法律声明

- 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

字符串

小象学院 ChinaHadoop.cn 邹博

主要内容

- 字符串循环移位
- LCS最长递增子序列
- 字符串全排列
- Manacher算法
- KMP模式串匹配
  - 附：BM算法
- 三字母字符串组合

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。
  - 循环左移n+k位和k位的效果相同。
  - 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)。

暴力无法满足要求

- 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);
}
```

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。

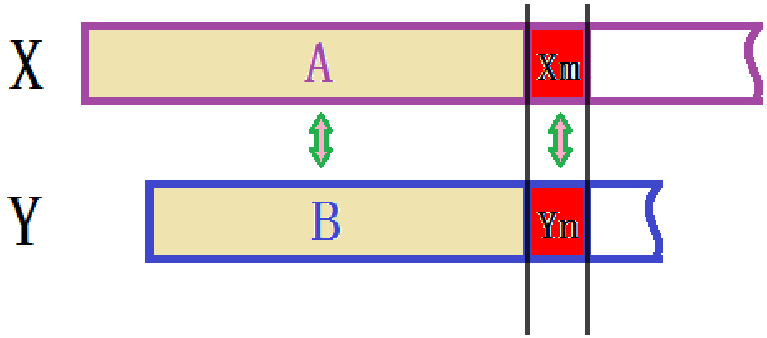
LCS的记号

- 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；
- X_i=＜x_1，…，x_i＞即X序列的前i个字符 ($1\le i\le m$)(X_i不妨读作“字符串X的i前缀”)
- Y_j=＜y_1，…，y_j＞即Y序列的前j个字符 ($1\le j\le n$) (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=＜z_1，…，z_k＞。
  - 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：$Z\in$ 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$LCS(X_m,Y_n)=LCS(X_{m-1},Y_{n-1})+x_m$

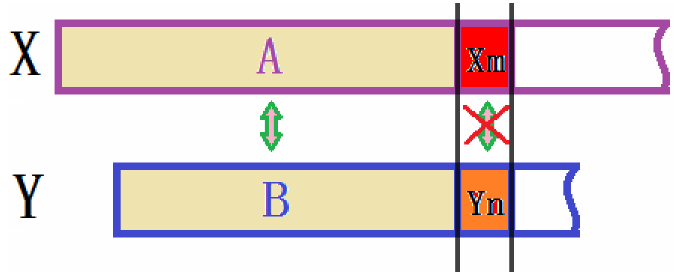


- 记$LCS(X_m,Y_n)=W+x_m$，则W是X_{m-1}的子序列；同理，W是Y_{n-1}的子序列；因此，W是X_{m-1}和Y_{n-1}的公共子序列。
  - 反证：若W不是X_{m-1}和Y_{n-1}的最长公共子序列，不妨记$LCS(X_{m-1},Y_{n-1})=W'$，且$|W'|>|W|$；那么，将W换成W'，得到更长的$LCS(X_m,Y_n)=W'x_m$，与题设矛盾。

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

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

- 显然，属于**动态规划**问题。

小象学院 ChinaHadoop.cn

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。

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

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

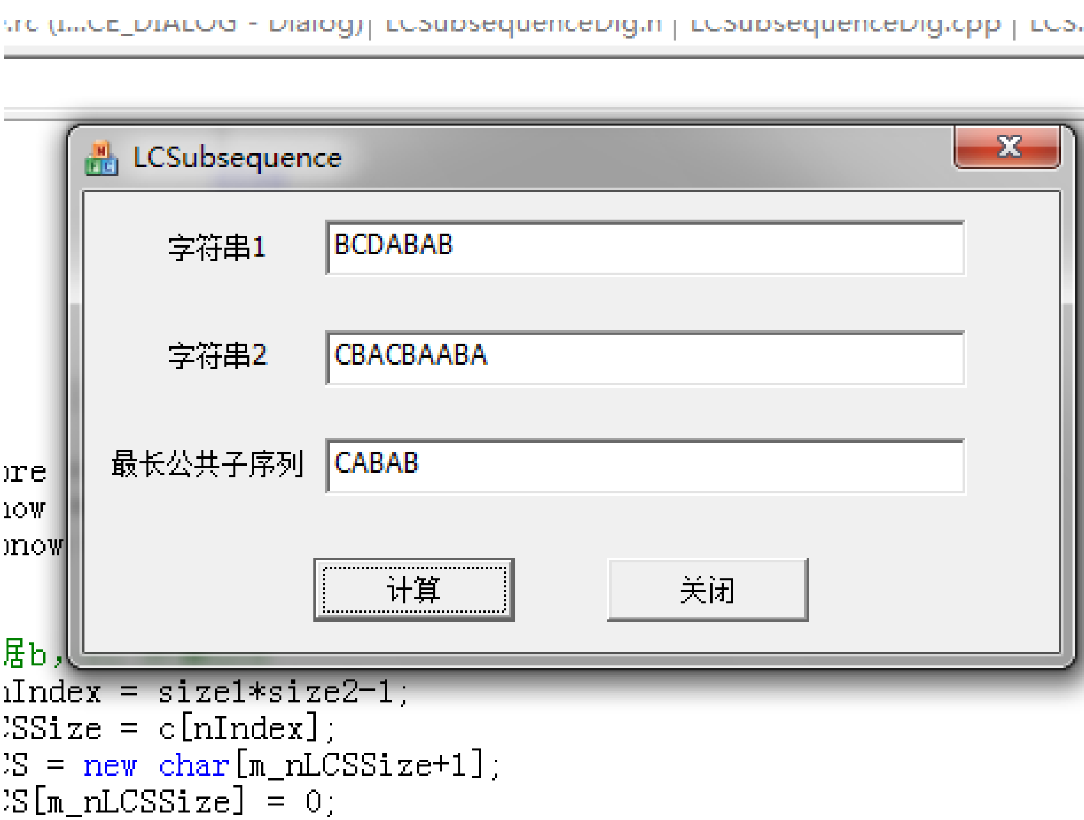
```
int _tmain(int argc, _TCHAR* argv[])
{
    const char* str1 = "TCGGATCGACTT";
    const char* str2 = "AGCCTACGTA";
    string str;
    LCS(str1, str2, str);
    cout << str.c_str() << endl;
    return 0;
}
```

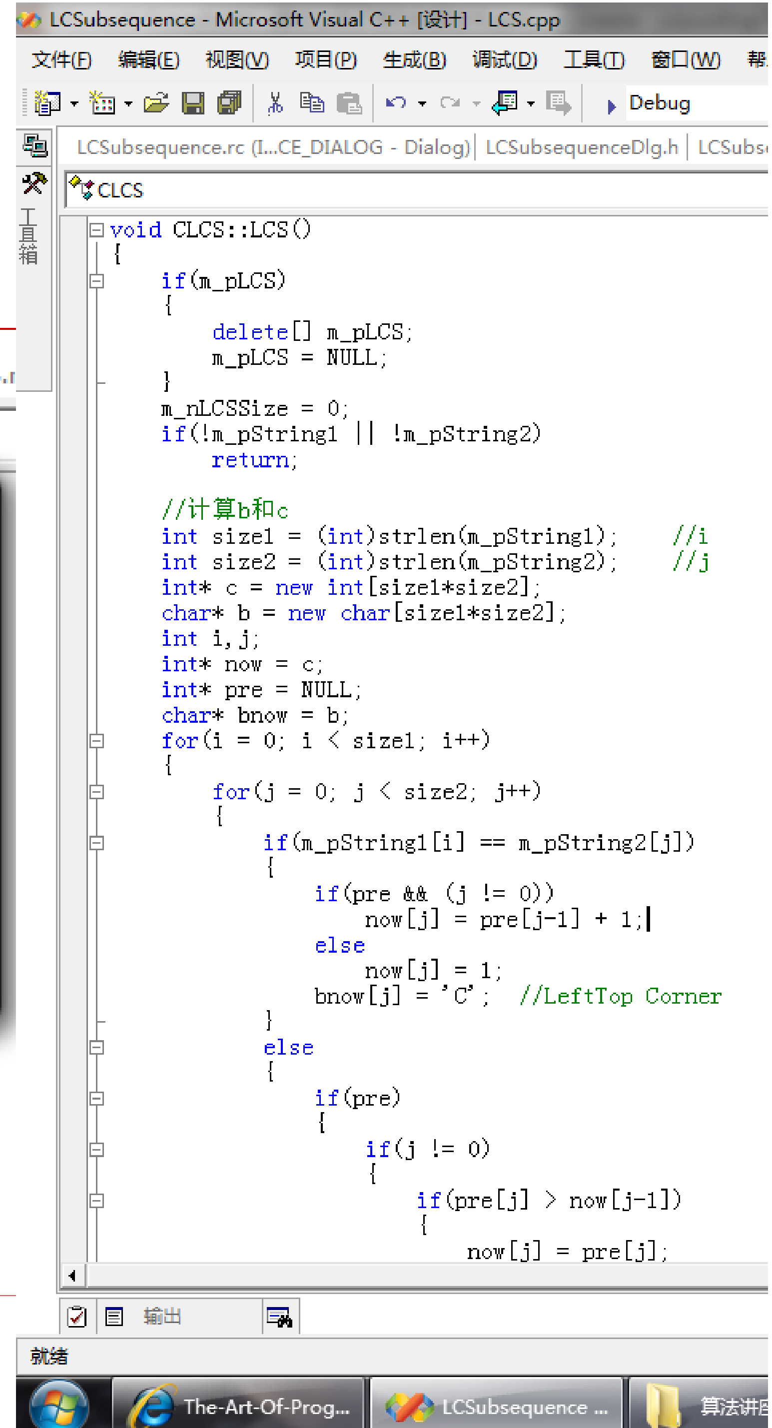
```
void LCS(const char* str1, const char* str2, string& str)
{
    int size1 = (int)strlen(str1);
    int size2 = (int)strlen(str2);
    const char* s1 = str1-1;    //从1开始数，方便后面的代码编写
    const char* s2 = str2-1;
    vector<vector<int> > chess(size1+1, vector<int>(size2+1));
    int i,j;
    for(i =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

思考

- 若只计算LCS的长度，可否使用滚动数组降低空间复杂度？

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)

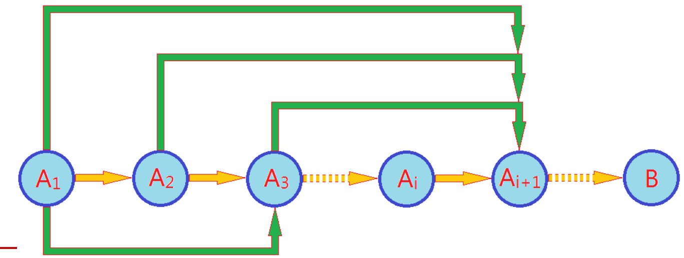
LCS的应用：最长递增子序列LIS

- Longest Increasing Subsequence，LCS：
  - 找出给定数组最长且单调递增的子序列。
- 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$L_0L_1......L_{i-1}$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…… L_{i-1}后面，是否允许呢？
  - 如果$a_i \ge a_j$，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：$b[i]=\{\max(b[j])+1, 0 \le j < i$且$a_j \le a_i\}$
  - 计算b[i]：遍历在i之前的所有位置j，找出满足条件$a_j \le a_i$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

```cpp
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;
}
```

```cpp
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());
}
```

```cpp
#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;
}
```

```cpp
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。

小象学院 ChinaHadoop.cn

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

```cpp
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(前)与第j个字符(后)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;
}
```

进一步思考

- 下排列算法能够天然解决重复字符的问题。
  - 不妨还是考察926520的下一个字符串
- STL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。
- 思考：
  - 如何计算N个无重复元素的某个排列是第几个排列？
  - “CDAEFB”是从“ABCDEF”到“FEDCBA”的第几个排列？
  - 提示：Cantor数组

最长回文子串

- 给定字符串str，若子串s是回文串，称s为str的回文子串。设计算法，计算str的最长回文子串。
  - 枚举所有子串，显然是一种解法。
  - 是否可以有更快的算法呢?

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？）

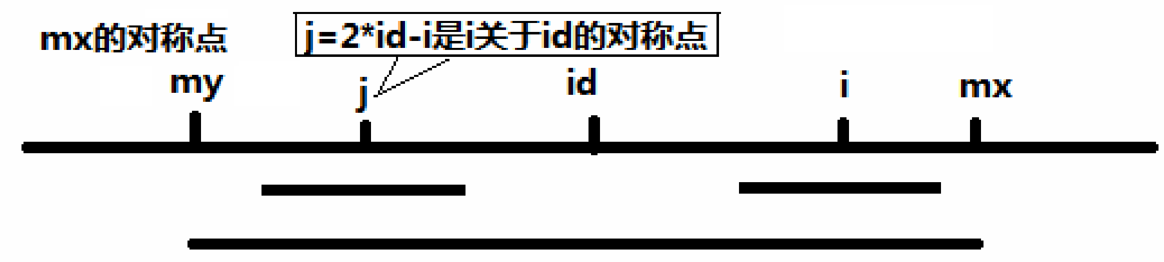
分析算法核心

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。

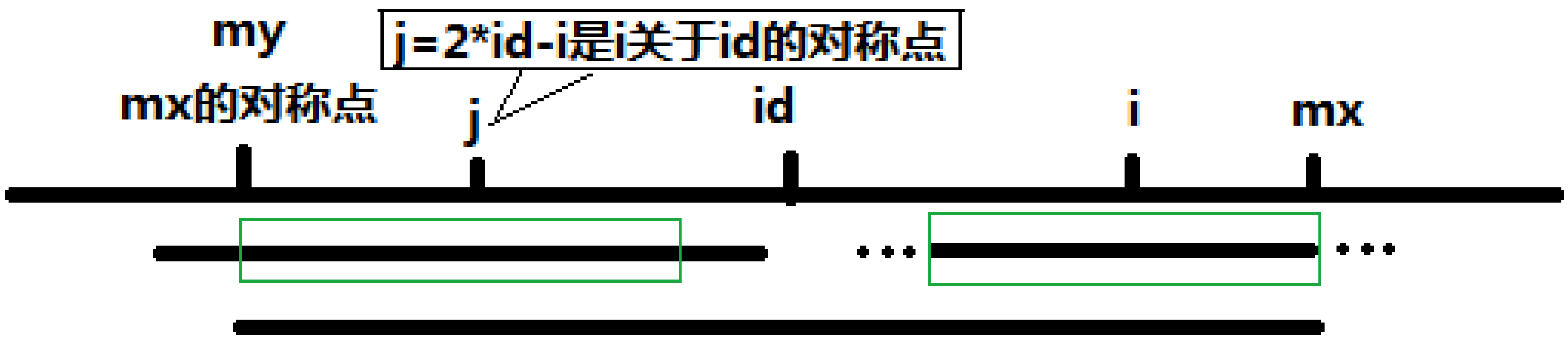
Manacher递推关系

- 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] ≥P[j]
  - 基本Manacher算法，红色的等号都是≥

Manacher改进版

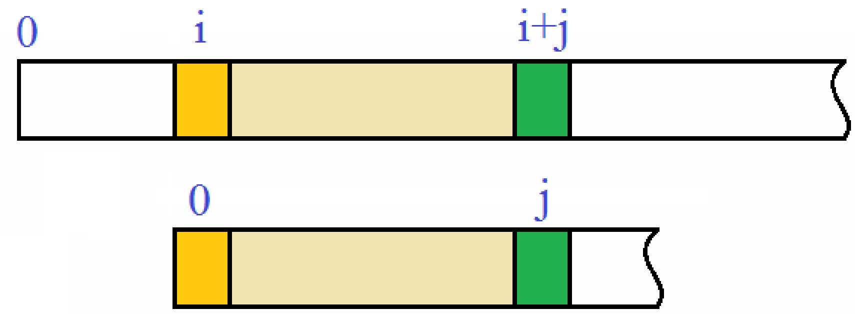
```
void Manacher(char* s, int* P)
{
    int size = strlen(s);
    P[0] = 1;
    int id = 0;
    int mx = 1;
    for(int i = 1; i < size; i++)
    {
        if(mx > i)
        {
            if(P[2*id-i] != mx-i)
            {
                P[i] = min(P[2*id-i], mx-i);
            }
            else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

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;
}
```

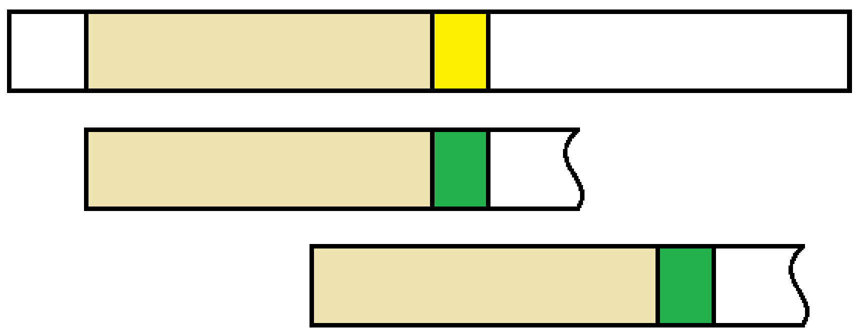
分析BF与KMP的区别

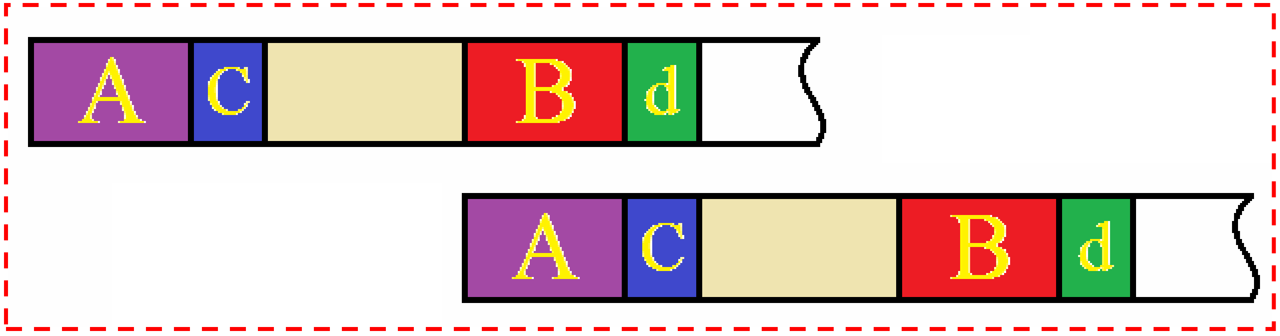
- 假设当前文本串text匹配到i位置，模式串pattern串匹配到j位置。
- BF算法中，如果当前字符匹配成功，即text[i+j]== pattern[j]，令j++，继续匹配下一个字符；
  - 若失配，即text[i+j]≠pattern[j]，令i++，j=0，即匹配失败时，模式串pattern相对于文本串text向右移动了一位。
- KMP算法中，若当前字符匹配成功，即text[i+j]== pattern[j]，令j++，继续匹配下一个字符；
  - 若失配，即text[i+j]≠pattern[j]，令j=next[j](next[j]≤j-1)，即模式串pattern相对于文本串text向右移动至少1位(实际移动位数为：j-next[j]≥1)

描述性说法

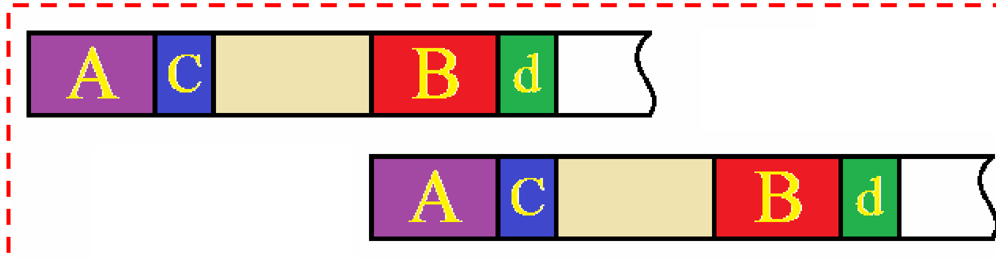
- 在暴力求解中，为什么模式串的索引会回溯?
  - 因为模式串存在重复字符
  - 思考：如果模式串的字符两两不相等呢?
    - 可以方便快速的编写线性时间的代码
  - 更弱一些的条件：如果模式串的首字符和其他字符不相等呢?

挖掘字符串比较的机制





分析后的结论



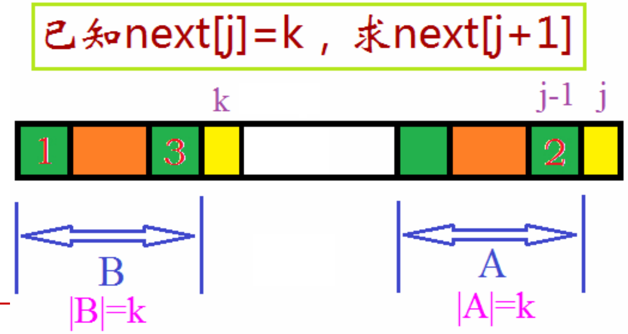
- 对于模式串的位置j，考察$Pattern_{j-1} = p_0p_1 \ldots p_{j-2}p_{j-1}$，查找字符串$Pattern_{j-1}$的最大相等k前缀和k后缀。
  - 注：计算next[j]时，考察的字符串是模式串的前j-1个字符，与pattern[j]无关。
- 即：查找满足条件的最大的k，使得
  - $p_0p_1 \ldots p_{k-2}p_{k-1} = p_{j-k}p_{j-k+1} \ldots p_{j-2}p_{j-1}$

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



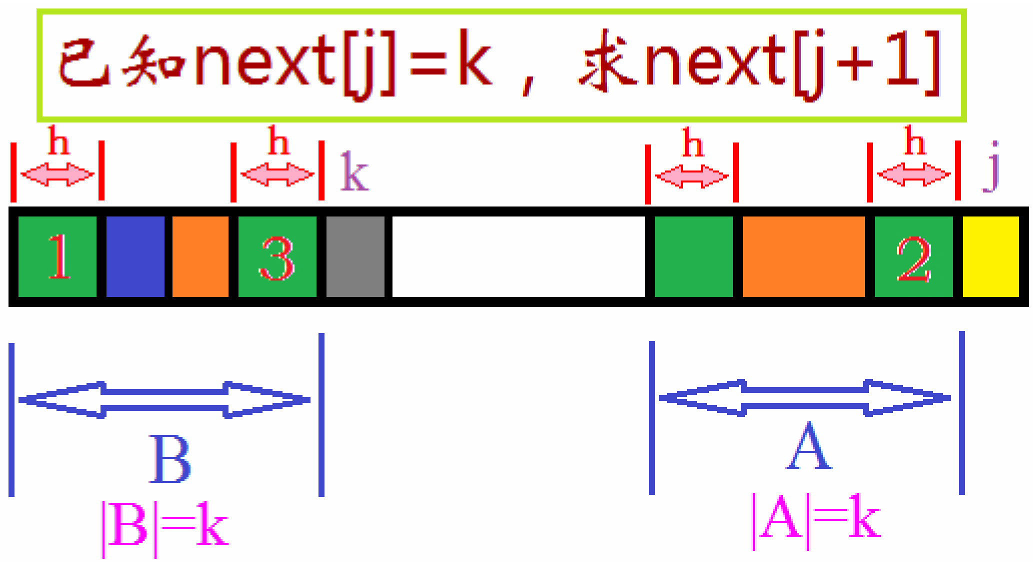
next的递推关系

- 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



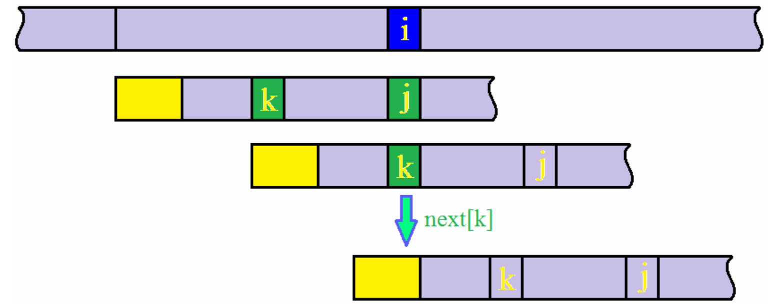
计算Next数组

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

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

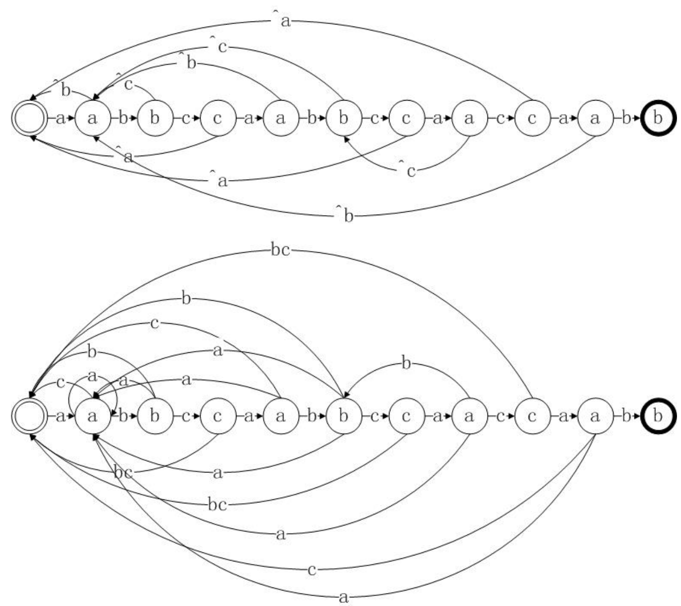
　　　aaaaa

　　　　　　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。

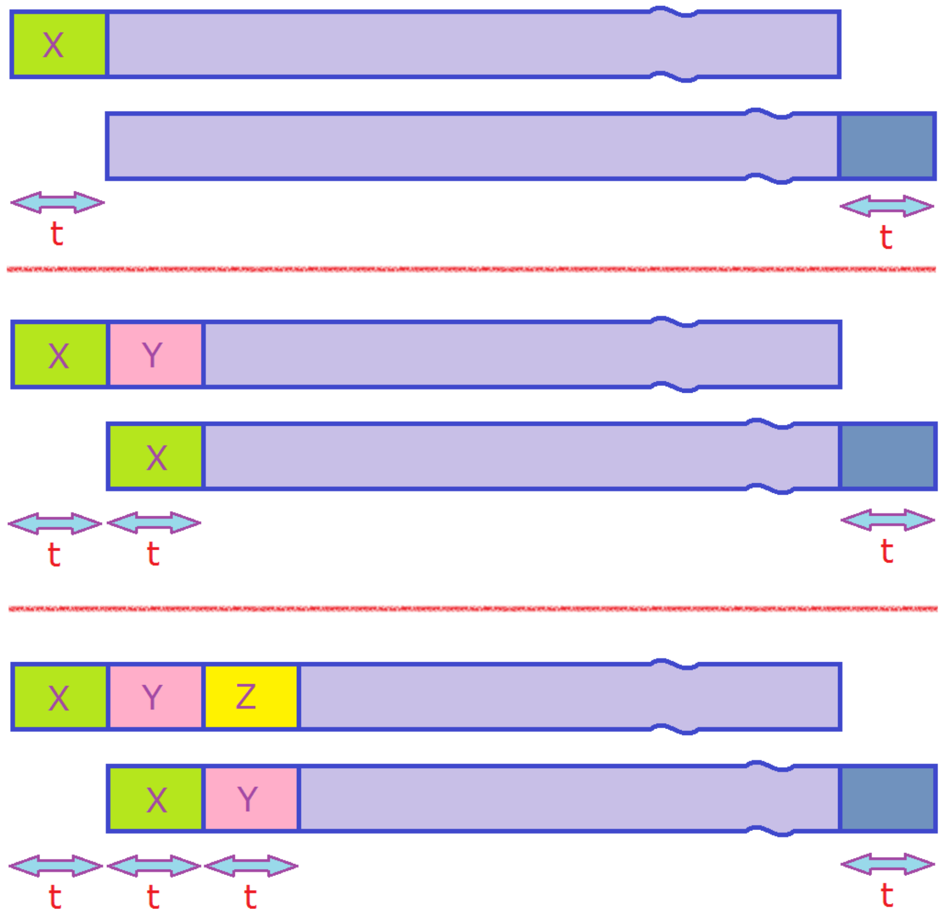
KMP应用：PowerString问题

- 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

- 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
  - ……

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;
}
```

三字母字符串组合

- 仅由三个字符A、B、C构成字符串，且字符串任意三个相邻元素不能完全相同。如“ACCCAB”不合法，“ABBCBCA”合法。求满足条件的长度为n的字符串个数。
  - 假定不考虑整数溢出
  - 要求时间和空间复杂度不高于O(N)。

问题分析

- 若当前已经有了所有长度为n-1的合法字符串，如何在末端增加一个字符，形成长度为n的字符串呢？
- 将长度为n-1字符串分成“末尾两个字符不相等”和“末尾两个字符相等”两种情况，各自数目记做dp[n-1][0], dp[n-1][1]：

dp[n][0]结尾不相等 / dp[n][1]结尾相等

- ♂ ♀ ◎
- ×× …… ×♂ ♀ / ×× …… ×♂ ♂
- dp[n][0]=2*dp[n-1][0]+ 2*dp[n-1][1]
  - ×× …… × ♂ ♀ ♂， ×× …… × ♂ ♀ ◎
  - ×× …… × ♂ ♂ ♀， ×× …… × ♂ ♂ ◎
- dp[n][1]=dp[n-1][0]
  - ×× …… × ♂ ♀ ♀
- 初始条件
  - dp[1][0]=3
  - dp[1][1]=0

状态转移方程总结与改进

- 状态转移方程：

$$\begin{cases} dp[n][0] = 2*dp[n-1][0] + 2*dp[n-1][1] \\ dp[n][1] = dp[n-1][0] \end{cases}$$

- 滚动数组：

$$\begin{cases} dp[0] = 2*dp[0] + 2*dp[1] \\ dp[1] = dp[0] \end{cases}$$

  - 使用滚动数组，将空间复杂度由O(N)降到O(1)

Code

$$\begin{cases} dp[0] = 2*dp[0] + 2*dp[1] \\ dp[1] = dp[0] \end{cases}$$

```
int CalcCount(int n)
{
    int nNonRepeat = 3;
    int nRepeat = 0;
    int t;
    for(int i = 2; i <= n; i++)
    {
        t = nNonRepeat;
        nNonRepeat = 2*(nNonRepeat + nRepeat);
        nRepeat = t;
    }
    return nRepeat + nNonRepeat;
}
```

矩阵表示与O(logN)时间复杂度

- 由状态转移方程：$\begin{cases} dp[0] = 2 * dp[0] + 2 * dp[1] \\ dp[1] = dp[0] \end{cases}$
- 得矩阵形式：$\begin{pmatrix} dp[0] & dp[1] \end{pmatrix}_{new} = \begin{pmatrix} dp[0] & dp[1] \end{pmatrix}_{old} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}$
- 从而：

$$\begin{pmatrix} dp[0] & dp[1] \end{pmatrix}_{n} = \begin{pmatrix} dp[0] & dp[1] \end{pmatrix}_{n-1} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}$$

$$= \begin{pmatrix} dp[0] & dp[1] \end{pmatrix}_{n-2} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix} = \begin{pmatrix} dp[0] & dp[1] \end{pmatrix}_{n-3} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}$$

$$= \cdots$$

$$= \begin{pmatrix} dp[0] & dp[1] \end{pmatrix}_{1} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}^{n-1} = \begin{pmatrix} 3 & 0 \end{pmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 2 & 0 \end{bmatrix}^{n-1}$$

Code2

```
typedef struct tagSMatrix22
{
    //第一列
    int a;
    int b;
    //第二列
    int c;
    int d;

    tagSMatrix22(int _a, int _b, int _c, int _d)
        :a(_a), b(_b), c(_c), d(_d) {}

    void Set(int _a, int _b, int _c, int _d)
    {
        a = _a;
        b = _b;
        c = _c;
        d = _d;
    }
}SMatrix22;
```

```
void MatrixMulti(SMatrix22& m, SMatrix22& n)//m *= n
{
    int a = m.a * n.a + m.c * n.b;
    int b = m.b * n.a + m.d * n.b;
    int c = m.a * n.c + m.c * n.d;
    int d = m.b * n.c + m.d * n.d;
    m.Set(a,b,c,d);
}

void MatrixN(SMatrix22& m, int n)   //矩阵的n次方
{
    if(n == 0)
    {
        m.Set(1,0,0,1); //单位阵
        return;
    }
    if(n == 1)
        return;
    if(n % 2 == 0)  //偶数
    {
        MatrixN(m, n/2);
        MatrixMulti(m,m);
    }
    else            //奇数
    {
        SMatrix22 x = m;
        MatrixN(m, n/2);
        MatrixMulti(m,m);
        MatrixMulti(m,x);
    }
}

int CalcCount2(int n)
{
    int nNonRepeat = 3;
    int nRepeat = 0;
    SMatrix22 m(2,2,1,0);
    MatrixN(m, n-1);
    return 3*(m.a + m.c);   //(3 0) * m
}
```

总结与思考

- 字符串和树相结合，往往会产生查找思路上的变革，如Trie树：
  - 给定约一百万行的某文本文件，每行一个词，统计最频繁出现的前10个词
- 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。
- 问题规模的变化会导致算法的不同。
  - 如：两个文本如何计算相似度？

我们在这里

- http://wenda.ChinaHadoop.cn
  - 视频/课程/社区
- 微博
  - @ChinaHadoop
  - @邹博_机器学习
- 微信公众号
  - 小象学院
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！

附： BM算法

- Boyer-Moore算法是1977年Robert S. Boyer和J Strother Moore发明的字符串匹配算法，最坏情况下的时间复杂度为O(N)，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
```

坏字符规则

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

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

- 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。

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

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
```