KMP算法用于字符串之间的比较，典型的应用是在一个字符串中查找另一个字符串是否为其子串。

假设目标字符串：

待匹配字符串：

一般的比较算法：

当发生失配时：，从0开始，然后与比较，这样做效率比较低。算法的时间复杂度为

KMP算法：

当发生失配时：，跳转到，然后比较与

数组



其中，是一个最优值，当失配跳转时，要从它开始。

含义：若，取的模式值

1. ，表示与不相等，下次比较与
2. ，表示下一次比较与
3. ，表示之前的个字符与开始的的字符是相同的，下次比较与。

KMP算法的思想：

若发生失配，则说明肯定与之前的个字符是匹配的。而对应的模式值为，则说明，而且是一个最优值。因为之前的前个字符肯定与相匹配，所以其与也肯定匹配，因而下次比较时，直接比较与。

如何求数组

，设，则：



有两种情况：

1. 

则，

1. 

则比较与，如果，则，否则继续按照数组跳转，直到匹配，若最终未匹配，则

这样比较的理由：

类似于之前目标数组和待匹配数组之间出现失配情况。如果，则：

首先：参考数组定义，



其次：

因为，所以



从跳转是最佳的选择。

例：程序KMP\_test1

void GetNext(const char\* substr,int\* next)

{

// 求模式串T的next函数值并存入数组next。

int j = 0;

int k = -1;

next[0] = -1; // 初始条件

int len = strlen(substr);

while (j < len - 1)

{

// ||运算符前面的条件满足，不会再计算后面的条件

if (k == -1 || substr[j] == substr[k])

{

++k;

++j;

next[j] = k; // 对应算法中的P[j]=P[k]

}

else

{

k = next[k]; // 对应算法中的P[j] != P[k]

}

}

}

int KMP(const char\* src, const char\* substr)

{

if ((src == NULL) || (substr == NULL))

{

return -1;

}

int\* next = NULL;

int substr\_length = strlen(substr);

int src\_length = strlen(src);

if (src\_length < substr\_length)

{

return -1;

}

next = new int[substr\_length];

GetNext(substr, next);

int i = 0;

int j = 0;

while ((i != src\_length) && (j != substr\_length))

{

// next[0] = -1，所以这里要判断j是否为-1

if ((j == -1) || (src[i] == substr[j]))

{

++i;

++j;

}

else {

j = next[j]; // 失配状况发生

}

}

if (j == substr\_length)

{

return (i - j);

}

return -1;

}

GetNext算法的时间复杂度为，其中为的长度，KMP函数之后的while循环的时间复杂度为，其中为的长度，所以KMP算法的时间复杂度为