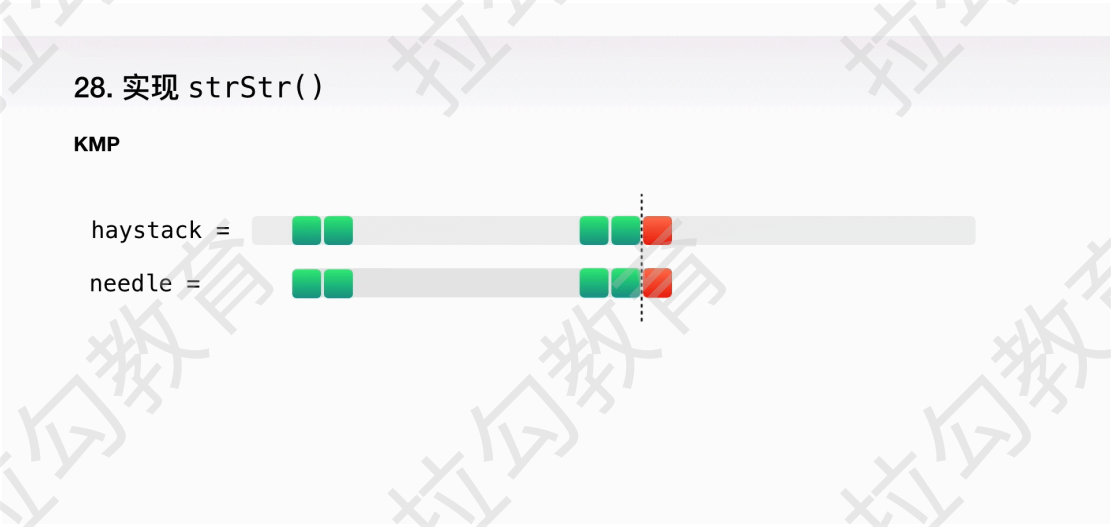
28. 实现 strStr()

KMP

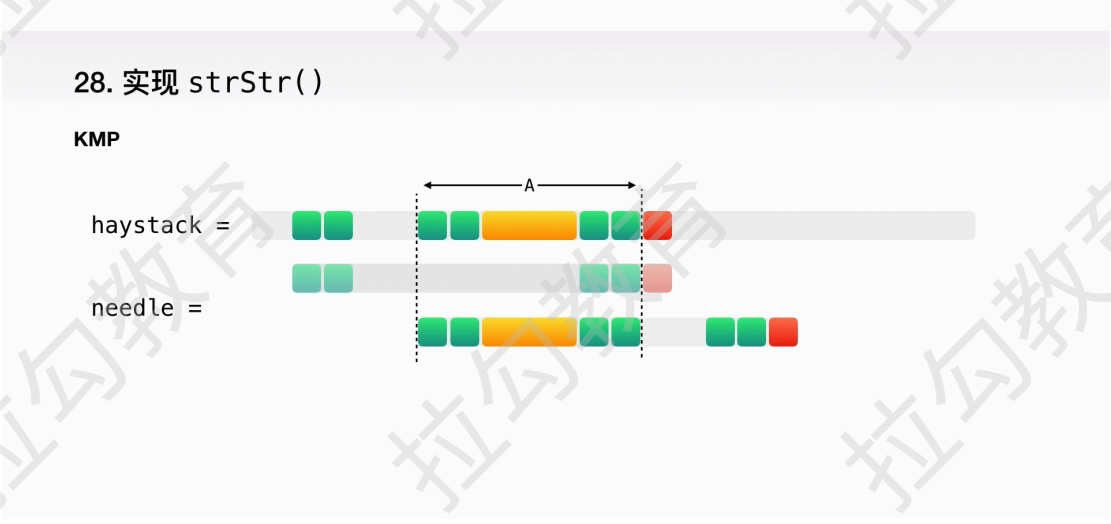
```
haystack = “ABC ABCDAB ABCDABCDABDE”
needle   =     “ABCDABD”
```

其中红色的方块表示不相同的字符，分别对应 haystack 中的空格以及 needle 当中的 D 字符；而绿色的方块表示相同的最大前缀和后缀，对应字符串里的 AB。

现在，假设向右挪动了，使得 needle 能与 haystack 完美地匹配，如下所示，可以标出 haystack 与 needle 完美匹配时的关系。即，在 haystack 和 needle 里，有一段区间 A，它们是相同的。



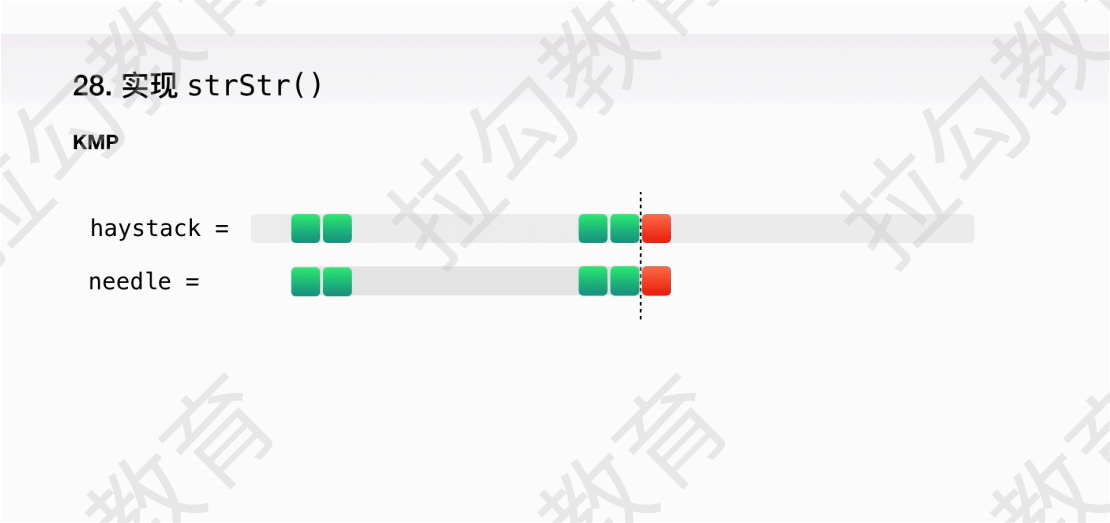
那么，needle 里，红色方块前的一段区间其实和 needle 开头的一段区间是相同的，它们都是 A，如下所示。



即，红色方块前的 needle 字符串，A 是共同的前缀和后缀。而它比两个绿色的方块要长得多，这与之前定义的“两个绿色方块是最长的公共前缀和后缀”相互矛盾。

因此，当知道两个绿色的方块就是最大的公共前缀和后缀时，可以放心地进行跳跃操作，而不必担心会错过完全匹配的情况发生。完美匹配不可能在跳跃的区间内发生。

那么，具体在算法上如何进行跳跃操作呢？



1. j 指针指向红色方块的位置，needle 的字符与 haystack 的字符不一样。
2. LPS[j - 1] = 2，即 j 指针前一个字符作为结尾时的最长公共前缀和后缀长度是 2，因此，只需要将 j 移动到 2 的位置即可，也就是 j = LPS[j - 1]。

以上就是 KMP 算法的核心思想，下面来看代码如何实现。

代码实现

假如已经求出了 LPS 数组，如何实现上述跳跃策略？代码实现如下。

```
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;

    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;
}
```

代码解释：

1. 分别用变量 m 和 n 记录 haystack 字符串和 needle 字符串的长度。
2. 若 n=0，返回 0，符合题目要求。
3. 求出 needle 的 LPS，即最长的公共前缀和后缀数组。

4. 分别定义两个指针 i 和 j，i 扫描 haystack，j 扫描 needle。
5. 进入循环体，直到 i 扫描完整个 haystack，若扫描完还没有发现 needle 在里面，就跳出循环。
6. 在循环体里面，当发现 i 指针指向的字符与 j 指针指向的字符相等的时候，两个指针一起向前走一步，i++，j++。
7. 若 j 已经扫描完了 needle 字符串，说明在 haystack 中找到了 needle，立即返回它在 haystack 中的起始位置。
8. 若 i 指针指向的字符和 j 指针指向的字符不相同，进行跳跃操作，j = LPS[j - 1]，此处必须要判断 j 是否大于 0。
9. j=0，表明此时 needle 的第一个字符就已经和 haystack 的字符不同，则对比 haystack 的下一个字符，所以 i++。
10. 若没有在 haystack 中找到 needle，返回 -1。

复杂度分析

KMP 算法需要 O(n) 的时间计算 LPS 数组，还需要 O(m) 的时间扫描一遍 haystack 字符串，整体的时间复杂度为 O(m + n)。这比暴力法快了很多。

例题三扩展

如何求出 needle 字符串的最长公共前缀和后缀数组？

解题思路一：暴力法

解法：检查字符串的每个位置。

举例：若字符串长度为 m，先尝试比较长度为 m−1 的前缀的后缀，如果两者一样，就记录下来；如果不一样，就尝试长度为 m−2 的前缀和后缀。以此类推。

复杂度：O(n2)。

解题思路二

解法：对于给定的字符串 needle，用一个 i 指针从头到尾扫描一遍字符串，并且用一个叫 len 的变量来记录当前的最长公共前缀和后缀的长度。举例说明如下。

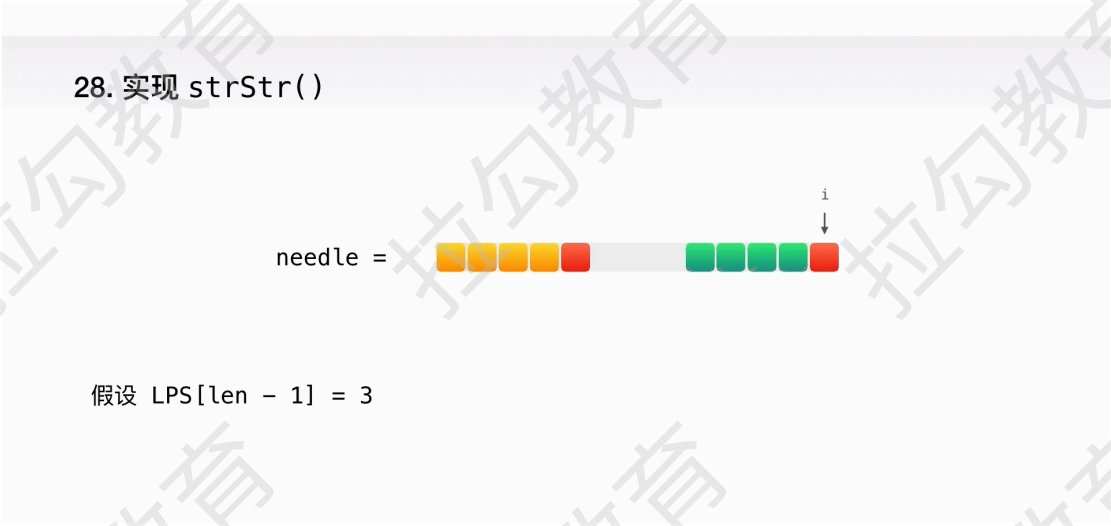
28. 实现 strStr()

needle = 

当 i 扫描到这个位置的时候，len=4，表明在 i 之前的字符串里，最长的前缀和后缀长度是 4，也就是那 4 个绿色的方块。

现在 needle[i] 不等于 needle[4]，怎么计算 LPS[i] 呢？

既然无法构成长度为 5 的最长前缀和后缀，那便尝试构成长度为 4，3，或者 2 的前缀和后缀，但做法并非像暴力法一样逐个尝试比较，而是通过 LPS[len - 1] 得知下一个最长的前缀和后缀的长度是什么。举例说明如下。



- LPS[len - 1] 记录的是橘色字符串的最长的前缀和后缀，假如 LPS[len - 1]=3，那么前面 3 个字符和后面的 3 个字符相等
- 绿色的部分其实和橘色的部分相同。
- LPS[len - 1] 记录的其实是 i 指针之前的字符串里的第二长的公共前缀和后缀（最关键点）。
- 更新 len = LPS[len - 1]，继续比较 needle[i] 和 needle[len]。

代码实现

```
int[] getLPS(String str) {
    // 初始化一个 lps 数组用来保存最终的结果
        int[] lps = new int[str.length()];

    // lps 的第一个值一定是 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。并且，初始化当前最长
```

```
的 lps 长度为 0，用 len 变量记录下
        int i = 1, len = 0;

        // 指针 i 遍历整个输入字符串
        while (i < str.length()) {
        // 若 i 指针能延续前缀和后缀，则更新 lps 值为 len+1
        if (str.charAt(i) == str.charAt(len)) {
            lps[i++] = ++len;
        // 否则，判断 len 是否大于 0，尝试第二长的前缀和后缀，是
否能继续延续下去/
        } else if (len > 0) {
            len = lps[len - 1];
        // 所有的前缀和后缀都不符合，则当前的 lps 为 0，i++
        } else {
            i++;
        }
    }

  return lps;

}
```

复杂度分析

时间复杂度为 O(n) ，这是一种比较高效的做法。

举例说明

下面通过举例来加深印象。

例题：needle 是 ADCADBADCADC。

28. 实现 strStr()

```
needle = ADCADB
LPS    = 000000
```

1. 一开始，初始化 LPS 数组全部为 0。

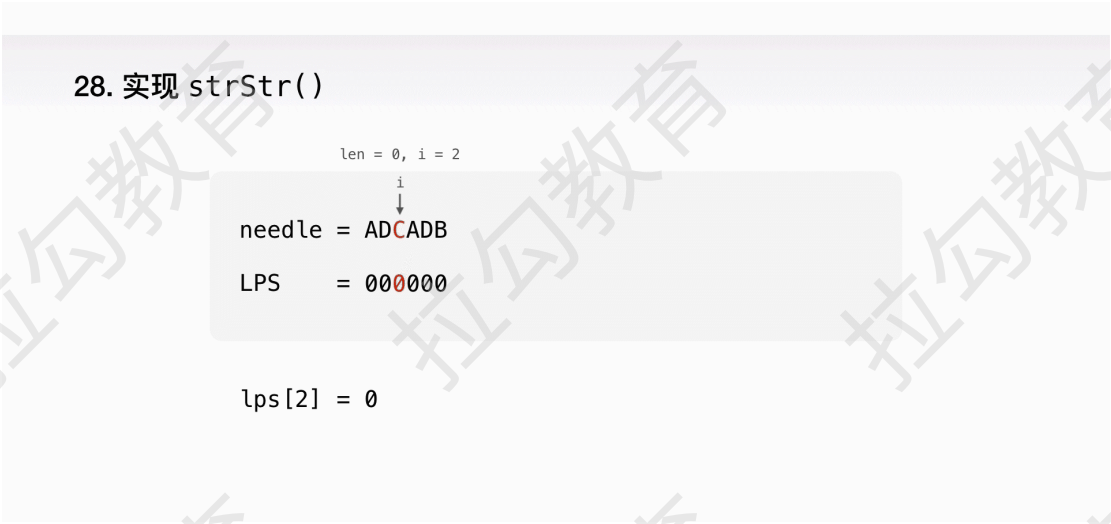
规定前缀和后缀不能是同一个字符串，所以从第二个字符开始扫描，此时 len = 0，i = 1。AD 字符串的最长公共前缀和后缀为 0，因为 A 不等于 D，所以 LPS[1] = 0。

28. 实现 strStr()

```
          len = 0, i = 1
                i
                ↓
needle = ADCADB
LPS    = 000000

lps[1] = 0
```

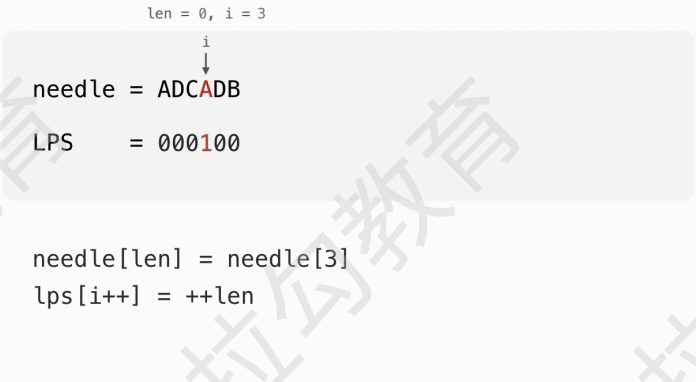
2. 移动到 C。同样，对于 ADC ，最长的公共前缀和后缀也是 0，所以 LPS[2] = 0，此时，len 变量一直是 0。



3. 移动到 A，此时 i=3。

对于字符串 ADCA，因为 needle[len] = needle[3]，所以执行代码 lps[i++] = ++len，也就是把 len+1 赋给 lps[i]，然后 i + 1，len + 1，表明对于字符串 ADCA，最长的公共前缀和后缀的长度为 1。

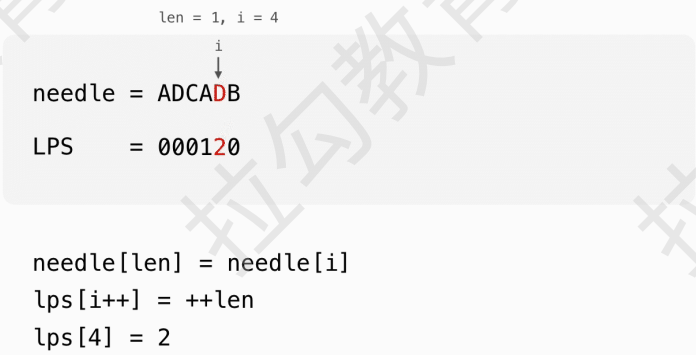
28. 实现 strStr()



4. 接下来到 D，此时 i = 4，len = 1。

同样，由于 needle[len] 等于 needle[i]，都是字符 D，所以再次执行代码 lps[i++] = ++len，这样一来，lps[4] 就等于 2，表明对于字符串 ADCAD，最长的公共前缀和后缀长度是 2。

28. 实现 strStr()



5. 接下来是 B，此时 i = 5，len = 2。

needle[len] ='C'，而 needle[i] ='B'，两者不相等，同时，len 大于 0，将 len 修改为 lps[len - 1] ,取出字符串 AD 的最长公共前缀和后缀的长度 ,也就是 0。当循环再次进行，needle[len] 仍不等于 neele[i]，因此对于 ADCADB ，最长的公共前缀后缀长度为 0。