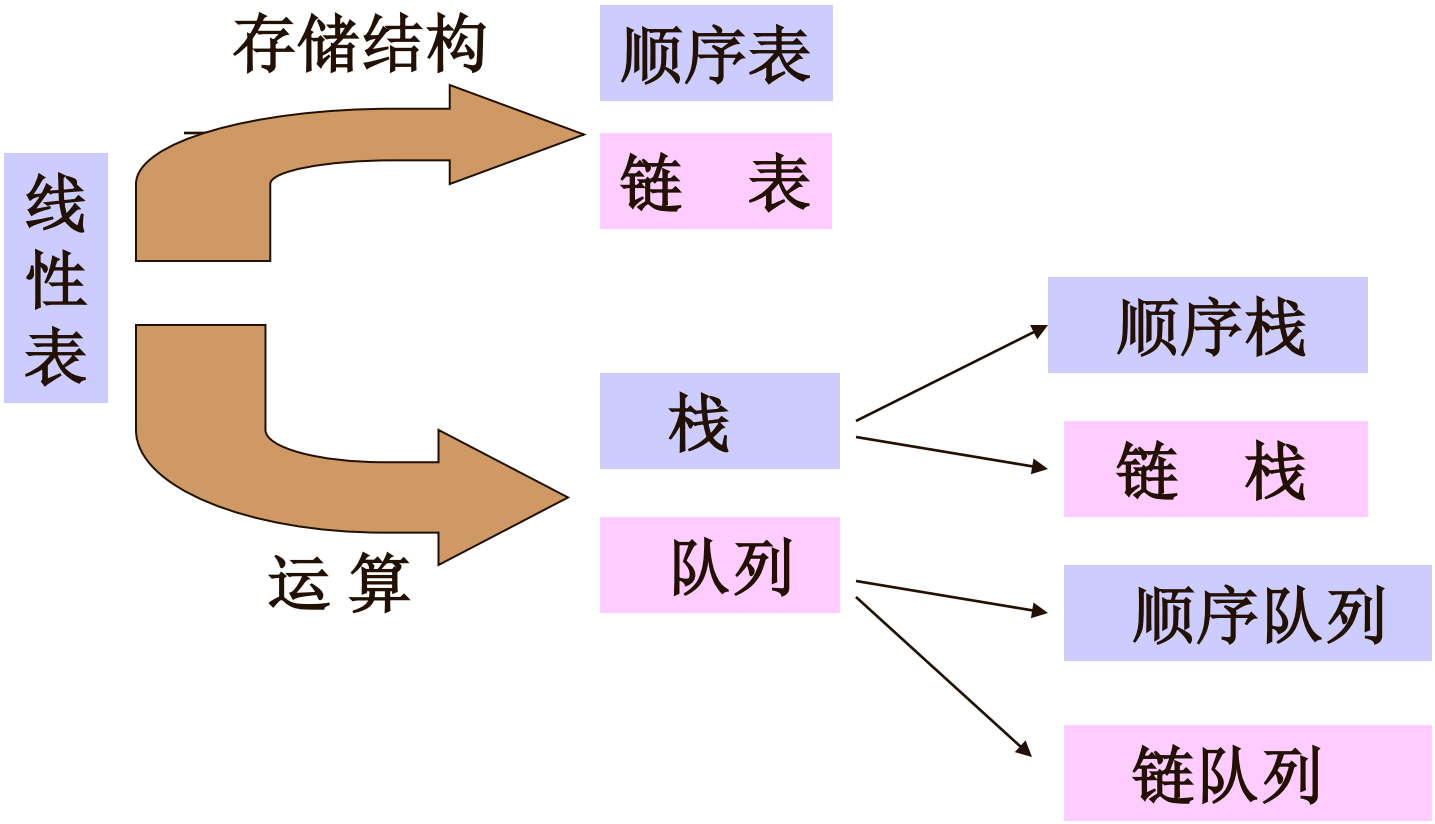
回顾



回顾

线性表、栈、队的异同点：

不同点：① 运算规则不同：

- ✓ 线性表为随机存取；
- ✓ 而栈是只允许在一端进行插入和删除运算，因而是后进先出表LIFO；
- ✓ 队列是只允许在一端进行插入、另一端进行删除运算，因而是先进先出表FIFO。

② 用途不同，线性表比较通用；堆栈用于函数调用、递归和简化设计等；队列用于离散事件模拟、OS作业调度和简化设计等。

回顾

例：数组Q[n]用来表示一个循环队列，f为当前队列头元素的前一位置，r为队尾元素的位置。假定队列中元素的个数小于n，计算队列中元素个数的公式为：

(A) r - f　　(B) (n+f - r) % n
(C) n+r - f　　(D)√ (n+r - f) % n

答：　要分析4种公式哪种最合理？

当r ≥f时（A）合理；
当r＜f时（C）合理；

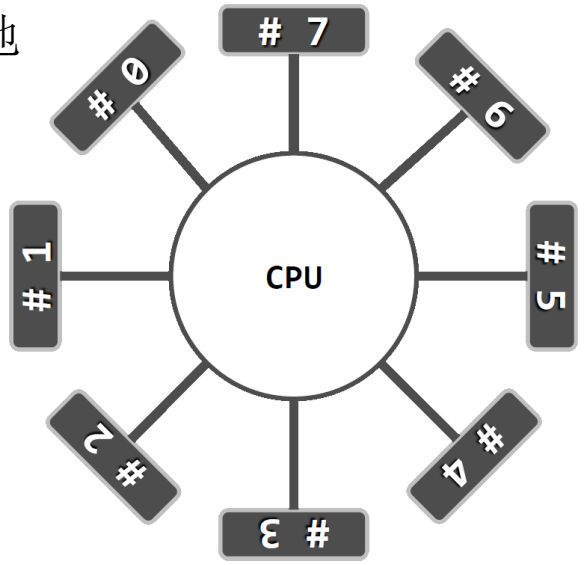
} 综合2种情况，以（D）的表达最为合理

回顾

队列应用——资源循环分配

一群客户(client)共享同一资源时，如何兼顾公平与效率。比如，多个应用程序共享CPU，实验室成员共享打印机，……

```
RoundRobin { //循环分配器
    SqQueue Q(clients);//参与资源分配的所有客户组成队列
    while(!serviceclosed()){ //在服务关闭之前，反复地
        e= Q.DeQueue();//令队首的客户出队，并
         serve(e);//接受服务，然后
         Q.EeQueue(e);//重新入队
    }
}
```



- 利用队列改进迷宫算法，找出最短的通路

数据结构与算法设计

第四章　　串

上海自来水来自海上

两相思
宋 李禺

枯眼望遥山隔水，
往来曾见几心知？
壶空怕酌一杯酒，
笔下难成和韵诗。
途路阻人离别久，
讯音无雁寄回迟。
孤灯夜守长寥寂，
夫忆妻兮父忆儿。

儿忆父兮妻忆夫，
寂寥长守夜灯孤。
迟回寄雁无音讯，
久别离人阻路途。
诗韵和成难下笔，
酒杯一酌怕空壶。
知心几见曾来往，
水隔山遥望眼枯。

第四章 串

lover	over
friend	end
believe	lie

4.1 串类型的定义

串(string)即字符串，是由零个或多个字符组成的有限序列，是数据元素为单个字符的特殊线性表。

记为：　s = ‘a_1 a_2 …….. a_n’　(n≥0)

串名　串值（用‘ ’ 括起来）

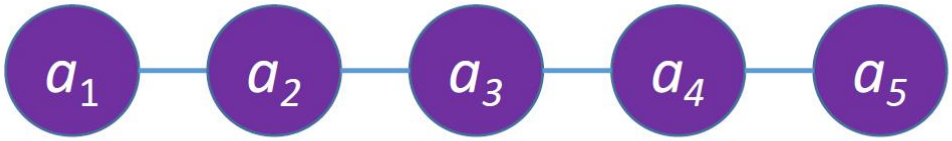
隐含结束符‘\0’，即ASCII码NULL

其中:

- S是串名字
- ‘$c_1c_2c_3\cdots c_n$’是串值
- c_i是串中字符
- n是串的长度，表示串中字符的数目

4.1 串类型的定义

串是一种特殊的线性表，数据元素之间呈线性关系



串的数据对象限定为字符集（如中文字符、英文字符、数字字符、标点字符等）

串的基本操作，如增删改查等通常以子串为操作对象

4.1 串类型的定义

串的表示：用一对单引号括起来

例：S="HelloWorld! “

T=“HUAWEI Mate 60 Pro?”

有的地方用双引号(如Java、C)
有的地方用单引号(如Python)

空串：零个字符的串称为空串记作“∅”

空白串：由一个或多个空格符组成的串

问：空串和空白串有无区别?

答：有区别。

空串(Null String)是指长度为零的串；

空白串(Blank String)，是指包含一个或多个空白字符‘ ’(空格键)的字符串。

4.1 串类型的定义

子串：串中任意个连续的字符组成的子序列

主串：包含子串的串

字符在串中的位置：字符在序列中的序号

子串在串中的位置：子串的第一个字符在主串中的位置

串的长度：串中字符的个数

一个串中任意个连续字符组成的子序列(含空串)称为该串的子串。例如，“a”、“ab”、“abc”和“abcd”等都是“abcde”的子串。

4.1 串类型的定义

例1：现有以下4个字符串：

a = ‘BEI’ b = ‘JING’ c = ‘BEIJING’ d = ‘BEI JING’

问：① 它们各自的长度？ a =3，b =4，c = 7，d=8

② a是哪个串的子串？在主串中的位置是多少？

a是c和d的子串，在c和d中的位置都是1

③ 空串是哪个串的子串？ a是不是自己的子串？

“空串是任意串的子串；任意串S都是S本身的子串，除S本身外，S的其他子串称为S的真子串。”

4.1 串类型的定义

串相等：当且仅当两个串的长度相等并且各个对应位置上的字符都相同时，这两个串才是相等的

串的操作：以“串的整体”为操作对象

①串的元素全为字符

②线性表的操作基本是单个元素；串的基本操作是“串的整体”

4.1 串类型的定义

ADT String{

Objects: $D=\{a_i \mid a_i \in CharacterSet, i=1, 2, \ldots, n, n \geq 0\}$

Relations: $R1=\{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2, \ldots, n\}$

functions: //至少有13种基本操作

基本操作子集

```
StrAssign(&T, chars)      // 串赋值，生成值为chars的串T
StrCompare(S,T)           // 串比较，若S>T，返回值大于0…
StrLength(S)              // 求串长，即返回串S中的元素个数
Concat(&T, S1, S2)        // 串连接，用T返回S1＋S2的新串
SubString(&Sub, S, pos, len)  // 求S中pos起长度为len的子串
                ……
Index(S, T, pos)          //子串定位函数（模式匹配），返回位置
Replace(&S, T,V)          // 用子串V替换子串T
```

}ADT String

C语言中已有类似串运算函数!

4.1 串类型的定义

注：用C处理字符串时，要调用标准库函数 #include<string.h>

C语言	类C
串比较：int strcmp(char *s1,char *s2);	StrCompare(S,T)
求串长：int strlen(char *s);	StrLength(S)
串连接：char strcat(char *to,char *from)	Concat(&T, S1, S2)
子串T定位：char strchr(char *s,char *c);	Index(S, T, pos)
……	

4.1 串类型的定义

例1：设 s =’I AM A STUDENT’, t =’GOOD’, q=’WORKER’。求：

StrLength(s) ＝ 14 //参见P71

StrLength(t) ＝ 4

SubString(&sub, s, 8, 7)= ‘STUDENT’

SubString(&sub, t, 2, 1)= ‘O’

Index(s, ‘A’)= 3

Index(s, t)= 0

Replace(&s, ‘STUDENT’, q)= ’I AM A WORKER’

（ s中没有t=’GOOD’！ ）

Index(S, T, pos)
// 返回子串T在pos之后的位置

Replace(&S, T,V)
// 用子串V替换子串T

4.1 串类型的定义

提问： 当s =’I AM A STUDENT’时,

INDEX（s,’A’，pos）=3，若想搜索后面那个‘A’怎么办?

答: 根据教材P71倒1行的函数说明， INDEX（s,’A’）返回的只是“第一次”出现的位置。

如果还要搜索后面的A，则pos变量要跟着变才行。

也就是说，要把得到的“第一次”位置再代入INDEX（ s,’A’，pos+1）函数中循环操作才行。

4.1 串类型的定义

例2：设 s =’I AM A STUDENT’, t =’GOOD’,求：
Concat(SubString(s,6,2), Concat(t,SubString(s,7,8))) ＝?

解：因为

SubString(s,6,2)＝‘A’;

SubString(s,7,8)＝‘ STUDENT’

Concat(,t,SubString(s,7,8))＝’ GOOD STUDENT’

所以:

Concat(,SubString(s,6,2), Concat(t,SubString(s,7,8)))

＝‘A GOOD STUDENT’

4.2 串的表示和实现

- 1. 定长顺序存储表示
- 2. 堆分配存储表示
- 3. 串的块链存储表示
- 4. 串的基本操作

4.2 串的表示和实现

串是一种特殊的线性表，在非紧缩格式中，它的每个结点仅由一个字符组成，因此存储串的方法也就是存储线性表的一般方法。存储串最常用的方式是采用顺序存储，即把串的字符顺序地存储在内存一片相邻的空间，这称为顺序串。

强调：串与线性表的运算有所不同，是以“串的整体”作为操作对象，例如查找某子串，在主串某位置上插入一个子串等。

4.2 串的表示和实现

串的物理存储表示方法：

顺序存储

- 定长顺序存储表示

  ——用一组地址连续的存储单元存储串值的字符序列，属静态存储方式。

- 堆分配存储表示

  ——用一组地址连续的存储单元存储串值的字符序列,但存储空间是在程序执行过程中动态分配而得。

4.2 串的表示和实现

串的物理存储表示方法：

链式存储

- 串的块链存储表示

——链式方式存储

1. 定长顺序存储表示

- 静态分配
  - 每个串预先分配一个固定长度的存储区域。
  - 实际串长可在所分配的固定长度区域内变动
    - 以下标为0的数组分量存放串的实际长度——PASCAL
    - 在串值后加入”\0”表示结束，此时串长为隐含值

- 用定长数组描述：

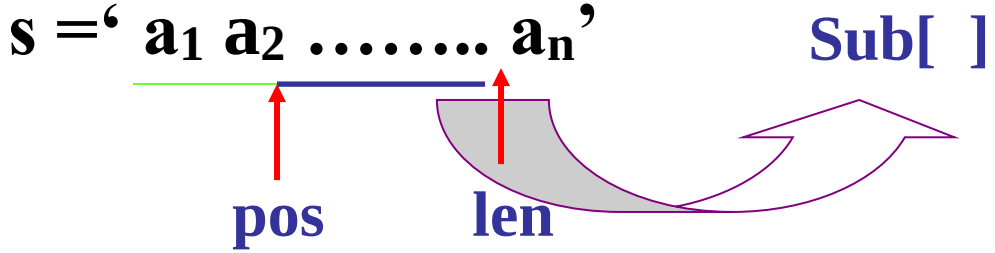
```
#define MAXSTRLEN 255 //最大串长
typedef unsigned char SString[MAXSTRLEN + 1]
   //0号单元存放串的长度
```

1. 定长顺序存储表示

例：用顺序存储方式编写求子串函数

SubString(&Sub,S,pos,len)

将串S中从第pos个字符开始、长度为len的字符序列复制到串Sub中。（注：考虑到函数的通用性，应当让串Sub的预留长度与S一样）



1. 定长顺序存储表示

例： 用顺序存储方式编写求子串函数SubString(&Sub,S,pos,len)

```
Status SubString(SString &sub, SString S, int pos, int len )
{  if(pos<1 || pos>S[0] || len<0 || len>S[0]-pos+1) return ERROR;
//若pos和len参数越界，则告警
 Sub[1……len]=S[pos……pos+len-1];
 Sub[0]=len;   return OK;
}
```

子串长度

讨论：想存放超长字符串怎么办?
改用动态分配的一维数组——堆

2. 堆分配存储表示

- 以一组地址连续的存储单元存放串值字符序列；
- 存储空间动态分配，用malloc()和free()来管理
- P75示例

思路：利用**malloc**函数合理预设串长空间。

特点：若在操作中串值改变，还可以利用**realloc**函数按新串长度增加空间（即动态数组概念）。

2. 堆分配存储表示

堆T的存储结构描述

```
typedef struct {
        char *ch;   //若非空串,按串长分配空间; 否则ch=NULL
        int length;   //串长度
}HString
```

T.ch
T.length

例1：编写建堆函数 （参见教材）

```
Status StrAssign(HString &T, char *chars)
{ //生成一个串T, T值←串常量chars
  if (T.ch)  free(T.ch);   //释放T原有空间
  for (i=0, c=chars;  c;  ++i, ++c);      //求chars的串长度i
  if ( !i ) {T.ch = NULL; T.length = 0;}
            else{
              if (!(T.ch = (char*)malloc( i *sizeof(char))))
              exit(OVERFLOW);
             T.ch[0..i-1] = chars[0..i-1];
             T.length =i;
            }
            Return OK;
} //StrAssign
```

c是指针变量，可以自增！意即每次后移一个数据单元。

直到终值为“假”停止，串尾特征是c＝ ‘\0’＝NULL＝0

此处T.ch[0]没有用来装串长，因为另有T.length 分量

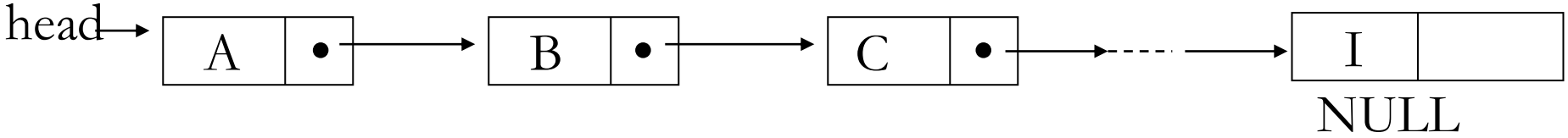
例2：用“堆”方式编写串插入函数

```
Status StrInsert ( HString &S,  int pos,  HString T )
{   //在串S的第pos个字符之前（包括尾部）插入串T
if (pos<1||pos>S.length+1) return ERROR;    //pos不合法则告警
  if(T.length){ //只要串T不空，就需要重新分配S空间，以便插入T
 if (!（S.ch=(char*)realloc(S.ch, (S.length+T.length)*sizeof(char)) ))
exit(OVERFLOW);  //若开不了新空间，则退出
      for ( i=S.length-1;  i>=pos-1; --i ) S.ch[i+T.length] = S.ch[i];
   // 为插入T而腾出pos之后的位置，即从S的pos位置起全部字符均后移
   S.ch[pos-1…pos+T.length-2] = T.ch[0…T.length-1]; //插入T，略/0
     S.length + = T.length;          //刷新S串长度
} return OK;
} //StrInsert
```

3. 串的块链存储表示

- 串的链式存储方式:用链表存储串值，易插入和删除。
- 结点大小：一个或多个字符
  - 存储密度=串值所占的存储位/实际分配的存储位

法1：链表结点的数据分量长度取1（个字符）



法2：链表结点（数据域）大小取n(例如n=4)

讨论：法1存储密度为 1/2 ；法2存储密度为 9/15=3/5 。

显然，若数据元素很多，用法2存储更优—称为块链结构

3. 串的块链存储表示

对每个结点的描述

```
#define  CHUNKSIZE  80     //每块大小，可由用户定义
typedef  struct  Chunk {          //首先定义结点类型
        char    ch [ CHUNKSIZE ];  //每个结点中的数据域
        struct  Chunk * next ;        //每个结点中的指针域
}Chunk;
```

```
typedef  struct {                  //其次定义用链式存储的串类型
        Chunk  *head;              //头指针
        Chunk  *tail;              //尾指针
        int  curLen;               //结点个数
  } Lstring;     //串类型只用一次，前面可以不加Lstring
```

对块链表的整体描述

4. 串的基本操作

- 串插入　Status StrInsert(HString &S,int pos,HString T)
- 串赋值　Status StrAssign(HString &S,char *chars)
- 求子串　Status SubString(HString &Sub,HString S,int pos,int len)
- 求串长　int StrLength(HString S)
- 串比较　int StrCompare(HString S,HString T)
- 串联接　Status Concat(HString &S,HString S1,HString S2)
- 串清空　Status ClearString(HString &S)
- 串定位
- 删除
- 置换

4. 串的基本操作

```
int StrLength(HString S)
//求串的长度
{
  return S.length;
}
```

4. 串的基本操作

```
int StrCompare(HString S,HString T)
//比较两个串，若相等返回0
{
  int i;
  for (i=0;i<S.length && i<T.length; ++i)
      if (S.ch[i] != T.ch[i]) return S.ch[i]-T.ch[i];
      return S.length-T.length;
}
```

4. 串的基本操作

```
Status Concat(HString &S,HString S1,HString S2)
//用S返回由S1和S2联接而成的新串
{ int j;
  if (!(S.ch = (char*)malloc((S1.length+S2.length)*sizeof(char))))
       exit(OVERFLOW);
   for (j=0;j<=S1.length-1;j++)
        { S.ch[j]=S1.ch[j]; }
   S.length=S1.length+S2.length;
   for (j=0;j<=S2.length-1;j++)
        { S.ch[S1.length+j]=S2.ch[j]; }
   return OK;
}
```

4. 串的基本操作

```
Status SubString(HString &Sub,HString S,int pos,int len)
//用Sub返回串S的第pos个字符开始长度为len的子串
{
  if (pos<1 || pos>S.length || len<0 ||len>S.length-
  pos+1)
      return ERROR;
  if (!len) { Sub.ch=NULL; Sub.length=0;}
  else {
      Sub.ch=(char *)malloc(len*sizeof(char));
      for (int j=0;j<=len-1;j++){
            Sub.ch[j]=S.ch[pos-1+j];}
      Sub.length=len;
  }
  return OK;
}
```

4. 串的基本操作

Status ClearString(HString &S)
将S清为空串

```
{
  if (S.ch) { free(S.ch); S.ch=NULL;}
  S.length=0;
  return OK;
}
```

4.3 串的模式匹配算法

- 定义　在串中寻找子串（第一个字符）在串中的位置
- 词汇　在模式匹配中，子串称为模式，串称为目标。
- 示例　目标 S ：“Beijing”
  模式 P ：“jin”

匹配结果 = 4

典型函数为Index(S,T,pos)

4.3 串的模式匹配算法

字符串模式匹配经典算法：

- BF算法 （又称古典的、经典的、朴素的、穷举的）
- KMP算法

主串匹配指针避免回溯，匹配速度快，

主串匹配指针带回溯，速度慢

1. BF算法的实现——即编写Index(S,T,pos)函数

例1： S='ababcabcacbab'，T='abcac'，pos=1，

求：串T在串S中第pos个字符之后的位置。

BF算法设计思想：

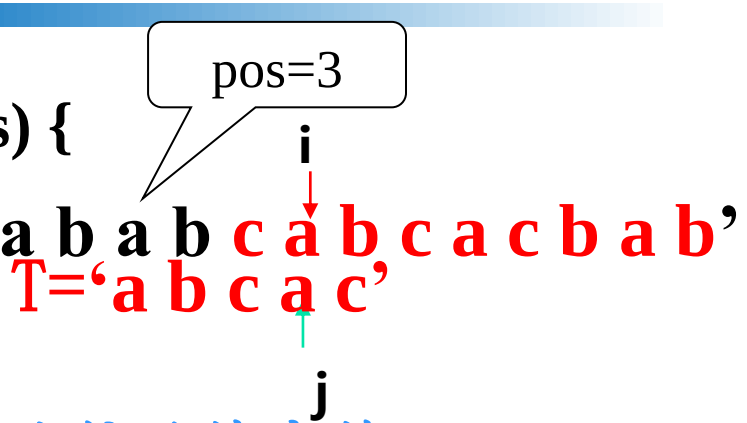
（1）将主串S的第pos个字符和模式T的第1个字符比较，

若相等，继续逐个比较后续字符；

若不等，从主串S的下一字符（pos+1）起，重新与T第一个字符比较。

（2）直到主串S的一个连续子串字符序列与模式T相等。返回值为S中与T匹配的子序列第一个字符的序号，即匹配成功。否则，匹配失败，返回值0.

1. BF算法的实现——即编写Index(S,T,pos)函数




```
int Index(SString S, SString T, int pos) {
  i=pos;      j=1;
  while ( i<=S[0] && j<=T[0] ) {
     if (S[i] = = T[j] ) {++i, ++j}   //继续比较后续字符
     else {i=i-j+2; j=1;}      //指针回溯到 下一首位，重新开始匹配
  }
  if(j>T[0])  return i-T[0];  //子串结束，说明匹配成功
  else return0;
}//Index
```

相当于子串向右滑动一个字符位置

匹配成功后指针仍要回溯！因为要返回的是被匹配的首个字符位置。

算法的时间复杂度计算

最坏情况：若n为主串长度，m为子串长度，则串的匹配算法最坏的情况下需要比较字符的总次数为

(n-m+1)*m＝O(n*m)

最恶劣情况是：主串前面n-m个位置都部分匹配到子串的最后一位，即这n-m位比较了m次，别忘了最后m位也各比较了一次，还要加上m！所以总次数为：**(n-m)*m+m ＝(n-m+1)*m**

算法的时间复杂度计算

最好的情况是：一配就中！　只比较了m次。

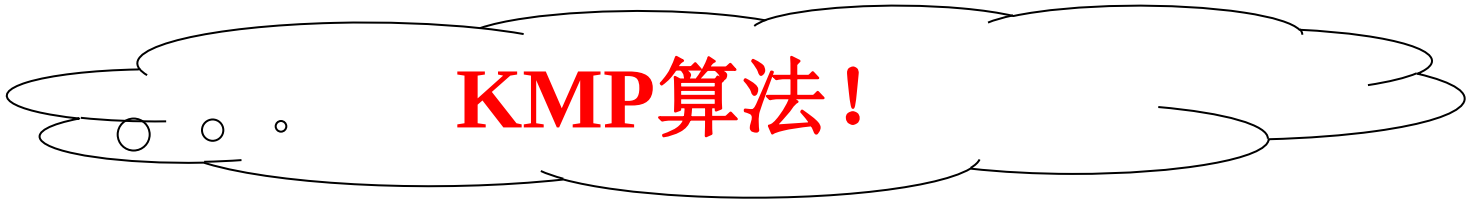
一般的情况是：**O(n+m)**

推导方法：要从最好到最坏情况统计总的比较次数，然后取平均。

算法的时间复杂度计算

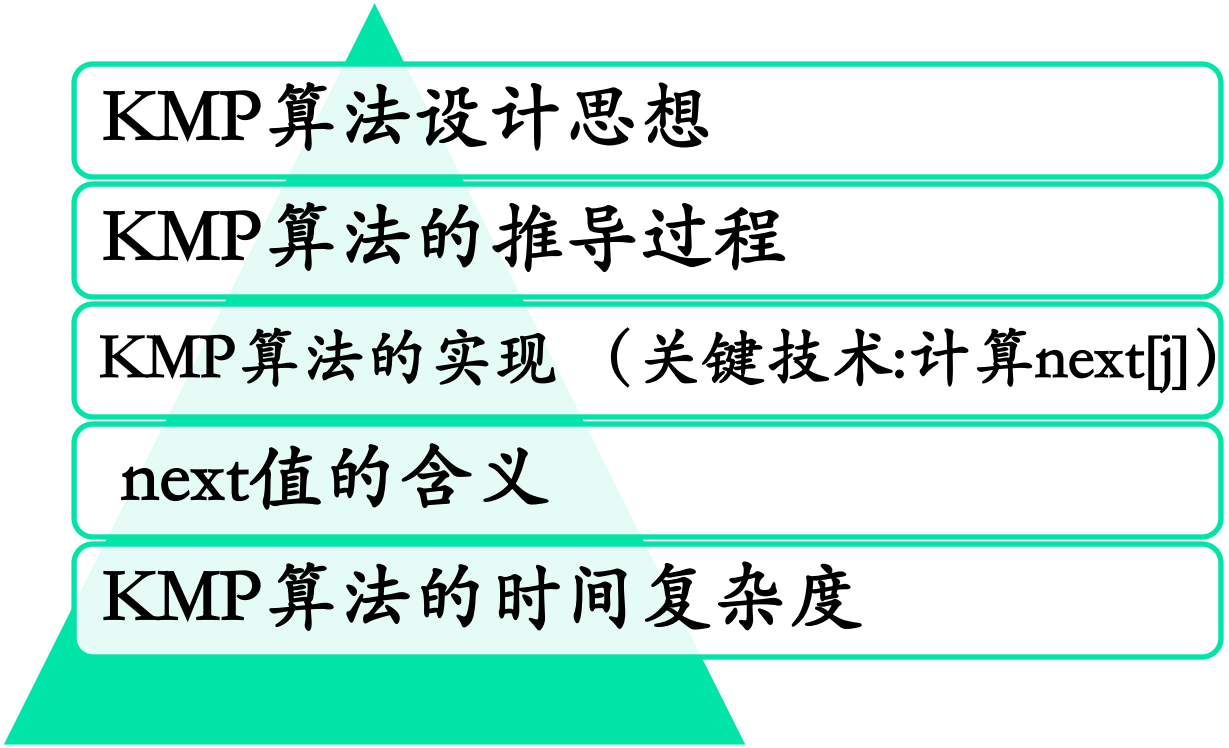
能否利用已部分匹配过的信息而加快模式串的滑动速度?

能！ 而且主串S的指针i不必回溯！ 最坏情况也能达到O(n+m)



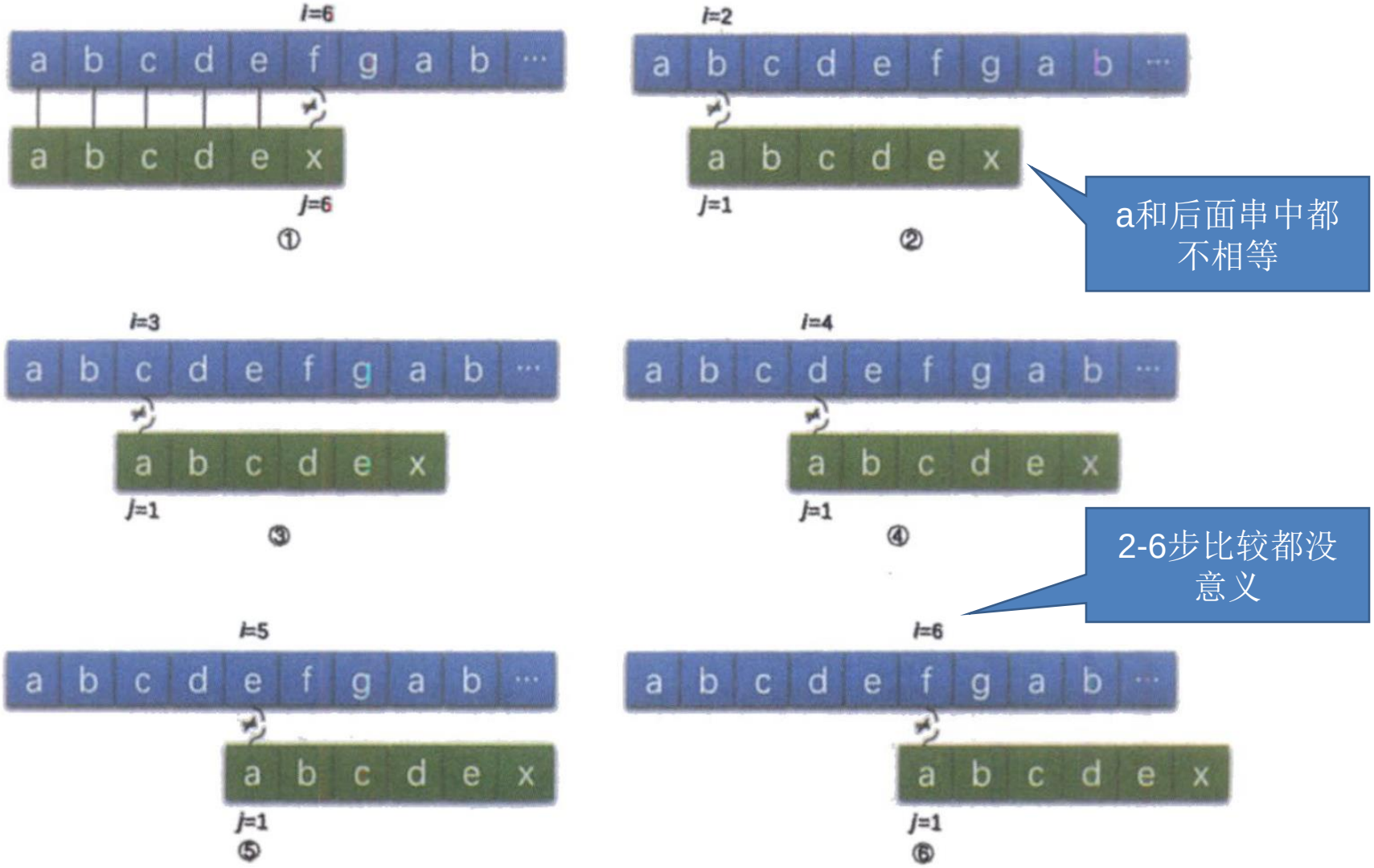
2. KMP快速模式匹配

- D. E. Knuth， J. H. Morris， V. R. Pratt同时发现
- 无回溯的模式匹配



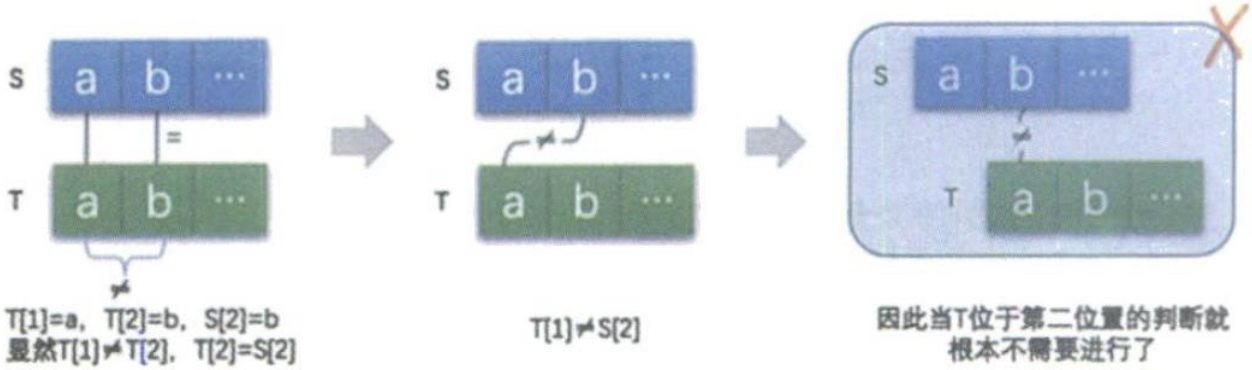
KMP算法（特点：速度快）

2. KMP快速模式匹配

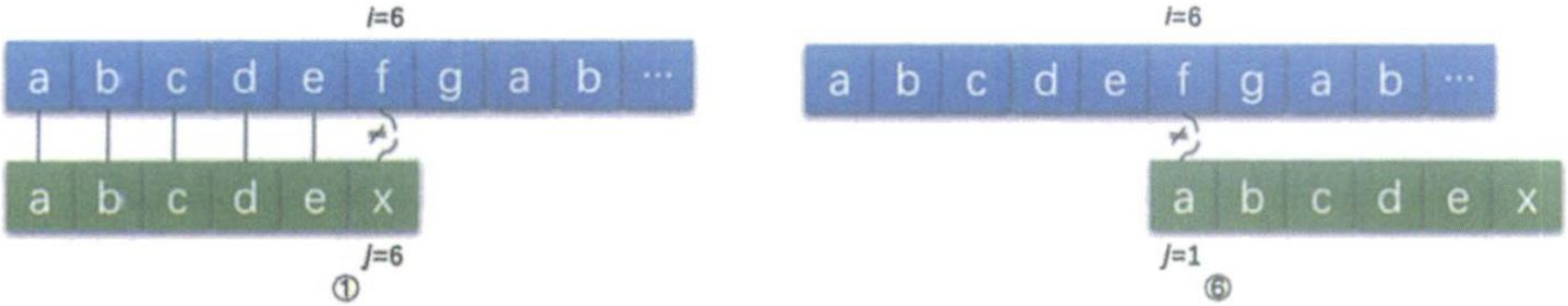

i=6
a b c d e f g a b ···
a b c d e x
j=6
①
i=2
a b c d e f g a b ···
a b c d e x
j=1
②
a和后面串中都不相等
i=3
a b c d e f g a b ···
a b c d e x
j=1
③
i=4
a b c d e f g a b ···
a b c d e x
j=1
④
2-6步比较都没意义
i=5
a b c d e f g a b ···
a b c d e x
j=1
⑤
i=6
a b c d e f g a b ···
a b c d e x
j=1
⑥

2. KMP快速模式匹配

首字符“a,’ 与T中后面的字符均不相等，且后续字符已与主串进行比较且相等，则可以省略步骤。

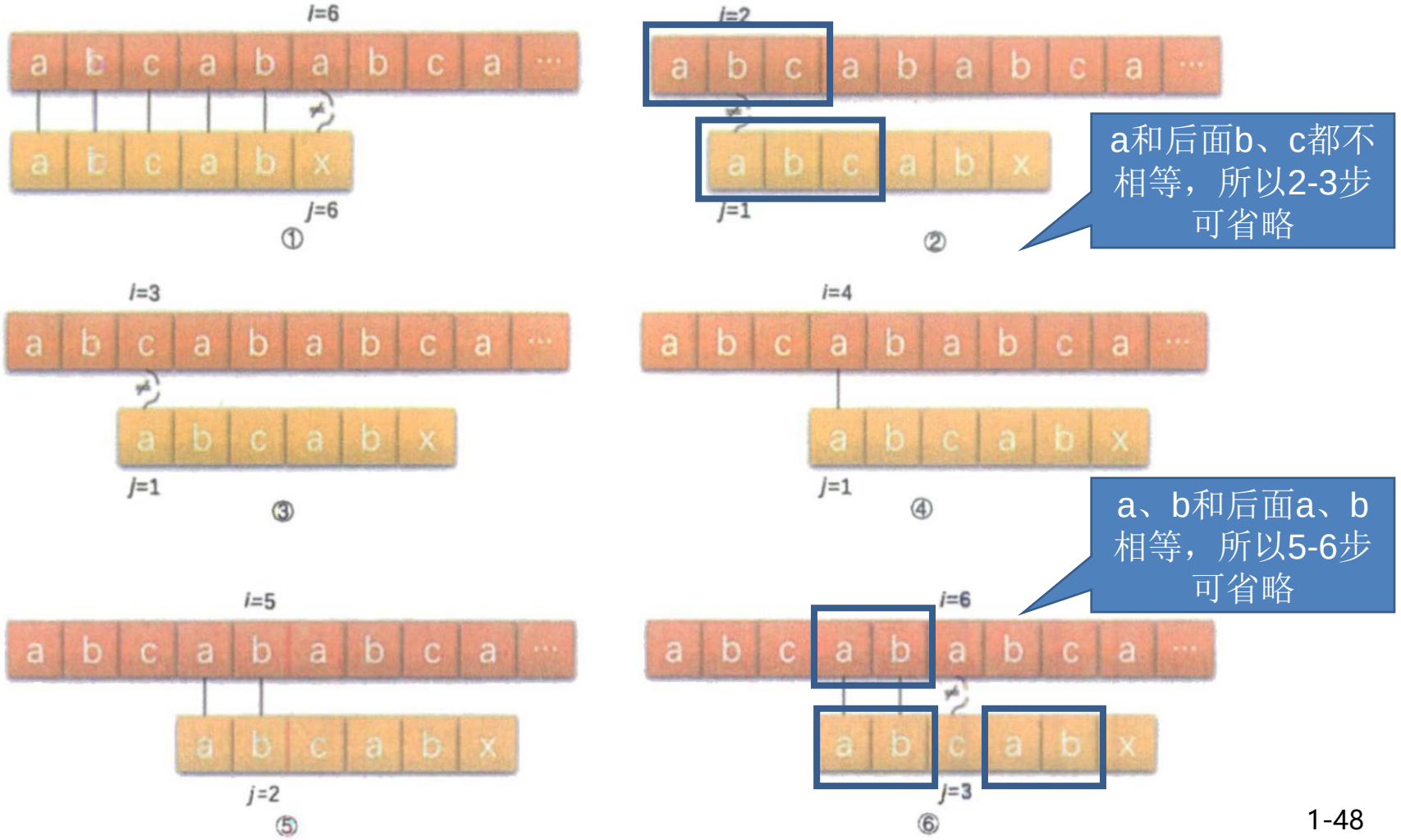


直接从主串中不匹配的位置开始新一轮查找。



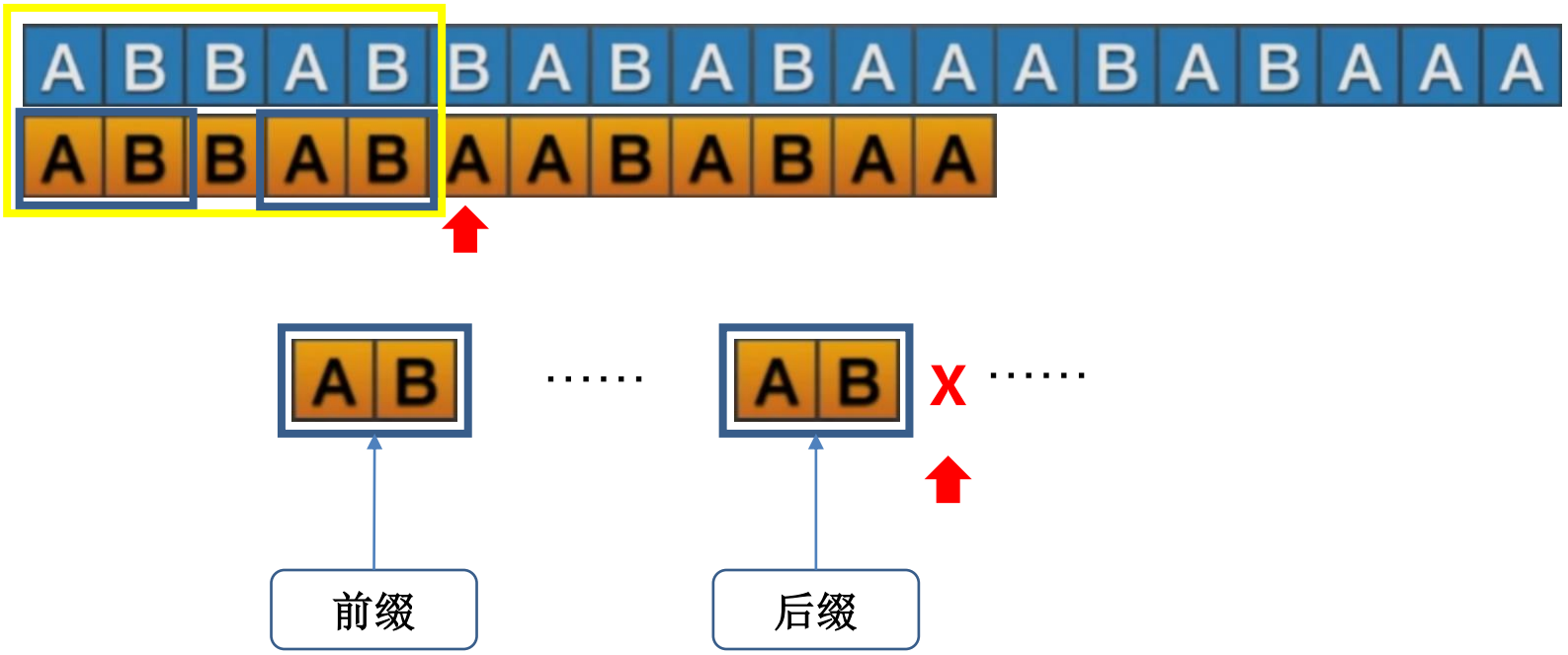
2. KMP快速模式匹配

设：S＝”abcababca” T＝”abcabx”

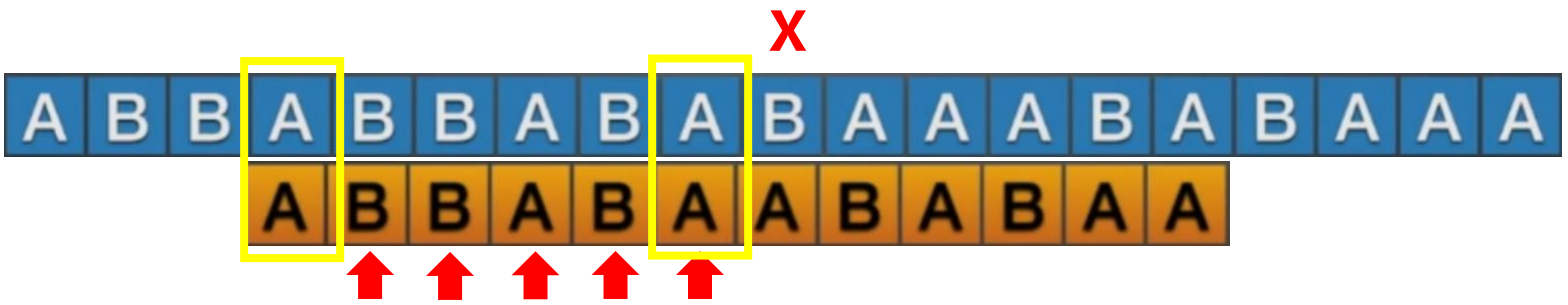


2. KMP快速模式匹配

2. KMP快速模式匹配



2. KMP快速模式匹配



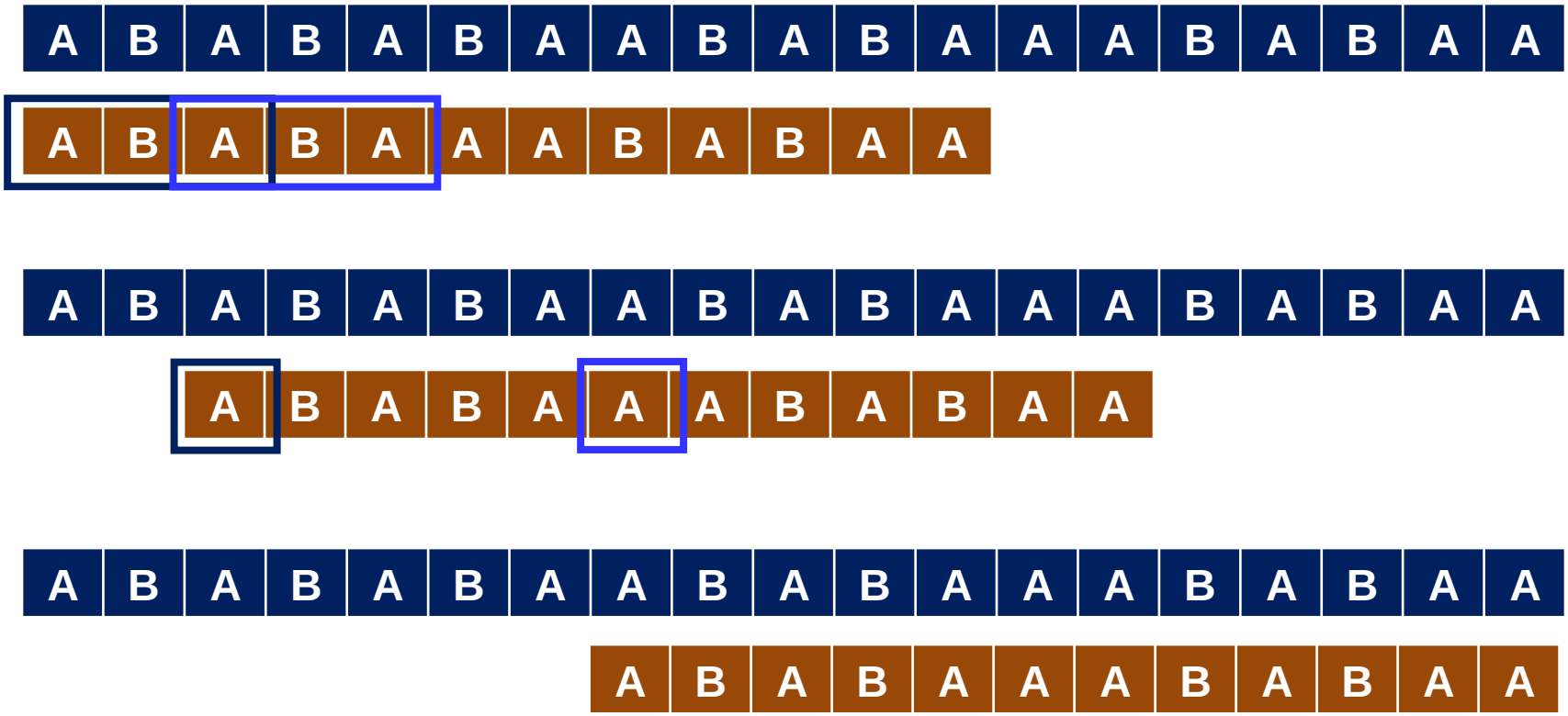
2. KMP快速模式匹配

需要讨论两个问题:

①如何由当前部分匹配结果确定模式向右滑动的新比较起点k?

② 模式应该向右滑多远才是高效率的?

2. KMP快速模式匹配


A B A B A B A A B A B A A A B A B A A
A B A B A A A B A B A A
A B A B A B A A B A B A A A B A B A A
A B A B A A A B A B A A
A B A B A B A A B A B A A A B A B A A
A B A B A A A B A B A A

2. KMP快速模式匹配

根据模式串T的规律：　‘$P_1 \cdots P_{k-1}$’=‘$P_{j-(k-1)} \cdots P_{j-1}$’

由当前失配位置j(已知)，可以归纳出计算新起点k的表达式。

令next[j] = k（k 与j 显然具有函数关系），则

$$next[j]=\begin{cases} 0 & \text{当}j=1\text{时}\quad \text{//不比较} \\ \max\{k \mid 1<k<j \text{ 且 } ‘P_1 \cdots P_{k-1}’=‘P_{j-(k-1)} \cdots P_{j-1}’\} \\ 1 & \text{其他情况} \end{cases}$$

取P首之后与Pj处之前最大的相同子串

2. KMP快速模式匹配

K=next[j]物理含义

当主串在i处和模式串在j处失配时，主串下一次比较位置在失配位置处不变，保持为i，主串指针不用回朔；

当主串在i处和模式串在j处失配时，next[j]的含义是用模式串在next[j]所在位置的字符和主串在i处的字符重新开始比较，next[j]值越小，模式串向右移动越快。

所以kmp算法的关键是如何求模式串所有位置的next值。

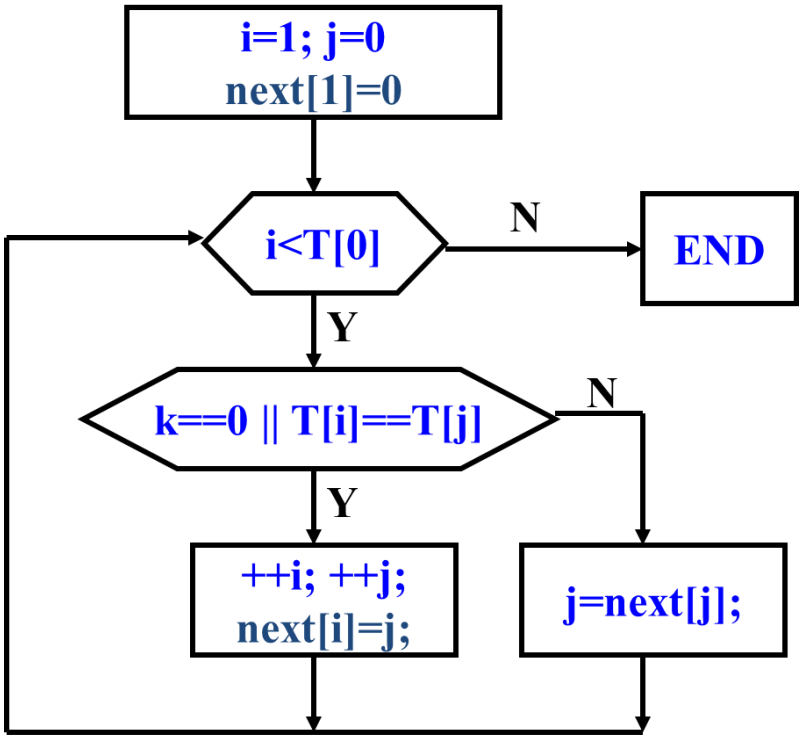
求next数组的步骤

A	B	A	B	A	A	A	B	A	B	A	A

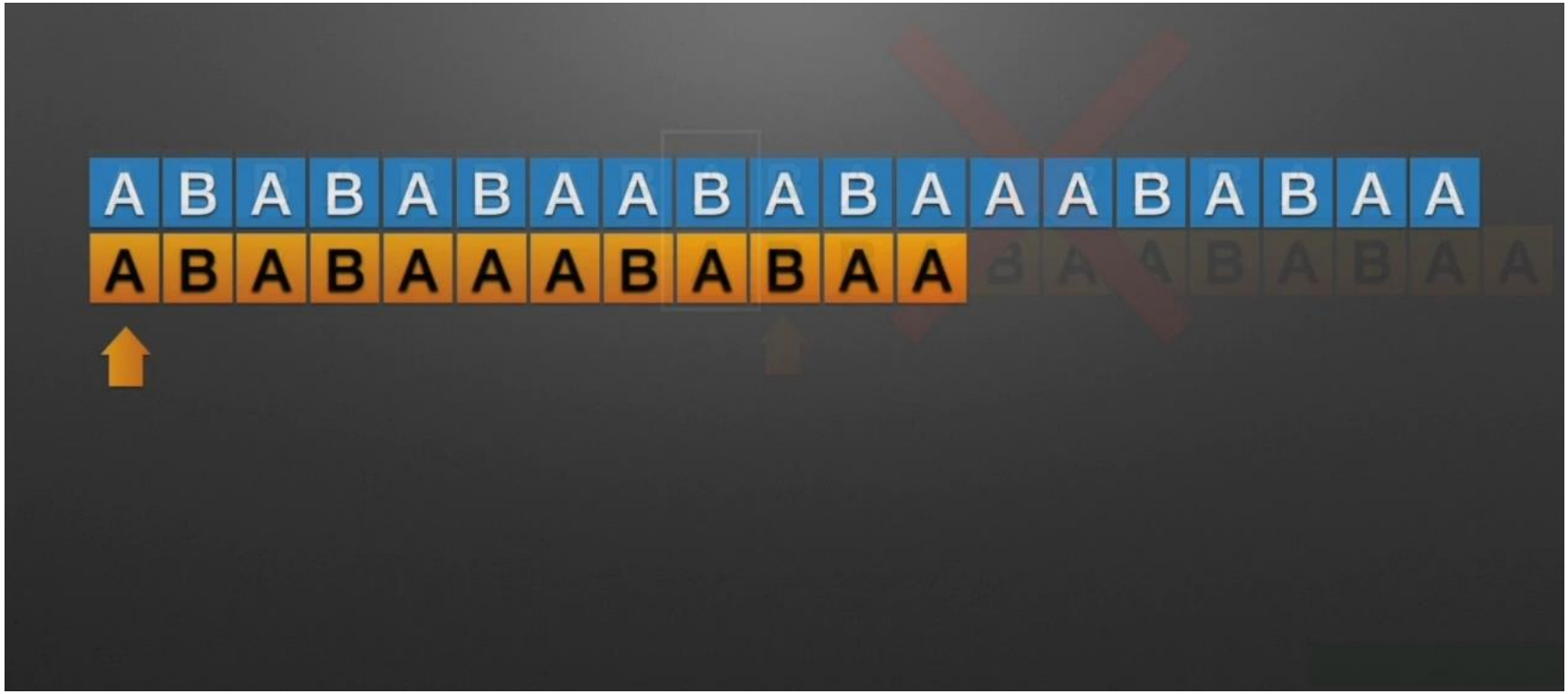
下标	1	2	3	4	5	6	7	8	9	10	11	12
next	1	1	1	2	3	4	2	2	3	4	5	6

2. KMP快速模式匹配

```
void get_next(SString T, int  &next[ ] ){ //
      //求模式串T的next函数值并存入数组next[ ]。
  i=1;  next[1]=0; j=0;
 while(i<T[0] ){
   if(j= = 0||T[i]= =T[j]){
        ++i; ++j; next[i]=j;
   }else
        j=next[j];
  }
}// get_next
```



2. KMP快速模式匹配



改进的求next数组方法

next [j]是否完美无缺?

前面定义的next函数在某些情况下还是有缺陷的，

例如模式aaaab与主串aaabaaaab匹配时的情况：

先计算

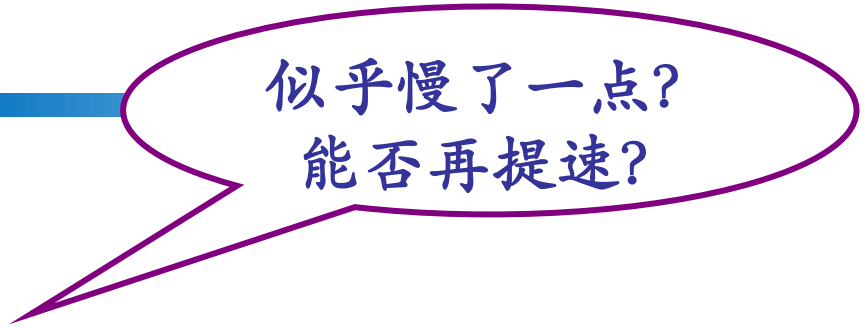
next[j]：

j：1 2 3 4 5

T：a a a a b

next[j]：0 1 2 3 4

改进的求next数组方法



i: 1 2 3 4 5 6 7 8 9

S: a a a b a a a a b

T: a a a a b

a a a a a b b b b

此时效率不高的原因为：子串前4位相同时，主串字符若与其中一个不相等，则不必再与其余3个比较。而实际上还在依次比较。

由此派生出next函数的改进算法

改进的求next数组方法

设next[i]=j

若T[i]=T[j]，则nextval[i]=nextval[j]

j	1 2 3 4 5
模式	*a a a a b*
next[j]	0 1 2 3 4
nextval[j]	0 0 0 0 4

模式匹配比较

主串S=“00000000020000000002000000000200000000020000000001”

子串T=“0000000001”

从主串S中查找是否有子串存在，返回对应位置

BF模式匹配算法　　循环 285 次 得到查询结果

KMP模式匹配算法　　循环 110 次 得到查询结果

KMP模式匹配改进算法　　循环 70 次 得到查询结果

模式匹配比较

- BF算法的模式匹配算法时间代价：
  最坏情况比较n-m+1趟，每趟比较m次，
  总比较次数达(n-m+1)*m
- 原因在于每趟重新比较时，目标串的检测指针要回退。改进的模式匹配算法可使目标串的检测指针每趟不回退。
- 改进的模式匹配(KMP)算法的时间代价：
  - 若每趟第一个不匹配，比较n-m+1趟，总比较次数最坏达(n-m)+m = n
  - 若每趟第m个不匹配，总比较次数最坏亦达到 n
  - 求next函数的比较次数为m, 所以总的时间复杂度是O(n+m)

正在答疑