

数据结构与算法

DATA STRUCTURE

第十讲 字符串匹配算法

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

字符串string

自定义串

- 我们同样要对c-style string进行封装
- 使得支持insert, delete, search, replace, etc
- 尤其是string匹配算法

串的实现

- 之前作业实现MyString类
- 参考书上例子和之前的IntArray
- 今天看几种加亮的string匹配算法

- | | |
|-------------|--------|
| • FindBasic | 暴力算法 |
| • FindKR | KR 算法 |
| • FindKMP | KMP 算法 |
| • FindBM | BM 算法 |

```
class MyString
{
public:
    MyString(const char * pszValue);
    ~MyString();

    // copy constructor
    MyString(const MyString & other);

    // Overloaded assignment
    MyString& operator= (const MyString& rhs);
    // see Append
    MyString& operator+=(const MyString& rhs);
    // return a new MyString
    MyString operator+(const MyString& rhs);
    // Access character by index
    char & operator[] (int index);
    const char & operator[] (int index) const;
    // Compare two strings
    int operator== (const MyString& str) const;

    // String matching function.
    int FindBasic(const MyString & pattern) const;
    int FindKR(const MyString & pattern) const;
    int FindKMP(const MyString & pattern) const;
    int FindBM(const MyString & pattern) const;

    // Implement an efficient algorithm. Test append lots of small string.
    MyString & Append(const MyString & str);

    // Copy/cut previous string, if size is smaller. Otherwise, nothing is changed.
    void Resize(int size);
    // Change capacity size. If capacity is larger, then new buffer is allocated while string is copi
    void Reserve(int capacity);

    // Insert a new string at index; previous content after index are placed afterwards.
    void Insert(int index, const char * pszStr);
    // Delete substring of size length starting at index.
    void Remove(int index, int length);

    const char * GetString() const { return _pszData; }

    // Usually "size" includes null terminator and length does not.
    inline int Length() const { return _size - 1; }

    inline bool IsEmpty() const { return Length() <= 0 || _pszData == nullptr; }

    friend std::ostream& operator<<(std::ostream& out, const MyString &str)
    {
        out << "\"" << str.GetString() << "\"";
        return out;
    }
}

private:
    char* _pszData;
    int _capacity;
    int _size;
```

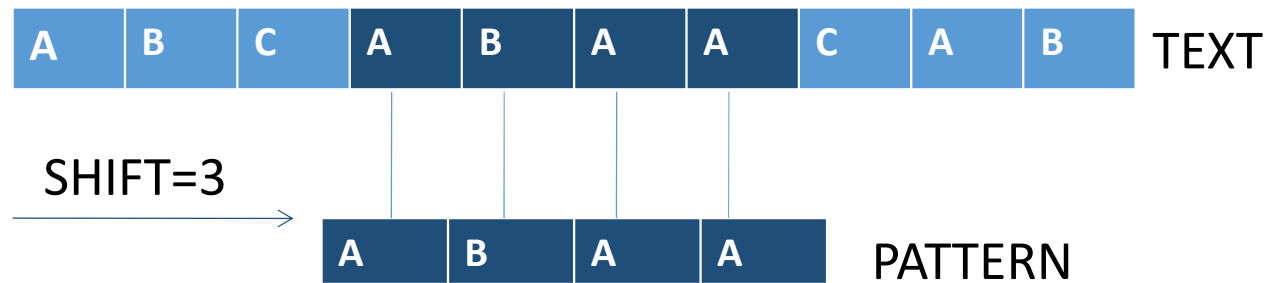
STRING MATCHING ALGORITHMS



串的模式匹配算法

- [字符串匹配](#)是计算机的基本任务之一。
- 举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？
- 串的匹配算法解决的是在长串中查找短串的一个、多个或所有出现的问题。
- 通常，我们称长串为`text`，称短串为`pattern`。
- 一个长度为`M`的`pattern`可被表述为`Pattern[0 ... M - 1]`；
- 长度为`N`的`text`可被表述为`Text[0 ... N - 1]`
- 而模式匹配的任务是找到`pattern`在`text`中的出现。

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)

暴力算法

(Brute-Force)暴力算法

- BF算法是最基本的串模式匹配算法。
- 其过程是：
 - 依次比对 *pattern* 和滑动窗口中的对应位置上的字符；
 - 比对完成后将滑动窗口向右移动1，直到找到所需的匹配为止。

第一轮比较：

y C G T A G C G T C T C T C A T A T G T C A T G C
1 2 3 4

x C G T C T C T C

比较窗口右移 1 个位置

第二轮比较：

y C G T A G C G T C T C T C A T A T G T C A T G C
1

x C G T C T C T C

比较窗口右移 1 个位置

第三轮比较：

y C G T A G C G T C T C T C A T A T G T C A T G C
1

x C G T C T C T C

比较窗口右移 1 个位置

...

第六轮比较：

y C G T A G C G T C T C T C A T A T G T C A T G C
1 2 3 4 5 6 7 8

x C G T C T C T C

找到子串，程序结束

Brute-Force算法示例

```
int MyString::FindBasic(const MyString & pattern) const
{
    int M = pattern.Length();

    // i iterates in text string; j iterates in pattern string.
    for (int i = 0; i <= Length() - M; i++)
    {
        int j = 0;
        while (j < M && pattern[j] == _pszData[i + j])
        {
            j++;
        }

        // This is the case of matching.
        if (j == M)
        {
            return i;
        }
    }

    return -1;
}
```

暴力算法的特点总结

- 1) 不需要预处理过程
- 2) 只需固定的存储空间
- 3) 滑动窗口的移动每次都为1
- 4) 字符串的比对可按任意顺序进行（从左到右、从右到左、或特定顺序均可）
- 5) 算法的时间复杂度为 $O(M \times N)$
- 6) 暴力算法的效率并不理想：
 - 逐一匹配效率低
 - 该算法太健忘，前一次匹配的信息扔掉

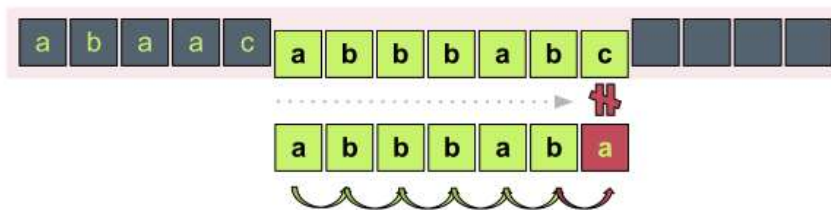
Karp-Rabin算法

KR(Karp-Rabin)算法

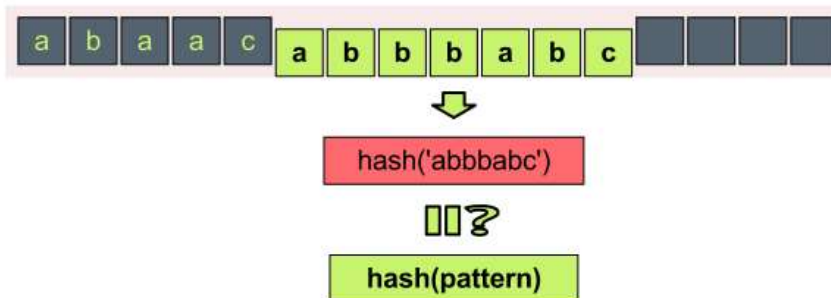
- KR算法优化 “滑动窗口内容逐一匹配”
- KR算法思路：
 - 通过一个（哈希）函数计算`pattern`串对应的一个值，*FingerPrint*
 - 用同样的函数计算`text`滑动窗口内`M`个字符对应的指纹值
 - 比较`pattern`串指纹值与`text`滑动窗口内字符的指纹值
 - 仅当相等时，再来比较窗口内的子串是否相等
- 注意计算指纹值的时候可以重复利用之前的计算结果

对比示例

Brute-force matching



Rabin-Karp



哈希Hash函数

- KR算法的效率取决于哈希函数的选取。

- 滑动窗口的指纹值计算公式之一：

$$\text{hash}(W[0 \dots M-1]) = (W[0] \times 2^{M-1} + \dots + W[M-1] \times 2^0) \bmod q$$

其中， q 为一个素数。

- 当滑动窗口右移时，可以重复利用之前的计算结果：

$$\text{rehash}(\text{hash}, a, b) = ((\text{hash} - a \times 2^{M-1}) \times 2 + b) \bmod q$$

其中，

hash 为上一滑动窗口的指纹值

a 为即将移出滑动窗口的字符

b 是即将移入滑动窗口的字符

哈希Hashing函数

```
long MyString::Hashing(int leng) const
{
    long result = 0;
    for (int i = 0; i < leng && i < Length(); i++)
    {
        result = ((result << 1) + _pszData[i]) % Prime;
    }

    return result;
}

long MyString::Rehashing(long hashing, int a, int b, int offset) const
{
    long result = (hashing + Prime - (a << offset) % Prime) % Prime;
    result = ((result << 1) + b) % Prime;

    return result;
}
```

KR算法示例

第一轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C
x C G T C T C T C

Hash(y[0..7])=17910

第二轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C
x C G T C T C T C

Hash(y[1..8])=18735

第三轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C
x C G T C T C T C

Hash(y[2..9])=19378

...

第六轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C
1 2 3 4 5 6 7 8
x C G T C T C T C

Hash(y[5..12])=Hash(x)=18055

找到子串, 函数结束

```
int MyString::FindKR(const MyString & pattern) const
{
    int M = pattern.Length();
    if (M > Length())
    {
        return -1;
    }

    long patHash = pattern.Hashing(M);
    long txtHash = Hashing(M);

    // Whether pattern is matched at the beginning
    if (patHash == txtHash)
    {
        // Check characters one by one if hashing value is the same.
        if (Check(_pszData, pattern.GetString(), M))
        {
            return 0;
        }
    }

    for (int i = M; i < Length(); i++)
    {
        // Rehashing based on previous hash value.
        txtHash = Rehashing(txtHash, _pszData[i - M], _pszData[i], M-1);
        if (patHash == txtHash)
        {
            // Check characters one by one if the same hashing value is found.
            if (Check(_pszData + i - M + 1, pattern.GetString(), M))
            {
                return i - M + 1;
            }
        }
    }

    return -1;
}
```

KR算法的特点

- 1) 利用哈希的方法进行数值比较，但是比字符比较慢
- 2) 预处理需要 $O(M)$ 的时间和常数的存储空间
- 3) 实际算法时间通常为 $O(M + N)$ ，如果有个好的哈希函数
- 4) 最坏情况下，算法时间复杂度为 $O(M \times N)$
 - 比如 *text* = “aaaaaaaaaaaaa ... aaaaaaaaaaaaaaaaaa”
 - 查找 *pattern* = “aaaa aaab”

KR算法的特点

- 能够处理多模式匹配，
比如检测抄袭

Rabin-Karp can detect plagiarism!



Knuth-Morris-Pratt算法

KMP算法

- [Knuth-Morris-Pratt算法](#)（简称KMP）是最常用的匹配算法。它以三个人在1976年独立发明命名，起头的那个K就是著名科学家 Donald Knuth。



- 其中一个黑客，theory meets practice
- 保证匹配在线性时间内完成 $O(M + N)$

实例

比如 *text* = “BBC ABCDAB ABCDABCDABDE”

查找 *pattern* = “ABCDABD”

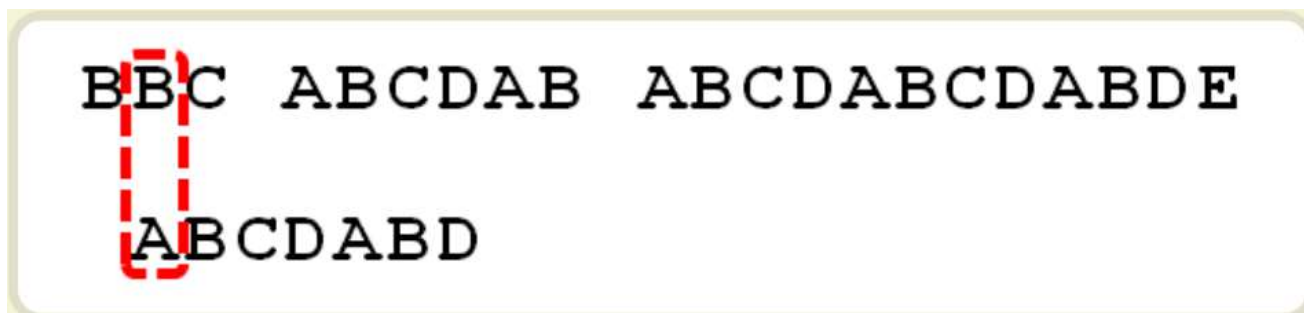


1) 首先，因为B与A不匹配，所以滑动窗口后移一位。

实例

比如 *text* = “BBC ABCDAB ABCDABCDABDE”

查找 *pattern* = “ABCDABD”

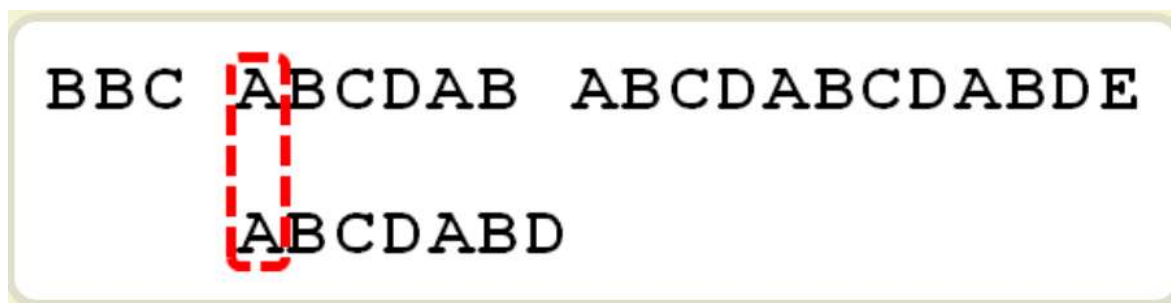


2) 因为B与A不匹配，滑动窗口再往后移。

实例

比如 *text* = “BBC ABCDAB ABCDABCDABDE”

查找 *pattern* = “ABCDABD”

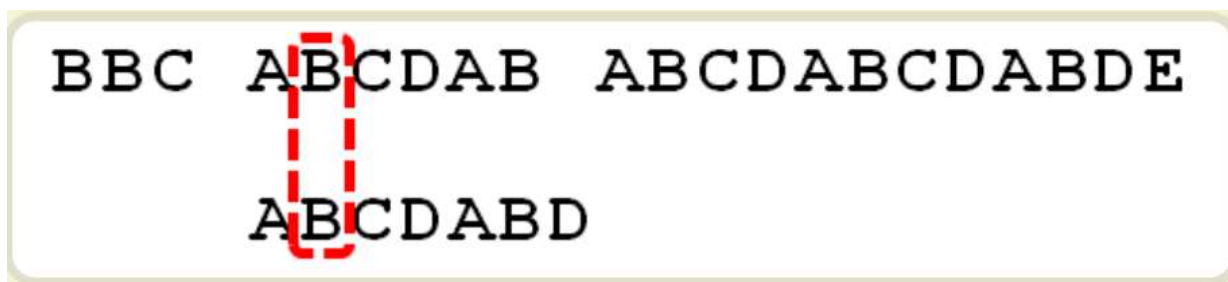


3) 就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。

实例

比如 *text* = “BBC ABCDAB ABCDABCDABDE”

查找 *pattern* = “ABCDABD”



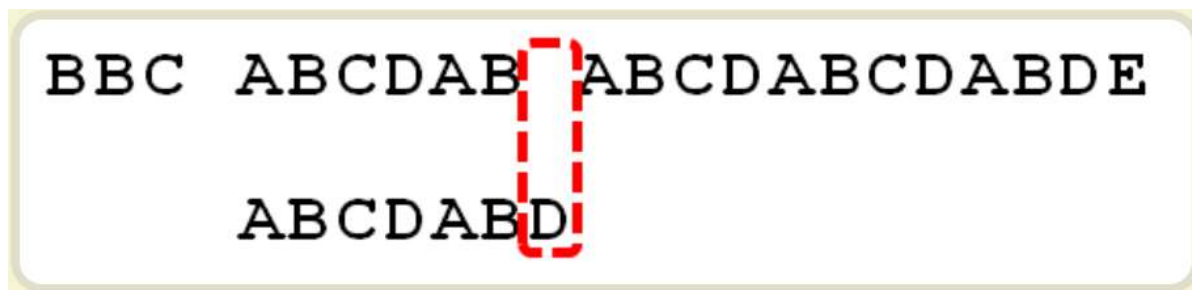
BBC ABCDAB ABCDABCDABDE
ABCDABD

4) 接着比较字符串和搜索词的下一个字符，还是相同。

实例

比如 *text* = “BBC ABCDAB ABCDABCDABDE”

查找 *pattern* = “ABCDABD”

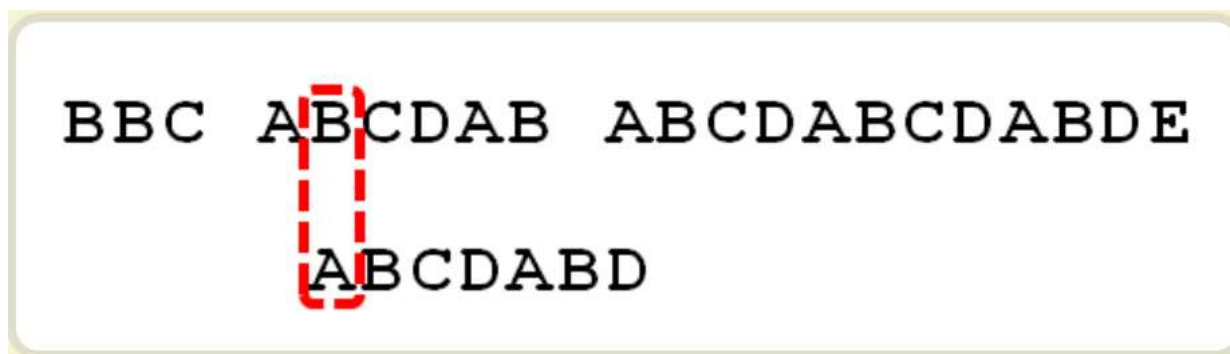


BBC ABCDAB ABCDABCDABDE
ABCDABD

5) 直到字符串有一个字符，与搜索词对应的字符不相同为止。

实例

比如 *text* = “BBC ABCDAB ABCDABCDABDE”
查找 *pattern* = “ABCDABD”

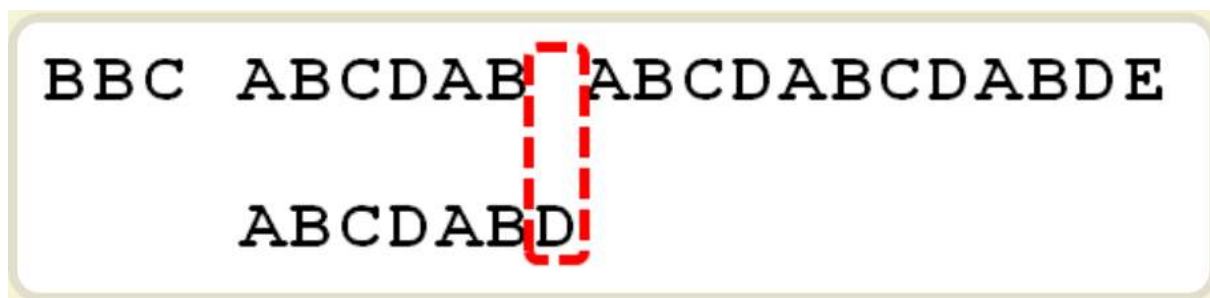


6) 这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把“搜索位置”移到已经比较过的位置，重比一遍。

实例

比如 *text* = “BBC ABCDAB ABCDABCDABDE”

查找 *pattern* = “ABCDABD”



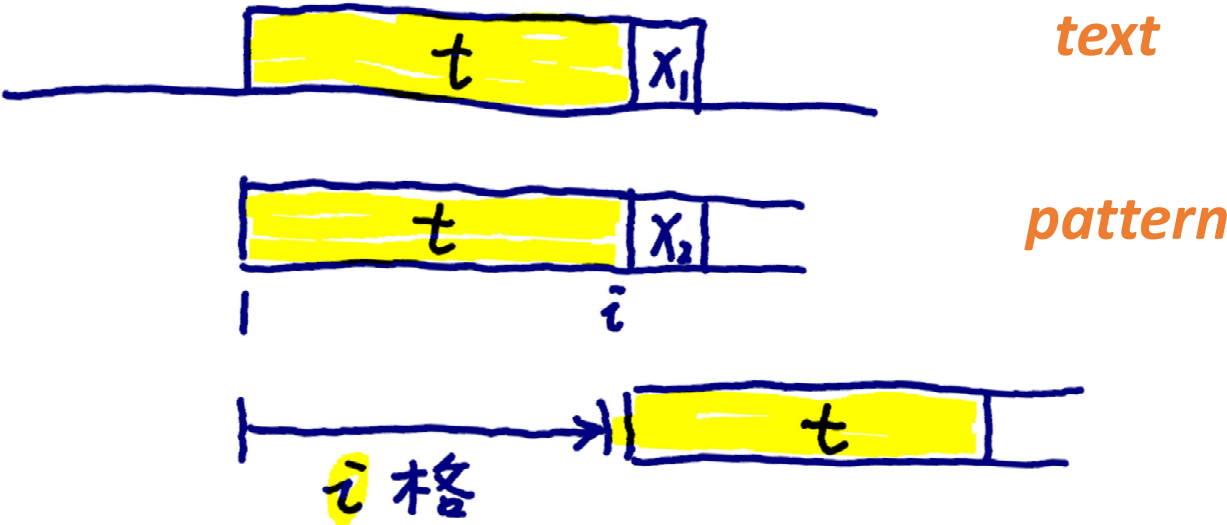
BBC ABCDAB ABCDABCDABDE
ABCDABD

7) 注意一个基本事实是，当空格与D不匹配时，你其实知道前面六个字符是"ABCDAB"。

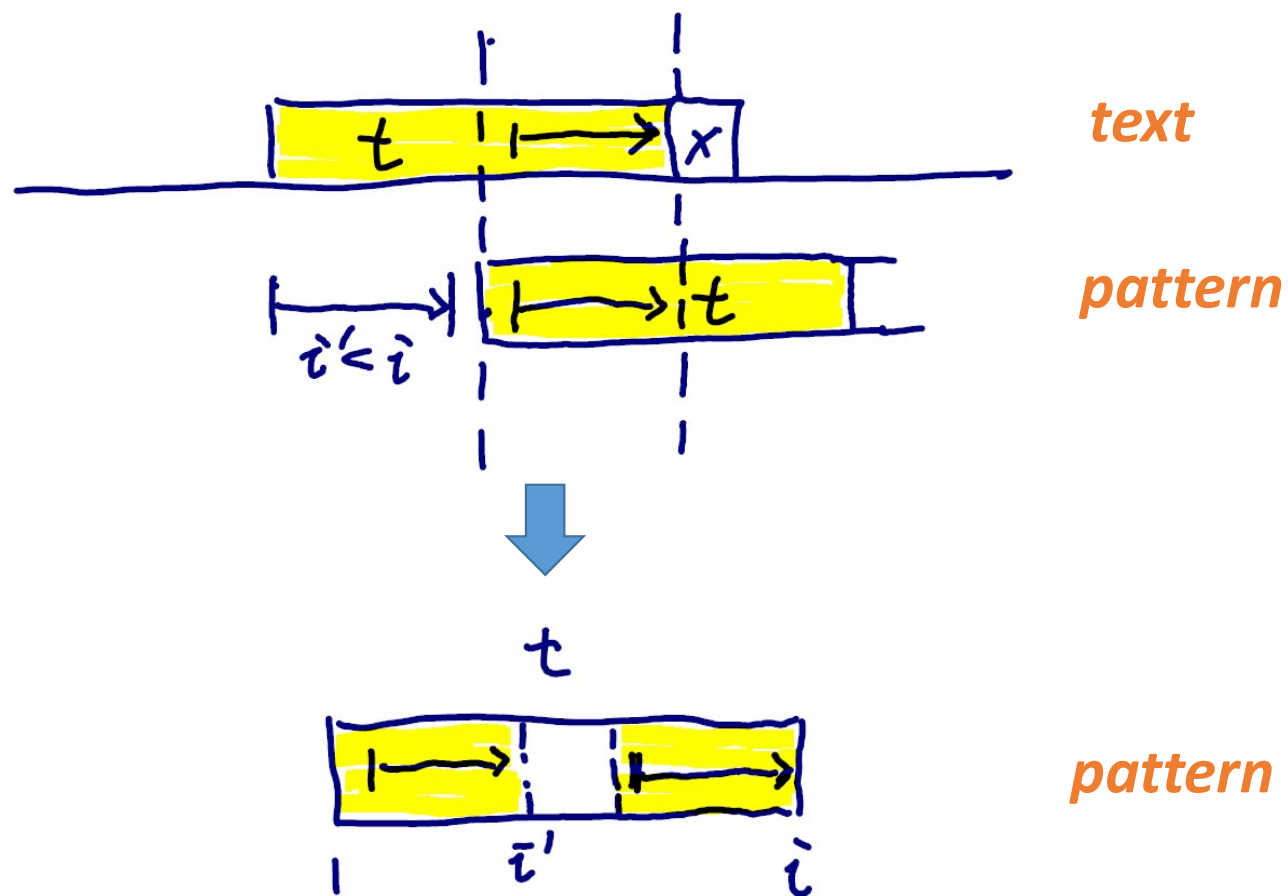
KMP算法思路

- 暴力匹配时候是每次移动一格，再从头开始匹配
- 能不能利用之前的匹配结果直接跳过几个，不必重复？
- 如果能，跳几个合适？
- **KMP**算法的思路是，设法利用这个已知匹配信息，把“搜索位置”跳过一些肯定不会匹配的位置，继续把滑动窗口向后移多位。

理想情况

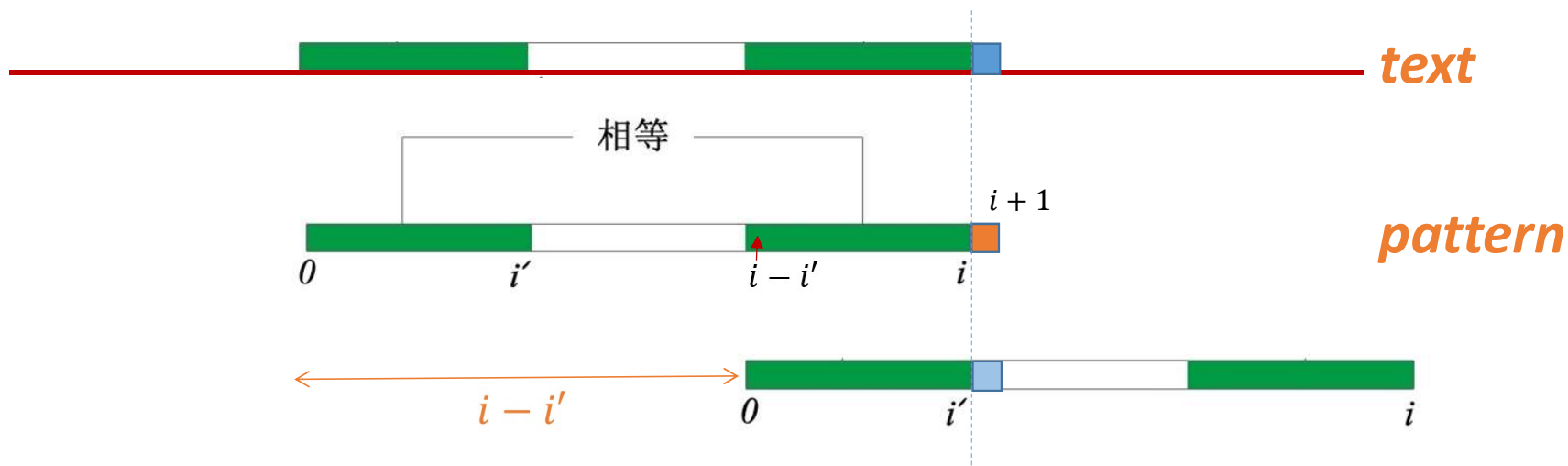


如果不能，是什么情况



观察

如果在第 $i + 1$ 位置出现不匹配，那么下一个可能匹配的位置是 $i' + 1$ ，也就是最长首尾匹配串的长度



为什么用最长？

- 因为用别的匹配位置可能漏下最近匹配点；并且原来的不是最长的匹配点还会出现在下次匹配中。
- 所以可后移的安全长度是 $i - i'$ ，也就是说任何小于 $i - i'$ 的后移都不可能匹配。

小结

- 如果知道，在任何Pattern[0 ... i] 中，最长的匹配前后缀的长度
 - 那么，当在 $i + 1$ 处不匹配时，就可以推算下一次可能的匹配位置
 - 这个只和`pattern`有关，和`text`没关
-
- 所以关键是先寻找最长匹配的前缀后缀，称为“最大长度表”，再用它来匹配`text`

什么是"前缀"和"后缀"

字符串： "bread"

前缀： b , br , bre , brea

后缀： read , ead , ad , d

- "前缀"指除了最后一个字符以外，一个字符串的全部头部组合
- "后缀"指除了第一个字符以外，一个字符串的全部尾部组合

1. 穷举寻找abababca的最大长度表

1	char:		a		b		a		b		a		b		c		a	
2	index:		0		1		2		3		4		5		6		7	
3	value:		0		0		1		2		3		4		0		1	

[0] : “a”的前缀和后缀都为空集，最大长度为0；

[1] : “ab”的前缀为[a]，后缀为[b]，最大长度为0；

[2] : “aba”的前缀为[a, ab]，后缀为[ba, a]，最大长度为1；

[3] : “abab”的前缀为[a, ab, aba]，后缀为[bab, ab, b]，最大长度为2；

[4] : “ababa”的前缀为[a, ab, aba, abab]，后缀为[baba, aba, ba, a]，最大长度为3；

[5] : “ababab”的前缀为[a, ab, aba, abab, ababa]，后缀为[babab, abab, bab, ab, b]，最大长度为4；

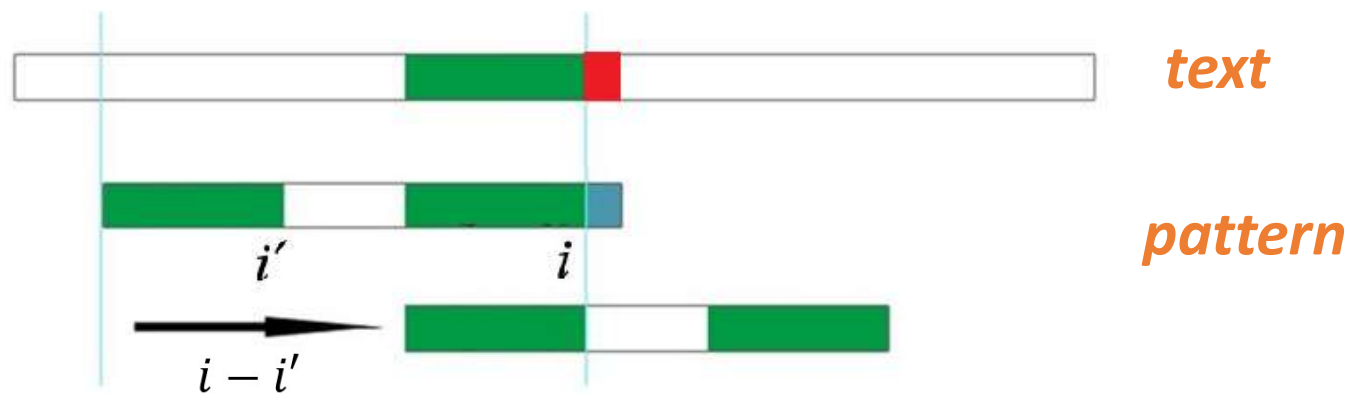
[6] : “abababc”的前缀为[a, ab, aba, abab, ababa, ababab]，后缀为[bababc, ababc, babc, abc, bc, c]，最大长度为0。

[7] : “abababca”的前缀为[a, ab, aba, abab, ababa, ababab, abababc]，后缀为[bababca, ababca, babca, abca, bca, ca, a]，最大长度为1。

2. 用最大长度表指导匹配

- 滑动窗口从第一位开始和 *pattern* 串匹配
- 匹配时，如果第一个字符不同，滑动窗口后移一位后重新匹配
- 要不然，在滑动窗口内匹配相同字符，直到 $[0 \dots i]$
- 当在 $i + 1$ 处不匹配时，滑动窗口后移多位：

移动位数 = 已匹配的字符数 - 最大长度表对应的值



回到实例

实例： ABCDABD的最大长度表

搜索词	A	B	C	D	A	B	D
最大长度表	0	0	0	0	1	2	0

- [0] : “A”的前缀和后缀都为空集，最大长度为0；
- [1] : “AB”的前缀为[A]，后缀为[B]，最大长度为0；
- [2] : “ABC”的前缀为[A, AB]，后缀为[C, BC]，最大长度0；
- [3] : “ABCD”的前缀为[A, AB, ABC]，后缀为[D, CD, BCD]，最大长度为0；
- [4] : “ABCD A”的前后缀共有元素为“**A**”，长度为1；
- [5] : “ABCDAB”的前后缀共有元素为“**AB**”，长度为2；
- [6] : “ABCDABD”的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，最大长度为0。

实例

BBC ABCDAB ABCDABCDABDE
ABCDABD

搜索词	A	B	C	D	A	B	D
最大长度表	0	0	0	0	1	2	0

已知空格与D不匹配时，前面六个字符“ABCDAB”是匹配的。查表可知，最后一个匹配字符B对应的值为2，因此按照下面的公式算出向后移动的位数：

$$\text{移动位数} = \text{已匹配的字符数} - \text{最大长度表对应的值}$$

因为 $6 - 2$ 等于4，所以将搜索词向后移动4位。

实例

BBC ABCDAB ABCDABCDABDE
 ABCDABD

搜索词	A	B	C	D	A	B	D
最大长度表	0	0	0	0	1	2	0

因为空格与C不匹配，搜索词还要继续往后移。
这时，已匹配的字符数为2（“AB”），最大长度表中对应的值为0。
所以，移动位数 = 2 - 0，结果为2，于是将搜索词向后移2位。

实例

BBC ABCDAB ABCDABCDABDE
ABCDABD

搜索词	A	B	C	D	A	B	D
最大长度表	0	0	0	0	1	2	0

因为空格与A不匹配，继续后移一位，不用查表。

实例

BBC ABCDAB ABCDABCDABDE
 ABCDABD

搜索词	A	B	C	D	A	B	D
最大长度表	0	0	0	0	1	2	0

逐位比较，直到发现C与D不匹配。
于是，移动位数 = $6 - 2$ ，继续将搜索词向后移动4位。

实例

BBC ABCDAB ABCDABCDABDE
ABCDABD

搜索词	A	B	C	D	A	B	D
最大长度表	0	0	0	0	1	2	0

逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。
如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，
再将搜索词向后移动7位，这里就不再重复了

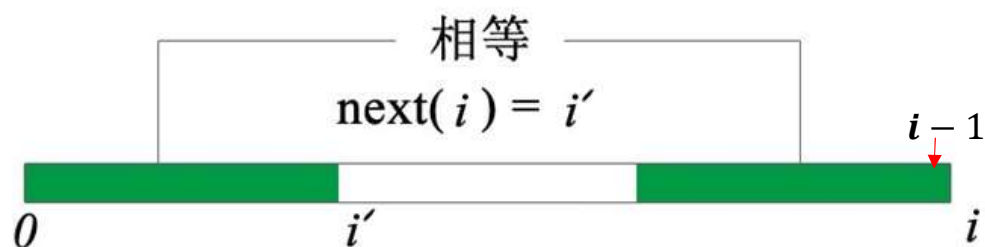
问题

- 到此为止我们用肉眼识别最大长度表，效率不高
- 还有当第一个字符不匹配的时候，最大长度表失效
- 为此，我们引入一个类似的表， $next[0 \dots M - 1]$
- 这里 $next[i]$ 表示 $Pattern[0 \dots i - 1]$ 中最长的前后缀匹配的长度

数学定义

用 $next[i]$ 表示Pattern[0 ... $i - 1$]中最长的前后缀匹配的长度

$$next(i) = \begin{cases} -1 & \text{如果 } i = 0, \\ \max\{i' \mid P[0 \dots i' - 1] = P[i - i' \dots i - 1]\} & \text{如果有前后缀匹配,} \\ 0 & \text{其它情况.} \end{cases}$$



注意边角情形，比如

- $i = 0$ ，Pattern[0 ... - 1]无意义，初始状态，设为-1
- $i = 1$ ，也就是只有一个字符，没有前后缀，所以为 $next[1] = 0$
- 其它情况： $i > 1$ ，并且没首尾匹配时，为0

如何有效计算 $next[]$

迭代计算 $next[]$

已知 $next[\leq j]$, 计算 $next[j + 1]$, 图中 $next[j] = i$

1. 如果 $p[j] == p[i]$, 那么 $next[j + 1] = next[j] + 1$
2. 要不然, 循环查找 $next[next[j]] = k$, 然后验证 $p[j]$ 和 $p[k]$
3. 最后如果回溯到 -1 , 就设置 $next[j + 1] = 0$



类似数学归纳法证明迭代算法

假定已知 $next[\leq j]$ ，如何计算 $next[j + 1]$

图中 $next[j] = i$ 是红色块， $next[i] = k$ 是绿色块， $next[k]$ 是黄色块

- 首先 $next[j + 1]$ 最多为 $i + 1$ ，当 $p[j] = p[i]$ 时。【反证法】
- 要不然，最多为 $next[i] + 1 = k + 1$ ，当 $p[j] = p[next[i]] = p[k]$ 时。【反证法】
- 要不然，最多为 $next[k] + 1$ ，当 $p[j] = p[next[next[i]]] = p[next[k]]$ 时。【反证法】
-, 直到0



next[]实现

这里为了符合初始状态，添加next[0]起始值-1。真实值是next[1 ... M - 1]

1. 如果 $p[j] == p[i]$ ，那么
 $next[j] = i + 1$
2. 要不然，循环查找
 $next[next[i]] = k$ ，然后验证
 $p[j] == p[k]$
3. 最后如果回溯到-1，就设置
 $next[j] = 0$

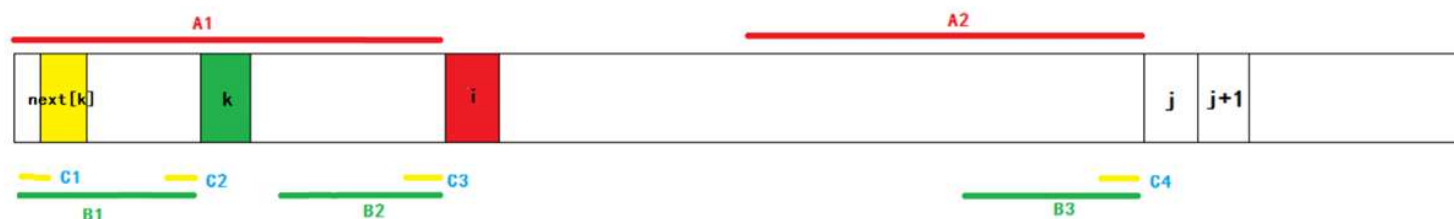
```
vector<int> MyString::GetNext(const char * pszPattern, int M) const
{
    vector<int> next;
    next.push_back(-1);

    // It is position of pattern[0...j], where we need to calculate next value.
    int j = 0;
    // i is the index of max prefix matching with suffix of pattern[0...j]
    // Its initial value is -1, since initial pattern[0] has neither suffix nor prefix.
    // Thus, the base case is that next[j] == next[0] == i == -1.
    int i = -1;

    // Now assume we have next[<=j], calculate next[j+1], one by one.
    while(j < M)
    {
        while(i >= 0 && pszPattern[j] != pszPattern[i])
        {
            // Using known result, find the next[i] where [0...i-1] matches with suffix.
            i = next[i];
        }

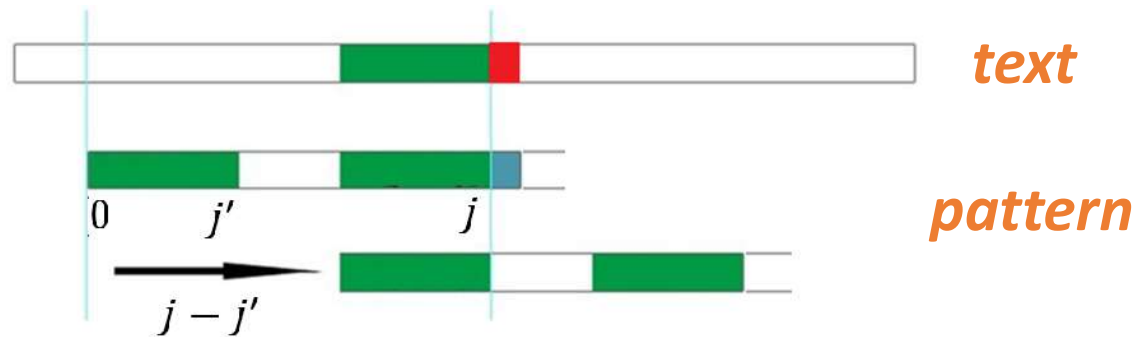
        // Now next[j+1] = i+1, here i might be next[next[next[j]]] or -1.
        // That is, either we find a prefix, or no prefix. Both cases should increase i by 1
        next.push_back(++i);
        j++;
    }

    return next;
}
```



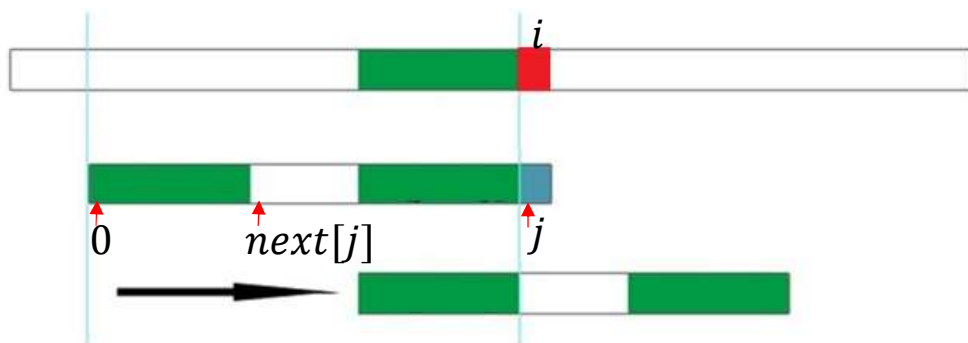
kmp匹配算法

- 滑动窗口从第一位开始和`pattern`串匹配
- 在滑动窗口内匹配相同字符，直到 $[0 \dots j]$ 。如果 $j = -1$ ，表示第一个字符就不匹配。
- 当在 $j + 1$ 处不匹配时，回溯到 $next[j + 1] = j' + 1$ ，即绿块长度
- 相当于，滑动窗口后移多位： $j + 1 - next[j + 1] = j - j'$



注意如果第一个字符不同，用 $j = -1$ 代入上面公式，滑动窗口移动 $0 - next[0] = 1$ ，即后移一位后再重新匹配

KMP算法实现



$next[j]$ 代表Pattern[0 ... $j - 1$]中最长的匹配前后缀的长度

注意起始状态 $next[0] = -1$

```
int MyString::FindKMP(const MyString & pattern) const
{
    int M = pattern.Length();
    if (M > Length())
    {
        return -1;
    }

    vector<int> vecNext = GetNext(pattern.GetString(), M);

    int i = 0, j = 0;
    while(i < Length() && j < M)
    {
        if (j < 0 || _pszData[i] == pattern[j])
        {
            i++;
            j++;
        }
        else
        {
            j = vecNext[j];
        }
    }

    if(j == M)
    {
        return i - j;
    }

    return -1;
}
```

KMP算法的特点

- 1) 预处理需要 $O(M)$ 的时间和存储空间
 - 2) 算法复杂度 $O(M + N)$ ，即便是最坏情况
 - 3) 实现比较tricky
- 为什么是线性复杂度？
 - 循环体内， i 不减； j 可能回溯，即变小。不是明显线性。
 - 另外的角度：每次循环，要么是 i 增加，要么是滑动窗口滑动多位，甚至同时发生。

Q&A

Thanks!