问题来源

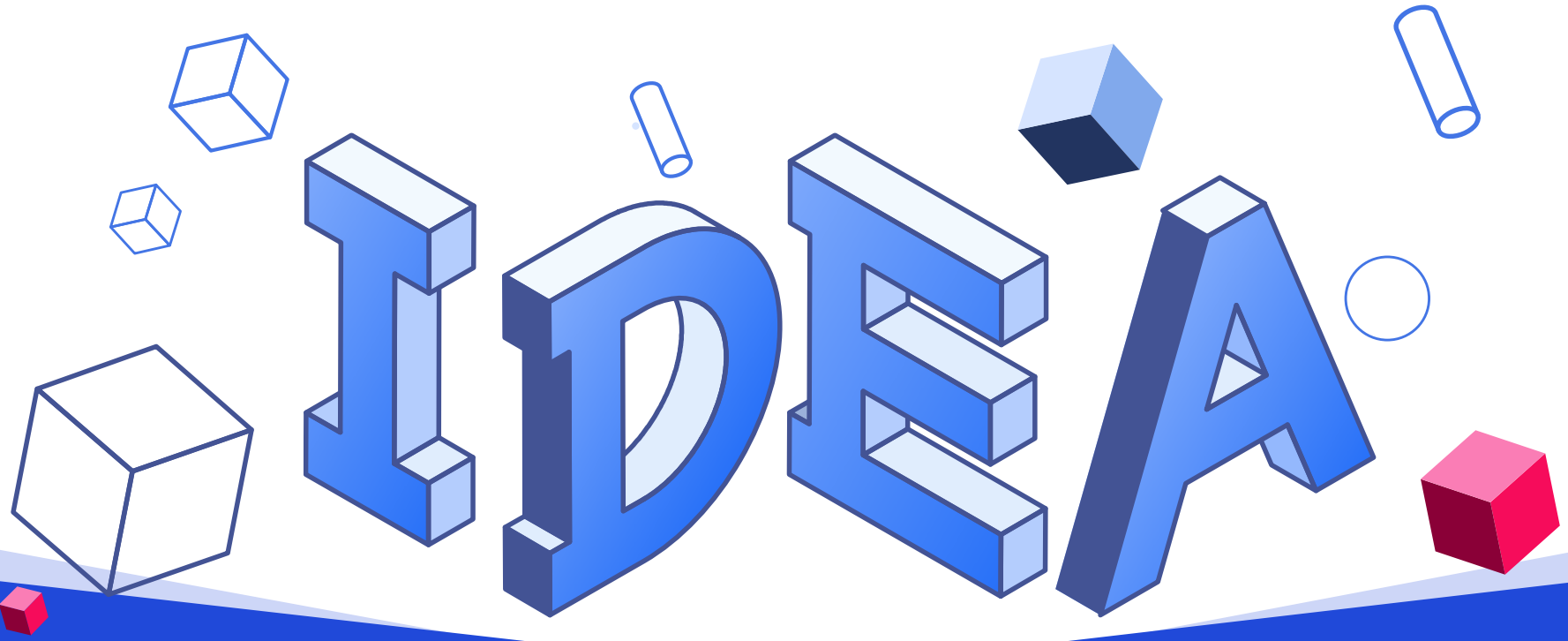
在字符串（也叫主串）中的模式（pattern）定位问题

简单点就是我们平时常说的关键字搜索，判断一个模式串是否在另外一个字符串中出现，出现几次等问题

当然这个问题通过预处理哈希值也能做

但是人的第一思维肯定是去字符一个个去比较、判断是否相等

BF算法和KMP算法就是这样的一个过程



信息学

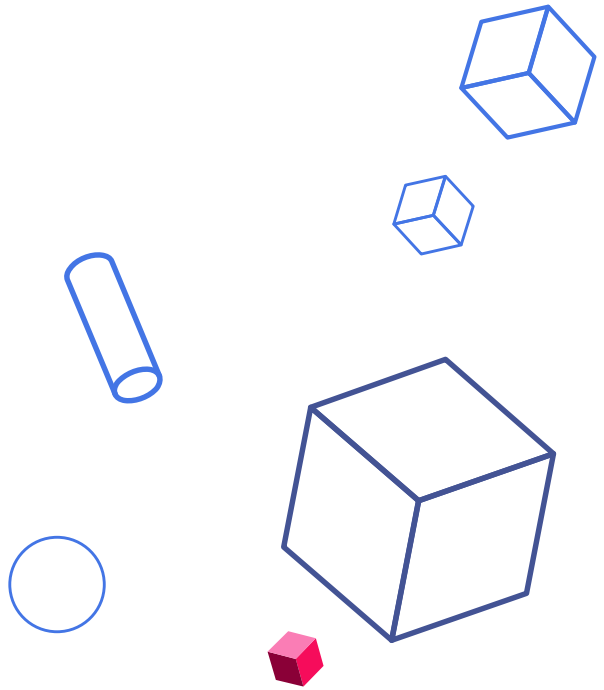
BF算法和KMP算法

西南大学附属中学校

信息奥赛教练组

FZOI
西大附中信息学竞赛
High School Affiliated to Southwest University

西南大学附属中学
High School Affiliated to Southwest University

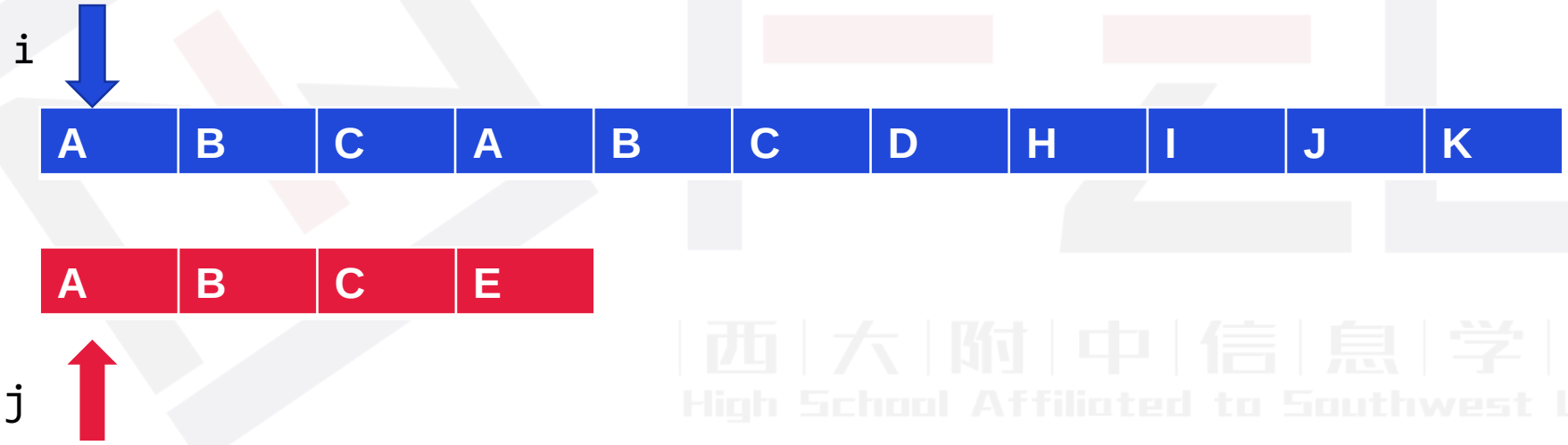


BF算法

BF算法

Brute Force，直译暴力算法
究竟有多暴力呢?
来看一个例子

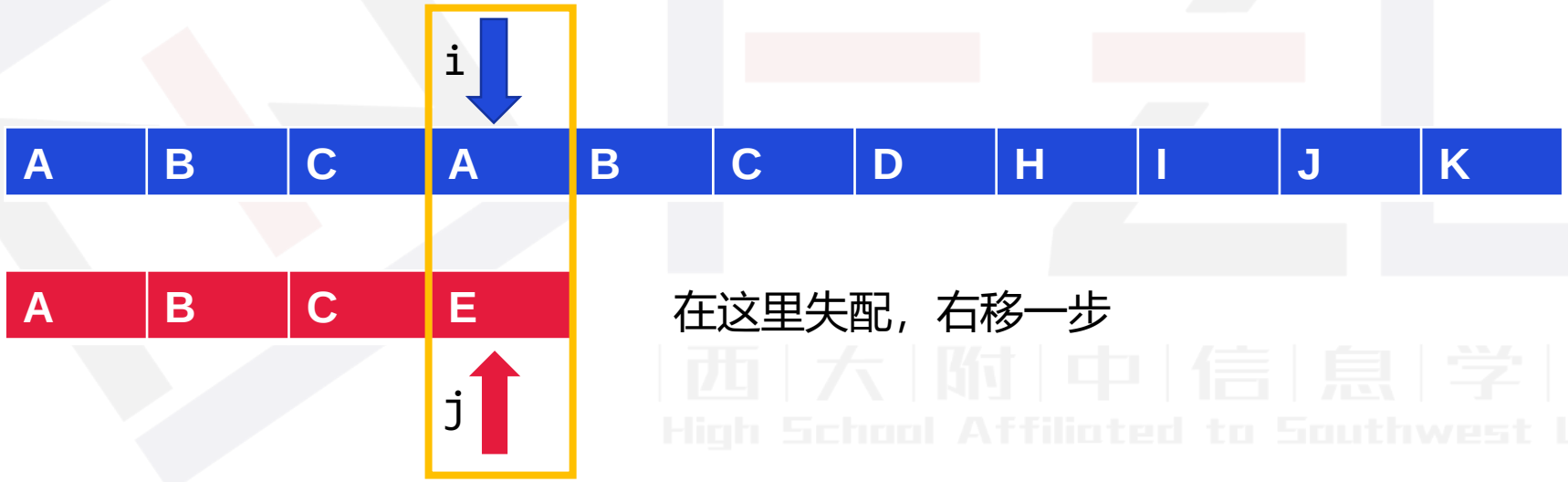
- 从左到右一个个匹配，如果这个过程中有某个字符不匹配，就跳回去，将模式串向右移动一位。



BF算法

Brute Force，直译暴力算法
究竟有多暴力呢?
来看一个例子

- 需要比较i指针指向的字符和j指针指向的字符是否一致。

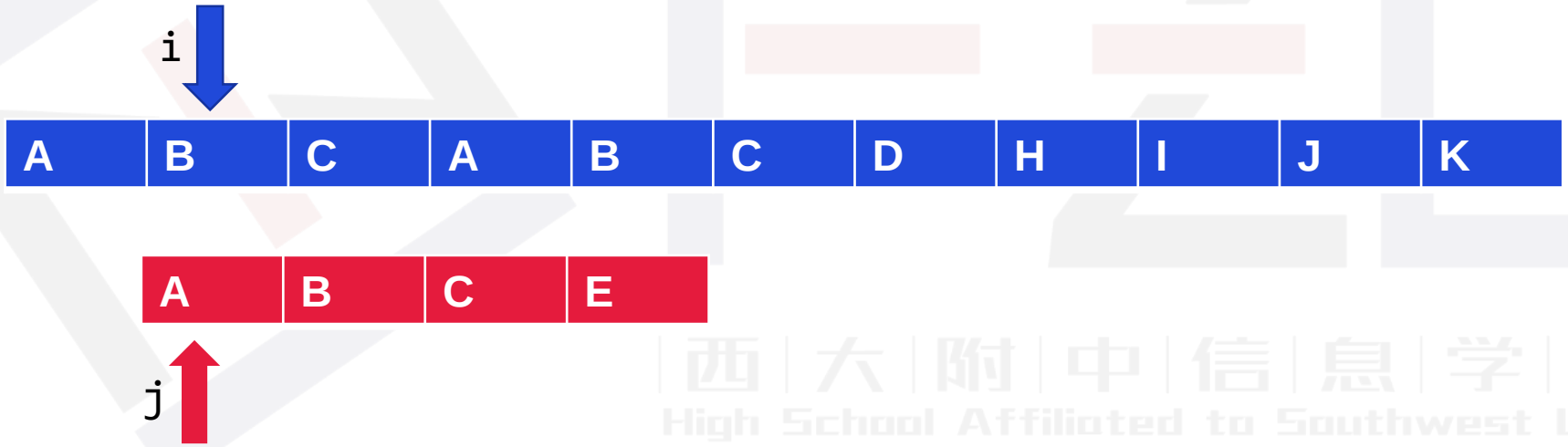


在这里失配，右移一步

BF算法

Brute Force，直译暴力算法
究竟有多暴力呢?
来看一个例子

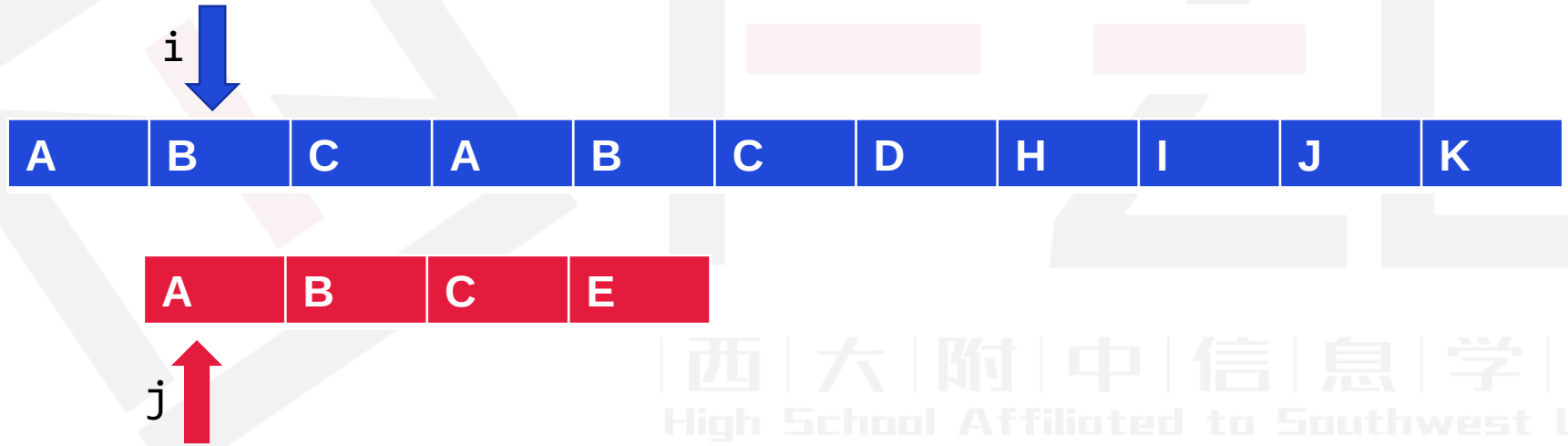
- 那就把i指针移回第1位（假设下标从0开始），j移动到模式串的第0位



BF算法

Brute Force，直译暴力算法
究竟有多暴力呢?
来看一个例子

- 重复以上过程，直到在主串里找到对应的模式串



算法过程

txt	a	a	a	c	a	a	a	b

	0	1	2	3
pat	a	a	a	b

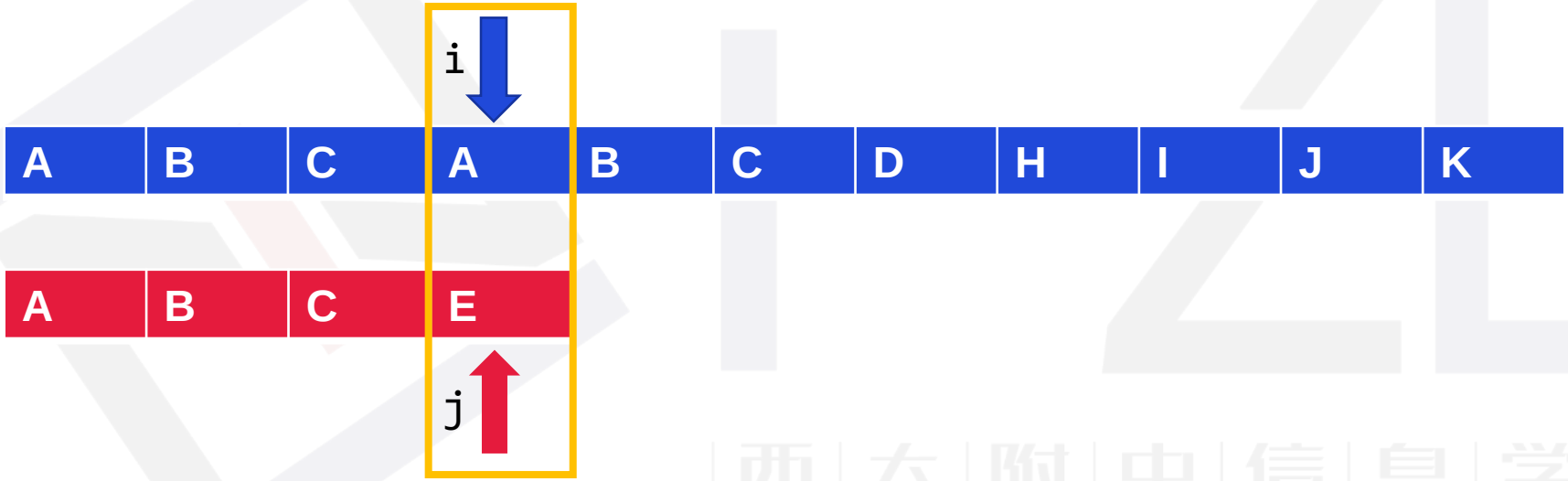
BF算法-核心代码


```
int i = 0; // 主串的位置
int j = 0; // 模式串的位置
while (i < t.length && j < p.length) {
    if (t[i] == p[j]) { //当两个字符相同，就比较下一个
        i++;
        j++;
    }
    else {
        i = i - j + 1; // 一旦不匹配，i后退
        j = 0; // j归0
    }
}
if (j == p.length) return i - j;  //返回模式串在主串中的下标
else return -1;
```

BF算法分析

上述这个算法很直观，但是时间复杂度不够好

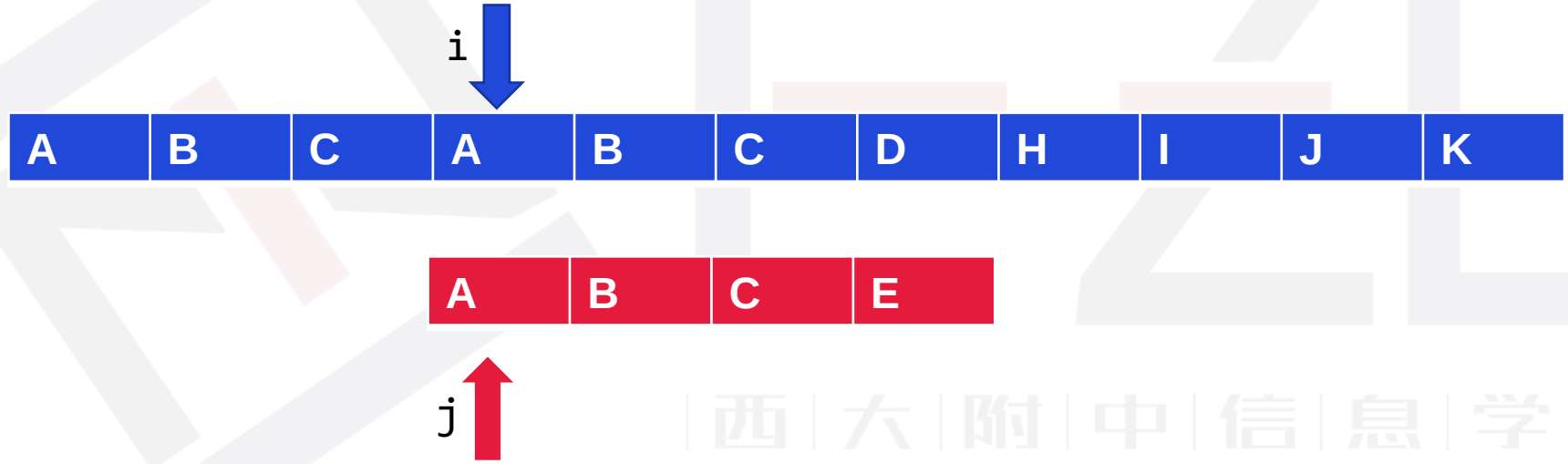
如果是人来操作会怎么移动字符串？



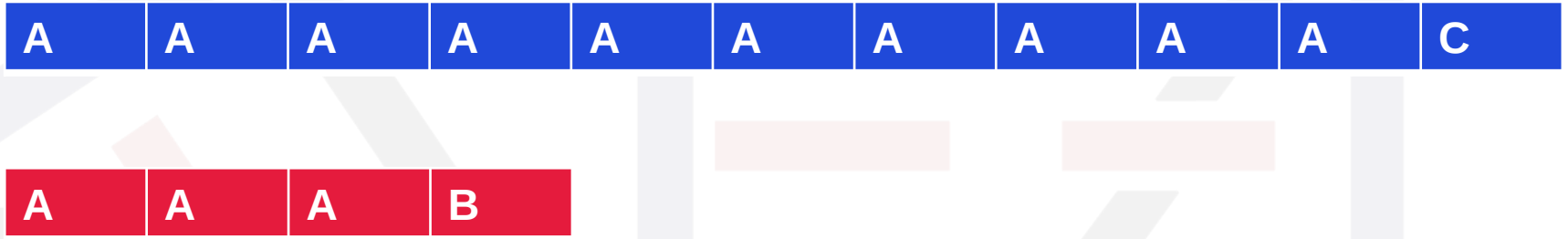
算法慢在哪里？ 比较的过程

BF算法分析

可以发现前三个都是匹配的，并且了前三个里只有最开始有A
再次移动i肯定也是不匹配的
所以，i可以不动，我们只需要移动j即可，如下图：



BF算法分析



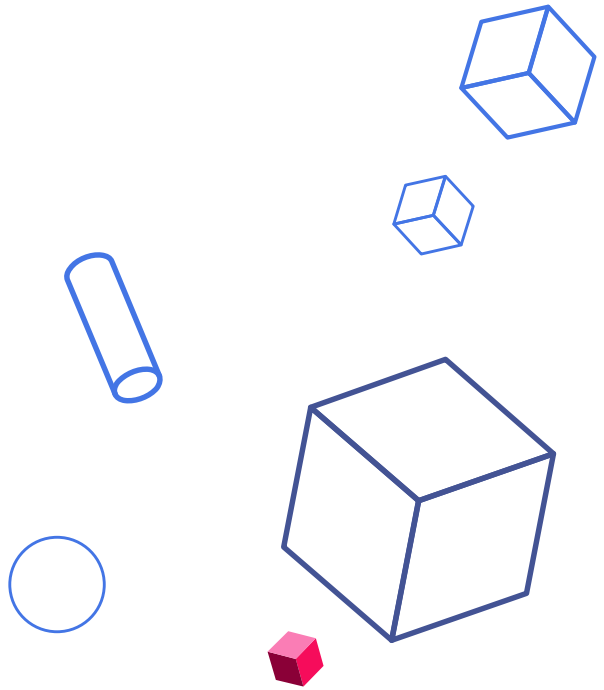
而在这种情况下，我们需要在最后才知道不匹配，刚才的做法，效率显然很低

如果任何一个位置失配都能寻找到最有效的回溯位置，那就好了

逐步优化成**KMP算法**

FZOI
西大附中信息学竞赛
High School Affiliated to Southwest University

西南大学附属中学
High School Affiliated to Southwest University



KMP算法

KMP算法

KMP是三位大牛：D.E.Knuth、J.H.Morris和V.R.Pratt同时发现的

KMP的核心算法思想：

- **尽可能的利用已有的匹配信息来提高匹配效率**
- **保持i指针不回溯，通过修改j指针，让模式串尽量地移动到有效的位置**

KMP核心的问题：指针j应该在失配后如何高效的移动

KMP的重中之重：求解一个**next**数组

next[j]表示的是模式串长度为j的**前缀和后缀**相等的最大长度

作用：在下标j处失配时，指针将会回退到next[j]的位置，重新匹配

$$\text{next}[j] = \begin{cases} -1 & j = 0(\text{不存在t[0]到t[j-1]的子串，无意义}) \\ k, 1 <= k <= j & \text{t[0]到t[j-1]序列中最长的相同前缀与后缀长度为k，即t[0]...t[k] = t[j-k]...t[j-1]} \\ 0 & \text{t[0]到t[j-1]序列中不存在相同的前缀与后缀)} \end{cases}$$

前缀和后缀

前缀，除最后一个字符的所有子串。
后缀，除第一个字符的所有子串。
最长公共前后缀。假设有一个串P=“$p_0p_1p_2 \dots p_{j-1}p_j$”。
如果存在$p_0p_1 \dots p_{k-1}p_k = p_{j-k}p_{j-k+1} \dots p_{j-1}p_j$，就存在一个长度位k+1的最长公共前后缀。

模式串的子串	前缀	后缀	公共前后缀最长长度
a	无	无	0
ab	a	b	0
aba	a,ab	a,ba	1
abaa	a,ab,aba	a,aa,baa	1
abaab	a,ab,aba,abaa	b,ab,aab,baab	2
abaabc	a,ab,aba,abaa,abaab	c,bc,abc,aabc,baabc	0
abaabca	a,ab,aba,abaa,abaab,abaabc	a,ca,bca,abca,aabca,baabca	1

字符	a	b	a	a	b	c	a
最长公共前后缀	0	0	1	1	2	0	1

前缀和后缀

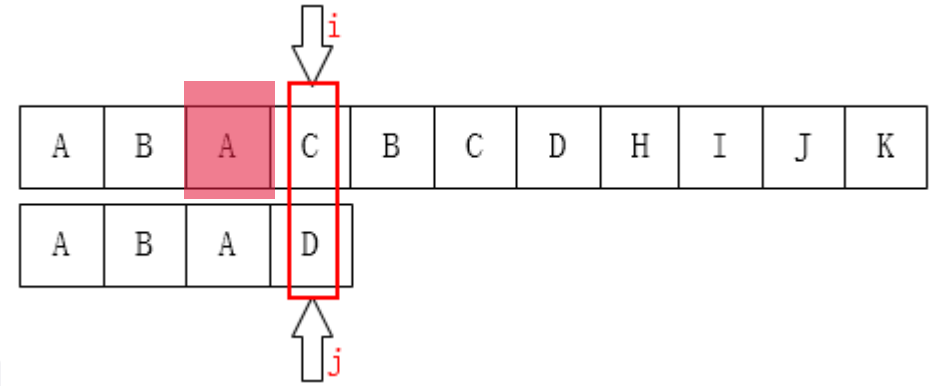
模式串的子串	前缀	后缀	公共前后缀最长长度
a	无	无	0
ab	a	b	0
aba	a,ab	a,ba	1
abaa	a,ab,aba	a,aa,baa	1
abaab	a,ab,aba,abaa	b,ab,aab,baab	2
abaabc	a,ab,aba,abaa,abaab	c,bc,abc,aabc,baabc	0
abaabca	a,ab,aba,abaa,abaab,abaabc	a,ca,bca,abca,aabca,baabca	1

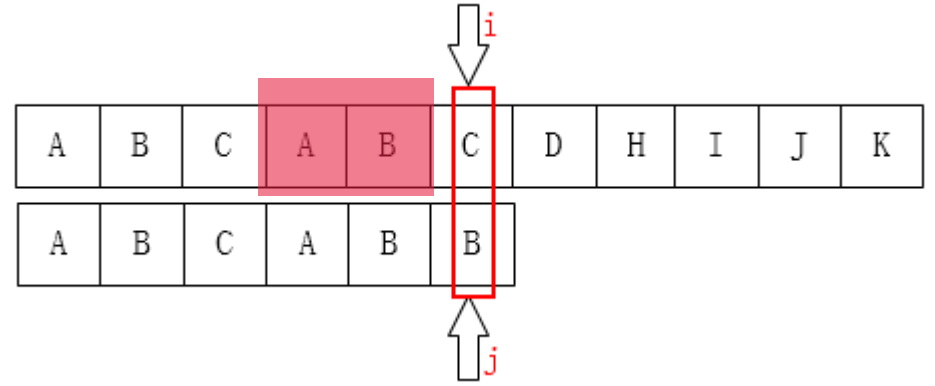
next在计算时，不包括当前字符

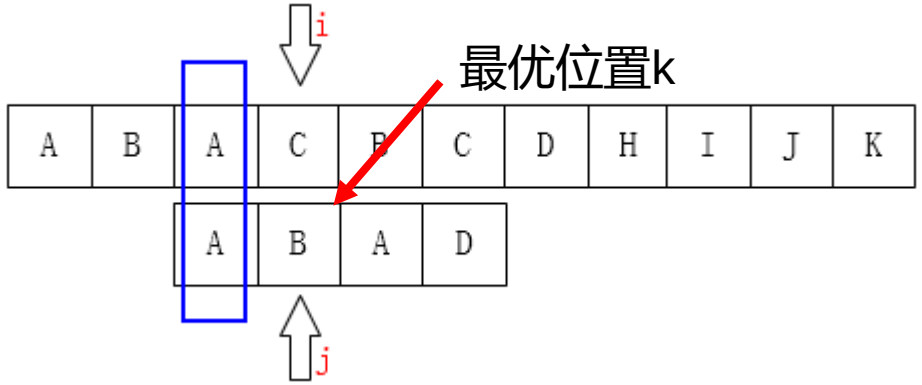
字符	a	b	a	a	b	c	a
next	-1	0	0	1	1	2	0

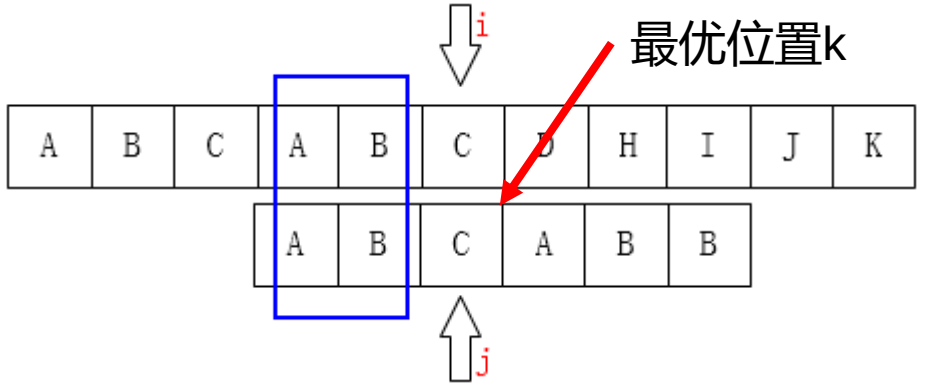
KMP算法

为什么要求解这个next数组，我们看个例子







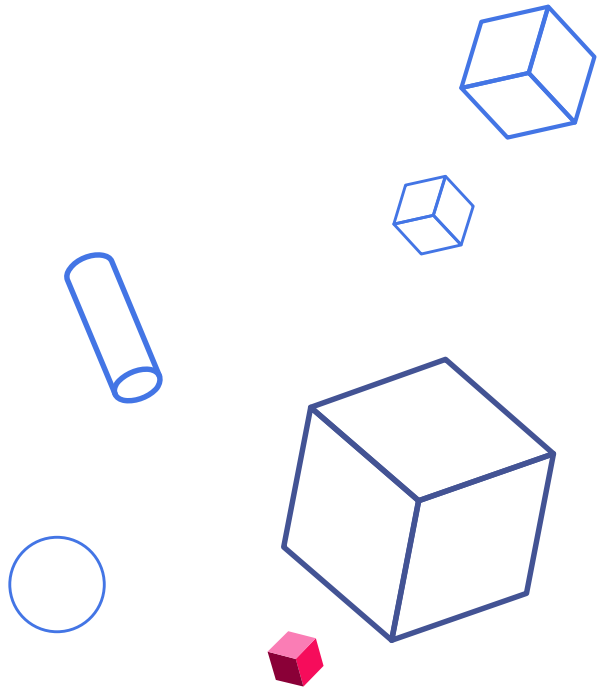


当匹配失败时，j要移动的下一个位置k。

这个k存在着这样的性质：模式串T最前面的k个字符和j之前的最后k个字符是一样的

根据next的定义：next[j]=k

next数组求解过程

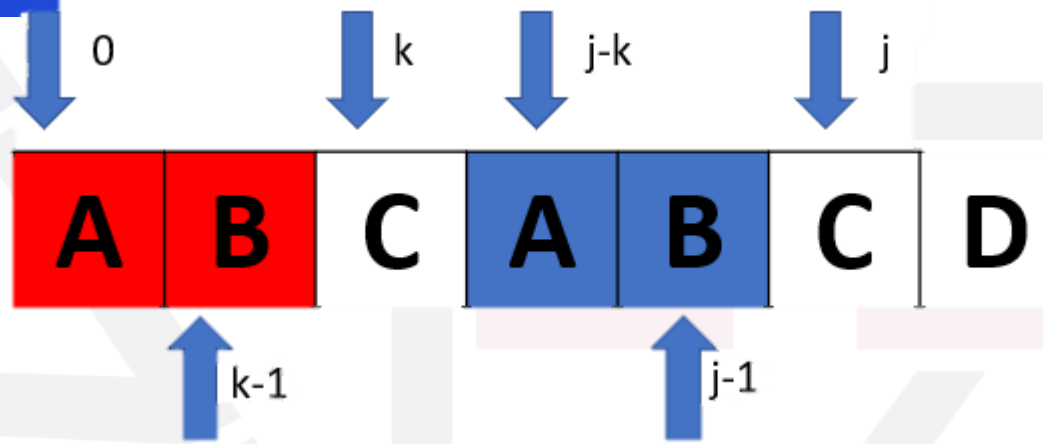


求解next

T为模式串

当 T[j] == T[k] 的情况

模式串：



next[j+1]=next[j]+1=k+1。

观察上图可知，当 T[j] == T[k] 时，必然有“T[0]…T[k-1]" == " T[j-k]…T[j-1]",
此时的k即是相同子串的长度。
因为有“T[0]…T[k-1]” == “ T[j-k]…T[j-1]”，且 T[j] == T[k]，则有“T[0]…T[k]
" == “T[j-k]…T[j]"

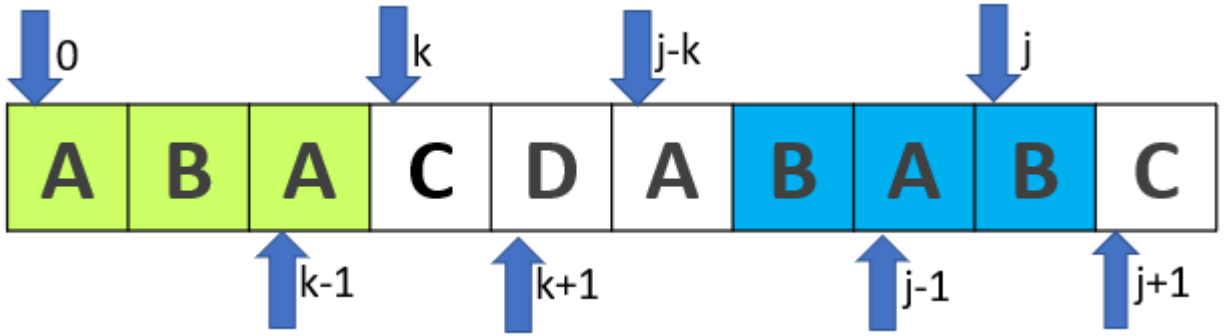
根据数学归纳法可得：

next[j+1]=k+1。

求解next

当 T[j] != T[k] 的情况

模式串：



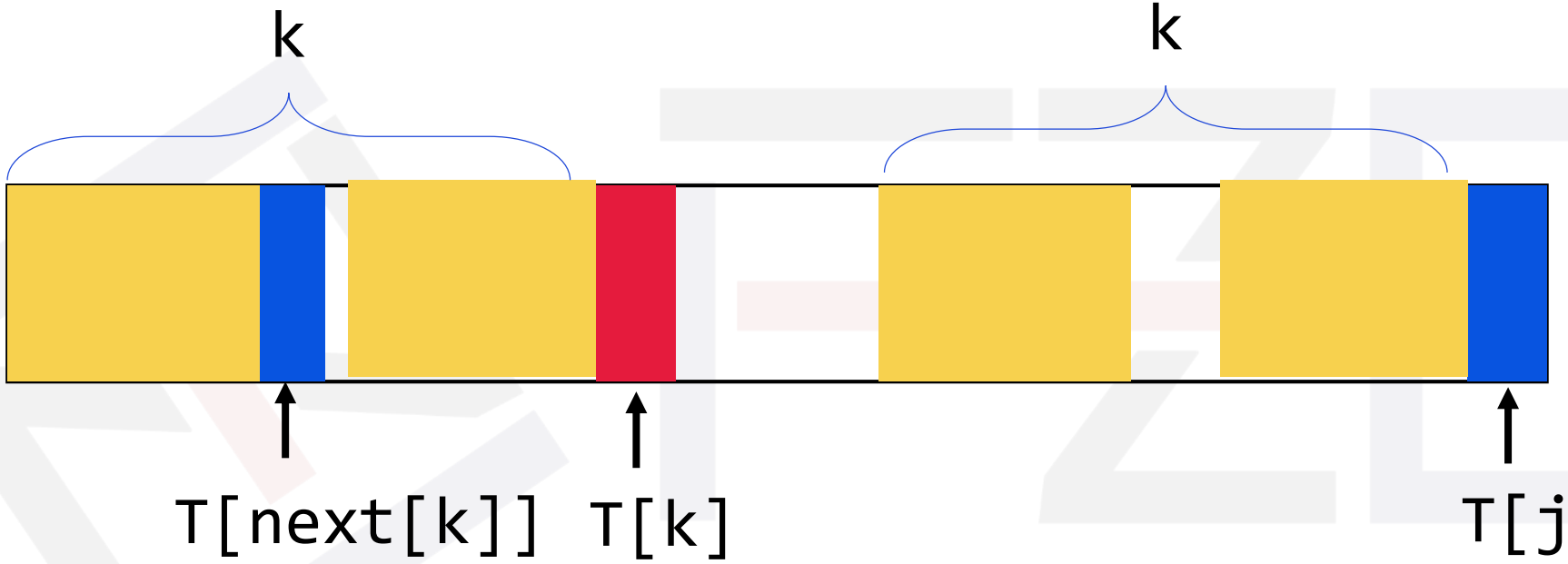
这时候next[j+1]!= k+1

KMP精髓代码来了： k = next[k];

为什么要会退到 next[k] 的位置？为什么不是回退到k-1，k-2

求解next

当 T[j] != T[k] 的情况



由于T[j] != T[k] ，所以不能使得next[j+1]=k+1，也就是说不存在k+1的公共前后缀
由next[j]=k，可知，存一个长度为k的已匹配串，那就说明可能存在一个长度为next[k]公共前后缀
如果不存在next[k]的公共前后缀，那么可以继续next[next[next[next[k]]]]…

```
k = next[k];
```

求解next

特殊情况

next[0] = -1,next[1] =0。

next函数


```
void Getnext(string t){
    int j=0,k=-1;
    nex[j] = k;
    while(j<t.length()-1){
        if(k == -1 || t[j] == t[k]){
            j++;
            k++;
            nex[j] = k;
         }
        else k = nex[k];
    }
}
```

因为求解next数组的过程其实就是一个模式串自匹配的过程

本质：DP

next求解


```
void Getnext(string t){
   int j=0,k=-1;
   nex[j] = k;
   while(j<t.length()-1){
      if(k == -1 || t[j] == t[k]){
         j++;
         k++;
         nex[j] = k;
       }
      else k = nex[k];
   }
}
```

数组下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
字符	a	g	c	t	a	g	c	a	g	c	t	a	g	c	t

根据next求解过程，完善表格

字符	a	g	c	t	a	g	c	a	g	c	t	a	g	c	t
next															

next求解

西南大学附属中学
High School Affiliated to Southwest University

数组下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
模式串元素	a	g	c	t	a	g	c	a	g	c	t	a	g	c	t
next	-1	0	0	0	0	1	2	3	1	2	3	4	5	6	7

KMP函数


```
int KMP(string s,string t){  //s是主串，t是模式串
   int nex[MaxSize],i=0;j=0;
   Getnext(t,nex);  //计算出一个next数组
   while(i<s.length()&&j<t.length()){
      if(j==-1 || s[i]==t[j]){
         i++;
         j++;
      }
      else j=nex[j];          //j回退next[j]步
   }
   if(j>=t.length())
       return (i-t.length()); //匹配成功，返回子串的位置
   else
      return -1;              //没找到
}
```

O(m+n)

m处理next数组，n遍历串

- KMP算法只预处理模式串，关键点在于求解next数组
- 理解了next数组的作用，KMP算法就理解得八九不离十了
- 由于KMP只处理模式串，所以很适合求解这样的问题：

“给定一个模式串和若干不同的主串，问模式串是多少个主串的子串？”

KMP算法给了我们一个解题的启示：
充分利用已有的信息，或许能找到解决问题的答案

参考资料

PPT图文部分搬运自：
https://www.cnblogs.com/dusf/p/kmp.html

(58条消息) KMP算法—终于全部弄懂了_dark_cy的博客-CSDN博客_kmp算法

KMP算法详解_yyzsir的博客-CSDN博客_kmp算法（理解层面，墙裂推荐看看）

更本质的理解，从有限状态机的角度理解KMP
https://zhuanlan.zhihu.com/p/83334559

Thanks

For Your Watching