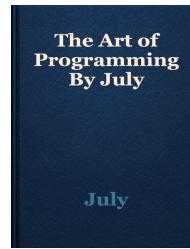


The Art of Programming By July

本书是July和他伙伴们的《程序员编程艺术》的电子书

《程序员编程艺术：面试和算法心得》



目录

第一部分、面试心得

- 第一章、字符魅影
 - 1.0: 本章导读
 - 1.1: 字符的移动
 - 1.2: 字符串包含
 - 1.4: 字符串转换成整数
 - 1.5: 回文判断
 - 1.6: 最长回文子串
 - 1.7: 本章习题
- 第二章、数组玄妙
 - 2.0: 本章导读
 - 2.1: 寻找最小的 k 个数
 - 2.2: 求给定下标区间内的第 K 小元素
 - 2.3: 求解 500 万以内的亲和数
 - 2.4: 寻找和为定值的两个数
 - 2.5: 寻找和为定值的多个数
 - 2.6: 求连续子数组的最大和
 - 2.7: 荷兰国旗问题
 - 2.8: 矩阵相乘之Strassen算法
 - 2.9: K 个最小和
 - 2.10: 本章习题
- 第三章、查找排序
 - 3.1: 二分查找实现
 - 3.2: 杨氏矩阵查找
 - 3.3: 出现次数超过一半的数字
- 第四章、算法寻优
 - 4.1: 木块砌墙
 - 4.3: 完美洗牌问题
 - 4.4: 最近公共祖先LCA问题
 - 4.5: 打印螺旋矩阵
- 第五章、动态规划
 - 5.0: 本章导读
 - 5.1: 最长公共子序列(LCS)问题
 - 5.2: 最大连续乘积子串
 - 5.3: 字符串编辑距离
 - 5.4: 格子取数
 - 5.5: 交替字符串
 - 5.6: 最长递增子序列
 - 5.10: 本章习题
- 第六章、细节实现
 - 6.2: 等概率随机取元素
 - 6.5: 全排列
 - 6.6: 跳台阶
 - 6.7: 奇偶排序
 - 6.11: 最小操作数
 - 6.12: 本章习题
- 第七章、系统设计
 - 7.1: 一致性哈希算法
 - 7.2: 最短摘要的生成

- 7.5: 搜索智能提示 suggestion
- 7.6: 附近地点搜索

第二部分、算法心得

- 第八章、树型结构
 - 8.0: 本章导读
 - 8.1: Trie树
 - 8.2: 后缀树
 - 8.3: 红黑树
 - 8.4: B树
 - 8.5: R树
- 第九章、海量数据
 - 9.0: 本章导读
 - 9.1: 关联式容器
 - 9.3: 分而治之
 - 9.4: 双层桶划分
 - 9.5: Bitmap
 - 9.6: Bloom filter
 - 9.7: Trie树
 - 9.8: 数据库
 - 9.9: 倒排索引
 - 9.10: 外排序
 - 9.11: Mapreduce
 - 9.12: 本章习题
- 第十章、图像处理
 - 10.1.1: sift算法的编译
 - 10.1.2: sift算法的C实现、上
 - 10.1.3: sift算法的C实现、下
 - 10.2.1: 傅里叶变换算法、上
 - 10.2.2: 傅里叶变换算法、下
- 第十一章、机器学习
 - 11.1: 支持向量机
 - 11.2: K 近邻算法

第一章导读

一般来说，笔试面试中常考的知识点有：

- 数据结构：字符串、链表、数组、堆、哈希表、树（Trie树、后缀树、红黑树、B树、R树）、图（遍历：BFS、DFS、Dijkstra）；
- 算法：基于各个数据结构的查找（二分、二叉树）、排序、遍历，分治、递归、回溯，贪心算法、动态规划、海量数据，外加字符串匹配和资源调优；
- 数学：排列组合概率；
- 操作系统、网络协议、数据库等等。

本第一章，咱们便来看看基于字符串的面试题。

[prev](#) | [next](#)

[Back to home](#)

Generated by [mdtogh](#)

字符的移动

题目描述

给定一个字符串，要求把字符串前面的若干个字符移动到字符串的尾部，如把字符串“abcdef”前面的2个字符“ab”移动到字符串的尾部，即变成“cdefab”。请写一个函数完成此功能，要求对长度为n的字符串操作的时间复杂度为 O(n)，空间复杂度为 O(1)。

分析与解法

解法一、暴力移位法

初看此题，可能最先想到的方法是按照题目所要求的，把需要移动的字符一个一个的移动到字符串的尾部，如此我们可以实现一个函数 `LeftShiftOne(char *s, int n)`，以完成移动一个字符到字符串尾部的功能，代码如下所示：

```
void LeftShiftOne(char *s, int n)
{
    assert(s != NULL);
    char t = s[0]; //保存第一个字符
    for (int i = 1; i < n; i++)
    {
        s[i - 1] = s[i];
    }
    s[n - 1] = t;
}
```

因此，若要把字符串开头的m个字符移动到字符串的尾部，则可以如下操作：

```
void LeftShiftString(char *s, int n, int m)
{
    while (m--)
    {
        LeftShiftOne(s, n);
    }
}
```

下面，我们来分析下这个方法的时间复杂度和空间复杂度。

针对长度为n的字符串来说，假设需要移动m个字符到字符串的尾部，那么总共需要 $m \times n$ 次操作，同时设立一个变量保存第一个字符，如此，时间复杂度为 $O(m \times n)$ ，空间复杂度为 $O(1)$ ，空间复杂度符合题目要求，但时间复杂度不符合，所以，我们需要寻找其它更好的办法来降低时间复杂度。

解法二、三步反转法

对于这个问题，换一个角度：

将一个字符串分成X和Y两个部分，在每部分字符串上定义反转操作，如 X^T ，即把X的所有字符反转（如， $X="abc"$ ，那么 $X^T="cba"$ ），那么就得到下面的结论： $(X^T Y^T)^T = Y X$ 。显然这解决了字符串的反转问题。

例如字符串 $abcdef$ ，若要让 def 翻转到 abc 的前头，只要按上述3个步骤操作即可：

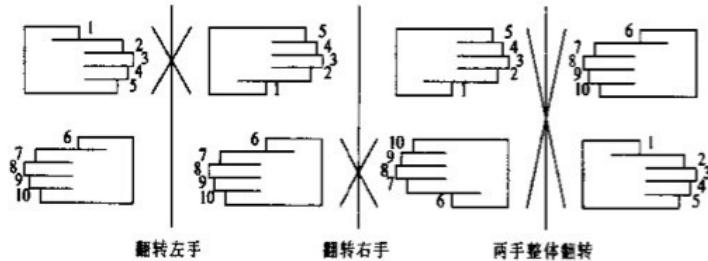
- 首先分为俩部分， $X:abc$, $Y:def$;
- 将X反转， $X \rightarrow X^T$ ，即得： $abc \rightarrow cba$ ；将Y反转， $Y \rightarrow Y^T$ ，即得： $def \rightarrow fed$ 。
- 反转上述得到的结果字符串 $X^T Y^T$ ，即字符串 $cba fed$ 的两部分（ cba 和 fed ）给予反转，得到： $cba fed \rightarrow def abc$ ，形式化表示为 $(X^T Y^T)^T = Y X$ ，这就实现了整个反转。

是不是一目了然了？

在《编程珠玑》上也有这样一个类似的问题，它的解法同本思路一致，如下图所示：

*. 可操作的证明方法-手摇法

如果要将一个具有10个元素（我们只有10个手指啊）的数组向上旋转5个位置，先让两只手的掌心正对自己，左右放在右手上面（其实两只手是同意平面上的），看下图：



代码则可以这么写：

```
//updated@caopengcs && July
//2014-1-6
void reverse(char *s,int from,int to)
{
    while (from < to)
    {
        char t = s[from];
        s[from++] = s[to];
        s[to--] = t;
    }
}

void LeftRotateString(char *s,int n,int m)
{
    m %= n; //若要左移动大于n位，那么和%n 是等价的
    reverse(s, 0, m - 1); //反转[0..m - 1]，套用到上面举的例子中，就是X->X^T，即 abc->cba
    reverse(s, m, n - 1); //反转[m..n - 1]，例如Y->Y^T，即 def->fed
    reverse(s, 0, n - 1); //反转[0..n - 1]，即如整个反转，(X^TY^T)^T=YX，即 cbafed->defabc。
}
```

举一反三

1、链表翻转。给出一个链表和一个数k，比如链表1→2→3→4→5→6，k=2，则翻转后2→1→6→5→4→3，若k=3，翻转后3→2→1→6→5→4，若k=4，翻转后4→3→2→1→6→5，用程序实现。

2、编写程序，在原字符串中把尾部m个字符移动到字符串的头部，要求：长度为n字符串操作时间复杂度为O(n),空间复杂度为O(1)。如：原字符串为"ilovebaofeng"，m=7，输出结果："baofengillove"。

3、单词翻转。输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变，句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。例如输入"I am a student."，则输出"student. a am I"。

[prev](#) | [next](#)

[Back to home](#)

字符串包含

题目描述

给定两个分别由字母组成的字符串A和字符串B，字符串B的长度较字符串A短。请问，如何最快地判断字符串B中所有字母是否都在字符串A里？也就是说判断字符串B是不是字符串A的真子集（为了简化，姑且认为两个集合都不是空集，即字符串都不为空）。且为了简单起见，我们规定输入的字符串只包含大写英文字母。

请实现函数bool StringContains(string &A,string &B)

比如，如果是下面两个字符串：

String 1: ABCDEFGHLMNOPQRS

String 2: DCGSRQPO

答案是true，即String2里的字母在String1里也都有，或者说String2是String1的真子集。

如果是下面两个字符串：

String 1: ABCDEFGHLMNOPQRS

String 2: DCGSRQPZ

答案是false，因为字符串String2里的Z字母不在字符串String1里。

分析与解法

题目描述虽长，但题意很明了，就是给定一长一短的两个字符串A, B，假设A长B短，要求判断B是否包含在字符串A中，即B?(-A)。

初看似乎简单，但实现起来并不轻松，且如果面试官步步紧逼，一个一个否决你能想到的方法，要你给出更好、最好的方案时，恐怕就要伤不少脑筋了。

解法一、暴力轮询

判断string2中的字符是否在string1中？

String 1: ABCDEFGHLMNOPQRS

String 2: DCGSRQPO

最直观也是最简单的思路是，针对string2中每一个字符，逐个与string1中每个字符比较，看它是否在String1中。

代码可如下编写：

```
//copyright@caopengcs 2013-01-14
bool StringContain(string &a,string &b)
{
    for (int i = 0; i < b.length(); ++i) {
        int j;
        for (j = 0; (j < a.length()) && (a[j] != b[i]); ++j)
            ;
        if (j >= a.length())
        {
            return false;
        }
    }
    return true;
}
```

假设n是字符串String1的长度，m是字符串String2的长度，那么此算法，需要O (n*m) 次操作，以上面的例子来说，将会有 $16*8 = 128$ 次操作。显然，时间开销太大，应该找到一种更好的办法。

解法二、普通排序

如果允许排序的话，我们可以考虑下排序。比如可先对这两个字符串的字母进行排序，然后再同时对两个字串依次轮询。两个字串的排序需要(常规情况) $O(m \log m) + O(n \log n)$ 次操作，之后的线性扫描需要 $O(m+n)$ 次操作。

同样拿上面的字串做例子，将会需要 $16*4 + 8*3 = 88$ ，再加上对两个字串线性扫描的 $16 + 8 = 24$ 的操作(随着字串长度的增长，你会发现这个算法的效果会越来越好)。

关于排序方法，可采用最常用的快速排序，C有库函数qsort,C++有库函数sort，排序不是重点，因此使用库函数好一些。

```
//copyright@caopengcs 2014-01-14
//注意A B中可能包含重复字符，所以注意A下标不要轻易移动。这种方法改变了字符串。如不想改变请自己复制
bool StringContain(string &a,string &b)
{
    sort(a.begin(),a.end());
    sort(b.begin(),b.end());
    for (int pa = 0, pb = 0; pb < b.length();)
    {
        while ((pa < a.length()) && (a[pa] < b[pb]))
        {
            ++pa;
        }
        if ((pa >= a.length()) || (a[pa] > b[pb]))
        {
            return false;
        }
        //a[pa] == b[pb]
        ++pb;
    }
    return true;
}
```

解法三、计数比较

有没有比快速排序更好的方法呢？进一步，我们可以采用线性时间的计数方法，假设需要比较字符串A（n）中是否包含字符串B（m），统计A中出现的字符 $O(n)$ ，比较B中是否有出现的字符 $O(m)$ ，总计时间复杂度为： $O(n+m)$ 。

代码如下：

```
// copyright@caopengcs 2014-01-14
// modified by @古道西风 2014-01-14
bool StringContain(string &a,string &b)
{
    vector<int> have;
    have.resize(26,0);
    for (int i = 0; i < a.length(); ++i)
    {
        ++have[a[i] - 'A'];
    }
    for (int i = 0; i < b.length(); ++i)
    {
        //若A中只需要包含同一个相同的字符即可代表B中重复出现的字符
        //即A: CDEFAB B: AABBCC 合法
        if (have[b[i] - 'A'] == 0)
        {
            return false;
        }
        //若A中需要包含B中所有重复出现字符
        //即A: AAACCDDBCCCE B: AABBCC 合法
        //即A: BCDEFA B: AABBC 非法
        //if (have[b[i] - 'A']-- == 0) {
        //    return false;
        //}
    }
    return true;
}
```

解法四、巧用hashtable

上述方案中，较好的方法是解法三中的计数方法，总的时间已经优化到了： $O(m+n)$ ，貌似到了极限，还有没有更好的办法？

更进一步，可以对短字符串进行轮询，把其中的每个字母都放入一个Hashtable里（始终设m为短字符串的长度，那么此项操作成本是 $O(m)$ 或8次操作）。然后轮询长字符串，在Hashtable里查询短字符串的每个字符，看能否找到。如果找不到，说明没有匹配成功，轮询长字符串

将消耗掉16次操作，这样两项操作加起来一共只有8+16=24次。

当然，理想情况是如果长字串的前缀就为短字串，只需消耗8次操作，这样总共只需8+8=16次。

具体算法流程如下： 1. hash[26]，先全部清零，然后扫描短的字符串，若有相应的置1， 2. 计算hash[26]中1的个数，记为m 3. 扫描长字符串的每个字符a；若原来hash[a] == 1，则修改hash[a] = 0，并将m减1；若hash[a] == 0，则不做处理 4. 若m == 0 or 扫描结束，退出循环。

代码实现，也不难，如下：

```
// copyright@caopengcs 2014-StringContains01-14
bool StringContain(string &a, string &b)
{
    vector<int> hash;
    hash.resize(26, 0);
    int m = 0;
    for (int i = 0; i < b.length(); ++i)
    {
        int x = b[i] - 'A';
        if (hash[x] == 1)
        {
            hash[x] = 1;
            ++m;
        }
    }
    for (int i = 0; i < a.length() && m > 0; ++i)
    {
        int x = a[i] - 'A';
        if (hash[x] == 1)
        {
            --m;
            hash[x] = 0;
        }
    }
    return m == 0;
}
```

解法五、素数相乘

如果面试官继续追问，还有没有更好的办法呢？

我们换一种角度思考本问题：

假设有一个仅由字母组成字串，让每个字母与一个素数对应，从2开始，往后类推，A对应2，B对应3，C对应5，.....。遍历第一个字串，把每个字母对应素数相乘。最终会得到一个整数。

利用上面字母和素数的对应关系，对应第二个字符串中的字母，然后轮询，用每个字母对应的素数除前面得到的整数。如果结果有余数，说明结果为false。如果整个过程中没有余数，则说明第二个字符串是第一个的子集了（判断是不是真子集，可以比较两个字符串对应的素数乘积，若相等则不是真子集）。

思路总结如下：

- 1. 按照从小到大的顺序，用26个素数分别与字符'A'到'Z'一一对应。
- 2. 遍历长字符串，求得每个字符对应素数的乘积。
- 3. 遍历短字符串，判断乘积能否被短字符串中的字符对应的素数整除。
- 4. 输出结果。

如前所述，算法的时间复杂度为O(m+n)的最好的情况为O(n)（遍历短的字符串的第一个数，与长字符串素数的乘积相除，即出现余数，便可退出程序，返回false），n为长字串的长度，空间复杂度为O(1)。

```

// copyright@caopengcs
//此方法只有理论意义，因为整数乘积很大，有溢出风险
bool StringContain(string &a, string &b)
{
    const int p[26] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101;
    int f = 1;
    for (int i = 0; i < a.length(); ++i)
    {
        int x = p[a[i] - 'A'];
        if (f % x)
        {
            f *= x;
        }
    }
    for (int i = 0; i < b.length(); ++i)
    {
        int x = p[b[i] - 'A'];
        if (f % x)
        {
            return false;
        }
    }
    return true;
}

```

此种素数相乘的方法看似完美，但缺点是素数相乘的结果容易导致整数溢出。

解法六、位运算

最好的思路是对字符串A,用位运算（26bit整数表示）计算出一个“签名”，再用B中的字符到A里面进行查找。这个方法的实质是用一个整数代替了hashtable，空间复杂度可以降低为O(1)。时间复杂度还是O(n + m)。

```

// copyright@caopengcs
// “最好的方法”，时间复杂度O(n + m)，空间复杂度O(1)
bool StringContain(string &a, string &b)
{
    int hash = 0;
    for (int i = 0; i < a.length(); ++i)
    {
        hash |= (1 << (a[i] - 'A'));
    }
    for (int i = 0; i < b.length(); ++i)
    {
        if ((hash & (1 << (b[i] - 'A'))) == 0)
        {
            return false;
        }
    }
    return true;
}

```

举一反三

1、本问题的几个变形

- 一个单词单词字母交换，可得另一个单词，如army->mary，成为兄弟单词。提供一个单词，在字典中找到它的兄弟。描述数据结构和查询过程。
- 如果两个字符串的字符一样，但是顺序不一样，被认为是兄弟字符串，问如何在迅速匹配兄弟字符串（如，bad和adb就是兄弟字符串）。
 - 假设两个字符串中所含有的字符和个数都相同我们就叫这两个字符串匹配，比如：abcda和adabc,由于出现的字符个数都是相同，只是顺序不同，所以这两个字符串是匹配的，要求高效。

分析：上述3个问题的解决思路基本同本节的解决方法一致，只是题目做了个变形。

带通配符的字符串匹配问题

题目描述

字符串匹配问题：给定一个字符串，按照指定规则对其进行匹配，并将匹配的结果保存至output数组中。多个匹配项用空格间隔，最后一个不需要空格。

要求：

- 匹配规则中包含通配符?和*，其中?表示匹配任意一个字符，*表示匹配任意多个(≥ 0)字符。
- 匹配规则要求匹配最大的字符串子串，例如a*d，匹配abbdd而非abbd，即最大匹配子串。
- 匹配后的输入串不再进行匹配，从当前匹配后的字符串重新匹配其他字符串。

请实现函数：

```
char* my_find(char input[], char rule[])
```

举例说明：

input: abcabcdefg

rule:a?c

output:abc

input :newsadfanewfdadsf

rule: new

output: new new

input :breakfastfood

rule: f*d

output:fastfood

注意事项：

- 自行实现函数my_find，勿在my_find函数里夹杂输出，且不准用C、C++库，和Java的String对象；
- 请注意代码的时间，空间复杂度，及可读性，简洁性；
- input=aaa, rule=aa时，返回一个结果aa，即可。

分析与解法

解法一、直接匹配

本题与“字符串转换成整数”的题目不同，那题偏向于考察思维的全面性和细节的处理。而本题则偏向于编程技巧。闲不多说，直接上代码：

```
//copyright@cao_peng 2013/4/23
int str_len(char *a)
{
    //字符串长度
    if (a == 0)
    {
        return 0;
    }
    char *t = a;
    for (; *t; ++t)
        ;
    return (int) (t - a);
}

void str_copy(char *a, const char *b, int len)
{
    //拷贝字符串 a = b
    for (; len > 0; --len, ++b, ++a)
```

```

        *a = *b;
    }
    *a = 0;
}

char *str_join(char *a, const char *b, int lenb)
{
    //连接字符串 第一个字符串被回收
    char *t;
    if (a == 0)
    {
        t = (char *) malloc(sizeof(char) * (lenb + 1));
        str_copy(t, b, lenb);
        return t;
    }
    else
    {
        int lena = str_len(a);
        t = (char *) malloc(sizeof(char) * (lena + lenb + 2));
        str_copy(t, a, lena);
        *(t + lena) = ' ';
        str_copy(t + lena + 1, b, lenb);
        free(a);
        return t;
    }
}

int canMatch(char *input, char *rule)
{
    // 返回最长匹配长度 -1表示不匹配
    if (*rule == 0)
    {
        //已经到rule尾端
        return 0;
    }
    int r = -1, may;
    if (*rule == '*')
    {
        r = canMatch(input, rule + 1); // *匹配0个字符
        if (*input)
        {
            may = canMatch(input + 1, rule); // *匹配非0个字符
            if ((may >= 0) && (++may > r))
            {
                r = may;
            }
        }
    }
    if (*input == 0)
    {
        //到尾端
        return r;
    }
    if ((*rule == '?') || (*rule == *input))
    {
        may = canMatch(input + 1, rule + 1);
        if ((may >= 0) && (++may > r))
        {
            r = may;
        }
    }
    return r;
}

char * my_find(char input[], char rule[])
{
    int len = str_len(input);
    int *match = (int *) malloc(sizeof(int) * len); //input第i位最多能匹配多少位 匹配不上是-1
    int i, max_pos = -1;
    char *output = 0;

    for (i = 0; i < len; ++i)
    {
        match[i] = canMatch(input + i, rule);
        if ((max_pos < 0) || (match[i] > match[max_pos]))
        {
            max_pos = i;
        }
    }
}

```

```

    }
}

if ((max_pos < 0) || (match[max_pos] <= 0))
{
    //不匹配
    output = (char *) malloc(sizeof(char));
    *output = 0; // \0
    return output;
}
for (i = 0; i < len;)
{
    if (match[i] == match[max_pos])
    {
        //找到匹配
        output = str_join(output, input + i, match[i]);
        i += match[i];
    }
    else
    {
        ++i;
    }
}
free(match);
return output;
}

```

解法二、动态规划

本题也可以直接写出DP(Dynamic Programming, 动态规划)方程。如下所示：

```

//copyright@chpeih 2013/4/23
char* my_find(char input[], char rule[])
{
    //write your code here
    int len1, len2;
    for (len1 = 0; input[len1]; len1++);
    for (len2 = 0; rule[len2]; len2++);
    int MAXN = len1 > len2 ? (len1 + 1) : (len2 + 1);
    int **dp;

    //dp[i][j]表示字符串1和字符串2分别以i j结尾匹配的最大长度
    //记录dp[i][j]是由之前那个节点推算过来 i*MAXN+j
    dp = new int *[len1 + 1];
    for (int i = 0; i <= len1; i++)
    {
        dp[i] = new int[len2 + 1];
    }

    dp[0][0] = 0;
    for (int i = 1; i <= len2; i++)
        dp[0][i] = -1;
    for (int i = 1; i <= len1; i++)
        dp[i][0] = 0;

    for (int i = 1; i <= len1; i++)
    {
        for (int j = 1; j <= len2; j++)
        {
            if (rule[j - 1] == '*')
            {
                if (j >= 2 && rule[j - 2] == '*')
                {
                    dp[i][j] = dp[i][j - 1];
                    continue;
                }
                dp[i][j] = -1;
                if (dp[i - 1][j - 1] != -1)
                {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                }
            }
            if (dp[i - 1][j] != -1 && dp[i][j] < dp[i - 1][j] + 1)
            {
                dp[i][j] = dp[i - 1][j] + 1;
            }
        }
    }
}

```

```

        }
    }
    else if (rule[j - 1] == '?')
    {
        if (dp[i - 1][j - 1] != -1)
        {
            dp[i][j] = dp[i - 1][j - 1] + 1;
        }
        else dp[i][j] = -1;
    }
    else
    {
        dp[i][j] = -1;
        if (input[i - 1] == rule[j - 1])
        {
            if (dp[i - 1][j - 1] != -1)
            {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            }
            else if (j >= 2 && rule[j - 2] == '*')
            {
                int tmp = j - 2;
                while (tmp >= 0 && rule[tmp] == '*')
                {
                    tmp--;
                }
                tmp++;
                if (dp[i - 1][tmp] != -1)
                {
                    dp[i][j] = dp[i - 1][tmp] + 1;
                }
            }
        }
    }
}

int m = -1;//记录最大字符串长度
int *ans = new int[len1];
int count_ans = 0;//记录答案个数
char *returnans = new char[len1 + 1];
int count = 0;
for (int i = 1; i <= len1; i++)
{
    if (dp[i][len2] > m)
    {
        m = dp[i][len2];
        count_ans = 0;
        ans[count_ans++] = i - m;
    }
    else if (dp[i][len2] != -1 && dp[i][len2] == m && ans[count_ans] + m <= i - m)
    {
        ans[count_ans++] = i - m;
    }
}

if (count_ans != 0)
{
    int len = ans[0];
    for (int i = 0; i < m; i++)
    {
        printf("%c", input[i + ans[0]]);
        returnans[count++] = input[i + ans[0]];
    }
    for (int j = 1; j < count_ans; j++)
    {
        printf(" ");
        returnans[count++] = ' ';
        len = ans[j];
        for (int i = 0; i < m; i++)
        {
            printf("%c", input[i + ans[j]]);
            returnans[count++] = input[i + ans[j]];
        }
    }
    printf("\n");
    returnans[count++] = '\0';
}

```

```
    return returnans;
}
```

```
/*
 * 平均O(n+m) 最坏O(n*m)
 * 约定 * : ANY>=0 , ? : ANY=1
 *
 * @author Spance.Wong
 */
static class WildCardMatcher {

    /**
     * 仅为了方便实验
     *
     * @param input
     * @param pattern
     * @return
     */
    static List<String> matches(String input, String pattern) {
        String[] pa = pattern.split("\\*+");
        // 分割不是重点, 故未做重点实现
        return matches(input, pa);
    }

    /**
     * 从input中查找通配符序列
     *
     * @param input
     * @param patterns
     * @return
     */
    static List<String> matches(CharSequence input, String[] patterns) {
        int n = input.length(), m = patterns.length;
        List<String> result = new ArrayList<String>();
        for (int i = 0; i < n; ) {
            int left = -1, right = -1;
            for (int j = 0; j < m; j++) { // 以i为起点, 执行m趟匹配, 每趟i至少前进p[j].length长度
                long region = lookBehind(input, i, patterns[j]);
                if (j != 0 && region >= 0) { // 模式序列的第二个开始使用贪婪匹配
                    long greedyRegion;
                    for (int k = (int) region + 1; ; k = (int) greedyRegion + 1) {
                        greedyRegion = lookBehind(input, k, patterns[j]);
                        if (greedyRegion > 0) // 贪婪找到, 继续贪婪尝试
                            region = greedyRegion;
                        else
                            break;
                    }
                }
                if (region < 0) { // pattern[j]失败, 则本趟失败
                    i = ((int) -region) + 1;
                    break;
                } else {
                    i = (int) region + 1;
                    if (j == 0) // 模式序列的第一个找到, 记左边界, 在高32位
                        left = (int) (region >> 32);
                    if (j == m - 1) // 模式序列的最后一个找到, 记右边界, 在低32位
                        right = (int) region;
                }
            }
            if (left >= 0 && right >= 0)
                result.add(input.subSequence(left, right + 1).toString());
        }
        return result;
    }

    /**
     * 在input的i位置开始向后扫描非贪婪查找pattern, 在pattern尾匹配时回溯确认
     *
     * @param in
     * @param i
     * @param pattern
     * @return
     */
    static long lookBehind(CharSequence in, int i, CharSequence pattern) {
        int len = in.length(), pLen = pattern.length(), _pMax = pLen - 1;
        char pEnd = pattern.charAt(_pMax);
        if (len - i >= pLen) {
            for (i = i + _pMax; i < len; i++) { // 以 i + pLen - 1 起步
                if (in.charAt(i) == pEnd) {
                    if (i + pLen - 1 < len) {
                        if (lookBehind(in, i + pLen - 1, pattern) >= 0)
                            return i;
                    } else
                        return -1;
                }
            }
        }
        return -1;
    }
}
```

```
        if (in.charAt(i) == pEnd || pEnd == '?') {      // 与pa末尾相同, i即右边界
            if (pLen == 1)
                return ((long) i) << 32 | i;
            for (int j = i - 1; j >= i - _pMax; j--) {    // 则至多回溯pLen长找左边界
                char p = pattern.charAt(_pMax - i + j);
                if (in.charAt(j) == p || p == '?') {
                    if (j == i - _pMax)      // 找到左边界即j
                        return ((long) j) << 32 | i;
                } else
                    break;
            }
        }
    }
    return -i;
}
}
```

举一反三

1、给定一个带通配符问号的数W，问号可以代表任意数字。再给定一个整数X，和W具有同样的长度。问有多少个整数符合W的形式并且比X大？

[prev](#) | [next](#)

[Back to home](#)

Generated by [mdtogh](#)

字符串转换成整数

题目描述

输入一个由数字组成的字符串，把它转换成整数并输出。例如：输入字符串"345"，输出整数345。

给定函数原型 `int StrToInt(const char *str)`，实现字符串转换成整数的功能。不能使用库函数atoi。

分析与解法

我们来一步一步分析，直至写出第一份准确的代码。本题考查的实际上就是字符串转换成整数的问题，或者说是要你自行实现atoi函数。那如何实现把表示整数的字符串正确地转换成整数呢？以"345"作为例子：

- 当我们扫描到字符串的第一个字符'3'时，由于我们知道这是第一位，所以得到数字3。
- 当扫描到第二个数字'4'时，而之前我们知道前面有一个3，所以便在后面加上一个数字4，那前面的3相当于30，因此得到数字： $3 \times 10 + 4 = 34$ 。
- 继续扫描到字符'5'，'5'的前面已经有了34，由于前面的34相当于340，加上后面扫描到的5，最终得到的数是： $34 \times 10 + 5 = 345$ 。

因此，此题的基本思路：从左至右扫描字符串，把之前得到的数字乘以10，再加上当前字符表示的数字。

但有一些细节需要注意：

- 正负符号：整数不仅包含数字，还有可能是以'+'或'-'开头表示正负整数，因此如果第一个字符是'-'号，则要把得到的整数转换成负整数。
- 空指针输入：输入的是指针，在访问空指针时程序会崩溃，因此在使用指针之前需要先判断指针是否为空。
- 非法字符：输入的字符串中可能含有不是数字的字符。因此，每当碰到这些非法的字符，程序应停止转换。
- 整型溢出：输入的数字是以字符串的形式输入，因此输入一个很长的字符串将可能导致溢出。

如图，输入字符串用例和分析的结果（假定运行环境是32位系统，且编译环境是VS2008以上）：

"	0
"1"	1
"+1"	1
"-1"	-1
"123"	123
"-123"	-123
"010"	10
"+00131204"	131204
"-01324000"	-1324000
"2147483647"	2147483647
"-2147483647"	-2147483647
"-2147483648"	-2147483648
"2147483648"	2147483647
"-2147483649"	-2147483648
"abc"	0
"-abc"	0
"1a"	1
"23a8f"	23
"-3924x8fc"	-3924
" 321"	321
" -321"	-321
"123 456"	123
"123 "	123
" - 321"	0
" +4488"	4488
" + 413"	0
" ++c"	0
" ++1"	0
" --2"	0
" -2"	-2

解法一

初步实现：

```

//copyright@zhedahht 2007
enum Status {kValid = 0, kInvalid};
int g_nStatus = kValid;

int StrToInt(const char* str)
{
    g_nStatus = kInvalid;
    long long num = 0;

    if (str != NULL)
    {
        const char* digit = str;

        // 判断字符串是否以'+'或'-'开头
        bool minus = false;
        if (*digit == '+')
            digit++;
        else if (*digit == '-')
        {
            digit++;
            minus = true;
        }

        // 从左至右扫描字符串
        while (*digit != '\0')
        {
            if (*digit >= '0' && *digit <= '9')
            {
                num = num * 10 + (*digit - '0');

                //处理型溢出问题
                if (num > std::numeric_limits<int>::max())
                {
                    num = 0;
                    break;
                }
            }

            digit++;
        }
        //处理字符问题。如“1a”
        else
        {
            num = 0;
            break;
        }
    }

    if (*digit == '\0')
    {
        g_nStatus = kValid;
        if (minus)
            num = 0 - num;
    }
}

return static_cast<int>(num);
}

```

这份实现有两个问题：

当输入的字符串不是数字，而是字符的时候，比如“1a”，上述程序直接返回了0（而正确的结果应该是得到1）：

```

//处理字符问题。如“1a”
else
{
    num = 0;
    break;
}

```

处理溢出时，有问题。因为它遇到溢出情况时，直接返回了0：

```

//处理型溢出问题
if (num > std::numeric_limits<int>::max())
{
    num = 0;
    break;
}

```

解法二

对代码做下微调。当发生溢出时，取最大或最小的int值。当输入字符串包含字母时，停止转换并返回结果。（注：库函数atoi规定：大于int值，取最大值maxint: 2147483647；小于-int取最小值minint: -2147483648）

```

//copyright@SP_daiyq 2013/5/29
int StrToInt(const char* str)
{
    int res = 0; //存放结果
    int i = 0; // 字符串下标
    int signal = '+'; // 数字的符号, '+' 或 '-'
    int cur; // 当前数字
    //处理空指针问题
    if (!str)
        return 0;

    //跳过空格
    while (isspace(str[i]))
        i++;

    //跳过符号
    if (str[i] == '+' || str[i] == '-')
    {
        signal = str[i];
        i++;
    }

    //计算结果
    while (str[i] >= '0' && str[i] <= '9')
    {
        cur = str[i] - '0';

        //当发生正溢出时, 返回INT_MAX; 发生负溢出时, 返回INT_MIN
        if ( (signal == '+') && (cur > INT_MAX - res * 10) )
        {
            res = INT_MAX;
            break;
        }
        else if ( (signal == '-') && (cur - 1 > INT_MAX - res * 10) )
        {
            res = INT_MIN;
            break;
        }

        res = res * 10 + cur;
        i++;
    }

    return (signal == '-') ? -res : res;
}

```

此时会发现，非法字符的问题解决了：

"-2147483648"	-2147483648	-2147483648	✓
"2147483648"	2147483647	2147483647	✓
"_2147483649"	-2147483648	-2147483648	✓
"abc"	0	0	✓
"-abc"	0	0	✓
"1a"	1	1	✓
"23a8f"	23	23	✓
"-3924x8fc"	-3924	-3924	✓
" 321"	321	321	✓
" -321"	-321	-321	✓
"123 458"	123	123	✓
"123 "	123	123	✓
" - 321"	0	0	✓
" +4488"	4488	4488	✓
" + 413"	0	0	✓
" ++c"	0	0	✓
" ++1"	0	0	✓
" --2"	0	0	✓
" -2"	-2	-2	✓

但整形溢出问题却没有解决：

" 10522545459"	1932610867	2147483647	✗
" +10523538441s"	1933603849	2147483647	✗
" +10432359437"	1842424845	2147483647	✗

当上述代码转换" 10522545459"时，它应得到2147483647，但程序运行结果却是：1932610867。所以程序没有解决好溢出问题。

分析下代码，看是如何具体处理溢出情况的：

```
// judge overlap or not
if ( (signal == '+') && (cur > INT_MAX - res * 10) )
{
    res = INT_MAX;
    break;
}
else if ( (signal == '-') && (cur - 1 > INT_MAX - res * 10) )
{
    res = INT_MIN;
    break;
}
```

给定字符串" 10522545459"，除去空格有11位。而MAX_INT是2147483647，有10位数。当扫描到最后一个字符'9'的时候，程序会比较 9 和 2147483647 - 1052254545*10的大小。

问题立马就暴露出来了，因为此时让res*10，即让1052254545*10 > MAX_INT，溢出无疑。程序已经出错，再执行下面这行代码已无意义：

解法三

上面说给的程序没有“很好的解决溢出问题。由于输入的数字是以字符串的形式输入，因此有可能输入一个很大的数字转换之后会超过能够表示的最大的整数而溢出”。

看看下面的代码

```

//copyright@fuwutu 2013/5/29
int StrToInt(const char* str)
{
    bool negative = false;
    long long result = 0; // 返回值定义成了long Long
    while (*str == ' ' || *str == '\t')
    {
        ++str;
    }
    if (*str == '-')
    {
        negative = true;
        ++str;
    }
    else if (*str == '+')
    {
        ++str;
    }

    while (*str != '\0')
    {
        int n = *str - '0';
        if (n < 0 || n > 9)
        {
            break;
        }

        if (negative)
        {
            result = result * 10 - n;
            if (result < -2147483648LL)
            {
                result = -2147483648LL;
            }
        }
        else
        {
            result = result * 10 + n;
            if (result > 2147483647LL)
            {
                result = 2147483647LL;
            }
        }
        ++str;
    }

    return result;
}

```

运行上述程序：

" +1q0384297885"	1	1	✓
" -1034946q1019"	-1034946	-1034946	✓
" 11810097701"	2147483647	2147483647	✓
"115784825v67"	115784825	115784825	✓
" 1030q2849284"	1030	1030	✓
" -120m15417222"	-120	-120	✓
"1095502006p8"	1095502006	1095502006	✓
" r11384376420"	0	0	✓
" 10522545459"	2147483647	2147483647	✓
" +1u0557196150"	1	1	✓
" 10315g546111"	10315	10315	✓
" +o11950655481"	0	0	✓
" 1052223016k4"	1052223016	1052223016	✓
" 1206z8519909"	1206	1206	✓
" -115a64905859"	-115	-115	✓
" 1191597x7433"	1191597	1191597	✓
" 1094m3066812"	1094	1094	✓

上图所示程序貌似通过了，然而实际上它还是未能处理数据溢出的问题。因为它只是做了个取巧：把返回的值result定义成了long long。严格说来，我们依然未写出准确的规范代码。

解法四

根据我们最初的思路：“从左至右扫描字符串，把之前得到的数字乘以10，再加上当前字符表示的数字”，相信读者已经觉察到，在扫描到最后一个字符的时候，如果之前得到的数比较大，此时若再让其扩大10倍，相对来说是比较容易溢出的。

但车到山前必有路，既然让一个比较大的int整型数据大10倍，比较容易溢出，那么在不好判断是否溢出的情况下，可以尝试使用除法：

- 与其将n扩大10倍，冒着溢出的风险，再与MAX_INT进行比较（如果已经溢出，则比较的结果没有意义），
- 不如未雨绸缪先用n与MAX_INT/10进行比较：若n>MAX_INT/10（当然同时还要考虑n==MAX_INT/10的情况），说明最终得到的整数一定会溢出，故此时可以当即进行溢出处理，直接返回最大值MAX_INT，从而也就免去了计算n*10这一步骤。

也就是说，计算n*10前，先比较n与MAX_INT/10大小，若n>MAX_INT/10，那么n*10肯定大于MAX_INT，即代表最后得到的整数n肯定溢出，既然溢出，不能再计算n*10，直接提前返回MAX_INT就行了。

一直以来，我们努力的目的归根结底是为了更好的处理溢出，但上述做法最重要的是巧妙的规避了计算n*10这一乘法步骤，转换成计算除法MAX_INT/10代替，不能不说此法颇妙。

对于正数来说，它溢出的可能性有两种：1. 一种是诸如2147483650，即n > MAX/10 的；
2. 一种是诸如2147483649，即n == MAX/10 && c > MAX%10。

部分代码如下：

```
//copyright@njnu_mjn 2013
c = *str - '0';
if (sign > 0 && (n > MAX / 10 || (n == MAX / 10 && c > MAX % 10)))
{
    n = MAX;
    break;
}
else if (sign < 0 && (n > (unsigned)MIN / 10
    || (n == (unsigned)MIN / 10 && c > (unsigned)MIN % 10)))
{
    n = MIN;
    break;
}
```

测试结果如下，暂未发现什么问题

输入	输出
10522545459 :	2147483647
-10522545459 :	-2147483648

咱们再来总结下上述代码是如何处理溢出情况的。

仍有一些细节是值得改进的，如：

将MAX/10,MAX%10,(unsigned)MIN/10及(unsigned)MIN%10保存到变量中，防止重复计算

解法五

优化后，完整的代码为：

```

//copyright@njnu_mjn 2013
int StrToInt(const char* str)
{
    static const int MAX = (int)((unsigned)~0 >> 1);
    static const int MIN = -(int)((unsigned)~0 >> 1) - 1;
    static const int MAX_DIV = (int)((unsigned)~0 >> 1) / 10;
    static const int MIN_DIV = (int)((((unsigned)~0 >> 1) + 1) / 10);
    static const int MAX_R = (int)((unsigned)~0 >> 1) % 10;
    static const int MIN_R = (int)((((unsigned)~0 >> 1) + 1) % 10);
    int n = 0;
    int sign = 1;
    int c;

    while (isspace(*str))
        ++str;
    if (*str == '+' || *str == '-')
    {
        if (*str == '-')
            sign = -1;
        ++str;
    }
    while (isdigit(*str))
    {
        c = *str - '0';
        if (sign > 0 && (n > MAX_DIV || (n == MAX_DIV && c >= MAX_R)))
        {
            n = MAX;
            break;
        }
        else if (sign < 0 && (n > MIN_DIV
            || (n == MIN_DIV && c >= MIN_R)))
        {
            n = MIN;
            break;
        }
        n = n * 10 + c;
        ++str;
    }
    return sign > 0 ? n : -n;
}

```

部分数据的测试结果如下图所示：

输入	输出
10522545459	: 2147483647
-10522545459	: -2147483648
2147483648	: 2147483647
-2147483648	: -2147483648

程序已实现题目的要求。但如程序作者MJN君所说“我的实现与linux内核的atoi函数的实现，都有一个共同的问题：即使出错，函数也返回了一个值，导致调用者误认为自己传入的参数是正确的，但是可能会导致程序的其他部分产生莫名的错误且很难调试”。

类似问题

- 实现string到double的转换

提醒：此题虽然类似于atoi函数，但毕竟double为64位，而且支持小数，因而边界条件更加严格，写代码时需要更加注意。

[prev](#) | [next](#)

[Back to home](#)

回文判断

题目描述

回文，英文palindrome，指一个顺着读和反过来读都一样的字符串，比如madam、我爱我，这样的短句在智力性、趣味性和艺术性上都颇有特色，中国历史上还有很多有趣的回文诗。

那么，我们的第一个问题就是：判断一个字串是否是回文？

分析与解法

回文判断是一类典型的问题，尤其是与字符串结合后呈现出多姿多彩，在实际中使用也比较广泛，而且也是面试题中的常客，所以本节就结合几个典型的例子来体味下回文之趣。

解法一

同时从字符串头尾开始向中间扫描字串，如果所有字符都一样，那么这个字串就是一个回文。采用这种方法的话，我们只需要维护头部和尾部两个扫描指针即可。

代码如下：

```
/*
 * 检查字符串s是不是回文，字符串s的长度为n
 * Copyright(C) fairywell 2011
 */
bool IsPalindrome(const char *s, int n)
{
    if (s == NULL || n < 1) return false; // 非法输入
    char *front, *back;
    front = s; back = s + n - 1; // 初始化头指针和尾指针
    while (front < back) {
        if (*front != *back)
            return false; // 不是回文，立即返回
        ++front;
        --back;
    }
    return true; // 是回文
}
```

这是一个直白且效率不错的实现，时间复杂度： $O(n)$ ，空间复杂度： $O(2)$ 。

解法二

上述解法一从两头向中间扫描，那么是否还有其它办法呢？我们可以先从中间开始、然后向两边扩展查看字符是否相等。参考代码如下：

```
/*
 * 检查字符串s是不是回文，字符串s的长度为n
 * Copyright(C) fairywell 2011
 */
bool IsPalindrome2(const char *s, int n)
{
    if (s == NULL || n < 1) return false; // 非法输入
    char *first, *second;
    int m = ((n >> 1) - 1) >= 0 ? (n >> 1) - 1 : 0; // m is the middle point of s
    first = s + m; second = s + n - 1 - m;
    while (first >= s)
        if (s[first--] != s[second++]) return false; // not equal, so it's not a palindrome
    return true; // check over, it's a palindrome
}
```

时间复杂度： $O(n)$ ，空间复杂度： $O(1)$ 。

虽然本解法二的时空复杂度和解法一是一样的，但很快我们会看到，在某些回文问题里面，这个方法有着自己的独到之处，可以方便的解决

一类问题。

举一反三

1、判断一条单向链表是不是“回文”

分析：对于单链表结构，可以用两个指针从两端或者中间遍历并判断对应字符是否相等。但这里的关键就是如何朝两个方向遍历。由于单链表是单向的，所以要向两个方向遍历的话，可以采取经典的快慢指针的方法，即先位到链表的中间位置，再将链表的后半逆置，最后用两个指针同时从链表头部和中间开始同时遍历并比较即可。

2、判断一个栈是不是“回文”

分析：对于栈的话，只需要将字符串全部压入栈，然后依次将各字符出栈，这样得到的就是原字符串的逆置串，分别和原字符串各个字符比较，就可以判断了。

[prev](#) | [next](#)

[Back to home](#)

Generated by [mdtogh](#)

最长回文子串的长度

题目描述

给定一个字符串，求它的最长回文子串的长度。

分析与解法

最容易想到的办法是枚举所有的子串，分别判断其是否为回文。这个思路初看起来是正确的，但却做了很多无用功，如果一个长的子串包含另一个短一些的子串，那么对子串的回文判断其实是不需要的。

解法一

那么如何高效的进行判断呢？借鉴KMP算法的做法，既然对短的子串的判断和包含它的长的子串的判断重复了，我们何不复用下短的子串的判断呢，即让短的子串的判断成为长的子串的判断的一个部分。

没错，扩展法。从一个字符开始，向两边扩展，看看最多能到多长，使其保持为回文。这也就是为什么我们在上一节里面要提出解法二的原因。

具体而言，我们可以枚举中心位置，然后再在该位置上用扩展法，记录并更新得到的最长的回文长度，即为所求。代码如下：

```
/*
*find the Longest palindrome in a string, n is the length of string s
*Copyright(C) fairywell 2011
*/
int LongestPalindrome(const char *s, int n)
{
    int i, j, max;
    if (s == 0 || n < 1) return 0;
    max = 0;
    for (i = 0; i < n; ++i) { // i is the middle point of the palindrome
        for (j = 0; (i-j) >= 0 && (i+j < n); ++j) // if the Lengthof the palindrome is odd
            if (s[i-j] != s[i+j])
                break;
        if (j*2+1 > max)
            max = j * 2 + 1;
        for (j = 0; (i-j) >= 0 && (i+j+1 < n); ++j) // for theeven case
            if (s[i-j] != s[i+j+1])
                break;
        if (j*2+2 > max)
            max = j * 2 + 2;
    }
    return max;
}
```

代码稍微难懂一点的地方就是内层的两个 for 循环，它们分别对于以 i 为中心的，长度为奇数和偶数的两种情况，整个代码遍历中心位置 i 并以此之扩展，找出最长的回文。

当然，还有更先进但也更复杂的方法，比如用 s 和逆置 s' 的组合 $s\$s'$ 来建立后缀树的方法也能找到最长回文，但构建的过程比较复杂，所以在实践中用的比较少，感兴趣的朋友可以参考相应资料。--well。

解法二、O(N)解法

参考：<http://bbs.dlut.edu.cn/bbstcon.php?board=Competition&gid=23474>，或这篇文章：<http://www.felix021.com/blog/read.php?2040>，或：<http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html>，待续。

[prev](#) | [next](#)

[Back to home](#)

字符串其它问题

1、第一个只出现一次的字符

在一个字符串中找到第一个只出现一次的字符。如输入abaccdeff，则输出b。

分析：这道题是2006年google 的一道笔试题。它在今年又出现了，不过换了一种形式。即最近的搜狐笔试大题：数组非常长，如何找到第一个只出现一次的数字，说明算法复杂度。此问题已经在程序员编程艺术系列第二章中有所阐述，在此不再作过多讲解。

代码，可编写如下：

```
//查找第一个只出现一次的字符, 第2个程序
//copyright@ yansha
//July, updated, 2011.04.24.
char FirstNotRepeatChar(char* pString)
{
    if (!pString)
        return '\0';

    const int tableSize = 256;
    int hashTable[tableSize] = {0}; //存入数组，并初始化为0

    char* pHashKey = pString;
    while (*pHashKey != '\0')
        hashTable[*pHashKey++]++;

    while (*pString != '\0')
    {
        if (hashTable[*pString] == 1)
            return *pString;

        pString++;
    }
    return '\0'; //没有找到满足条件的字符，退出
}
```

2、带通配符的字符串匹配问题

字符串匹配问题：给定一个字符串，按照指定规则对其进行匹配，并将匹配的结果保存至output数组中。多个匹配项用空格间隔，最后一个不需要空格。

要求：

- 匹配规则中包含通配符?和*，其中?表示匹配任意一个字符，*表示匹配任意多个(≥ 0)字符。
- 匹配规则要求匹配最大的字符串，例如a*d，匹配abbdd而非abbd，即最大匹配子串。
- 匹配后的输入串不再进行匹配，从当前匹配后的字符串重新匹配其他字符串。

请实现函数：

```
char* my_find(char input[], char rule[])
```

举例说明：

input:abcadefg

rule:a?c

output:abc

input :newsadfanewfdadsf

rule: new

output: new new

input :breakfastfood

rule: f*d

output:fastfood

注意事项：

1. 自行实现函数my_find，勿在my_find函数里夹杂输出，且不准用C、C++库，和Java的String对象；
2. 请注意代码的时间，空间复杂度，及可读性，简洁性；
3. input=aaa, rule=aa时，返回一个结果aa，即可。

3、对称子字符串的最大长度

题目：输入一个字符串，输出该字符串中对称的子字符串的最大长度。比如输入字符串“google”，由于该字符串里最长的对称子字符串是“goog”，因此输出4。

分析：可能很多人都写过判断一个字符串是不是对称的函数，这个题目可以看成是该函数的加强版。

4、实现memcpy函数

已知memcpy的函数为：void* memcpy(void dest, const void src, size_t count)其中dest是目的指针，src是源指针。不调用c++/c的memcpy库函数，请编写memcpy。

分析：参考代码如下：

```
//copyright@July 2013/9/24
void* memcpy(void *dst, const void *src, size_t count)
{
    //安全检查
    assert( (dst != NULL) && (src != NULL) );

    unsigned char *pdst = (unsigned char *)dst;
    const unsigned char *psrc = (const unsigned char *)src;

    //防止内存重复
    assert(! (psrc <= pdst && pdst < psrc + count));
    assert(! (pdst <= psrc && psrc < pdst + count));

    while(count--)
    {
        *pdst = *psrc;
        pdst++;
        psrc++;
    }
    return dst;
}
```

5、实现memmove函数

分析：memmove函数是的标准函数，其作用是把从source开始的num个字符拷贝到destination。

最简单的方法是直接复制，但是由于它们可能存在内存的重叠区，因此可能覆盖了原有数据。比如当source+count>=dest&&source<dest时，dest可能覆盖了原有source的数据。

解决办法是从后往前拷贝，对于其它情况，则从前往后拷贝。

参考代码如下：

```
//void * memmove ( void * destination, const void * source, size_t num );
void* memmove(void* dest, void* source, size_t count)
{
    void* ret = dest;

    if (dest <= source || dest >= (source + count))
    {
        //正向拷贝
        //copy from Lower addresses to higher addresses
        while (count --)
            *dest++ = *source++;
    }
    else
    {
        //反向拷贝
        //copy from higher addresses to lower addresses
        dest += count - 1;
        source += count - 1;

        while (count--)
            *dest-- = *source--;
    }
    return ret;
}
```

[prev](#) | [next](#)

[Back to home](#)

Generated by [mdtogh](#)