

**串 (Sequence)**

# 串 (Sequence)

- 本课程研究的串是开发中非常熟悉的字符串，是由若干个字符组成的有限序列

String text = "Thank";				
T	h	a	n	k

- 字符串 thank 的前缀 (prefix)、真前缀 (proper prefix)、后缀 (suffix)、真后缀 (proper suffix)

前缀	t, th, tha, than, thank
真前缀	t, th, tha, than
后缀	thank, hank, ank, nk, k
真后缀	hank, ank, nk, k

# 串匹配算法

- 本课程主要研究串的匹配问题，比如
  - 查找一个模式串（pattern）在文本串（text）中的位置

```
String text = "Hello World";  
String pattern = "or";  
text.indexOf(pattern); // 7  
text.indexOf("other"); // -1
```

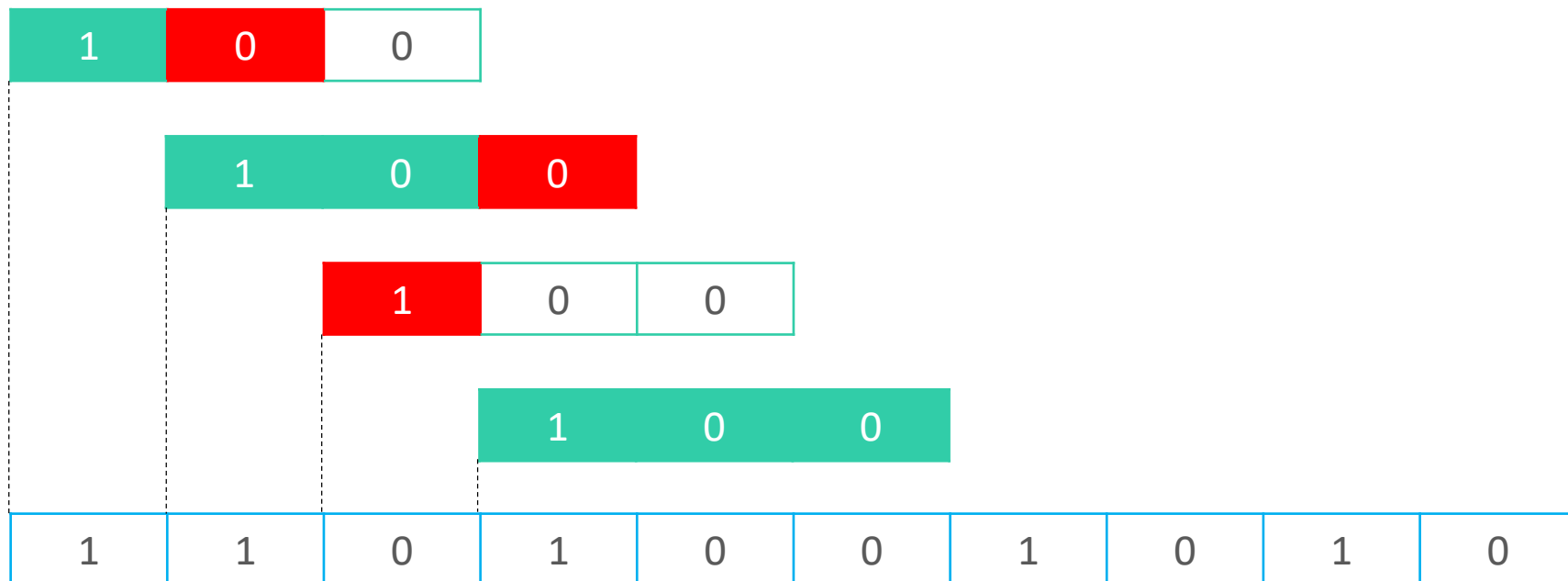
- 几个经典的串匹配算法

- 蛮力（Brute Force）
- KMP
- Boyer-Moore
- Karp-Rabin
- Sunday

- 本课程用 `tlen` 代表文本串 `text` 的长度，`plen` 代表模式串 `pattern` 的长度

# 蛮力 (Brute Force)

- 以字符为单位，从左到右移动模式串，直到匹配成功



- 蛮力算法有 2 种常见实现思路

# 蛮力1 – 执行过程

pi=0

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=0

pi=1

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=1

pi=2

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=2

pi 的取值范围 [0, plen)

ti 的取值范围 [0, tlen)

# 蛮力1 – 执行过程

pi=3

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=3

pi=0

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=1

pi=4

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=7

pi = 0

ti -= pi - 1

pi == plen

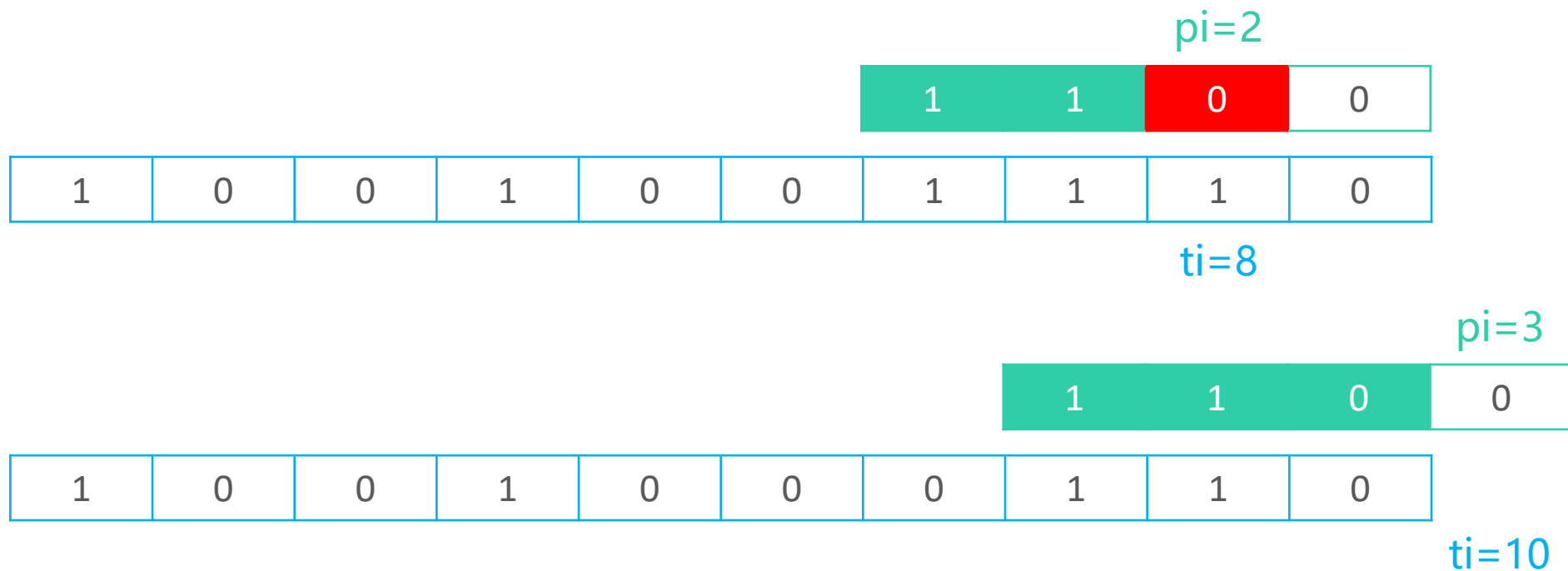
代表匹配成功

# 蛮力1 – 实现

```
public static int indexOf(String text, String pattern) {  
    if (text == null || pattern == null) return -1;  
    int tlen = text.length();  
    int plen = pattern.length();  
    if (tlen == 0 || plen == 0 || tlen < plen) return -1;  
    int pi = 0, ti = 0;  
    while (pi < plen && ti < tlen) {  
        if (text.charAt(ti) == pattern.charAt(pi)) {  
            ti++;  
            pi++;  
        } else {  
            ti -= pi - 1;  
            pi = 0;  
        }  
    }  
    return pi == plen ? ti - pi : -1;  
}
```

# 蛮力1 – 优化

- 此前实现的蛮力算法，在恰当的时候可以提前退出，减少比较次数



- 因此， $ti$  的退出条件可以从  $ti < tlen$  改为

□  $ti - pi \leq tlen - plen$

□  $ti - pi$  是指每一轮比较中 text 首个比较字符的位置

这是完全没必要的比较



# 蛮力1 – 优化实现

```
public static int indexOf(String text, String pattern) {
    if (text == null || pattern == null) return -1;
    int tlen = text.length();
    int plen = pattern.length();
    if (tlen == 0 || plen == 0 || tlen < plen) return -1;
    int pi = 0, ti = 0;
    int tmax = tlen - plen;
    while (pi < plen && ti - pi <= tmax) {
        if (text.charAt(ti) == pattern.charAt(pi)) {
            ti++;
            pi++;
        } else {
            ti -= pi - 1;
            pi = 0;
        }
    }
    return pi == plen ? ti - pi : -1;
}
```

# 蛮力2 – 执行过程

pi=0

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=0

pi=1

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=0    ti + pi

pi=2

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=0            ti + pi

- pi 的取值范围  $[0, \text{plen})$
- ti 的取值范围  $[0, \text{tlen} - \text{plen}]$

# 蛮力2 – 执行过程

pi=3

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=0

ti + pi

pi = 0

ti++

pi=0

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=1

pi=4

1	0	0	0
---	---	---	---

1	0	0	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---

ti=3

ti + pi

pi == plen

代表匹配成功

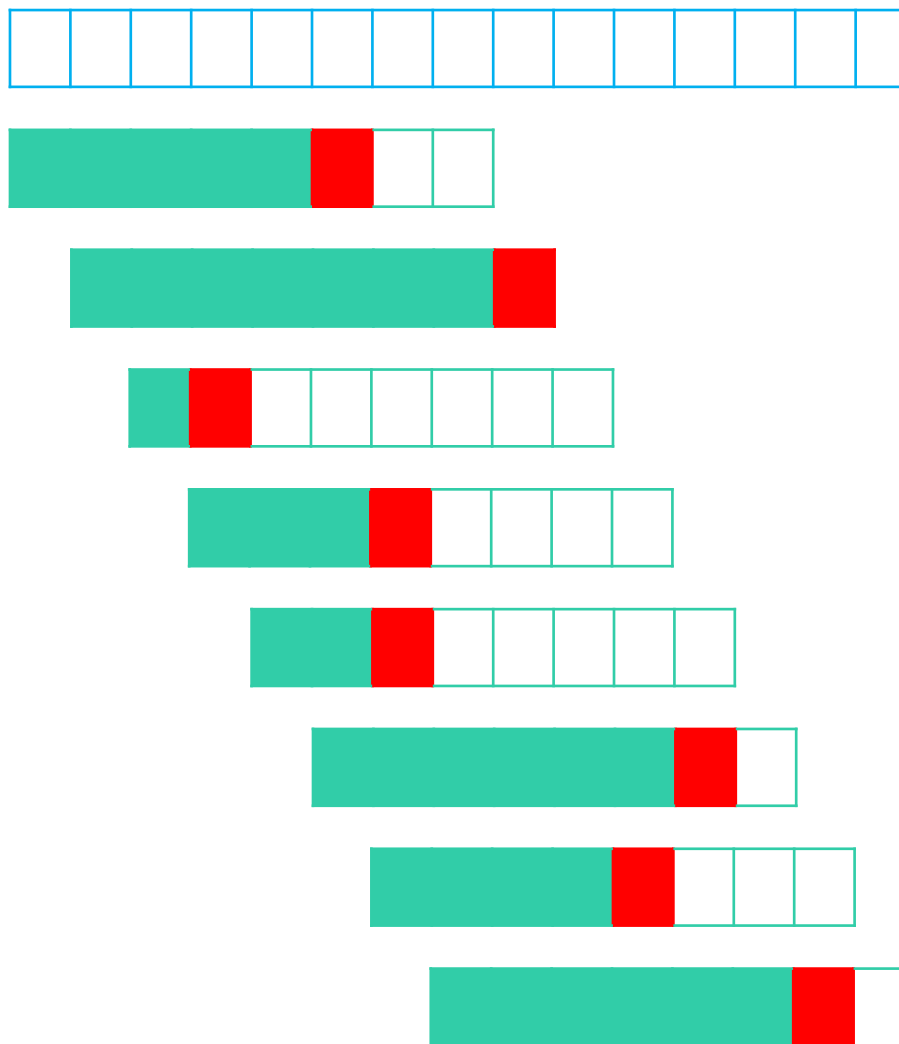
## 蛮力2 – 实现

```
public static int indexOf(String text, String pattern) {  
    if (text == null || pattern == null) return -1;  
    int tlen = text.length();  
    int plen = pattern.length();  
    if (tlen == 0 || plen == 0 || tlen < plen) return -1;  
    int tmax = tlen - plen;  
    for (int ti = 0; ti <= tmax; ti++) {  
        int pi = 0;  
        for (; pi < plen; pi++) {  
            if (text.charAt(ti + pi) != pattern.charAt(pi)) break;  
        }  
        if (pi == plen) return ti;  
    }  
    return -1;  
}
```

# 蛮力 – 性能分析

■  $n$  是文本串长度,  $m$  是模式串长度

最多  $n - m + 1$  轮



# 蛮力 - 性能分析

## ■ 最好情况

- 只需一轮比较就完全匹配成功，比较  $m$  次（ $m$  是模式串的长度）
- 时间复杂度为  $O(m)$

1 0

1	0	0	1	0
---	---	---	---	---

### ■ 最坏情况（字符集越大，出现概率越低）

- ❑ 执行了  $n - m + 1$  轮比较 ( $n$  是文本串的长度)
- ❑ 每轮都比较至模式串的末字符后失败 ( $m - 1$  次成功, 1 次失败)
- ❑ 时间复杂度为  $O(m * (n - m + 1))$ , 由于一般  $m$  远小于  $n$ , 所以为  $O(mn)$

1 1 0

[illegible]

# KMP

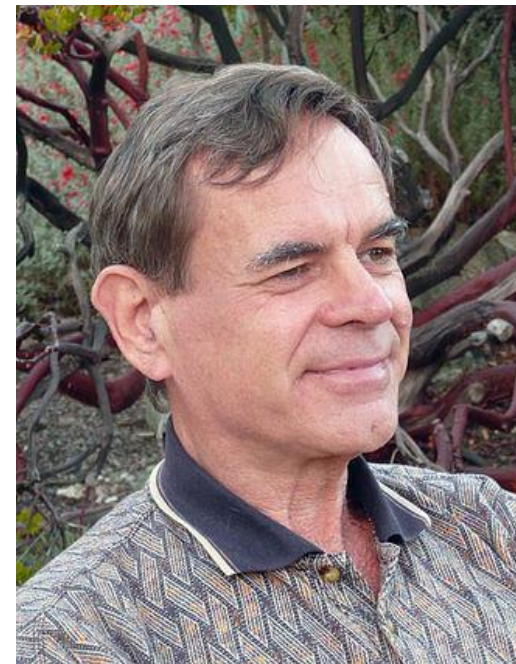
- KMP 是 **K**nuth-**M**orris-**P**ratt 的简称（取名自3位发明人的名字），于1977年发布



Donald **K**nuth



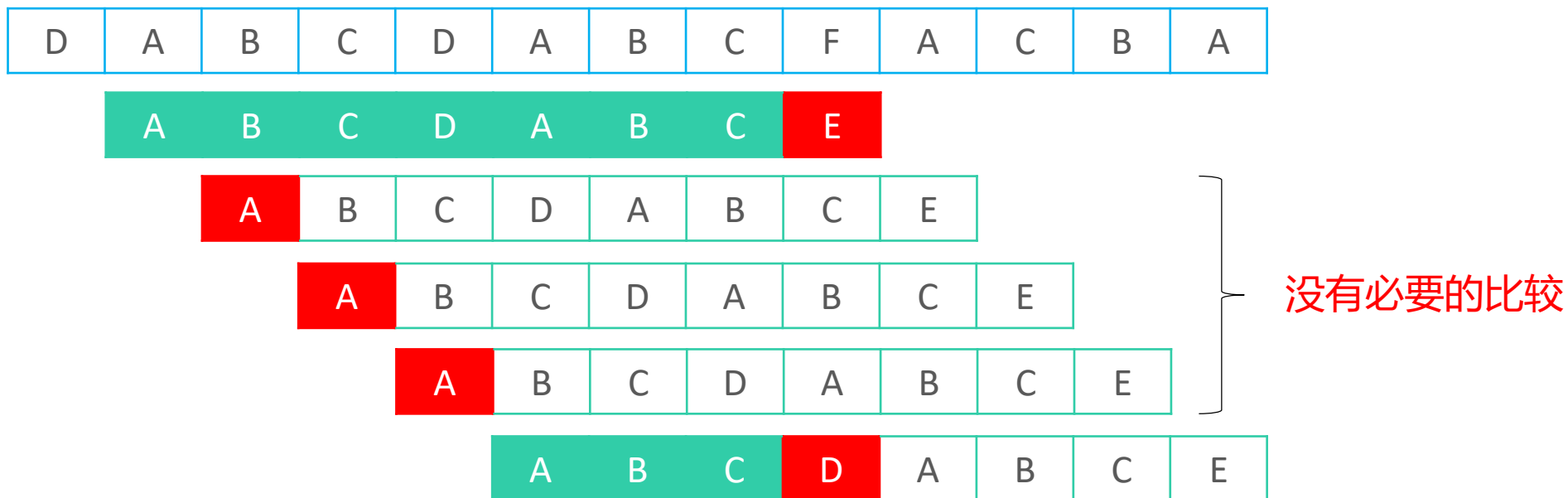
James Hiram **M**orris



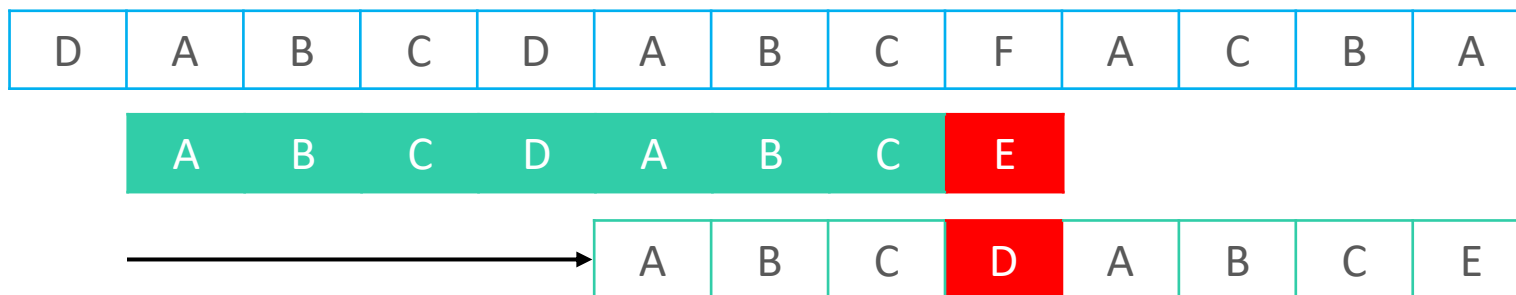
Vaughan **P**ratt

# 蛮力 vs KMP

蛮力算法



KMP算法



■ 对比蛮力算法，KMP的精妙之处：充分利用了此前比较过的内容，可以很聪明地跳过一些不必要的比较位置



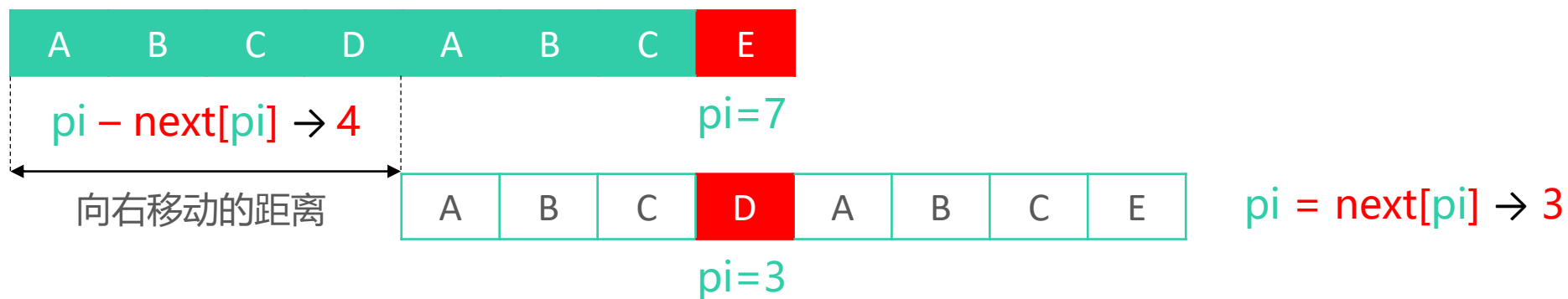
# KMP – next表的使用

- KMP 会预先根据模式串的内容生成一张 next 表（一般是个数组）

模式串 "ABCDABCE" 的 next 表																
模式串字符	A		B		C		D		A		B		C		E	
索引	0		1		2		3		4		5		6		7	
元素	-1	0	0	0	0	0	0	1	2	3						

ti=8

D	A	B	C	D	A	B	C	F	A	C	B	A
---	---	---	---	---	---	---	---	---	---	---	---	---



# KMP – next表的使用

模式串 "ABCDABCE" 的 next 表								
模式串字符	A	B	C	D	A	B	C	E
索引	0	1	2	3	4	5	6	7
元素	-1	0	0	0	0	1	2	3

ti=5

D	A	B	C	C	B	B	C	F	A	C	B	A
---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	C	D	A	B	C	E
---	---	---	---	---	---	---	---

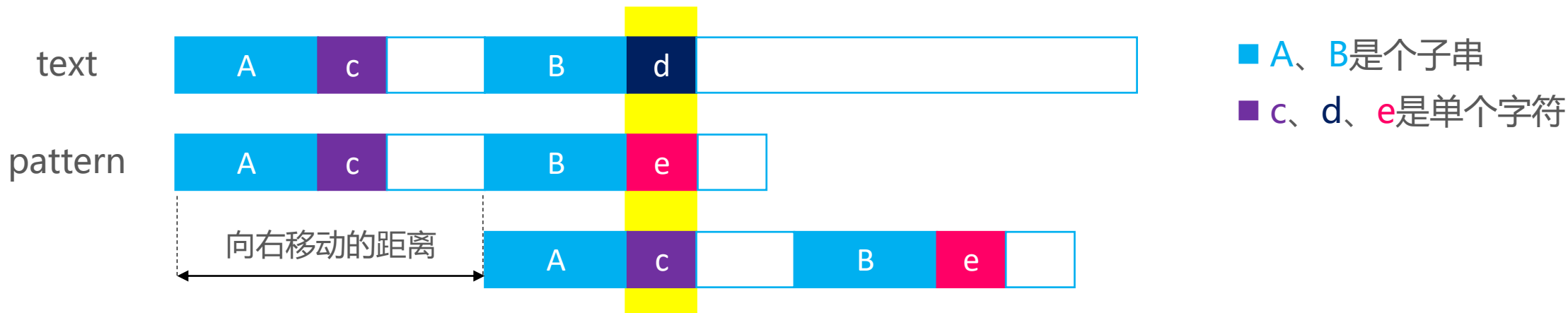
pi=3

A	B	C	D	A	B	C	E
---	---	---	---	---	---	---	---

pi=0

pi = next[pi] → 0

# KMP – 核心原理



■ 当 d、e 失配时，如果希望 pattern 能够一次性向右移动一大段距离，然后直接比较 d、c 字符

□ 前提条件是 A 必须等于 B

■ 所以 KMP 必须在失配字符 e 左边的子串中找出符合条件的 A、B，从而得知向右移动的距离

■ 向右移动的距离：e 左边子串的长度 - A 的长度，等价于：e 的索引 - c 的索引

■ 且 c 的索引 == next[e 的索引]，所以向右移动的距离：e 的索引 - next[e 的索引]

■ 总结

□ 如果在 pi 位置失配，向右移动的距离是 pi - next[pi]，所以 next[pi] 越小，移动距离越大

□ next[pi] 是 pi 左边子串的真前缀后缀的最大公共子串长度

# KMP – 真前缀后缀的最大公共子串长度

模式串	真前缀	真后缀	最大公共子串长度
ABCDABCE	A, AB, ABC, ABCD, ABCDA, ABCDAB, ABCDABC	BCDABCE, CDABCE, DABCE, ABCE, BCE, CE, E	0
ABCDABC	A, AB, ABC, ABCD, ABCDA, ABCDAB	BCDABC, CDABC, DABC, ABC, BC, C	3
ABCDAB	A, AB, ABC, ABCD, ABCDA	BCDAB, CDAB, DAB, AB, B	2
ABCD A	A, AB, ABC, ABCD	BCDA, CDA, DA, A	1
ABCD	A, AB, ABC	BCD, CD, D	0
ABC	A, AB	BC, C	0
AB	A	B	0
A			0

模式串字符	A	B	C	D	A	B	C	E
最大公共子串长度	0	0	0	0	1	2	3	0

# KMP – 得到next表

模式串字符	A	B	C	D	A	B	C	E
最大公共子串长度	0	0	0	0	1	2	3	0

■ 将最大公共子串长度都向后移动 1 位，首字符设置为 负1，就得到了 next 表

模式串 "ABCDABCE" 的 next 表								
模式串字符	A	B	C	D	A	B	C	E
索引	0	1	2	3	4	5	6	7
元素	-1	0	0	0	0	1	2	3

# KMP – 负1的精妙之处

ti=2

D	A	C	C	D	A	B	C	F	A	C	B	A
---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	C	D	A	B	C	E
---	---	---	---	---	---	---	---

pi=0

ti=3

D	A	C	C	D	A	B	C	F	A	C	B	A
---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	C	D	A	B	C	E
---	---	---	---	---	---	---	---

pi=0

ti=2

D	A	C	C	D	A	B	C	F	A	C	B	A
---	---	---	---	---	---	---	---	---	---	---	---	---

*	A	B	C	D	A	B	C	E
---	---	---	---	---	---	---	---	---

pi=-1

pi = next[pi] → -1

pi++ → 0

ti++ → 3

■ 相当于在负1位置有个假想的通配字符（哨兵）

□ 匹配成功后 ti++、pi++

# KMP – 主算法实现

```
public static int indexOf(String text, String pattern) {  
    if (text == null || pattern == null) return -1;  
    int plen = pattern.length();  
    int tlen = text.length();  
    if (tlen == 0 || plen == 0 || tlen < plen) return -1;  
    int[] next = next(pattern);  
    int pi = 0, ti = 0;  
    int tmax = tlen - plen;  
    while (pi < plen && ti - pi <= tmax) {  
        if (pi < 0 || text.charAt(ti) == pattern.charAt(pi)) {  
            ti++;  
            pi++;  
        } else {  
            pi = next[pi];  
        }  
    }  
    return pi == plen ? ti - pi : -1;  
}
```

# KMP – 为什么是“最大”公共子串长度？

■ 假设文本串是AAAAABCDEF，模式串是AAAAB

模式串	真前缀	真后缀	公共子串长度
AAAA	A, AA, AAA	A, AA, AAA	1, 2, 3
AAA	A, AA	A, AA	1, 2
AA	A	A	1

ti=4

A	A	A	A	A	B	C	D	E	F
---	---	---	---	---	---	---	---	---	---

A	A	A	A	B
---	---	---	---	---

pi=4

A	A	A	A	B
---	---	---	---	---

pi=3

A	A	A	A	B
---	---	---	---	---

pi=1

■ 应该将1、2、3中的哪个值赋值给 pi 是正确的？

■ 将 3 赋值给 pi

□ 向右移动了 1 个字符单位，最后成功匹配

■ 将 1 赋值给 pi

□ 向右移动了 3 个字符单位，错过了成功匹配的机会

■ 公共子串长度越小，向右移动的距离越大，越不安全

■ 公共子串长度越大，向右移动的距离越小，越安全



# KMP – next表的构造思路



■ 已知  $\text{next}[i] == n$

① 如果  $\text{pattern.charAt}(i) == \text{pattern.charAt}(n)$

□ 那么  $\text{next}[i + 1] == n + 1$

② 如果  $\text{pattern.charAt}(i) \neq \text{pattern.charAt}(n)$

□ 已知  $\text{next}[n] == k$

□ 如果  $\text{pattern.charAt}(i) == \text{pattern.charAt}(k)$

✓ 那么  $\text{next}[i + 1] == k + 1$

□ 如果  $\text{pattern.charAt}(i) \neq \text{pattern.charAt}(k)$

✓ 将  $k$  代入  $n$ ，重复执行 ②

# KMP – next表的代码实现

```
public static int[] next(String pattern) {  
    int len = pattern.length();  
    int[] next = new int[len];  
    int i = 0;  
    int n = next[i] = -1;  
    int imax = len - 1;  
    while (i < imax) {  
        if (n < 0 || pattern.charAt(i) == pattern.charAt(n)) {  
            next[++i] = ++n;  
        } else {  
            n = next[n];  
        }  
    }  
    return next;  
}
```

# KMP – next表的不足之处

- 假设文本串是 AAABAAAAB，模式串是 AAAAB

模式串 "AAAAB" 的 next 表					
模式串字符	A		A		B
索引	0		1		4
元素	-1	0	1	2	3

A	A	A	B	A	A	A	A	B
---	---	---	---	---	---	---	---	---

A	A	A	A	B
---	---	---	---	---

A	A	A	A	B
---	---	---	---	---

A	A	A	A	B
---	---	---	---	---

A	A	A	A	B
---	---	---	---	---

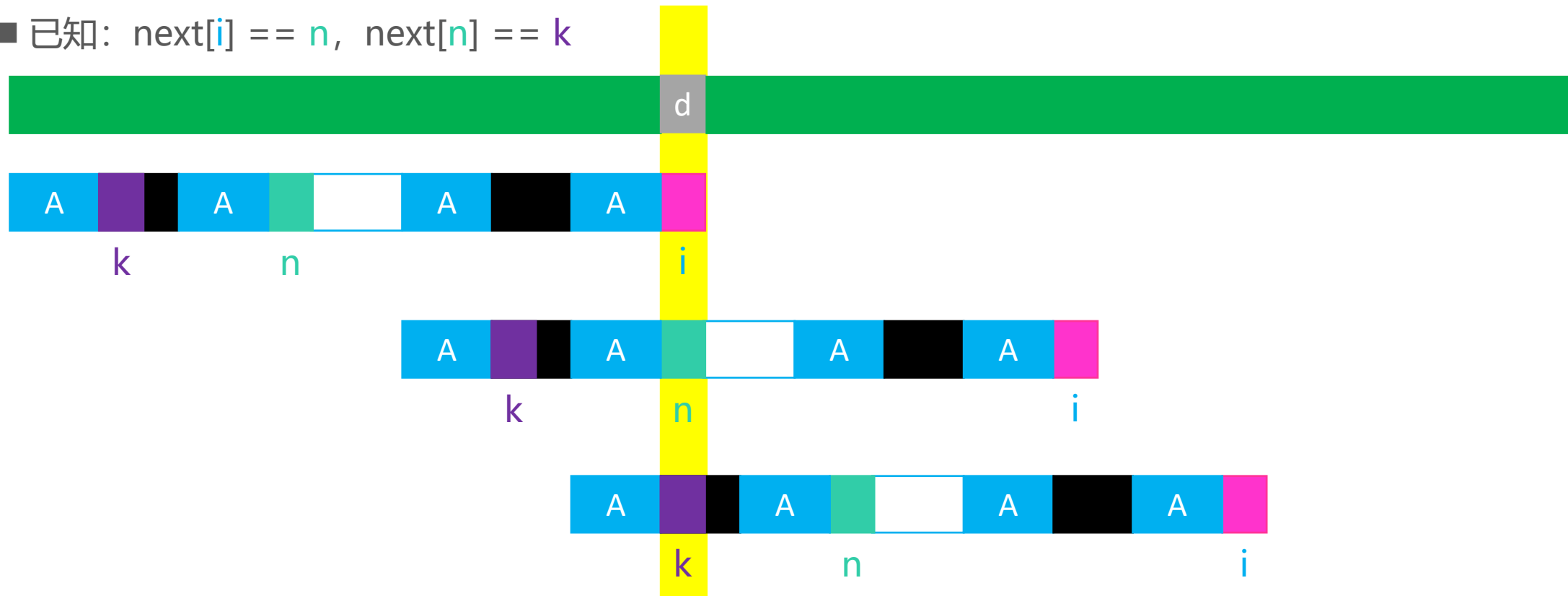
A	A	A	A	B
---	---	---	---	---

没有必要的比较

- 在这种情况下，KMP显得比较笨拙

# KMP – next表的优化思路

■ 已知:  $\text{next}[i] == n$ ,  $\text{next}[n] == k$



- 如果  $\text{pattern}[i] \neq d$ , 就让模式串滑动到  $\text{next}[i]$  (也就是n) 位置跟 d 进行比较
- 如果  $\text{pattern}[n] \neq d$ , 就让模式串滑动到  $\text{next}[n]$  (也就是k) 位置跟 d 进行比较
- 如果  $\text{pattern}[i] == \text{pattern}[n]$ , 那么当 i 位置失配时, 模式串最终必然会滑到 k 位置跟 d 进行比较
- 所以  $\text{next}[i]$  直接存储  $\text{next}[n]$  (也就是k) 即可

# KMP – next表的优化实现

```
public static int[] next(String pattern) {
    int len = pattern.length();
    int[] next = new int[len];
    int i = 0;
    int n = next[i] = -1;
    int imax = len - 1;
    while (i < imax) {
        if (n < 0 || pattern.charAt(i) == pattern.charAt(n)) {
            i++;
            n++;
            if (pattern.charAt(i) == pattern.charAt(n)) {
                next[i] = next[n];
            } else {
                next[i] = n;
            }
        } else {
            n = next[n];
        }
    }
    return next;
}
```

# KMP – next表的优化效果

模式串 "AAAAB" 的 next 表					
模式串字符	A	A	A	A	B
索引	0	1	2	3	4
优化前	-1	0	1	2	3
优化后	-1	-1	-1	-1	3

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

# KMP – 性能分析

## ■ KMP 主逻辑

□ 最好时间复杂度:  $O(m)$

□ 最坏时间复杂度:  $O(n)$ , 不超过  $O(2n)$

## ■ next 表的构造过程跟 KMP 主体逻辑类似

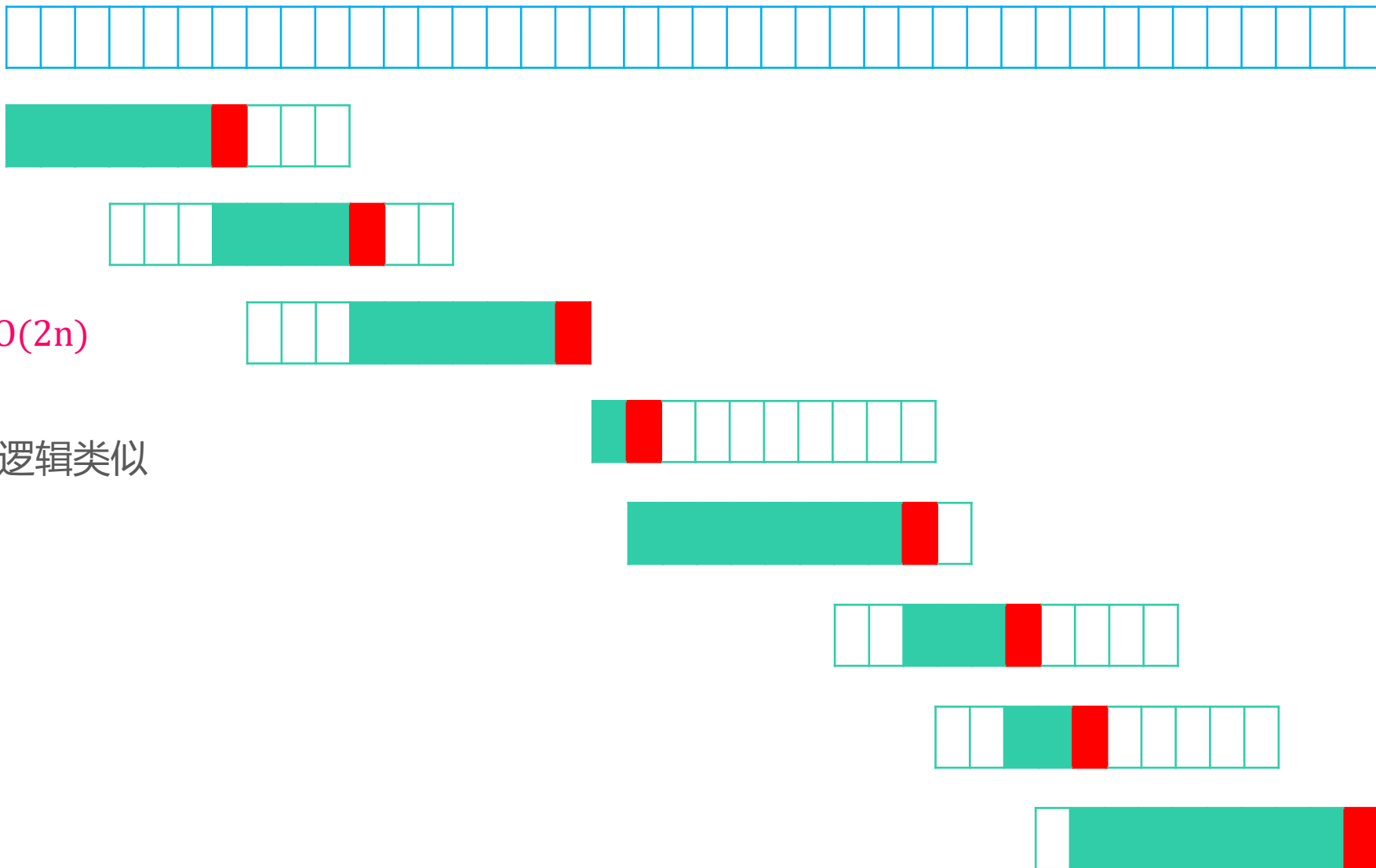
□ 时间复杂度:  $O(m)$

## ■ KMP 整体

□ 最好时间复杂度:  $O(m)$

□ 最坏时间复杂度:  $O(n + m)$

□ 空间复杂度:  $O(m)$



# 蛮力 vs KMP

- 蛮力算法为何低效?

- 当字符失配时

- 蛮力算法

- ✓  $t_i$  回溯到左边位置

- ✓  $p_i$  回溯到 0

- KMP 算法

- ✓  $t_i$  不必回溯

- ✓  $p_i$  不一定要回溯到 0