06. 串

6.1 串的定义

定义：串（string）是由零个或多个字符组成的有限队列，又叫字符串

一般记为s=“$a_1a_2a_3……a_n$”（n≥0），其中，s是串的名称，用双引号（有些书中
也用单引号）括起来的字符序列是串的值，注意引号不属于串的内容。

$a_i(1≤i≤n)$**可以是字母、数字或其他字符**，i就是该字符在串中的**位置**。串中的字符数目n称为**串的长度**，定义中谈到“有限”是指长度n是一个有限的数值。

零个字符的串称为空串（nullstring），它的长度为零，可以直接用两个双引号“”表示，也可以用希腊字母“Φ”来表示。所谓的序列，说明串的相邻字符之间具有前驱和后继的关系。

子串与主串，串中任意个数的连续字符组成的子序列称为该串的子串，相应地，包含子串的串称为主串。
子串在主串中的位置就是子串的第一个字符在主串中的序号。
例如：“over” in “lover”、“lie” in “believe”

6.2 串的比较

串相等：
串的长度以及它们各个对应位置的字符都相等时，才算是相等。
即给定两个串：s = "$a_1a_2...a_n$"，t = "$b_1b_2...b_m$"，当且仅当n = m && $a_1 = b_1$，$a_2 = b_2$，... $a_n = b_m$时，我们认为s = t。

串判断大小：
给定两个串：s = "$a_1a_2...a_n$"，t = "$b_1b_2...b_m$"，当满足以下条件之一时（s<t）

- n < m 且 $a_i = b_i$ (i = 1,2,...,n)

  例如当s = "hap"，t = "happy"，就有s < t，因为t比s多了两个字母

- 存在某个k <= min(m,n)，使得$a_i = b_i$(i = 1,2,...,k - 1)，$a_k<b_k$

  例如当s = "happen"，t = "happy"，因为两串的前四个字母均相同，二两串第五个字母(k值)，字母e的ASCII码是101，而字母y的ASCII码为121，显然e<y，所以s < t

6.3 串的存储结构

6.3.1 串的顺序存储结构

串的顺序存储结构是用一组地址连续的存储单元来存储串中的字符序列的。

一般用`\0`来标志串的终结

6.3.2 串的链式存储结构

与线性表相似，但是如果每一个结点对应一个字符，就会存在很大的空间浪费。因此，一个结点可以考虑存放多个字符，最后一个结点若是未被占满时，可以用`#`或其他非串字符补全。

一个结点存多少个字符是影响串的处理效率的重要因素

总的来说，串的链式存储结构除了在串与串的链接时会方便一点，其他操作都不如顺序存储结构。

完整实现

```
#include <iostream>
#include <string>
#include <cstring>

using namespace std;

#define maxsize 20

class String
{
public:
    void initStr(String* p);   //初始化串
    int getLen(String* p);    //获取串的长度
    void createStr(String* p);  //创建串
    bool emptyStr(String* p);    //判断串是否为空
    void printfStr(String* p);    //输出串
    void strAssign(String* p, char temp[]); // 生成一个其值等于字符串chars的串
```

```
    void strCopy(String* p1,  String* p2);     //如果p2不为空，就由串p2复制得串p1
    void clearStr(String* p);   //将串S清空
    int strCompare(String* p1, String* p2);     //比较两串大小 p1 < p2 返回-1
    String* concat(String* p1, String* p2);  //连接两串，返回链接的结果
    void insertStr(String* p1, int pos, String* p2);  //在p1串的pos位置上插入一个p2
串，返回插入的结果
    void deleteStr(String* p, int i, int j);   //删除子串，从p1中删除第i个字符开始的长
度为j的子串
    String* substr(String* p, int i, int j); //返回串p中第i个字符起长度为j的子串
    int getIndex(String* p1, String* p2);   //p1 p2均为非空串，若p1中存在子串与p2相等，
返回位置。否则，返回0
private:
    char data[maxsize];
    int length;
};

void String::initStr(String* p)
{
    p->length = 0;
}

int String::getLen(String* p)
{
    int i = 0;
    while (p->data[i] != '\0')
    {
        i++;
    }
    return i;
}

void String::createStr(String* p)
{
    cin >> p-> data;
    p->length = getLen(p);
}

bool String::emptyStr(String* p)
{
    if (p->length == 0)
    {
        puts("字符串为空");
        return true;
    }
    else
    {
        puts("字符串非空");
        return false;
```

```
    }
}

void String::printfStr(String* p)
{
    if (emptyStr(p) == true) puts("字符串为空");
    else
    {
        for (int i = 0; i < p->length; i++) cout << p->data[i];
        puts("");
    }
}

void String::strAssign(String* p, char temp[])
{
    int i;
    for (i = 0; temp[i] != '\0'; i++) p->data[i] = temp[i];
    p->length = i;
}

void String::strCopy(String* p1, String* p2)
{
    int i;
    for (i = 0; i < p2->length; i++) p1->data[i] = p2->data[i];
    p1->length = p2->length;
}

void String::clearStr(String* p)
{
    int i;
    for (i = 0; i < p->length; i++) p->data[i] = '\0';
    p->length = 0;
}

int String::strCompare(String* p1, String* p2)
{
    int i = 0;
    while ((p1->data[i] == p2->data[i]) && p1->data[i] != '\0' && p2->data[i] !=
'\0')
    {
        i++;
    }
    int res = p1->data[i] - p2->data[i];        //res < 0 p1 < p2; res == 0 p1 ==
p2; res > 0 p1 > p2;
    return res;
}

String* String::concat(String* p1, String* p2)
```

```
{
    //TODO: 判断两串加起来是否会超出上限
    int i, j;
    String* t;
    for (i = 0; i < p1->length; i++) t->data[i] = p1->data[i];
    for (j = 0; j < p2->length; j++) t->data[j + i] = p2->data[i];
    t->data[j] = '\0';
    t->length = p1->length + p2->length;

    return t;
}

void String::insertStr(String* p1, int pos, String* p2)
{
    //TODO: 判断插入后是否会超出上限
    //此处假设插入后不会超出上限
    int i;
    //为要插入的串留出位置，也就是把pos及其后面的子串向后移p2->length
    for (i = p1->length - 1; i >= pos; i--)
    {
        p1->data[i + p2->length] = p1->data[i];
    }
    for (int j = 0; i < p2->length; j++)
    {
        i++;
        p1->data[i] = p2->data[j];
    }
    p1->data[p1->length + p2->length] = '\0';
    p1->length += p2->length;
}

void String::deleteStr(String* p, int i, int j)
{
    if (i < 0 || i > p->length || i - 1 + j > p->length)
    {
        puts("删除的位置不对");
        return;
    }
    for (int k = i - 1; k < p->length - j; k++)
    {
        p->data[k] = p->data[k + j];
    }
    p->data[p->length] = '\0';
}

String* String::substr(String* p, int i, int j)
{
    if (i < 1 || i > p->length || i - 1 + j > p->length)
```

```
    {
        puts("没有这个子串，长度不足");
        return NULL;
    }
    String* str2 = new String;
    int k;
    for (k = 0; k < j; k++)
    {
        str2->data[k] = p->data[i - 1];
        i++;
    }
    str2->length = j;
    str2->data[j] = '\0';
    return str2;
}

int String::getIndex(String* p1, String* p2)
{
    int i = 1;
    while (i <= p1->length - p2->length + 1)
    {
        String* temp = substr(p1, i, p2->length);
        if (strCompare(temp,p2) != 0)
        {
            i++;
        }
        else
        {
            return i;
        }
    }
    puts("找不到这个子串");
    return -1;
}
```

6.4 朴素的模式匹配算法

找一个单词在一篇文章中的定位问题—串的模式匹配（子串的定位）

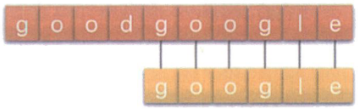
以主串`S="goodgoogle"`中，找到`T="google"`这个子串的位置为例，模拟朴素的模式匹配算法。

1. 主串S第一位开始，S与T前三个字符都匹配成功，但S第四个字母是d而T的是g。第一位匹配失败。

2. 主串S第二位开始，主串S首字母是o，要匹配的T的首字母是g，匹配失败

3. blablabla以此类推将子串整体向后移一位

4. 主串S第五位开始，S与T，6个字母全匹配，匹配成功。



简单地说，就是对主串的每一个字符作为子串的开头，与要匹配的字符串进行匹配。对主串做一层循环，对每个字符开头做T的长度的二层循环，直到匹配成功或全部遍历完成

```
int String::strCompare(String* p1, String* p2)
{
    int i = 0;
    while ((p1->data[i] == p2->data[i]) && p1->data[i] != '\0' && p2->data[i] !=
'\0')
    {
        i++;
    }
    int res = p1->data[i] - p2->data[i];        //res < 0 p1 < p2; res == 0 p1 ==
p2; res > 0 p1 > p2;
    return res;
}
int String::getIndex(String* p1, String* p2)
{
    int i = 1;
    while (i <= p1->length - p2->length + 1)
    {
        String* temp = substr(p1, i, p2->length);
        if (strCompare(temp,p2) != 0)
        {
            i++;
        }
        else
```

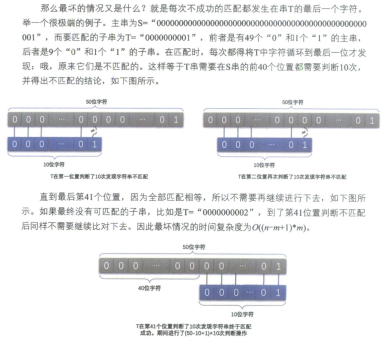
```
        {
            return i;
        }
    }
    puts("找不到这个子串");
    return -1;
}
```

6.4.1 朴素算法的时间复杂度分析与优化

最好情况：
一开始就匹配成功，例如`"googlegood"`中找`"google"`，因为需要遍历子串的长度来逐个判断是否会相等，时间复杂度为$O(m)$，m为子串长度

其他情况：
像朴素算法的例子中第二、三、四位一样，每次都是首字母都不匹配，那么就没必要进行子串的循环。那么时间复杂度为$O(n+m)$，其中n为主串长度，m为要匹配的子串长度。根据等概率原则，平均是$(n+m)/2$次查找。时间复杂度是$O(n+m)$

最坏情况：
每次子串和目标串的前面部分的匹配都成功，但是每次都在子串的最后一位失败，直到能匹配到目标串的最后一位为止。时间复杂度为$O(nm)$



显然朴素算法是低效的，然而模式匹配操作在计算机运算中随处可见，如果频繁使用朴素的算法运行效率极慢。

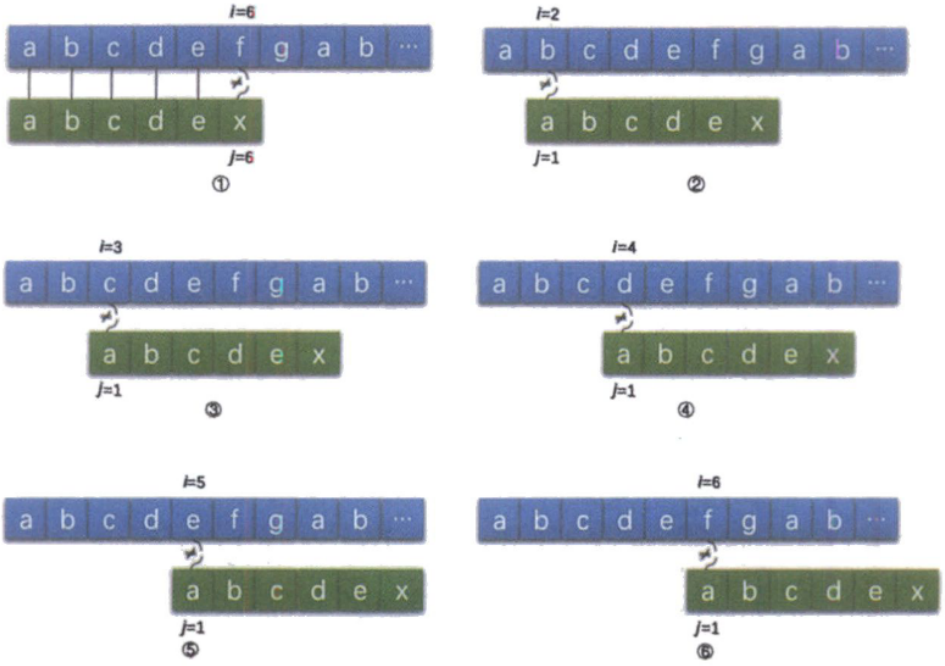
6.5 KMP模式匹配算法

6.5.1 从朴素到KMP

step.1

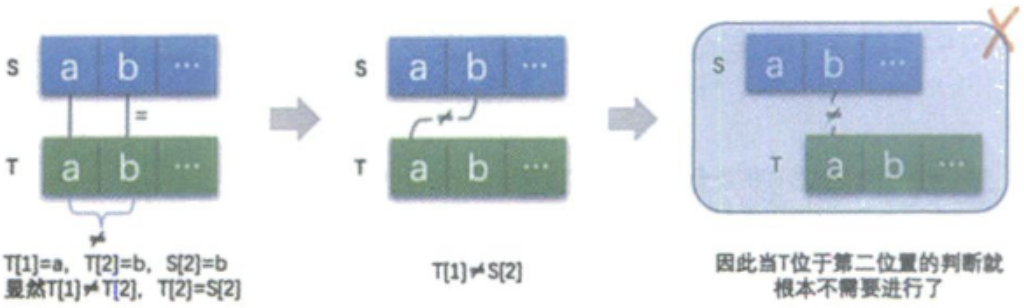
从朴素算法开始，寻找哪里有可以优化的地方

如果主串S=“abcdefgab”，其实还可以更长一些，我们就省略掉只保留前9位，我们要匹配的T=“abcdex”，那么如果用前面的朴素算法的话，前5个字母，两个串完全相等，直到第6个字母，“f”与“x”不等，如下图的①所示。

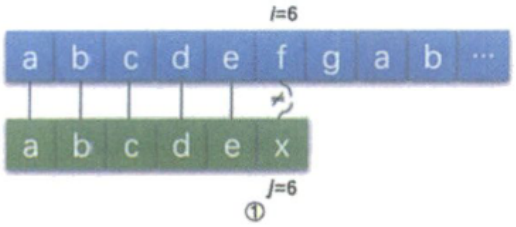


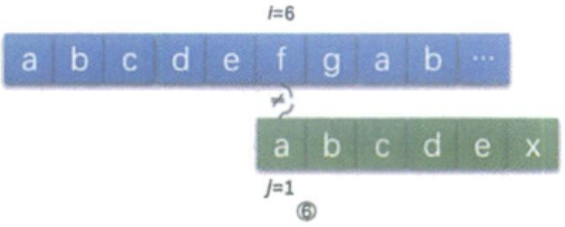
从②到⑤的暴力式遍历匹配显然是需要优化的。观察可以发现在①中子串的“abcde”与主串的前五个字符都匹配，而在**子串中“abcdex”首字母“a”与后面的“bcdex”都不相同**，也就是说在①中匹配成功，意味着已经暗含了②~⑤中将子串的首字母“a”与主串的字符匹配的结果，也就是都匹配失败了（“a”与“bcde”都不相同）。那么②~⑤的判断就是多余的，可以优化。

简洁而言，如果我们知道T串的**首字母“a”与T中后面的字符均不相等**，而T串的第二位的“b”与S串中的“b”在上图中的①中已经判断是相等的，那么也就是T串中的首字符“a”与S串中的第二位“b”是不需要判断也知道是不可能相等的，这样上图中的②就是可以省略的。



同理，在我们知道T串的**首字母“a”与T中后面的字符均不相等**的前提下，T串的“a”与S串的“c”“d”“e”也都可以在①之后就可以确定是不相等的，所以朴素算法中的②~⑤都是可以省略的。





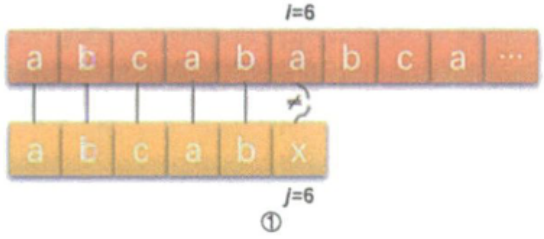
为什么要保留⑥而不直接跳到`i=7`呢？因为我们在第一步判断的是`S[6] != T[6]`，但是`T[1]`有可能等于`S[6]`，需要重新判断一下。

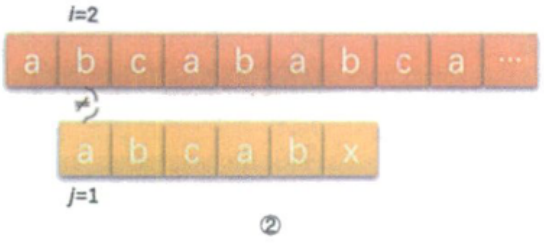
step.2

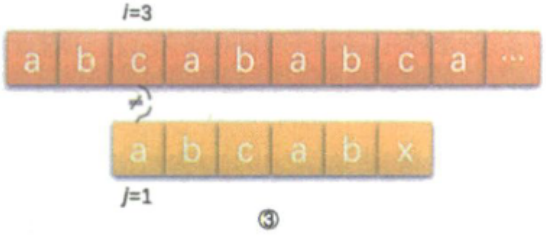
在上一步中，我们都保留了一个默认条件，也就是**T串的首字母“a”与T中后面的字符均不相等**，如果在T串后面也有和首字母相同的“a”字符该怎么办呢？

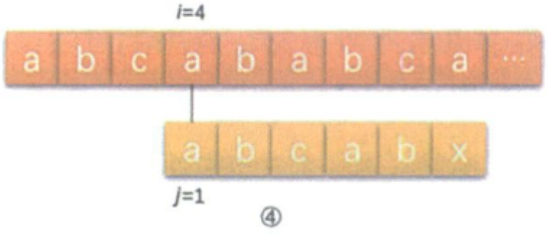
例如 S=“abcababca”，T=“abcabx”。

根据step.1中的优化，T串中的首位“a”与第二位、第三位均不相同，所以②③步是可以直接优化掉的。

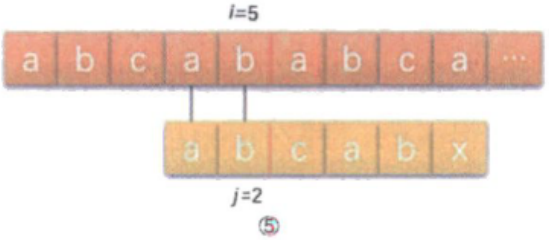


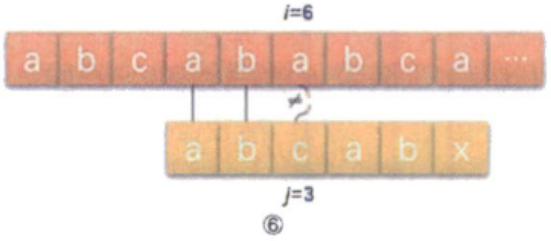






然后到了④，因为T串的首位“a”与T的第四位“a”相等，第二位的“b”与第五位的“b”相等，这是在①中已经与主串的相应位置判断过的，所以④⑤两步也可以优化掉。





也就是说，在子串中存在与首字符相等的字符也能优化掉一些步骤，虽然没有step.1中优化得多。

step.3

根据step.1 & step.2中的例子，我们发现在①中的i值，也就是主串当前进行判断位置的下标是6，经过②~⑤再到了⑥，i才回到了6。

也就是说，在朴素算法中，主串的i值是不断回溯的。有回溯就有优化的空间。

不想回溯i值也就是i值不能变小，i值就应该是随着匹配的进度增大的，那么需要优化的地方就是j值的变化。

在step.1中我们反复提到了 *“T串的首字母“a”与T中后面的字符均不相等”* ，在step.2中也许我们做的变化只是将T中不相等的区域缩小了一点。那么我们就可以发现，如果子串中有相等的字符，j值就会发生变化（j值的变化其实和主串没有什么关系，应该就是取决于子串的结构有没有重复），这应该就是优化效率的关键。下面对j继续探索

例如T="abcdex"，这个串当中没有任何重复的字符。所以在①到⑥之后，j由6变成了1。

而在step.2中的例子T=“abcabx”中，前缀的“ab”与最后的“x”之前的串的后缀“ab”是相等的，所以j由6变成了3。由此我们得出了规律—**j值的大小取决于当前字符之前的串的前后缀的相似度**。（abcabx中就是x之前的串，abcab的前后缀的相似度）。

那么想要优化字符串的查找，我们就要先对这个字符串进行分析，找出这个j值，以提高查找效率。

我们把T串的各个位置的j值的变化定义为一个数组next，那么next的长度就是T串的长度，于是我们可以得到下面的函数定义：

$$next[j]=\begin{cases}0, \text{当} j=1 \text{时} \\ Max\left\{k \mid 1<k<j, \text{且}' P_1 \cdots P'_{k-1}=' P_{j-k+1} \cdots P'_{j-1}\right\}, \text{当此集合不为空时} \\ 1, \text{其他情况}\end{cases}$$

6.5.2 next数组值的推导

以T="abcabx”为例

j	123456
模式串T	abcabx
next[j]	011123

1. 当j=1时，next[j] = 0
2. 当j=2时，j由1到j-1就只有字符“a”，属于其他情况，next[2]=1
3. 当j=3时，j由1到j-1的串是“ab”，显然“a”与“b”不相等，属于其他情况，next[2]=1
4. j=4同理，next[4]=1
5. 当j=5，此时j由j-1的串是“abca”，前缀字符“a”与后缀字符“a”相等，由$' P_1 \cdots P'_{k-1}=' P_{j-k+1} \cdots P'_{j-1}$得到$P_1=P_4$ 得到k=2
6. 当j=6时，j由1到j-1的串是“abcab”，由于前缀字符“ab”与后缀“ab”相等，所以next[6]=3

由此我们可以根据经验得：如果前后缀只有一个字符相等，k值就是2，两个字符相等k就是3，**前后缀n个字符相等k=n+1**

（3）T=“ababaaaba”（如下表所示）

j	123456789
模式串T	ababaaaba
next[j]	011234223

① 当j=1时，next[1]=0。

② 当j=2时，同上next[2]=1。

③ 当j=3时，同上next[3]=1。

④ 当j=4时，j由1到j-1的串是“ab*a*”，前缀字符“a”与后缀字符“a”相等，next[4]=2。

⑤ 当j=5时，j由1到j-1的串是“ab*ab*”，由于前缀字符“ab”与后缀“ab”相等，所以next[5]=3。

⑥ 当j=6时，j由1到j-1的串是“ab*aba*”，由于前缀字符“aba”与后缀“aba”相等，所以next[6]=4。

⑦ 当j=7时，j由1到j-1的串是“ababa*a*”，由于前缀字符“ab”与后缀“aa”并不相等，只有“a”相等，所以next[7]=2。

⑧ 当j=8时，j由1到j-1的串是“ababaa*a*”，只有“a”相等，所以next[8]=2。

⑨ 当j=9时，j由1到j-1的串是“ababaa*ab*”，由于前缀字符“ab”与后缀“ab”相等，所以next[9]=3。

（4）T=“aaaaaaaab”（如下表所示）

j	123456789
模式串T	aaaaaaaab
next[j]	012345678

① 当j=1时，next[1]=0。

② 当j=2时，同上next[2]=1。

③ 当j=3时，j由1到j-1的串是“a*a*”，前缀字符“a”与后缀字符“a”相等，next[3]=2。

④ 当j=4时，j由1到j-1的串是“a*aa*”，由于前缀字符“aa”与后缀“aa”相等，所以next[4]=3。

⑤ ……

⑥ 当j=9时，j由1到j-1的串是“*aaaaaaa*a”，由于前缀字符“aaaaaaa”与后缀“aaaaaaa”相等，所以next[9]=8。

6.5.3 KMP算法的实现

一个及其简洁的版本

```cpp
#include <iostream>

using namespace std;

const int N = 100010;
const int M = 1000010;

int n, m;
int ne[N];
char s[M], p[N];

int main ()
{
    cin >> n >> p + 1 >> m >> s + 1;        //字符数组下标从1开始, 而不是从0 开始!!!

    // 获取next数组
    // i从1开始时因为下标从1开始, ne[0]没有意义, 而ne[1]只有一个字母, 所以ne[1] == 0
    for (int i = 2, j = 0; i <= n; i++)
    {
        while (j != 0 && p[i] != p[j + 1]) j = ne[j];
        if (p[i] == p[j + 1]) j++;
        ne[i] = j;
    }

    //进行KMP匹配
    for (int i = 1, j = 0; i <= m; i++)
    {
        while (j != 0 && s[i] != p[j + 1]) j = ne[j];
        if (s[i] == p[j + 1]) j ++;
        if (j == n)
        {
            printf("%d", i - n);
            j = ne[j];
        }
    }

    return 0;
}
```