数据结构与算法（Python）-03+/KMP

刘云淮 Yunhuai.liu@pku.edu.cn
http://www.yunhuai.net/DSA2026/CoursePage/DSA2026.html
北京大学计算机学院

Some slides adapted from https://ics.uci.edu/~goodrich/teach/cs262P/ppt/05-ExactMatch.pptx

字符串匹配

给定一个**模式串(子串)**和一个**母串**， 求模式串在母串中出现的位置。母串中找不到模式串位置就算**-1**。

示例说明

母串 (Text)

ABABDABACDABABCABAB

模式串 (Pattern)

ABABCABAB

匹配结果：**位置 10（从0开始计数）**

暴力字符串匹配算法

核心概念：母串匹配起点 (MS)

母串 a ，子串 b ， 若a[i]和b[0]比较， 则称**"母串匹配起点"**为a[i]。

母串指针

指向母串中**待比较的字符**

模式串指针

指向子串中**待比较的字符**

算法流程

1. 比较母串指针和子串指针指向的字符
2. **如果相等**，则两个指针都加1
3. **如果不相等**.....

暴力字符串匹配算法(fgrep)

⚠核心问题

暴力算法**母串指针需要回溯**。 假设子串前 `n-1` 个字符已经被匹配，第 `n` 个失配：

1失配前

母串：

a0a1a2…an-1an…

子串：

b0b1b2…bn-1bn…

MS位置

2失配后（回溯）

母串：

a0a1a2…an-1an…

子串：

a1b0b1…bn-2bn-1…

MS+1，指针回溯

关键操作：母串匹配起点+1，母串指针回溯到母串匹配起点

Alfred Aho

Alfred V. Aho
Columbia University

1975 发明了 fgrep
Fast Global Regular Expression Print

2020 图灵奖
Turing Award

计算机科学最高荣誉

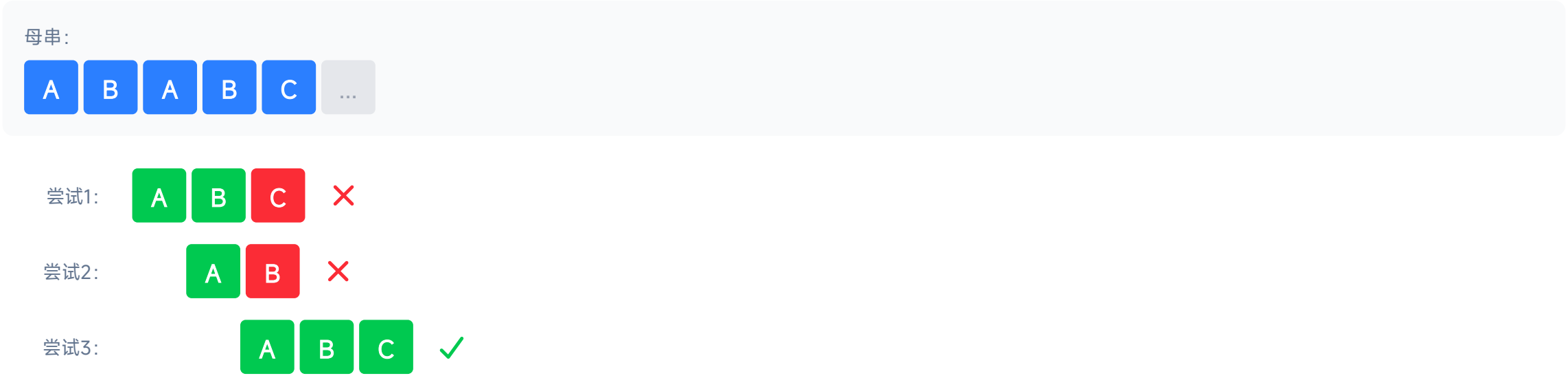
Images from https://awards.acm.org/about/2020-turing

暴力字符串匹配算法示例

算法特点

每个位置都会**尝试比较**

匹配过程可视化



需要尝试多个起始位置，直到找到匹配或遍历完母串

暴力字符串匹配算法性能分析

最坏情况

$O(mn)$

其中 m 为模式串长度，n 为母串长度

期望情况

线性时间

对于随机字符串，期望运行时间为 O(n)

数学推导

假设：模式串 P 和文本串 T 分别是从大小为 k 的字母表中均匀且独立地随机选取的长度为 m 和 n 的字符串。

定义随机变量：$X_{i,j}$，当且仅当 P[i] 与 T[j] 进行比较时，$X_{i,j} = 1$

概率：$P(X_{i,j}=1) = 1/k^i$（需要前 i 个字符完全匹配才能触发此次比较）

期望的线性性：

$E[\text{比较次数}] = 1/k + 1/k^2 + 1/k^3 + \ldots + 1/k^m$

期望值 ≈ 2（当 k 较大时）

结论：比较次数的期望值是 $2n$

Donald Knuth

Donald E. Knuth
Stanford University

1973 发现 KMP 算法

Discovered the KMP algorithm (Morris and Pratt also published in a technical report in 1970 — all three published a joint paper in 1977)

1974 图灵奖
Turing Award

TAOCP

"The Art of Computer Programming" — formalized and popularized algorithm analysis (e.g., the "big O")

Image from https://en.wikipedia.org/wiki/Donald_Knuth

KMP字符串匹配算法

◎核心目标

不希望母串指针回溯！

问题场景

母串：

a0a1a2a3a4...an-1an...

子串：

a1a2a3b0b1...bk-1bk...bn...

⚠ 潜在问题

如果直接用 an 和 b0 比， 可能会**忽略**母串匹配起点为 a3 时的情况

✔关键发现

如果 a3a4....an-1 和 b0b1 ...bk-1 可以匹配上， 则**值得继续做下去**

🔑 **关键：**要在母串指针不回溯的情况下避免忽略上述情况

KMP字符串匹配算法

概念：字符串的前缀和后缀

定义

前缀 (Prefix)

b0b1…bn-1bn 的前缀是 b0b1…bk

(k=0,1,2…n)

后缀 (Suffix)

b0b1…bn-1bn 的后缀是 bk…bn-1bn

(k=0,1,2…n)

真前缀 (Proper Prefix)

b0b1…bn-1bn 的真前缀是 b0b1…bk

(k=0,1,2…n-1)

真后缀 (Proper Suffix)

b0b1…bn-1bn 的真后缀是 bk…bn-1bn

(k=1,2…n)

示例说明

字符串： "ABC"

前缀："", "A", "AB", "ABC"
真前缀："", "A", "AB"

字符串： "ABC"

后缀："", "C", "BC", "ABC"
真后缀："C", "BC", "ABC"

KMP字符串匹配算法

前后缀概念的应用

场景分析

若发生了下述情况：

母串：

$a_0a_1a_2a_3a_4\ldots a_{n-1}a_n\ldots$

子串：

$a_1a_2a_3b_0b_1\ldots b_{k-1}b_k\ldots b_n\ldots$

条件：$a_3a_4\ldots a_{n-1}$ 和 $b_0b_1\ldots b_{k-1}$ 可以匹配上

推导过程

1. 则 $b_0b_1\ldots b_{k-1}$ 是**子串的前缀**
2. 同时也是 $a_0a_1\ldots a_{n-1}$ 的**后缀**
3. 由于 $a_0a_1\ldots a_{n-1} = b_0b_1\ldots b_{n-1}$
4. 因此 $b_0b_1\ldots b_{k-1}$ 也是 $b_0b_1\ldots b_{n-1}$ 的**后缀**

关键定义

若有 $b_0b_1\ldots b_{k-1}$ 是 $b_0b_1\ldots b_{n-1}$ 的**真后缀**，则称 $b_0b_1\ldots b_{k-1}$ 是字符 b_n 的**前后缀**。

KMP字符串匹配算法

最长前后缀的应用

场景分析

发生了下述情况时：

母串：

a0a1a2a3a4...an-1an...

子串：

a1a2a3b0b1...bk-1bk...bn...

条件：a3a4....an-1an 和 b0b1 ...bk-1 bk 可以匹配上

重要发现

字符 bn 的**前后缀可能不止一个**，要考查**最长的**

匹配策略

1. an 和 bn 失配时，直接比较 an 和 bn **最长前后缀**的后一个字符 bk
2. 即**母串指针不回溯**，**子串指针移到k**
3. 若还失配，则比较 an 和 bk 的**最长前后缀**(即 bn 的**次长前后缀**)后一个字符 bp
4. 若再失配，则比较 an 和 bp **最长前后缀**(即 bn 的**第三长前后缀**)的后一个字符.....
5. 直到 an 和 b0 比较，若还失配，则**母串指针+1，子串指针回0**

KMP字符串匹配算法

算法合理性证明

❓需要证明的问题

发生了下述情况时：

母串：

$a_0a_1a_2a_3a_4\ldots a_{n-1}a_n\ldots$

子串：

$a_1a_2a_3b_0b_1\ldots b_{k-1}b_k\ldots b_n\ldots$

✅ 条件：$a_3a_4\ldots a_{n-1}a_n$ 和 $b_0b_1\ldots b_{k-1}b_k$ 可以匹配上

⚠️关键问题

母串匹配位置从 a_0 跳到了 a_3，**忽略了**母串匹配位置在 a_1 和 a_2 的情况，即忽略了母串分别从 a_1 和 a_2 开始与子串进行比较的情况。

✔需要证明

如果能证明母串分别从 a_1 和 a_2 开始与子串进行比较，**都不会使得 $a_1\ldots a_{n-1}$ 被匹配成功**，则忽略是**合理的**。

KMP字符串匹配算法

反证法证明

⚖反证法

假设母串分别从 a1 开始与子串进行比较， a1...an-1 和 b0...bm-1 **匹配成功**：

1. 则 b0b1b2b3...bm-1 是 a0a1a2a3a4...an-1 的**后缀**
2. 即是也是子串 b0b1b2b3...bn-1 的**后缀** （因 a0a1a2a3a4...an-1 = b0b1b2b3...bn-1 ）
3. 即为字符 bn 的**前后缀**
4. len(b0b1b2b3...bm-1) > len(b0b1 ...bk-1) ， 这和 b0b1 ...bk-1 是字符 bn 的**最长前后缀矛盾**

✓ **因此母串匹配位置为 a1 时不可能成功**

💡 **同理可证**：母串匹配位置为 a2 时也不可能成功

KMP字符串匹配算法

next数组的定义与性质

next数组定义

用 next[i] 表示字符 bi 的**最长前后缀的长度**

次长前后缀

字符 bi 的次长前后缀的长度为 next[next[i]]

第三长前后缀

字符 bi 的第三长前后缀的长度为 next[next[next[i]]]

递推关系

设 bi 最长前后缀长度为 k ， 则 b0b1...bk-1 是最长前后缀，和 bi-k...bi-1 相同

bi 次长前后缀就是 bk 的最长前后缀

故 Len(bi次长前后缀) = next[k] = next[next[i]]

KMP字符串匹配算法

算法核心思想

算法核心思想

1. 设立列表 `next` 。 `next[i]` 就是 `b[i]` 的**最长前后缀的长度**。
2. `next[i]` 表示匹配到子串字符 `b[i]` 时，若发生了**失配**， 则**母串指针不动**，**子串指针应该变为 next[i]**， 然后继续匹配，再失配，则子串指针变为 `next[next[i]]` .....

重要性质

- next**只和子串有关**，和母串无关
- 有 `next[1] = 0`

哨兵机制

- 记 `next[0] = -1` (哨兵)
- 则 `next[i] >= 0` (i = 1,2....)

特殊情况：若 `b[0]` 失配，则**母串指针+1，子串指针置为0**

KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的?

示例演示

母串:

acabacakg……

子串:

acabacaef…

Step 1	Step 2	Step 3	Step 4	Step 5
初始对齐	部分匹配	发现失配	利用next	继续匹配

核心优势：母串指针**从不回溯**，只移动子串指针!

KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的?

Step 2: 部分匹配

母串:

acabacakg……

已匹配
aca

acabacaef…

Step 1	Step 2	Step 3	Step 4	Step 5
✓	进行中	待续	待续	待续

→ 继续比较后续字符...

KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的?

Step 3: 发现失配

母串:

acabacakg……

已匹配
acabac

acabacaef…

✗ 失配!
k ≠ a

Step 1	Step 2	Step 3	Step 4	Step 5
✓	✓	失配	待续	待续

⚠ 传统算法：母串指针需要回溯！KMP算法：利用next数组避免回溯！

KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的?

Step 4: 利用next数组

母串:

acabacakg……

母串指针位置不变!

next数组
移动

cabacaef…

✓ 继续!
利用前后缀

关键: 子串 "acabaca" 的最长前后缀是 "aca", 长度为3, 所以 next[6] = 3

Step 1	Step 2	Step 3	Step 4	Step 5
✓	✓	✓	进行中	待续

魔法时刻: 母串指针**没有回溯**, 子串指针跳到正确位置!

KMP字符串匹配算法

KMP算法是如何避免母串指针回溯的?

Step 5: 另一种情况

母串:

aabcdaakg……

从头开始
next[0]=-1

acabacaef…

情况：当 next[0] = -1 时（哨兵），说明子串第一个字符就不匹配，需要**母串指针+1，子串指针回0**

Step 1 ✓ | Step 2 ✓ | Step 3 ✓ | Step 4 ✓ | Step 5 完成

总结：KMP算法通过next数组，实现了**母串指针永不回溯**！

求next列表

示例分析

字符串示例

字符串 b:

a c a b a c a e f

next数组:

-1	0	0	1	0	1	2	3	0
a	c	a	b	a	c	a	e	f

next[i] 的含义

next[i]: 字符 b[i] 的**最长前后缀长度**

求next列表

KMP算法Python实现

Python代码实现

```
def kmp(a, b, Next):  # a是母串，b是子串
    La, Lb = len(a), len(b)
    pa = pb = 0  # 母串指针和子串指针

    while pa < La and pb < Lb:
        if pb == -1 or a[pa] == b[pb]:
            # pb==-1说明b[0]失配,因为只有next[0]才为-1
            pa, pb = pa + 1, pb + 1
        else:
            pb = Next[pb]  # 执行次数不会多于pa+1的执行次数

    if pb == Lb:
        return pa - pb
    return -1
```

时间复杂度

O(La)

在Next列表已经算出的情况下

关键性质

`pb = Next[pb]` 的执行次数**不会多于** `pa + 1` 的执行次数

求next列表

递推求解方法

🔑关键问题

求子串 b 的next列表

⏩递推思路

已知 next[0], next[1], ..., next[i] ， 如何求 next[i+1] ?

设 next[i] = k ，则若 b[i] = b[k] ，则 next[i+1] = k + 1

b0b1b2...bk-1bk...bi-1bibi+1

- 若 next[i] = k ，说明 b0b1b2...bk-1 = bi-k...bi-1
- 此时若 b[i] = b[k] ，则 b0b1b2...bk-1bk = bi-k...bi-1bi
- 即 next[i+1] = k + 1

求next列表

不匹配情况的处理

❓问题

设 next[i] = k ，则若 b[i] != b[k] 则?

💡解决思路

b[i+1] 的最长前后缀，必然是**以b[i]的某个前后缀加上b[i]构成**

1. 如果 b[i] 的最长前后缀(长度 k)，加上 b[i] ，**不能**构成 b[i+1] 的最长前后缀
2. 则要考虑 b[i] 的**次长前后缀**（长度为 next[k] ），能否加上 b[i] ，构成 b[i+1] 的最长前后缀
3. 次长的不行，则考虑**次次长前后缀**(长度为 next[next[k]])……
4. 最终看 b[0] 能否成为 b[i+1] 最长前后缀

求next列表

countNext函数实现

Python代码实现

```
def countNext(b):
    i, k, Lb = 0, -1, len(b)
    Next = [-1 for i in range(Lb)]

    while i < Lb - 1:
        if k == -1 or b[i] == b[k]:
            Next[i+1] = k + 1
            i, k = i + 1, k + 1
        else:
            k = Next[k]

    return Next
```

时间复杂度

O(len(b))

线性时间复杂度

关键变量

- i : 当前处理的字符索引
- k : 最长前后缀长度
- Next : next数组

KMP性能分析

因此，KMP算法在字母表大小为常数的情况下，其运行时间为 **O(m+n)**

该算法模拟了将**非确定有限自动机（NFA）**转换为**确定有限自动机（DFA）**的过程， 但由于它在每个模式位置构建了一个**失败表**（用于记录最长的既是前缀又是后缀的子串）， 而不是对所有前缀进行编码，因此**无需额外的预处理时间**

线性时间
O(m+n)

空间效率
O(m)

最优算法
理论最优

State machine of "nano"
Image from http://www.zrzahid.com/linear-time-string-matching-using-kmp-matching-algorithm/