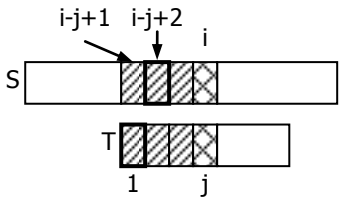
1. 串的模式匹配

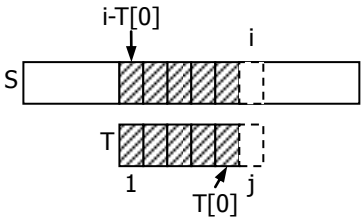
在主串中寻找与给定的模式串相等的子串（首次）出现的位置，即子串的定位操作，通常称作串的模式匹配。例如：主串 S="THIS IS HIS BAG"，模式串 T="IS"。如果从主串 S 的开头开始定位模式串 T，模式匹配的结果就是 3，即 Index(S, T)=3。通常，还可以指定一个在主串中查找的起始位置，使算法更加灵活。如：从主串 S 中第 7 个字符开始定位模式串 T，结果就是 10，即 Index(S, T, 7)=10。一般地，用 Index(S, T, pos) 表示从主串 S 中第 pos 个字符开始查找模式串 T 出现的位置。

(1) 简单的模式匹配算法

一种简单的模式匹配算法思路是：从主串 S 的第 pos 个字符开始和模式串的第一个字符比较，若相等，则继续逐个比较后续字符；否则再从主串的下一个字符开始重新和模式串比较。依次类推，直到匹配成功或失败。算法比较主串和模式串中的字符 S[i]和 T[j]时，一旦两者不匹配（不相等）（如图（a）所示），主串要回溯到匹配开始时的下一个字符，而模式串则需要回溯到开头。而匹配成功时，模式串在主串中的位置是 i-T[0] （如图（b）所示）。



(a) S[i]与 T[j]失配时的回溯位置



(b) 匹配成功时的情况

算法实现如下：

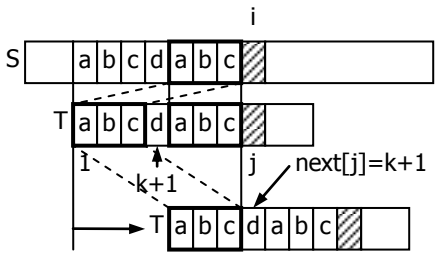
```
int Index ( SString S, SString T, int pos )
{
  i = pos;   j = 1;
  while ( i<=S[0] && j<=T[0] ) {
    if ( S[i]==T[j] ) { ++i; ++j; }
    else { i = i-j+2;   j = 1; }
  }
  if ( j>T[0] )   return i-T[0];
  else   return 0;
}
```

算法在匹配过程中失配时主串需要回溯到匹配开始的下一个字符，而模式串则回溯到开头。如果主串长度为 n，模式串长度为 m，则简单模式匹配算法的时间复杂度在最好情况下为 O(n+m)，但最坏情况下却达到 O(n*m)。例如：主串 S= “aaaaaaaaab”，模式串 T= “aaab”。模式匹配过程中需要反复回溯，最终比较的字符数达到 7*4 个，此时的时间复杂度即 O(n*m)。在通常的实际应用中，这种最坏情况出现的可能性并不大，该算法的时间复杂度接近于 O(n+m)。但如果经常出现这种最坏情况，就需要修改算法了。KMP 算法就能使模式匹配的时间复杂度在最坏情况下也能达到 O(n+m)。

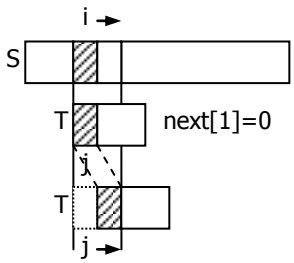
(2) KMP 算法

KMP 算法是对简单模式匹配算法的改进，它可以保证在 O(n+m)的时间内完成模式匹配。KMP 算法的主要改进在于：在匹配过程中 S[i]≠T[j]出现字符失配时，主串指针 i 不回溯，而是充分利用已经得到的“部分匹配”结果，将模式串指针 j 回溯到一个“恰当”的位置，然后继续进行比较。这个模式串回溯的“恰当”位置通常由失配函数 next[j]来定义，而理解 KMP 算法的关键就是理解失配函数 next[j]的含义。

为了进一步理解失配函数 next[j]的含义，举例如下图（a）。当 S[i]与 T[j]失配时（这里 j=8），显然，经过前面的比较，S[i]与 T[j]前面 j-1 个字符的子串是相等的，即 S[i-j+1..i-1]=T[1..j-1]=”abcdabc”。观察该子串不难发现，其开头 k=3 个字符的子串与其末尾 k 个字符的子串相等，都等于”abc”。这也意味着，主串 S[i]前面的 k 个字符 S[i-k..i-1]与模式串 T 开头的 k 个字符 T[1..k]已经完全匹配。所以，接下来，主串不用回溯，只要让模式串回溯到 T[k+1]的位置（即令 j=k+1）继续比较就可以了。从失配函数的角度来看，这里就有 next[j]=k+1，该例中即有 next[8]=4。



(a) KMP 算法中 S[i]与 T[j]失配时模式串 T 的回溯位置为 next[j]



(b) KMP 算法中 S[i]与 T[1]失配时主串和模式串都要向后滑动

根据前面对具体实例的分析，可以大致总结出失配函数 next[j]的计算方法：观察模式串失配字符前面的子串 T[1..j-1]，若发现其开头（从 T[1]开始向后数）k 个字符的子串与其末尾（从 T[j-1]开始向前数）k 个字符的子串相等，则有 next[j]=k+1。需要说明的是，为了让模式串回溯后，已经匹配的子串尽可能长，这里的 k 应该尽可能大，但要小于子串 T[1..j-1]的长度 j-1。例如：若 T[j]失配时 T[1..j-1]="aaaaa"，则应取 k=4，从而得到 next[j]=5。另外，这里 k 的最小值可以是 0，即当在模式串 T[1..j-1]中找不到开头 k 个字符和结尾 k 个字符相等时，可以认为 k=0。例如：当 T[j]失配时 T[1..j-1]="abcde"，此时在开头和结尾找不到相等的子串，可以认为 k=0，于是 next[j]=1。

例 1：模式串 T= "ababaaab"，则 next[6]=_____。
【解析】求模式串任意位置 j 对应的 next[j]值的问题，可以采用观察法，即直接观察前面的子串 T[1..j-1]，确定首、尾最长相等子串的长度 k，即可得到 next[j]=k+1。这里，求模式串 T 的 next[6]，观察第 6 个字符前面的子串 "ababa"，可以看出分别从开头和末尾最多取长度为 k=3 的子串使两者相等，都是 "aba"，所以，next[6]=k+1=4。

根据以上分析，在 KMP 算法中，主串 S[i]与模式串 T[j]失配时，主串 i 不回溯，模式串 j 根据失配函数回溯到 next[j]。在已经求得失配函数的情况下，只要在前面简单模式匹配算法的基础上稍加修改即可：

```
    if ( S[i]==T[j] ) { ++i; ++j; }
    else j = next[j];    // 主串 i 不回溯，模式串 j 回溯到 next[j]
```

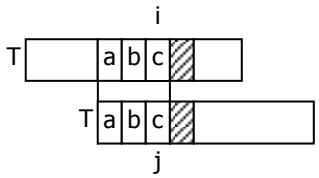
但是，还要注意一点：我们约定 next[1] = 0。当主串 S[i]与模式串第一个字符 T[1]比较失配时（如上图（b）所示），因为 next[1]=0，执行 j=next[j]就会使得 j 变成 0。接下来，若拿 T[0]与主串 S[i]比较显然是错误的，这里的 T[0]实际上是字符串的长度。实际上，若主串中 S[i]与模式串第一个字符 T[1]不相等，就没有必要回溯再继续比较了，此时主串位置 i 应当加 1 向后移动一个字符，再从模式串开头 j=1 开始继续比较（注意到此时 j==0，只要让 j 加 1 就可以了）。
最终，KMP 算法的实现如下：

```
int Index_KMP ( SString S, SString T, int pos )
{
  i = pos;  j = 1;
  while ( i<=S[0] && j<=T[0] ) {
    if ( j==0 || S[i]==T[j] ) { ++i; ++j; }
    else j = next[j];
  }
  if ( j>T[0] )  return i-T[0];
  else   return 0;
}
```

KMP 算法的时间复杂度是 O(n+m)，而且只需对主串扫描一次，但在进行模式匹配之前需要先计算模式串的失配函数，而且需要额外的 m 个空间存储失配函数的结果。下面讨论求模式串失配函数的具体方法。

(3) 求 next[]算法

求模式串失配函数 next[]的过程与 KMP 算法类似，只是主串与模式串是相同的。
若匹配过程中遇到 T[i]==T[j]，j<i，如下图所示，容易看出子串 T[1..i]中开头 j 个字符的子串与末尾 j 个字符的子串相等，于是若 T[i+1]失配时模式串应回溯至 T[j+1]继续比较，即得到失配函数值 next[i+1]=j+1。反之，若匹配过程中 T[i]≠T[j]，则利用 KMP 算法，令 j 回溯至 next[j]。



匹配至 T[i]==T[j]时，有 next[i+1]=j+1

下面是求失配函数 next[]的算法实现。注意算法开始时，要初始化 next[1]=0。由于循环中访问的是 next[i+1]，为防止下标越界，循环条件修改为 i<T[0]（而不是 i<=T[0]）。同时，语句 next[i+1]=j+1 在 i 和 j 都加 1 之后也简化成了 next[i]=j。

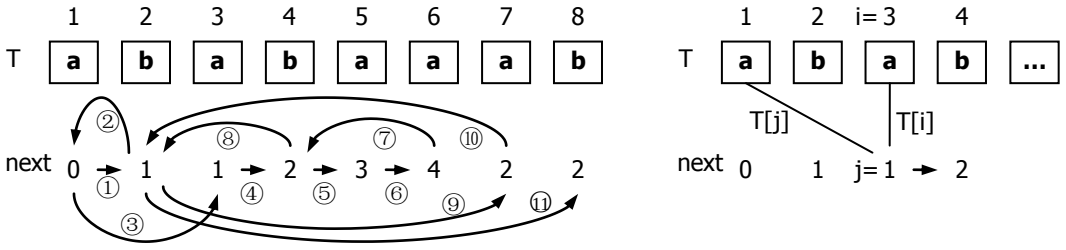
```
int get_next ( SString T, int next[] )
{
  i = 1;   j = 0; next[1] = 0;
  while ( i<T[0] ) {
    if ( j==0 || T[i]==T[j] ) { ++i; ++j; next[i] = j;}
    else j = next[j];
  }
}
```

例 2：模式串 T=“abababaab”的 next 函数值依次为_____。

【解析】求给定模式串的所有 next[]值，可以有两种方法。第一种方法是观察法，逐个求解 next[]值，具体步骤不再赘述。这种方法在模式串较长的情况下比较繁琐，容易出错。

第二种方法是利用前面的 get_next()算法进行计算。为了便于计算，下面将计算过程用图形表示（如下图（a））。图中向右的箭头表示“计算过程”，即算法中的“i 加 1、j 加 1 然后令 next[i]=j”，向左的箭头表示 j 的“回溯过程”，即算法中的“j=next[j]”。满足条件 j==0 或 T[i]==T[j]时，执行“计算过程”，并用一个由 next[i]指向 next[i+1]的向右箭头表示；否则，执行“回溯过程”，用从当前 next 值（等于 j）向左指向 next[j]的箭头表示。

计算模式串 T 的 next[]值的整个过程如下图（a）所示，最终得到结果为 0、1、1、2、3、4、2、2。下图（b）则进一步说明了从 next[3]到 next[4]的计算过程中，主要变量 i、j、T[i]和 T[j]之间的关系。

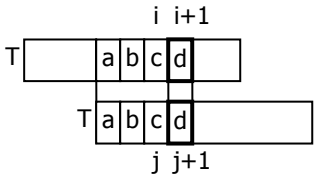


(a) 按照 get_next()算法求 next[]的过程　　(b) 求 next[4]的计算过程

实际计算过程中，只要画出表示“计算过程”和“回溯过程”的箭头连线，就能够完整地反映出 get_next()算法的执行过程。这种计算 next[]值的方法简便、直观，只要稍加练习，该方法应该不难掌握。

(4) 求 next[]修正值算法

前面算法求得的失配函数 next[]值在特定条件下还可以进一步改进，以减少不必要的回溯。在前面的算法中已经知道，若匹配过程中遇到 T[i]==T[j]，可以得到 next[i+1]=j+1，即 T[i+1]失配时模式串应回溯至 T[j+1]继续比较。进一步考虑，如果此时还有 T[i+1]==T[j+1]，如下图中均为‘d’，那么 T[i+1]失配时，模式串回溯到 T[j+1]也必然失配，此时不妨直接回溯至 next[j+1]效率更高。若仍然失配，还可以继续类推。



匹配至 T[i]==T[j]时，如果有 T[i+1]==T[j+1]，
则修正为 next[i+1]=next[j+1]

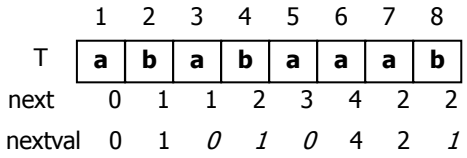
例如，对于模式串 T="aaaab"，失配函数值为{0，1，2，3，4}，其中 next[3]==2，同时 T[3]==T[2]=='a'，也就是说，如果 T[3]失配（主串中对应字符不等于'a'）回溯到 T[2]的话，结果一定还是失配，此时应继续回溯到 next[2]==1。实际上 T[1]==T[2]==T[3]=='a'，可以继续回溯到 next[1]==0。最终修正的失配函数值为{0，0，0，0，4}。

求失配函数 next[]修正值的算法实现如下：

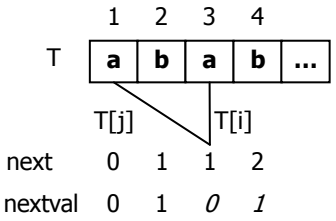
```
int get_nextval ( SString T, int nextval[] )
{
  i = 1;   j = 0; nextval[1] = 0;
  while ( i<T[0] ) {
    if ( j==0 || T[i]==T[j] ) {
      ++i; ++j;
      if (T[i]!=T[j]) nextval[i] = j;
      else nextval[i] = nextval[j];   // 修正失配函数
    }
    else j = nextval[j];
  }
}
```

例 3：模式串 T= "ababaaab" 的 nextval[]值为_____。

【解析】计算给定模式串的 nextval[]（修正）值时，可以根据 get_nextval()算法一次求出，但如果先计算出模式串的 next[]值，再计算失配函数的修正值 nextval[]就很简单了。开始时，nextval[1]初值为 0，然后依次计算模式串中后续字符的 nextval[]值。对于其中第 i 个字符，假设有 next[i]==j，如果此时若 T[i]≠T[j]，则不需修正，令 nextval[i]=next[i]=j 即可；反之，若 T[i]==T[j]，则需要修正，令 nextval[i]=nextval[j]即可。下图（a）是计算模式串 T 的 nextval[]值的过程。其中，已知 next[3]==1 计算 nextval[3]时（下图（b）），考虑到 T[3]==T[1]== 'a'，所以修正为 nextval[3]=nextval[1]=0。



(a) 根据模式串的 next[]值计算 nextval[]修正值



(b) 求 nextval[3]的计算过程

典型例题：

1．串 'ababaaababaa' 的next数组为（ ）。

A．012345678999

B．012121111212

C．011234223456

D．012312322345

【解析】注意到四个选项中第5个字符的next[]值分别为4、2、3、1，均不同，因此直接计算next[5]即可作答。观察第5个字符之前的子串"abab"，显然开头与末尾最多各取2个字符组成的子串相等，都是"ab"，故next[5]=3。正确选项为C。

2．字符串‘ababaabab’的nextval 为（ ）

A．(0,1,0,1,0,4,1,0,1)

B．(0,1,0,1,0,2,1,0,1)

C．(0,1,0,1,0,0,0,1,1)

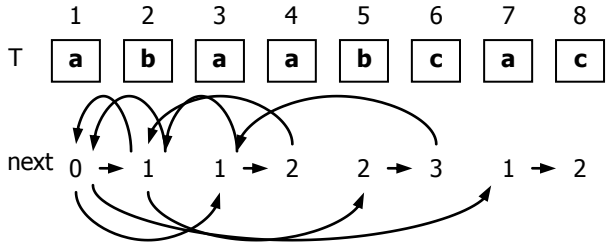
D．(0,1,0,1,0,1,0,1,1)

【解析】注意到四个选项中第6个字符的nextval[]值分别为4、2、0、1，各不相同，因此直接计算nextval[6]即可作答。不妨记模式串为T，观察第6个字符之前的子串“ababa”，开头与末尾最多各取3个字符组成的子串相等，都是“aba”，因此next[6]=4。若要求得nextval[6]的值，还要考察T[6]与T[4]。因为T[6]=’a’≠T[4]=’b’，故不需要修正，即nextval[6]=4。正确选项为A。

3．模式串P=‘abaabcac’的next函数值序列为_____。

【解析】本题计算给定模式串的next函数值，可采用观察首尾最大相等子串的方法计算，也可以根据get_next()算法计算，下面分别说明。观察法可以直接计算任意位置上字符的next值。比如对于P[3]，观察前面的子串“ab”，找不到首尾最大相等子串，故next[3]=1；又如P[4]，观察前面子串“aba”，可以在首尾分别找到长度为1且相等的子串“a”，故next[4]=2；再如P[6]，观察前面的子串“abaab”，可以分别在首尾截取长度为2的子串且两者相等，故next[6]=3。最终得到结果01122312。

若按照get_next()算法，执行过程如下图，结果是0112312。



4．字符串’abababaaab’的 nextval 函数值为_____。

【解析】先计算next[]值为0112345612，再修正为nextval[]值为0101010601。