字符串

七月算法 邹博

2015年10月18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_____。

  - 字符串问题的晦涩代表：KMP、Manacher

主要内容

- 需要掌握的内容
  - 字符串循环左移
  - LCS最长递增子序列
  - 字符串全排列
    - 递归、非递归
  - KMP
  - Huffman编码
- 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)
- 该问题可以作为“完美洗牌”算法的子算法。

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);
}
```

LCS的定义

- 最长公共子序列，即Longest Common Subsequence，LCS。
- 一个序列S任意删除若干个字符得到新序列T，则T叫做S的子序列；
- 两个序列X和Y的公共子序列中，长度最长的那个，定义为X和Y的最长公共子序列。
  - 字符串13455与245576的最长公共子序列为455
  - 字符串acdfg与adfc的最长公共子序列为adf
- 注意区别最长公共子串(Longest Common Substring)
  - 最长公共字串要求连续

LCS的意义

- 求两个序列中最长的公共子序列算法，广泛的应用在图形相似处理、媒体流的相似比较、计算生物学方面。生物学家常常利用该算法进行基因序列比对，由此推测序列的结构、功能和演化过程。
- LCS可以描述两段文字之间的“相似度”，即它们的雷同程度，从而能够用来辨别抄袭。另一方面，对一段文字进行修改之后，计算改动前后文字的最长公共子序列，将除此子序列外的部分提取出来，这种方法判断修改的部分，往往十分准确。简而言之，百度知道、百度百科都用得上。

暴力求解：穷举法

- 假定字符串X，Y的长度分别为m，n；
- X的一个子序列即下标序列{1, 2, …, m}的严格递增子序列，因此，X共有2^m个不同子序列；同理，Y有2^n个不同子序列，从而穷举搜索法需要指数时间$O(2^m \cdot 2^n)$；
- 对X的每一个子序列，检查它是否也是Y的子序列，从而确定它是否为X和Y的公共子序列，并且在检查过程中选出最长的公共子序列；
- 显然，不可取。

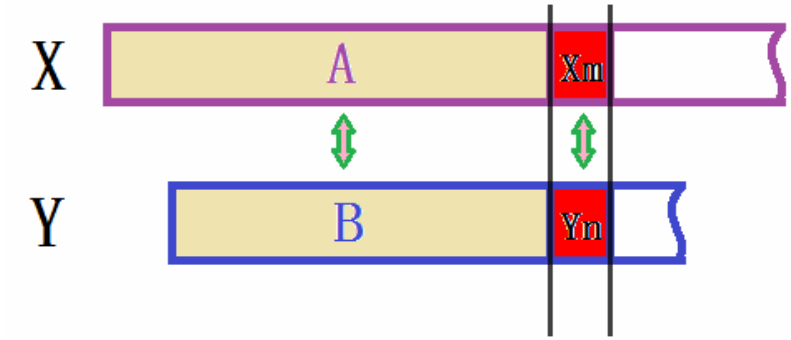
LCS的记号

- 字符串X，长度为m，从1开始数；
- 字符串Y，长度为n，从1开始数；
- Xi=<x1，…，xi>即X序列的前i个字符 (1≤i≤m)(Xi不妨读作“字符串X的i前缀”)
- Yj=<y1，…，yj>即Y序列的前j个字符 (1≤j≤n) (字符串Y的j前缀)；
- LCS(X , Y) 为字符串X和Y的最长公共子序列，即为Z=<z1，…，zk>。
  - 注：不严格的表述。事实上，X和Y的可能存在多个子串，长度相同并且最大，因此，LCS(X,Y)严格的说，是个字符串集合。即：Z∈ LCS(X , Y) .

LCS解法的探索： $x_m=y_n$

- 若$x_m=y_n$(最后一个字符相同)，则：X_m与Y_n的最长公共子序列Z_k的最后一个字符必定为$x_m(=y_n)$。
  - $z_k=x_m=y_n$
  - $LCS(X_m , Y_n) = LCS(X_{m-1} , Y_{n-1}) + x_m$

结尾字符相等，则$LCS(X_m,Y_n)=LCS(X_{m-1},Y_{n-1})+x_m$



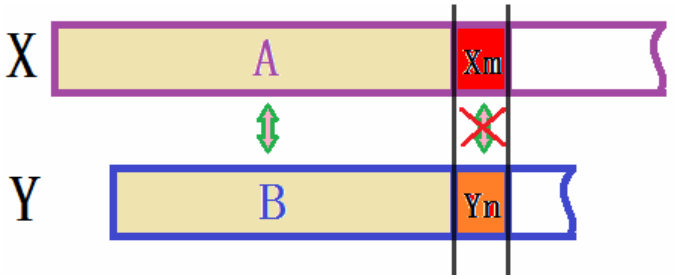
- 记$LCS(X_m,Y_n)=W+x_m$，则W是X_{m-1}的子序列；同理，W是Y_{n-1}的子序列；因此，W是X_{m-1}和Y_{n-1}的公共子序列。
  - 反证：若W不是X_{m-1}和Y_{n-1}的最长公共子序列，不妨记$LCS(X_{m-1},Y_{n-1})=W'$，且$|W'|>|W|$；那么，将W换成W'，得到更长的$LCS(X_m,Y_n)=W'x_m$，与题设矛盾。

举例： $x_m=y_n$

	1	2	3	4	5	6	7
X	B	D	C	A	B	A	
Y	A	B	C	B	D	A	B

- 对于上面的字符串X和Y：
- $x_3=y_3$=‘C’， 则： LCS(BDC,ABC)=LCS(BD,AB)+‘C’
- $x_5=y_4$=‘B’， 则： LCS(BDCAB,ABCB)=LCS(BDCA,ABC)+‘B’

LCS的探索：$x_m \neq y_n$



- 若$x_m \neq y_n$，则：
  - 要么：$LCS(X_m,Y_n)=LCS(X_{m-1}, Y_n)$
  - 要么：$LCS(X_m,Y_n)=LCS(X_m, Y_{n-1})$
- 证明：
  - 令$Z_k=LCS(X_m,Y_n)$；由于$x_m \neq y_n$则$z_k \neq x_m$与$z_k \neq y_n$至少有一个必然成立，不妨假定$z_k \neq x_m$($z_k \neq y_n$的分析与之类似)
  - 因为$z_k \neq x_m$，则最长公共子序列Z_k是X_{m-1}和Y_n得到的，即：$Z_k=LCS(X_{m-1},Y_n)$
  - 同理，若$z_k \neq y_n$，则$Z_k=LCS(X_m,Y_{n-1})$
- 即，若$x_m \neq y_n$，则：
  - $LCS(X_m,Y_n)= \max\{LCS(X_{m-1},Y_n),LCS(X_m,Y_{n-1})\}$

举例： $x_m \neq y_n$

	1	2	3	4	5	6	7
X	B	D	C	A	B	A	
Y	A	B	C	B	D	A	B

- 对于字符串X和Y：
- $x_2 \neq y_2$，则：LCS(BD,AB)=max{ LCS(BD,A), LCS(B, AB) }
- $x_4 \neq y_5$，则：LCS(BDCA,ABCBD)=

  max{ LCS(BDCA,ABCB), LCS(BDC,ABCBD) }

LCS分析总结

$$LCS(X_m, Y_n) = \begin{cases} LCS(X_{m-1}, Y_{n-1}) + x_m & \text{当} x_m = y_n \\ \max\{LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1})\} & \text{当} x_m \neq y_n \end{cases}$$

- 显然，属于动态规划问题。

算法中的数据结构：长度数组

- 使用二维数组C[m,n]
- c[i,j]记录序列Xi和Yj的最长公共子序列的长度。
  - 当i=0或j=0时，空序列是Xi和Yj的最长公共子序列，故c[i,j]=0。

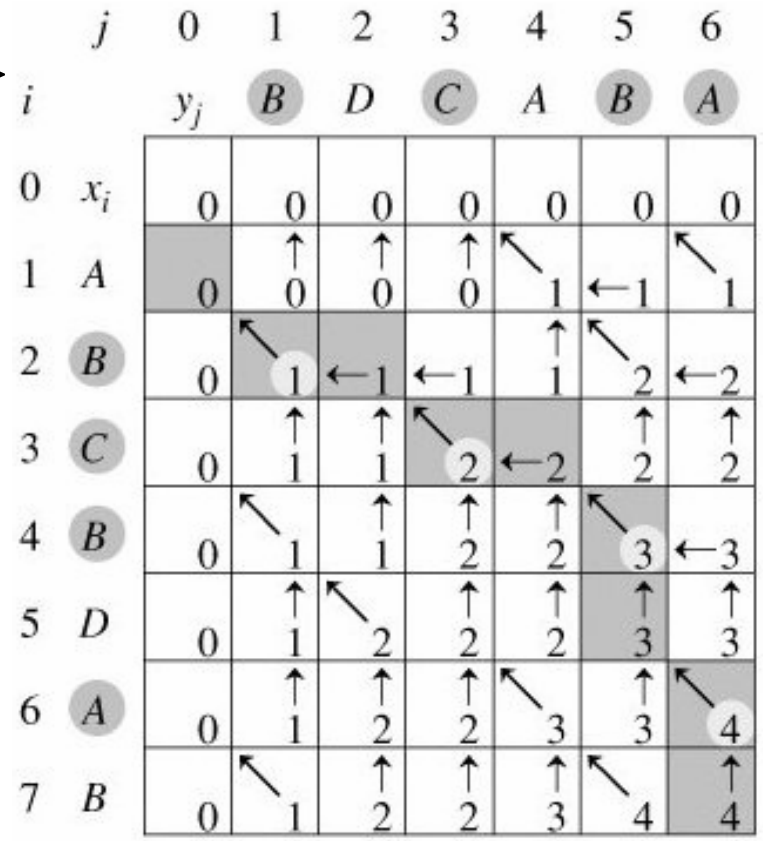
$$c(i,j)=\begin{cases}0 & \text{当}i=0\text{或者}j=0\\ c(i-1,j-1)+1 & \text{当}i>0,j>0,\text{且}x_i=y_j\\ \max\{c(i-1,j),c(i,j-1)\} & \text{当}i>0,j>0,\text{且}x_i\neq y_j\end{cases}$$

算法中的数据结构：方向变量

- 使用二维数据B[m,n]，其中，b[i,j]标记c[i,j]的值是由哪一个子问题的解达到的。即c[i,j]是由c[i-1,j-1]+1或者c[i-1,j]或者c[i,j-1]的哪一个得到的。取值范围为Left，Top，LeftTop三种情况。

实例

- X=< A，B，C，B，D，A，B >
- Y=< B，D，C，A，B，A >



i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const char* str1 = "TCGGATCGACTT";
    const char* str2 = "AGCCTACGTA";
    string str;
    LCS(str1, str2, str);
    cout << str.c_str() << endl;
    return 0;
}
```

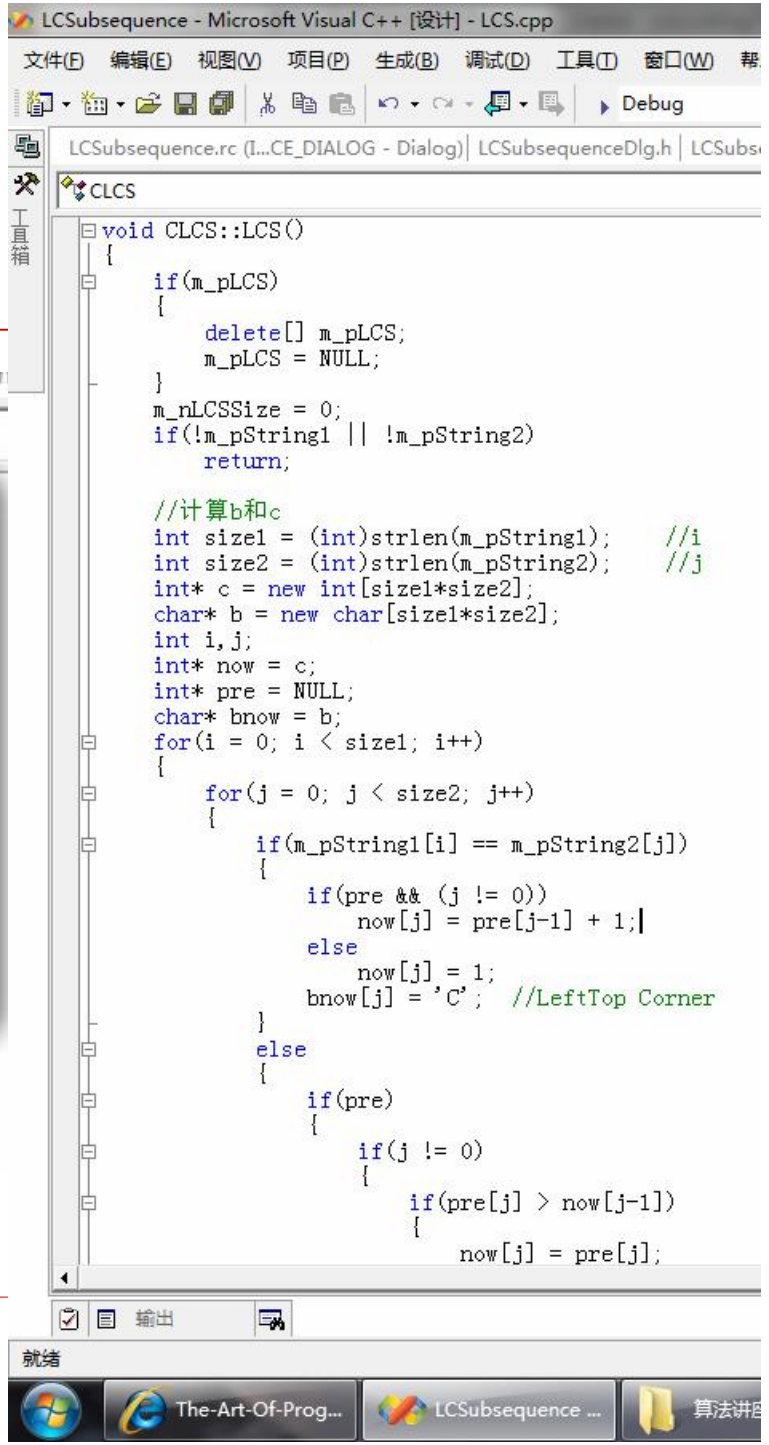
```
void LCS(const char* str1, const char* str2, string& str)
{
    int size1 = (int)strlen(str1);
    int size2 = (int)strlen(str2);
    const char* s1 = str1-1;    //从1开始数，方便后面的代码编写
    const char* s2 = str2-1;
    vector<vector<int> > chess(size1+1, vector<int>(size2+1));
    int i,j;
    for(i = 0; i <= size1; i++) //第0列
        chess[i][0] = 0;
    for(j = 0; j <= size2; j++) //第0行
        chess[0][j] = 0;

    for(i = 1; i <= size1; i++)
    {
        for(j = 1; j <= size2; j++)
        {
            if(s1[i] == s2[j])  //i,j相等
                chess[i][j] = chess[i-1][j-1] + 1;
            else
                chess[i][j] = max(chess[i][j-1], chess[i-1][j]);
        }
    }

    i = size1;
    j = size2;
    while((i != 0) && (j != 0))
    {
        if(s1[i] == s2[j])
        {
            str.push_back(s1[i]);
            i--;
            j--;
        }
        else
        {
            if(chess[i][j-1] > chess[i-1][j])
                j--;
            else
                i--;
        }
    }
    reverse(str.begin(), str.end());
}
```

算法实现Demo


LCSubsequence
字符串1
BCDABAB
字符串2
CBACBAABA
最长公共子序列
CABAB
计算
关闭
Index = size1*size2-1;
SSize = c[nIndex];
S = new char[m_nLCSSize+1];
S[m_nLCSSize] = 0;


LCSubsequence - Microsoft Visual C++ [设计] - LCS.cpp
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) 工具(T) 窗口(W)
Debug
LCSubsequence.rc (I...CE_DIALOG - Dialog) LCSubsequenceDlg.h
CLCS
void CLCS::LCS()
{
if(m_pLCS)
{
delete[] m_pLCS;
m_pLCS = NULL;
}
m_nLCSSize = 0;
if(!m_pString1 || !m_pString2)
return;
//计算b和c
int size1 = (int)strlen(m_pString1); //i
int size2 = (int)strlen(m_pString2); //j
int* c = new int[size1*size2];
char* b = new char[size1*size2];
int i,j;
int* now = c;
int* pre = NULL;
char* bnow = b;
for(i = 0; i < size1; i++)
{
for(j = 0; j < size2; j++)
{
if(m_pString1[i] == m_pString2[j])
{
if(pre && (j != 0))
now[j] = pre[j-1] + 1;
else
now[j] = 1;
bnow[j] = 'C'; //LeftTop Corner
}
else
{
if(pre)
{
if(j != 0)
{
if(pre[j] > now[j-1])
{
now[j] = pre[j];
输出
就绪
The-Art-Of-Prog...
LCSubsequence ...

进一步思考的问题

- 方向数组b是完全可以省略的：
  - 数组元素c[i,j]的值仅由c[i-1,j-1]，c[i-1,j]和c[i,j-1]三个值之一确定，因此，在计算中，可以临时判断c[i,j]的值是由c[i-1,j-1]，c[i-1,j]和c[i,j-1]中哪一个数值元素所确定，代价是O(1)时间。
- 若只计算LCS的长度，则空间复杂度为O(min(m, n))。
  - 在计算c[i,j]时，只用到数组c的第i行和第i-1行。因此，只要用2行的数组空间就可以计算出最长公共子序列的长度。

最大公共子序列的多解性：求所有的LCS

$$LCS(X_m,Y_n)=\begin{cases} LCS(X_{m-1},Y_{n-1})+x_m & \text{当} x_m = y_n \\ \max\{LCS(X_{m-1},Y_n),LCS(X_m,Y_{n-1})\} & \text{当} x_m \neq y_n \end{cases}$$

- 当$x_m \neq y_n$时：

  若$LCS(X_{m-1},Y_n)=LCS(X_m,Y_{n-1})$，会导致多解：有多个最长公共子序列，并且它们的长度相等。
- B的取值范围从1,2,3扩展到1,2,3,4
- 深度/广度优先搜索

		Yj	A	B	C	D	C	D	A	B
		0	1	2	3	4	5	6	7	8
Xi	0	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
B	1	0(0)	0(4)	1(1)	1(3)	1(3)	1(3)	1(3)	1(3)	1(1)
A	2	0(0)	1(1)	1(4)	1(4)	1(4)	1(4)	1(4)	2(1)	2(3)
D	3	0(0)	1(2)	1(4)	1(4)	2(1)	2(3)	2(1)	2(4)	2(4)
C	4	0(0)	1(2)	1(4)	2(1)	2(4)	3(1)	3(3)	3(3)	3(3)
D	5	0(0)	1(2)	1(4)	2(2)	3(1)	3(4)	4(1)	4(3)	4(3)
C	6	0(0)	1(2)	1(4)	2(1)	3(2)	4(1)	4(4)	4(4)	4(4)
B	7	0(0)	1(2)	2(1)	2(4)	3(2)	4(2)	4(4)	4(4)	5(1)
A	8	0(0)	1(1)	2(2)	2(4)	3(2)	4(2)	4(4)	5(1)	5(4)

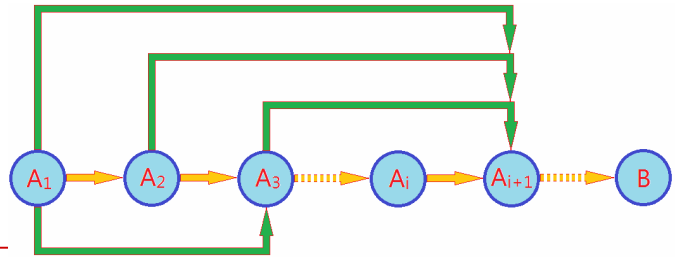
LCS的应用：最长递增子序列LIS

- Longest Increasing Subsequence
- 给定一个长度为N的数组，找出一个最长的单调递增子序列。
- 例如：给定数组{5，6，7，1，2，8}，则其最长的单调递增子序列为{5，6，7，8}，长度为4。
  - 分析：其实此LIS问题可以转换成最长公子序列问题，为什么呢？

使用LCS解LIS问题

- 原数组为A {5，6，7，1，2，8}
- 排序后：A’{1，2，5，6，7，8}
- 因为，原数组A的子序列顺序保持不变，而且排序后A’本身就是递增的，这样，就保证了两序列的最长公共子序列的递增特性。如此，若想求数组A的最长递增子序列，其实就是求数组A与它的排序数组A’的最长公共子序列。
  - 此外，本题也可以直接使用动态规划/贪心法来求解

附：LIS的动态规划解法



- 长度为N的数组记为A=$\{a_0a_1a_2...a_{n-1}\}$；
- 记A的前i个字符构成的前缀串为A_i=$a_0a_1a_2...a_{i-1}$，以a_i结尾的最长递增子序列记做L_i，其长度记为b[i]；
- 假定已经计算得到了b[0,1…,i-1]，如何计算b[i]呢？
  - 已知L_0 L_1…… L_{i-1}的前提下，如何求L_i？

附：求解LIS

Array	1	4	6	2	8	9	7
LIS	1	2	3	2	4	5	4

- 根据定义，L_i必须以a_i结尾；
- 如果将a_i分别缀到$L_0 L_1 \ldots\ldots L_{i-1}$后面，是否允许呢？
  - 如果$a_i \geqslant a_j$，则可以将a_i缀到L_j的后面，得到比L_j更长的字符串。
- 从而：$b[i]=\{\max(b[j])+1, 0 \leqslant j < i \text{且} a_j \leqslant a_i\}$
  - 计算b[i]：遍历在i之前的所有位置j，找出满足条件$a_j \leqslant a_i$的最大的b[j]+1；
  - 计算得到b[0…n-1]后，遍历所有的b[i]，找出最大值即为最大递增子序列的长度。
- 时间复杂度为$O(N^2)$。

附：Code

```
#include <vector>
#include <algorithm>
using namespace std;

int LIS(const int* p, int length, int* pre, int& nIndex)
{
    int* longest = new int[length];
    int i,j;

    for(i = 0; i < length; i++)
    {
        longest[i] = 1;
        pre[i] = -1;
    }

    int nLis = 1;
    nIndex = 0;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                if(longest[i] < longest[j]+1)
                {
                    longest[i] = longest[j]+1;
                    pre[i] = j;
                }
            }
        }
        if(nLis < longest[i])
        {
            nLis = longest[i];
            nIndex = i;
        }
    }

    delete[] longest;

    return nLis;
}

void GetLIS(const int* array, const int* pre, int nIndex, vector<int>& lis)
{
    while(nIndex >= 0)
    {
        lis.push_back(array[nIndex]);
        nIndex = pre[nIndex];
    }
    reverse(lis.begin(), lis.end());
}

void Print(int* p, int size)
{
    for(int i = 0; i < size; i++)
        cout << p[i] << '\t';
    cout << '\n';
}

int _tmain(int argc, _TCHAR* argv[])
{
    int array[] = {1,4,5,6,2,3,8,9,10,11,12,12,1};
    int size = sizeof(array)/sizeof(int);
    int* pre = new int[size];
    int nIndex;
    int max = LIS(array, size, pre, nIndex);
    vector<int> lis;
    GetLIS(array, pre, nIndex, lis);
    delete[] pre;
    cout << max << endl;
    Print(&lis.front(), (int)lis.size());
    return 0;
}
```

附：Code *split*

```
int _tmain(int argc, _TCHAR* argv[])
{
    int array[] = {1,4,5,6,2,3,8,9,10,11,12,12,1};
    int size = sizeof(array)/sizeof(int);
    int* pre = new int[size];
    int nIndex;
    int max = LIS(array, size, pre, nIndex);
    vector<int> lis;
    GetLIS(array, pre, nIndex, lis);
    delete[] pre;
    cout << max << endl;
    Print(&lis.front(), (int)lis.size());
    return 0;
}
```

```
void GetLIS(const int* array, const int* pre,
            int nIndex, vector<int>& lis)
{
    while(nIndex >= 0)
    {
        lis.push_back(array[nIndex]);
        nIndex = pre[nIndex];
    }
    reverse(lis.begin(), lis.end());
}
```

```
#include <vector>
#include <algorithm>
using namespace std;

int LIS(const int* p, int length, int* pre, int& nIndex)
{
    int* longest = new int[length];
    int i,j;

    for(i = 0; i < length; i++)
    {
        longest[i] = 1;
        pre[i] = -1;
    }

    int nLis = 1;
    nIndex = 0;
    for(i = 1; i < length; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(p[j] <= p[i])
            {
                if(longest[i] < longest[j]+1)
                {
                    longest[i] = longest[j]+1;
                    pre[i] = j;
                }
            }
        }
        if(nLis < longest[i])
        {
            nLis = longest[i];
            nIndex = i;
        }
    }

    delete[] longest;

    return nLis;
}
```

```
void Print(int* p, int size)
{
    for(int i = 0; i < size; i++)
        cout << p[i] << '\t';
    cout << '\n';
}
```

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

```
void Print(const int* a, int size)
{
    for(int i = 0; i < size; i++)
        cout << a[i] << ' ';
    cout << endl;
}

void Permutation(int* a, int size, int n)
{
    if(n == size-1)
    {
        Print(a, size);
        return;
    }
    for(int i = n; i < size; i++)
    {
        swap(a[i], a[n]);
        Permutation(a, size, n+1);
        swap(a[i], a[n]);
    }
}

int main(int argc,char* argv[])
{
    int a[] = {1,2,3,4};
    Permutation(a, sizeof(a)/sizeof(int), 0);
    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。

Code

```
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 IsDuplicate(const int* a, int n, int t)
{
    while(n < t)
    {
        if(a[n] == a[t])
            return false;
        n++;
    }
    return true;
}

void Permutation(int* a, int size, int n)
{
    if(n == size-1)
    {
        Print(a, size);
        return;
    }
    for(int i = n; i < size; i++)
    {
        if(!IsDuplicate(a, n, i))//a[i]是否与[n,i)重复
            continue;
        swap(a[i], a[n]);
        Permutation(a, size, n+1);
        swap(a[i], a[n]);
    }
}

int main(int argc,char* argv[])
{
    int a[] = {1,2,2,3};
    Permutation(a, sizeof(a)/sizeof(int), 0);
    return 0;
}
```

重复字符的全排列递归算法时间复杂度

- ∵ f(n) = n*f(n-1) + n^2
- ∵ f (n-1)=(n-1)*f(n-2) + (n-1)^2
- ∴ f(n) = n*((n-1)*f(n-2) + (n-1)^2) + n^2
- ∵ f(n-2) = (n-2)*f(n-3) + (n-2)^2
- ∴ 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
- =.....
- < n! + n! + n! + n! + ... + n!
- = (n+1)*n!
- 时间复杂度为O((n+1)!)
  - 注：当n足够大时：n! > n+1

空间换时间

```
void Permutation(char* a, int size, int n)
{
    if(n == size-1)
    {
        Print(a, size);
        return;
    }
    int dup[256] = {0};
    for(int i = n; i < size; i++)
    {
        if(dup[a[i]] == 1)
            continue;
        dup[a[i]] = 1;
        swap(a[i], a[n]);
        Permutation(a, size, n+1);
        swap(a[i], a[n]);
    }
}

int main(int argc,char* argv[])
{
    char str[] = "abbc";
    Permutation(str, sizeof(str)/sizeof(char)-1, 0);
    return 0;
}
```

空间换时间的方法

- 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]；
- 如果是浮点数或其他结构，考虑使用Hash。
  - 事实上，如果发现整数间变化太大，也应该考虑使用Hash；
  - 可以认为整数/字符的情况是最朴素的Hash。

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 Reverse(int* from, int* to)
{
    int t;
    while(from < to)
    {
        t = *from;
        *from = *to;
        *to = t;
        from++;
        to--;
    }
}
```

```
bool GetNextPermutation(int* a, int size)
{
    //后找
    int i = size-2;
    while((i >= 0) && (a[i] >= a[i+1]))
        i--;
    if(i < 0)
        return false;

    //小大
    int j = size-1;
    while(a[j] <= a[i])
        j--;

    //交换
    swap(a[j], a[i]);

    //翻转
    Reverse(a+i+1, a+size-1);
    return true;
}

int main(int argc,char* argv[])
{
    int a[] = {1,2,2,3};
    int size = sizeof(a)/sizeof(int);
    Print(a, size);
    while(GetNextPermutation(a, size))
        Print(a, size);
    return 0;
}
```

几点说明

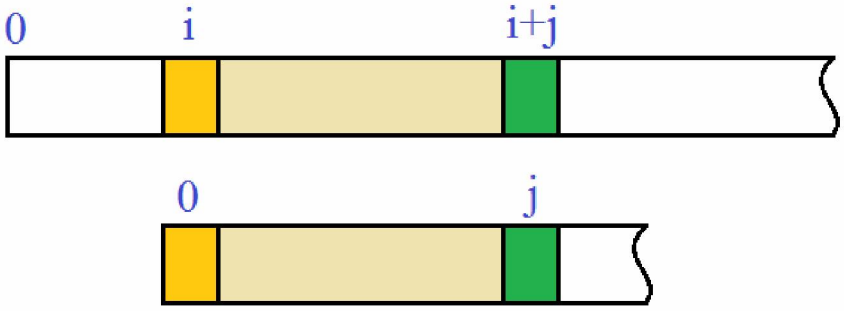
- 下一个排列算法能够天然的解决重复字符的问题！
  - 不妨还是考察926520的下一个字符串
- C++STL已经在Algorithm中集成了next_permutation
- 可以将给定的字符串A[0...N-1]首先升序排序，然后依次调用next_permutation直到返回false，即完成了非递归的全排列算法。

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(p);
    int nLast = (int)strlen(s) - size;
    while((i <= nLast) && (j < size))
    {
        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 >= size)
        return i;
    return -1;
}
```

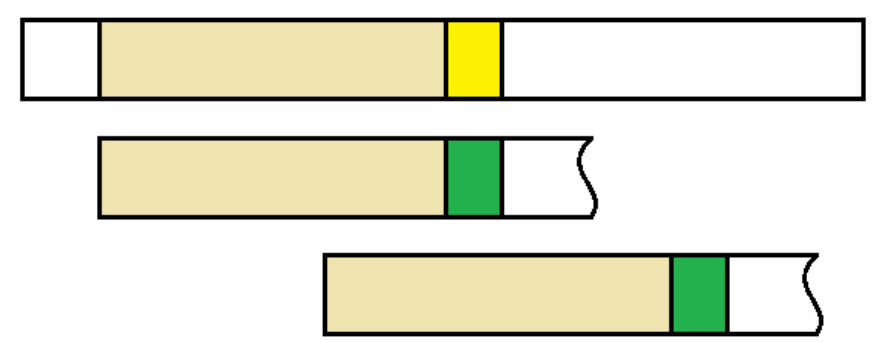
分析BF与KMP的区别

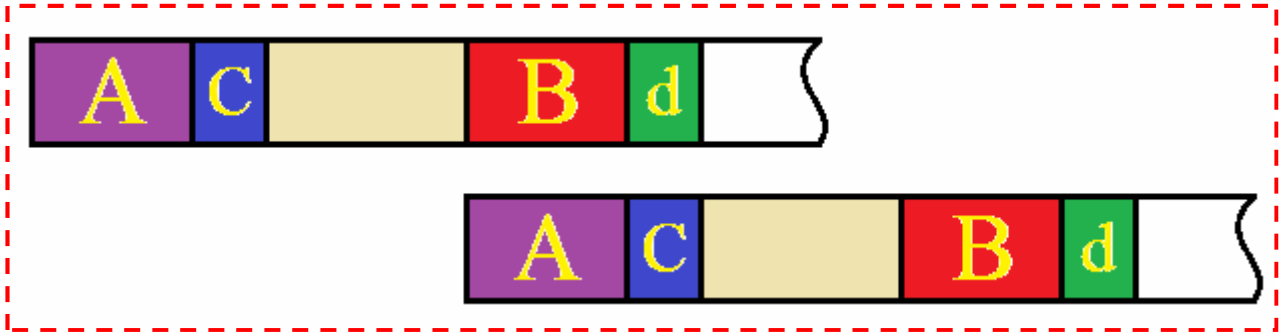
- 假设当前文本串text匹配到i位置，模式串pattern串匹配到j位置。
- BF算法中，如果当前字符匹配成功，即text[i+j]==pattern[j]，令i++，j++，继续匹配下一个字符；
  - 如果失配，即text[i+j]≠pattern[j]，令i++，j=0，即每次匹配失败的情况下，模式串pattern相对于文本串text向右移动了一位。
- KMP算法中，如果当前字符匹配成功，即text[i+j]== pattern[j]，令i++，j++，继续匹配下一个字符；
  - 如果失配，即text[i+j]≠pattern[j]，令i不变，j=next[j]，(这里，next[j]≤j-1)，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

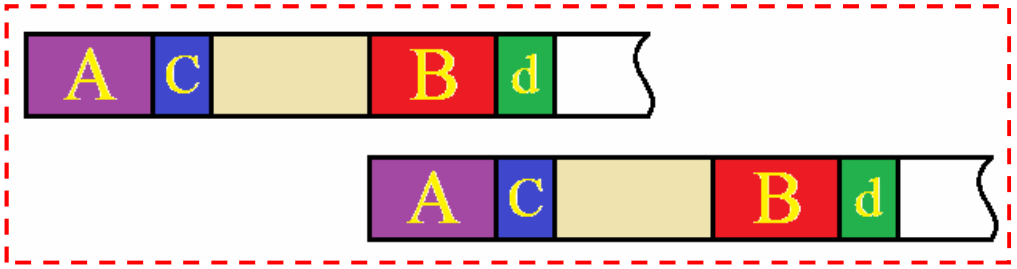
- 在暴力求解中，为什么模式串的索引会回溯？
  - 因为模式串存在重复字符
  - 思考：如果模式串的字符两两不相等呢？
    - 可以方便快速的编写线性时间的代码
  - 更弱一些的条件：如果模式串的首字符和其他字符不相等呢？

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

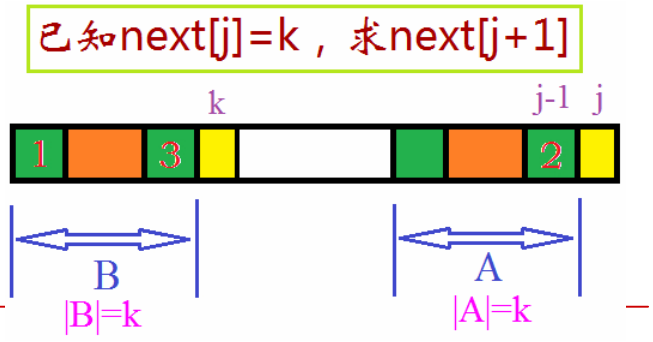
求模式串的next

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



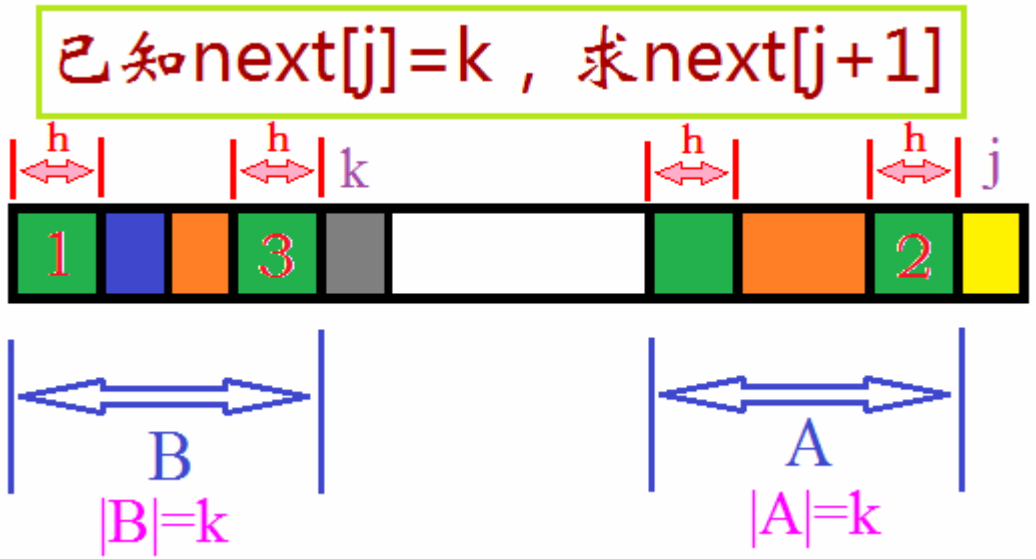
- 对于模式串的位置j，有next[j]=k，即：

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

- 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



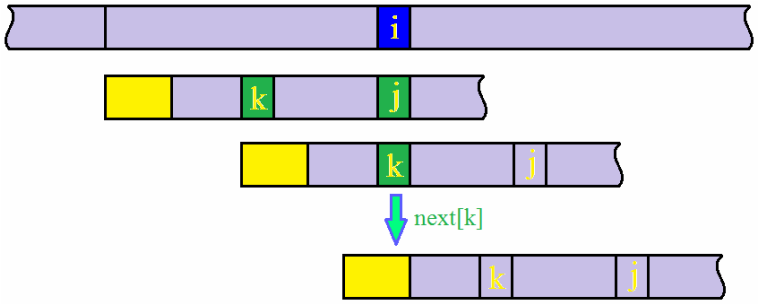
计算Next数组

```
void GetNext(char* p,int next[])
{
    int nLen = (int)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])
        {
            ++j;
            ++k;
            next[j] = k;
        }
        else    //p[j]与p[k]失配，则继续递归计算前缀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]。

Code2

```
void GetNext2(char* p,int next[])
{
    int nLen = (int)strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < nLen - 1)
    {
        if (k == -1 || p[j] == p[k])
        {
            ++j;
            ++k;
            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

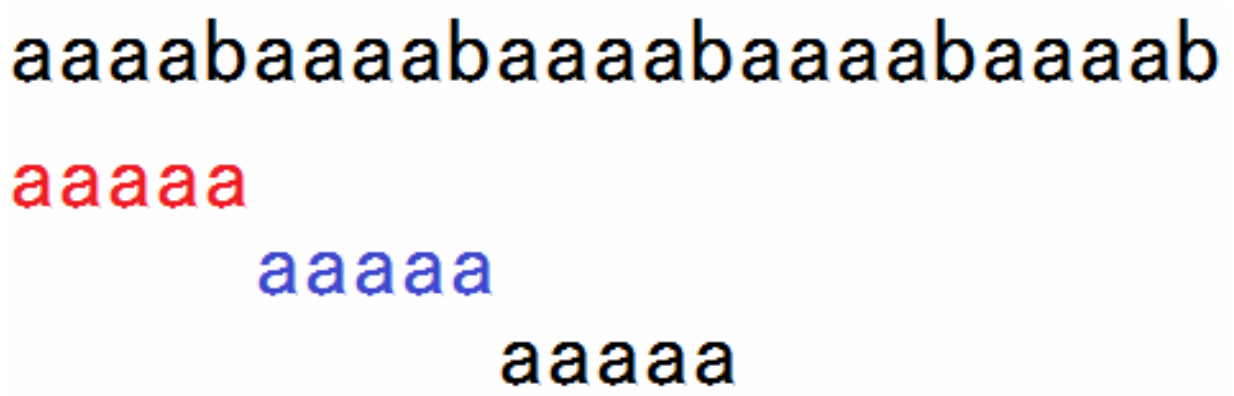
- 最好情况：当模式串的首字符和其他字符都不相等时，模式串不存在相等的k前缀和k后缀，next数组全为-1
  - 一旦匹配失效，模式串直接跳过已经比较的字符。比较次数为N
- 最差情况：当模式串的首字符和其他字符全都相等时，模式串存在最长的k前缀和k后缀，next数组呈现递增样式：-1,0,1,2...
  - 举例说明

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：$\left(2-\frac{1}{M}\right)\cdot N<2N$

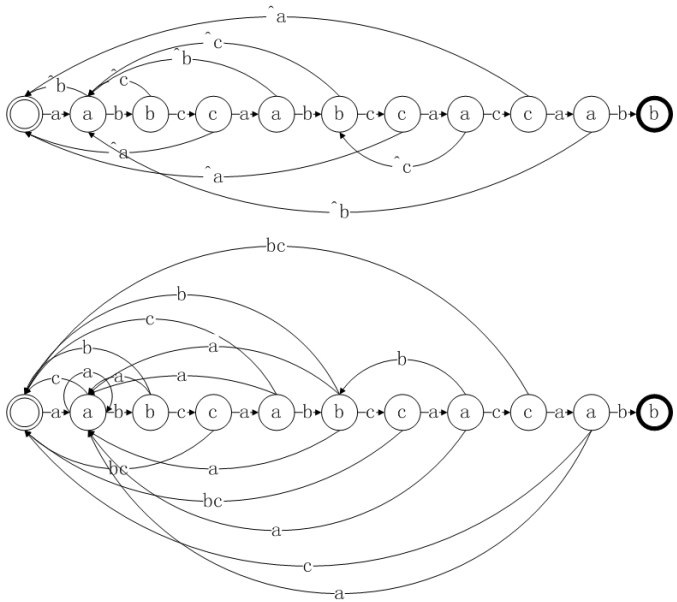
最差情况下，变种KMP的运行情况



- 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

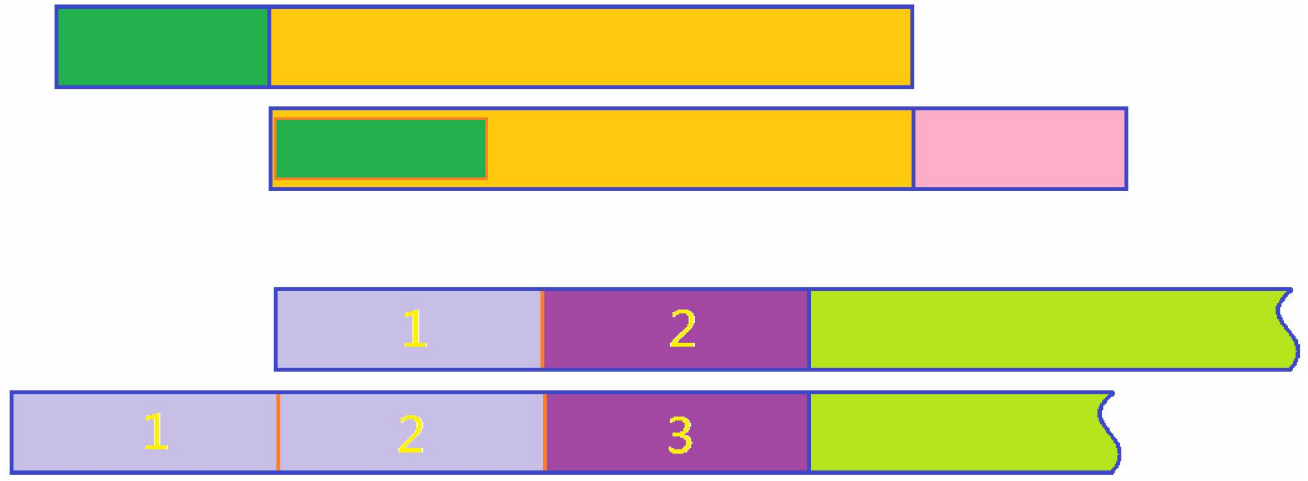
- 给定一个长度为n的字符串S，如果存在一个字符串T，重复若干次T能够得到S，那么，S叫做周期串，T叫做S的一个周期。
- 如：字符串abababab是周期串，abab、ab都是它的周期，其中，ab是它的最小周期。
- 设计一个算法，计算S的最小周期。如果S不存在周期，返回空串。

使用next，线性时间解决问题

- 计算S的next数组；
  - 记k=next[len]，p=len-k；
  - 若len%p==0，则p为最小周期长度，前p个字符就是最小周期。
- 说明：
  - 使用的是经典KMP的next算法，非变种KMP的next算法；
  - 要“多”计算到len，即next[len]。
- 思考：如何证明？
  - 考察字符串S的k前缀first和k后缀tail：
  - 1、first和tail的前p个字符
  - 2、first和tail的前2*p个字符
  - 3、first和tail的前3*p个字符
  - ……

求字符串周期

序号	0	1	2	3	4	5	6	7	8	9	10	11	12
字符串	a	b	c	a	b	c	a	b	c	a	b	c	\0
next	-1	0	0	0	1	2	3	4	5	6	7	8	9



Code

```
int MinPeriod(char* p)
{
    int nLen = (int)strlen(p);
    if(nLen == 0)
        return -1;
    int* next = new int[nLen];  //仿照KMP求"伪next"
    next[0] = -1;   //哨兵：串首标志
    int k = -1;
    int j = 0;
    while (j < nLen - 1)
    {
        if((k == -1) || (p[j+1] == p[k]))
        {
            ++k;
            ++j;
            next[j] = k;
        }
        else
        {
            k = next[k];
        }
    }
    next[0] = 0;    //恢复成逻辑上的0

    int nLast = next[nLen-1];
    delete[] next;
    if(nLast == 0)
        return -1;
    if(nLen % (nLen-nLast) == 0)
        return nLen-nLast;
    return -1;
}
```

思考题：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。
- Manacher算法

重建Manacher

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

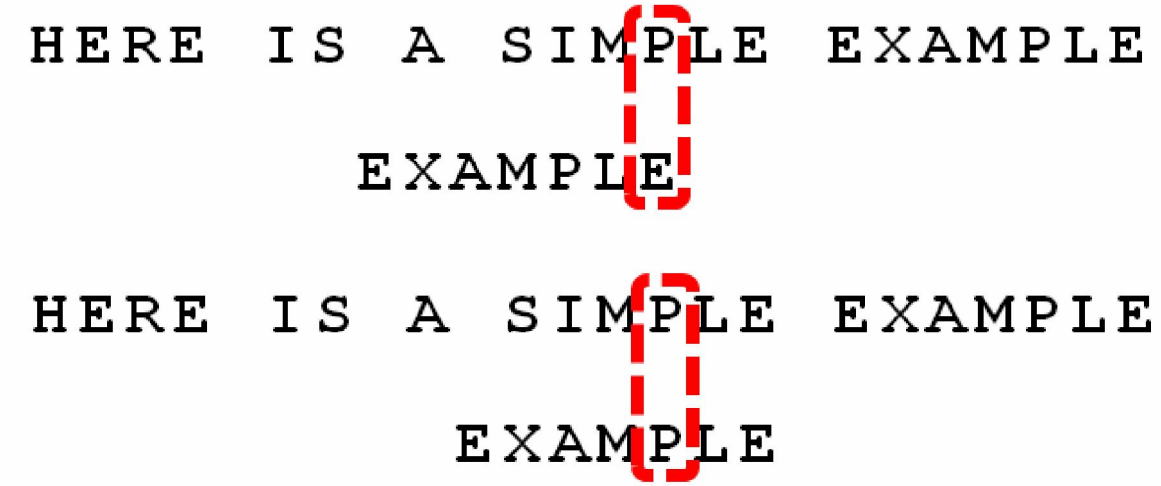
- 我们的任务：假定已经得到了前i个值，考察i+1如何计算
  - 即：在P[0...i-1]已知的前提下，计算P[i]的值。换句话说，算法的核心，是在P[0...i-1]已知的前提下，能否给P[i]的计算提供一点有用的信息呢？
- 1、通过简单的遍历，得到i个三元组{k,P[k],k+P[k]}，0≤k≤i-1
  - trick：以k为中心的字符形成的最大回文子串的最右位置是k+P[k]-1
- 2、以k+P[k]为关键字，挑选出这i个三元组中，k+P[k]最大的那个三元组，不妨记做(id, P[id]，P[id]+id)。进一步，为了简化，记mx=P[id]+id，因此，得到三元组为(id,P[id],mx)，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。

思考：BM算法

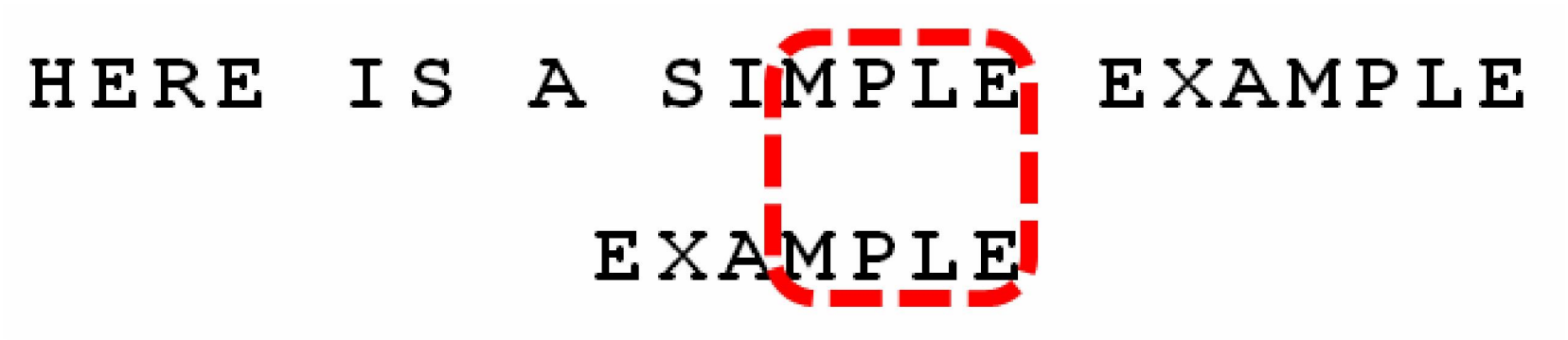
- Boyer-Moore算法是1977年，德克萨斯大学的Robert S. Boyer教授和J Strother Moore教授发明的字符串匹配算法，拥有在最坏情况下O(N)的时间复杂度，并且，在实践中，比KMP算法的实际效能高。
- BM算法不仅效率高，而且构思巧妙，容易理解。
- 坏字符 - 好后缀

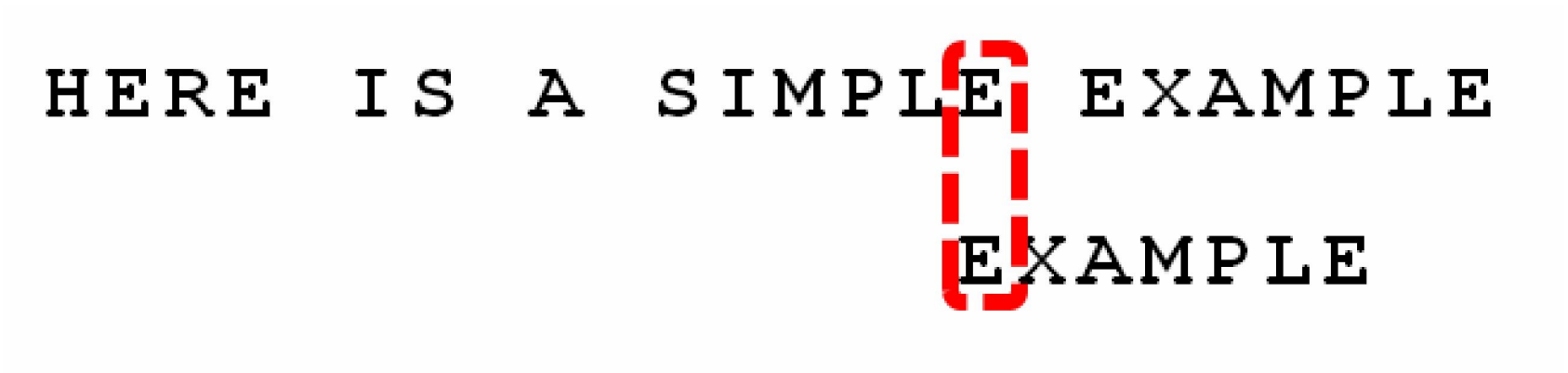
附：坏字符引起的模式滑动

- 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。



附：考虑好后缀


HERE IS A SIMPLE EXAMPLE
EXAMPLE


HERE IS A SIMPLE EXAMPLE
EXAMPLE

面试题

- 用二进制来编码字符串“uarejulyapp”，需要能够根据编码，解码回原来的字符串，最少需要_____位的二进制字符串？

  - 仅修改了字符串本身。

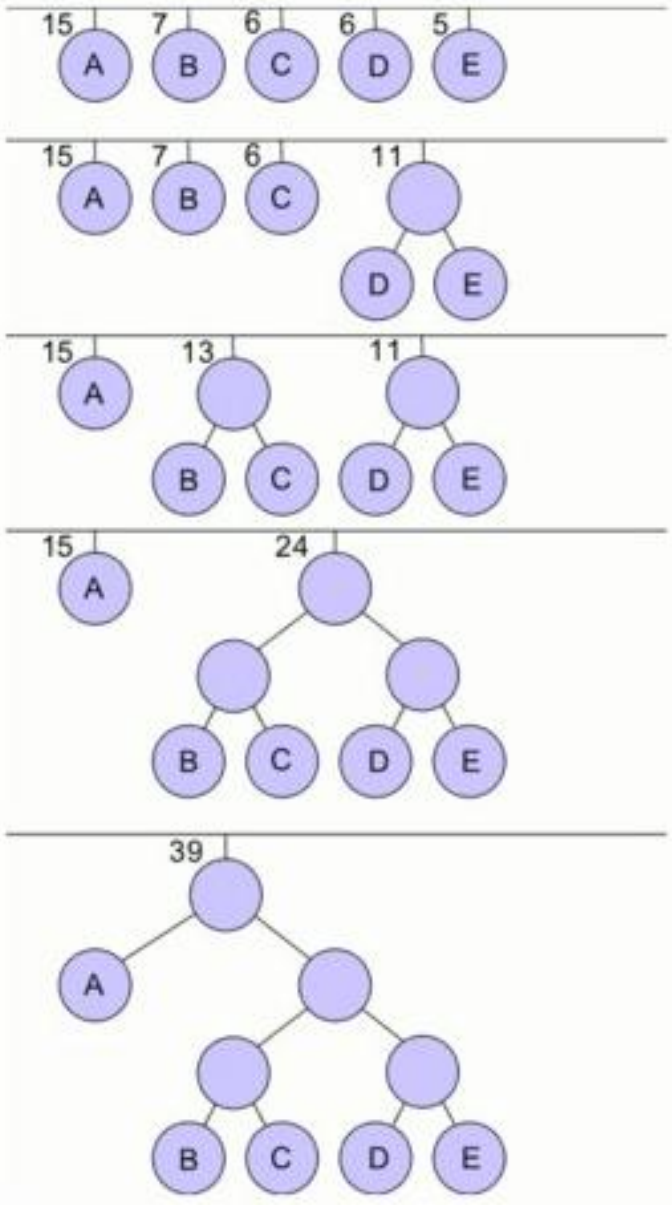
二叉树的结点

- 令有2个孩子、1个孩子和0个孩子的结点个数分别为n2、n1、n0
- 所有结点的出度为2*n2+1*n1+0*n0
- 除了根结点，其他所有结点的入度都是1，从而所有结点的入度为(n2+n1+n0)-1；
- 总入度等于总出度，2*n2+1*n1+0*n0=n2+n1+n0-1
- 化简得n0-n2=1
- 二叉树叶子节点数目比两个孩子的结点数目多1。

Huffman编码

- Huffman编码是一种无损压缩编码方案。
- 思想：根据源字符出现的(估算)概率对字符编码，概率高的字符使用较短的编码，概率低的使用较长的编码，从而使得编码后的字符串长度期望最小。
- Huffman编码是一种贪心算法：每次总选择两个最小概率的字符结点合并。
  - 称字符出现的次数为频数，则概率约等于频数除以字符总长；因此，概率可以用频数代替。

算法演示



Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int N = 256;
    char str[] = "when I was young I'd listen to the radio\
                 waiting for my favorite songs\
                 when they played I'd sing along,\
                 it make me smile.\
                 those were such happy times and not so long ago\
                 how I wondered where they'd gone.\
                 but they're back again just like a long lost friend\
                 all the songs I love so well.\
                 every shalala every wo'wo\
                 still shines.\
                 every shing-a-ling-a-ling \
                 that they're starting\
                 to sing so fine";
    int pWeight[N] = {0};
    CalcFrequency(str, pWeight);
    pWeight['\t'] = 0;
    vector<int> pChar;
    CalcExistChar(pWeight, N, pChar);
    int N2 = (int)pChar.size();
    vector<vector<char> > code(N2);
    HuffmanCoding(pWeight, N2, code);
    Print(code, pChar);
    return 0;
}
```

Code

```
void CalcFrequency(const char* str, int* pWeight)
{
    while(*str)
    {
        pWeight[*str]++;
        str++;
    }
}

void CalcExistChar(int* pWeight, int N, vector<int>& pChar)
{
    int j = 0;
    for(int i = 0; i < N; i++)
    {
        if(pWeight[i] != 0)
        {
            pChar.push_back(i);
            if(j != i)
            {
                pWeight[j] = pWeight[i];
            }
            j++;
        }
    }
}
```

Main Code

```
void HuffmanCoding(int *pWeight, int N, vector<vector<char> >& code)
{
    if (N <= 0)
        return;
    int m = 2 * N - 1;  //N个结点的Huffman树需要2N-1个结点
    HuffmanNode* pHuffmanTree = new HuffmanNode[m];
    int s1, s2;

    int i;
    //建立叶子结点
    for (i = 0; i < N; i++)
        pHuffmanTree[i].nWeight = pWeight[i];

    //每次选择权值最小的两个结点，建树
    for(i = N; i < m; i++)
    {
        SelectNode(pHuffmanTree, i, s1, s2);
        pHuffmanTree[s1].nParent = pHuffmanTree[s2].nParent = i;
        pHuffmanTree[i].nLeft = s1;
        pHuffmanTree[i].nRight = s2;
        pHuffmanTree[i].nWeight = pHuffmanTree[s1].nWeight + pHuffmanTree[s2].nWeight;
    }

    //根据建好的Huffman树从叶子到根计算每个叶结点的编码
    int node, nParent;
    for(i = 0; i < N; i++)
    {
        vector<char>& cur = code[i];
        node = i;
        nParent = pHuffmanTree[node].nParent;
        while(nParent != 0)
        {
            if(pHuffmanTree[nParent].nLeft == node)
                cur.push_back('0');
            else
                cur.push_back('1');

            node = nParent;
            nParent = pHuffmanTree[node].nParent;
        }
        reverse(cur.begin(), cur.end());
    }
}
```

Aux Code

```
void SelectNode(const HuffmanNode* pHuffmanTree, int n, int& s1, int &s2)
{
    s1 = -1;    //无效值
    s2 = -1;
    int nMin1 = -1; //无效值
    int nMin2 = -1;
    for(int i = 0; i < n; i++)
    {
        if((pHuffmanTree[i].nParent == 0) && (pHuffmanTree[i].nWeight > 0))
        {
            if((s1 < 0) || (nMin1 > pHuffmanTree[i].nWeight))
            {
                s2 = s1;
                nMin2 = nMin1;
                s1 = i;
                nMin1 = pHuffmanTree[s1].nWeight;
            }
            else if((s2 < 0) || (nMin2 > pHuffmanTree[i].nWeight))
            {
                s2 = i;
                nMin2 = pHuffmanTree[s2].nWeight;
            }
        }
    }
}
```

Aux Code

```
void PrintCode(char c, vector<char>& code)
{
    cout << (int)c << "    " << c << ":    ";
    for(vector<char>::iterator it = code.begin(); it != code.end(); it++)
    {
        cout << *it;
    }
    cout << '\n';
}

void Print(vector<vector<char> >& code, vector<int>& pChar)
{
    int size = (int)code.size();
    for(int i = 0; i < size; i++)
    {
        PrintCode(pChar[i], code[i]);
    }
}
```

实验结果

```
when I was young I'd listen to the radio
waiting for my favorite songs
when they played I'd sing along,
it make me smile.
those were such happy times and not so long ago
how I wondered where they'd gone.
but they're back again just like a long lost friend
all the songs I love so well.
every shalala every wo'wo
still shines.
every shing-a-ling-a-ling
that they're starting
to sing so fine
```

32	:	110	106	j:	01010101
39	':	111000	107	k:	0101011
44	,:	01010100	108	l:	0010
45	-:	1110011	109	m:	001111
46	.:	1111000	110	n:	1001
73	I:	001110	111	o:	1010
97	a:	0110	112	p:	1110010
98	b:	11110110	114	r:	10111
99	c:	11110111	115	s:	0111
100	d:	00110	116	t:	1000
101	e:	000	117	u:	1111010
102	f:	1111001	118	v:	010100
103	g:	11101	119	w:	01011
104	h:	11111	121	y:	10110
105	i:	0100			

Huffman编码总结：前缀编码

- Huffman编码是不等长编码
  - 字符的编码长度不完全相同。
- 不等长编码如果需要译码，必须满足“前缀编码”的条件：任何一个字符的编码都不是另外一个字符编码的前缀。
- 字符串：ABBC
- 使用编码方案：
  - A:0　　B:1　　C:00
- 则，ABBC的 编码为：01100
- 01100的译码可以是ABBC，也可以是ABBAA。

Huffman实现带来的思考

- Huffman编码是如何解决前缀编码问题的？
- 实际算法往往是由多个“小算法”堆砌而成的。
  - 空格压缩问题
  - 取数组最大/小的两个数
- 代码实现中并非直接使用指针形成的二叉树结点。而是事先开辟足够大的缓冲空间(2n+1)，每次从缓冲区获取一个结点，使用数组代替二叉树。
  - 在堆排序、双数组Trie树结构等问题中会再次遇到。
- 最后，由于Huffman树的结点权值(频数)可能相等，因此，对某些文本，Huffman编码不唯一。
  - “左赋1，右赋0”或者“左赋0，右赋1”都可以。

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，如Trie树、后缀树(后缀数组)；
  - 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。

字符串总结

- 字符串查找：增删改查
  - KMP/BM
  - map/set：R-BTree
  - Hash
  - Trie树
- 对字符串本身的操作
  - 全排列
  - Manacher
  - 回文划分

我们在这里

- 七月算法 http://www.julyedu.com/
  - 视频/课程/社区
- 七月题库APP：Android/iOS
  - http://www.julyapp.com/
- 微博
  - @研究者July
  - @七月题库
  - @邹博_机器学习
- 微信公众号
  - julyedu

感谢大家

恳请大家批评指正！