串

一、引言

字符串作为编程语言中表示文本的数据类型，想必大家都不陌生。C中的`char[]`，以及C++中封装性更好的`string`类，都是字符串的具体实现。

字符串依然是一种线性表，只不过表中元素类型具体化为字符型。相比于其他线性表，字符串更贴近生活、应用更广泛、操作也更特殊。具体来说，对于我们生活中大量的自然语言文本，字符串是最直接、常用的实现方法。字符串的操作也通常以子串为单位进行。

二、基本概念

字符串是有限长度的字符序列。长度为n的字符串可表示为：

$$S ='' a_0a_1a_2\dots a_{n-1}''$$

术语

- 子串：

$$S.substr(i,k)=S[i,i+k),0\le i<n,i\le k,i+k\le n$$

- 前缀（前k个字符构成的子串）：

$$S.prefix(k)=S.substr(0,k)=S[0,k),0\le k\le n$$

- 后缀（后k个字符构成的子串）：

$$S.suffix(k)=S.substr(n-k,k)=S[n-k,n),0<k\le n$$

存储表示

字符串既然作为一种线性结构，依旧可分为**顺序存储**和**链式存储。**

顺序存储可分为**定长存储**和**变长存储。**定长存储为每个串预先分配固定规模的存储区域，空间利用率低，且有数据截断风险；变长存储（也就是我们目前最常见到的存储表示方式）通过在字符末尾额外增加**“\0”**表示字符的结束（需要占一个字符的空间），可以根据实际需要分配存储区域的规模。

链式存储亦称块链存储，链表节点一般不是单个字符，而是顺序存储的子串块。

基本操作

针对串常进行的基本操作有：创建，销毁，复制，清空，判空，比较，求长，串接，截取子串，子串定位（串匹配），子串删除，插入，关键词替换等。

其中，复制、求长、串接、比较、子串定位在`<string.h>`中有对应函数：`strcpy()`、`strlen()`、`strcat()`、`strcmp()`、`strstr()`.

便于封装和说明，我们基于`char[]`定义以下`String`类进行后续讲解。

```
class String
{
private:
    char a[DEFAULT_SIZE];
public:
    //构造一个空串
```

```
    String() { a[0]='\0'; }
    //复制构造
    String(const char* S);
    String(String& S);
    //销毁
    ~String() {};
    //显示
    void ShowString()const;
    //复制
    void StrCopy(String& S);
    //清空
    void ClearString();
    //判空
    bool StrEmpty()const;
    //比较
    int StrCompare(String& S)const;
    //求长
    int StrLength()const;
    //串接
    void Concat(String& S);                     //将串S串接本串后
    void Concat(String& S1, String& S2);        //将S1，S2串接后的新串存至本串
    //反转
    void Reverse();
    //截取字串
    void SubString(String& Sub, int pos, int len);
    //子串定位
    int Index(String& P,int pos)const;
    //插入
    void StrInsert(int pos, String& T);
    //删除子串
    void StrDelete(int pos, int len);
    //关键词替换
    void Replace(String& T, String& V);

    //模式串匹配
    //……

    //重载下标运算符
    char& operator[](int n);
}
```

操作的具体实现见codes文件夹。

三、字符串的模式匹配和简单匹配算法

字符串模式匹配是计算机科学中非常重要的数据处理技术。在现实生活中，字符串模式匹配也有着广泛的应用。如：

1. 文本搜索：字符串模式匹配用于实现现代搜索引擎中的文本搜索，用来查找网页、文档等中包含指定关键字的内容。
2. 数据库查询：数据库系统使用字符串模式匹配来查找满足特定条件的记录。

3. 文件查找：当我们在计算机上查找文件时，文件名的匹配通常是通过字符串模式匹配来实现的。
4. 信用卡和身份证号码等标识号码的验证和识别，使用一些字符串模式匹配算法，包括正则表达式。
5. 生物信息学：在基因组学和蛋白质组学中，字符串模式匹配用于 DNA，RNA 和蛋白质序列的比对和匹配，以识别相似序列、同源序列等重要信息。

字符串模式匹配：已知文本串 `T(Text string）` 和模式串 `P(Patten string)`，需在 `T` 中找出 `P` 第一次出现的位置，匹配成功则返回串 `P` 在 `T` 中的位置（即 `P` 的第一个字符在 `T` 中出现的位置），匹配失败返回-1。

我们可以设

$$n = |T|, m = |P|$$

对于一般问题，有

$$2 << m << n$$

模式串匹配可有多种算法：**蛮力法（`BruteForce`）**，`KMP` **算法**，`Horsepool` **算法**，`BM` **算法**，`Sunday` 算法，`KR` 算法等。我们

仅介绍前4种，其余请感兴趣的同学自行查阅相关资料。

蛮力法（BruteForce）

基本思路

穷举，将 `T` 的所有位置都作为比较可能的起始位置，与 `P` 依次进行比较。即需要依次检查 `T` 的每一个长度为 `m` 的子串是否等于 `P`。

实现

```
int BruteForceMatch(String& T, String& P)
{
    int n = T.StrLength();
    int m = P.StrLength();
    if (n < m || m == 0 || n == 0) return -1;    //若模式串长度大于目标串，或其一为空串，匹配必然失败
    int i = 0, j = 0;
    while (j < m && i < n - m + 1) {                    //自左向右逐次比对
        if (T[i] == P[j]) { i++; j++; }         //若匹配，转到下一字符串
        else { i -= j - 1; j = 0; }             //失配，则T回退、P复位
        if (j == m) return i - j;               //匹配成功，返回对齐位置
    }
    return -1;                                  //匹配失败
}
```

复杂度分析

- 时间复杂度

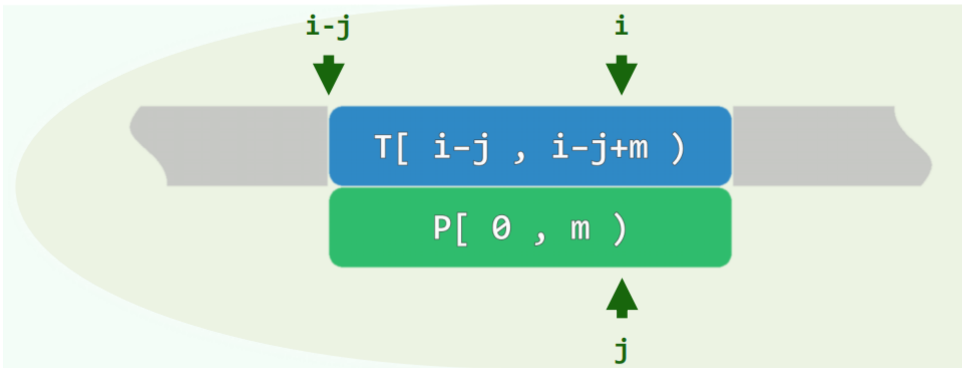
  最好情况：每次匹配时，`P` 的第一个字符就与 `T` 不匹配，这样仅需 `i` 指针移动，最多移动 `n-m~n` 次。

最坏情况：每次模式串的前`m-1`个字符都与目标串匹配，但最后一个字符不匹配，这样`j`指针移动的次数达到最大，移动`（n-m）m~nm`次。

则蛮力法的时间复杂度为

$$Best : \Omega(n), Worst : O(n * m)$$

注：这里的`i`，`j`指针指当前需要进行比较的`T`和`P`的下标。如图（截取自邓俊辉老师课件），可用指针移动形象地描述`i`和`j`的变化。**便于交流，后续我们用`T[i]`,`P[j]`特指当前`T`和`P`待匹配的位置。**



- 空间复杂度：无需额外开销。

总结

显然，蛮力法忽略了在比对过程中可以利用的一些信息，导致`i`指针每次都需要回退，比较低效。

KMP算法

如果使用蛮力法，如果`T[i]`与`P[j]`失配，我们会很沮丧，毕竟前面匹配得再好也前功尽弃。不过对于善于从失败中总结经验的人，或许还不算太坏。既然失配点前面的部分我们已经比较过了，那么意味着这部分信息我们我们已经获取。能不能加以利用呢?

基本思路

若`T[i]`与`P[j]`失配，则至少有`T[i - j, i)`与`P[0, j)`是匹配的，亦即我们其实已掌握`T[i - j, i)`的所有信息，从而知道哪些位置不值得对齐。从而下次匹配时，我们可以令`i`指针直接跳过这些不可能的位置。

如此，`i`**指针永远不必回退**。比对成功，`i`与`j`同步前进一个字符；若失配，`j`更新为更小的`j'`。这样就避免了重复比对。

如何确定`j'`？如果`j'`仅与`P`有关，我们可以进行一个一劳永逸的预处理，即根据每个位置失配时`P`可向前移动的最大距离，制作查询表`next[]`。

实现方式1（依据课本）

查询表定义为`j'=next[j-1]`。

主函数

```
int KMP_Match(String& T, String& P)
{
    int n = P.StrLength(), m = P.StrLength();
    int* next = new int[m];
    Next(P, next);                              //预处理，我们先假定next[]已有
    int i = 0, j = 0;
    while (i < n) {                             //已匹配j+1个字符
        if (T[i] == P[j]) {
            if (j == m - 1) return i - j;       //匹配成功，返回匹配位置
            else { i++; j++; }                  //比较下一位置
        }
        else {
            if (j > 0) j = next[j - 1];         //失配，j更新为j'
            else i++;                           //如果第一个位置（j==0）就失配了，那么i直接后
移一位即可
        }
    }
    return -1;                                  //匹配失败
 }
```

查询表

具体如何构造，首先要了解最大自匹配的问题。

如果`T[i]`和`P[j]`失配(`j>0`)，那么之前比较过的子串是相等的，即

$$T[i-j,\ i-1]=P[0,j-1]$$

设`j`更新后的位置为`j'`，显然`j'<j`,从上式左右两边相等的子串中，再分别截取长度为`j'`的后缀，仍然相等

$$T[i-j',i-1]=P[j-j',j-1]$$

同时`j`更新后，令`j'`与`i`对齐，该位置前的部分也应该保证是相等的

$$T[i-j',\ i-1]=P[0,j'-1]$$

那么等量代换就有

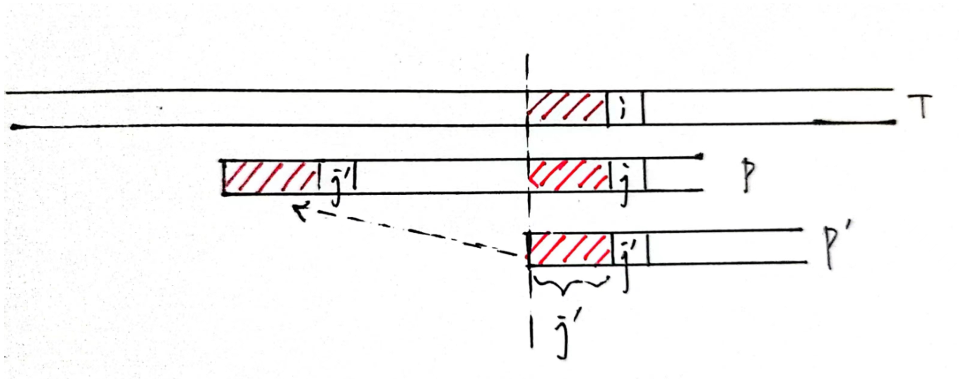
$$P[0,j'-1]=P[j-j',j-1]$$

也即对于`P[0,j)=P.prefix(j)`,其长度为`j'`的前后缀相等，即

$$P[0,j).prefix(j')=P[0,j).suffix(j')$$

注：注意由于`j'<j`，**后缀是不可以包含`P[0]`的。**

直接看图，更加直观：



我们称**最长**（也即`j'`**最大**）的一对这样相等的前后缀为**部分匹配串**。因此要确定`j'`，就等同于确定`P[0,j)`部分匹配串的长度。

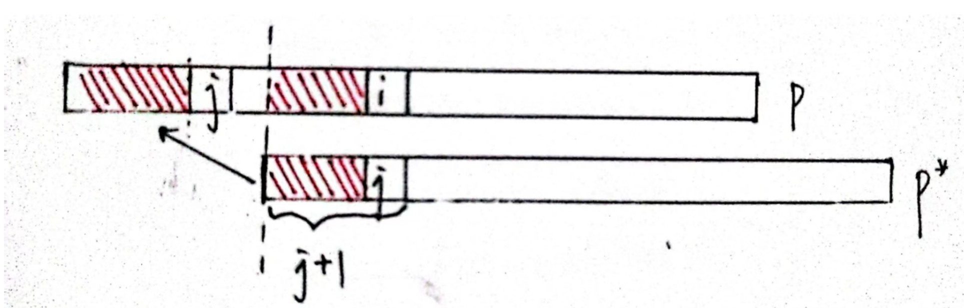
我们诚然希望`P`相对于`T`向前移动更多，意味着`j'`越小越好。但为何这里要求`j'`最大？这是因为如果存在更大的`j'`满足条件，而我选择了较小的`j'`使得移动距离更大，就会**错过可能匹配的子串**。也就是移动在有意义的前提下，要尽量小步。

构造查询表：

```
void Next(String&P, int next[])
{
    int m = P.StrLength();
    next[0] = 0;                          //起始条件
    int i = 1, j = 0;                     //错一位
    while (i < m) {
        if (P[i] == P[j]) {               //此时已匹配j+1个字符
            next[i] = j + 1;              //部分匹配串长度加一
            i++; j++;                     //比较位置各进一
        }
        else if (j > 0) j = next[j - 1];      //移动：用部分匹配串对齐
        else { next[i++] = 0; }               //j在串头时部分匹配串长度为0
    }
}
```

查询表的构造原理与主函数的思路有些相似。本质上是我们复制一份`P`与自己做匹配。我们的目标是随着`i`指针的移动计算`next[i]`。

当`P[i]`与`P[j]`相等时（`j<i`）,我们知道`P[0,i]=P[0,i+1)`的部分匹配串长度为`j+1`，如图。



如果`P[i]`与`P[j]`失配，与主函数相同地，我们更新`j'=next[j-1]`(`next[j-1]`已算过，`j>0`)。这实际是递推的思路。作为起始，自然地，`next[0]=0`，如果你认为`next[0]`应当等于1，不要忘记后缀不可包括P[0]。如果串头就失配（`j==0`），显然此时的位置`i`没有部分匹配串。

实现方式2（更常用）

基本思路和实现方法一致。只是在`next[]`的定义上有些差别。

之前我们定义`j'=next[j-1]`，似乎有些反直觉，为何不直接定义`j'=next[j]`？

其实也可以，构造方式也类似，只是初值`next[0]`需要更改。自然地，`next[0]=-1`，这是因为如果`j==0`时就失配，那`P`串只能再向后移一格，相当于`j`指针指向**-1**的位置（记得吗，我们之前不是让`j=-1`，而是让`j`为0不变，`i+1`）。当然，后续不能直接用-1做下标，我们需要判断`j==-1`的情况，如果是，则`i,j`分别+1（最后仍是实现`j=0`,`i=i+1`,没有本质区别）。好处是我们不用担心`next[j]`的下标会出现负数。（原先next`[j-1]`需要考虑`j==0`的情况）。

这样的话，实际上`next[1]=0`也是确定的（相当于实现方式1中`next[0]=0`）。

按此`next[]`定义调整后的代码：

主函数

```
int KMP_Match(String& T, String& P)
{
    int n = P.StrLength(), m = P.StrLength();
    int* next = new int[m];
    Next(P, next);
    int i = 0, j = 0;
    while (i < n) {
        if (j < 0) { i++; j++; }             //增加j==-1的判断
        else if (T[i] == P[j]) {
            if (j == m - 1) return i - j;
            else { i++; j++; }
        }
        else j = next[j];                    //next[]定义改变，且无需再对j>0和j==0进行讨论
    }
    return -1;
}
```

查询表

```
void Next(String& P, int next[])
{
    int m = P.StrLength();
    next[0] = -1, next[1] = 0;              //起始条件改变
    int i = 1, j = 0;
    while (i < m - 1) {
        if (P[i] == P[j]) next[++i] = ++j;
        else j = next[j];
    }
}
```

优化

我们考虑最坏的情况，比如

$$T ='' 000000000000000000001''$$
$$P ='' 000001''$$

我们发现`P`串和`T`串每次匹配到最后一个字符时才失配，而每次失配`P`串相对于`T`串只能移动一位。

我们发现`KMP`一个很naive的缺陷在于，我们费劲周折地利用了`T[i-j，i）`的信息，却从没考虑过`T[i]`本身。也即`P`移动后，我们甚至无法保证原失配点不会再次失配。

其实，**我们不仅可以利用`T[i-j，i）`是什么，还可以利用`T[i]`不是什么。**

优化的基本原理就是，**在需要`j`指针回溯到`j'`时，将`P[j]`与`P[j']`比较，如果相等，那么就没有必要再进行比较了，`j`继续回溯。如此往复。**

由此，我们可以这样构造`next[]`表（基于实现方式1）

```
void Next_Pro(String&P, int n[])
{
    int m = P.StrLength();
    n[0] = 0;
    int i = 1, j = 0;
    while (i < m) {
        if (P[i] == P[j]) {
            if (P[i] == P[n[i - 1]]) n[i] = n[i - 1];  //如果当前位置元素与回溯之后位置元素相等，继续回溯
            else n[i] = j + 1;
            i++; j++;
        }
        else if (j > 0) j = n[j - 1];
        else { n[i++] = 0; }
    }
}
```

不妨针对上述最坏的情况，直观感受一下优化效果：

优化前:

```
next[]: 0 1 2 3 4 0
循环次数: 36
匹配点: 15
```

优化后：

```
next[]: 0 0 0 0 0 0
循环次数: 24
匹配点: 15
```

都能得到正确的结果，但显然后者的`next[]`更合理，跳步更快。

复杂度分析

- 时间复杂度

主函数的循环，或`i`加一,或`P`的位移量`i-j`至少加一。考虑最坏的情况下（一直不匹配且循环次数最多）何时退出循环，或`i>n-1`，或P的位移量`i-j>n-m`，意味着循环不超过`2n`次。同理构造`next[]`表循环不超过`2m`次。故`KMP`算法的时间复杂度为

$$O(m+n)$$

- 空间复杂度

`next[]`表的开销，复杂度为`O（m）`。

总结

`KMP`充分利用以往比对所提供的信息，使得`T`**无需回退，**`P`**快速右移。**

经验：**利用了**`T[i-j，i）`**是什么，优化后的版本还可以利用**`T[i]`**不是什么。**

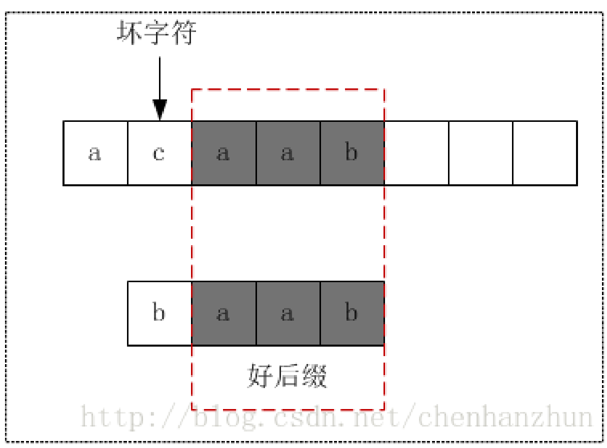
教训：**仍然无法利用**`T[i]`**是什么。**

优势：**单次匹配概率越大，即字符集越小，优势越明显**（可对比`Horspool`，如果字符集较小，同一字符在`T`、`P`中出现频率很高，`Horsepool`每次的位移量会很小而失去优势，而`KMP`不受限制）。否则，`KMP`的速度提升较蛮力法也并不显著。

BM算法

`BM(Boyer+Moore)`算法实际是比`KMP`更加高效且常用的的算法。原因是`BM`同时采用**坏字符准则**与**好后缀准则**，分别利用了**当前失配点的信息**和**之前比对的信息。**

所谓**坏字符，就是与当前P[j]不匹配的T[i]**；所谓**好后缀，就是指在遇到坏字符之前，**`T`**与**`P`**已匹配成功的字符子串**。如图：

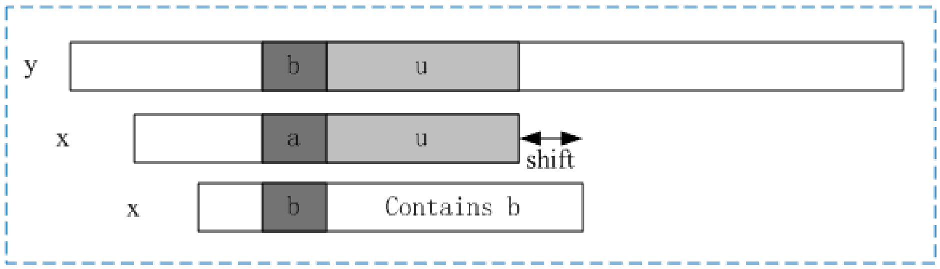


坏字符准则（Horspool）

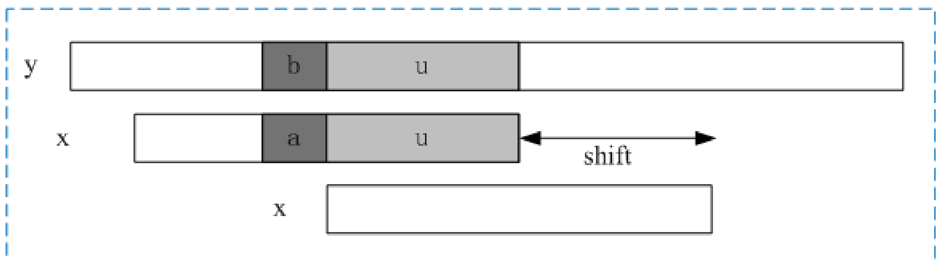
坏字符准则单拿出来，稍作处理，其实就是课上讲过的`Horspool`算法，dll给它起了个生动形象的名字：“小马过河”算法。

思路

前面反复提到`KMP`算法的缺点：没有考虑失配点`T[i]`的信息。其实更直观的想法是如果`T[i]==a`和`P[j]==b`失配了，我至少应该保证`P`移动后的`P[j']`能与`T[i]`相配。**也就是拿`T`中不匹配的字符，在`P`中找到相同的字符，对齐。**



Case1:模式串存在坏字符b，则模式串的字符b与
文本串的坏字符b对齐



Case2:模式串不存在坏字符b，则移动模式串到
文本串的坏字符b的下一个位置

另外需要提的是`BM`是反向匹配的。实际上正向匹配和逆向匹配没有本质的逻辑上的优劣，但对于一般情况下，字符集较大，单次匹配概率较小（也就是常常匹配不了几位就失配了）。如果正向匹配，我们在往往`j`比较小的位置就失配了，这时可移动的距离`d<=j+1`也会受限制，每次移动的距离不多；而反向匹配的话，我们在`j`比较大的位置失配，如果坏字符出现频率低，大概率在`P`中稀疏分布甚至不出现，使得`j'`较小甚至`j'==-1`（直接右移整个`P`），这样就可以移动较大的距离。因此在实际中，反向匹配显然更快。

为了确定`j'`，与我们需要建立`bc[]`，`j'=bc[T[i]]`。出现特定坏字符`T[i]==a`时，我们需要找到`P[j']==a`去对齐（找不到就令`j'=-1`）。为了保证不遗漏，`P`应小步移动，我们需要找到一个尽可能大的`j'`，也即**找到坏字符在`P`中最后一次出现的位置**。

但仍有问题，**如果`j'>j`，`P`就会反向移动**！我们或许还可以约束`j'<j`，但这是`j'`的确定既和坏字符`T[i]`有关，又和当前`j`有关，比较复杂。利用好后缀准则可以解决这个问题。如果不想那么复杂，我们不妨依然找最大的`j'`，但**若出现`j'>j`，我们令`P`强制右移一格避免回退**，就是`Horspool`的做法。

实现

构造bc[]

由于坏字符可能是字符集中任一字符，我们需要对字符集中每一字符计算其在`P`中最后出现的位置，制备`bc[]`(用哈希表等方式实现均可)。

比如假定字符集为ASCII码表0~255，可有如下实现：

```
void Build_BC(String& P, int* bc) {
    int m = P.StrLength();
    for (int i = 0; i < 256; i++) {
        bc[i] = -1;
    }
    for (int i = 0; i < m; i++) {
        bc[int(P[i])] = i;
    }
}
```

Horspool

```
int Horspool(String& T, String& P) {
    int n = T.StrLength();
    int m = P.StrLength();
    int bc[256];
    Build_BC(P, bc);
    int pos = 0;                              //pos代表此时P[0]与T对应的位置，真正的待
匹配位置i = pos + j
    while (pos <= n - m) {
        int j = m - 1;
        while (j >= 0 && P[j] == T[pos + j]) --j;
        if (j == -1) return pos;              // 匹配成功
        pos += max(1, j - bc[T[pos + j]]);    // 当有坏字符时，移动模式串（若j'>j,强制
移动一位）
    }
    return -1;                                // 未匹配到
}
```

复杂度分析

- 时间复杂度

  最好情况：每次`j==m-1`就失配，坏字符不在`P`中，每次跳跃距离为m。

  最坏情况：每次j==0才失配，每次只能移动一位。

  可知`Horsepool`的时间复杂度（仅考虑查找）为

$$Best : \Omega(n/m), Worst : O(n*m)$$

  若考虑预处理，还需在加上字符集规模`|S|`。

- 空间复杂度：`bc[]`表的开销，复杂度为`O（|S|）`。

小结

一般情况下，`Horsepool`已经可以表现出相当不错的查找效率（除法量级）。但在某些特殊情况下，效率可能退化为乘法量级，和蛮力法无异。

如，考虑如下情况：

$$T ='' 000000000000000''$$
$$P ='' 10000''$$

`Horspool`就显得举步维艰。目前，我们还只利用了失配点的信息。有了`KMP`的经验，我们还可以对之前匹配得到的信息加以利用，故而有了好后缀准则。

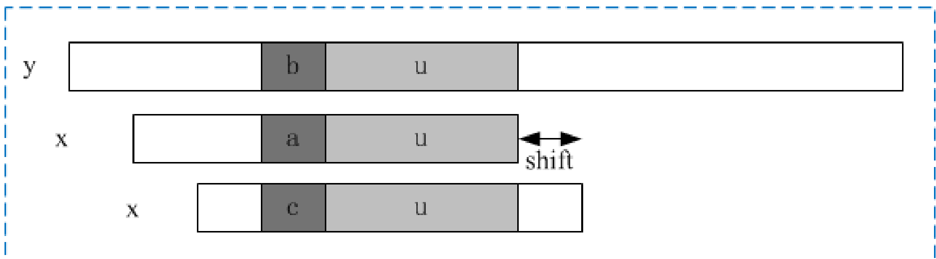
好后缀准则

思路

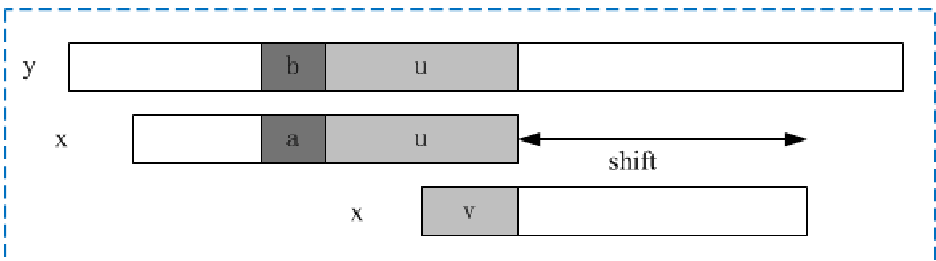
和`KMP`的思路异曲同工。

若`T`和`P`已经匹配了一个好后缀`u`，如果下一个字符是坏字符，则须要移动`P`又再一次匹配。

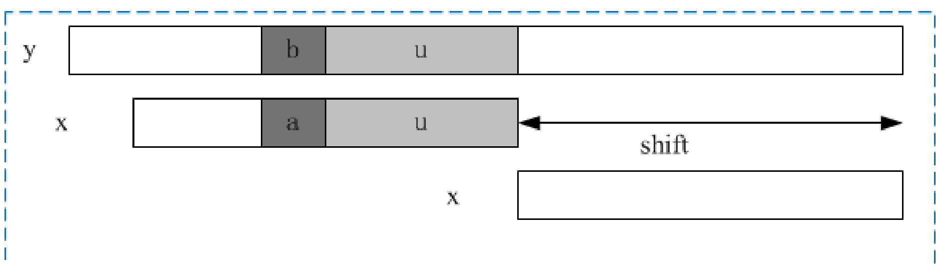
若在`P`中依然存在与好后缀完全或部分匹配的子串`v`，则将`v`与`T`中对应部分对齐。若`P`中再无这样的`v`。则直接右移整个`P`。如图



Case1:模式串存在与好后缀完全匹配的子串



Case2:模式串存在与好后缀部分匹配的子串

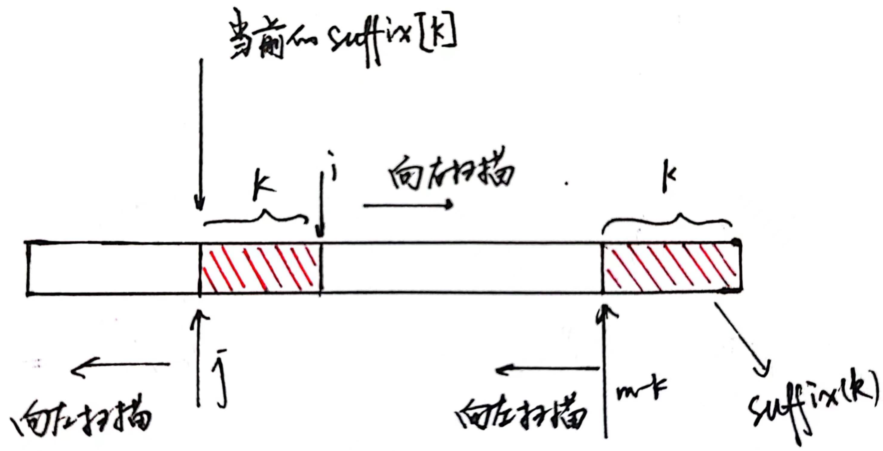


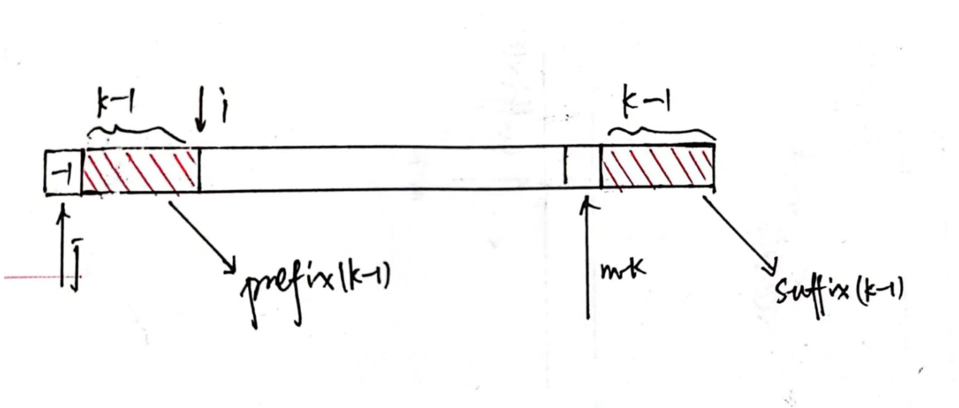
Case3:模式串不存在与好后缀匹配的子串

实现

构造gs[]

我们令`gs[k]=suffix[k]`记录`P.suffix(k)`在`P`中除末尾外最后出现的位置q，若`P.suffix（k）`在`P`中不再出现则`suffix[k]=-1`。另设`prefix[]`，如果某`suffix(k)`不仅是`P`的后缀，还是`P`的前缀，则令`prefix[k]=true`，否则为`false`。

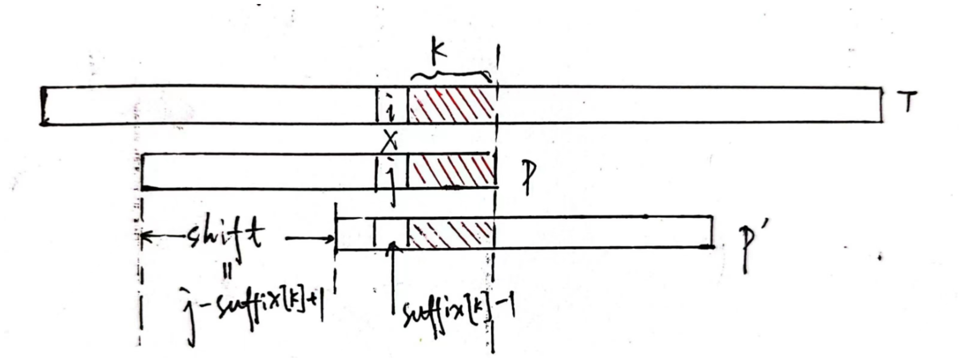




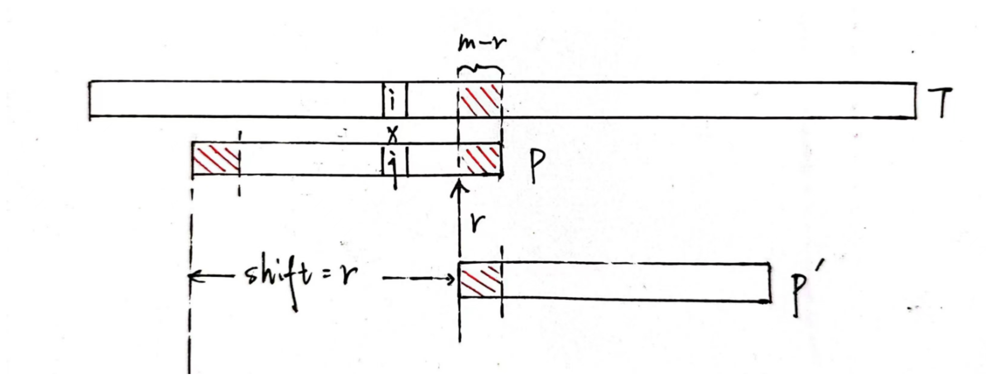

```
void Build_GS(String& P, int* suffix, bool* prefix) {
    int m = P.StrLength();
    for (int i = 0; i < m; i++) {          // 初始化suffix和prefix
        suffix[i] = -1;                    //suffix(i)默认为-1，表示该后缀在前面不再出现
        prefix[i] = false;
    }
    for (int i = 0; i < m - 1; i++) {      //i作为起始位置向后扫描
        int j = i, k = 1;                  //k表示后缀长度
        while (j >= 0 && P[j] == P[m - k]) {
            suffix[k] = j;                 //P.suffix(k)会又一次出现在j位置,随着i向右扫描，可能更新
            --j; ++k;                      //j和m-k同步向前扫描
        }
        if (j == -1)  prefix[k - 1] = true;    //检测到前缀
    }
}
```

计算移动量

`case1:` 存在与好后缀完全匹配的子串:



`case2:` 存在与好后缀部分匹配的子串:




```
int move(int* suffix, bool* prefix, int j, int m) {
    int k = m - 1 - j;                        // 好后缀的长度
    if (suffix[k] != -1) {                    // 如果存在与好后缀完全匹配的子串
        return j - suffix[k] + 1;,
    }
    for (int r = j + 2; r <= m - 1; r++) {   // 如果仅存在与好后缀部分匹配的子串
        if (prefix[m - r]) {
            return r;
        }
    }
    return m;                                 // 没有好后缀可以匹配，则移动整个模式串长度
}
```

用之前的例子直观感受以下优化后的效果:

$$T ='' \ 000000000000000''$$
$$P ='' \ 10000''$$

只用坏字符准则，甚至倒退。即使采用`Horspool`，每次移动量为1。

添加好后缀准则后，每次可以匹配到好后缀"0000"，而`gs[4]==-1`，没有好后缀可以匹配，移动量为5，移动加快了。

综合

将两种准则结合起来，每次需要移动时，我们**分别计算两种准则的位移量，取大者即可。**

总函数

```
int BM_Match(String& T, String& P) {
    int n = T.StrLength();
    int m = P.StrLength();

    int bc[256];
    bool* prefix = new bool[m];
    int* gs = new int[m];
    Build_BC(P, bc);
    Build_GS(P, gs, prefix);

    int i = 0;
    while (i <= n - m) {
        int j = m - 1;
        while (j >= 0 && P[j] == T[i + j]) {
            --j;
        }
        if (j == -1) {
            return i;
        }
        int x = j - bc[int(T[i + j])];      //x:采用坏字符准则的移动量
        int y = 0;                          //y:采用好后缀准则的移动量
        if (j < m - 1) {
            y = move(gs, prefix, j, m);
        }
        i += max(x, y);
    }
    return -1;
}
```

复杂度分析

- 时间复杂度

  最好情况：继承`Horspool`的优势，达到除法量级。

  最坏情况：由于添加了好后缀原则，类似`KMP`，由原来的乘法量级优化至加法量级。

  `BM`的查找效率为

  $$Best: \Omega(n/m), Worst: O(n+m)$$

  预处理的复杂度

  $$O(|bc|+|gs|) = O(|S|+m)$$

- 空间复杂度

`bc[]`和`gs[]`的开销，即`O（|S| + m）`。

小结

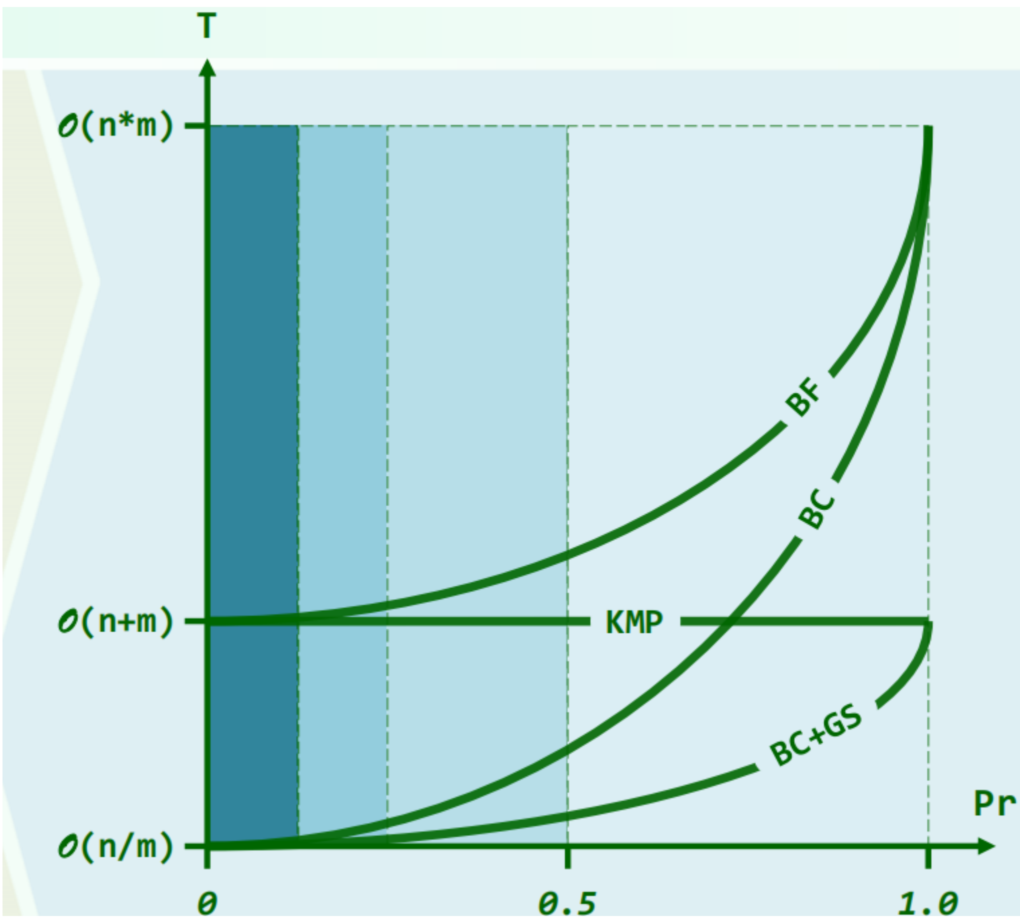
`BM`最大程度地利用了已知信息，使得实际查找效率比`KMP`还要快3-5倍。

字符集大，单次匹配概率小时，主要采用坏字符准则，速度优势更明显。反之，坏字符准则优势丧失，好后缀准则提高下限，使得查找效率不至于太低。

总结

从`BF`的无记忆，到`KMP`利用已匹配部分的信息，到`Horspool`利用当前失配点信息，再到最后的集大成者`BM`综合了`KMP`和`Horspool`的优势。我们一步步挖掘已知信息，一边总结经验，一边吸取教训，逐步优化。

对于各种匹配方式，其查找效率总结如下（*截取自邓俊辉老师课件*）：



注：**横坐标`Pr`表示单次匹配成功的概率**。假设字符集中的字符出现的概率相等，则`Pr=1/|S|`。可大致认为`Pr`

和`|S|`反相关。