Chapter 3

第3章 字符串处理技巧题

HDOJ2895-Edit distance

【题目大意】

给定一个字符串，编辑脚本是一组指令，用于将其转换为另一个字符串。在编辑脚本中，有 4 种指令。

添加('a')：输出一个字符。该指令不使用源字符串中的任何字符。

删除('d')：删除一个字符。从源字符串中消耗一个字符，不输出任何内容。

修改('m')：修改一个字符。从源字符串中消耗一个字符，并输出一个字符。

复制('c')：复制一个字符。从源字符串中消耗一个字符，并输出相同的字符。

现在，我们定义一个最短的编辑脚本，就是可以最小化添加和删除的总数。

给定两个字符串，生成一个最短的编辑脚本，将第一个更改为第二个。

输入

输入由两个字符串组成，分别在不同的行中。

字符串只包含字母、数字、字符。每个字符串的长度在 1～10 000。

输出

输出是一个最短的编辑脚本。每行都是一条指令，由指令的单字母代码(a，d，m 或 c)给出，后跟一个空格，后跟写入的字符(如果指令是删除，则是删除)。

在指定测试例中，必须生成最短的编辑脚本，并且必须按照 a，d，m，c 的顺序进行排序，所以只有一个答案。

【算法分析】

给出一个原始字符串和一个目标字符串，共有增加、删除、移动、复制 4 种编辑方式，求

字典序最小的编辑策略,使得原始字符串变为目标字符串。

基于贪心算法的思想。实际上产生距离的只有 add 和 delete,因为 modify 是不算距离的,那么两个字符串之间的最小编辑距离就一定是它们之间的长度差,可以先通过 add 或者 delete,然后再通过 modify 完成。题目最后还要求排序,就是要保证使用这种方法。

示例:

abcde 和 xabzdey 的长度差为 2,那么先 add x,a,然后 modify b,z,d,e,y。

xabzdey 和 abcde 的长度差为 2,那么先 delete x,a,然后 modify a,b,c,d,e。

【程序核心代码】

```
//两个字符串
string a,b;
while(cin>>a>>b)
{
    int index=0;
    //需要更改的字符
    string modify = "";
    //源字符串,删除操作
    if(a.size()>b.size())
    {
        while(a.size()-index>b.size())
            cout<<"d "<<a[index++]<<endl;
        //剩下的字符,通过 modify 实现
        for(int i = 0; index+i < a.size(); i++)
            modify += b[i];
    }
    //源字符串短,增加操作
    else if(a.size()<b.size())
    {
        while(a.size()+index<b.size())
            cout<<"a "<<b[index++]<<endl;
        //剩下的字符,通过 modify 实现
        for(int i = 0; index+i < b.size(); i++)
            modify += b[index+i];
    }
    //两个串相等,直接通过 modify 实现
    else
    {
        for(int i = 0; i<a.size(); i++)
            modify += b[i];
    }
```

```
    //输出通过 modify 实现的操作
    for(int i=0; i<modify.size(); i++)
        cout<<"m "<<modify[i]<<endl;
}
```

算法实现源代码：hdu2895.cpp。

ZJU1014-Operand

【题目大意】

Maple 在一所大学里讲授数学。他发明了一个函数，用于从一个表达式中取出指定的操作数。该函数命名为 op(i,e)，描述如下。

根据表达式中优先级最低的运算符，可以把表达式 e 分成若干个子表达式。例如，把“$a \cdot b+b \cdot c+c \cdot d$”分成 3 个子表达式“$a \cdot b$”、“$b \cdot c$”和“$c \cdot d$”，因为运算符“+”的优先级是最低的。该函数的目的是提取第 i 个子表达式。如上例，op(2, e)=$b \cdot c$。

如果把子表达式也视为主表达式，那么它也可以细分。显然，这是一个递归的过程。下面是一些更复杂的例子。

```
Let p := a^b * c+(d * c)^f * z+b
op(1,op(1,op(2,p)))=(d * c)
op(1,op(1,op(1,op(2,p)))) = d * c
op(2,op(2,p)) = z
op(3,p) = b
op(1,op(3,p)) = b
```

当表达式很长而且很复杂时，Maple 教授就懒得亲自动手了，打算交给计算机完成。当然，没有你的程序，计算机也无法工作。

输入

有多组测试例。以一个“*”表示输入数据全部结束。每组测试例分为两个部分：第一部分是表达式，第二部分是根据表达式进行计算的问题。

每个测试例的第一行是表达式，包括表达式名称、“:=”和表达式的内容。表达式名称是一个小写字母，其内容由小写字母与“+”“(”“)”“*”和“^”组成。例如，p:=a^b*c+(d*c)^f*z+b 是一个合法的表达式。操作符的优先级是：“(”和“)”优先级最高，“^”次之，接下来是“*”，优先级最低的是“+”。

测试例的第二行是一个整数 n，表示对上述表达式有 n 个问题。接下来有 n 行整数，每行描述一个问题。例如，一个问题是 3 个整数“2 1 1”，表示函数 op(1, op(1, op(2, e)))。按下列步骤计算表达式：首先，根据第一个数字 2，划分表达式并把第二个子表达式取出来；然后，根据第二个数字 1，划分该子表达式并把第一个子表达式取出来；最后，根据第三个数

字1,划分这个子表达式并得到结果。

输出

对每个测试例,在第一行输出表达式名称和一个冒号,然后每行输出一个结果,输出方式见样例。

可以假定每个表达式和函数都是合法的。

每个测试例之间有一个空行。

【算法分析】

1. 样例分析

第一个样例表达式:$p:=a\wedge b*c+(d*c)\wedge f*z+b$。

第一个问题2 1 1。

如题目所述,首先,根据第一个数字2得到$(d*c)\wedge f*z$;然后,根据第二个数字1得到$(d*c)\wedge f$;最后,根据第三个数字1得到结果$(d*c)$。

2. 数据结构

```
int count;                          //问题的个数
string src;                         //表达式
char ename;                         //表达式名称
```

题目中的每个问题,提取子表达式的次数是未知的,而且要逆序输出操作步骤,所以使用一般的数组存放问题比较困难。利用C++的标准模板库,如stack()容器、vector()容器或者list()容器都能很好地解决这个问题,本题使用list()容器。

```
list<int>question;
```

读取问题时,依次进入队列;处理问题时,依次从队列中弹出,刚好是逆序操作。

```
int n;
string ops = src;                   //只处理表达式的副本
question.clear();
while(cin.peek() != '\n')           //问题没有结束
{
    cin >> n;                       //读取问题内容
    question.push_back(n);          //构建问题队列
    ops = Operand(n,ops);           //提取子表达式
}
```

其中用到cin.peak()方法,该方法读取下一个字符,但不移动缓冲区指针,通过判断'\n'我们就知道这一行是不是读完了。使用C语言的getchar()也可以做到,该函数读取下一个字符,而下一个字符是空格或者回车,不影响结果。

3. 提取子表达式

由函数 string Operand(int n, string s)完成,其中形参:

```
int n;                                    //提取第 n 个子表达式
string s                                  //待提取的表达式
```

(1) 优先级的实现。

将运算符放到数组中:

```
char optSign[3]={'+','*','^'};
```

由优先级最低的运算符开始处理。

题目保证问题是合法的,所以从优先级最低的运算符开始处理必定能够得到答案。

(2) 括号的处理。

只要有运算符,括号的优先级就比该运算符高,因此括号里面的内容就得原样写进答案。

```
if(s[j] == '(')
{
    bracket++;                            //左括号计数
    continue;
}
else if(s[j] == ')')
{
    bracket--;                            //右括号出现
    continue;
}
else if(bracket) continue;                //还有左括号
```

变量 bracket 是统计左括号的。

(3) 本身运算符的处理。

构建的答案中,如果不是最后一个子表达式,里面会包含当前的运算符,如样例第一个问题的第一个操作,answer=$(d*c)^f*z+$,需要去掉它:

```
return answer.substr(0,answer.length()-1);
```

(4) 子表达式中根本就没有运算符。

如表达式:“(((x)))”,或者“good”。

有括号时,去掉一层括号;没有括号时,原样返回。

4. 输出结果

如果问题中的操作要求已经保存在队列 question 中,输出就比较方便了。

依次从队列中弹出,刚好是逆序,就构成了答案的左边。

```
while(!question.empty())
{
    n = question.back();
    question.pop_back();
    cout<<"op("<<n<<",";
}
```

【程序代码】

```
char optSign[3]={'+','*','^'};

//提取第 n 个子表达式
string Operand(int n, string s)
{
    int count;                          //子表达式计数
    int bracket;                        //左括号的个数
    bool found;                         //是否找到了结果
    string answer;                      //结果
    //分别对应"+","*"和"^"运算符
    for(int i = 0; i < 3; i++)
    {
        count = 1;                      //子表达式计数
        bracket = 0;                    //左括号计数
        found = false;                  //找到结果?
        answer = "";                    //结果
        //对应每一个字符
        for(int j = 0; j < s.length(); j++)
        {
            //构造答案
            answer += s[j];
            //越过括号
            if(s[j] == '(')
            {
                bracket++;
                continue;
            }
            else if(s[j] == ')')
            {
                bracket--;
                continue;
            }
```

```
            else if(bracket) continue;
            //越过当前运算符
            else if(s[j] != optSign[i]) continue;
            else
            {
                //是第 n 个子表达式,返回结果(去掉最后面的运算符)
                if(count == n)
                    return answer.substr(0,answer.length()-1);
                else
                {
                    //不是第 n 个子表达式,处理下一个子表达式
                    found = true;
                    answer = "";
                    count++;
                }
            }
        }
        //结果的最后面不是运算符,直接返回
        if(found) return answer;
    }
    //子表达式没有运算符
    //有括号
    if(s[0] == '(') return s.substr(1,s.length()-2);
    //没有括号
    else return s;
}

int main()
{
    int count;                          //问题的个数
    string src;                         //表达式
    char ename;                         //表达式名称
    bool lineFlag = true;               //换行标志
    //问题描述
    list<int>question;
    while(cin>>src && src != "*")
    {
        if(lineFlag) lineFlag = false;
        else cout<<endl;
        //得到表达式名称
        ename = src[0];
```

```
        //去掉名称后的表达式
        src = src.substr(3);
        cout<<"Expression "<<ename<<":"<<endl;
        //读取问题的个数
        cin>>count;
        cin.ignore();
        //处理每一个问题
        while(count--)
        {
            int n;
            //只处理表达式的副本
            string ops = src;
            question.clear();
            //问题没有结束
            while(cin.peek()!='\n')
            {
                //读取问题内容
                cin >> n;
                //构建问题队列
                question.push_back(n);
                //提取子表达式
                ops = Operand(n,ops);
            }
            cin.ignore();
            //问题中操作的次数
            int bkc = question.size();
            //输出答案的左边,逆序输出
            while(!question.empty())
            {
                n = question.back();
                question.pop_back();
                cout<<"op("<<n<<",";
            }
            //输出问题名称
            cout<<ename;
            //与左边对应的右括号
            for(int i = 1; i <= bkc; i++)
                cout<<")";
            cout<<"="<<ops<<endl;   //输出答案
        }
    }
```

```
    return 0;
}
```

算法实现源代码：zju1014.cpp。

ZJU1044-Index Generation

【题目大意】

许多写实文学和参考书都有一个索引，以帮助读者找到专业术语和概念在书中的引用。每个索引条目都有一个主条目，后面没有也可能有多个辅条目，辅条目以‘+’开始。每个条目都有一个引用页码的列表，如果至少有一个辅条目，则主条目就没有引用页码（如上例中的条目 mango）。主条目和辅条目都是排序的，排序时区分字母大小写。每个条目的参考页码是升序的，且不包含重复页码（如果在同一页里有两个或者更多相同的条目，就会有重复页码）。

编程任务：读取一段文档，里面有索引信息，然后建立索引。文档有一行或者多行 ASCII 文本。页码从第 1 页开始。字符‘&’表示新的一页开始（即在当前页码上加 1）。索引条目是有标记的，其格式如下。

```
{text%primary$secondary}
```

这里，text 是要索引的文本，primary 是可选的主条目，secondary 是辅条目。“% primary”和“$ secondary”是可选项，如果同时出现，必须按给定的顺序。如果有 primary，就作为主条目，否则就用 text 作为主条目。如果有 secondary，就必须为辅条目增加参考页码。也可能一个标记里 primary 和 secondary 里面都没有参考页码。下面的例题是标记的 4 种可能的情况，上例就是相应的 4 个条目。

输入

输入有多组文档，如果一行中只有“* *”，标志输入结束。文档中隐含了从 1 开始的数字编号。每个文档都包含一行或多行文本，当一行为‘*’时，标志该文档结束。每行文本不超过 79 个字符，不包含回车符。

输出

对文档 i，输出一行“DOCUMENT i”，接着严格按照样例的格式输出排序的索引。

注意

文档至少包含 100 个标记，至少有 20 个主条目。

一个主条目至少有 5 个辅条目。

一个条目至少有 10 个不同的参考页码（不包含重复页码）。

字符‘&’不会出现在标记中，它在文档中至少出现 500 次。

字符‘*’仅用来标志文档的结束或者输入的结束。

字符‘{’,‘}’,‘%’和‘$’只用来定义标记,不会出现在文本或者条目中。

一个标记可能在一行内,也可能跨行。标记内的每个回车符必须转换为一个空格。

标记内的空格(包括由回车符转换来的空格)像其他字符一样,通常是文本/条目的一部分。但是,在‘{’之后,‘}’之前,或者与‘%’和‘$’相邻的任何空格,都必须忽略不计。

标记的总长度,即‘{’和‘}’之间的字符,包括由回车符转换来的空格,不超过 79 个字符。

【算法分析】

1. 数据结构

当前文档的页码,从 1 开始编号:

```
int page;
```

当前文档中的字符:

```
char ch;
```

当前文档中的标记符号:

```
char token;
```

是否以有效字符串开始,如果是空格标记符时,则是 false,否则是 true。

```
bool lookahead = false;
```

条目的数据结构:

```
struct Entry
{
    string primary;
    string secondary;
    int page;
    Entry (string p, string s)
        : primary(p), secondary(s), page(::page) {};
};
```

其中,primary 是主条目,secondary 是辅条目。注意,初始化中的 page(::page),实参 page 是全局变量 page(前面有域操作符::)。

所有的条目都存放在数组 entry 中,其类型是 STL 容器中的 vector。

```
vector<Entry> entry;
```

2. 判断标记

判断标记是本题最重要的任务,标记判断失误,其结果就面目全非。根据当前字符的情况,是否是标记,有时还要根据下一个字符的情况才能做出判断。

具体判断的情况较多，不宜在此列举，可看程序中的注释。判断函数为

```
char next_token()
```

3. 增加条目

增加条目是本题的目标，由函数 void add_entry()实现。

有 3 种情况。

(1) 标记中没有 primary，就以 text 作为主条目。

(2) 标记中有'%'，后面就是 primary，则构造 primary。

(3) 标记中有'$'，后面就是 secondary，则构造 secondary。

最后将构造的一个条目保存到数组中。

```
entry.push_back(Entry(primary, secondary));
```

注意，页码就是全局变量 page 的值。

4. 将条目排序

首先要区分字典序与字符串序：字典序是不考虑字符的大小写而排序的，字符串序是考虑字符的大小写而排序的。以字符串为例：

Most nonfiction and reference books have an index to help

下面是字典序，注意字符串排序不区分大小写。

an and books have help index **Most** nonfiction reference to

下面是字符串序，注意大写字符串在最前面。

Most an and books have help index nonfiction reference to

本题是字典序，条目的排序不区分大小写。

【程序代码】

```
//当前文档的页码，从 1 开始编号
int page;
//当前文档中的字符
char ch;
//当前文档中的标记符号
char token;
//是否以有效字符串开始
bool lookahead = false;

const int EndOfDocument = -1;
const int EndOfFile = -2;

//条目的数据结构
struct Entry
```

```
{
    string primary;                    //主条目
    string secondary;                  //辅条目
    int page;                          //页码
    //初始化
    Entry(string p, string s)
        : primary(p), secondary(s),page(::page) {};
};

vector<Entry>entry;                    //存放条目的数组

//实现字典序的排序算法
int string_compare(const string& s, const string& t)
{
    int m = s.length();
    int n = t.length();
    //取短字符串的长度
    int k = m <=n ? m : n;
    //先比字母序
    for (int i = 0; i < k; ++i)
    {
        //不区分大小写,比较时都转换为大写
        int a = toupper(s[i]);
        int b = toupper(t[i]);
        if (a != b) return a-b;
    }
    //字母序一样时,比字符串的长短
    return m == n ?0 : m<n ?-1 : 1;
}

//条目排序时的比较算法
bool less_than(const Entry& a, const Entry& b)
{
    //按主条目升序
    int cmp = string_compare(a.primary, b.primary);
    if (cmp < 0) return true;
    if (cmp > 0) return false;
    //按辅条目升序
    cmp = string_compare(a.secondary, b.secondary);
    if (cmp < 0) return true;
    if (cmp > 0) return false;
```

```
    //按页码升序
    return a.page < b.page;
}

//读取下一个字符,并置 lookahead 为 false
inline char next_char()
{
    if (lookahead) lookahead = false;
    else ch = cin.get();
    return ch;
}

//判断标记
char next_token()
{
    //当前字符
    switch (next_char())
    {
    //是输入结束,还是该文档结束?
    case '*':
        token = (next_char() == '*') ?EndOfFile : EndOfDocument;
        break;
    case ' ':
    case '\n':
        //遇空格或者回车时,再读取下一个字符
        next_char();
        //是标记
        if (ch == '%' || ch == '$' || ch == '}')
            token = ch;
        else {
            token = ' ';                //是普通字符
            lookahead = true;
            break;
    }
    //当前字符是标记
    case '{': case '%': case '$':
        token = ch;
        //忽略相邻的空格
        lookahead =! isspace(next_char());
        break;
    default:
```

```
        token = ch;
    }
    return token;
}

//判断标记
inline bool is_delimiter(char t)
{
    return t == '%' || t == '$' || t == '}';
}

//增加条目
void add_entry()
{
    string primary, secondary;
    //首先把 text 当作 primary
    while (! is_delimiter(next_token()))
        primary += token;
    //标记中有'%',后面就是 primary,则重新构造
    if (token == '%')
    {
        primary.erase();
        while (! is_delimiter(next_token()))
            primary += token;
    }
    //标记中有'$',后面就是 secondary
    if (token == '$')
        while (! is_delimiter(next_token()))
            secondary += token;
    //增加条目,注意页码就是全局变量 page 的值
    entry.push_back(Entry(primary, secondary));
}

int main()
{
    //变量 document 为文档编号
    for (int document = 1; ; ++document)
    {
        if (next_token() == EndOfFile) break;
        cout <<"DOCUMENT " <<document;
        page = 1;                          //从 1 开始编号
```

```
        entry.clear();                  //清空数组
        //第 0 个元素未用
        entry.push_back(Entry("", ""));
        do {
            if (token == '&')           //新的一页
                ++page;
            else if (token == '{')      //条目的标记开始
                add_entry();            //增加条目
        } while (next_token() != EndOfDocument);
        //排序,排序算法为 less_than
        sort(entry.begin(), entry.end(), less_than);
        //输出条目,entry.size()为数组元素的个数
        for (int i = 1; i < entry.size(); ++i)
        {
            //同一主条目时,输出下面的辅助条目
            if (entry[i].primary == entry[i-1].primary)
                //同一辅条目时,输出下面的参考页码
                if (entry[i].secondary == entry[i-1].secondary)
                {
                    //忽略相同的页码
                    if (entry[i].page != entry[i-1].page)
                        cout <<", " <<entry[i].page;
                }
                else                    //新的辅条目
                    cout<<"\n+"<<entry[i].secondary
                            <<", "<<entry[i].page;
            else
            {    //新的主条目
                cout <<'\n' <<entry[i].primary;
                //没有辅条目,直接输出参考页码
                if (entry[i].secondary == "")
                    cout <<", " <<entry[i].page;
                //有辅条目
                else
                    cout<<"\n+"<<entry[i].secondary
                            <<", "<<entry[i].page;
            }
        }
        cout <<endl;
    }
    return 0;
}
```

算法实现源代码：zju1044.cpp。

ZJU1046-Double Vision

【题目大意】

DoubleVision 公司设计出了人和机器都能很方便阅读的墨水和字体。他们设计的字体是一个矩形网格，如下所示

```
.o.   .o.   oo.   oo.   o.o
o.o   .o.   ..o   ..o   o.o
o.o   .o.   .o.   oo.   ooo
o.o   .o.   o..   ..o   ..o
.o.   .o.   ooo   oo.   ..o
```

是前 5 个数字最简单的 5×3 网格。

墨水看起来像普通的黑墨水，但是在表层下面 DoubleVision 公司加了一层特殊的高分子材料，用于红外线扫描仪的检测。人可以看见黑墨水，但看不见高分子材料，机器可以识别高分子材料，但不能识别墨水。问题是高分子材料比墨水贵得多，所以 DoubleVision 公司希望尽可能少地使用高分子材料。他们发现对多种字体，使用不超过两个像素就可以唯一区分每个符号。对每个符号，只要在一两个像素上添加高分子材料，成本将显著降低，又能确保扫描仪精度为 100%。如

```
.#.   .o.   #o.   oo.   o.#
#.o   .#.   ..o   ..o   o.o
o.o   .o.   .o.   #o.   ooo
o.o   .o.   #..   ..o   ..o
.o.   .o.   ooo   #o.   ..o
```

所示的字体就具有这个特性，能唯一地标识每个字母的像素都突出显示为'#'(也可以是其他方式)。

编程任务：判断一个给定的字体是否具有这种特性。如果有，则突出显示其像素。

输入

输入有多组测试例，当一行是“0 0 0”(3 个 0)时输入结束。对每个测试例，第一行是 3 个正整数 n，r 和 c，中间有一个空格，n 是字体中符号的个数，r 是网格的行数，c 是网格的列数。接下来 r 行是每个符号的图像，其格式如样例所示：点‘.’表示网格中空的地方，小写字母‘o’表示一个像素，相邻的网格之间有一个空格。每行不超过 79 个字符(不包括回车符)，r 最多为 10。测试例编号从 1 开始。

输出

对测试例 i，输出一行“Test i”。确定每个符号能否用一两个像素唯一地标识，如果不

能，则输出一行“impossible”；否则输出格式与输入格式相同。当然，每个符号的识别像素已替换成‘＃’。

通常唯一地标识一个符号，可以使用不同的像素。为确保输出是唯一的，可增加下列定义和规则：当比较两个像素时，最顶端-最左边的像素是指最接近网格顶端的一个像素；当比较网格一行上两个像素时，最顶端-最左边的像素是指最接近网格左边的一个像素。

如果只需要一个像素，则突出显示最顶端-最左边的像素，此时绝不能突出显示两个像素。如果需要两个像素，则突出显示最顶端-最左边的一对像素。如果有多对相同的最顶端-最左边的像素，则突出显示其他具有最顶端-最左边的像素。

【算法分析】

1. 样例分析

以样例 1 为例：第一个符号，它的 3 个像素在其他符号里面都有，所以不能用一个像素表示。最左边的两个像素在其他符号里面没有，则突出显示：

```
#o  oo  .o
#.  .o  o.
```

第二个符号，它右下面的那个像素是唯一的，突出显示：

```
#o  oo  .o
#.  .#  o.
```

第三个符号，它的两个像素在其他符号里面都有，无法唯一标识：

```
#o  oo  .o
#.  .#  o.
```

所以，整个字体是'impossible'。

以样例 2 为例：第一个符号，它的 3 个像素在其他符号里面都有，所以不能用一个像素表示。最左边的两个像素在其他符号里没有，则突出显示：

```
#o  oo  .o
#.  .o  o.
```

第二个符号，它左上角和右下角的一对像素是唯一的，突出显示：

```
#o  #o  .o
#.  .#  oo
```

第三个符号，它下面的一对像素是唯一的，突出显示：

```
#o  #o  .o
#.  .#  ##
```

每个符号都能唯一地标识，且符合原则：最顶端-最左边的像素。

2. 数据结构

一行中最多的字符数用常变量表示：

```
const int maxCols = 79;
```

符号的个数：

```
int symbols;
```

每个符号网格的行列数：

```
int rows;
int cols;
```

字体用一个二维数组表示(最多为 10 行)：

```
char picture[10][maxCols + 1];
```

3. 该符号中的一个像素是否唯一

使用函数判断：

```
bool match1(int g, int r, int c)
```

其中形参 g 是符号的序号,从 0 开始编号;(r，c)是该符号中的网格坐标。只要在其他符号中查找,与其坐标相同的点有没有像素？如果有,则不是唯一的。

```
for (int h = 0; h<symbols; ++h)
    if (h != g && notDot(h,r,c))
        return false;
```

4. 该符号中的一对像素是否唯一

使用函数判断：

```
bool match2(int g, int r, int c, int r2, int c2)
```

其中,形参(g，r，c)的含义同上,(r2，c2)是该符号中另一个像素的网格坐标。

如果该符号的每个像素都不是唯一的,则查找该符号的每对像素是否唯一。

```
for (int h = 0; h<symbols; ++h)
    if (h != g && notDot(h,r,c) && notDot(h,r2,c2))
        return false;
```

这种情况下只查找其他符号里相同的一对点是否有像素即可。

5. 判断一个符号能否被唯一标识

使用函数判断：

```
bool solve(int g)
```

其中，形参 g 是符号的序号，从 0 开始编号。

分两步实现：

(1) 判断该符号能否用一个像素突出显示。如果能，则突出显示该像素，并返回 true；如果不能，则进入下一步。

(2) 判断该符号能否用一对像素突出显示。如果能，则突出显示这对像素，并返回 true；如果不能，则返回 false。

6. 判断一个字体能否被唯一标识

如果每个符号都能够唯一标识，则输出结果；只要有一个符号不能被唯一标识，就输出'impossible'。

7. 如何保证突出显示的像素就是最顶端-最左边的像素

确定搜索的顺序，是从每个符号的左上角开始搜索，只要判断出该符号能够被唯一标识，函数 bool solve(int g)就返回 true。

【程序代码】

```
const int maxCols = 79;
//符号的个数
int symbols;
//每个符号网格的行数
int rows;
//每个符号网格的列数
int cols;
//字体 font
char picture[10][maxCols + 1];

//第 g 个符号中,(r, c)是不是像素
bool notDot(int g, int r, int c)
{
    //计算第 g 个符号中的列坐标
    int k = g * (cols +1) +c;
    if (picture[r][k] != '.') return true;
    else return false;
}

//第 g 个符号中的一个像素是不是唯一的
bool match1(int g, int r, int c)
{
    //在所有其他符号中查找
    for (int h=0; h <symbols; ++h)
        //只要找到一个就不是唯一的
```

```
        if (h != g && notDot(h,r,c))
            return false;
    return true;
}
//第 g 个符号中的一对像素(r, c)和(r2, c2)是不是唯一的?算法同上
bool match2(int g, int r, int c, int r2, int c2)
{
    for (int h = 0; h <symbols; ++h)
        if (h != g && notDot(h,r,c) && notDot(h,r2,c2))
            return false;
    return true;
}

//判断一个符号能否被唯一标识?g 是符号的序号
bool solve(int g)
{
    //从该符号的左上角开始判断,确保突出显示的像素就是最顶端-最左边的像素
    //判断该符号能否用一个像素突出显示
    for (int r = 0; r <rows; ++r)
        for (int c = 0; c <cols; ++c)
        {
            if (notDot(g,r,c) && match1(g,r,c))
            {
                //将符号的坐标换成字体的坐标
                int k = g * (cols +1) +c;
                picture[r][k] = '#';  //突出显示
                return true;
            }
        }
    //判断该符号能否用一对像素突出显示
    for (int r = 0; r <rows; ++r)
        for (int c = 0; c < cols; ++c)
        {
            if (notDot(g,r,c))
            {
                //在同一行的右边查找另一个像素(r, c2)
                for (int c2 = c+1; c2 < cols; ++c2)
                    if (notDot(g,r,c2) && match2(g,r,c,r,c2))
                    {    //(r, c)和(r, c2)是唯一的一对像素
                        int k = g * (cols + 1) +c;
                        picture[r][k] = '#';
```