

算法讨论班 28 期—李耀宗

2016.4.22

28. Implement strStr()

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

这道题考察的是 KMP 算法。在原字符串中寻找第一个匹配子字符串的位置并且返回，如果没有找到，返回-1.

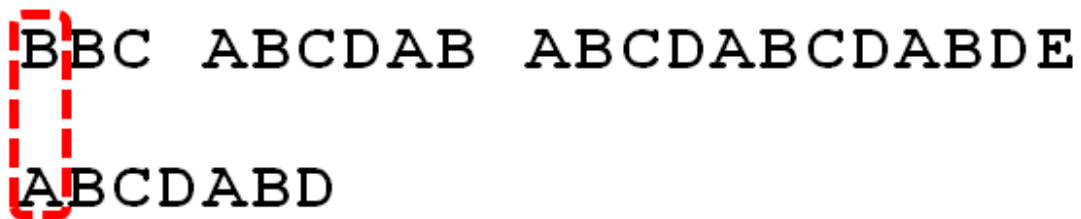
思路：

如果不用 kmp 算法，这道题显然存在 $O(m \times n)$ 的算法，即从原字符串的每个位置依次开始匹配，如果匹配成功就返回，如果不成功，就从下一个位置继续匹配。如果最后没有找到就返回-1.

但是在以上匹配过程中，我们并没有用到子串本身包含的信息，如果用到子串自身的信息，这道题是存在 $O(m+n)$ 算法的，即 KMP 算法。

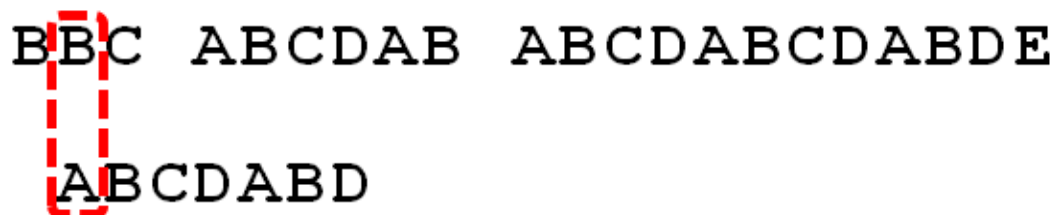
以下是利用 kmp 算法求解过程示意图：

1、待匹配的两个字符串



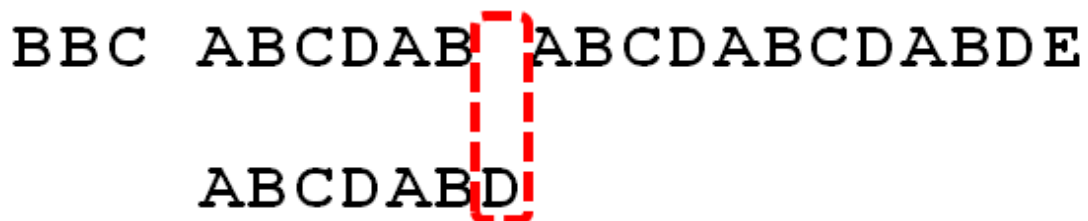
BBC ABCDAB ABCDABCDABDE
ABCDABD

2、如果位置不匹配向后移动。



BBC ABCDAB ABCDABCDABDE
ABCDABD

3、这时我们已经匹配到了子串的最后一个字符，发现不匹配，我们并没有回到子串第一个字符和原字符串的第六个字符 B 开始比较。而是从下图所示位置开始比较。



BBC ABCDAB ABCDABCDABDE
ABCDABD

4、这时我们从子串的第三个字符开始比较，因为 D（最后一个）前面的两个字符和 C 前面的两个字符是完全相同的，都是 AB。我们已经能够匹配到 D，说明 D 前面的字符都已经匹配成功，这时 C 前面的字符和 D 前面的字符相同的话，就一定也能匹配成功，这就是利用了子串的信息进行移位。

BBC ABCDAB ABCDABCDABDE
ABCDABD

5、现在问题的关键变为，如何求取位置不匹配后，下一个比较的位置呢？我们把这个位置存在数组里，叫做 next 数组。

问题核心部分，next 数组求解！

Next 的数组的含义比较好理解，但是求解过程比较复杂：

定义两个指针 i, j，其中 i 为指向子串求解位置的指针，j 可以理解为匹配前缀尾指针。

初始化条件：

求解 next[0]，当第一个位置就不匹配的时候，如图：

BBC ABCDAB ABCDABCDABDE
ABCDABD

next 数组存储的是当 i 不匹配时，i 向前跳转的位置。这时 i 已经在最前面的位置了，显然无法向前跳转。

这时应该设置 next 数组值为 -1。即 next 数组值为 -1 表示子串指针 i 已经无法向前跳转的合适的位置，如果要继续匹配，需要移动原字符串指针从新开始匹配。

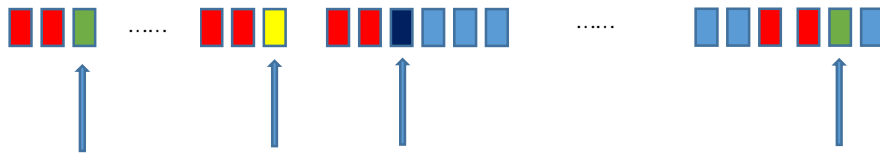
那么 j 的值如何设置呢，此时 i 指向第一个位置，i 前根本没有子串，我们把 j 的值也设置为 -1；

下面开始循环求解：

当 i 为 1，即指向第二个字符时，j 为 0，此时如果 $\text{char}[i] == \text{char}[j]$ ，即头两个字符相等时， $\text{next}[1] = -1$ ，如果 $\text{char}[i] != \text{char}[j]$ ， $\text{next}[i] = 0$ ；因为当第一个字和第二个字符相等，第二个字符不匹配，第一个字符肯定也不匹配，所以必须移动原字符串指针从新匹配。否则 i 可以跳转到第一个位置开始匹配。

循环过程中，新一次计算开始时，如果 i 指向 m，j 指向 n ($n \neq -1$)。此时表示从 $\text{char}[0:n] == \text{char}[m-n:m]$ ，此时执行 $i++$ ， $j++$ ，如果 $\text{char}[i] == \text{char}[j]$ ，则 $\text{next}[i] = \text{next}[j]$ ，否则 $\text{next}[i] = j$ 。这个原因与前面一样。因为相等的话， $\text{char}[i]$ 不匹配， $\text{char}[j]$ 肯定也不匹配， $\text{char}[j]$ 不匹配跳转到 $\text{next}[j]$ ，所以 $\text{char}[i]$ 肯定也跳转到 $\text{next}[j]$ ，如果不相等，跳转到 j 继续匹配即可。

还有一个问题，当 $\text{char}[i] \neq \text{char}[j]$ 的时候，如何移动 j 呢， j 只需要一直跳转到 $\text{next}[j]$ ，直到 $j = -1$ ，或者 $\text{char}[j] == \text{char}[i]$ 即可。为什么，如下图所示：



因为在 j 在移动过程中，要始终保持前缀一致性，即图中红色部分相匹配。

具体实现代码：

```
1 public class Solution {
2     public static int strStr(String haystack, String needle) {
3         char[] s = haystack.toCharArray();
4         char[] t = needle.toCharArray();
5         int[] next = next(t);
6         int i = 0, j = 0;
7         while (i <= s.length - 1 && j <= t.length - 1) {
8             if (j == -1 || s[i] == t[j]) {
9                 i++;
10                j++;
11            } else {
12                j = next[j];
13            }
14        }
15        if (j < t.length) {
16            return -1;
17        } else {
18            return i - t.length;
19        }
20    }
21    public static int[] next(char[] t) {
22        int[] next = new int[t.length];
23        if (t.length > 0)
24            next[0] = -1;
25        int i = 0;
26        int j = -1;
27        while (i < t.length - 1) {
28            if (j == -1 || t[i] == t[j]) {
29                i++;
30                j++;
31            } else {
32                next[i] = j;
33            }
34        }
35        j = next[j];
36    }
37    }
38    return next;
39 }
40 }
41 }
```

30. Substring with Concatenation of All Words

You are given a string, **s**, and a list of words, **words**, that are all of the same length. Find all starting indices of substring(s) in **s** that is a concatenation of each word in **words** exactly once and without any intervening characters.

For example, given:

s: "barfoothefoobarman"

words: ["foo", "bar"]

You should return the indices: [0,9]

这道题的题意是，在原字符串中寻找开始下标，使得从开始下标向后 **n** 个字符形成的单词集合与字典中的单词集合完全对应，**n** 为字典中单词的总长度，完全对应单词种类和每种单词数量完全匹配。

思路：

首先这道题也很容易想到穷举法，即从 **s** 的开始位向后依次计算。但是在计算过程中，会有很多浪费，所以我们可以用经典的双指针算法。

左指针指向匹配开始位置，右指针指向结束位置。先从开始位置移动右指针，当左右指针包含全部单词，每个单词数量大于或者等于集合中每个单词数量时停止右指针。然后移动左指针，左指针移动时，如果指针当前对应单词在左右指针之间的数量刚好等于集合中对应单词数量，进行一次判定，判定是完全匹配则将左指针保存，继续移动右指针。

在算法执行过程中的一个重要问题是，如何保存单词集合和每个单词的数量。当然可以利用 Map 进行保存，key 为单词，value 为数量。但是由于 value 经常变动，所以建议 value 设置为数组索引，在数组中保存单词数量。

以下是算法实现过程：

```

1 public class Solution {
2     public List<Integer> findSubstring(String s, String[] words) {
3         int N = s.length();
4         List<Integer> indexes = new ArrayList<Integer>(s.length());
5         if (words.length == 0) {
6             return indexes;
7         }
8         int M = words[0].length();
9         if (N < M * words.length) {
10             return indexes;
11         }
12         int last = N - M + 1;
13         //map each string in words array to some index and compute target counters
14         Map<String, Integer> mapping = new HashMap<String, Integer>(words.length);
15         int [][] table = new int[2][words.length];
16         int failures = 0, index = 0;
17         for (int i = 0; i < words.length; ++i) {
18             Integer mapped = mapping.get(words[i]);
19             if (mapped == null) {
20                 ++failures;
21                 mapping.put(words[i], index);
22                 mapped = index++;
23             }
24             ++table[0][mapped];
25         }
26         //find all occurrences at string S and map them to their current integer,
27         //-1 means no such string is in words array
28         int [] smapping = new int[last];
29         for (int i = 0; i < last; ++i) {
30             String section = s.substring(i, i + M);
31             Integer mapped = mapping.get(section);
32             if (mapped == null) {
33                 smapping[i] = -1;
34             } else {
35                 smapping[i] = mapped;
36             }
37         }
38         //fix the number of linear scans
39         for (int i = 0; i < M; ++i) {
40             //reset scan variables
41             int currentFailures = failures; //number of current mismatches
42             int left = i, right = i;
43             Arrays.fill(table[1], 0);
44             //here, simple solve the minimum-window-substring problem
45             while (right < last) {
46                 while (currentFailures > 0 && right < last) {
47                     int target = smapping[right];
48                     if (target != -1 && ++table[1][target] == table[0][target]) {
49                         --currentFailures;
50                     }
51                     right += M;
52                 }
53                 while (currentFailures == 0 && left < right) {
54                     int target = smapping[left];
55                     if (target != -1 && --table[1][target] == table[0][target] - 1) {
56                         int length = right - left;
57                         //instead of checking every window, we know exactly the length we want
58                         if ((length / M) == words.length) {
59                             indexes.add(left);
60                         }
61                         ++currentFailures;
62                     }
63                     left += M;
64                 }
65             }
66         }
67         return indexes;
68     }
69 }
70

```

42. Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,

Given `[0,1,0,2,1,0,1,3,2,1,2,1]`, return 6.



这道题是求解存水量，即给定数组表示每个位置的高度，求数组间的最大存水量。

解题思路：

这道题可以这样想，在任何位置的存水量，都等于其左右两边较小的 **bar** 与其差值。例如，某个位置高度为 2，左右两边的 **bar** 分别为 3,4，那么其存水量一定为 1。所以只要找到每个位置两边的 **bar**，就可以求解。

首先，数组的第一位和最后一位肯定不能存水，我们可以将其设置为初始的左右 **bar** 高度。然后依次求解。

选择下一位计算时，要看左 **bar** 和右 **bar** 的相对高度，如果右 **bar** 高于左 **bar**，就计算左 **bar** 右边那一位。就是第二位的存水量，加入第二位高于第一位，即高于它的左 **bar**，那么显然不可以存水，此时更新左 **bar**，继续迭代，如果低于第一位，不更新左 **bar**，计算存水量并累加即可。

算法最终复杂度为 $O(n)$ ，算法实现如下：

```

1 public int trap(int[] height) {
2     int lo = 0, hi = height.length - 1;
3     int total = 0, level = 0;
4     while (lo < hi) {
5         if (height[lo] > height[hi]) {
6             if (height[hi] > level)
7                 level = height[hi];
8             else
9                 total += level - height[hi];
10            hi--;
11        } else {
12            if (height[lo] > level)
13                level = height[lo];
14            else
15                total += level - height[lo];
16            lo++;
17        }
18    }
19    return total;
20 }
21

```

81. Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array":

What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

这道题是在一个旋转数组中找到某个特定的值，如果找到就返回 **true**，否则返回 **false**。

这道题是 33. *Search in Rotated Sorted Array* 的升级版，原题中不允许数组中数字重复，而这道题中可以。

解题思路：

在原题中，如果数字都是不重复，我们可以直接利用二分法进行求解。将数组分成两半，这两半分别是原问题的一个独立子问题，而且其中一半一定是旋转排列数组，另一个正常的排列数组。

通过找出正常排列数组，就能立刻判断出寻找的数字是否在其范围内。如果在，就递归寻找正常数组，否则递归寻找旋转数组。

这道题因为加入的重复元素，所以在用二分法的时候，取到的中间数字可能和两边是相同的。这时候如果和左边相同就移动左指针，如果和右边相同，同样移动右指针。直到两边数字都和中间数字不同，就能继续二分求解，方法和原问题一样了。解法最坏时间复杂度为 $O(n)$ 。

代码实现：

```
1 public boolean search(int[] nums, int target) {
2     int start = 0;
3     int end = nums.length - 1;
4
5     while (start <= end) {
6         int mid = start + (end - start) / 2;
7         // System.out.format("start=%d,mid=%d,end=%d\n",start,mid,end);
8         if (nums[mid] == target) return true;
9
10        // need to handle: 1,3,1,1,1
11        while (nums[start] == nums[mid] && start != mid) {
12            start ++;
13        }
14        while (nums[mid] == nums[end] && mid != end) {
15            end --;
16        }
17
18        // the following is the same as problem I
19        if (nums[start] <= nums[mid]) {
20            if (nums[start] <= target && target < nums[mid]) {
21                end = mid - 1;
22            } else {
23                start = mid + 1;
24            }
25        } else {
26            if (nums[mid] < target && target <= nums[end]) {
27                start = mid + 1;
28            } else {
29                end = mid - 1;
30            }
31        }
32    }
33
34    return false;
35 }
```


224. Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus `-` sign, **non-negative** integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

```
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23
```

这道题是给定字符串，求解字符串表示的算式最终运算结果。只包含加减运算和括号运算。题目保证所给定的运算都是合法的。

解题思路：

这道题是堆栈结构运用典型例题。假如没有括号，我们只需要从左向右依次运算即可。每次取到符号和数字，然后与结果相累加。

但是因为有了括号，每次遇到左括号，我们就要进行压栈，将运算结果和符号依次压入栈中，每次遇到右括号，就进行出栈，将当前结果与出栈结果累加即可。

以下是具体的算法实现。

```
1 public class Solution {
2     public static int calculate(String s) {
3         int len = s.length(), sign = 1, result = 0;
4         Stack<Integer> stack = new Stack<Integer>();
5         for (int i = 0; i < len; i++) {
6             if (Character.isDigit(s.charAt(i))) {
7                 int sum = s.charAt(i) - '0';
8                 while (i + 1 < len && Character.isDigit(s.charAt(i + 1))) {
9                     sum = sum * 10 + s.charAt(i + 1) - '0';
10                    i++;
11                }
12                result += sum * sign;
13            } else if (s.charAt(i) == '+')
14                sign = 1;
15            else if (s.charAt(i) == '-')
16                sign = -1;
17            else if (s.charAt(i) == '(') {
18                stack.push(result);
19                stack.push(sign);
20                result = 0;
21                sign = 1;
22            } else if (s.charAt(i) == ')') {
23                result = result * stack.pop() + stack.pop();
24            }
25        }
26        return result;
27    }
28 }
29 }
```