

数据结构与算法

DATA STRUCTURE

第九讲 字符串匹配算法（2）

胡浩栋

信息管理与工程学院

2017 - 2018 第一学期

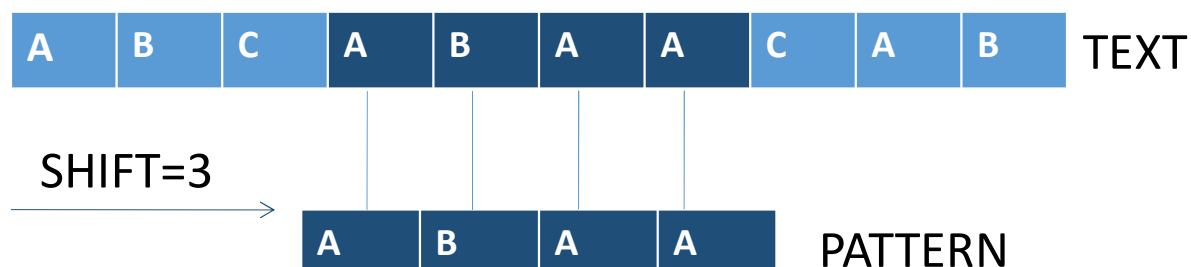
课堂内容

- 字符串匹配算法二

STRING MATCHING ALGORITHMS



例子



- 模式匹配过程中，程序会查看`text`中长度为`M`的窗口，即用`pattern`串和`text`的窗口中的子串进行比对。比对完成后，将窗口向右滑动，并不断重复这一过程。直到根据需要找到所需匹配为止。这种机制被称为滑动窗口机制。
- 本节所讨论的若干串模式匹配算法都是基于滑动窗口机制的算法。

串的精确匹配算法

- 1) 暴力的匹配算法(滑动一位)
- 2) Krap-Rabin 算法(滑动一位)
- 3) The Knuth-Morris-Pratt 算法(滑动多位)
- 4) **BM** 算法(滑动多位)
- 5) 后续优化

Boyer-Moore算法

BM(Boyer-Moore)算法

BM算法与KMP算法的主要差别在于：

- 在进行匹配比较时，不是自左向右进行，而是自右向左进行；
- 预先计算出`text`中可能的字符在`pattern`中出现的位置信息，利用这些信息来减少比较次数。
- KMP算法用`pattern`前缀计算可移动长度（从左到右）
- BM 算法用`pattern`后缀计算可移动长度（从右到左）

从右向左扫描的暴力算法

- 从右向左扫描
- 逐一滑动窗口
- **BM**算法做的改进全在黄色一行

```
int MyString::FindRightLeft(const MyString & pattern) const
{
    int M = pattern.Length();
    if (M < 1)
    {
        return 0;
    }

    /* Searching with right-to-left matching */
    for (int j = 0; j <= Length() - M;)
    {
        int last = M - 1;
        while (pattern[last] == _pszData[last + j])
        {
            last--;
            if (last < 0)
            {
                return j;
            }
        }

        // BM algorithm will move j further
        j++;
    }

    return -1;
}
```


BM算法思想

为了右移更大的距离，**BM**算法使用两个预计算函数来指导窗口向右移动的距离

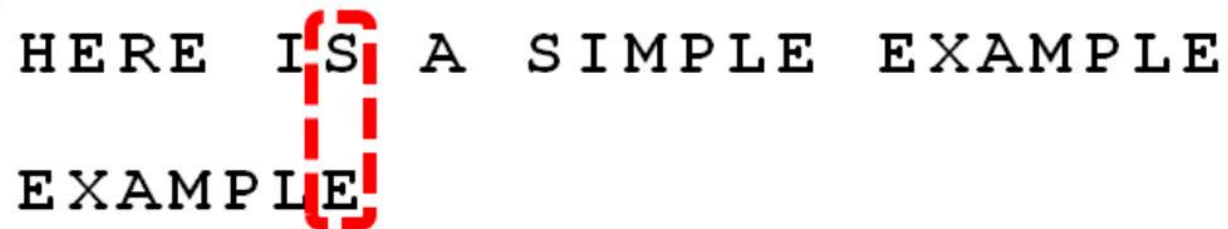
- 坏字符规则 bad character shift rule
- 好后缀规则 good suffix shift rule
- 例子：
 - 主串`text` : x y z **b** G s x y z
 - 模式串`pattern` : b G s **d** G s
 - 从右至左扫描，`pattern`的后缀Gs与`text`匹配，则称模式串的Gs为好后缀；
 - `pattern`与`text`出现第一次不匹配，称主串的不匹配字符**b**为坏字符。

算法描述

- 1) 先对 *pattern* 进行预处理，计算出
 - 坏字符规则表
 - 好后缀规则表
 - 2) 把 *pattern* 和 *text* 的滑动窗口从右向左匹配
 - 3) 如果出现不匹配，用两个规则表指导最大后移长度
-
- 如果没有坏字符规则表，BM算法与KMP算法类似
 - KMP算法第一个字符不匹配，一定只能后移一位。BM算法不是。

观察实例

比如 *text* = “HERE IS A SIMPLE EXAMPLE”
查找 *pattern* = “EXAMPLE”



HERE IS A SIMPLE EXAMPLE
EXAMPLE

The diagram illustrates the overlap between the text "HERE IS A SIMPLE EXAMPLE" and the pattern "EXAMPLE". A red dashed box highlights the overlapping region, which is the word "EXAMPLE". The text is displayed in a monospaced font, and the pattern is shown below it, aligned to the left. The red dashed box is drawn around the word "EXAMPLE" in the text and the word "EXAMPLE" in the pattern, showing that they match exactly.

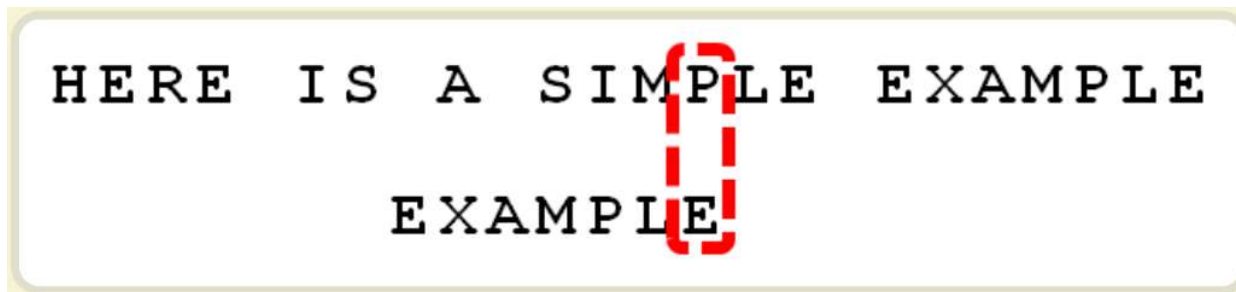
第一次匹配从最初滑动窗口最后一个开始



HERE IS A SIMPLE EXAMPLE
EXAMPLE

1) 首先，这里“S”就被称为“坏字符”（bad character），即不匹配的字符。我们还发现，“S”不包含在搜索词“EXAMPLE”之中，这意味着可以把滑动窗口直接后移到“S”的后一位。

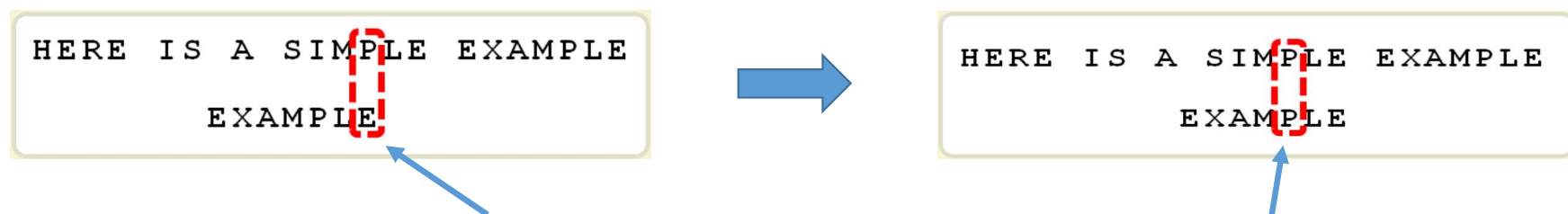
第二次匹配从新的滑动窗口最后一个开始



A diagram illustrating a sliding window for string matching. It consists of a light yellow rounded rectangle containing two lines of text in a monospaced font. The top line is "HERE IS A SIMPLE EXAMPLE" and the bottom line is "EXAMPLE". A red dashed rectangle highlights the last two characters of the top line, "P" and "E", and the last two characters of the bottom line, "P" and "E". This indicates a comparison between the two substrings.

2) 依然从尾部开始比较，发现“P”与“E”不匹配，所以“P”是“坏字符”。但是，“P”包含在搜索词“EXAMPLE”之中。所以，可以将搜索词后移两位，两个"P"对齐。

观察坏字符移动规则



后移位数 = 坏字符对应位置 - 搜索词中的上一次出现位置

“上一次出现位置”：

- 如果“坏字符”不包含在搜索词之中，则上一次出现位置为 -1。
- 否则，上一次出现位置按在搜索词中最靠右的位置计算

以“P”为例，它作为“坏字符”，在 *pattern* 串对应位置是第6位（从0开始编号）
在搜索词中的上一次出现位置为4，所以后移 $6 - 4 = 2$ 位。

再以前面第二步的“S”为例，对应位置在 *pattern* 串第6位，
上一次出现位置是 -1（即未出现），则整个搜索词后移 $6 - (-1) = 7$ 位。

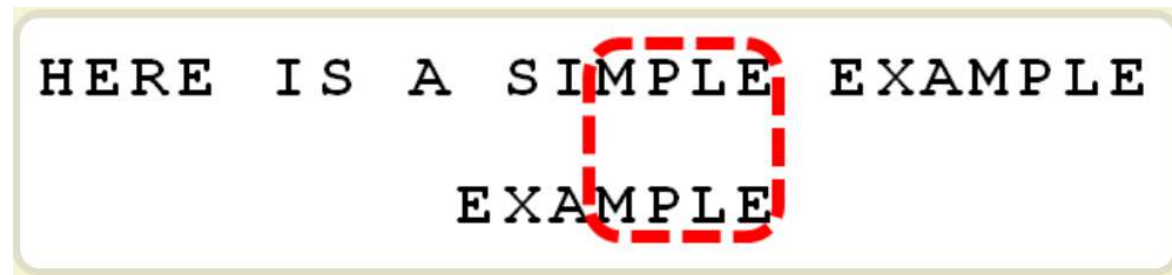
第三次匹配从新的滑动窗口最后一个开始



A diagram illustrating a string matching process. It features a yellow rounded rectangle containing the text "HERE IS A SIMPLE EXAMPLE" on the top line and "EXAMPLE" on the bottom line. A red dashed box highlights the letter 'E' at the end of "SIMPLE" on the top line and the letter 'E' at the end of "EXAMPLE" on the bottom line, indicating a match.

- 3) 依然从尾部开始比较，发现“E”与“E”匹配
即开始出现匹配后缀，继续往左匹配看有没有更长的后缀匹配

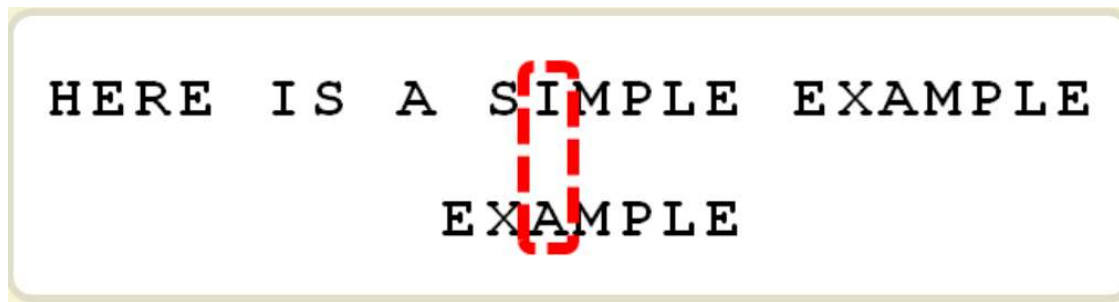
继续直到最长后缀匹配（滑动窗口不动）



HERE IS A SIMPLE EXAMPLE
EXAMPLE

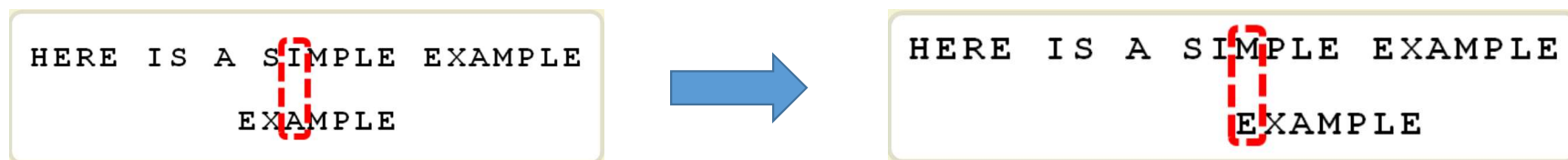
- 4) 发现“MPLE”与“MPLE”匹配。我们把这种情况称为“**好后缀**” (good suffix)，即所有尾部匹配的字符串注意，“MPLE”、“PLE”、“LE”、“E”都是好后缀。

继续发现当前滑动窗口内第一次不匹配



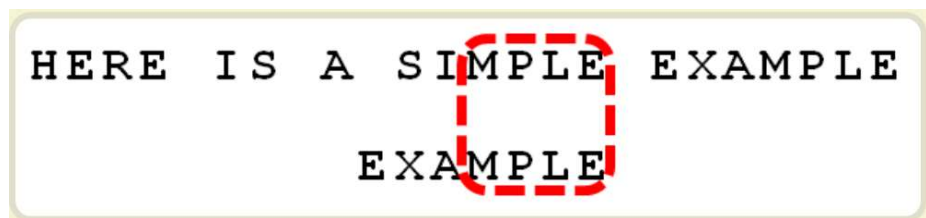
- 5) 再向左比较前一位，发现“I”与“A”不匹配。
所以，“I”是“坏字符”。此时，我们同时还有“好后缀”。

? 如果采用坏字符规则



- 4) 根据“坏字符规则”，此时搜索词应该后移 $2 - (-1) = 3$ 位。
问题是，此时有没有更好的移法？

观察好后缀移动规则



后移位数 = $\min\{\text{好后缀位置} - \text{搜索词中的上一次出现位置}\}$

- “好后缀”位置以最后一个字符为准（就是 *pattern* 最后一位）。这里，“EXAMPLE”的“MPLE”是好后缀，则它的位置以“E”为准，即6（从0开始计算）。
- 如果“好后缀”在搜索词中只出现一次，则它的上一次出现位置为 -1。
比如，“MPLE”在“EXAMPLE”之中只出现一次，则它的上一次出现位置为 -1（即未出现）。
- 如果“好后缀”有多个，则除了最长的那个“好后缀”（这里是“MPLE”），其他“好后缀”的上一次出现位置必须在头部。为什么？

这里，好后缀规则后移更多



HERE IS A SIMPLE EXAMPLE
EXAMPLE

- 5) 根据“**好后缀**规则”，所有的“好后缀”（MPLE、PLE、LE、E）之中，只有“E”在“EXAMPLE”还出现在头部，所以后移 $6 - 0 = 6$ 位。

接下来,



HERE IS A SIMPLE EXAMPLE
EXAMPLE

- 6) 继续从尾部开始比较, “P”与“E”不匹配, 因此,
“P”是“坏字符”。根据“坏字符规则”, 后移 $6 - 4 = 2$ 位。

最后




Diagram illustrating a string matching process. The text "HERE IS A SIMPLE EXAMPLE" is shown on the top line, and "EXAMPLE" is shown on the bottom line. A red dashed box highlights the last character 'E' of the top string and the entire bottom string "EXAMPLE", indicating a comparison or alignment step.

7) 从尾部开始逐位比较，发现全部匹配，于是搜索结束。

如果还要继续查找（即找出全部匹配），则根据"好后缀规则"，后移 $6 - 0 = 6$ 位，即头部的"E"移到尾部的"E"的位置。

小结

- 所以，**Boyer-Moore**算法的基本思想是，每次后移这两个规则之中的较大值。
- 并且，这两个规则的移动位数，只与搜索词有关，与原字符串无关。因此，可以预先计算生成《坏字符规则表》和《好后缀规则表》。

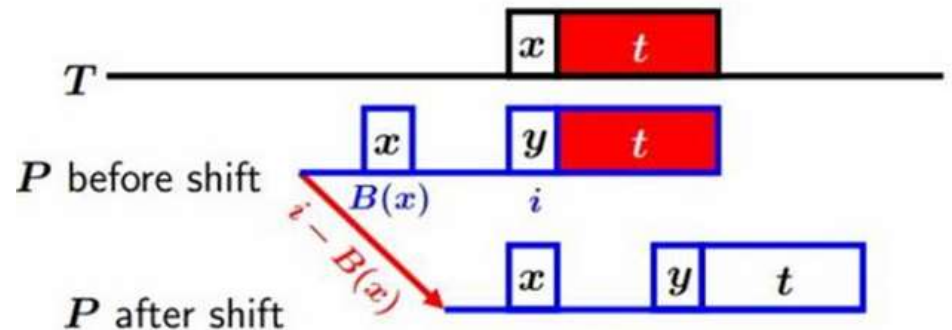
坏字符规则

坏字符规则定义

- 令 $B(x)$ 是字符 x 在 *pattern* 串里最靠右的位置
- 如果 x 不在 *pattern* 串出现,
 $B(x) = -1$

那么, 如果 $x \neq y$, 滑动窗口
可以后移的安全长度是
 $i - B(x)$

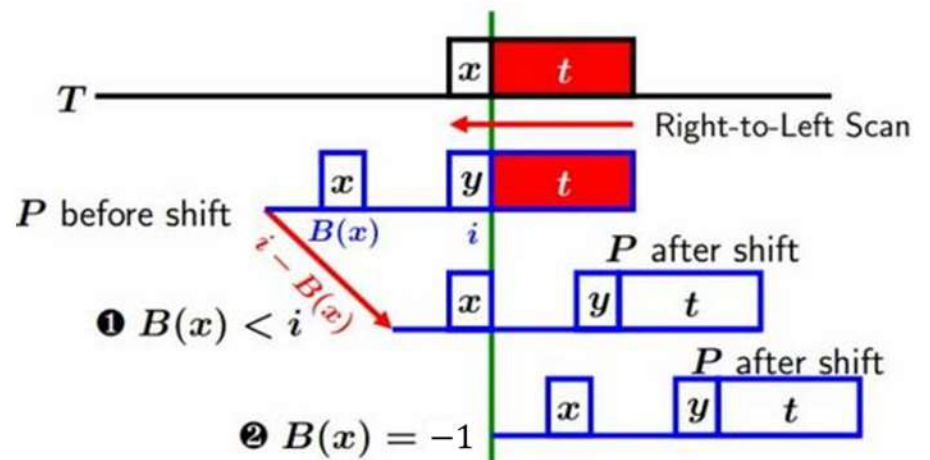
这里, i 是与坏字符 x 对应的 y
在 *pattern* 串的位置



坏字符规则适用情形

- 如果 $x \neq y$, 滑动窗口可以后移的**安全**长度是
 $i - B(x)$

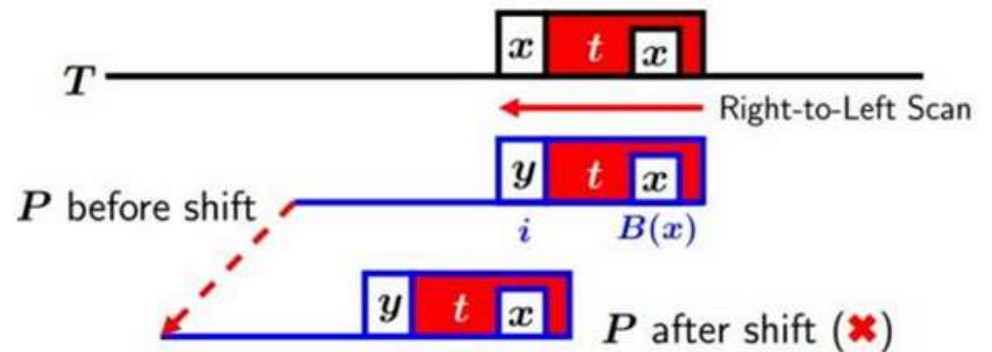
- $B(x) < i$
- $B(x) = -1$



坏字符移动规则失效情形

- 如果 $x \neq y$ ，滑动窗口可以后移的安全长度是
 $i - B(x)$

1. $B(x) < i$
2. $B(x) = -1$
3. $B(x) > i$



注意，这时候肯定有好后缀。

或者说，如果不匹配时没有好后缀，那么坏字符规则一定有效。

构造坏字符规则表

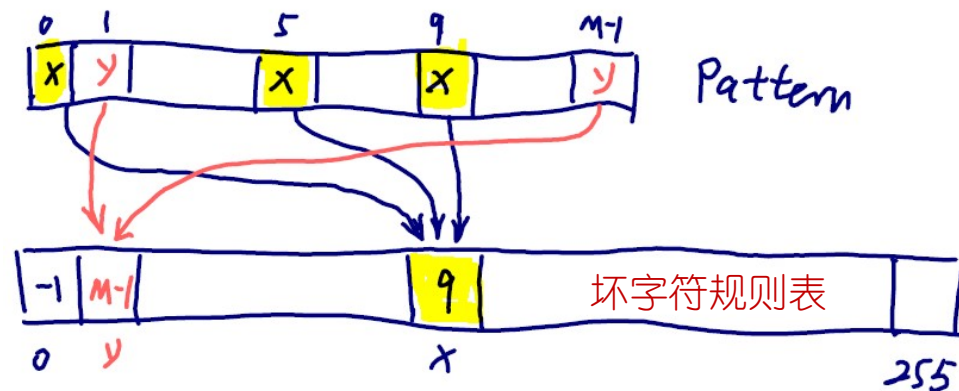
计算字符 x 在 $pattern$ 串里最靠右的位置

```
vector<int> MyString::GetBadCharTable(const char *pszPattern, int M) const
{
    // Initialize vector for each ascii character with -1,
    // which is rightmost position before first element at index 0.
    // That is, if a character does not appear in pattern, it is -1.
    vector<int> vecBadChars(256, -1);

    // This records the rightmost position that character patten[i] appears.
    // Starting from first to last, so that the rightmost position is updated eventually.
    for (int i = 0; i < M; ++i)
    {
        vecBadChars[(int)pszPattern[i]] = i;
    }

    return vecBadChars;
}
```

图中 y 对应的值有没有问题？



好后缀规则

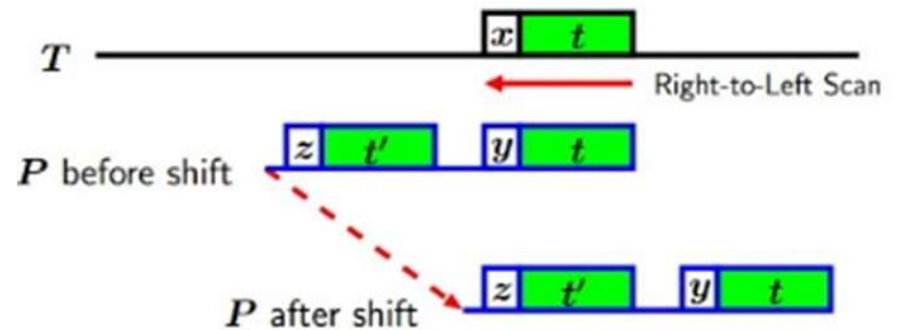
好后缀规则定义

情形一：

如果 $x \neq y$ ，好后缀是 t ，存在和 t 相同的子串 t' ，并且满足 $z \neq y$ ，

那么滑动窗口可以后移的**安全**长度是：

Case ①



t 的最后一个字符位置 - 最靠右的 t' 的最后一个字符位置

好后缀规则定义

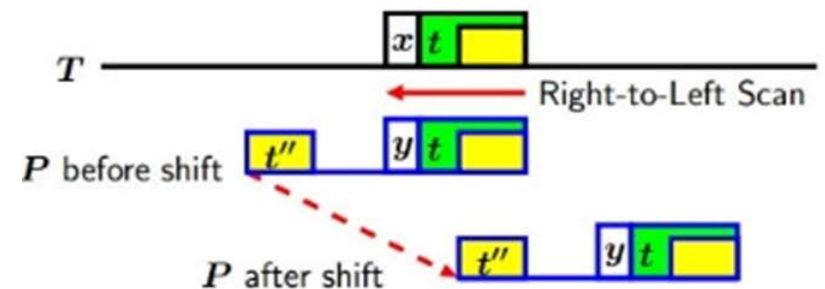
情形二：

如果 $x \neq y$ ，好后缀是 t ，不存在和 t 相同的子串 t' ，但是存在模式串的前缀 t'' 和 t 的后缀相同，

那么滑动窗口可以后移的**安全**长度是：

t 的最后一个字符位置 - 最长的 t'' 的最后一个字符位置

Case ②



好后缀规则定义

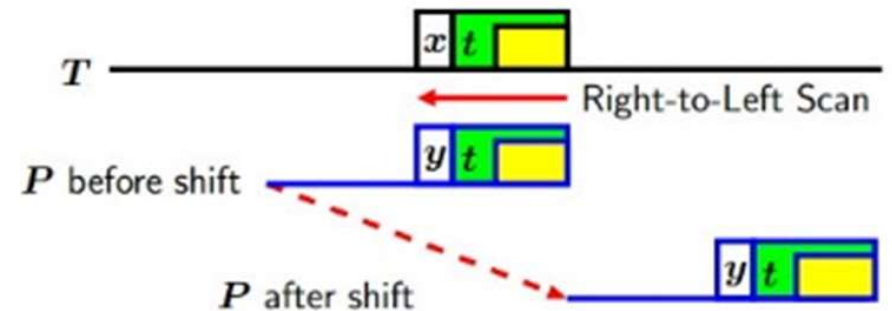
情形三：

如果 $x \neq y$ ，好后缀是 t ，不存在和 t 相同的子串 t' ，也不存在模式串的前缀 t'' 和 t 的后缀相同，

那么滑动窗口可以后移的**安全**长度是：

t 的最后一个字符位置 $-(-1) =$ 模式串的长度 M

Case ③



小结

情形三后移距离最理想

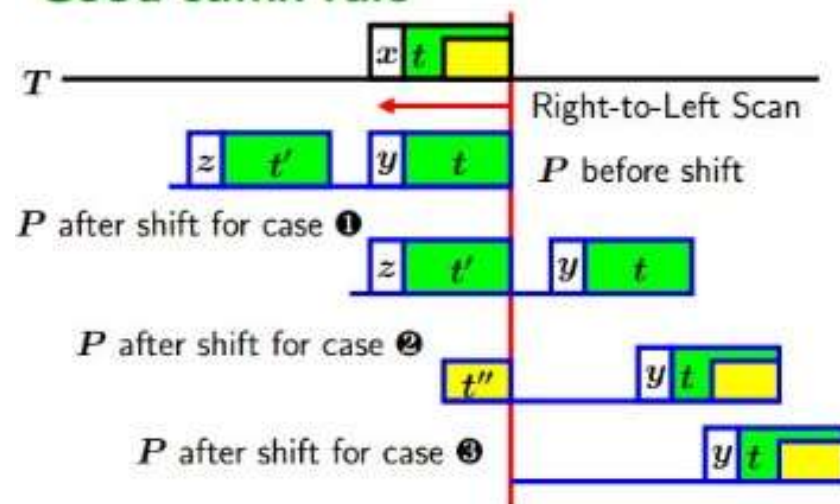
情形二也不错，是 t 中包含的在整个 $pattern$ 上的最长匹配前后缀

情形三看上去和最大长度表相像。区别是KMP算法是 $pattern$ 的所有前缀的最长匹配前后缀长度；这里的后移长度是基于 $pattern$ 的所有后缀 t ，和某个更长的后缀，其前缀和 t 匹配

问题是如何有效计算？

注意好后缀 t 其实是取决于 $text$ 。如果类似于KMP算法，我们得枚举所有可能的 t 。

Good suffix rule



如何有效计算好后缀规则

思路

- 理解规则时，是假定从任意好后缀 t 开始，然后……。这个在实现的时候会复杂，因为和 $text$ 串相关。
- 实现规则可以反过来，我们先看 $pattern$ 串里有没有适合的后缀。
- 构建好后缀规则表也先从情形三初始化，然后用情形二的结果更新，最后用情形一的结果更新。

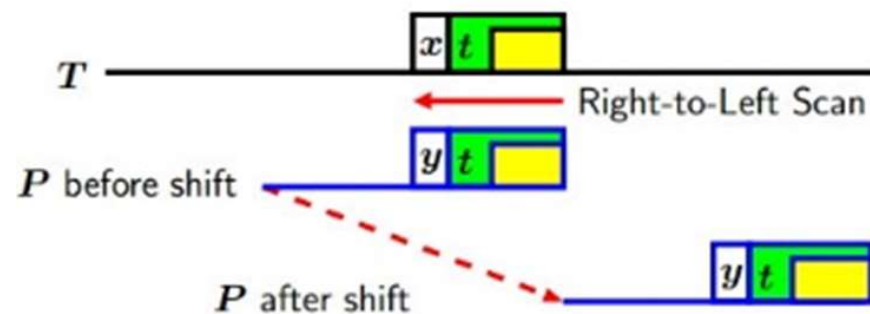
情形三

t 的最后一个字符位置 $-(-1) =$
模式串的长度 M

好后缀规则表初始化:

$$GS[i] = M, \forall i \in [0, M)$$

Case ③



情形二

后移位置：

$M - 1$ - 最长的 t'' 的最后一个字符位置

另外视角：

T 表示模式串 *pattern* 中最长的匹配前后缀

$GS[j] = M - \text{Leng}(T), \forall j \in [0, M - \text{Leng}(T))$

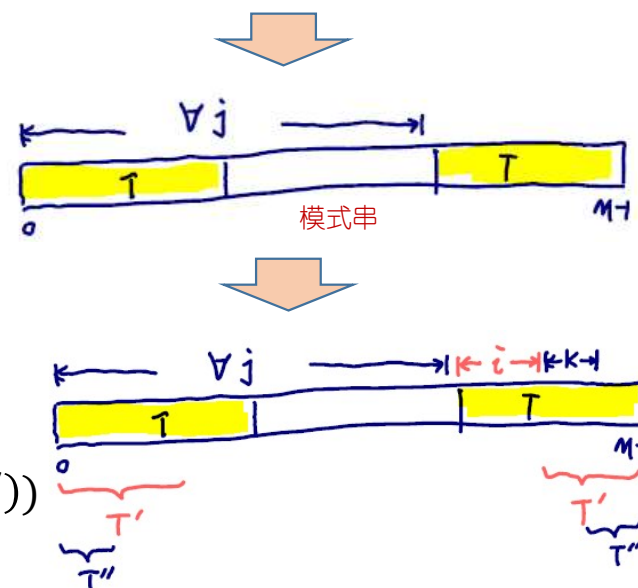
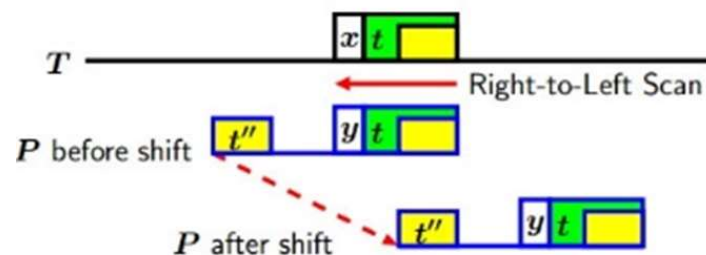
进一步：

T' 表示 T 中最长的匹配前后缀

$GS[i] = M - \text{Leng}(T'), \forall i \in [M - \text{Leng}(T), M - \text{Leng}(T'))$

注意其实是 *next[]* 的一个特殊情形。这里如何计算 T , T' 稍后再讲

Case ②



情形一

后移位置：

$M - 1$ - 最靠右的 t' 的最后一个字符位置

另外视角：

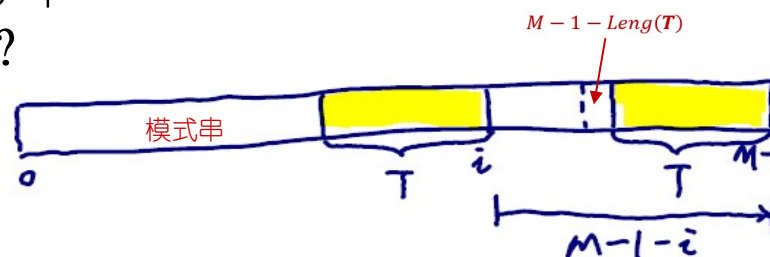
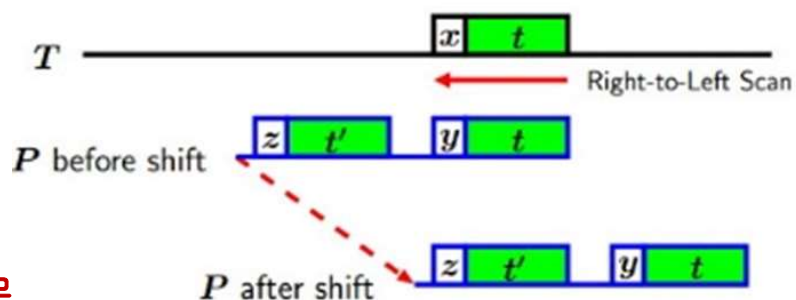
T 表示 *pattern* 中从 i 开始往左最长的与后缀匹配的子串

我们来看 $M - 1 - \text{Leng}(T)$ 处的好后缀移动是什么？

$$GS[M - 1 - \text{Leng}(T)] \leq M - 1 - i, \forall i \in [0, M - 1)$$

进一步：< 成立时表示还有更靠右的 T 。

Case ①



观察

- 结合情形一和情形二，我们需要知道

1) T ，也就是 *pattern* 中从 i 开始往左最长的与后缀匹配的子串



2) T ，也就是 *pattern* 中最长的匹配前后缀；

或者说从 i 开始往左，与后缀匹配的最长子串，并且要到头部。



- 归纳来说，如果定义

$Suffix[i]$ 是从 i 开始往左，与 *pattern* 后缀匹配的最长子串的长度，
需要先计算出 **$Suffix[i]$** , $\forall i \in [0, M - 1]$ 。

构造 *Suffix*[*i*]



```
vector<int> MyString::GetSuffixTable(const char * pszPattern, int M) const
```

```
{
```

```
// base case, when position i is at the last character.
```

```
vector<int> vecSuffixes(M, 0);
```

```
vecSuffixes[M-1] = M;
```

```
// For any position i, vecSuffixes[i] means length of longest substring matching with suffix
```

```
// pattern[i+1 - vecSuffixes[i]...i] == pattern[M-vecSuffixes[i]...M-1]
```

```
// Thus vecSuffixes[i] <= i+1. if equal, then it must be prefix.
```

```
for (int i = M - 2; i >= 0; --i)
```

```
{
```

```
int prev = i;
```

```
while (prev >= 0 && pszPattern[prev] == pszPattern[M - 1 - (i - prev)])
```

```
{
```

```
// decrease prev, until the first time that pattern[prev] is different or first character.
```

```
prev--;
```

```
}
```

```
// vecSuffixes[i] is the length between i and prev (prev might be -1).
```

```
vecSuffixes[i] = i - prev;
```

```
}
```

```
return vecSuffixes;
```

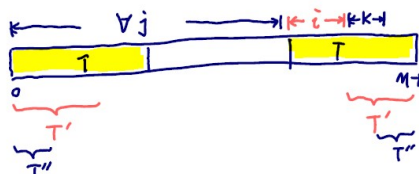
```
}
```

初始化情形，最后一位开始的*suffix*和*pattern*重合

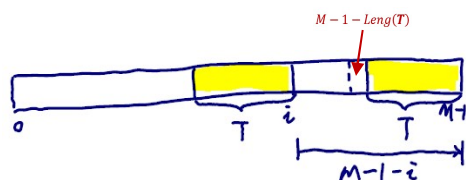
从*i*开始，向左查找最长*suffix*

构造好后缀规则表

情形三：初始化位最大移位



情形二：从右到左检查最长 T ，然后更新“ j ”段值，“ i ”段值，.....
注意这里 $Suffix[i] = i + 1$ 的意义



情形一：从左到右更新
“ $M - 1 - Suffix[i]$ ”处的值

```
vector<int> MyString::GetGoodSuffixTable(const char * pszPattern, int M) const
{
    vector<int> vecSuffixes = GetSuffixTable(pszPattern, M);

    // The case that good suffix has neither copy nor prefix in pattern
    // farthest move
    vector<int> vecGoodSuffixes(M, M);

    // The case that good suffix has prefix matching in pattern
    // Loop starting from last to first is because longer prefix will move slower.
    int j = 0;
    for (int i = M - 1; i >= 0; --i)
    {
        // This is when prefix [0...i] matches with suffix of pattern
        if (vecSuffixes[i] == i + 1)
        {
            // Any position j before M-1-i, we know its next move must be M-1-i
            // Later on, if there are smaller matching prefix i_1 < i,
            // the position j, after M-1-i and before M-1-i_1, should move M-1-i_1, which is bigger.
            for (; j < M - 1 - i; ++j)
            {
                // This line is to make sure we do not assign larger move.
                if (vecGoodSuffixes[j] == M)
                {
                    vecGoodSuffixes[j] = M - 1 - i;
                }
            }
        }

        // The case that good suffix has copy in pattern
        // shortest move.
        // Starting from first to last, because if vecSuffixes[i]==vecSuffixes[j] and i<j,
        // then m-1-i > m-1-j. We need to pick the smaller one, that is , j.
        for (int i = 0; i < M - 1; ++i)
        {
            vecGoodSuffixes[M - 1 - vecSuffixes[i]] = M - 1 - i;
        }
    }

    return vecGoodSuffixes;
}
```

BM算法实现

预处理《坏字符规则表》
和《好后缀规则表》



选一个大的j后移



```
int MyString::FindBM(const MyString & pattern) const
{
    int M = pattern.Length();
    if (M < 1)
    {
        return 0;
    }

    /* Preprocessing */
    vector<int> vecBadCharTable = GetBadCharTable(pattern._pszData, M);
    vector<int> vecGoodSuffixTable = GetGoodSuffixTable(pattern._pszData, M);

    /* Searching */
    for (int j = 0; j <= Length() - M;)
    {
        int last = M - 1;
        while (pattern[last] == _pszData[last + j])
        {
            last--;
            if (last < 0)
            {
                return j;
            }
        }

        // last is the position in pattern, where matching fails;
        // last + j is the corresponding position in text string.
        // bad character moves is based on mismatching char in text string.
        j += Max(vecGoodSuffixTable[last], last - vecBadCharTable[(int)_pszData[last + j]]);
    }

    return -1;
}
```

BM算法的特点

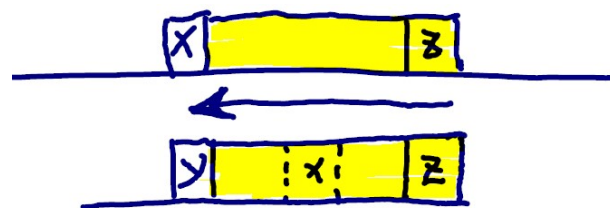
- 按照从右到左的顺序进行比较；
 - 预处理需要 $O(M + \delta)$ 的时间和存储空间；
 - 匹配时间的复杂性为 $O(M \times N)$ ？可以改进到 $O(N)$ （Galil rule）
 - 最好情况下，时间复杂性为 $O(N/M)$ ；
 - 最坏情况下需要 $3N$ 次字符比较。
-
- 坏字符规则表在实际应用中效率更高
 - 不用好后缀规则表，BM算法性能依旧好，而且实现大大简化
 - Boyer-Moore-Horspool 算法是BM算法的简化版本

Boyer-Moore-Horspool算法

回忆坏字符规则失效场景：因为坏字符 x 出现在 y 之后，黄色区域内

改进的办法是：

- 总是用 $text$ 里的最后一个和 $pattern$ 对齐的字符 z 来计算偏移量
- 只用坏字符规则，不过不用算 $pattern$ 里最后一个位置



只有坏字符规则，没有好后缀规则

```
vector<int> MyString::GetBadCharTable(const char *pszPattern, int M) const
{
    // Initialize vector for each ascii character with -1,
    // which is rightmost position before first element at index 0.
    // That is, if a character does not appear in pattern, it is -1.
    vector<int> vecBadChars(256, -1);

    // Horspool Algo: Do not include position i = M - 1
    // This records the rightmost position that character patten[i] appears.
    // Starting from first to last, so that the rightmost position is updated eventually.
    for (int i = 0; i < M-1; ++i)
    {
        vecBadChars[(int)pszPattern[i]] = i;
    }

    return vecBadChars;
}
```

- 只用坏字符规则，不过不用算`pattern`里最后一个位置

滑动窗口后移总是按最后一个字符偏移量

```
/* Preprocessing */
vector<int> vecBadCharTable = GetBadCharTable(pattern._pszData, M);
vector<int> vecGoodSuffixTable = GetGoodSuffixTable(pattern._pszData, M);

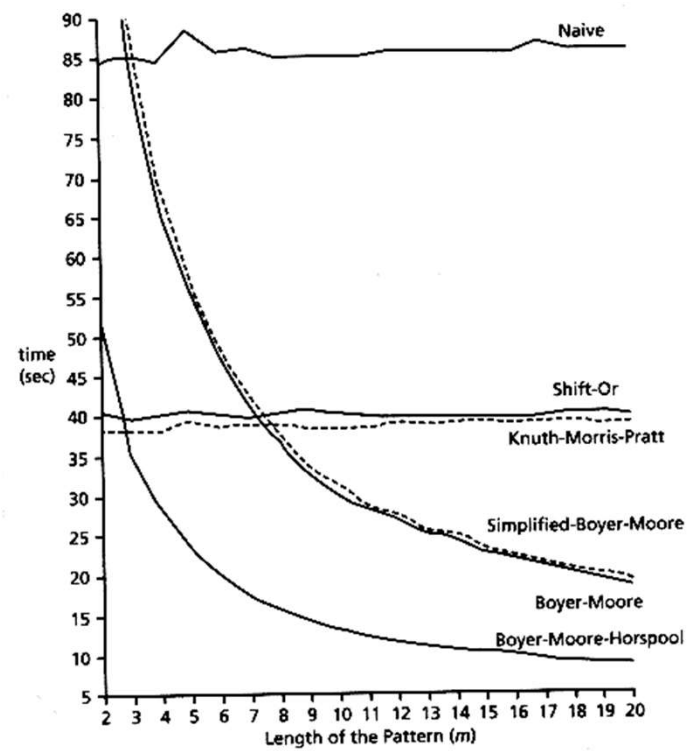
/* Searching */
for (int j = 0; j <= Length() - M;)
{
    int last = M - 1;
    while (pattern[last] == _pszData[last + j])
    {
        last--;
        if (last < 0)
        {
            return j;
        }
    }

    // Horspool alg: always use last character to do shift: last = M - 1
    // last is the position in pattern, where matching fails;
    // last + j is the corresponding position in text string.
    // bad character moves is based on mismatching char in text string.
    j += M - 1 - vecBadCharTable[(int) _pszData[M - 1 + j]];
}
```

- 总是用滑动窗口的最后一个字符来计算偏移量
- 甚至你可以考虑用窗口后的第一个字符, `_pszData[M + j]`

```
// last is the position in pattern, where matching fails;
// last + j is the corresponding position in text string.
// bad character moves is based on mismatching char in text string.
j += Max(vecGoodSuffixTable[last], last - vecBadCharTable[(int) _pszData[last + j]]);
```

匹配算法比较



练习

1. 把几个匹配算法加入MyString，然后修改课件中的算法，使得返回值是text中匹配的次數，而不是第一个匹配位置
2. 任给一个字符串，比如“abacabadd”，输出它的一个最长的是回文的前缀。这里是“abacaba”。
提示：对“abacabadd” + “#” + “ddabacaba”，求next[M]

课后拓展：

- 比较暴力算法，kmp算法，bm算法，horspool算法，和strstr（）或者std::string.find()，

```
#include <cstring>
char* strstr ( const char* text, const char* pattern );
```
- 测试文本查找：
 - Pattern长度小于10，匹配次数为0；匹配次数很多次
 - Pattern长度100，1000，10000，pattern字符集10个，26个
 - Fibonacci word: $S_n = S_{n-1}S_{n-2}$
 - 小说 <http://www.gutenberg.org/files/55455/55455-0.txt>
 - 基因序列 <http://hgdownload.soe.ucsc.edu/goldenPath/priPac1/bigZips/chromFa.tar.gz>

Q&A

Thanks!