字符串

七月算法 邹博

2015年4月9日

字符串

- 字符串的范畴非常广泛；
- 难题往往在此节出现；
- 掌握字符串的法门是__________。

  - 面试字符串不会太难，KMP＋Manacher就够了

主要内容

- 需要掌握的内容
  - 字符串循环左移
  - 字符串全排列
    - 递归、非递归
  - KMP
- 需要了解的内容
  - Manacher算法
  - BM算法

字符串循环左移

- 给定一个字符串S[0…N-1]，要求把S的前k个字符移动到S的尾部，如把字符串“abcdef”前面的2个字符‘a’、‘b’移动到字符串的尾部，得到新字符串“cdefab”：即字符串循环左移k。
  - 多说一句：循环左移k位等价于循环右移n-k位。
- 算法要求：
  - 时间复杂度为 O(n)，空间复杂度为 O(1)。

问题分析

- 暴力移位法
  - 每次循环左移1位，调用k次即可
  - 时间复杂度O(kN)，空间复杂度O(1)
- 三次拷贝
  - S[0…k] → T[0…k]
  - S[k+1…N-1] → S[0…N-k-1]
  - T[0…k] →S[N-k…N-1]
  - 时间复杂度O(N)，空间复杂度O(k)

优雅一点的算法

- (X’Y’)’=YX
  - 如： abcdef
  - X=ab　X’=ba
  - Y=cdef　Y’=fedc
  - (X’Y’)’=(bafedc)’=cdefab
- 时间复杂度O(N)，空间复杂度O(1)
- 该问题会在“完美洗牌”算法中再次遇到。

Code

```
void ReverseString(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;
    ReverseString(s, 0, m - 1);
    ReverseString(s, m, n - 1);
    ReverseString(s, 0, n - 1);
}
```

字符串的全排列

- 给定字符串S[0...N-1]，设计算法，枚举S的全排列。

递归算法

- 以字符串1234为例：
- 1 – 234
- 2 – 134
- 3 – 214
- 4 – 231
- 如何保证不遗漏
  - 保证递归前1234的顺序不变

递归Code

```
char str[] = "1234";
int size = sizeof(str) / sizeof(char);

void Permutation(int from, int to)
{
    if(from == to)
    {
        for(int i = 0; i <= to; i++)
        {
            cout << str[i];
        }
        cout << '\n';
        return;
    }
    for(int i = from; i <= to; i++)
    {
        swap(str[i], str[from]);
        Permutation(from+1, to);
        swap(str[i], str[from]);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    Permutation(0, size-2);
    return 0;
}
```

```
1234
1243
1324
1342
1432
1423
2134
2143
2314
2341
2431
2413
3214
3241
3124
3142
3412
3421
4231
4213
4321
4312
4132
4123
```

如果字符有重复

- 去除重复字符的递归算法
- 以字符1223为例：
- 1 – 223
- 2 – 123
- 3 – 221

- 带重复字符的全排列就是每个字符分别与它后面非重复出现的字符交换。
- 即：第i个字符与第j个字符交换时，要求[i,j)中没有与第j个字符相等的数。

有重复字符的递归

1:	1223
2:	1232
3:	1322
4:	2123
5:	2132
6:	2213
7:	2231
8:	2321
9:	2312
10:	3221
11:	3212
12:	3122

```
bool IsSwap(int from, int to)
{
    bool bCan = true;
    for(int i = from; i < to; i++)
    {
        if(str[to] == str[i])
        {
            bCan = false;
            break;
        }
    }
    return bCan;
}
```

```
void Permutation(int from, int to)
{
    if(from == to)
    {
        count++;
        cout << count << ":\t";
        for(int i = 0; i <= to; i++)
        {
            cout << str[i];
        }
        cout << '\n';
        return;
    }

    for(int i = from; i <= to; i++)
    {
        if(!IsSwap(from, i))
            continue;
        swap(str[i], str[from]);
        Permutation(from+1, to);
        swap(str[i], str[from]);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    Permutation(0, size-2);
    return 0;
}
```

重复字符的全排列递归算法时间复杂度

- $\because f(n) = n*f(n-1) + n^2$
- $\because f(n-1)=(n-1)*f(n-2) + (n-1)^2$
- $\therefore f(n) = n*((n-1)*f(n-2) + (n-1)^2) + n^2$
- $\because f(n-2) = (n-2)*f(n-3) + (n-2)^2$
- $\therefore f(n) = n*(n-1)*((n-2)*f(n-3) + (n-2)^2) + n*(n-1)^2 + n^2$
- $= n*(n-1)*(n-2)*f(n-3) + n*(n-1)*(n-2)^2 + n*(n-1)^2 + n^2$
- $= \ldots\ldots\ldots$
- $< n! + n! + n! + n! + \ldots + n!$
- $= (n+1)*n!$
- 时间复杂度为$O((n+1)!)$
  - 注：当n足够大时：$n! > n+1$

用空间换时间

- 如果是单字符，可以使用mark[256]
- 如果是整数，可以遍历整数得到最大值max和最小值min，使用mark[max-min+1]
- 如果是浮点数或者其他结构数据，用Hash(事实上，如果发现整数间变化太大，也应该考虑使用Hash；并且，可以认为整数情况是最朴素的Hash)

```
char str[] = "1223";
int size = sizeof(str) / sizeof(char);
int count = 0;

void Permutation(int from, int to)
{
    if(from == to)
    {
        count++;
        cout << count << ":\t";
        for(int i = 0; i <= to; i++)
        {
            cout << str[i];
        }
        cout << '\n';
        return;
    }

    int mark[256];
    for(int i = 0; i < 256; i++)
        mark[i] = 0;
    for(int i = from; i <= to; i++)
    {
        if(mark[str[i]] == 1)
            continue;
        mark[str[i]] = 1;
        swap(str[i], str[from]);
        Permutation(from+1, to);
        swap(str[i], str[from]);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    Permutation(0, size-2);
    return 0;
}
```

全排列的非递归算法

- 起点：字典序最小的排列，例如12345
- 终点：字典序最大的排列，例如54321
- 过程：从当前排列生成字典序刚好比它大的下一个排列
- 如：21543的下一个排列是23145
  - 如何计算？

21543的下一个排列的思考过程

- 逐位考察哪个能增大
  - 一个数右面有比它大的数存在，它就能增大
  - 那么最后一个能增大的数是——x = 1
- 1应该增大到多少？
  - 增大到它右面比它大的最小的数——y = 3
- 应该变为23xxx
- 显然，xxx应由小到大排：145
- 得到23145

全排列的非递归算法：整理成算法语言

- 步骤：后找、小大、交换、翻转——
- 后找：字符串中最后一个升序的位置i，即：S[k]>S[k+1](k>i)，S[i]<S[i+1]；
- 查找(小大)：S[i+1…N-1]中比Ai大的最小值Sj；
- 交换：Si，Sj；
- 翻转：S[i+1…N-1]
  - 思考：交换操作后，S[i+1…N-1]一定是降序的
- 以926520为例，考察该算法的正确性。

非递归算法Code

```
void Swap(char *a, char *b)
{
    char t = *a;
    *a = *b;
    *b = t;
}

//反转区间
void Reverse(char *a, char *b)
{
    while (a < b)
        Swap(a++, b--);
}
```

```
//下一个排列
bool Next_permutation(char a[])
{
    char *pEnd = a + strlen(a);
    if (a == pEnd)
        return false;
    char *p, *q, *pFind;
    pEnd--;
    p = pEnd;
    while (p != a)
    {
        q = p;
        --p;
        if (*p < *q) //找降序的相邻2数,前一个数即替换数
        {
            //从后向前找比替换点大的第一个数
            pFind = pEnd;
            while (*pFind <= *p)
                --pFind;
            //替换
            Swap(pFind, p);
            //替换点后的数全部反转
            Reverse(q, pEnd);
            return true;
        }
    }
    Reverse(p, pEnd);//没有下一个排列,全部反转后返回true
    return false;
}
```

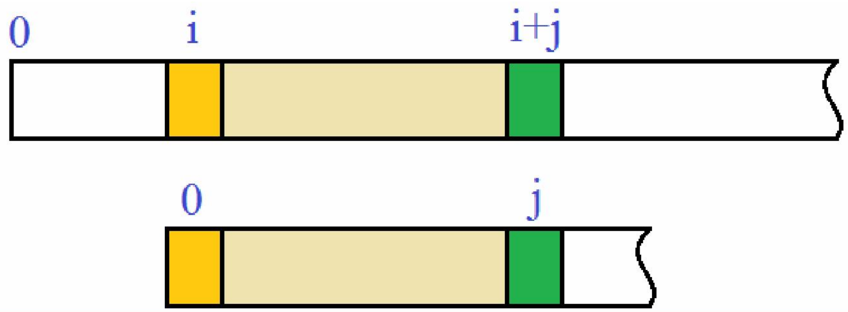
几点说明

- 下一个排列算法能够天然的解决重复字符的问题！
  - 不妨还是考察926520的下一个字符串
- C++STL已经在Algorithm中集成了next_permutation
- 可以将给定的字符串A[0…N-1]首先升序排序，然后依次调用next_permutation直到返回false，即完成了非递归的全排列算法。

KMP算法

- 字符串查找问题
  - 给定文本串text和模式串pattern，从文本串text中找出模式串pattern**第一次出现**的位置。
- 最基本的字符串匹配算法
  - 暴力求解(Brute Force)：时间复杂度O(m*n)
- KMP算法是一种线性时间复杂度的字符串匹配算法，它是对BF算法改进。

- 记：文本串长度为N，模式串长度为M
  - BF算法的时间复杂度O(M*N)，空间复杂度O(1)
  - KMP算法的时间复杂度O(M+N)，空间复杂度O(M)



暴力求解

```
//查找s中首次出现p的位置
int BruteForceSearch(const char* s, const char* p)
{
    int i = 0;  //当前匹配到的原始串首位
    int j = 0;  //模式串的匹配位置
    int size = (int)strlen(s);
    int psize = (int)strlen(p);
    while((i < size) && (j < psize))
    {
        if(s[i+j] == p[j])  //若匹配，则模式串匹配位置后移
        {
            j++;
        }
        else    //不匹配，则比对下一个位置，模式串回溯到首位
        {
            i++;
            j = 0;
        }
    }
    if(j >= psize)
        return i;
    return -1;
}
```

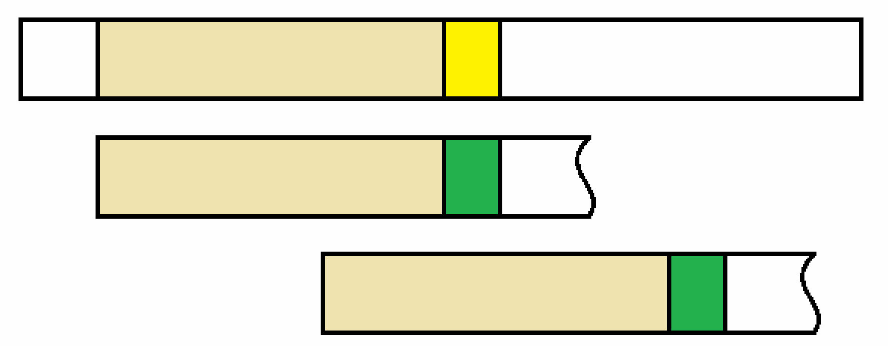
分析BF与KMP的区别

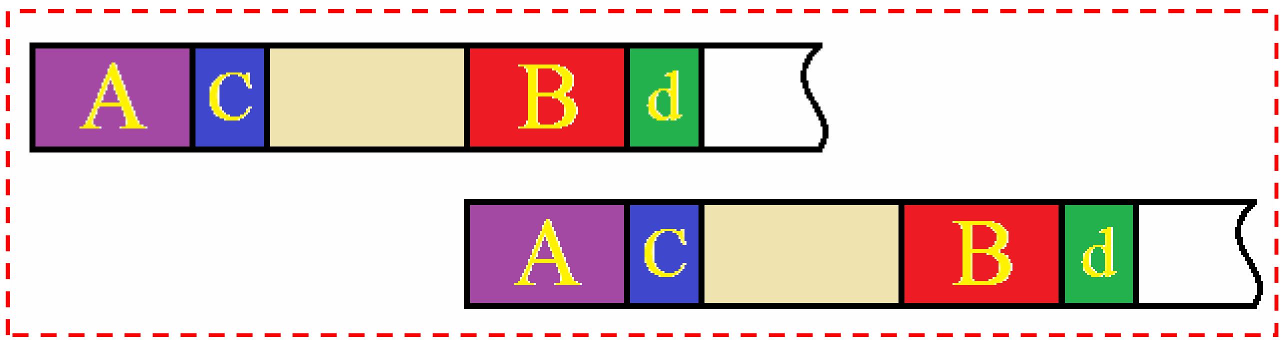
- 假设当前文本串text匹配到i位置，模式串pattern串匹配到j位置。
- BF算法中，如果当前字符匹配成功，即text[i+j]== pattern[j]，令i++，j++，继续匹配下一个字符；
  - 如果失配，即text[i+j]≠pattern[j]，令i++，j=0，即每次匹配失败的情况下，模式串pattern相对于文本串text向右移动了一位。
- KMP算法中，如果当前字符匹配成功，即text[i+j]== pattern[j]，令i++，j++，继续匹配下一个字符；
  - 如果失配，即text[i+j]≠pattern[j]，令i不变，j=next[j]，(这里，next[j]≤j-1)，即模式串pattern相对于文本串text向右移动了至少1位(移动的实际位数j-next[j]≥1)

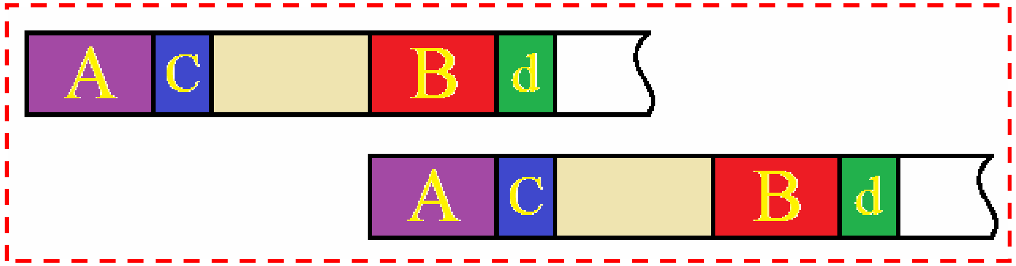
描述性说法

- 在暴力求解中，为什么模式串的索引会回溯？
  - 因为模式串存在重复字符
  - 思考：如果模式串的字符两两不相等呢？
    - 可以方便快速的编写线性时间的代码
  - 更弱一些的条件：如果模式串的首字符和其他字符不相等呢？

挖掘字符串比较的机制







分析后的结论

- 对于模式串的位置j，考察$Pattern_{j-1}=p_0p_1 \ldots p_{j-2}p_{j-1}$，查找字符串$Pattern_{j-1}$的最大相等k前缀和k后缀。
  - 注：计算next[j]时，考察的字符串是模式串的前j-1个字符，与pattern[j]无关。
- 即：查找满足条件的最大的k，使得
  - $p_0p_1 \ldots p_{k-2}p_{k-1} = p_{j-k}p_{j-k+1} \ldots p_{j-2}p_{j-1}$

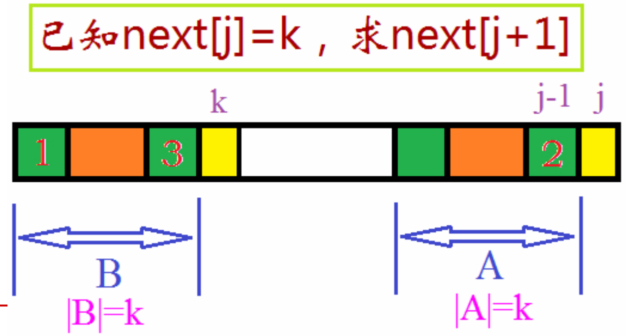
求模式串的next

模式串	a	b	a	a	b	c	a	b	a
next	-1	0	0	1	1	2	0	1	2

- 如：j=5时，考察字符串“abaab”的最大相等k前缀和k后缀

前缀串	后缀串
a	b
ab	ab
aba	aab
abaa	baab
abaab	abaab

next的递推关系



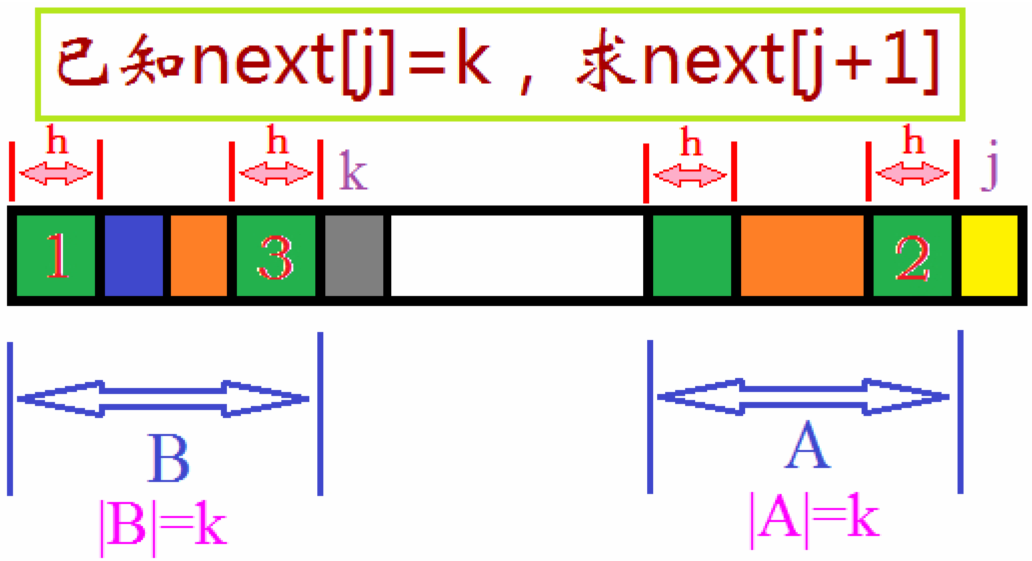
- 对于模式串的位置j，有next[j]=k，即：

  $$p_0p_1\cdots p_{k-2}p_{k-1} = p_{j-k}p_{j-k+1}\cdots p_{j-2}p_{j-1}$$

- 则，对于模式串的位置j+1，考察p_j：
- 若p[k]==p[j]
  - next[j+1]=next[j]+1
- 若p[k]≠p[j]
  - 记h=next[k]；如果p[h]==p[j],则next[j+1]=h+1，否则重复此过程。

考察不相等时，为何可以递归下去

- 若p[k]≠p[j]
  - 记h=next[k]；如果p[h]==p[j],则next[j+1]=h+1，否则重复此过程



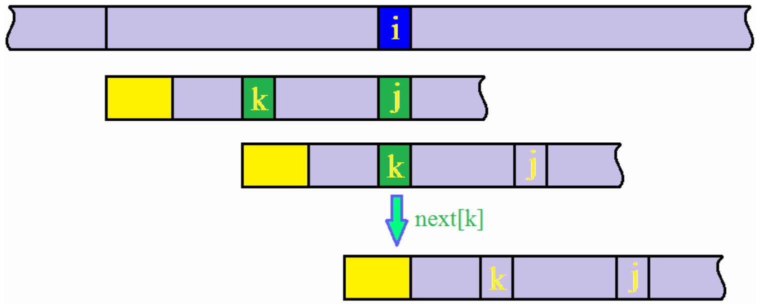
计算Next数组

```
void CalcNext(char* p, int next[])
{
    int nLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < nLen - 1)
    {
        //此刻，k即next[j-1]，且p[k]表示前缀，p[j]表示后缀
        //注：k==-1表示未找到k前缀与k后缀相等，首次分析可先忽略
        if (k == -1 || p[j] == p[k])
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else    //p[j]与p[k]失配，则继续递归前缀索引p[next[k]]
        {
            k = next[k];
        }
    }
}
```

KMP Code

```
int KMP(int n)
{
    int ans = -1;
    int i = 0;
    int j = 0;
    int pattern_len = strlen(g_pattern);
    while(i < n)
    {
        if(j == -1 || g_s[i] == g_pattern[j])
        {
            ++i; ++j;
        }
        else
            j = g_next[j];
        if(j == pattern_len)
        {
            ans = i - pattern_len;
            break;
        }
    }
    return ans;
}
```

进一步分析next



- 文本串匹配到i，模式串匹配到j，此刻，若text[i]≠pattern[j]，即失配的情况：
- 若next[j]=k，说明模式串应该从j滑动到k位置；
- 若此时满足pattern[j]==pattern[k]，因为text[i]≠pattern[j]，所以， text[i] ≠pattern[k]
  - 即i和k没有匹配，应该继续滑动到next[k]。
  - 换句话说：在原始的next数组中，若next[j]=k并且pattern[j]==pattern[k]，next[j]可以直接等于next[k]。

Next2 Code

```
void CalcNext2(char* p, int next[])
{
    int nLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < nLen - 1)
    {
        if (k == -1 || p[j] == p[k])
        {
            ++k;
            ++j;
            if(p[j] == p[k])
                next[j] = next[k];
            else
                next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
}
```

求模式串的next——变种

模式串	a	b	a	a	b	c	a	b	a
原始next	-1	0	0	1	1	2	0	1	2
新next	-1	0	-1	1	0	2	-1	0	-1

理解KMP的时间复杂度

- 我们考察模式串的“串头”和主串的对应位置(也就是暴力算法中的i)。
- 不匹配：串头后移，保证尽快结束算法；
- 匹配：串头保持不动(仅仅是i++、j++，但串头和主串的对应位置没变)，但一旦发现不匹配，会跳过匹配过的字符(next[j])。
- 最坏的情况，当串头位于N-M的位置，算法结束
- 因此，匹配的时间复杂度为O(N)，算上计算next的O(M)时间，整体时间复杂度为O(M+N)。

考察KMP的时间复杂度

- 最好情况：当模式串的首字符和其他字符都不相等时，模式串不存在相等的k前缀和k后缀，next数组全为-1
  - 一旦匹配失效，模式串直接跳过已经比较的字符。比较次数为N
- 最差情况：当模式串的首字符和其他字符全都相等时，模式串存在最长的k前缀和k后缀，next数组呈现递增样式：-1,0,1,2…
  - 举例说明

KMP最差情况

- next:-1 0 1 2 3
- 比较次数：5 1 1 1 1
- 周期：n/5
- 总次数：1.8n

- 每个周期中: m 1 1 1…
- 周期：n/m
- 总次数：$\left(2-\frac{1}{M}\right)\cdot N < 2N$

```
aaaabaaaabaaaabaaaabaaaab
aaaaa
 aaaaa
  aaaaa
   aaaaa
    aaaaa
     aaaaa
      aaaaa
       aaaaa
        aaaaa
         aaaaa
          aaaaa
```

最差情况下，变种KMP的运行情况

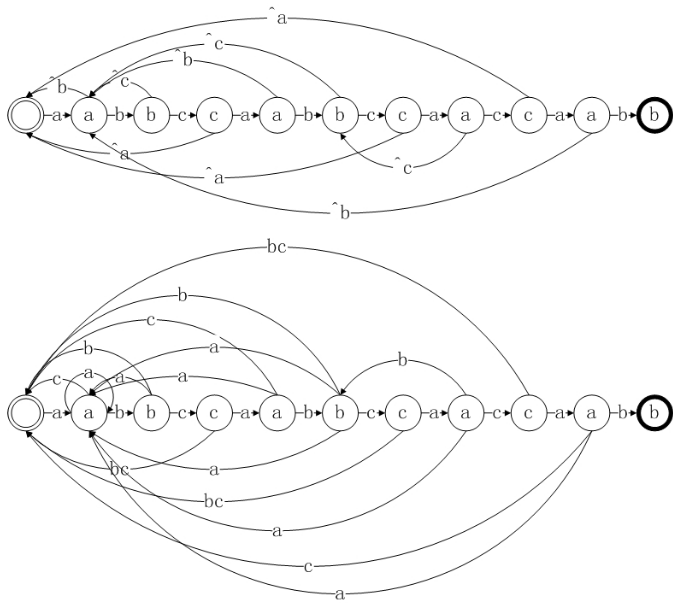
```
aaaabaaaabaaaabaaaabaaaab

aaaaa
     aaaaa
          aaaaa
```

- next:-1 -1 -1 -1 -1
- 比较次数：5
- 周期：n/5
- 总次数：n

KMP的next，实际上是建立了DFA

- 以当前位置为DFA的状态，以模式串的字符为DFA的转移条件，建立确定有穷自动机
  - Deterministic Finite Automaton



图片来自网络

附：DFA和NFA

- DFA的五要素
  - 非空有限的状态集合Q
  - 输入字母表Σ
  - 转移函数δ
  - 开始状态S
  - 结束状态F
- 对于一个给定的DFA，存在唯一一个对应的有向图；有向图的每个结点对应一个状态，每条有向边对应一种转移。习惯上将结点画成两个圈表示接受状态，一个圈表示拒绝状态。用一条没有起点的边指向起始状态。
- 如果从某个状态，在确定的输入条件下，状态转移是多个状态，则这样的自动机是非确定有穷自动机。
- 可以证明，DFA和NFA是等价的，它们识别的语言成为正则语言。

KMP应用：PowerString问题

- 给定一个长度为n的字符串S，如果存在一个字符串T，重复若干次T能够得到S，那么，S叫做周期串，T叫做S的一个周期。
- 如：字符串abababab是周期串，abab、ab都是它的周期，其中，ab是它的最小周期。
- 设计一个算法，计算S的最小周期。如果S不存在周期，返回空串。

使用next，线性时间解决问题

- 解：
- 计算S的next数组；
  - 记k=next[len-1]，p=len-k；
  - 若len能够整除p，则p就是最小周期长度，前p个字符就是最小周期。
  - 如何证明？

求字符串的最长回文子串

- 回文子串的定义：
  - 给定字符串str，若s同时满足以下条件：
    - s是str的子串
    - s是回文串
  - 则，s是str的回文子串。
- 该算法的要求，是求str中最长的那个回文子串。

解法1 – 枚举中心位置

```
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 length of 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 the even case
      if (s[i - j] != s[i + j + 1])
         break;
      if (j * 2 + 2 > max)
         max = j * 2 + 2;
   }
   return max;
}
```

算法解析 step1——预处理

- 因为回文串有奇数和偶数的不同。判断一个串是否是回文串，往往要分开编写，造成代码的拖沓。
- 一个简单的事实：长度为n的字符串，共有n-1个“邻接”，加上首字符的前面，和末字符的后面，共n+1的“空”(gap)。因此，字符串本身和gap一起，共有2n+1个，必定是奇数；
  - abbc → #a#b#b#c#
  - aba → #a#b#a#
- 因此，将待计算母串扩展成gap串，计算回文子串的过程中，只考虑奇数匹配即可。

数组int P[size]

- 字符串12212321→ S[] = "$#1#2#2#1#2#3#2#1#";
  - trick：为处理统一，最前面加一位未出现的字符，如$
- 用一个数组P[i]来记录以字符S[i]为中心的最长回文子串向左/右扩张的长度(包括S[i])，比如S和P的对应关系：
- S # 1 # 2 # 2 # 1 # 2 # 3 # 2 # 1 #
- P 1 2 1 2 5 2 1 4 1 2 1 6 1 2 1 2 1
  - P[i]-1正好是原字符串中回文串的总长度
    - 若P[i]为偶数，考察x=P[i]/2、 2*x-1
    - 思考：若P[i]为奇数呢？
      - 答：不考虑！(为何？)

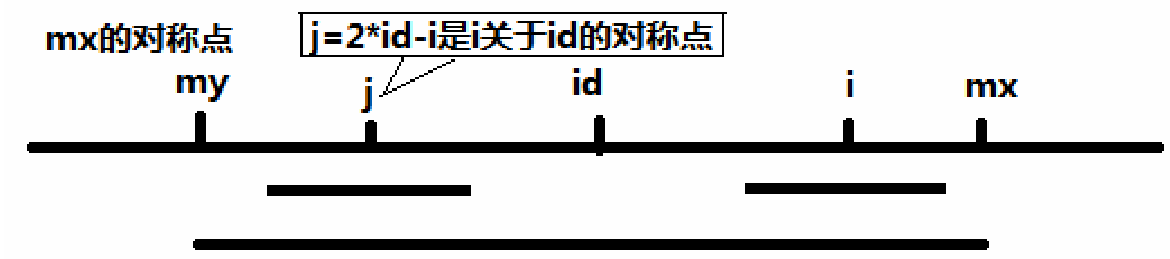
分析算法核心

S	#	1	#	2	#	2	#	1	#	2	#	3	#	2	#	1	#
P	1	2	1	2	5	2	1	4	1	2	1	6	1	2	1	2	1

- 我们的任务：假定已经得到了前i个值，考察i+1如何计算
  - 即：在P[0...i-1]已知的前提下，计算P[i]的值。换句话说，算法的核心，是**在P[0...i-1]已知的前提下，能否给P[i]的计算提供一点有用的信息**呢？
- 1、通过简单的遍历，得到i个三元组{k,P[k],k+P[k]}，0≤k≤i-1
  - trick：以k为中心的字符形成的最大回文子串的最右位置是k+P[k]-1
- 2、以k+P[k]为关键字，挑选出这i个三元组中，k+P[k]最大的那个三元组，不妨记做(id, P[id]，P[id]+id)。进一步，为了简化，记mx=P[id]+id，因此，得到三元组为(id,P[id],mx)，这个三元组的含义非常明显：所有i个三元组中，向右到达最远的位置，就是mx；
- 3、在计算P[i]的时候，考察i是否落在了区间[0,mx)中；
  - 若i在mx的右侧，说明[0,mx)没有能够控制住i，P[0..i-1]的已知，无法给P[i]的计算带来有价值信息；
  - 若i在mx的左侧，说明[0,mx)控制(也有可能部分控制)了i，现在以图示来详细考察这种情况。

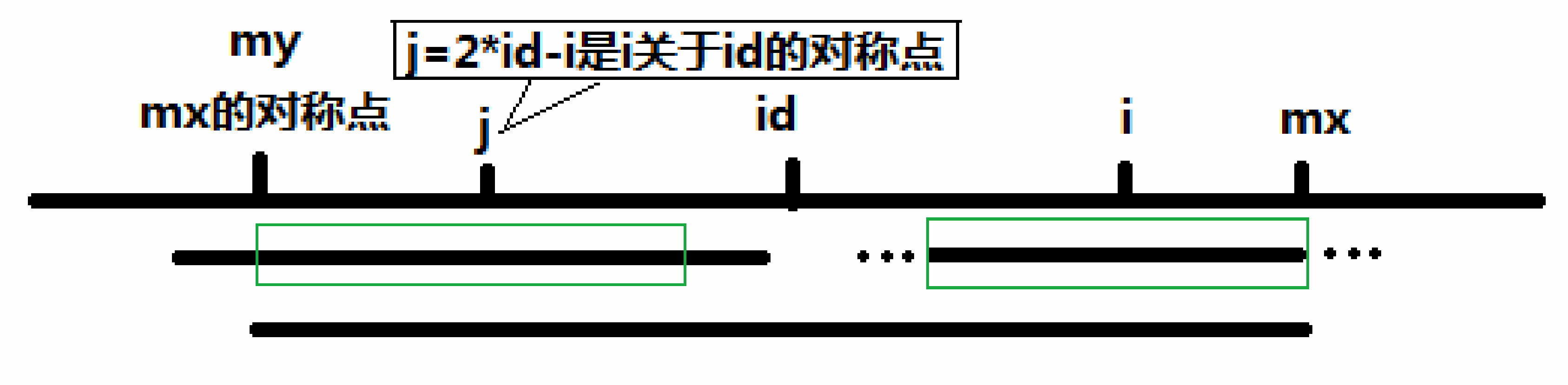
Manacher递推关系

- 记i关于id的对称点为j(=2*id-i)，若此时满足条件mx-i>P[j]；
- 记my为mx关于id的对称点(my=2*id-mx)；
- 由于以S[id]为中心的最大回文子串为S[my+1…id…mx-1]，即：S[my+1…,id]与S[id,…,mx-1]对称，而i和j关于id对称，因此P[i]=P[j](P[j]是已知的)。



Manacher递推关系

- 记i关于id的对称点为j(=2*id-i)，若此时满足条件mx-i＜P[j]：
- 记my为mx关于id的对称点(my=2*id-mx)；
- 由于以S[id]为中心的最大回文子串为S[my+1…id…mx-1]，即：S[my+1…,id]与S[id…,mx-1]对称，而i和j关于id对称，因此P[i]至少等于mx-i(图中绿色框部分)。



Manacher Code

```
void Manacher(char* s, int* P)
{
    int size = strlen(s);
    P[0] = 1;
    int id = 0;
    int mx = 1;
    for(int i = 1; i < size; i++)
    {
        if(mx > i)
        {
            P[i] = min(P[2*id-i], mx-i);
        }
        else
        {
            P[i] = 1;
        }
        for(; s[i+P[i]] == s[i-P[i]]; P[i]++);

        if(mx < i+P[i])
        {
            mx = i + P[i];
            id = i;
        }
    }
}
```

原始算法的个人改进意见

- $P[j] > mx - i$：$P[i] = mx - i$
- $P[j] < mx - i$：$P[i] = P[j]$
- $P[j] = mx - i$：$P[i] \geqslant P[j]$
  - 基本Manacher算法，红色的等号都是$\geqslant$

Manacher改进版

```
void Manacher(char* s, int* P)
{
    int size = strlen(s);
    P[0] = 1;
    int id = 0;
    int mx = 1;
    for(int i = 1; i < size; i++)
    {
        if(mx > i)
        {
            if(P[2*id-i] != mx-i)
            {
                P[i] = min(P[2*id-i], mx-i);
            }
            else
            {
                P[i] = P[2*id-i];
                for(; s[i+P[i]] == s[i-P[i]]; P[i]++);
            }
        }
        else
        {
            P[i] = 1;
            for(; s[i+P[i]] == s[i-P[i]]; P[i]++);
        }

        if(mx < i+P[i])
        {
            mx = i + P[i];
            id = i;
        }
    }
}
```

BM算法

- Boyer-Moore算法是1977年，德克萨斯大学的Robert S. Boyer教授和J Strother Moore教授发明的字符串匹配算法，拥有在最坏情况下O(N)的时间复杂度，并且，在实践中，比KMP算法的实际效能高。
- BM算法不仅效率高，而且构思巧妙，容易理解。

举例说明BM算法的运行过程

字符串	HERE IS A SIMPLE EXAMPLE
搜索词	EXAMPLE

坏字符

```
HERE IS A SIMPLE EXAMPLE

EXAMPLE
```

- 首先，"字符串"与"搜索词"头部对齐，从尾部开始比较。
- 这是一个很聪明的想法，因为如果尾部字符不匹配，那么只要一次比较，就可以知道前7个字符肯定不是要找的结果。
- “S”与“E”不匹配。这时，“S”就被称为“坏字符”(bad character)，即不匹配的字符。同时，“S”不包含在搜索词“EXAMPLE”之中，这意味着可以把搜索词直接移到“S”的后一位。
  - 还记得“暴力+KMP”中谈过的“模式串的字符两两不相等”的强要求么？放松成“模式串的首字符和其他字符不相等”，这里，迁移这个结论：模式串的尾字符和其他字符不相等。

坏字符引起的模式滑动

- 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，将搜索词后移两位，两个"P"对齐。

```
HERE IS A SIMPLE EXAMPLE
      EXAMPLE

HERE IS A SIMPLE EXAMPLE
        EXAMPLE
```

```
HERE IS A SIMPLE EXAMPLE
          EXAMPLE
```

```
HERE IS A SIMPLE EXAMPLE
            EXAMPLE
```

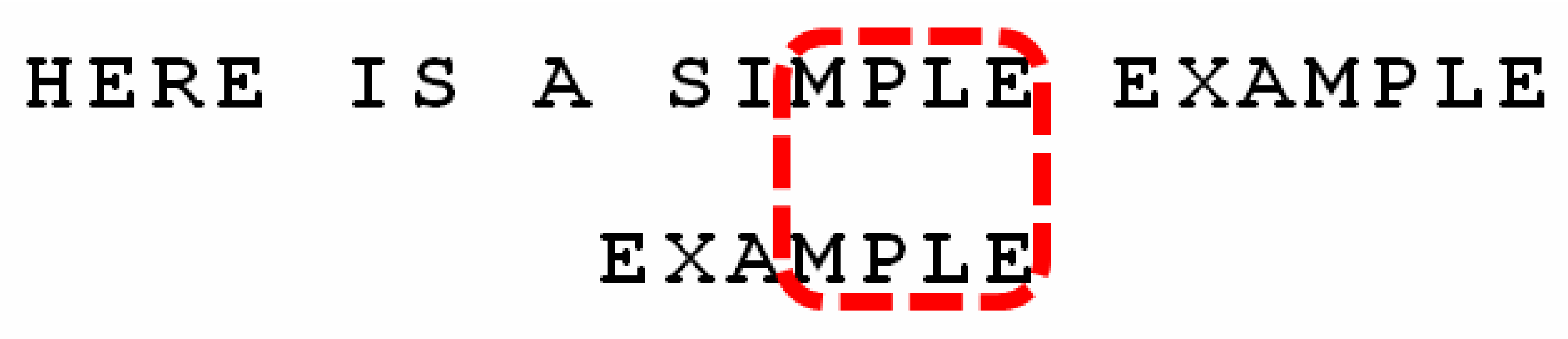
坏字符规则

- 后移位数 = 坏字符位置 - 坏字符在搜索词中的最右出现的位置
  - 如果"坏字符"不包含在搜索词之中，则最右出现位置为-1
- 以“P”为例，它作为“坏字符”，出现在搜索词的第6位(从0开始编号)，在搜索词中的最右出现位置为4，所以后移6-4=2位。
- 以前面的"S"为例，它出现在第6位，最右出现位置是-1(即未出现)，则整个搜索词后移6-(-1)=7位。

```
HERE IS A SIMPLE EXAMPLE
EXAMPLE
```

好后缀

- 依次比较，得到“MPLE”匹配，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。



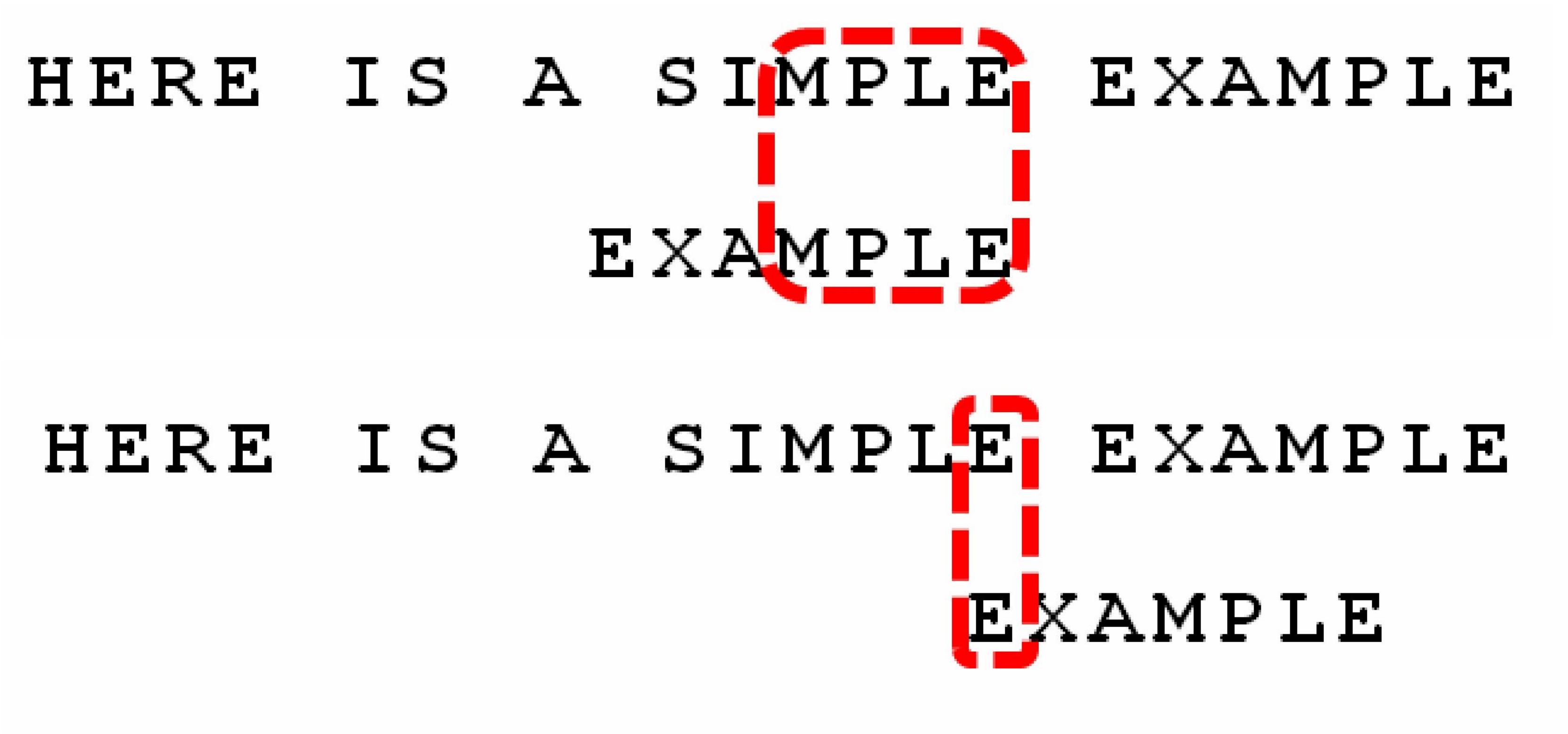
遇到坏字符

- 发现“I”与“A”不匹配：“I”是坏字符。根据坏字符规则，此时搜索词应该后移2-(-1)=3位。问题是，有没有更优的移法？

```
HERE IS A SIMPLE EXAMPLE
          EXAMPLE
```

```
HERE IS A SIMPLE EXAMPLE
             EXAMPLE
```

考虑好后缀


HERE IS A SIMPLE EXAMPLE
EXAMPLE
HERE IS A SIMPLE EXAMPLE
EXAMPLE

好后缀规则

- 后移位数=好后缀的位置-好后缀在搜索词其余部分中最右出现位置
  - 如果好后缀在搜索词中没有再次出现，则为-1。
- 所有的“好后缀”(MPLE、PLE、LE、E)之中，只有“E”在“EXAMPL”之中出现，所以后移6-0=6位。
- “坏字符规则”只能移3位，“好后缀规则”可以移6位。每次后移这两个规则之中的较大值。
- 这两个规则的移动位数，只与搜索词有关，与原字符串无关。
  - 注：KMP中，往往称作文本串、模式串。

坏字符

- 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。
  - 因为是最后一位就失配，尚未获得好后缀。

```
HERE IS A SIMPLE EXAMPLE
                 EXAMPLE

HERE IS A SIMPLE EXAMPLE
                   EXAMPLE
```

字符串查找的思考

- 字符串和树相结合，往往会产生查找思路上的变革，可查阅Trie树、后缀树(后缀数组)，用于开阔思路；
  - 一个文本文件，大约有一百万行，每行一个词，要求统计出其中最频繁出现的前10个词
  - 将在树、海量数据搜索等章节详细论述。
- 海量数据的字符串查找，往往需要Hash表。
  - 在10亿个URL中，查找某URL的出现位置
  - 千万别回答：计算待查找字符串的next数组，用KMP算法。

我们在这里

- 更多算法面试题在7 | 七月算法官网
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @研究者July
  - @七月问答
  - @邹博_机器学习

参考文献

- https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.01.md(字符串循环左移)
- http://blog.csdn.net/morewindows/article/details/7370155/(全排列)
- http://bbs.dlut.edu.cn/bbstcon.php?board=Competition&gid=23474(最长回文子串)
- http://www.felix021.com/blog/read.php?2040 (最长回文子串)
- http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html (最长回文子串)
- http://baike.baidu.com/view/1436430.htm(字符串匹配)
- http://blog.csdn.net/geniusluzh/article/details/8483010 (KMP)
- http://www.cnblogs.com/waytofall/archive/2012/10/27/2742163.html (KMP)
- http://www.cppblog.com/abilitytao/archive/2009/08/01/91865.html(KMP)
- http://blog.csdn.net/joylnwang/article/details/6778316(KMP)
- http://zh.wikipedia.org/wiki/DFA(DFA)
- http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=2177(周期串)
- http://blog.csdn.net/iJuliet/article/details/4200771(BM算法)
- http://kb.cnblogs.com/page/176945/(BM算法)
- http://blog.csdn.net/fanzitao/article/details/8042015(后缀树)
- http://blog.csdn.net/v_july_v/article/details/6897097(后缀树)
- http://baike.baidu.com/view/1240197.htm(后缀数组)

感谢大家

恳请大家批评指正！