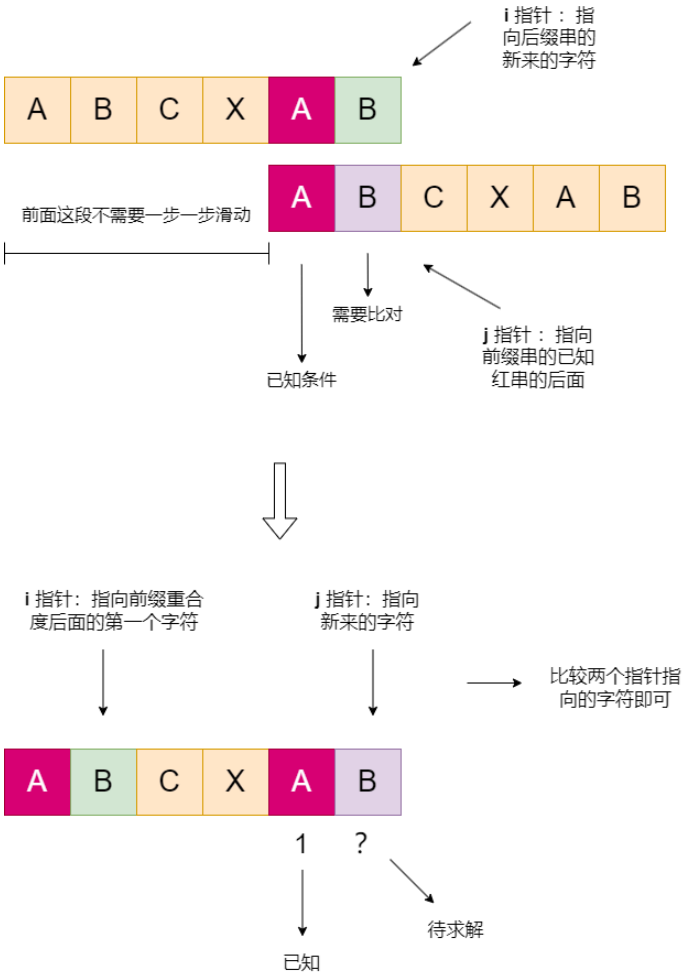
第8章 串与KMP算法

8.1 最长公共前后缀

问题定义
在讲解这章主要内容之前，先穿插一个算法，先把这个算法讲明白后再详细介绍后面的内容，
何为求解最长公共前后缀，这里举几个例子：
对于字符串aba，其前缀a和后缀a相同；
对于字符串abcxabc，其前缀abc和后缀abc相同；
对于字符串abxdeeabxd，其前缀abxd和后缀abxd相同；
通过这三个例子，首先你明白我这里讲的前后缀是什么意思了把，就是字符串的前面的一些字符，
后缀就是尾巴的一些字符，最长相同的前缀和后缀，就是上面三个例子中所讲的那样，首先
前缀几个字符和后缀几个字符要一样，其次长度能有多长就多长。

ok，问题明确了，就是给你一个字符串，你把这个字符串的最长公共前后缀长度求出来，这里
知道长度就行，不需要把前后缀像上面例子一样弄出来，上面三个字符串的最长公共前后缀长度分别为1，2，4

暴力求解思路
求解abcxabc的最长公共前后缀长度：
第一步比较长度为6的前缀abcxab和后缀bcxabc是否一样，
第二步比较长度为5点前缀abcxa和后缀cxabc是否一样，
第三步比较长度为4的前缀abcx和后缀xabc是否一样，
第四步比较长度为3的前缀abc和后缀abc是否一样，发现一样，返回 算法时间复杂度为O(n^2)



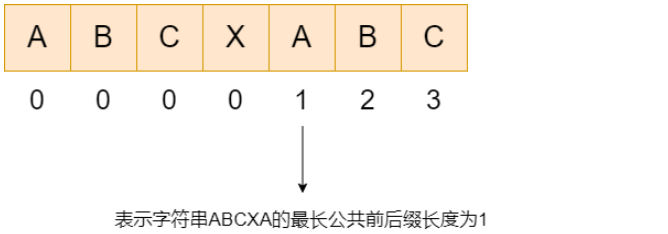
对于上面abcxabc，单单求它的最长公共前后缀，暴力求解是简单的，但是我告诉你，在后面KMP算法中，
我们还需要求这个字符串的所有长度前缀的这个东西（最长公共前后缀），这么说有点绕口，我直接告诉你，
对于我们前面讲到的这个字符串abcxab，需要把下面所有的字符串的最长公共前后缀长度都求出来：
a，ab，abc，abcx，abcxa，abcxab，abcxabc(它自己)
那还不简单，对这里的每个字符串，都使用上面暴力求解算法过一遍不就行了？
是的，这么做确实能求出来，不过效率极低O(n^3)，这个效率是无法忍受的。

迭代求解算法
这里说的迭代，意思如下：
如果知道字符串a的最长公共前后缀长度，就能推演出ab的最长公共前后缀长度；
如果知道字符串ab的最长公共前后缀长度，就能推演出abc的最长公共前后缀长度；
如果知道字符串abc的最长公共前后缀长度，就能推演出abcx的最长公共前后缀长度；
如果知道字符串abcx的最长公共前后缀长度，就能推演出abcxa的最长公共前后缀长度；
如果知道字符串abcxa的最长公共前后缀长度，就能推演出abcxab的最长公共前后缀长度；
如果知道字符串abcxab的最长公共前后缀长度，就能推演出abcxabc的最长公共前后缀长度；

为了后面更加清晰的讲解这件事情，我们先瞪眼看把所有长度的前缀子串的最长公共前后缀长度求出来：
a,没有最长公共前后缀，长度为0

ab,没有最长公共前后缀，长度为0
abc,没有最长公共前后缀，长度为0
abcx,没有最长公共前后缀，长度为0
abcxa,其最长公共前后缀是a，长度为1
abcxab,其最长公共前后缀是ab，长度为2
abcxabc,其最长公共前后缀是abc，长度为3

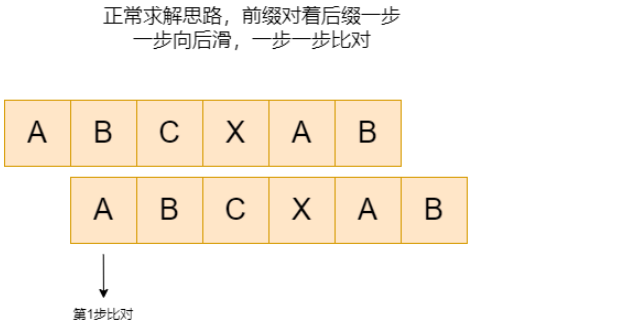
但是这种表示方式不好，我们下面画张图，把数字表示在各种长度前缀字符串的最后一个字母下面

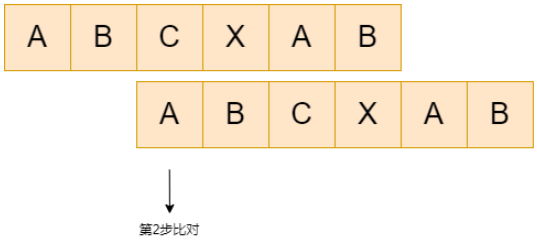


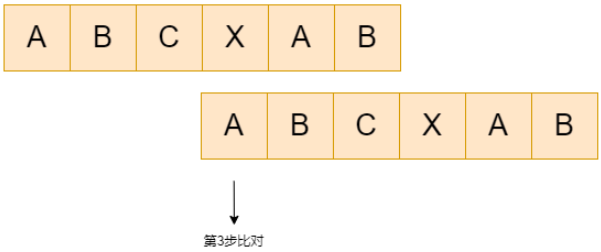
答案我们已经瞪眼看给求解完毕了，但是如何递归求解呢，我们把这个例子中最后三个递推过程讲解一下，看看能否发现递推规律

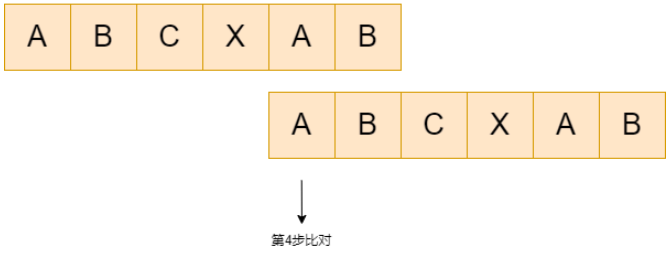
第一种情况

- 如果暴力求解字符串abcxab的最长公共前后缀

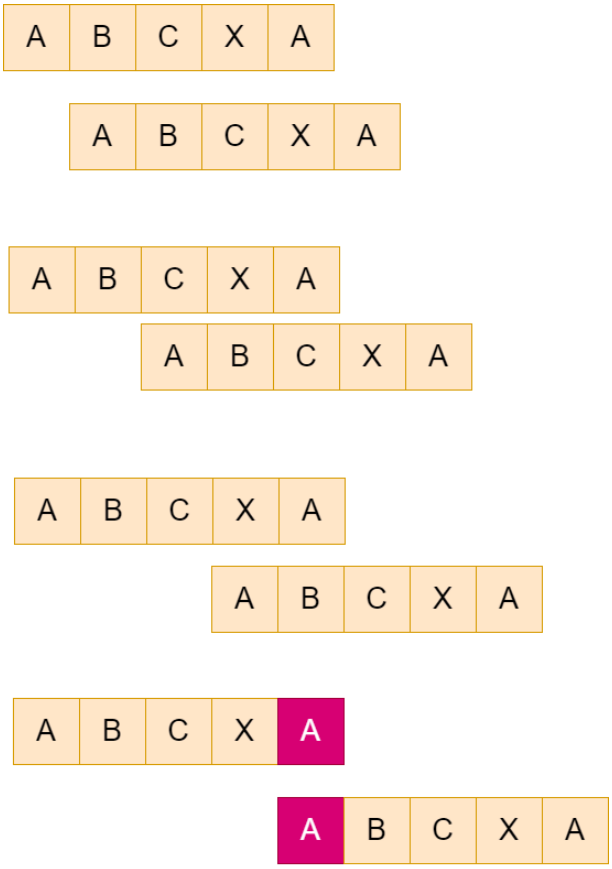








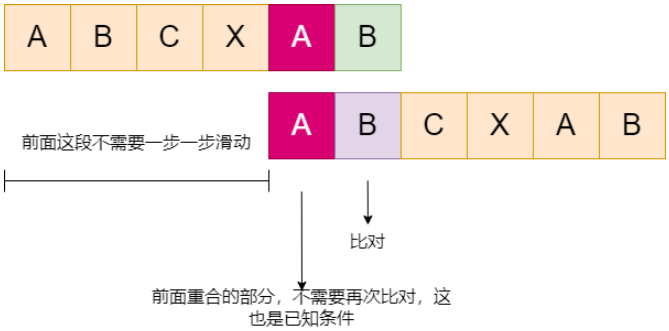
- 挖掘已知条件，就是已经知道了字符串ABCXA的最长公共前后缀长度
- 本来我们不必拿着窗口滑动，但是为了发现规律，这里还是画了出来



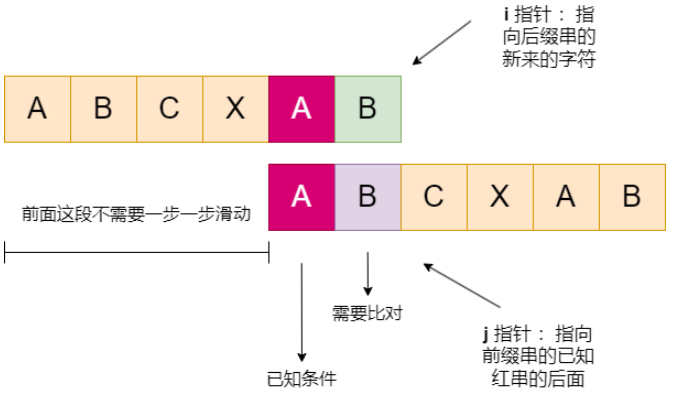
已知条件：字符串ABCXA的最长公共前后缀长度为1，如上图。
由于这是已知条件，所以前三步比对实际是不存在的，这里
画出来，只是让你把这里和前面那个多一个字符的字符串
进行暴力求解时的前三步进行比较，从而发现规律

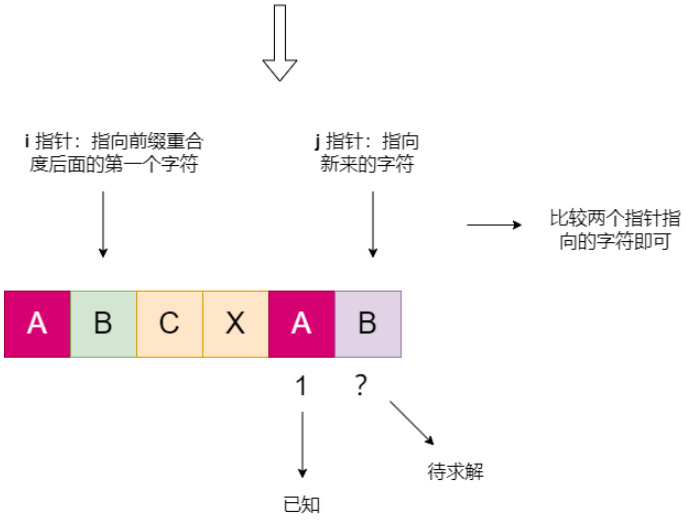
从字符串abcxa递推到abcxab的规律：
从上图不难看出，求解这两个字符串的的前三步完全一样，都是对比相同的字符，对比失败的原因也是完全一样，也就是说我们求解abcxab这个字符串的时候，前3步比对都可以省略，直接从第4步进行比对，而第四步比对的前缀和后缀的第一个字符都是A，说明第一个字符重合了，而这个原本就是已知条件，所以我们仅仅需要比对的只是A后面新加进来的字符B，如果这个字符比对重合了，那么仅仅需要在原有重合1的基础上+1即可。

这就是第一个规律，简而言之，就是仅仅需要下面这一步比对：



把上面比对的过程，使用下面两个指针进行简化

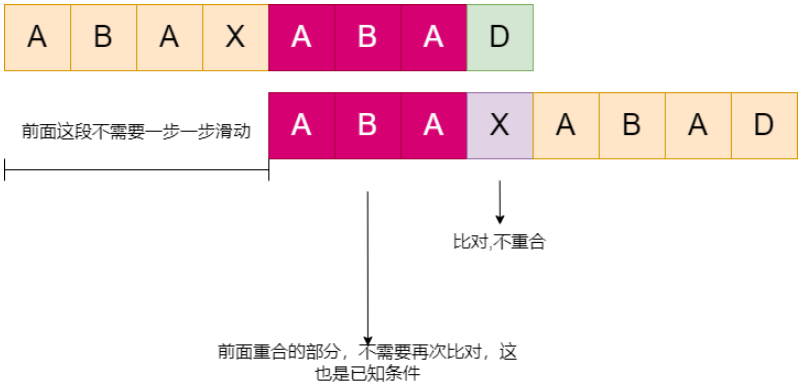




第二种情况

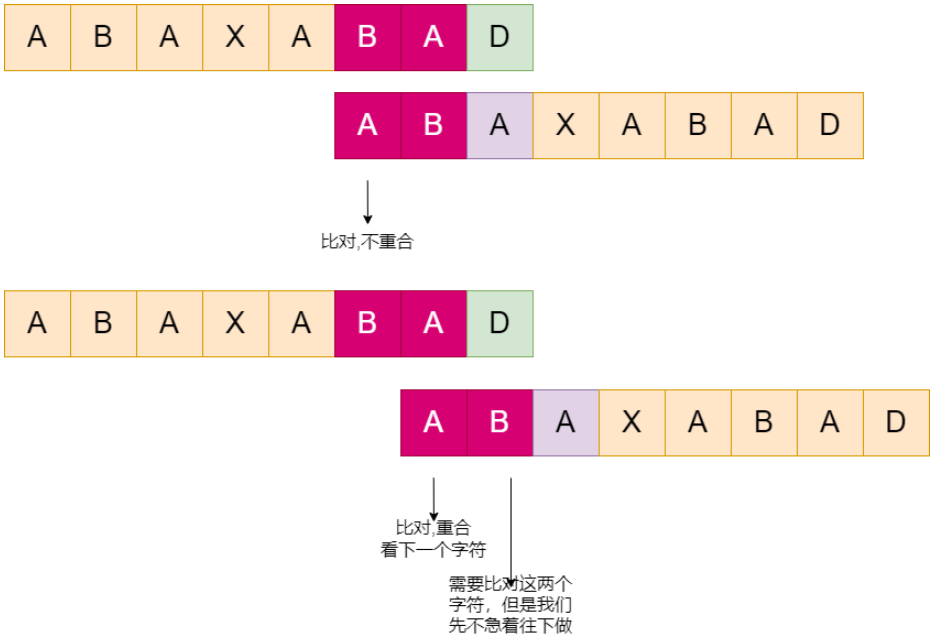
阶段一

上面是重合度后面的一个字符比对成功的情况，把前一个重合度+1即可求解，那么如果重合度后面的一个字符比对不成功呢？我们接下来就这种情况进行详细讲解，这次是从字符串ABAXABA递推ABAXABAD，没错，换例子了，如下图：

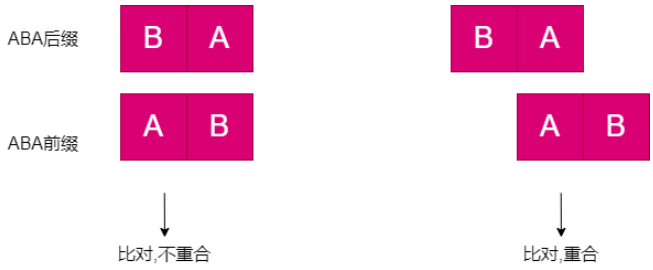


阶段二

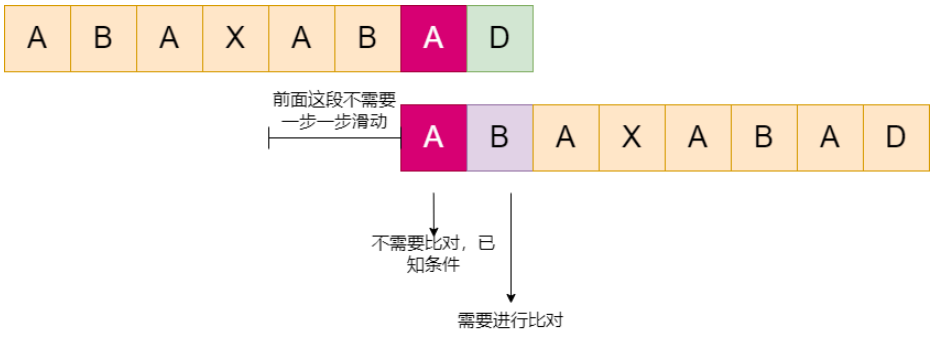
既然比对不重合，我们无非是沿着暴力求解的思路继续滑动窗口，假如我们按照暴力把后面求解下去，过程如下图：



然而这个过程实际上还是能够简化的，我们可以利用ABA这个字符串的最长公共前后缀长度为1这个信息，也就是说前面两步暴力滑动比较的过程和ABA前两步滑动比较的过程完全一样，如下图（注意，我们已知前面所有短的前缀串的长度）

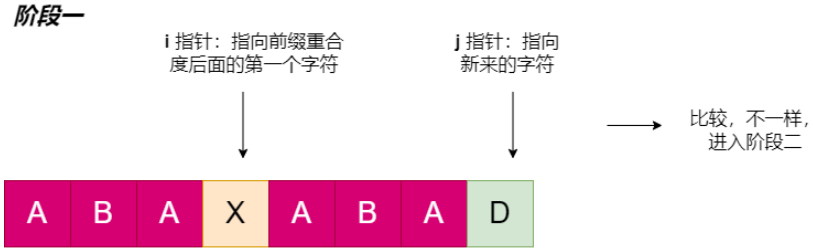


因此，上面两步暴力滑动过程，实际还是能够根据ABA的最长公共前后缀长度信息进行简化的，如下图

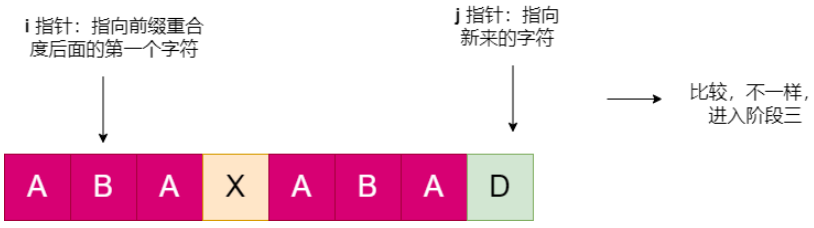


把两个阶段的滑动窗口改为指针描述

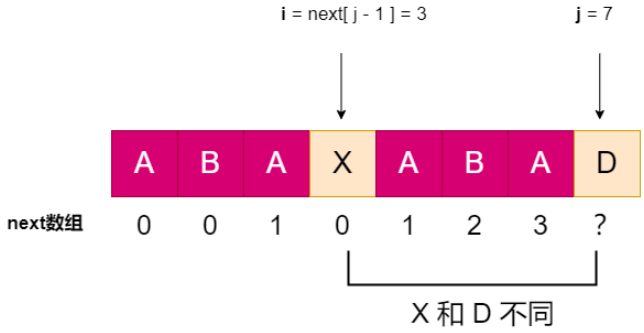
阶段一

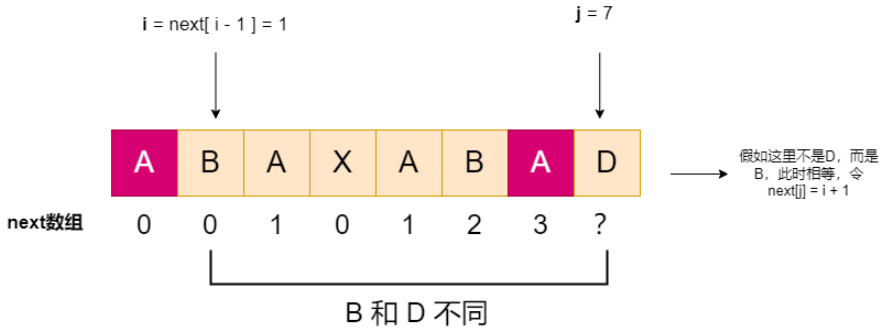


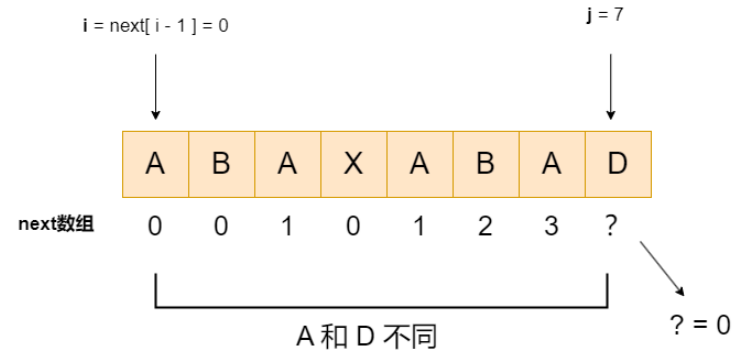
阶段二



为了方便将上面的过程转换为代码，我们把每个前缀字符串的最长公共前后缀长度信息标在下面，
然后把指针移动过程画出来







也就是i每次跳到的地方，都是上一次地方前面那个字符下面标注的位置，初始的时候i和j都设置在最后一个字母；
如果中间哪步比对成功了，就是它前面那个位置下面的数字+1，如果走到底还是一直比对不成功，结果为0；
注意，上面标红意思是跳过比对的部分，虽然前面已经详细讲过了，这里再统一重述一遍。
第一次比对的是长度为4的前缀ABAX和后缀ABAD，其中ABA无需比对；
第二次比对的是长度为2的前缀AB和后缀AD，其中A无需比对；
第三次比对的是长度为1的前缀A和后缀D，还是不一样，所以没有最长的公共前后缀。
这里顺带提一下，第一次比对为什么跳过了长度为7，6，5的前后缀，因为长度为7的这个串，已经知道重合度为3了，
因此暴力滑动直接跳过，直接划到前缀中的ABA和后缀中的ABA对其重合为止。
同理，比对完长度为4的前后缀之后，跳过比对长度3的前缀ABA和后缀BAD，是因为，前缀中的AB和后缀中的BA，
对齐后一定需要再滑动一位才能重合，这是根据ABA的前后缀重合度1知道的，因此滑完后，只剩下AB和AD进行比对了

代码实现

前面讲了如何利用迭代一步一步求解一个变长的字符串的最长公共前后缀长度，后面我们把这些长度表示在
一个next数组中。对于前面的i指针跳转，背后实际就是前面讲的省略的暴力求解中的某些滑动和比对过程，
指针的表示只是为了代码实现方便，滑动窗口的表示只是为了理解这个过程方便，不然你会一头雾水，不清楚
怎么就能利用这个前面短串的重合度信息，就能跳过去，很多视频里讲解到这里，只说利用这个信息就跳过去了，
没讲清楚为什么，只告诉你这么做，原因实际上没讲清楚，他把这种做法当成原因来讲，说“因为前面这串信息已知，
所以就这么跳了过来”，导致其中的来龙去脉你还是一无所知。

挖掘规律的关键
需要表示出前一个问题的暴力求解步骤，以及当前待求解问题的暴力求解步骤，然后这两套步骤一定有重合的部分
至于你找到规律后，能否用代码表示出来，这又是另一件事了

代码执行过程：
设要匹配的字符串放在数组P中，然后要求解的各个长度的最长公共前后缀长度，放在next数组中
首先设定next[0] = 0，初始求单个字符的最长公共前后缀长度，为0，直接写，然后后面使用递推，怎么递推呢？如下：
j首先设置在位置0，循环开始如下：
1.通过next[j-1]的值获取i的下标(赋值给i)，
2.然后比较P[i]和P[j]，
如果相等，令next[j] = i+1,然后令j++，执行下一次循环(回到第1步)；
如果不相等，令i=next[i-1],回到第2步。
由于执行了i向前跳，这里设置边界条件，如果i为0，且比较不相等，令 next[j] = 0，然后j++，开始下一次循环。
那么循环结束的条件是什么呢，就是j走到头的时候

In [1]:
```
#include <iostream>
using namespace std;
```

注意c++标准库中已经包含了std::next,命名不要冲突

In [2]:
```
int* getNext(char *P){
    int m = strlen(P);
    int *next_ = new int[m];
    next_[0] = 0;
    int i = 0;
    int j = 1;
    while (j < m){
        if (P[i] == P[j] ){
            next_[j] = i+1;
            j++;
            i = next_[j-1];
            continue;
        }
        else{
            if (i == 0) {next_[j]=0;j++;i = next_[j-1];continue;}
            else {i = next_[i-1];continue;}
        }
    }
    return next_;
}
```

由于上面回到第2步不好实现，所以稍微改了下代码逻辑：
设置好初始i，j位置，进入循环：
比较P[i]和P[j]，出现下面3种情况：
1.相等，更新next数组，j向后移动一格，i跳到next[j-1]的位置，回到循环入口；
2.不相等，且i已经到达了0的位置，设置next[j]为0，然后j向后移动一格，i跳到next[j-1]的位置，回到循环入口；
3.不相等，且i没有达到0点位置，i向前跳到next[i-1]的位置，回到循环开始进行下一轮比较；

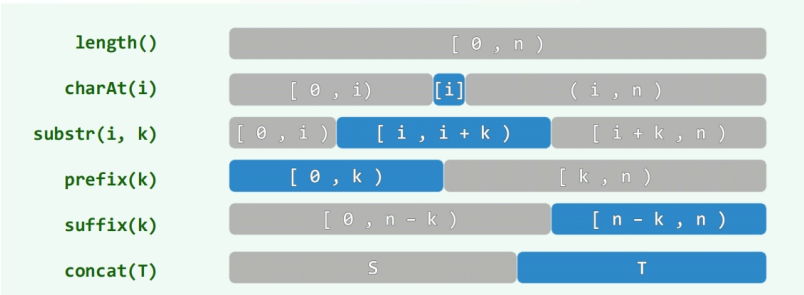
	A	B	A	X	A	B	A	D
next数组	0	0	1	0	1	2	3	0

In [3]:
```
char P[] = "ABAXABAD";
auto Next = getNext(P);
```

In [4]:
```
for (int i = 0; i < 8; i++)
    cout << Next[i] << " ";
```

```
0 0 1 0 1 2 3 0
```

8.2 串的定义和算法

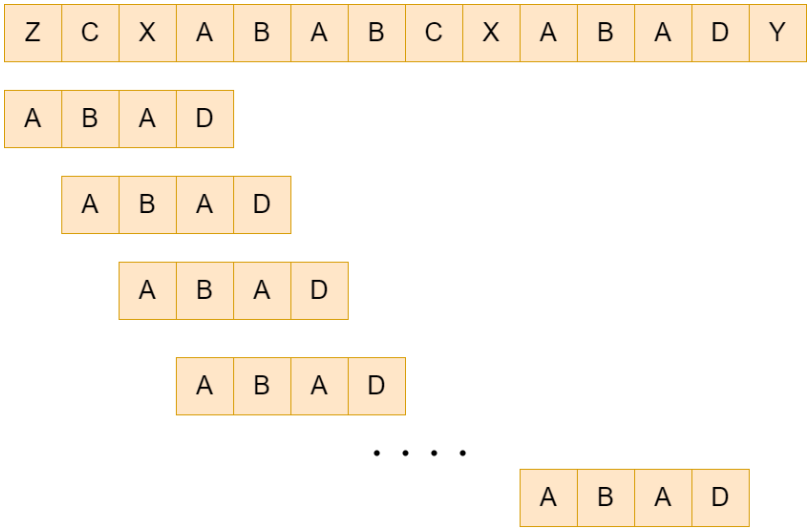


串，就是string，就是字符串，上面既是串的基本接口，也是串中最基本的概念，
substr(i,k)就是从i开始的长度为k的子串，
prefix(k)就是前k个字符组成的字符串，叫前缀，
suffix(k)就是最后k个字符组成的字符串，叫后缀。
串的基本接口和操作就是上面这几个，但是你可能和想到，怎么没有repalce替换，search查找，
是的，标准string接口都实现了在一个超长字符串中查找一个短模式串的功能，
文本串中查找模式串，主要需要解决：
模式串是否存在？
模式串出现在哪里？
模式串出现了几次？
这三个问题，KMP算法就是解决模式匹配问题的，这个基本问题解决后，其他什么各种高级功能，
如repalce字符串替换，就都能实现了，因此KMP算法是一个核心算法。 我们这里就不从头建立一个string类了，直接写出KMP算法

KMP算法

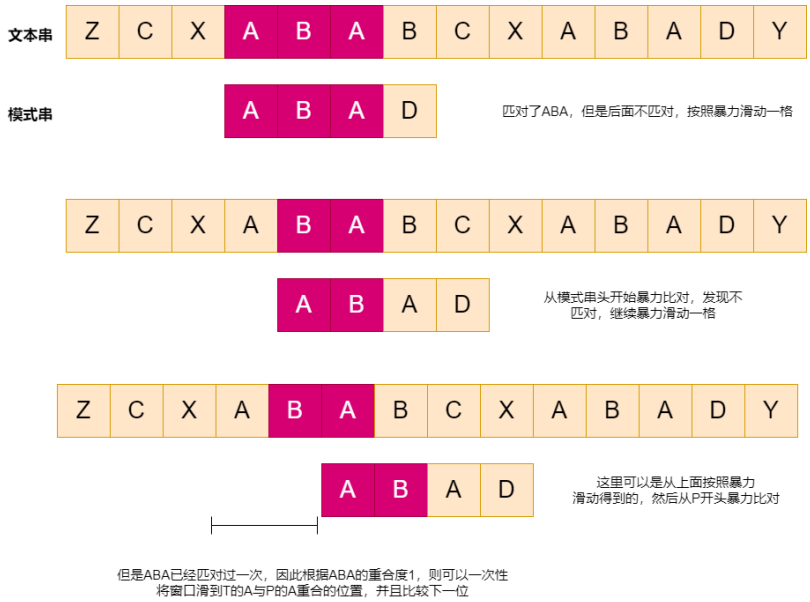
问题： 给定两个输入字符串，即文本串T，和模式串P，找出文本串中第一个匹配的模式串，如果找出来了，返回其位置

暴力求解思路
如下图，就是一格一格移动，然后分别比对

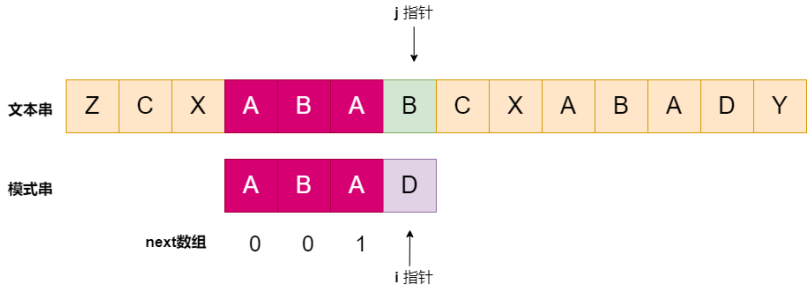


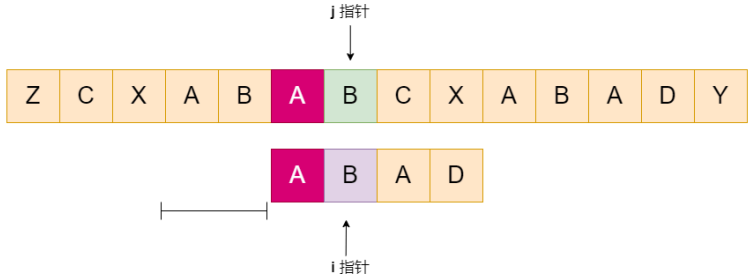
简化暴力求解的重复部分

注意，下面红色的部分，只是第一次进行了比对，用来观察后面两次滑动和第一次匹对有什么关系



从上面滑动比对我们可以看出，这里和前面讲的完全就是一回事啊，只不过前面把前缀拿下来，在后缀下面滑动，并且前面重合了一部分，而这里换成了模式串在文本串下面滑动，也有一部分重合，所以它们滑动滑一大截的规则完全一样，就是看前面匹对的部分的最大公共前后缀长度，于是上面可以简化成下面两步





ok，前面相同，遇到不同的处理办法有了，就是令i = next[i-1] 即可，效果看上去好像对窗口滑动了。
然后接下来比对P[i]和T[j]，遇到相同的怎么办呢，很简单，i和j同时向后移动一格即可。
注意，我们模式串P的next数组我们已经提前算出来了，可以直接查表

算法实现
初始化i=j=0，然后进入循环：
比较P[i]和T[j]，会有如下两种情况：
如果相等，i和j向后移动一格，回到循环开始继续；
如果不相等，令i = next[i-1]，然后回到循环开头继续；
关于i和j的边界条件，首先j一定小于文本串T的长度，
假如P[i]和T[j]一直相等，i抵达P结尾，i和j继续移动一格，i就等于strlen(P)了，这时候匹对成功返回；
假如P[i]和T[j]不相等，并且此时i在位置0，没有next[i-1],这时候没有公共串，j暴力向前走一格即可

In [5]:
```
int match(char* P,char* T){
    int n = strlen(T);
    int m = strlen(P);
    int* Next = getNext(P);
    int i = 0;
    int j = 0;
    while (j < n && i < m){
        if (P[i]==T[j]) {i++;j++;continue;}
        else {
            if (i == 0) {j++;continue;}
            else {i = Next[i-1];continue;}
        }
    }
    return j-i;
}
```

In [6]:
```
char P[] = "ABAD";
char T[] = "ZCXABABCXABADY";
```

In [7]:
```
match(P,T)
```

Out[7]: 9

In [8]:
```
T[9]
```

Out[8]: 'A'

In [9]:
```
T[10]
```

Out[9]: 'B'

In [10]:
```
T[11]
```

Out[10]: 'A'

In [11]:
```
T[12]
```

Out[11]: 'D'

In [ ]:
```

```