数据结构实验（5）

串

目录

- 串
  - 串与匹配问题
    - KMP
    - Rabin-Karp

- 串
  - 串与匹配问题
    - 字符串匹配只是字符串检索问题的一种
    - 字符串检索：
      - 判断字符串 A 中是否包含字符串 B；
      - 在一堆字符串中，找到某个字符串A出现的次数；
      - 在一堆字符串中，找到包含字符串A的字符串的个数；
      - 在一堆字符串中，找到与字符串A编辑距离不超过x 的字符串的个数；
      - …

目录

- 串
  - 串与匹配问题
    - KMP
    - Rabin-karp

- 串与匹配
  - 匹配问题

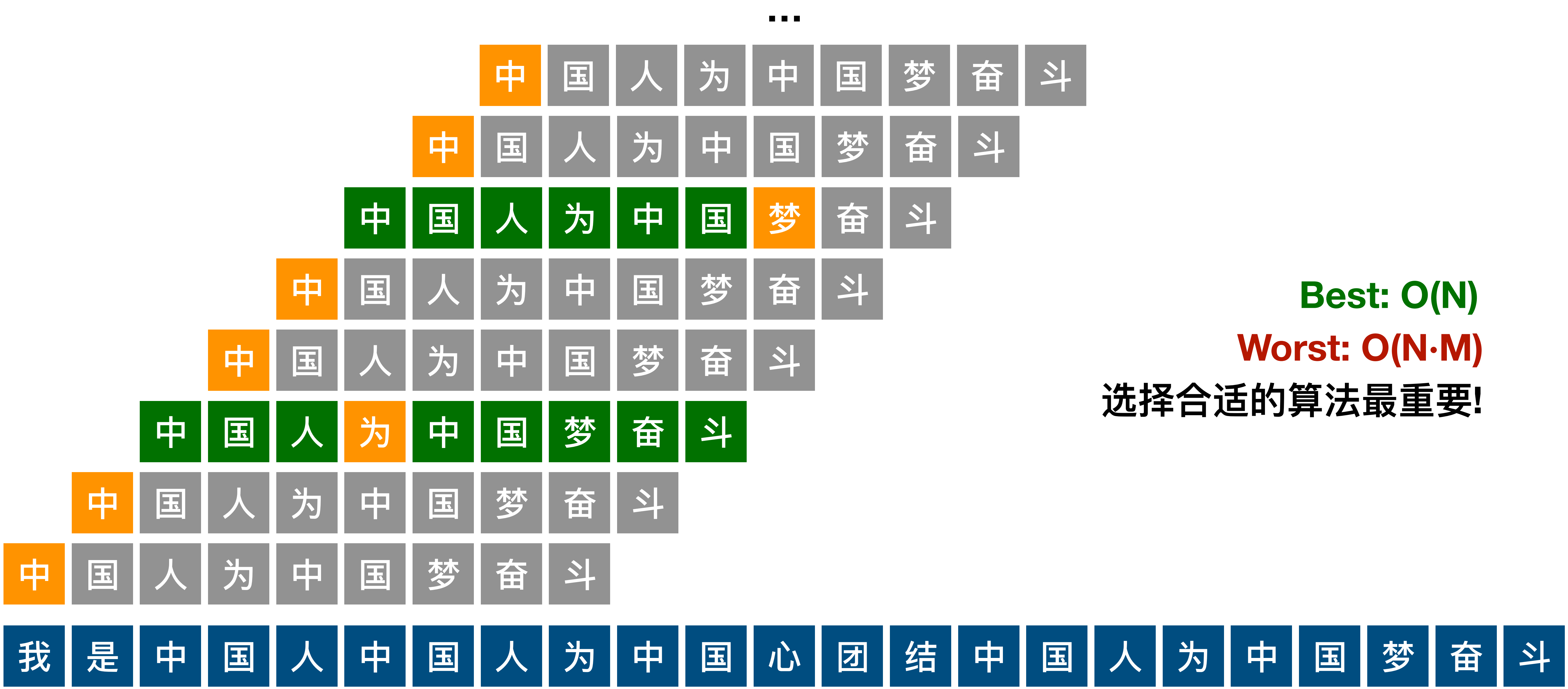
Pattern:

中	国	人	为	中	国	梦	奋	斗

Text:

我	是	中	国	人	中	国	人	为	中	国	心	团	结	中	国	人	为	中	国	梦	奋	斗

- 串与匹配
  - 匹配问题
    - 枚举法



Best: O(N)

Worst: O(N·M)

选择合适的算法最重要!

- 穿插话题1：SIMD - x86 高级指令集
    - 什么是 CPU 指令集?
    - SIMD 的演进：MMX、SSE、SSE2、SSSE3、SSE4、AVX、AVX2、AVX512 ...
        - SIMD 指令集的意义?
    - SSE / AVX 指令举例：
        - _mm256_add_ps / _mm256_mul_ps / _mm256_cmp_ps

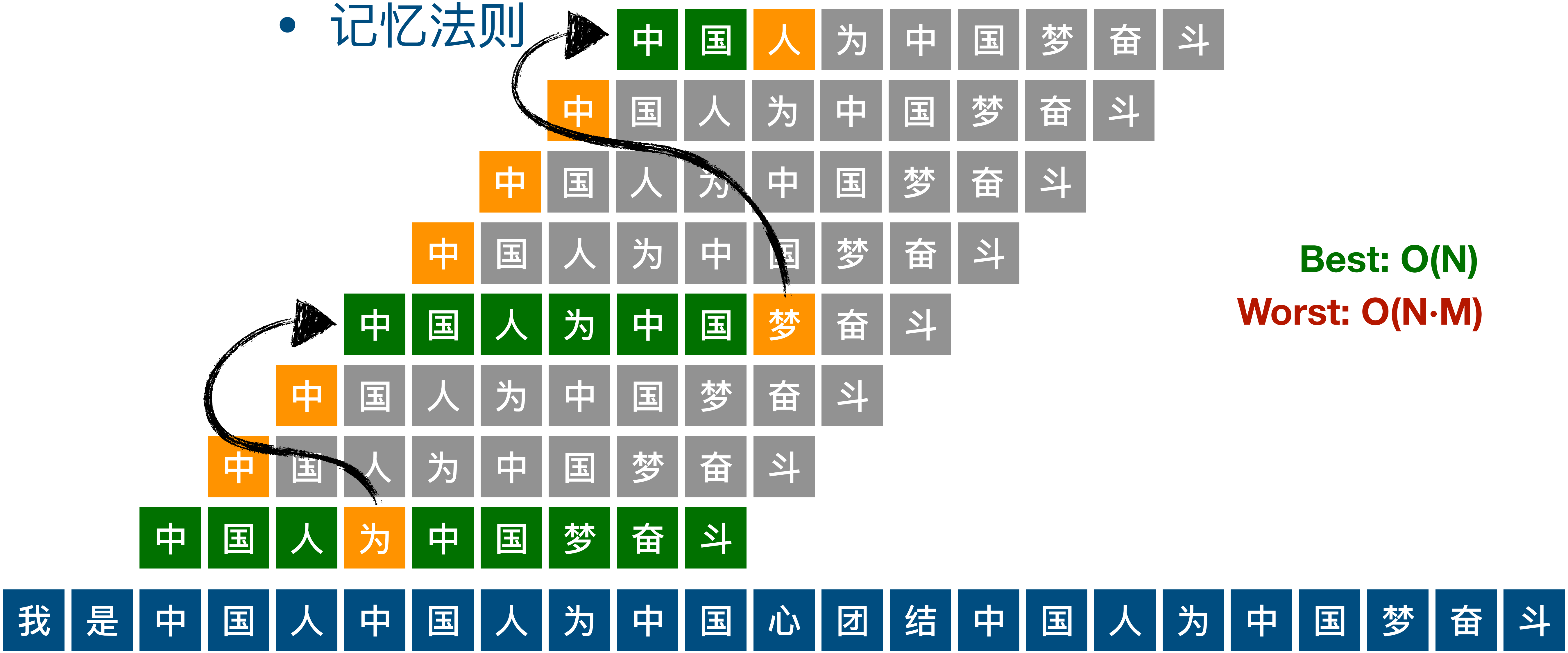
```
#include <x86intrin.h>

int main(int, char**) {
    __m256 a = _mm256_set_ps(1, 2, 3, 4, 5, 6, 7, 8);
    __m256 b = _mm256_set_ps(2, 2, 2, 2, 2, 2, 2, 2);
    __m256 c = _mm256_add_ps(a, b);                    // c = a + b
    __m256 d = _mm256_mul_ps(a, b);                    // d = a * b
    return 0;
}
```

- 串与匹配
  - 匹配问题
    - KMP
      - 记忆法则

- 串与匹配
  - 匹配问题
    - KMP
      - 记忆法则

不断局部匹配导致算法低效，分析枚举法低效的原因，改进！

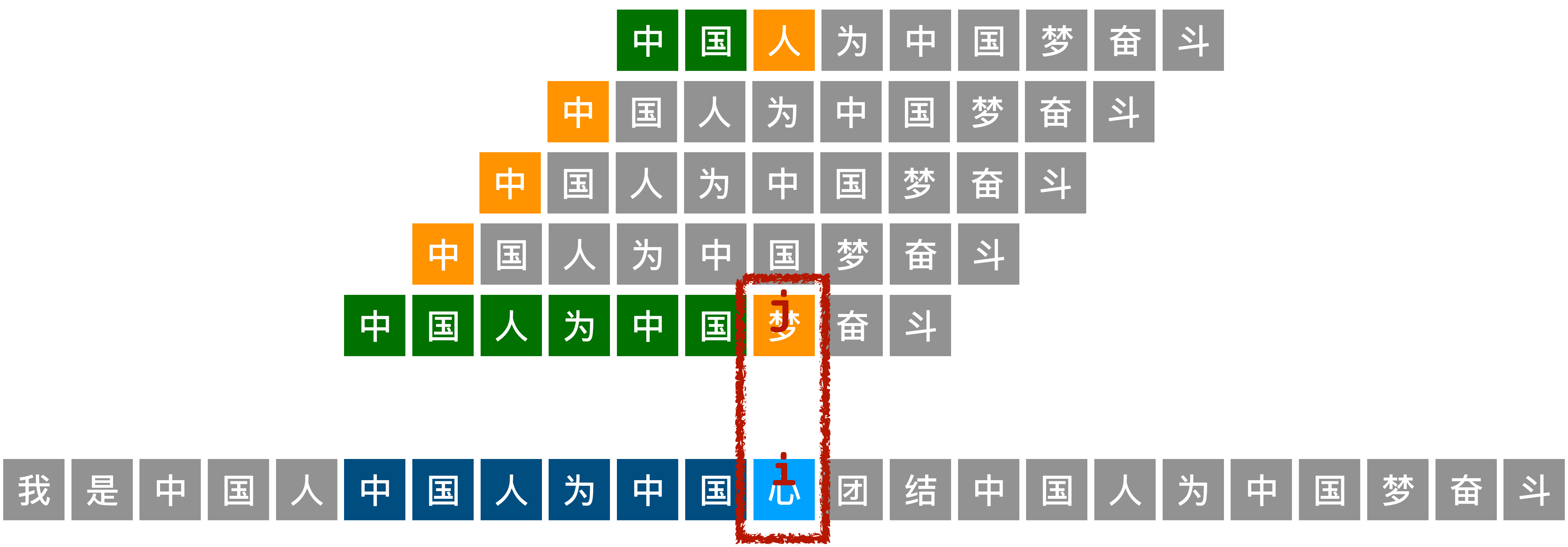


Best: O(N)

Worst: O(N·M)

1) 枚举法，遇到 `Text[i] != Pattern[j]`

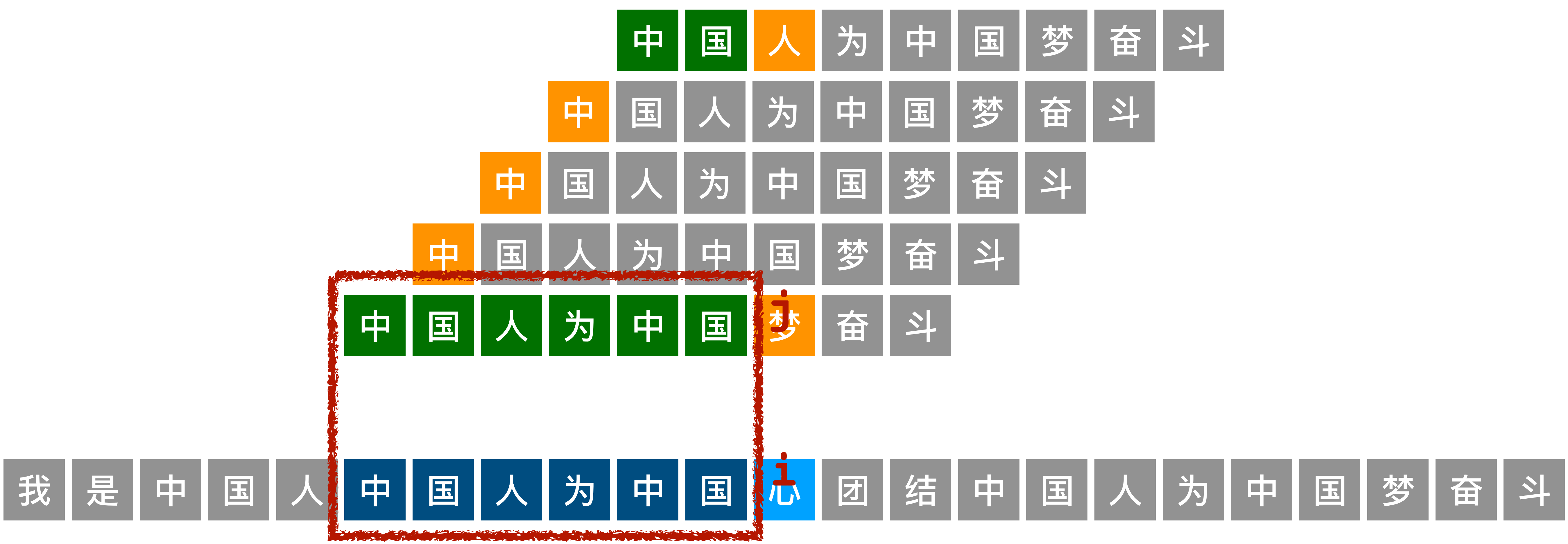
- 串与匹配
  - 匹配问题
    - KMP
      - 记忆法则



1) 枚举法，遇到 `Text[i] != Pattern[j]`

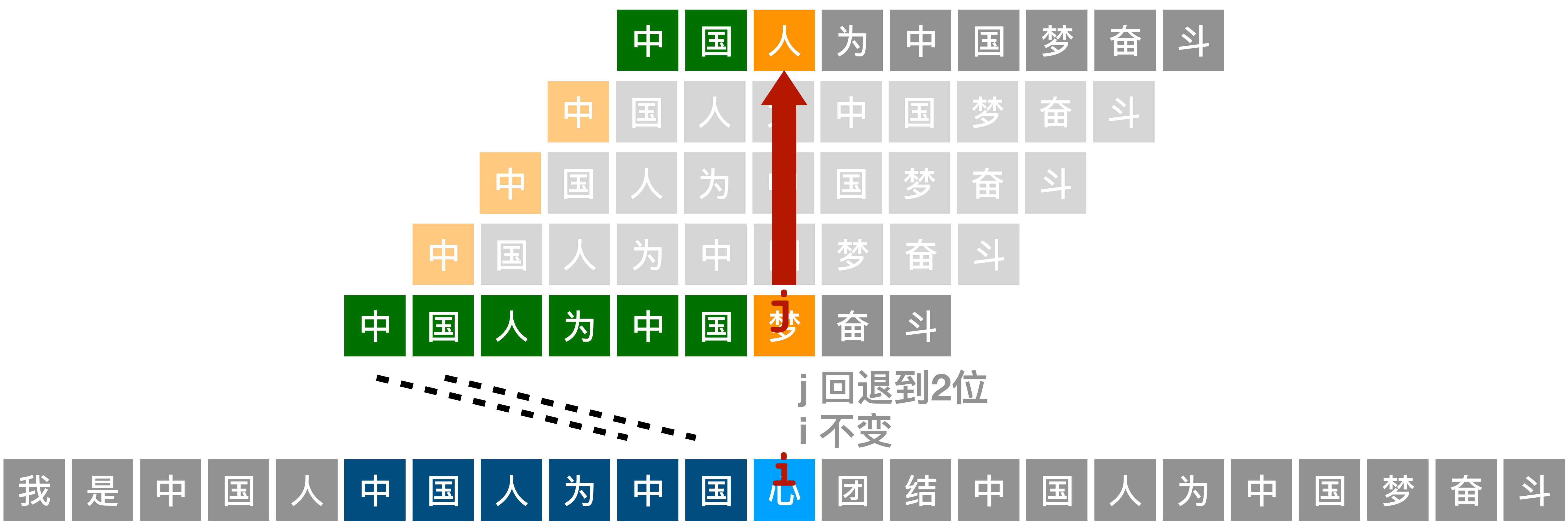
2) 此时必然 `Text[i-j, i) == Pattern[0, j)`

- 串与匹配
  - 匹配问题
    - KMP
      - 记忆法则



- 串与匹配
    - 匹配问题
        - KMP
            - 记忆法则

1) 枚举法，遇到 `Text[i] != Pattern[j]`
2) 此时必然 `Text[i-j, i) == Pattern[0, j)`
3) `Text[i-j, i)` 或 `Pattern[0, j)` 是已经枚举过的串，我们一定知道一些信息，例如 `Pattern[0, j)` 中一些字符与 `Text[i-j+1, i)` 中部分串匹配，这些部分匹配的串，能否帮助我们跳过中间三次失败的匹配，直接对齐 “中国”-“中国” 开始匹配?



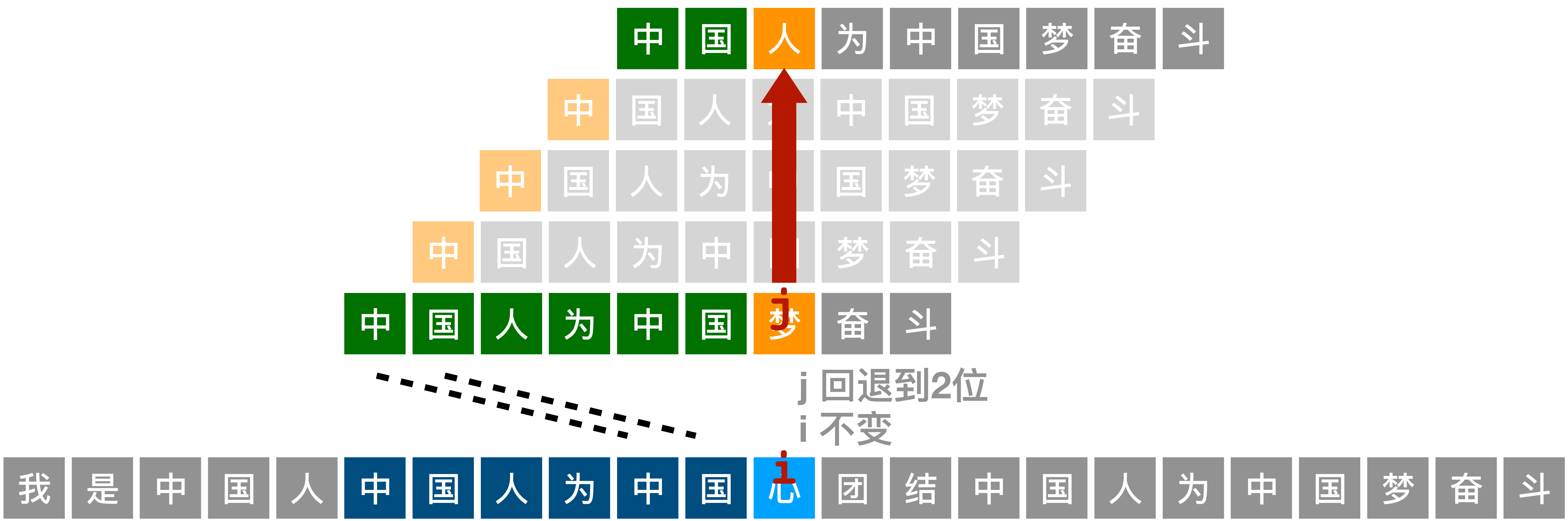
- 串与匹配
  - 匹配问题
    - KMP
      - 记忆法则

1) 枚举法，遇到 `Text[i] != Pattern[j]`

2) 此时必然 `Text[i-j, i) == Pattern[0, j)`

3) `Text[i-j, i)` 或 `Pattern[0, j)` 是已经枚举过的串，我们一定知道一些信息，例如 `Pattern[0, j)` 中一些字符与 `Text[i-j+1, i)` 中部分串匹配，这些部分匹配的串，能否帮助我们跳过中间三次失败的匹配，直接对齐 “中国”-“中国” 开始匹配?

4) “中国”是 P[0, j) 的前缀，也是 P[0, j) 的后缀，（也是 T[i-j, i) 的前缀和后缀），是符合同时是 P[0, j) 的前缀和后缀（必须不完全重叠）的最长子串



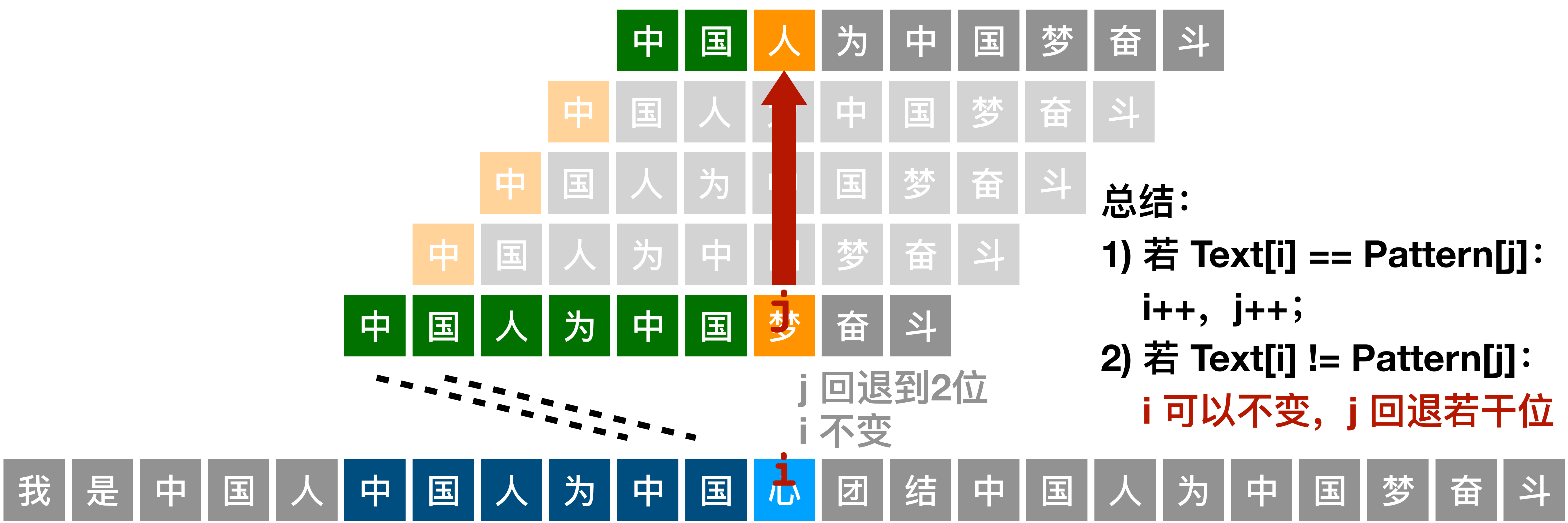
- 串与匹配
  - 匹配问题
    - KMP
      - 记忆法则

1) 枚举法，遇到 `Text[i] != Pattern[j]`

2) 此时必然 `Text[i-j, i) == Pattern[0, j)`

3) `Text[i-j, i)` 或 `Pattern[0, j)` 是已经枚举过的串，我们一定知道一些信息，例如 `Pattern[0, j)` 中一些字符与 `Text[i-j+1, i)` 中部分串匹配，这些部分匹配的串，能否帮助我们跳过中间三次失败的匹配，直接对齐 “中国”-“中国” 开始匹配?

4) “中国”是 P[0, j) 的前缀，也是 P[0, j) 的后缀，（也是 T[i-j, i) 的前缀和后缀），是符合同时是 P[0, j) 的前缀和后缀（必须不完全重叠）的最长子串



总结：

1) 若 Text[i] == Pattern[j]：
   i++，j++；

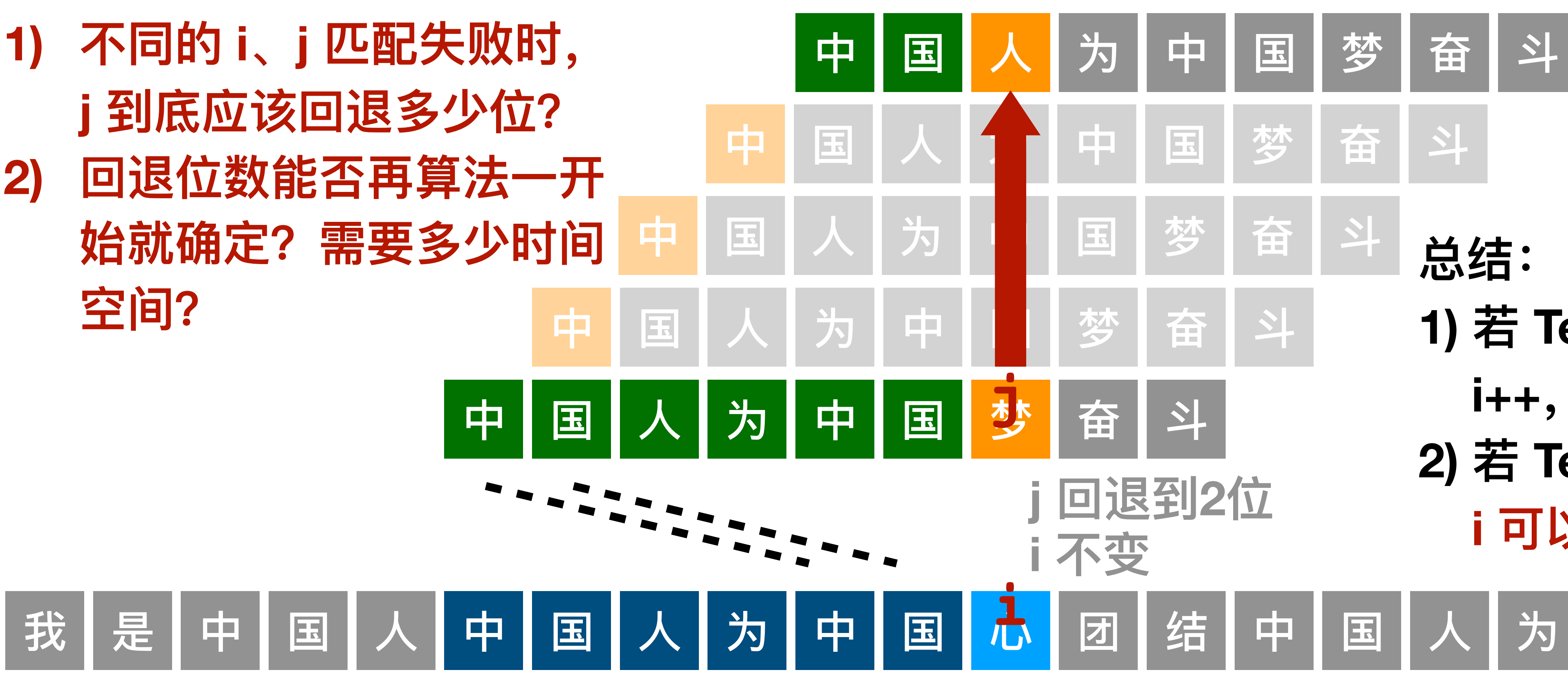
2) 若 Text[i] != Pattern[j]：
   i 可以不变，j 回退若干位

- 串与匹配
  - 匹配问题
    - KMP
      - 记忆法则

1) 枚举法，遇到 `Text[i] != Pattern[j]`
2) 此时必然 `Text[i-j, i) == Pattern[0, j)`
3) `Text[i-j, i)` 或 `Pattern[0, j)` 是已经枚举过的串，我们一定知道一些信息，例如 `Pattern[0, j)` 中一些字符与 `Text[i-j+1, i)` 中部分串匹配，这些部分匹配的串，能否帮助我们跳过中间三次失败的匹配，直接对齐 “中国”-“中国” 开始匹配?
4) “中国”是 P[0, j) 的前缀，也是 P[0, j) 的后缀，（也是 T[i-j, i) 的前缀和后缀），是符合同时是 P[0, j) 的前缀和后缀（必须不完全重叠）的最长子串

问题：

1) 不同的 i、j 匹配失败时，j 到底应该回退多少位?
2) 回退位数能否再算法一开始就确定？需要多少时间空间?



总结：

1) 若 Text[i] == Pattern[j]：
   i++，j++；
2) 若 Text[i] != Pattern[j]：
   i 可以不变，j 回退若干位

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

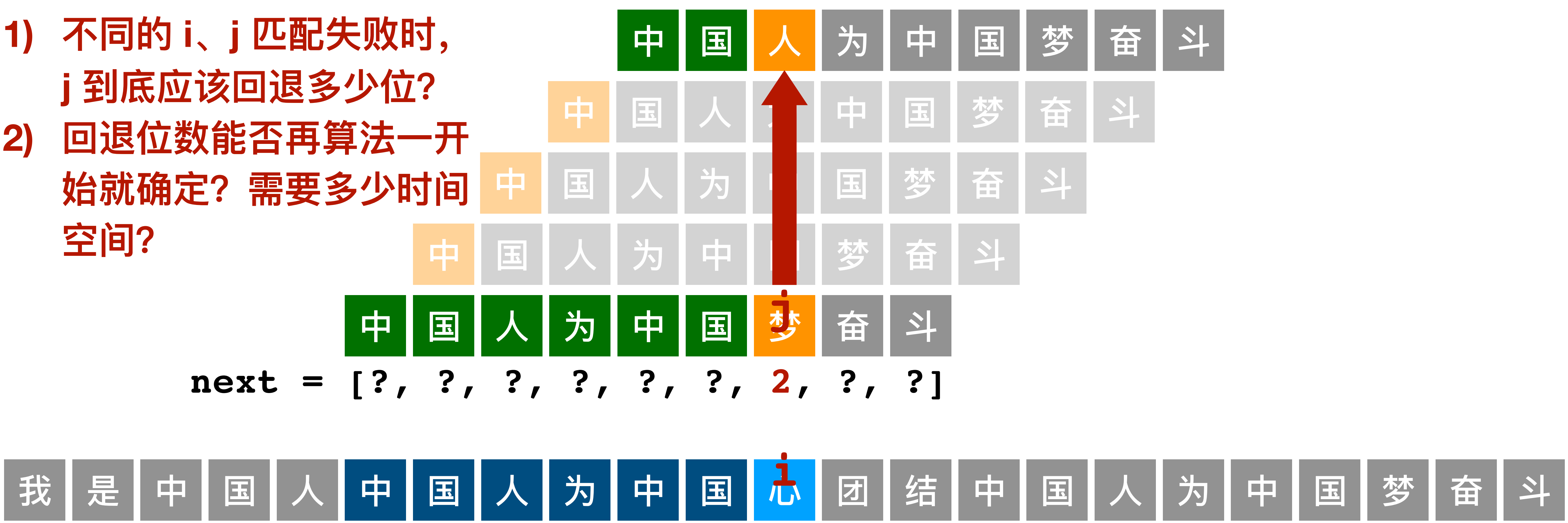
- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；
2) 那么，j 回退的位置不仅可以事先确定，还可以只根据 Pattern 就能确定，即：
   T[i] != P[j] 时，j 回退位数仅靠 P[0, j) 即可确定，与 T 无关
3) 构造查询表 next[0, m)，若一旦 P[j] 匹配失败（无论 T 匹配到哪里）：
   i. i 不变，j 回退到 next[j] 的位置
   ii. 继续用 P[j] 与 T[i] 比对

问题：

1) 不同的 i、j 匹配失败时，j 到底应该回退多少位？
2) 回退位数能否再算法一开始就确定？需要多少时间空间？



- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

中	国	人	中	国	人	中	国	中	国	好	多	人

-1

中	国	人	中	国	人	中	国	中	国	好	多	人

j=0 时， Pattern[0, j) 为空串，next[0] 设为 -1，这是一个习惯上的约定值，没有特殊含义

一个复杂一些的例子

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

中	国	人	中	国	人	中	国	中	国	好	多	人

-1	0	0	0												
中	国	人	中	国	人	中	国	中	国	好	多	人			
			中	国	人	中	国	人	中	国	中	国	好	多	人

j=1、2、3 时，Pattern[0, j) 为 “中”、“中国”、“中国人”，没有前缀、后缀的公共子串，next 均 设为 0

一个复杂一些的例子

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

中	国	人	中	国	人	中	国	中	国	好	多	人

-1	0	0	0	1												
中	国	人	中	国	人	中	国	中	国	好	多	人				
			中	国	人	中	国	人	中	国	中	国	好	多	人	

j=4 时，Pattern[0, j) 为 “中国人中”，最长前缀、后缀的公共子串为 "中“，next 设为 1

一个复杂一些的例子

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

中	国	人	中	国	人	中	国	中	国	好	多	人

-1	0	0	0	1	2											
中	国	人	中	国	人	中	国	中	国	好	多	人				
			中	国	人	中	国	人	中	国	中	国	好	多	人	

j=5 时，Pattern[0, j) 为 “中国人中国”，最长前缀、后缀的公共子串为 "中国“，next 设为 2

一个复杂一些的例子

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；
   ——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

中	国	人	中	国	人	中	国	中	国	好	多	人

-1	0	0	0	1	2	3									
中	国	人	中	国	人	中	国	中	国	好	多	人			
			中	国	人	中	国	人	中	国	中	国	好	多	人

j=6 时，Pattern[0, j) 为 “中国人中国人”，最长前缀、后缀的公共子串为 "中国人“，next 设为 3

一个复杂一些的例子

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

中	国	人	中	国	人	中	国	中	国	好	多	人			
-1	0	0	0	1	2	3	4								
中	国	人	中	国	人	中	国	中	国	好	多	人			
			中	国	人	中	国	人	中	国	中	国	好	多	人

j=7 时，Pattern[0, j) 为 “中国人中国人中”，
最长前缀、后缀的公共子串为 "中国人中“，
next 设为 4

一个复杂一些的例子

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

中	国	人	中	国	人	中	国	中	国	好	多	人

-1	0	0	0	1	2	3	4	5								
中	国	人	中	国	人	中	国	中	国	好	多	人				
			中	国	人	中	国	人	中	国	中	国	好	多	人	

j=8 时，Pattern[0, j) 为 “中国人中国人中国”，最长前缀、后缀的公共子串为 "中国人中国“，next 设为 5

一个复杂一些的例子

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：求 Pattern[0, j) 的最长前缀、后缀公共子串

中	国	人	中	国	人	中	国	中	国	好	多	人

-1	0	0	0	1	2	3	4	5	1			
中	国	人	中	国	人	中	国	中	国	好	多	人

中 国 人 中 国 人 中 国 中 国 好 多 人

j=9 时，Pattern[0, j) 为 “中国人中国人中国中”，最长前缀、后缀的公共子串为 "中“，next 设为 1

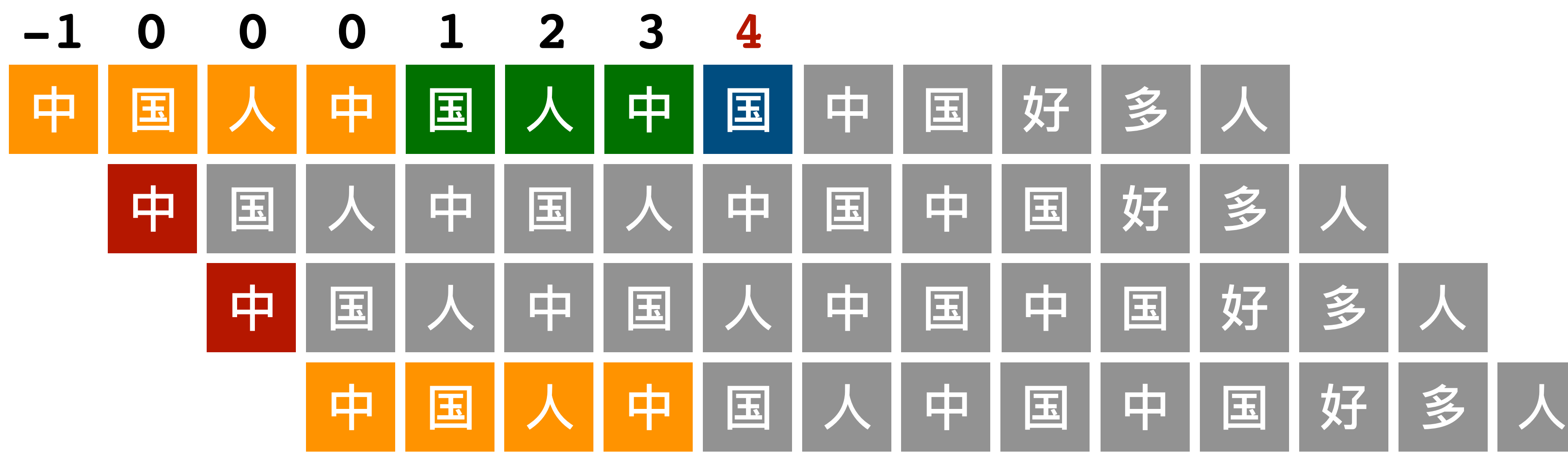
一个复杂一些的例子

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

——我们根据物理含义，得到第一个求 next 表的算法：**求 Pattern[0, j) 的最长前缀、后缀公共子串**




```cpp
// 枚举法，生成 next 表
std::vector<int> build_next(const std::string& P) {
    std::vector<int> next(P.length());
    next[0] = -1;
    for (int i = 1; i < P.length(); ++i) { // 枚举 next 表
        for (int t = i-1; t >=0; --t) {     // 从长到短枚举前缀、后缀字符串
            int j = 0;
            for (; j < t; ++j) {
                if (P[j] != P[i-t+j])       // 匹配字符串 前缀P[0, t) 和后缀P[i-t, i)
                    break;
            }
            if (j >= t) {                    // 若字符串匹配成功，则找到最长公共前缀、后缀字符串
                next[i] = t;
                break;
            }
        }
    }
    return std::move(next);
}
```

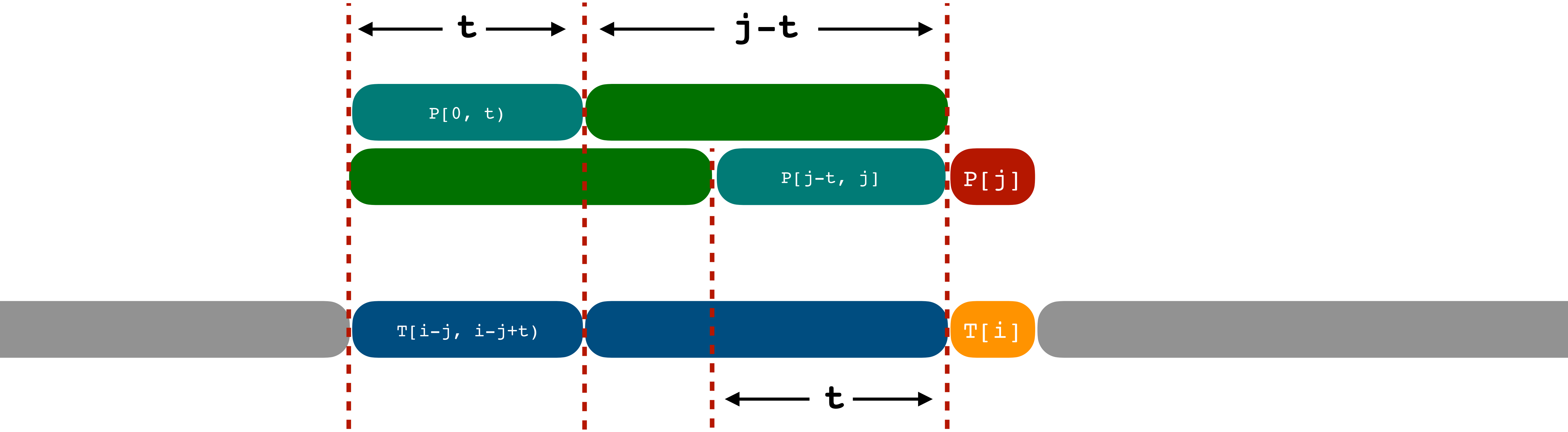
时间复杂度 O(M^3)

- 串与匹配
    - 匹配问题
        - KMP
            - 查询表

总结：

——我们给出 next 的严肃定义（理解前几页后，你并不需要太在意这个定义）

```
1 <= j < length(P),
next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}
```

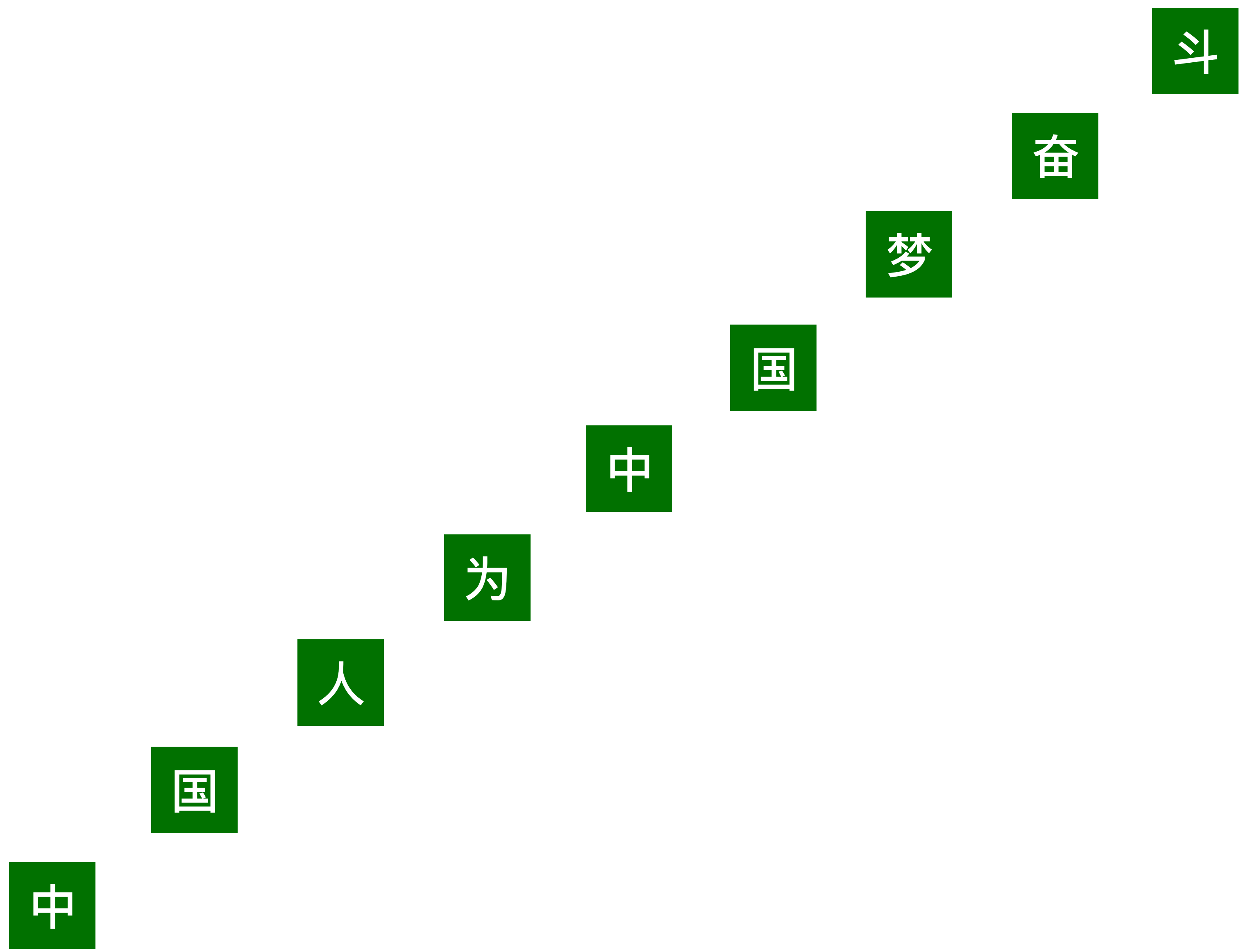


- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

—— 我们已经可以根据物理含义构造出“中国人为中国梦奋斗”这个 Pattern 的 next 表，




```
next = [-1, 0, 0, 0, 0, 1, 2, 0, 0]
```

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

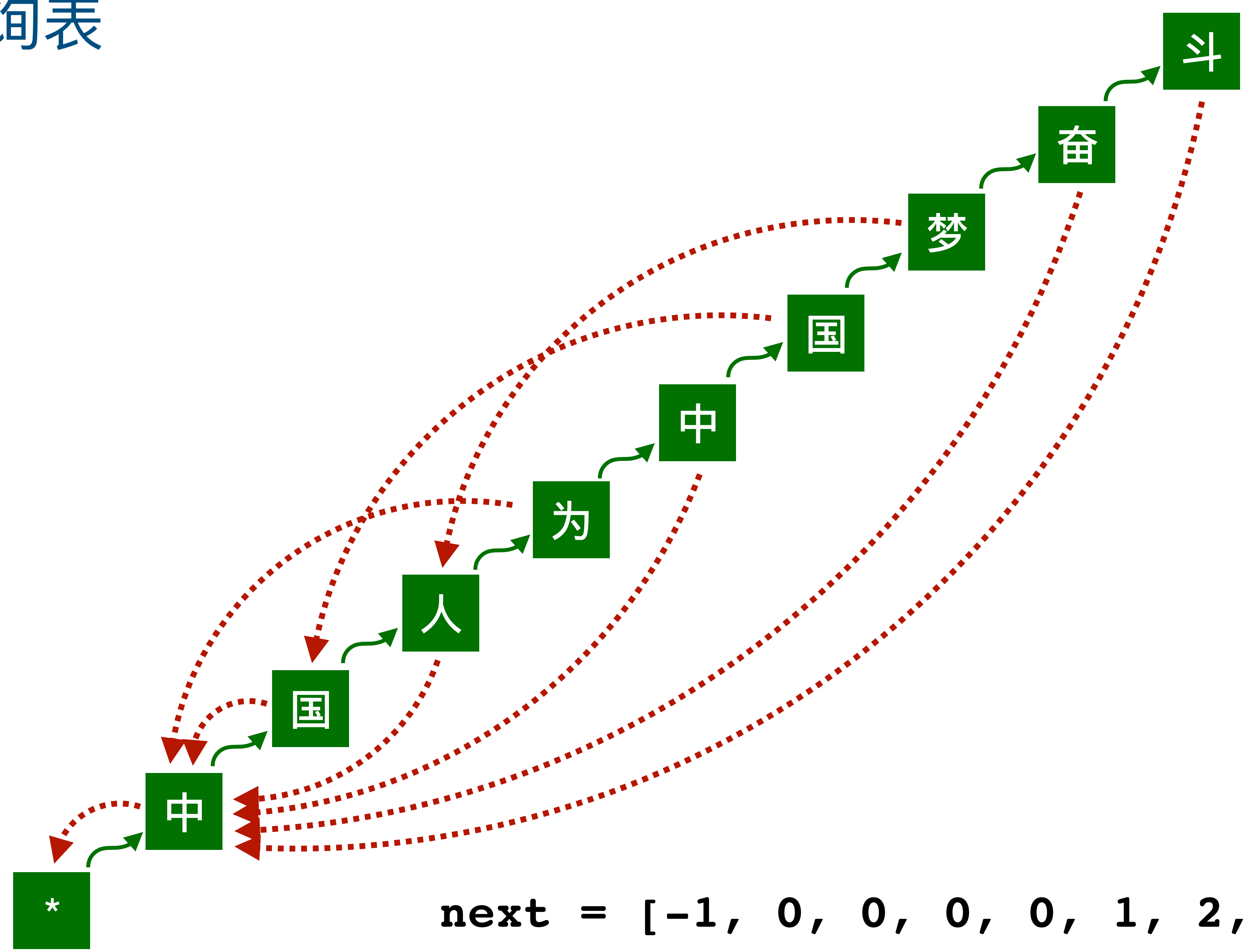
先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

—— 我们已经可以根据物理含义构造出“中国人为中国梦奋斗”这个 Pattern 的 next 表，它本质是一个有限状态自动机（注意 * 这个特殊状态，对应 next 表的 -1）

i. 若 T[i] == P[j]，则 i++，j 沿着绿色线转移（即 j++)

ii. 若 T[i] != P[j]，则 j 沿着红色线转移（即 j=next[j])



`next = [-1, 0, 0, 0, 0, 1, 2, 0, 0]`

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

先说结论：

1) 不同的 i、j 匹配失败时，j 回退到 Pattern[0, j) 最长前缀、后缀公共子串的位置；

—— 我们已经根据物理含义构造出 next 表，它本质是一个有限状态自动机（注意 * 这个特殊状态，对应 next 表的 -1）

i. 若 T[i] == P[j]，则 i++，j 沿着绿色线转移（即 j++)

ii. 若 T[i] != P[j]，则 j 沿着红色线转移（即 j=next[j])

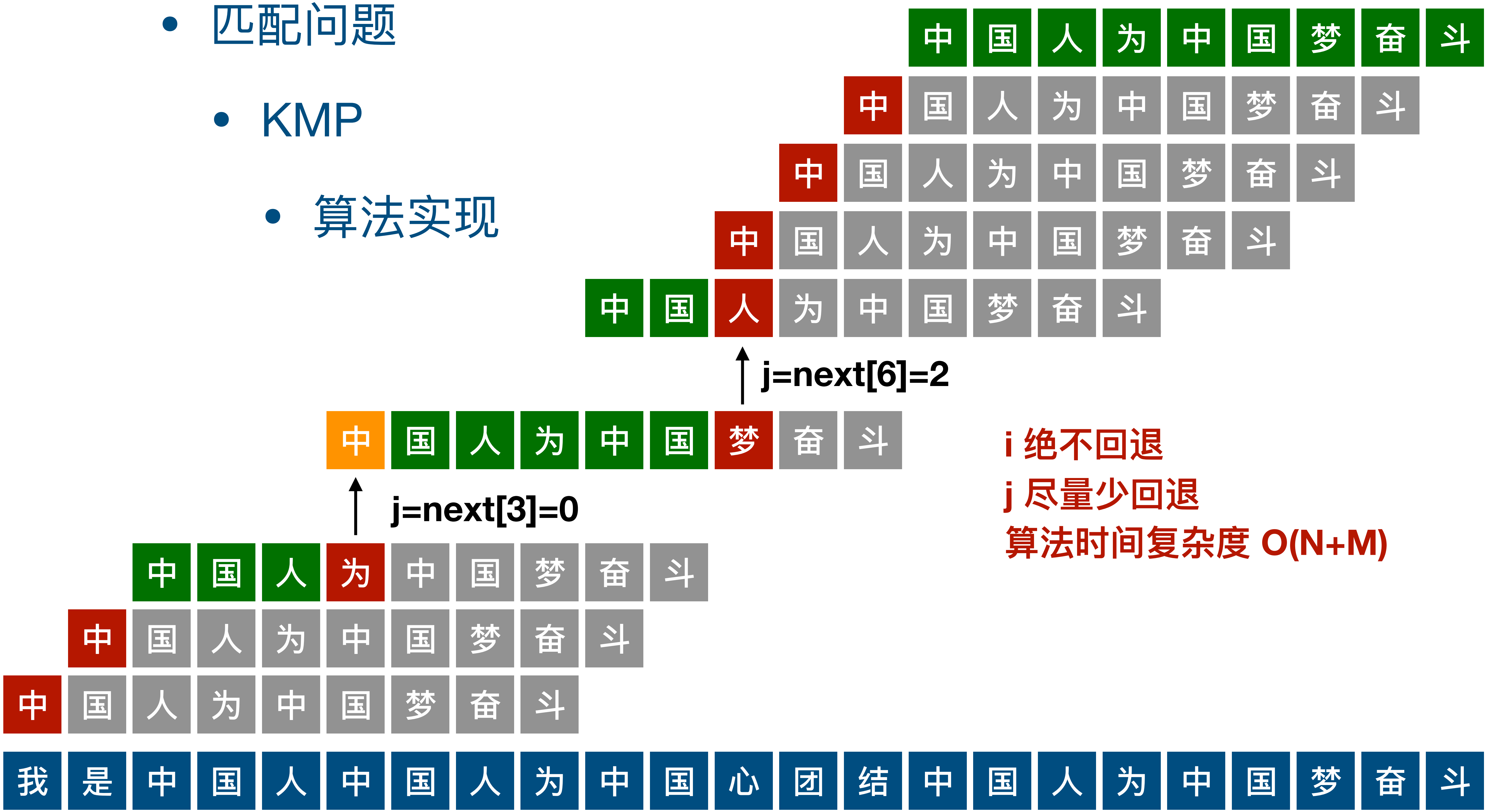
```
// 基于有限状态自动机的 KMP 算法实现，我们可以根据有限状态自动机（next 表）自动生成
// 这个函数代码，并在任意字符串 T 中匹配字符串 “中国人为中国梦奋斗”
int kmp1(const std::string& T) {
    int i = -1;
s_: ++i;
s0: (T[i] == '中') ? goto s_ : if (++i >= T.length()) return -1; // -1 表示匹配失败
s1: (T[i] == '国') ? goto s0 : if (++i >= T.length()) return -1;
s2: (T[i] == '人') ? goto s0 : if (++i >= T.length()) return -1;
s3: (T[i] == '为') ? goto s0 : if (++i >= T.length()) return -1;
s4: (T[i] == '中') ? goto s0 : if (++i >= T.length()) return -1;
s5: (T[i] == '国') ? goto s1 : if (++i >= T.length()) return -1;
s6: (T[i] == '梦') ? goto s2 : if (++i >= T.length()) return -1;
s7: (T[i] == '奋') ? goto s0 : if (++i >= T.length()) return -1;
s8: (T[i] == '斗') ? goto s0 : if (++i >= T.length()) return -1;
    return i-9;  // 返回匹配成功的起点下标，9 为 Pattern 字符串的长度
}
```

next = [-1, 0, 0, 0, 0, 1, 2, 0, 0]

- 串与匹配
  - 匹配问题
    - KMP
      - 算法实现

中 国 人 为 中 国 梦 奋 斗

`next = [-1,0, 0, 0, 0, 1, 2, 0, 0]`



i 绝不回退

j 尽量少回退

算法时间复杂度 O(N+M)

- 串与匹配
  - 匹配问题
    - KMP
      - 算法实现

```
int kmp2(const std::string& P, const std::string& T) {
    std::vector<int> next = build_next(P);
    int i = 0;
    int j = 0;
    while (j < P.length() && i < T.length()) {
        if (j < 0 || T[i] == P[j]) {
            i++;
            j++;
        } else {
            j = next[j]
        }
    }
    return i-j;
    // 如何通过返回值 i-j 判断是否匹配成功?
    // 若 i-j+P.length() < T.length(), 则匹配成功; 否则匹配失败
}
```

- 串与匹配
  - 匹配问题
    - KMP
      - 快速生成查询表

- 串与匹配
  - 匹配问题
    - KMP
      - 快速生成查询表

看公式:

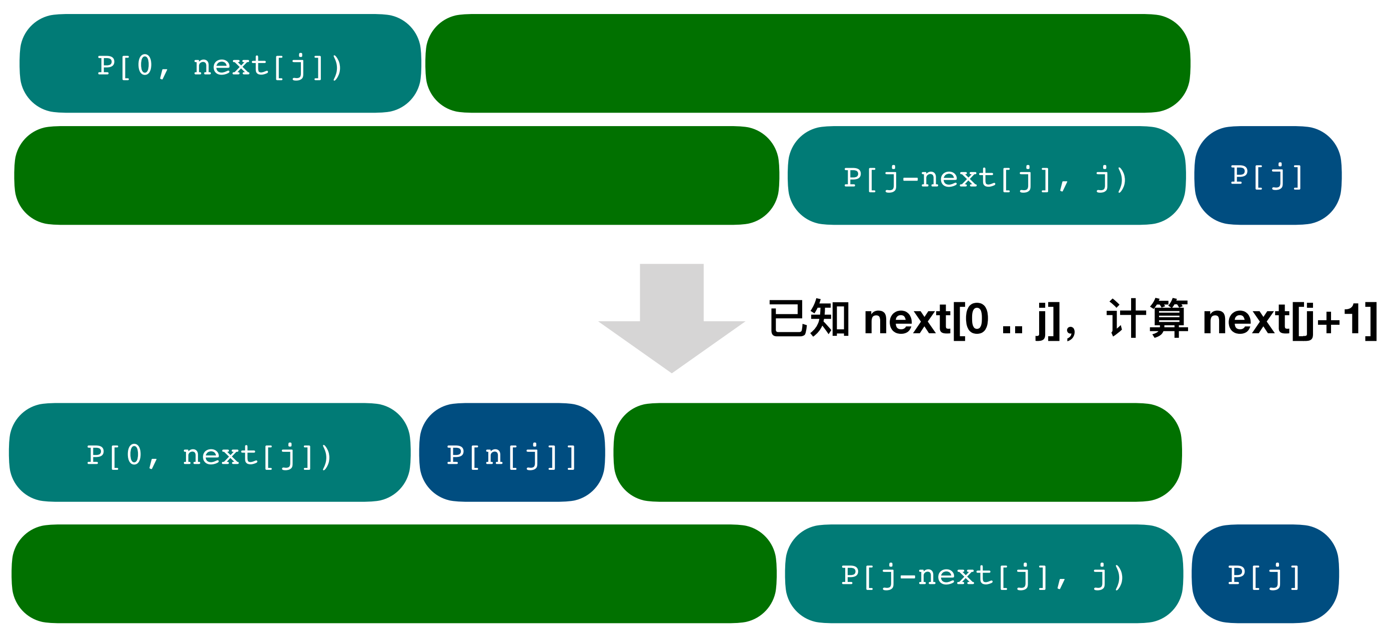
```
1 <= j < length(P),
next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}
```

说人话:

next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度

枚举法生辰 next 表，时间复杂度 O(M^3)

- 串与匹配
  - 匹配问题
    - KMP
      - 快速生成查询表



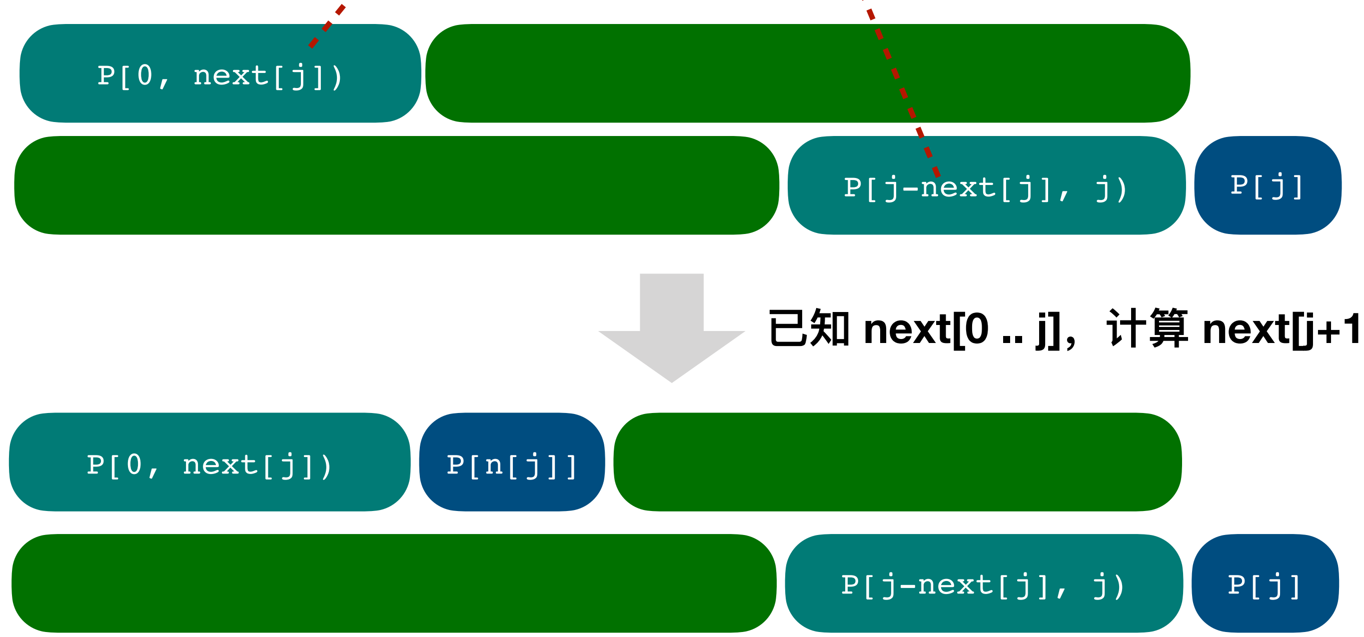
看公式:

```
1 <= j < length(P),
next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}
```

说人话: **next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度**

- 串与匹配
  - 匹配问题
    - KMP
      - 快速生成查询表

P[0, next[j]) 和 P[j-next[j], j) 相等，且是最长子串



看公式:

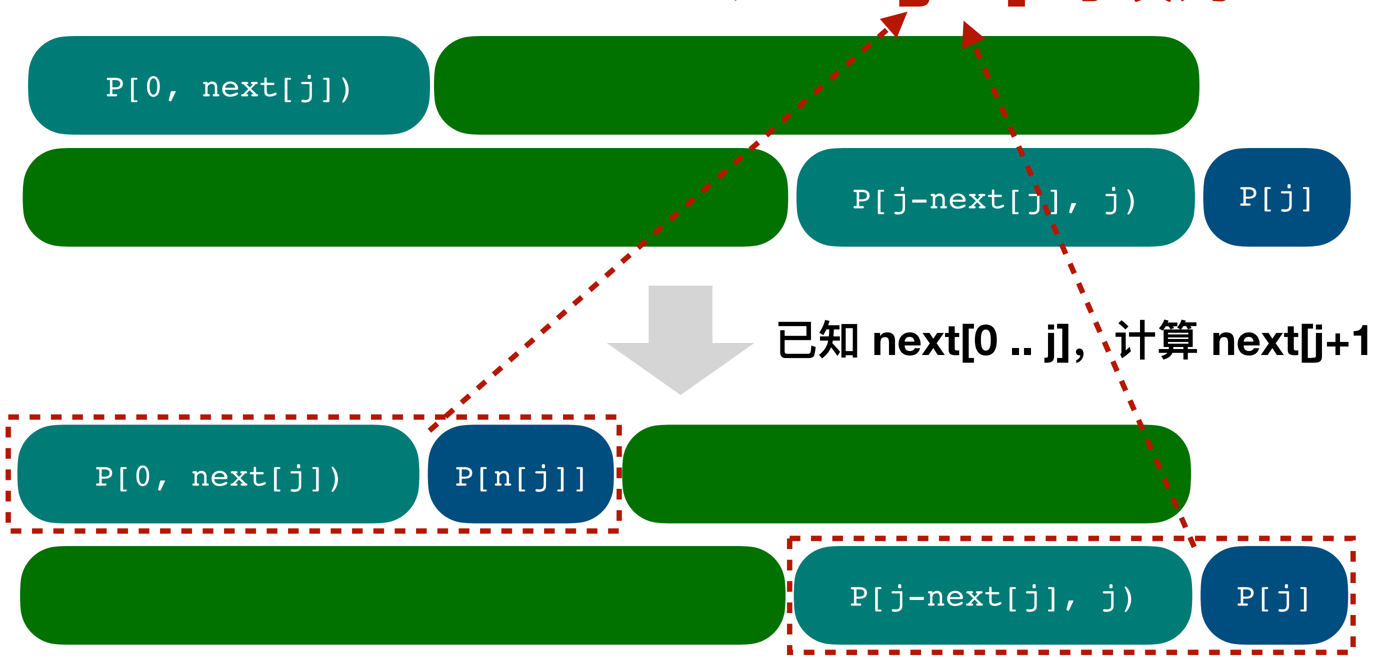
```
1 <= j < length(P),
next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}
```

说人话: **next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度**

- 串与匹配
  - 匹配问题
    - KMP
      - 快速生成查询表

P[0, next[j]) 和 P[j-next[j], j) 相等，且是最长子串

若 P[next[j]] == P[j]，则前缀 P[0, next[j])+P[n[j]] 和后缀 P[j-next[j], j)+P[j] 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1



看公式:

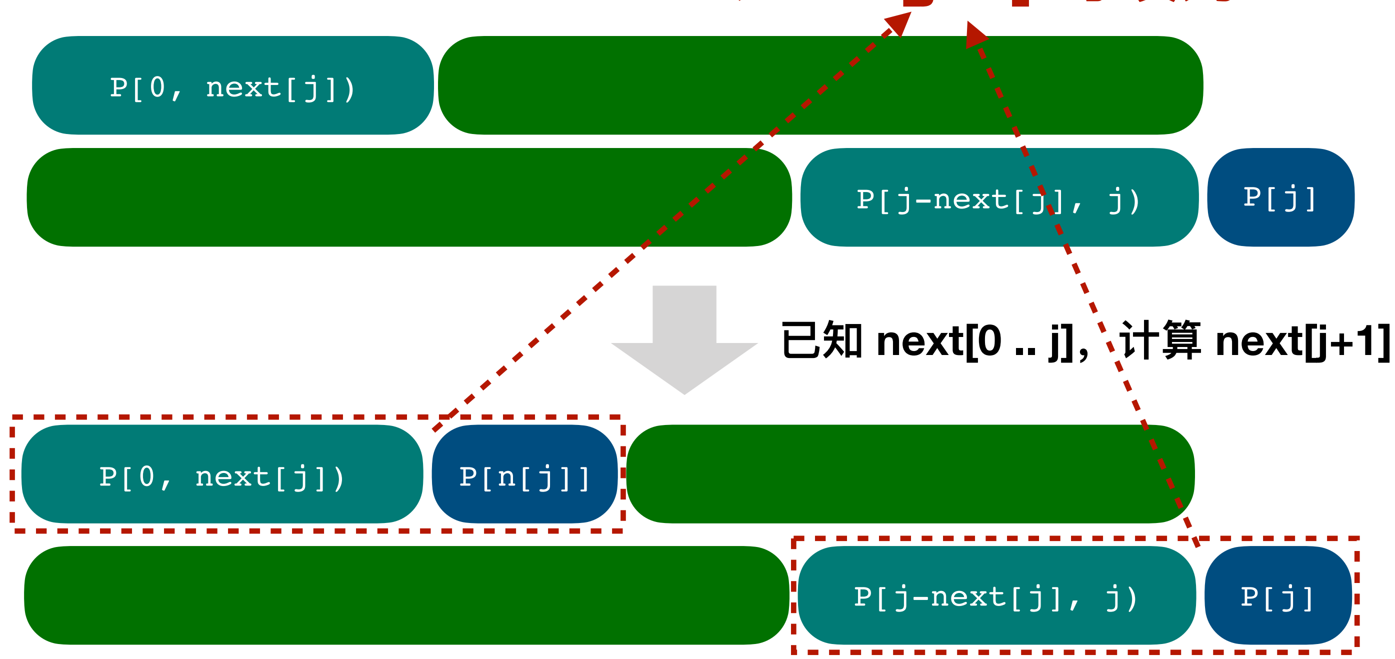
```
1 <= j < length(P),
next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}
```

说人话: next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度

- 串与匹配
  - 匹配问题
    - KMP
      - 快速生成查询表

P[0, next[j]) 和 P[j-next[j], j) 相等，且是最长子串

若 P[next[j]] == P[j]，则前缀 P[0, next[j])+P[n[j]] 和后缀 P[j-next[j], j)+P[j] 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1



看公式:

```
1 <= j < length(P),
next[j] = max{0 <= t <= j | P[0, t) = P[j-t  j)}
```

说人话: next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子 的长度

看公式: **next[j+1]=next[j] + 1 iff P[j] == P[next[j]]**

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

看公式: $1 <= j < length(P)$,
$next[j] = max\{0 <= t <= j \mid P[0, t) = P[j-t, j)\}$

说人话: next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度

看公式: next[j+1]=next[j] + 1 iff P[j] == P[next[j]]

说人话: 若 P[next[j]] == P[j]，则前缀 P[0, next[j]+1) 和后缀 P[j-next[j], j+1) 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1

```
// 输入 Pattern, 生成 next 表
std::vector<int> build_next2(const std::string& P) {
    int j = -1;
    std::vector<int> next(P.length());
    next[0] = -1;
    while (++j < m - 1) {
        // 公式: next[j+1]=next[j] + 1 iff P[j] == P[next[j]]
        if (j > 0 && next[j] > 0 && P[j] == P[next[j]]) {
            next[j+1] = next[j] + 1;
        } else {
            next[j+1] = 0;
        }
    }
    return std::move(next);
}
```

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

看公式: `1 <= j < length(P),`
`next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}`

说人话: next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度

看公式: `next[j+1]=next[j] + 1 iff P[j] == P[next[j]]`

说人话: 若 P[next[j]] == P[j]，则前缀 P[0, next[j]+1) 和后缀 P[j-next[j], j+1) 相等，且是最长子串

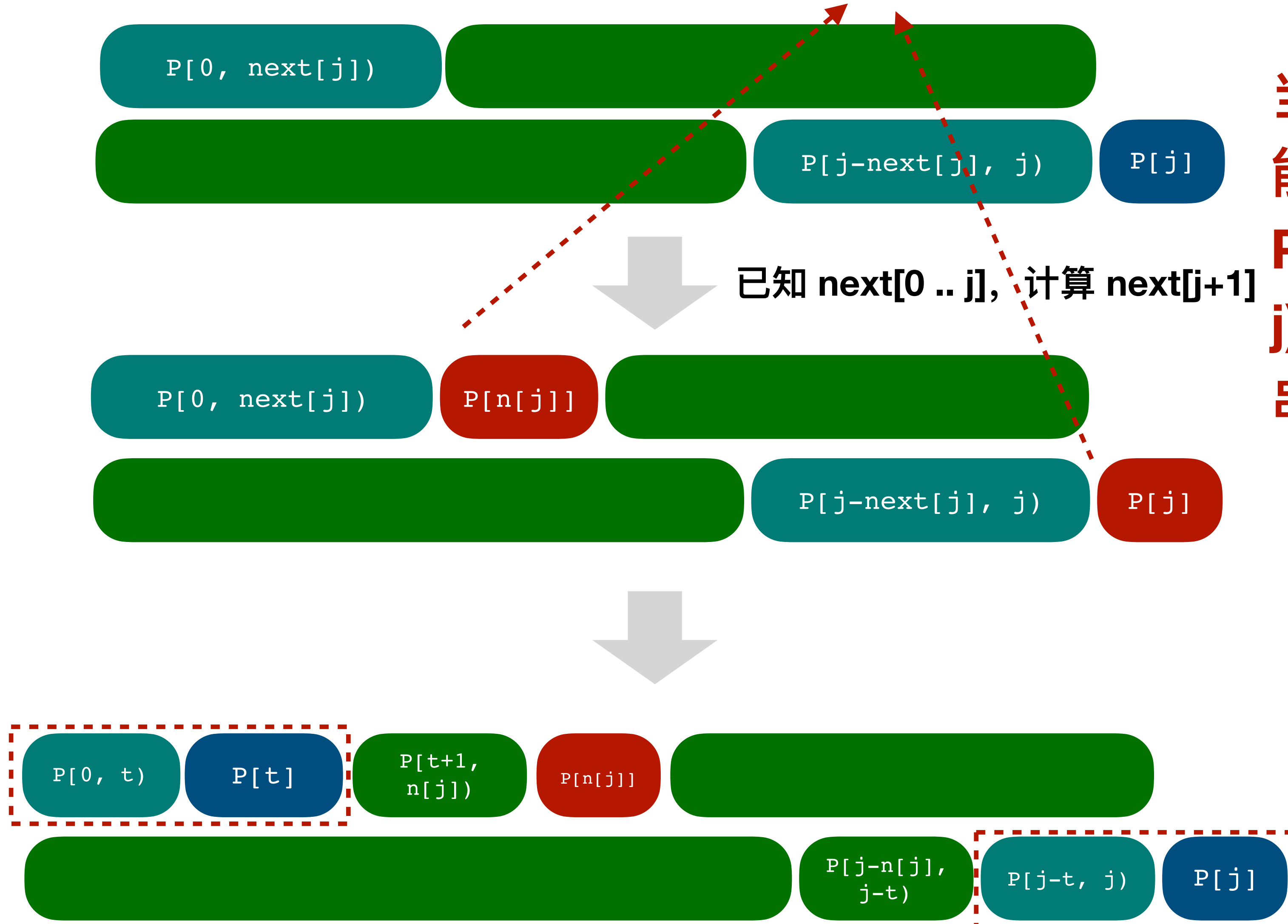
```
// 输入 Pattern，生成 next 表
std::vector<int> build_next2(const std::string& P) {
    int j = -1;
    std::vector<int> next(P.length());
    next[0] = -1;
    while (++j < m - 1) {
        // 公式: next[j+1]=next[j] + 1 iff P[j] == P[next[j]]
        if (j > 0 && next[j] > 0 && P[j] == P[next[j]]) {
            next[j+1] = next[j] + 1;
        } else {
            next[j+1] = 0;
        }
    }
    return std::move(next);
}
```

这个算法是错的！
当 P[j] != P[next[j]] 时，我们不能直接将 next[j+1] 置为0，WHY？！

- 串与匹配
  - 匹配问题
    - KMP
      - 生成查询表

P[0, next[j]) 和 P[j-next[j], j) 相等，且是最长子串

若 P[next[j]] == P[j]，则前缀 P[0, next[j])+P[n[j]] 和后缀 P[j-next[j], j)+P[j] 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1



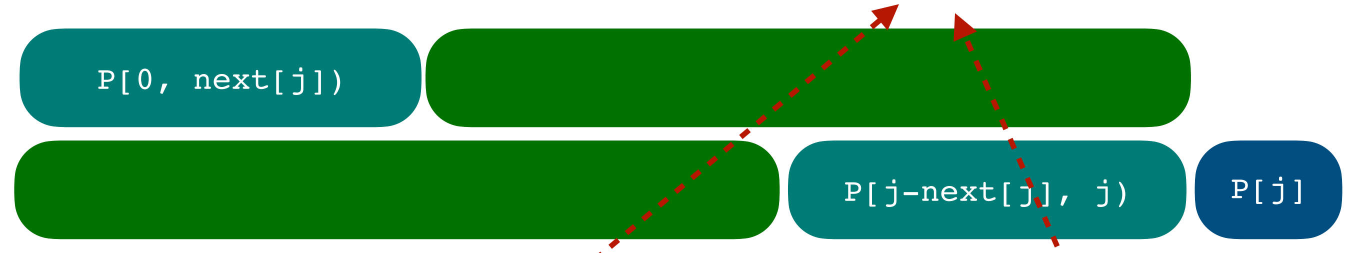
当 P[n[j]] != P[j]，有可能存在 t，使得前缀 P[0,)+P[t] 和后缀 P[j-t, j) +P[j]相等，且是最长子串，寻找t！

- 串与匹配
  - 匹配问题
    - KMP
      - 生成查询表

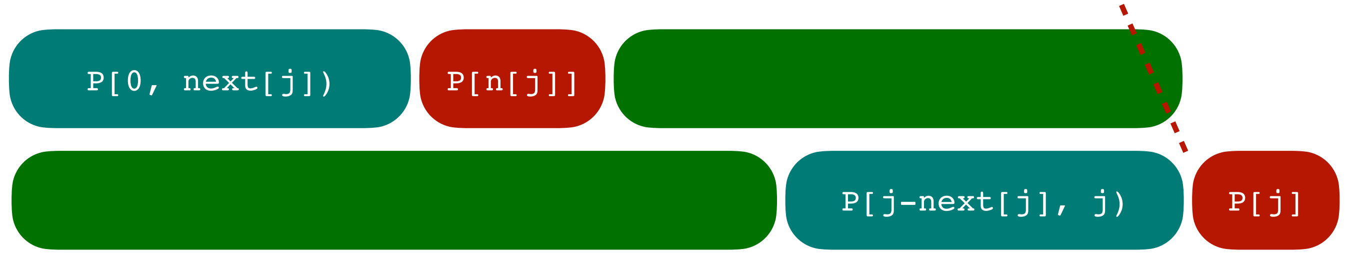
P[0, next[j]) 和 P[j-next[j], j) 相等，且是最长子串

若 P[next[j]] == P[j]，则前缀 P[0, next[j])+P[n[j]] 和后缀 P[j-next[j], j)+P[j] 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1

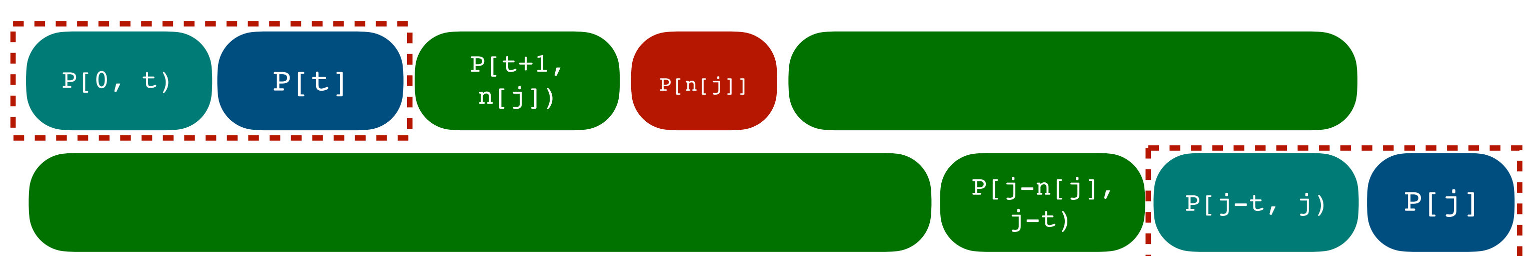
当 P[n[j]] != P[j]，有可能存在 t，使得前缀 P[0,)+P[t] 和后缀 P[j-t, j) +P[j]相等，且是最长子串，寻找t !



已知 next[0 .. j]，计算 next[j+1]



问题：如何寻找 t，使得前缀 P[0, t)+P[t] 和后缀 P[j-t, j)+P[j] 相等，且是最长子串?



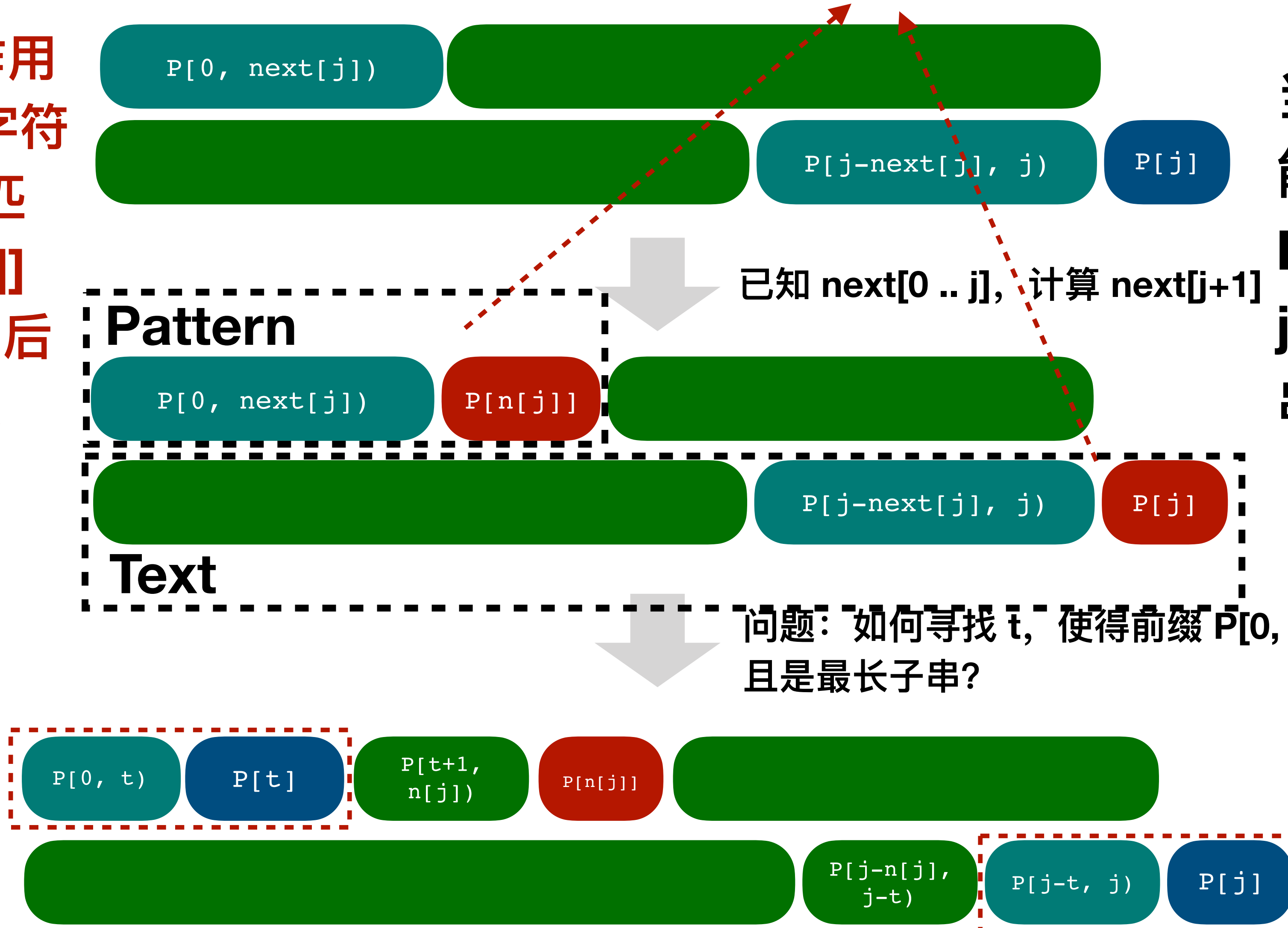
- 串与匹配
  - 匹配问题
    - KMP
      - 生成查询表

P[0, next[j]) 和 P[j-next[j], j) 相等，且是最长子串

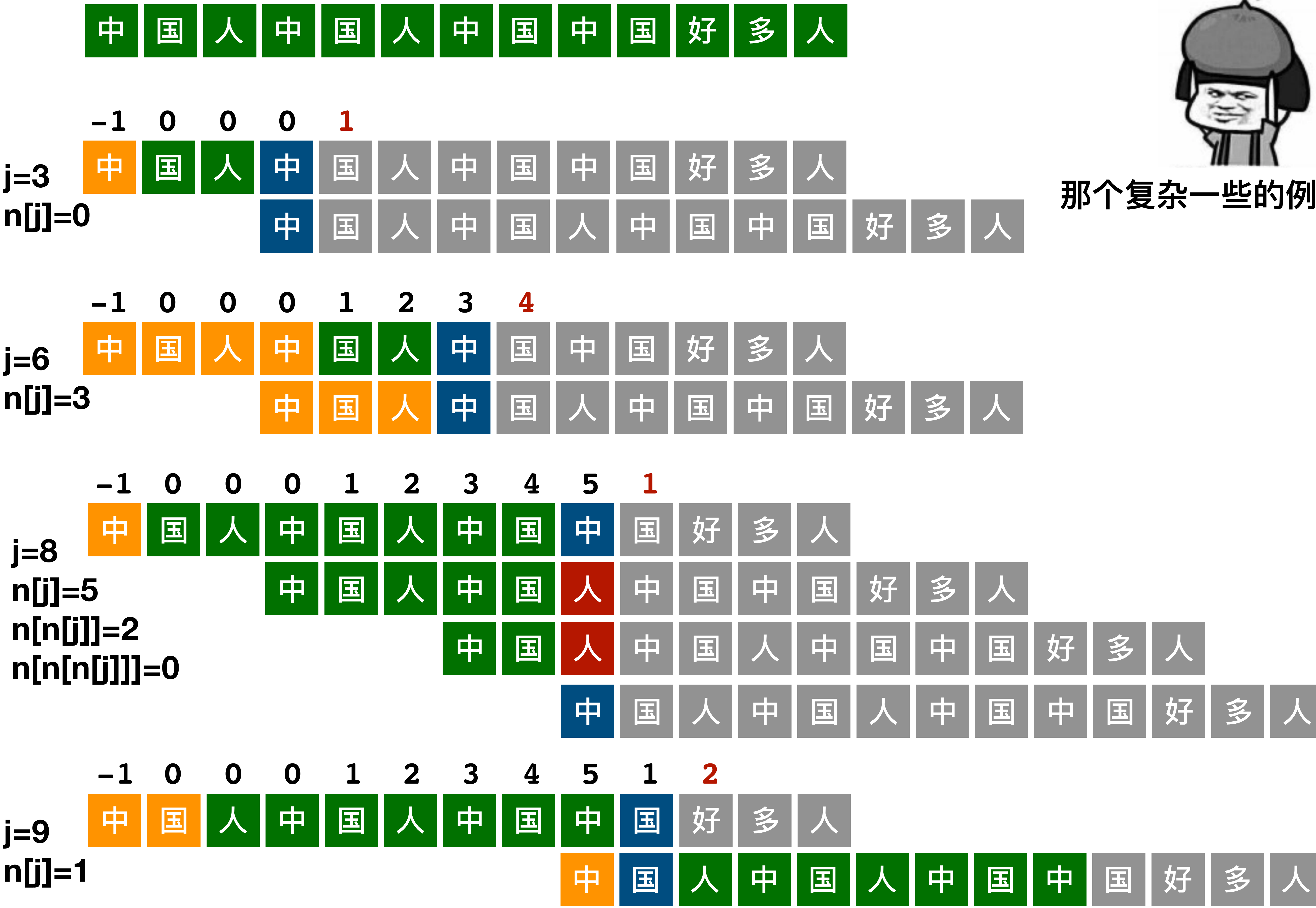
若 P[next[j]] == P[j]，则前缀 P[0, next[j])+P[n[j]] 和后缀 P[j-next[j], j)+P[j] 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1

当 P[n[j]] != P[j]，有可能存在 t，使得前缀 P[0,)+P[t] 和后缀 P[j-t, j) +P[j]相等，且是最长子串，寻找t！

一个思路：把生成 next 的过程，看作用 KMP 算法匹配在字符串 P[0, j) +P[j] 中匹配 P[0, n[j])+P[n[j]]（前者作为 Text，后者作为 Pattern）。当匹配到 P[j] != P[n[j]] 时，j 不变，n[j] 回退到 n[n[j]]，并继续匹配 P[j] 与 P[n[n[j]]]



问题：如何寻找 t，使得前缀 P[0, t)+P[t] 和后缀 P[j-t, j)+P[j] 相等，且是最长子串？


中国人中国人中国中国好多人
-1 0 0 0 1
j=3
n[j]=0
中国人中国人中国中国好多人
中国人中国人中国中国好多人
-1 0 0 0 1 2 3 4
j=6
n[j]=3
中国人中国人中国中国好多人
中国人中国人中国中国好多人
-1 0 0 0 1 2 3 4 5 1
j=8
n[j]=5
n[n[j]]=2
n[n[n[j]]]=0
中国人中国人中国中国好多人
中国人中国人中国中国好多人
中国人中国人中国中国好多人
中国人中国人中国中国好多人
-1 0 0 0 1 2 3 4 5 1 2
j=9
n[j]=1
中国人中国人中国中国好多人
中国人中国人中国中国好多人

那个复杂一些的例子

- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

看公式: `1 <= j < length(P),`
`next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}`

说人话: next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度

看公式: `next[j+1]=next[j] + 1 iff P[j] == P[next[j]]`
`else try P[j] and P[next[next[j]]`

说人话: 若 P[next[j]] == P[j]，则前缀 P[0, next[j])+P[n[j]] 和后缀 P[j-next[j], j)+P[j] 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1；

否则，递归比较 P[next[next[j]]] 与 P[j]；

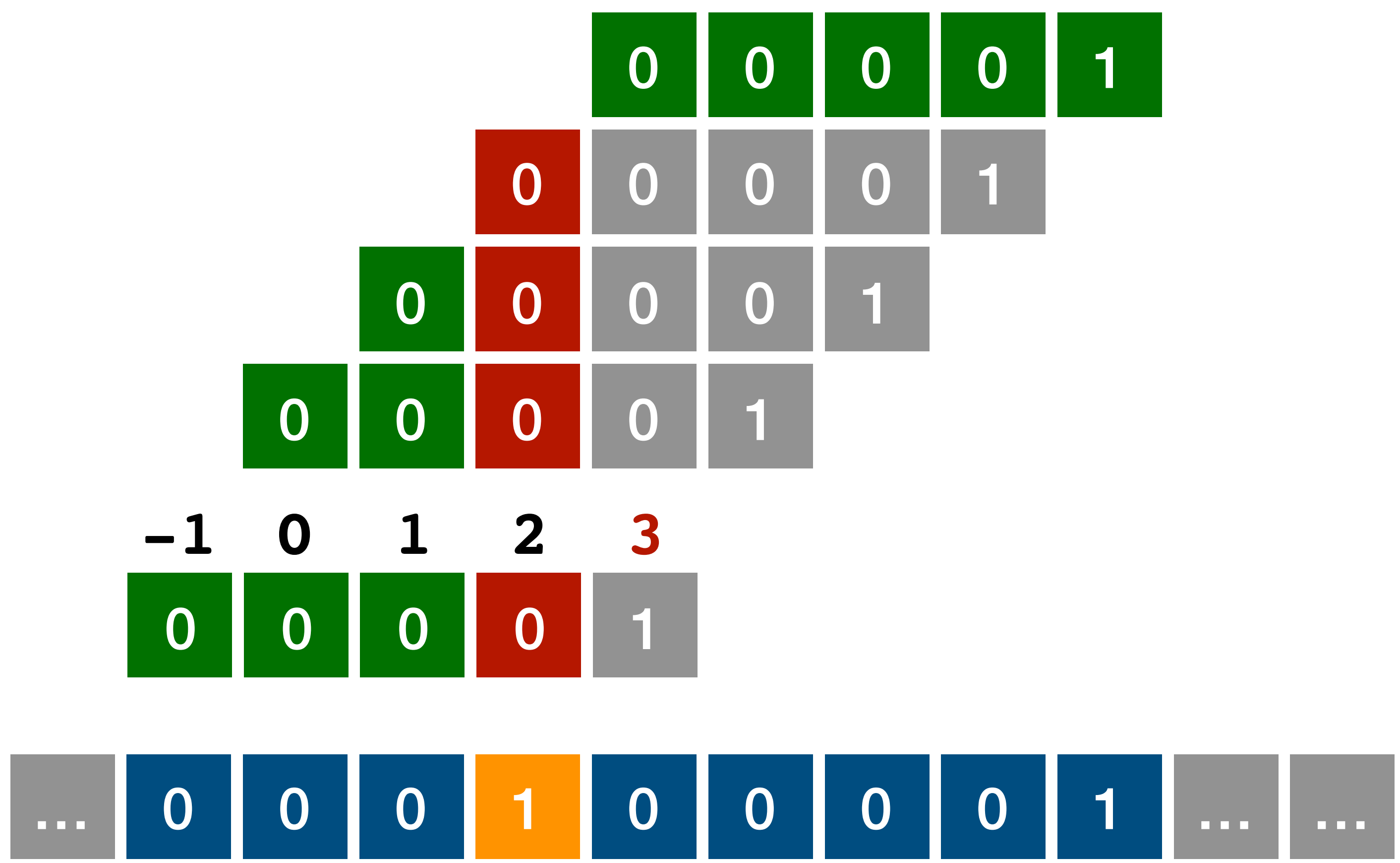
```cpp
// 输入 Pattern，生成 next 表
std::vector<int> build_next3(const std::string& P) {
    int j = -1;
    std::vector<int> next(P.length());
    next[0] = -1;
    while (++j < m - 1) {
        // if (j > 0 && next[j] > 0 && P[j] == P[next[j]]) {
        //     next[j+1] = next[j] + 1;
        // } else {
        //     next[j+1] = 0;
        // } 红色代码是前面错误的理解
        i = next[j];
        while (i >= 0 && P[j] != P[i]) {
            i = next[i];
        }
        next[j+1] = i + 1;
    }
    return std::move(next);
}
```

One More Thing

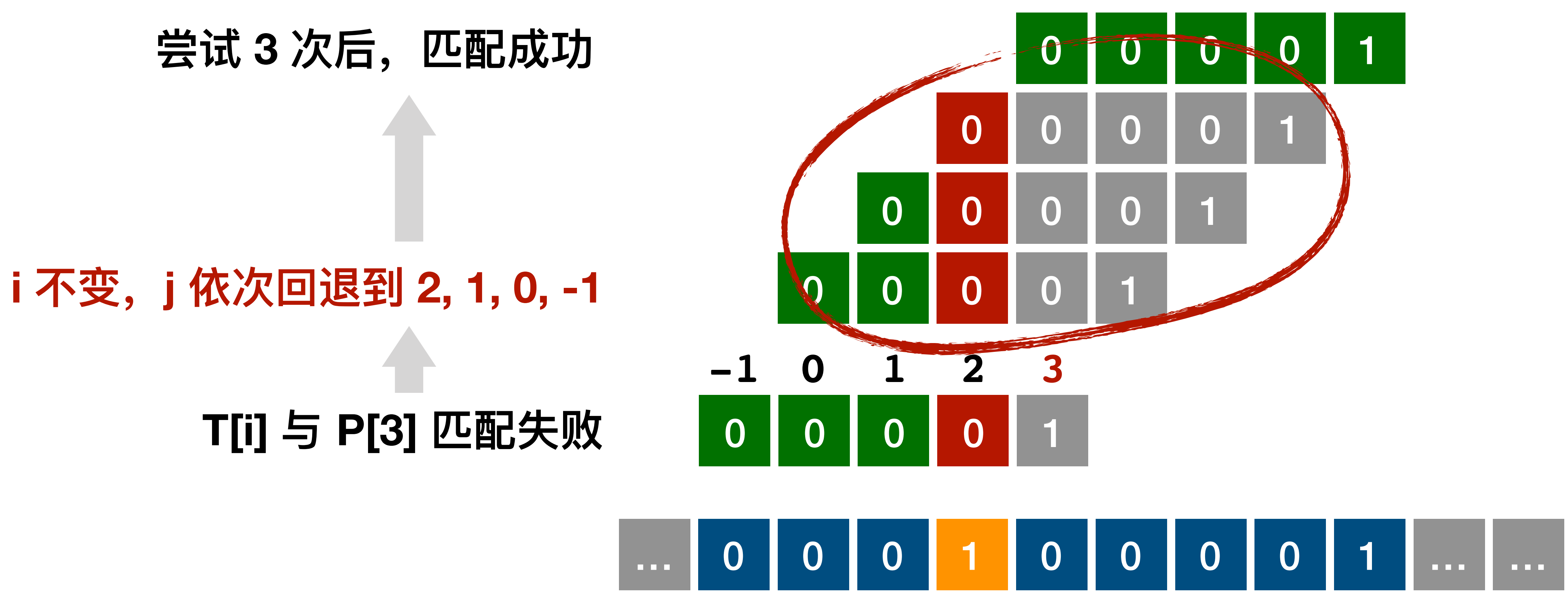
尝试 3 次后，匹配成功

i 不变，j 依次回退到 2, 1, 0, -1

T[i] 与 P[3] 匹配失败



- 串与匹配
  - 匹配问题
    - KMP
      - KMP 改进



- 串与匹配
  - 匹配问题
    - KMP
      - KMP 改进

1) 枚举法，遇到 `T[i] != P[3]`

2) 此时必然 `T[i-3, i) == Pattern[0, 3)`

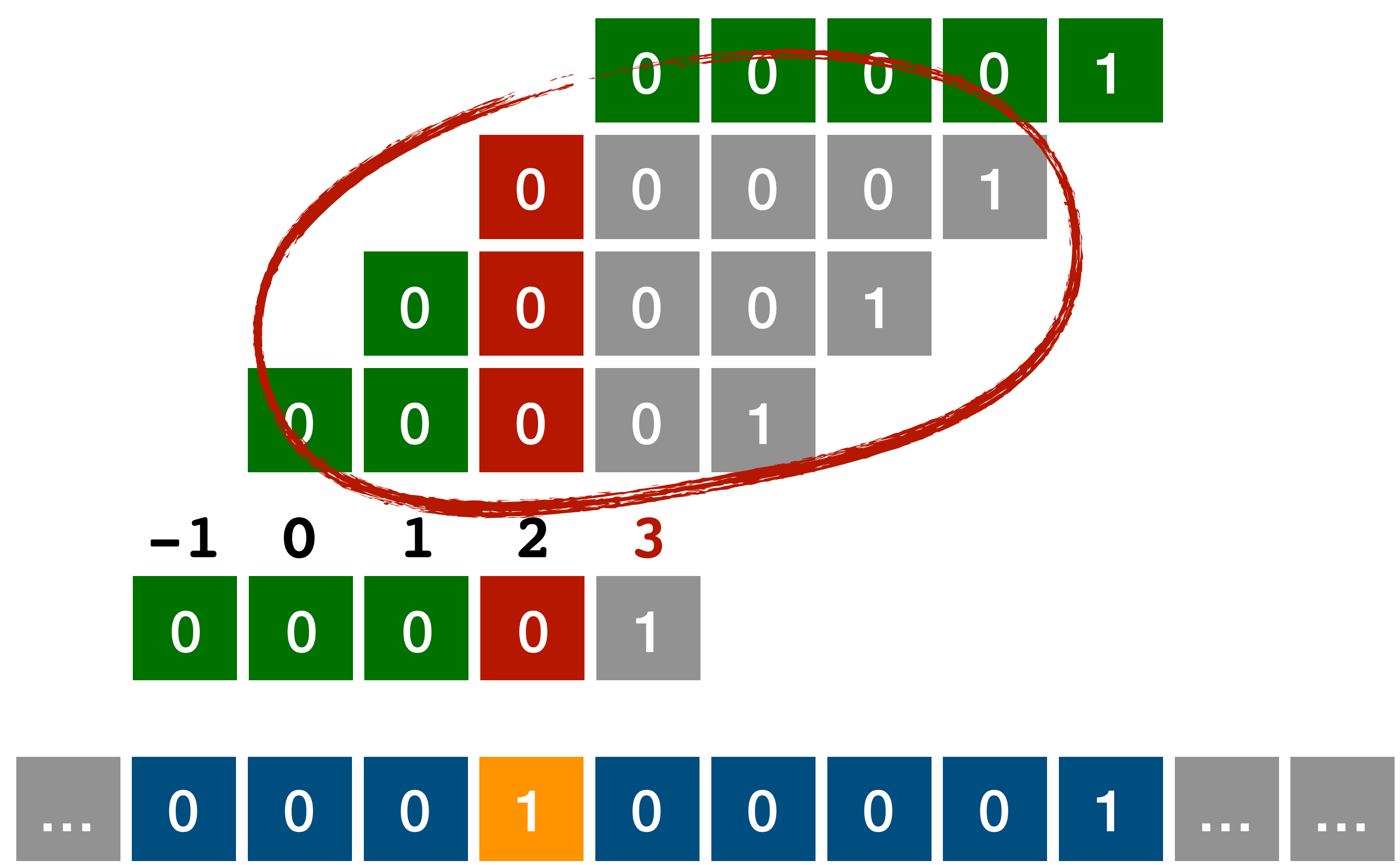
3) `T[i-3, i)` 或 `Pattern[0, 3)` 是已经枚举过的串，我们一定知道一些信息...

4) 如果 P[3] == P[next[3]]，并且在 P[3] 匹配失败，是否可以跳过 P[next[3]]，直接比较 P[next[next[3]]] 与 T[i]?

尝试 3 次后，匹配成功

i 不变，j 依次回退到 2, 1, 0, -1

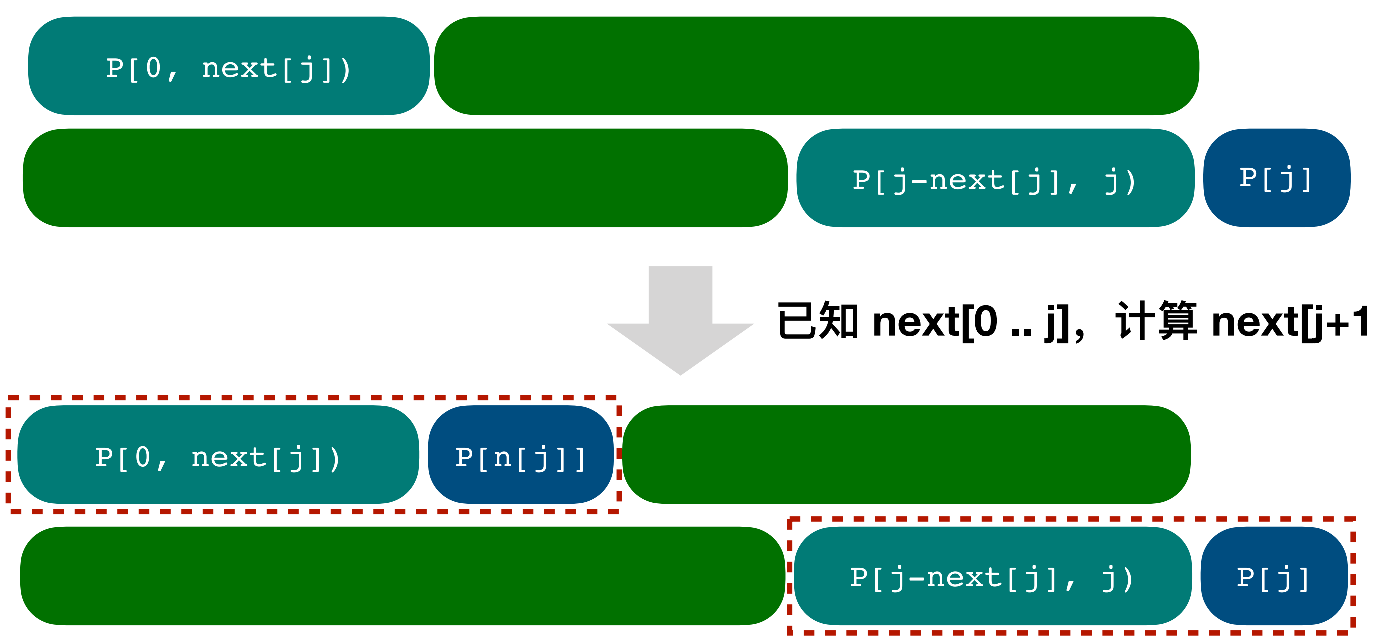
T[i] 与 P[3] 匹配失败



- 串与匹配
  - 匹配问题
    - KMP
      - KMP 改进

1) 枚举法，遇到 `T[i] != P[3]`
2) 此时必然 `T[i-3, i) == Pattern[0, 3)`
3) `T[i-3, i)` 或 `Pattern[0, 3)` 是已经枚举过的串，我们一定知道一些信息...
4) 如果 P[3] == P[next[3]]，并且在 P[3] 匹配失败，是否可以跳过 P[next[3]]，直接比较 P[next[next[3]]] 与 T[i]?

回顾求 next 表的方法：
若 P[next[j]] == P[j]，则前缀 P[0, next[j]+1) 和后缀 P[j-next[j], j+1) 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1

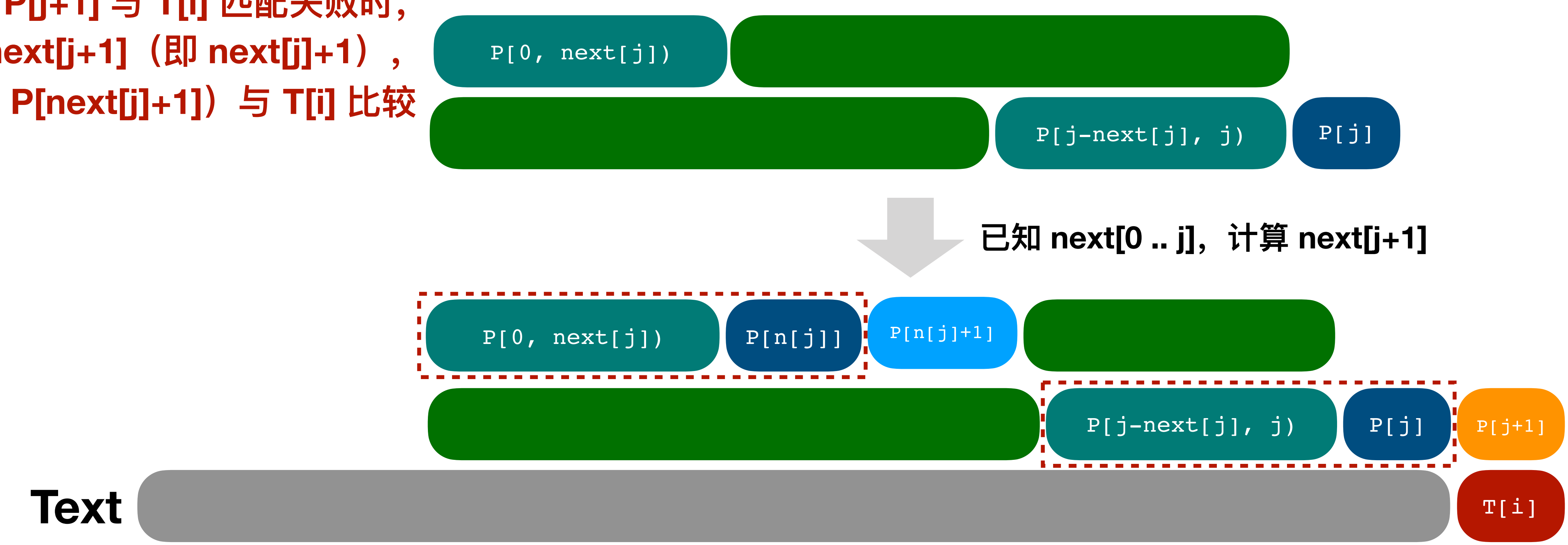


- 串与匹配
  - 匹配问题
    - KMP
      - KMP 改进

1) 枚举法，遇到 `T[i] != P[3]`
2) 此时必然 `T[i-3, i) == Pattern[0, 3)`
3) `T[i-3, i)` 或 `Pattern[0, 3)` 是已经枚举过的串，我们一定知道一些信息...
4) 如果 P[3] == P[next[3]]，并且在 P[3] 匹配失败，是否可以跳过 P[next[3]]，直接比较 P[next[next[3]]] 与 T[i]?

回顾求 next 表的方法：
若 P[next[j]] == P[j]，则前缀 P[0, next[j]+1) 和后缀 P[j-next[j], j+1) 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1

回顾 KMP 算法：当 P[j+1] 与 T[i] 匹配失败时，i 不变，j+1 回退到 next[j+1]（即 next[j]+1），用 P[next[j+1]] （即 P[next[j]+1]）与 T[i] 比较



- 串与匹配
  - 匹配问题
    - KMP
      - KMP 改进

1) 枚举法，遇到 `T[i] != P[3]`

2) 此时必然 `T[i-3, i) == Pattern[0, 3)`

3) `T[i-3, i)` 或 `Pattern[0, 3)` 是已经枚举过的串，我们一定知道一些信息...

4) 如果 P[3] == P[next[3]]，并且在 P[3] 匹配失败，是否可以跳过 P[next[3]]，直接比较 P[next[next[3]]] 与 T[i]?

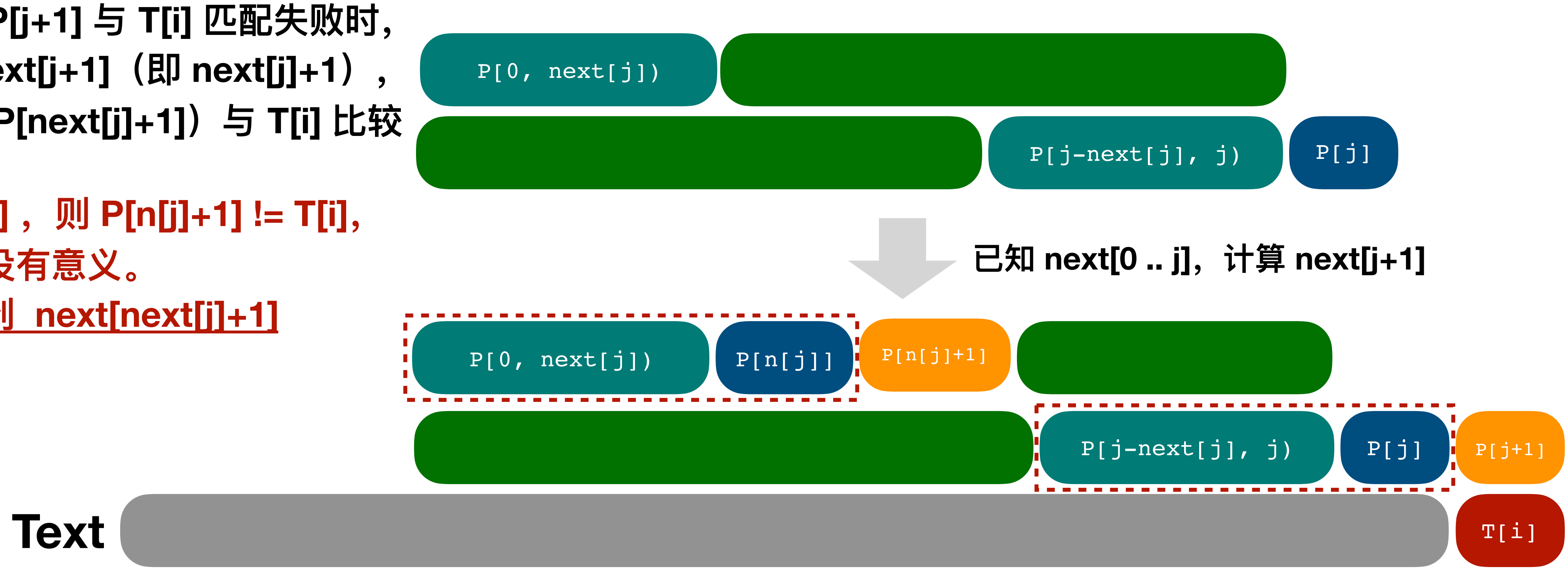
回顾求 next 表的方法：

若 P[next[j]] == P[j]，则前缀 P[0, next[j]+1) 和后缀 P[j-next[j], j+1) 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1

回顾 KMP 算法：当 P[j+1] 与 T[i] 匹配失败时，i 不变，j+1 回退到 next[j+1]（即 next[j]+1），用 P[next[j+1]] （即 P[next[j]+1]）与 T[i] 比较

若 P[n[j]+1] == P[j+1]，则 P[n[j]+1] != T[i], 接下来一次的匹配并没有意义。

<u>不如让 j+1 直接回退到 next[next[j]+1]</u>



- 串与匹配
  - 匹配问题
    - KMP
      - 查询表

看公式: `1 <= j < length(P),`
`next[j] = max{0 <= t <= j | P[0, t) = P[j-t, j)}`

说人话: next[j] 是 Pattern 的前缀和后缀的最长（不重叠）公共子串的长度

看公式: `next[j+1]=next[j] + 1 iff P[j] == P[next[j]]`
`else try P[j] and P[next[next[j]]`

说人话: 若 P[next[j]] == P[j]，则前缀 P[0, next[j])+P[n[j]] 和后缀 P[j-next[j], j)+P[j] 相等，且是最长子串，那么 next[j+1] 可设为 next[j]+1；

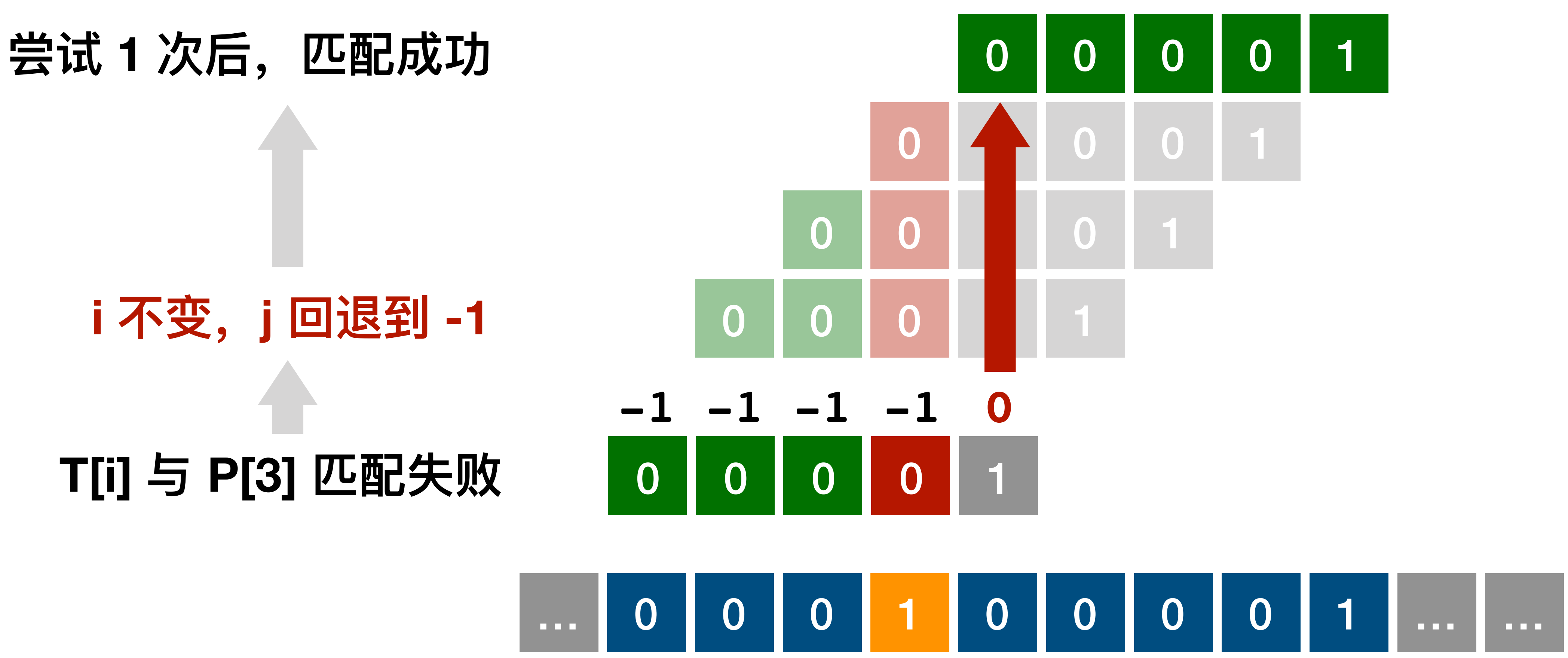
否则，递归比较 P[next[next[j]]] 与 P[j]；

若 P[n[j]+1] == P[j+1]，则 P[n[j]+1] != T[i]，
接下来一次的匹配并没有意义。
<u>不如让 j+1 直接回退到 next[next[j]+1]</u>

```
// 输入 Pattern，生成 next 表
std::vector<int> build_next4(const std::string& P) {
    int j = -1;
    std::vector<int> next(P.length());
    next[0] = -1;
    while (++j < m - 1) {
        i = next[j]
        while (i >= 0 && P[j] != P[i]) {
            i = next[i]
        }
        // KMP 优化，若 P[n[j]+1]==P[j+1]，则直接回退到 next[next[j]+1]
        next[j+1] = (P[j+1] != P[i+1] ? i + 1 : next[i + 1]);
    }
    return std::move(next);
}
```

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

- 串与匹配
  - 匹配问题
    - KMP
      - KMP 改进



- 串与匹配
  - 匹配问题
    - KMP
      - KMP 总结：
        - KMP 算法的核心思想：
          - 充分利用 Pattern 的信息，在匹配过程中 Pattern 尽量少回退（快速前进），Text 绝不回退
        - KMP 算法与枚举法比较：
          - 单次匹配成功概率越大（字符集越小，例如纯二进制01串），KMP 优势越明显；
          - 否则（例如中文文章），枚举法实际效率相差无几
          - —— 实际场景中，不要拘泥于算法的理论时间复杂度，选择合适的算法，**快速**、**准确**地实现它！

目录

- 串
  - 串与匹配问题
    - KMP
    - Rabin-karp

- 串与检索
  - 检索问题
    - Rabin-Karp
      - 字符串 T 和一个要匹配的字符串 P；
      - 为 T 里每一个长度为 P.length() 的子串求一个 Hash 值，然后和 P 的 Hash 值比较。如果 Hash 值相等，再去匹配字符串本身。

- 串与检索
  - 检索问题
    - Rabin-Karp

为 T 里每一个长度为 P.length() 的子串求一个 Hash 值，然后和 P 的 Hash 值比较。如果 Hash 值相等，再去匹配字符串本身。

字符	系数
我	0
是	1
中	2
国	3
人	4
为	5
心	6
团	7
结	8
梦	9
奋	10
斗	11

设计一种基于12进制数的 Hash 函数：

中	国	人	为	中	国	梦	奋	斗

2 3 4 5 2 3 9 10 11 $_{(12)}$ = 980694995

...

2 3 4 2 3 4 5 2 3 $_{(12)}$ = 979970283

1 2 3 4 2 3 4 5 2 $_{(12)}$ = 511645886

0 1 2 3 4 2 3 4 5 $_{(12)}$ = 42637157

我	是	中	国	人	中	国	人	为	中	国	心	团	结	中	国	人	为	中	国	梦	奋	斗
0	1	2	3	4	2	3	4	5	2	3	6	7	8	2	3	4	5	4	3	9	10	11

- 串与检索
  - 检索问题
    - Rabin-Karp

为 T 里每一个长度为 P.length() 的子串求一个 Hash 值，然后和 P 的 Hash 值比较。如果 Hash 值相等，再去匹配字符串本身。

字符	系数
我	0
是	1
中	2
国	3
人	4
为	5
心	6
团	7
结	8
梦	9
奋	10
斗	11

设计一种基于12进制数的 Hash 函数：

中	国	人	为	中	国	梦	奋	斗

2 3 4 5 2 3 9 10 11 (12) = 980694995

问题1：Pattern 较长时，对应的 Hash 值将很大，超过 64 位计算机的最大整数

改进 Hash 函数，增加一个求余操作，解决 Hash 值过大问题，带来额外的字符串匹配次数：

2 3 4 5 2 3 9 10 11 (12) % 137 = 86

- 串与检索
  - 检索问题
    - Rabin-Karp

为 T 里每一个长度为 P.length() 的子串求一个 Hash 值，然后和 P 的 Hash 值比较。如果 Hash 值相等，再去匹配字符串本身。

字符	系数
我	0
是	1
中	2
国	3
人	4
为	5
心	6
团	7
结	8
梦	9
奋	10
斗	11

设计一种基于12进制数的 Hash 函数：

问题2：计算 Hash 值的复杂度是 O(M)， 所以整个算法依然是 O(NM)!

$1\ 2\ 3\ 4\ 2\ 3\ 4\ 5\ 2_{(12)} = 511645886$

回想 X 进制数转十进制数算法，当 T 的子串往右前进一位时，相邻两个子串的 Hash 值是可以在 O(1) 内快速计算的，这样整体时间复杂度 O(N+M)

(42637157 - 0*12^8) * 12 + 2 = 511645886

$0\ 1\ 2\ 3\ 4\ 2\ 3\ 4\ 5_{(12)} = 42637157$

我	是	中	国	人	中	国	人	为	中	国	心	团	结	中	国	人	为	中	国	梦	奋	斗
0	1	2	3	4	2	3	4	5	2	3	6	7	8	2	3	4	5	4	3	9	10	11

- 串与匹配
  - 匹配问题
    - Rabin-Karp
      - Rabin-Karp 算法的核心思想：
        - 利用 Hash 函数，快速过滤不匹配的字符串；
      - Rabin-Karp 算法与 KMP 或其他基于后缀树（Trie）的方法比较：
        - Hash 函数的实现简单可依赖；
        - Hash 函数对字符串内容容忍度高，支持各种 UNICODE，对字符串、匹配串长度不那么敏感；
        - 当字符串非常非常多时，基于 Hash 函数的方法易于分布式；