String Match

28. Implement strStr()

KMP Match

解法一：KMP字符串匹配

时间复杂度：O(m + n)； 空间复杂度：O(1)

参考1：知乎回答

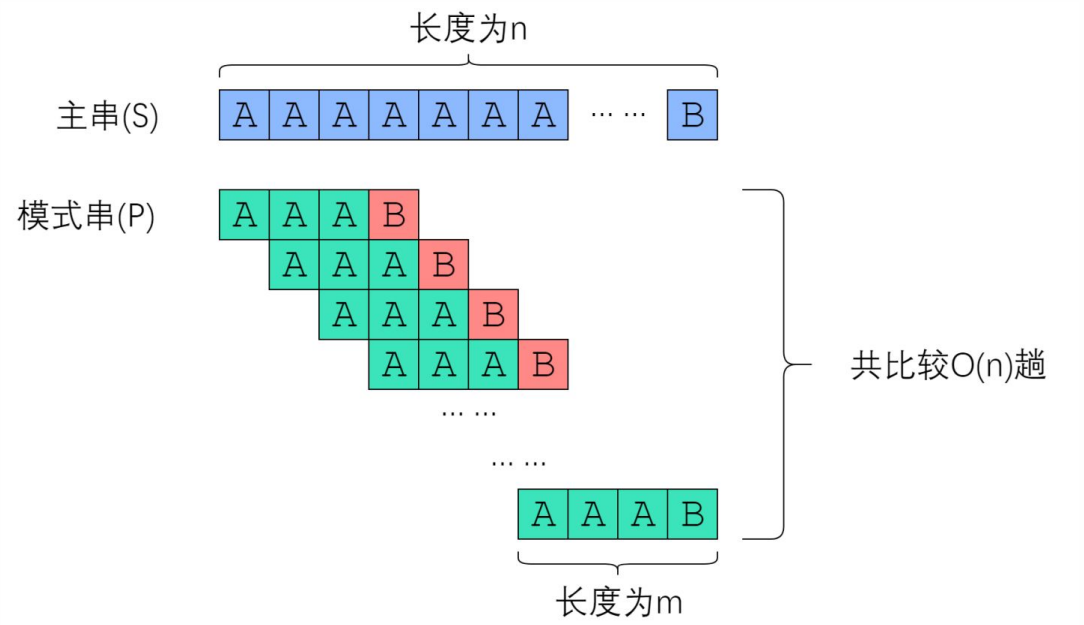
参考2：Youtube视频

参考3：评论区题解

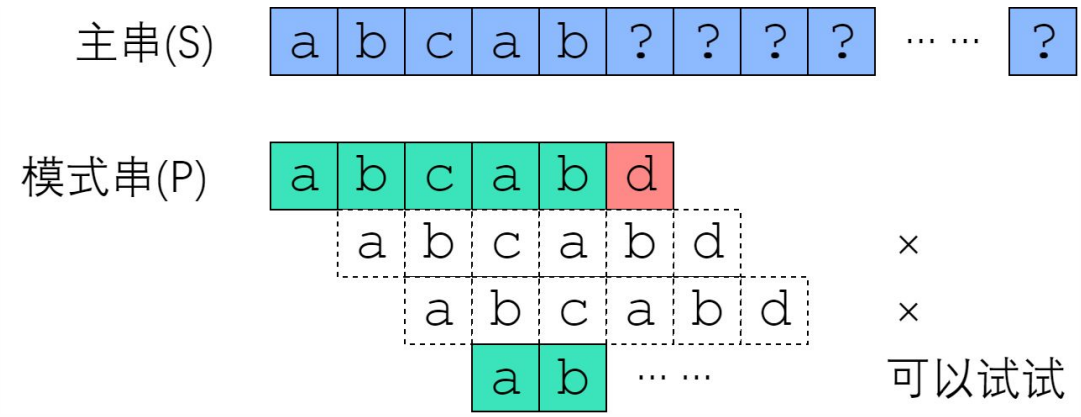
参考4：《大话数据结构》

引入

暴力搜索所需要的时间复杂度 O(mn) 较高，且有很多不必要的判断，而KMP算法的核心在于去除这些不必要的判断。

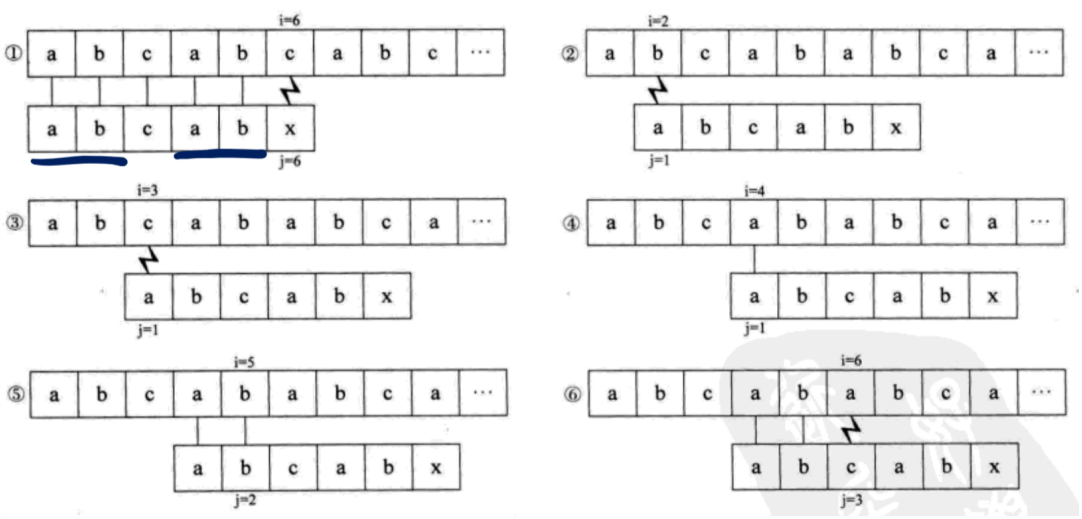


当模式串中存在一些重复的Pattern时，可以跳过一些已经确定不可能匹配的情况，且跳过一些已经知道能匹配的情况，从而减少一些比较。



如下图所示，第二次和第三次比较是不必要的，因为我们已经知道之前模式串0-5位已经和主串0-5位全部匹配，但模式串的第1-2位都和第0位不同，因此**不需要再将第0位和主串的第1位和第2位比较。**

另外第四次和第五次比较是不需要的，因为已经知道模式串第3-4位和模式串第0-1位一样，而模式串第3-4位和主串第3-4位已经完成匹配，所以**模式串第0-1位肯定和主串第3-4位一样。**因此进过第一步的比较可以直接跳到第四步的比较。

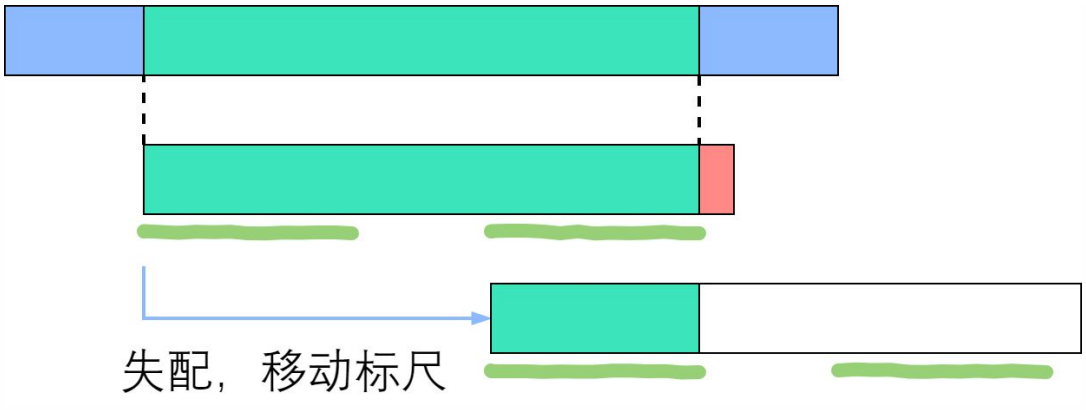


我们需要制作一个Next 数组来帮助我们知道，在某个位置失配时，接下来要将标尺（模式串）移动到哪里去进行下一次匹配检测。

Next数组：最长公共前后缀

> Next 数组的定义是：next[i] 表示 P[0] ~ P[i] 这一个子串，使得 **前k个字符恰等于后k个字符** 的最大的k。

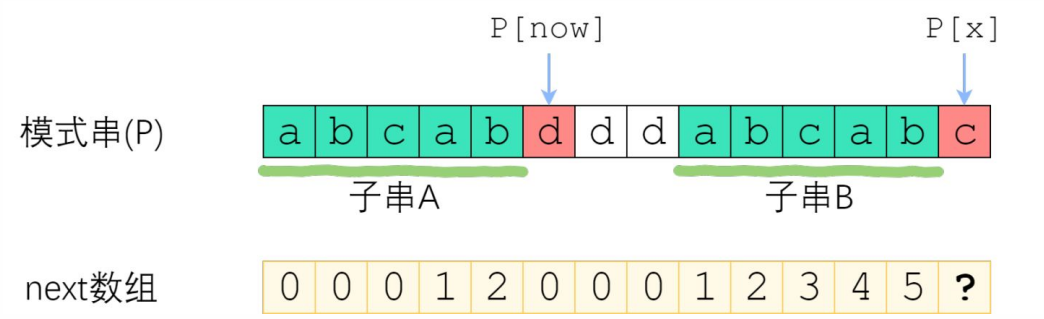
说得更简单一点，我们需要找到 P[0] ~ P[i] 这一个子串的最长公共前后缀。当我们找到之后，下一次移动标尺可以把前缀和之前后缀摆放的位置放在一起，因为已经可以确定能成功匹配了，然后再继续检测下一位。



求解next数组可以使用递推的思路。如果next[0], next[1], ... next[i - 1]均已知，将要求next[i]。

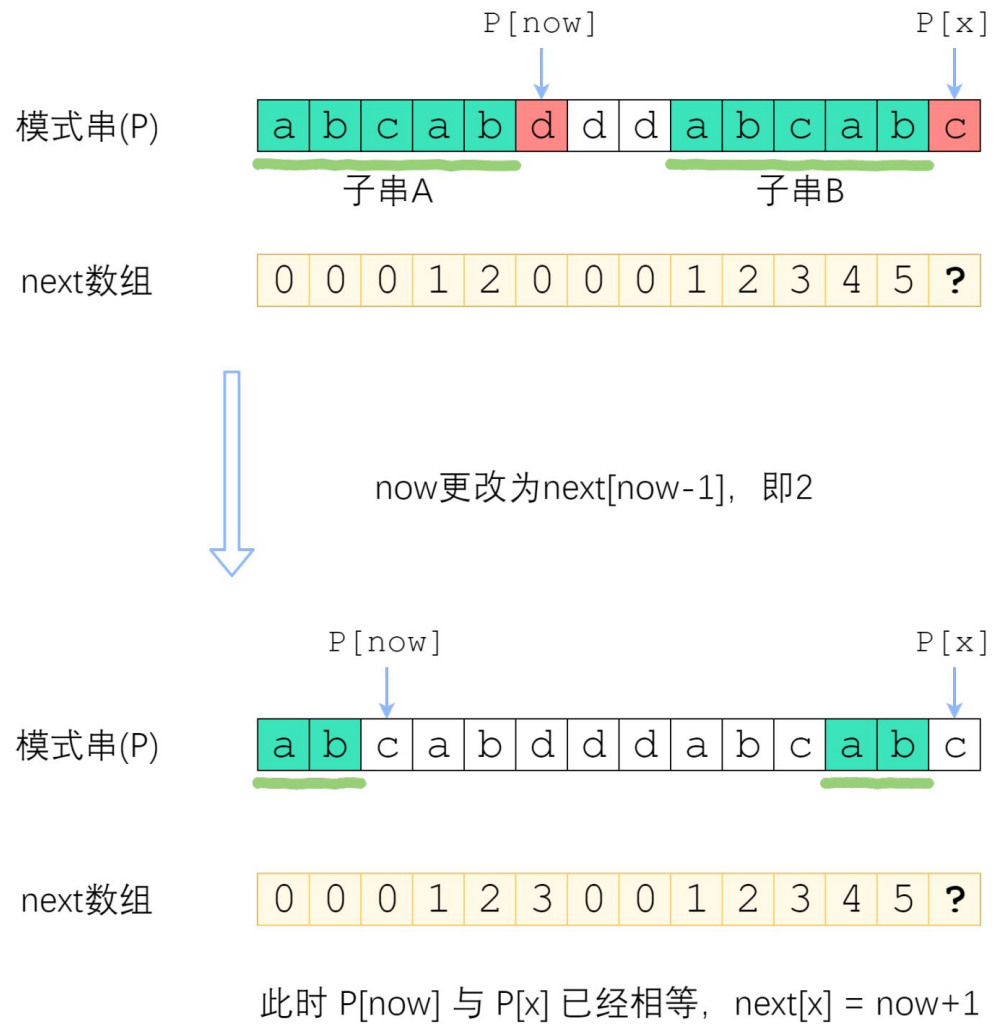
若next[i - 1] = k，则说明P[0] ~ P[k] 和 P[i - 1 - k] ~ P[i - 1] 是对于 P[0] ~ P[i - 1] 这个子串来说最长的公共前后缀。

1. P[i] = P[k]，则next[i] = k + 1
2. P[i] != P[k]，则应该寻找一个x < k 来 **是否等于** P[i]。



如上图所示，需要找到使得A的x-前缀等于B的x-后缀的最大的x。

另外，子串A和子串B相等，且知道我们已经知道了子串A的最长公共前后缀，A的后缀就是B的后缀，A的最长公共前后缀的长度就是能找到的最大x值，即next[i - 1]。接下来可以将k改为next[i - 1]，进入下一个循环判断P[k]是否等于P[i]，重复以上步骤。



查找匹配

当确定了next数组之后，进行匹配工作，对于主串而言，其指针i不需要回溯，只有模式串指针j需要回溯。 对于任意状态下的i和j：

1. 若S[i] = P[j]，则两个指针同时前移
2. 若S[i] != P[j]，则j = next[j - 1]，指针回溯，继续匹配尝试。
3. 若j == P.length - 1，则匹配成功，找到了子串。

```
class Solution {
    public int strStr(String haystack, String needle) {
        //Next array
        if (needle.length() == 0)
            return 0;
        if (haystack.length() < needle.length())
            return -1;
        int[] next = new int[needle.length()];
        int j = 0;
        next[0] = 0;
        for (int i = 1; i < needle.length(); ++i)
        {
            if (needle.charAt(i) == needle.charAt(j))
            {
```

```
                next[i] = j + 1;
                j++;
            }
            else if (j == 0)
                next[i] = 0;
            else
            {
                j = next[j - 1];
                i--;
            }

        }
        //Perform search
        j = 0;
        int i = 0;
        while (i < haystack.length())
        {
            if (haystack.charAt(i) == needle.charAt(j))
            {
                i++;
                j++;
                if (j == needle.length())
                    return i - j;
            }
            else if (j == 0)
                i++;
            else
                j = next[j - 1];
        }
        return -1;
    }
}
```

---

438. Find All Anagrams in a String

567. Permutation in String

Sliding Window

解法一：两重循环加HashMap暴力搜索，超时

解法二：很诡异的Sliding Window

首先为需要查找的String制作一个HashMap。 设置WinSt和WinEd两个指针，分三种情况：

1. 当WinEd指向的地方在HashMap中有记录且Freq还没有降为0，说明这是一个合法位置，则在HashMap中Freq减一做记录。将WinEd前移。

   还需额外判断此时窗口大小是否为我们想要的大小，如果是则说明找到了。

2. 如果winEd和winSt在同一个位置，说明窗口大小为0，则把两个指针一同前移。

3. 其他情况，则将winSt前移，同时把前移后落下的Freq还回HashMap。

这个解法就是移动两个指针使得winEd指向的位置永远是合法的，如果下个位置不合法，winEd会在原地等winSt追上，直到窗口缩小为0，两者再一起前进。

总之.....挺奇怪的，有点像Two Pointers。

```
class Solution {
public:
    bool checkInclusion(string s1, string s2) {
        map<char, int> mmap;
        for (int i = 0; i < s1.length(); ++i)
            mmap[s1[i]]++;
        if (s2.length() < s1.length())
            return false;
        int winSt = 0;
        int winEd = 0;
        while (winEd < s2.length())
        {
                  if (mmap.find(s2[winEd]) != mmap.end() &&
mmap[s2[winEd]] != 0)
            {
                if (winEd - winSt + 1 == s1.length())
                    return true;
                mmap[s2[winEd]]--;
                winEd++;
            }
            else if (winEd == winSt)
            {
                winEd++;
                winSt++;
            }
            else
            {
                mmap[s2[winSt]]++;
                winSt++;
            }
        }
```

```
            return false;
        }
    };
```

解法二：真正的Sliding Window

参考1：官方题解

参考2：GeeksforGeeks

真正的Sliding Window的窗口大小是限制为我们需要的大小的，因为**题目限制了字符串的长度为特定的k值**，则窗口的大小永远是该k值。我们只需一步步向前移动窗口，每一步判断该窗口位置是否符合要求即可将时间缩短到一重循环。

首先仍然要建立一个HashMap，但是可以用数组来完成，记录每个字符的Frequency，因为是26个英文字母。 其次为窗口（在起始位置）建立一个HashMap，记录每个字符的Frequency，同样用数组来完成。 其次每次窗口移动到某个位置的时候，首先进行比较两个Frequency HashMap是否相同，如果相同则说明找到了。否则继续移动，移动的时候，因为窗口离开，所以窗口尾端的字符Frequency需要更新，同时窗口前端的字符Frequency也需要更新，其他不变。

```
class Solution {
    private int cmpMap(int[] m1, int[] m2)
    {
        for (int i = 0; i < m1.length; ++i)
        {
            if (m1[i] > m2[i])
                return 1;
            else if (m1[i] < m2[i])
                return -1;
        }
        return 0;
    }
    public boolean checkInclusion(String s1, String s2) {
        if (s1.length() > s2.length())
            return false;
        int[] s1map = new int[26];
        int[] s2map = new int[26];
        for (int i = 0; i < s1.length(); ++i)
        {
            s1map[s1.charAt(i) - 'a']++;
            s2map[s2.charAt(i) - 'a']++;
        }
        for (int i = 0; i < s2.length() - s1.length(); ++i)
```

```
            {
                if (cmpMap(s1map, s2map) == 0)
                    return true;
                s2map[s2.charAt(i + s1.length()) - 'a']++;
                s2map[s2.charAt(i) - 'a']--;
            }
            if (cmpMap(s1map, s2map) == 0)
                return true;
            return false;
        }
    }
```

---

771. Jewels and Stones

HashMap
