1. KMP算法是一种线性时间复杂度的字符串匹配算法，它是对BF算法(Brute-Force，最基本的字符串匹配算法)的改进。对于给定的原始字符串string和模式字符串p，需要从字符串s中找出p(首次)出现的位置索引：

BF算法的时间复杂度为O(strlen(s)\*strlen(p))，空间复杂度为O(1)；

KMP算法的时间复杂度为O(strlen(s)+strlen(p))，空间复杂度为O(strlen(p))；

1. 两种算法的主要区别是失配时的处理：假设串s匹配到i位置，串p匹配到j位置：

BF算法中，如果当前字符匹配成功，即s[i+j]==p [j]，则令j++，继续匹配下一个字符；如果匹配失败，即s [i+j]!=p [j]，则让i++并且j=0，即每次失配时，原串匹配位置向右移动一位(从i+j回溯到i+1)，而j重置为0。

而KMP算法则是在发生失配时，根据模式中字符和失配在模式中出现的位置，来确定继续进行搜索(j)的位置,而i的位置不会向后回溯。

1. 例如：假定p=’abcabcacab’,令字符串s=,且假设现在要判断是否存在从 开始的匹配。
2. 如果,那么显然可以进行与a的比较;
3. 类似地，如果，那么进行与a的比较；
4. 如果，则出现如下情况：

s=

p=’abcabcacab’

符号?表示不知道s中此处字符是什么，在s中第一个?代表，因此，搜索的下一步应该是对p的首字符和进行比较。这里无需比较p的首字符和，因为已经知道与p的第二个字符相同为b，即；

1. 以此类推，假定出现了p前4个字母的匹配，然后失配，即。则出现如下情况：

s=

p=’abcabcacab’

通过观察可以看到，匹配搜索可以继续把与第二个字符b进行比较。这是在把模式p向右滑动时，发生部分匹配的第一处。因此，通过模式中的字符和失配在模式中出现的位置，就可以确定在模式中继续进行匹配搜索的位置，而不必在s中进行回溯。

1. 为了形式化说明，定义一个模式失配函数：

令p=是一个模式，则其失配函数f定义为：

例如: 对于模式p=’abcabcacab’,有：

j 0 1 2 3 4 5 6 7 8 9

p a b c a b c a c a b

f -1 -1 -1 0 1 2 3 -1 0 1

根据失配函数的定义，得到如下模式匹配规则：如果出现了形如的部分匹配，，若j,则下一趟模式匹配时，从失配字符和模式串字符处重新开始比较；若j=0，则继续比较和。

按上述匹配规则实现的匹配函数为：

int str\_pmatch(char \*s, char \*p, char\* failure)

{

if (NULL == s || NULL == p || NULL == failure)

{

return -1;

}

int lens = strlen(s);

int lenp = strlen(p);

int i = 0, j = 0;

while (i < lens && j < lenp)

{

if (s[i] == p[j])

{

i++;

j++;

}

else if (j == 0)

{

i++;

}

else

{

j = failure[j - 1] + 1;

}

}

free(failure);

return ((j == lenp) ? (i - lenp) : -1);

}

易知str\_pmatch的时间复杂度为O(strlen(s)),如果能在strlen(p)的时间复杂度内求出失配函数failure，则算法的整个时间复杂度为O(strlen(s) + strlen(p))。

1. 要想在O(strlen(p))时间内计算出失配函数，需要利用失配函数的令一种表达形式：

注意：，

(这个表达形式如何证明？？)

和以上定义对应的失配函数计算方法如下：

void str\_fail(char \*pat, char\* failure)

{

int n = strlen(pat);

failure[0] = -1;

for (int j = 1; j < n; j++)

{

int i = failure[j - 1];

while (pat[j] != pat[i + 1] && i >= 0)

{

i = failure[i];

}

if (pat[j] == pat[i + 1])

failure[j] = i + 1;

else

failure[j] = -1;

}

}