问题描述：给定目标字符串target（简称t）和模式串pattern（简称p），判断p是否包含在t中，即p是否是t的子串。为和代码统一，假设字符串的起始索引从0开始，p[i,j]表示pattern从i开始到j结束的子串（左右闭区间），分析时间复杂度均假设target长度问n，pattern长度为m。

示例：target = abcabccab, pattern = abcc, pattern是target的子串。

解决方法：

1. 暴力法：最显而易见的方案，以target的每个字符为起始位置，逐个比较该字符开始的等长字符串是否等于pattern，如果等于则包含，否则向后移位一个字符继续遍历比较直到结束。时间复杂度方面，最坏情况下不包含会比较n-m次，每次最多比较m位字符，所以总时间为O(m(n-m))。空间复杂度为O(1)，不需要额外的空间。
2. KMP算法：KMP算法相比暴力法来说改进的地方在于，利用模式串自身的特点和信息，在每次比较失败时尽可能多的移位。这里的信息指的是pattern的前缀信息，例如pattern=abcabcacab，abca是pattern的前缀，而且abca这个子串在第3位开始重复出现，即p[0,3]=p[3,6]，这样在比较字符串时，如果target=abcabcabcacaba，从t[0]开始比较到第7位时p[7]!=t[7]，不需要从t[1]开始从头比较，而是可以从t[3]开始，因为p[7]不等但p[0,3]=p[3,6]，既然比较到p[7]说明此前字符串是相等的，即t[0,6]=p[0,6]，而且p[0,3]=p[3,6]，可得p[0,3]=t[3,6]，所以直接从t[7]和p[4]处开始比较，相比于每次移位1个子符大幅提高了比较的效率。下面分为两部分做KMP算法的说明，第一部分介绍next数组的含义及求解方法，第二部分介绍KMP算法的流程。

1）next数组：next数组是用来标记模式串本身的前缀后缀信息的，正如上文举例，如果已知某个子串可匹配的位置，即第一个不匹配的字符，每次只需要移动模式串的当前匹配索引到下一个前缀没有覆盖到的字符位置，而在这个字符之前的所有pattern的子串都与target当前索引之前等长的子串相等。已知匹配的前缀即可将索引位置移动到第一个前缀不匹配的字符处，达到多级跳跃的效果，并且不会产生回溯，即始终向右移动，相比暴力法单字符移动要更快到达下一个可以匹配的位置。Next[j]=k的含义是，对于模式串pattern来说，k是最大的满足p[0,k-1]=p[j-k,j-1]且p[j]!=p[k]的值，在求next数组之前，先引入f(j)的概念，假设f(j)=k表示满足p[0,k-1]=p[j-k,j-1]的最大k值，但对p(j)和p(k)任何要求。例如patter=abcabcacab，求f[7]来说，需要求出最大的k满足p[0,k-1]=p[7-k,6]，如下表格所示

|  |  |  |  |
| --- | --- | --- | --- |
| k-1 | P[0,k-1] | 关系 | P[7-k,6] |
| 0 | a | == | a |
| 1 | ab | != | ca |
| 2 | abc | != | bca |
| 3 | abca | == | abca |
| 4 | abcab | != | cabca |
| 5 | abcabc | != | bcabca |

从表中可以看出，满足子串相等且k最大的值为k-1=3即，k=4，所以f(7)=4。但是这样存在一个问题，即每次求f(j)都需要从0开始遍历pattern，没有有效利用已经求得的前缀字符串的信息。例如求f(7)之前，已知f(6)=3，此时有p[0,2]=p[3,5]，如果p[3]=p[6]，那么也有p[0,3]=p[3,6]，根据定义可知f(7)=4=f(6)+1，而如果p[3]!=p[6]，那么求f[7]又变成了求新的前缀关系，而这个前缀是以p[3]结尾的（已知p[0,2]=p[3,5]，p[6]字符是未重复出现的前缀，那要找的新前缀一定不是以p[3]结尾且会在p[3]之前，所以需要从f(3)的位置开始找），即f(6)=f(3)。相比于从0开始遍历求最大k值来说，这种求法利用前缀信息节省了很多步骤。以abcabcacab为例，f数组和next数组值如下表：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Pattern[j] | a | b | c | a | b | c | a | c | a | b |
| f(j) | -1 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 |
| next[j] | -1 | 0 | 0 | -1 | 0 | 0 | -1 | 4 | -1 | 0 |

可以看到next[j]<=f(j)，这是因为此前说过的next相比f(j)多了一个约束条件:p[j]！=p[k]，因为p[j]!=p[k]时可以直接回退到k且k之前的前缀字符串和target是相等的，而p[j]==p[k]时，回退到k是没有意义的，因为p[j]不相等则p[k]也不相等，所以要在k之前。综上所述可以得到根据f(j)求解next[j]的递推公式：

if p[j] != p[k]: next[j] = f(j)

else: next[j] = next[f(j)]

关于为什么用next而不用f(j)则在上文讨论过，因为next数组回溯效率相比f数组更高，避免了很多多余即一定匹配失败的回溯。

代码如下：

def get\_next(pattern):  
 i = -1  
 j = 0  
 next = [-1] \* len(pattern)  
 while j < len(pattern)-1:  
 if i == -1 or pattern[i] == pattern[j]:  
 i += 1  
 j += 1  
 if pattern[i] != pattern[j]:  
 next[j] = i  
 else:  
 next[j] = next[i]  
 else:  
 i = next[i]  
 return next

2）KMP算法：已知next数组之后，KMP算法很简单，在每次匹配的过程中，如果出现不匹配的字符即target[i]!=pattern[j]，令j=next[j]，如果相等则i j++。因为不匹配时，p[0,j-1]=t[i-j, i-1]，j回溯到next[j]可保证p[0,next[j]-1]=t[i-next[j],i-1]，由next定义可知其正确性，这样就能相对暴力算法加快移位。

代码如下：

def kmp(target, pattern):

next = get\_next(pattern)

i, j = 0, 0

while i < len(target) and j < len(pattern):

if j == -1 or pattern[j] == target[i]:

i += 1

j += 1

else:

j = next[j]

if j >= len(pattern):

return i - len(pattern)

return -1

时间复杂度，首先对pattern遍历一次求next需要O(m)，然后遍历一次target需要O(n)，即最终的时间复杂度为O(n)。