

Hash Table & KMP

Updated 1434 GMT+8 May 20, 2025

2025 spring, Compiled by Hongfei Yan

一、散列表的查找

1.1 散列表的基本概念

参考：数据结构（C语言版 第2版）（严蔚敏），第7章 查找

前6周讨论了基于线性结构、树表结构的查找方法，这类查找方法都是以关键字的比较为基础的。

线性表是一种具有相同数据类型的有限序列，其特点是每个元素都有唯一的直接前驱和直接后继。换句话说，线性表中的元素之间存在明确的线性关系，每个元素都与其前后相邻的元素相关联。

线性结构是数据结构中的一种基本结构，它的特点是数据元素之间存在一对一的关系，即除了第一个元素和最后一个元素以外，其他每个元素都有且仅有一个直接前驱和一个直接后继。线性结构包括线性表、栈、队列和串等。

因此，线性表是线性结构的一种具体实现，它是一种最简单和最常见线性结构。

在查找过程中只考虑各元素关键字之间的相对大小，记录在存储结构中的位置和其关键字无直接关系，其查找时间与表的长度有关，特别是当结点个数很多时，查找时要大量地与无效结点的关键字进行比较，致使查找速度很慢。如果能在元素的存储位置和其关键字之间建立某种直接关系，那么在进行查找时，就无需做比较或做很少次的比较，按照这种关系直接由关键字找到相应的记录。这就是散列查找法（Hash Search）的思想，它通过对元素的关键字值进行某种运算，直接求出元素的地址，即使用关键字到地址的直接转换方法，而不需要反复比较。因此，散列查找法又叫杂凑法或散列法。

下面给出散列法中常用的几个术语。

- (1) **散列函数和散列地址**：在记录的存储位置 p 和其关键字 key 之间建立一个确定的对应关系 H ，使 $p = H(key)$ ，称这个对应关系 H 为散列函数， p 为散列地址。
- (2) **散列表**：一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常散列表的存储空间是一个一维数组，散列地址是数组的下标。
- (3) **冲突和同义词**：对不同的关键字可能得到同一散列地址,即 $key1 \neq key2$, 而 $H(key1) = H(key2)$ 这种现象称为**冲突**。具有相同函数值的关键字对该散列函数来说称作同义词， $key1$ 与 $key2$ 互称为**同义词**。

例如，在Python语言中，可以针对给定的关键字集合建立一个散列表。假设有一个关键字集合为 s_1 ，其中包括关键字 `main`, `int`, `float`, `while`, `return`, `break`, `switch`, `case`, `do`。为了构建散列表，可以定义一个长度为26的散列表 HT ，其中每个元素是一个长度为8的字符数组。假设我们采用散列函数 $H(key)$ ，该函数将关键字 key 中的第一个字母转换为字母表 $\{a, b, \dots, z\}$ 中的序号（序号范围为0~25），即 $H(key) = ord(key[0]) - ord('a')$ 。根据此散列函数构造的散列表 HT 如下所示：

```
1 HT = [['' for _ in range(8)] for _ in range(26)]
```

其中，假设关键字 key 的类型为长度为8的字符数组。根据给定的关键字集合和散列函数，可以将关键字插入到相应的散列表位置。

表1

0	1	2	3	4	5	...	8	...	12	...	17	18	...	22	...	25
	break	case	do		float		int		main		return	switch		while		

假设关键字集合扩充为:

$S_2 = S_1 + \{\text{short, default, double, static, for, struct}\}$

如果散列函数不变,新加入的七个关键字经过计算得到: $H(\text{short})=H(\text{static})=H(\text{struct})=18$, $H(\text{default})=H(\text{double})=3$, $H(\text{for})=5$, 而 18、3 和 5 这几个位置均已存放相应的关键字,这就发生了冲突现象,其中 switch、short、static 和 struct 称为同义词; float 和 for 称为同义词; do、default 和 double 称为同义词。

集合 S_2 中的关键字仅有 15 个, 仔细分析这 15 个关键字的特性, 应该不难构造一个散列函数避免冲突。但在实际应用中, 理想化的、不产生冲突的散列函数极少存在, 这是因为通常散列表中关键字的取值集合远远大于表空间的地址集。例如, 高级语言的编译程序要对源程序中的标识符建立一张符号表进行管理, 多数都采取散列表。在设定散列函数时, 考虑的查找关键字集合应包含所有可能产生的关键字, 不同的源程序中使用的标识符一般也不相同, 如果此语言规定标识符为长度不超过 8 的、字母开头的字母数字串, 字母区分大小写, 则标识符取值集合的大小为:

$$C_{52}^1 \times C_{62}^7 \times 7! = 1.09 \times 10^{12}$$

而一个源程序中出现的标识符是有限的, 所以编译程序将散列表的长度设为 1000 足矣。于是要将多达 10^{12} 个可能的标识符映射到有限的地址上, 难免产生冲突。通常, 散列函数是一个多对一的映射, 所以冲突是不可避免的, 只能通过选择一个“好”的散列函数使得在一定程度上减少冲突。而一旦发生冲突, 就必须采取相应措施及时予以解决。

综上所述, 散列查找法主要研究以下两方面的问题:

- (1) 如何构造散列函数;
- (2) 如何处理冲突。

1.2 散列函数的构造方法

构造散列函数的方法很多, 一般来说, 应根据具体问题选用不同的散列函数, 通常要考虑以下因素:

- (1) 散列表的长度;
- (2) 关键字的长度;
- (3) 关键字的分布情况;
- (4) 计算散列函数所需的时间;
- (5) 记录的查找频率。

构造一个“好”的散列函数应遵循以下两条原则: (1) 函数计算要简单, 每一关键字只能有一个散列地址与之对应; (2) 函数的值域需在表长的范围内, 计算出的散列地址的分布应均匀, 尽可能减少冲突。下面介绍构造散列函数的几种常用方法。

1.2.1 数字分析法

如果事先知道关键字集合, 且每个关键字的位数比散列表的地址码位数多, 每个关键字由 n 位数组成, 如 k_1k_2, \dots, k_n , 则可以从关键字中提取数字分布比较均匀的若干位作为散列地址。

1.2.3 折叠法将

关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列地址，这种方法称为**折叠法**。根据数位叠加的方式，可以把折叠法分为移位叠加和边界叠加两种。移位叠加是将分割后每一部分的最低位对齐，然后相加；边界叠加是将两个相邻的部分沿边界来回折叠，然后对齐相加。

例如，当散列表长为 1000 时，关键字 `key=45387765213`，从左到右按3 位数一段分割，可以得到 4个部分:453、877、652、13。分别采用移位叠加和边界叠加，求得散列地址为 995 和914，如图 1 所示。

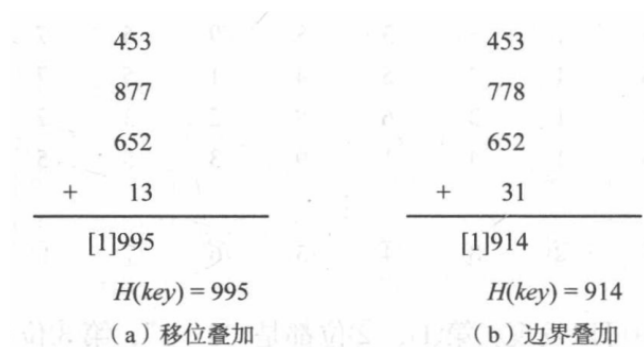


图 1 由折叠法求得散列地址

折叠法的适用情况：适合于散列地址的位数较少，而关键字的位数较多，且难于直接从关键字中找到取值较分散的几位。

1.2.4 除留余数法

假设散列表表长为 m ，选择一个不大于 m 的数 p ，用 p 去除关键字，除后所得余数为散列地址，即 $H(key) = key \% p$

这个方法的关键是选取适当的 p ，一般情况下，可以**选 p 为小于表长的最大质数**。例如，表长 $m=100$ ，可取 $p=97$ 。

除留余数法计算简单，适用范围非常广，是最常用的构造散列函数的方法。它不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模，这样能够保证散列地址一定落在散列表的地址空间中。

1.2.5 总结哈希函数的选取

取自 刘汝家、黄亮《算法艺术与信息学竞赛》2004年，P96。

对于数值来说：

- 1) 直接取余数（一般选取的除数，最好是个质数，这样冲突少些）。**容易产生分布不均匀的情况**。
- 2) 平方取中法：即计算关键值平方，再取中间 r 位形成一个大小为 2^r 的表。好很多，因为几乎所有位都对结果产生了影响。但是它的**计算量大，一般也较少使用**。

对于字符串，常用方法有：

- 1) 折叠法：即把所有字符的ASCII码加起来。
- 2) 采用ELFhash函数，即（它用于UNIX的“可执行链接格式，ELF”中）。这是一个有用的HASH函数，它对长短字符串都很有效。推荐把它作为字符串的HASH函数。

下面是基于ELFhash算法的字符串哈希函数的Python代码示例：

```
1 def ELFhash(string):
2     hash_value = 0
3     x = 0
4
5     for char in string:
6         hash_value = (hash_value << 4) + ord(char)
7         x = hash_value & 0xF0000000
8
9         if x != 0:
10             hash_value ^= (x >> 24)
11
12         hash_value &= ~x
13
14     return hash_value
15
16 # 测试
17 string = "Hello World"
18 print("Hash value of '{}' is: {}".format(string, ELFhash(string)))
19
20 # output: Hash value of 'Hello World' is: 186219373
```

在这个代码中，`ELFhash` 函数接受一个字符串作为参数，并计算其哈希值。它使用了ELFhash算法，通过遍历字符串的每个字符，将其ASCII码值加入到哈希值中，并进行一系列位运算来得到最终的哈希值。

这段代码定义并实现了一个经典的哈希函数：**ELF Hash (Extended Linear Feedback Hash)**，主要用于将字符串映射为整数哈希值。ELF Hash 最初用于 Unix 的 ELF 格式，也出现在很多编译器或链接器中。以下是详细解读：

初始化变量

```
1 hash_value = 0
2 x = 0
```

- `hash_value`: 存储当前计算的哈希值。
- `x`: 用于临时存储高位掩码，用来防止哈希溢出。

主循环

```
1 for char in string:
2     hash_value = (hash_value << 4) + ord(char)
```

- 每次迭代，将哈希值左移4位（相当于乘以16），并加上当前字符的 ASCII 值（`ord(char)`）。
- 这种方式让哈希值对字符的顺序和内容都很敏感。

```
1 x = hash_value & 0xF0000000
```

- 提取哈希值的高4位（前4个十六进制位），用于判断是否存在溢出风险。

```
1 if x != 0:
2     hash_value ^= (x >> 24)
```

- 如果高位 `x` 不为0，右移24位后与当前哈希值异或，混淆高位对低位的影响，防止模式重复。

```
1 hash_value &= ~x
```

- 清除高位，防止哈希值超出范围（限制在28位以内），提高分布的均匀性。

总结

ELF Hash 的特点是：

- 高效；
- 利用位操作进行散列值的混合；
- 对字符串内容非常敏感（即使轻微变化也会导致哈希值大变）；
- 适合用于符号表、哈希表等场景。

1.3 处理冲突的方法

选择一个“好”的散列函数可以在一定程度上减少冲突，但在实际应用中，很难完全避免发生冲突，所以选择一个有效的处理冲突的方法是散列法的另一个关键问题。创建散列表和查找散列表都会遇到冲突，两种情况下处理冲突的方法应该一致。下面以创建散列表为例，来说明处理冲突的方法。

处理冲突的方法与散列表本身的组织形式有关。按组织形式的不同，通常分两大类：**开放地址法和链地址法**。

1.3.1 开放地址法（闭散列法）

开放地址法的基本思想是：把记录都存储在散列表数组中，当某一记录关键字 `key` 的初始散列地址 $H_0 = H(\text{key})$ 发生冲突时，以 H_0 为基础，采取合适方法计算得到另一个地址 H_1 ，如果 H_1 仍然发生冲突，以 H_1 为基础再求下一个地址 H_2 ，若 H_2 仍然冲突，再求得 H_3 。依次类推，直至 H_k 不发生冲突为止，则 H_k 为该记录在表中的散列地址。

这种方法在寻找“下一个”空的散列地址时，原来的数组空间对所有的元素都是开放的所以称为开放地址法。通常把寻找“下一个”空位的过程称为探测，上述方法可用如下公式表示：

$$Hi = (H(key) + di)$$

其中，H(key)为散列函数，m 为散列表表长，d为增量序列。根据d取值的不同，可以分为以下3种探测方法。

(1) 线性探测法

$$di = 1, 2, 3, \dots, m-1$$

这种探测方法可以将散列表假想成一个循环表，发生冲突时，从冲突地址的下一单元顺序寻找空单元，如果到最后一个位置也没找到空单元，则回到表头开始继续查找，直到找到一个空位，就把此元素放入此空位中。如果找不到空位，则说明散列表已满，需要进行溢出处理。

(2) 二次探测法

$$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \dots, +k^2, -k^2 (k \leq m/2)$$

(3) 伪随机探测法

di = 伪随机数序列

例如，散列表的长度为 11，散列函数 $H(key)=key\%11$ ，假设表中已填有关键字分别为 17、60、29 的记录，如图2(a)所示。现有第四个记录，其关键字为38，由散列函数得到散列地址为 5，产生冲突。

若用线性探测法处理时，得到下一个地址6，仍冲突；再求下一个地址7，仍冲突；直到散列地址为8的位置为“空”时为止，处理冲突的过程结束，38填入散列表中序号为8的位置，如图2(b)所示。

若用二次探测法，散列地址5冲突后，得到下一个地址6，仍冲突；再求得下一个地址 4，无冲突，38填入序号为4的位置，如图 2(c)所示。

若用伪随机探测法，假设产生的伪随机数为9，则计算下一个散列地址为 $(5+9)\%11=3$ ，所以38 填入序号为3 的位置，如图 2(d)所示。

0	1	2	3	4	5	6	7	8	9	10
					60	17	29			

(a) 插入前

					60	17	29	38		
--	--	--	--	--	----	----	----	----	--	--

(b) 线性探测法

				38	60	17	29			
--	--	--	--	----	----	----	----	--	--	--

(c) 二次探测法

			38		60	17	29			
--	--	--	----	--	----	----	----	--	--	--

(d) 伪随机探测法，伪随机数序列为 9，...

图 2 用开放地址法处理冲突时，关键字为38的记录插入前后的散列表

从上述线性探测法处理的过程中可以看到一个现象：当表中 i , $i+1$, $i+2$ 位置上已填有记录时，下一个散列地址为 i 、 $i+1$ 、 $i+2$ 和 $i+3$ 的记录都将填入 $i+3$ 的位置，这种在处理冲突过程中发生的两个第一个散列地址不同的记录争夺同一个后继散列地址的现象称作“二次聚集”(或称作“堆积”)，即在处理同义词的冲突过程中又添加了非同义词的冲突。

可以看出，上述三种处理方法各有优缺点。**线性探测法的优点是**：只要散列表未填满，总能找到一个不发生冲突的地址。**缺点是**：会产生“二次聚集”现象。而**二次探测法和伪随机探测法的优点是**：可以避免“二次聚集”现象。**缺点**也很显然：不能保证一定找到不发生冲突的地址。

1.3.2 链地址法（开散列法）

链地址法的基本思想是：把具有相同散列地址的记录放在同一个单链表中，称为同义词链表。有 m 个散列地址就有 m 个单链表，同时用数组 $HT[0...m-1]$ 存放各个链表的头指针，凡是散列地址为 i 的记录都以结点方式插入到以 $HT[i]$ 为头结点的单链表中。

【例】已知一组关键字为 (19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)，设散列函数 $H(key) = key \% 13$ ，用链地址法处理冲突，试构造这组关键字的散列表。

由散列函数 $H(key) = key \% 13$ 得知散列地址的值域为 0~12，故整个散列表有 13 个单链表组成，用数组 $HT[0..12]$ 存放各个链表的头指针。如散列地址均为 1 的同义词 14、1、27、79 构成一个单链表，链表的头指针保存在 $HT[1]$ 中，同理，可以构造其他几个单链表，整个散列表的结构如图 3 所示。

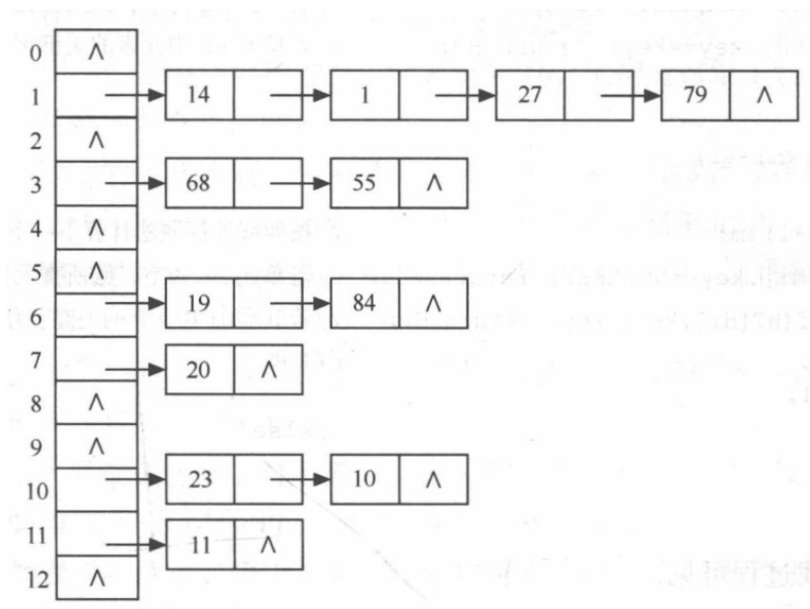


图 3 用链地址法处理冲突时的散列表

这种构造方法在具体实现时，依次计算各个关键字的散列地址，然后根据散列地址将关键字插入到相应的链表中。

处理散列表冲突的常见方法包括以下几种：

1. 链地址法 (Chaining)：使用链表来处理冲突。每个散列桶 (哈希桶) 中存储一个链表，具有相同散列值的元素会链接在同一个链表上。
2. 开放地址法 (Open Addressing)：
 - 线性探测 (Linear Probing)：如果发生冲突，就线性地探测下一个可用的槽位，直到找到一个空槽位或者达到散列表的末尾。
 - 二次探测 (Quadratic Probing)：如果发生冲突，就使用二次探测来查找下一个可用的槽位，避免线性探测中的聚集效应。
 - 双重散列 (Double Hashing)：如果发生冲突，就使用第二个散列函数来计算下一个槽位的位置，直到找到一个空槽位或者达到散列表的末尾。
3. 再散列 (Rehashing)：当散列表的**装载因子 (load factor)** 超过一定阈值时，进行扩容操作，重新调整散列函数和散列桶的数量，以减少冲突的概率。
4. 建立公共溢出区 (Public Overflow Area)：将冲突的元素存储在一个公共的溢出区域，而不是在散列桶中。在进行查找时，需要遍历溢出区域。

这些方法各有优缺点，适用于不同的应用场景。选择合适的处理冲突方法取决于数据集的特点、散列表的大小以及性能需求。

双重散列 (Double Hashing) 是一种处理散列表冲突的方法，它使用两个散列函数来计算冲突时下一个可用的槽位位置。下面是双重散列的一个示例：

假设有一个散列表，大小为10，使用双重散列来处理冲突。我们定义两个散列函数：

1. 第一个散列函数 `hash1(key)`：将关键字 `key` 转换为散列值，使用一种合适的散列算法，比如取模运算。
2. 第二个散列函数 `hash2(key)`：将关键字 `key` 转换为一个正整数，在本例中，我们使用简单的散列函数 `hash2(key) = 7 - (key % 7)`。

现在，通过以下步骤来插入一个关键字 `key` 到散列表中：

1. 使用第一个散列函数 `hash1(key)` 计算关键字 `key` 的初始散列值 `hash_value = hash1(key)`。
2. 如果散列表中的槽位 `hash_value` 是空的，则将关键字 `key` 插入到该槽位中。
3. 如果槽位 `hash_value` 不为空，表示发生了冲突。在这种情况下，我们使用**第二个散列函数 `hash2(key)` 来计算关键字 `key` 的步长 (step)**。
4. 通过计算 `step = hash2(key)`，我们将跳过 `step` 个槽位，继续在散列表中查找下一个槽位。
5. 重复步骤 3 和步骤 4，直到找到一个空槽位，将关键字 `key` 插入到该槽位中。

如果散列表已满而且仍然无法找到空槽位，那么插入操作将失败。

双重散列使用两个散列函数来计算步长，这样可以**避免线性探测中的聚集效应**，提高散列表的性能。每个关键字都有唯一的步长序列，因此它可以在散列表中的不同位置进行探测，减少冲突的可能性。

笔试例题：

有一个散列表如下图所示，其散列函数为 $h(key)=key \bmod 13$ ，该散列表使用再散列函数 $H2(key)=key \bmod 3$ 解决碰撞，问从表中检索出关键码 38 需进行几次比较（B）。

0	1	2	3	4	5	6	7	8	9	10	11	12
26	38			17			33		48			25


A: 1 B: 2 C: 3 D: 4

 第一步：计算关键码 38 的初始散列地址

使用 $h(38) = 38 \bmod 13 = 12$

所以要去下标为 **12** 的位置找它。

查看表中下标 12 的值是 **25**，不是 38。说明发生了碰撞，继续查找。

 第二步：使用再散列函数解决冲突

当前哈希地址为 $pos = 12$

再散列步长为：

```
1 | step = H2(38) = 38 mod 3 = 2
```

所以每次探测的公式为：

```
1 | pos = (pos + step) % size_of_table
```

散列表长度是 13。

我们开始探测：

 探测过程（线性再散列）

1. 初始 $pos = 12$
 - 表中值为 $25 \neq 38 \rightarrow$ 比较 1 次
 - 更新 $pos = (12 + 2) \% 13 = 1$
2. 新 $pos = 1$
 - 表中值为 $38 == 38 \rightarrow$ 找到！比较 2 次

 最终结论：

- 在检索过程中进行了 **2 次比较**。
- 正确答案是：**B: 2**

1.5 程序实现

字符串构建简单的散列函数。针对异序词，这个散列函数总是得到相同的散列值。要弥补这一点，可以用字符位置作为权重因子，

```
1 def hash(a_string, table_size):
2     sum = 0
3     for pos in range(len(a_string)):
4         sum = sum + (pos+1) * ord(a_string[pos])
5
6     return sum%table_size
7
8 print(hash('abba', 11))
```

使用两个列表创建HashTable类，以此实现映射抽象数据类型。其中，名为slots的列表用于存储键，名为data的列表用于存储值。两个列表中的键与值一一对应。在本节的例子中，散列表的初始大小是11。尽管初始大小可以任意指定，但选用一个素数很重要，这样做可以尽可能地提高冲突处理算法的效率。

hashfunction实现了简单的取余函数。处理冲突时，采用“加1”再散列函数的线性探测法。put函数假设，除非键已经在self.slots中，否则总是可以分配一个空槽。该函数计算初始的散列值，如果对应的槽中已有元素，就循环运行rehash函数，直到遇见一个空槽。如果槽中已有这个键，就用新值替换旧值。

同理，get函数也先计算初始散列值。如果值不在初始散列值对应的槽中，就使用rehash确定下一个位置。注意，第46行确保搜索最终一定能结束，因为不会回到初始槽。如果遇到初始槽，就说明已经检查完所有可能的槽，并且元素必定不存在。

HashTable类的最后两个方法提供了额外的字典功能。重载 `__getitem__` 和 `__setitem__`，以通过[]进行访问。这意味着创建HashTable类之后，就可以使用熟悉的索引运算符了。

```
1 class HashTable:
2     def __init__(self):
3         self.size = 11
4         self.slots = [None] * self.size
5         self.data = [None] * self.size
6
7     def put(self, key, data):
8         hashvalue = self.hashfunction(key, len(self.slots))
9
10        if self.slots[hashvalue] == None:
11            self.slots[hashvalue] = key
12            self.data[hashvalue] = data
13        else:
14            if self.slots[hashvalue] == key:
15                self.data[hashvalue] = data #replace
16            else:
17                nextslot = self.rehash(hashvalue, len(self.slots))
```

```

18         while self.slots[nextslot] != None and self.slots[nextslot] !=
key:
19             nextslot = self.rehash(nextslot,len(self.slots))
20
21             if self.slots[nextslot] == None:
22                 self.slots[nextslot] = key
23                 self.data[nextslot] = data
24             else:
25                 self.data[nextslot] = data #replace
26
27     def hashfunction(self,key,size):
28         return key%size
29
30     def rehash(self,oldhash,size):
31         return (oldhash+1)%size
32
33     def get(self,key):
34         startslot = self.hashfunction(key,len(self.slots))
35
36         data = None
37         stop = False
38         found = False
39         position = startslot
40         while self.slots[position] != None and not found and not stop:
41             if self.slots[position] == key:
42                 found = True
43                 data = self.data[position]
44             else:
45                 position=self.rehash(position,len(self.slots))
46                 if position == startslot:
47                     stop = True
48         return data
49
50     def __getitem__(self,key):
51         return self.get(key)
52
53     def __setitem__(self,key,data):
54         self.put(key,data)
55
56
57 H=HashTable()
58 H[54]="cat"
59 H[26]="dog"
60 H[93]="lion"
61 H[17]="tiger"
62 H[77]="bird"
63 H[31]="cow"
64 H[44]="goat"
65 H[55]="pig"
66 H[20]="chicken"
67 print(H.slots)
68 print(H.data)

```

```

69
70
71 print(H[20])
72 print(H[17])
73
74 H[20] = 'duck'
75 print(H[20])
76
77 print(H.data)
78
79 print(H[99])
80
81 """
82 [77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
83 ['bird', 'goat', 'pig', 'chicken', 'dog', 'lion', 'tiger', None, None, 'cow',
84  'cat']
85 chicken
86 tiger
87 duck
88 ['bird', 'goat', 'pig', 'duck', 'dog', 'lion', 'tiger', None, None, 'cow', 'cat']
89 None
90 """

```

注意，在11个槽中，有9个被占用了。占用率被称作**载荷因子（load factor）**，记作 λ ，定义如下。

$$\lambda = \frac{\text{元素个数}}{\text{散列表大小}}$$

在本例中, $\lambda = \frac{9}{11}$.

1.6 散列表的查找

在散列表上进行查找的过程和创建散列表的过程基本一致。

1.7 编程题目

练习17968: 整型关键字的散列映射

<http://cs101.openjudge.cn/practice/17968/>

给定一系列整型关键字和素数P，用除留余数法定义的散列函数H (key)=key%M，将关键字映射到长度为M的散列表中，用线性探查法解决冲突

输入

输入第一行首先给出两个正整数N ($N \leq 1000$) 和M ($M \geq N$ 的最小素数)，分别为待插入的关键字总数以及散列表的长度。

第二行给出N个整型的关键字。数字之间以空格分隔。

输出

在一行内输出每个整型关键字的在散列表中的位置。数字间以空格分隔。

样例输入

```
1 | 4 5
2 | 24 13 66 77
```

样例输出

```
1 | 4 3 1 2
```

这个题目的输入数据可能不是标准形式，特殊处理，整体读入 `sys.stdin.read`

```
1 def insert_hash_table(keys, M):
2     table = [0.5] * M # 用 0.5 表示空位
3     result = []
4
5     for key in keys:
6         index = key % M
7         i = index
8
9         while True:
10            if table[i] == 0.5 or table[i] == key:
11                result.append(i)
12                table[i] = key
13                break
14            i = (i + 1) % M
15
16     return result
17
18 # 使用标准输入读取数据
19 import sys
20 input = sys.stdin.read
21 data = input().split()
22
23 N = int(data[0])
24 M = int(data[1])
25 keys = list(map(int, data[2:2 + N]))
26
27 positions = insert_hash_table(keys, M)
28 print(*positions)
29
```


练习17975: 用二次探查法建立散列表

<http://cs101.openjudge.cn/practice/17975/>

给定一系列整型关键字和素数P，用除留余数法定义的散列函数 $H(\text{key}) = \text{key} \% M$ ，将关键字映射到长度为M的散列表中，用二次探查法解决冲突。

本题不涉及删除，且保证表长不小于关键字总数的2倍，即没有插入失败的可能。

输入

输入第一行首先给出两个正整数N ($N \leq 1000$) 和M (一般为 $\geq 2N$ 的最小素数)，分别为待插入的关键字总数以及散列表的长度。

第二行给出N个整型的关键字。数字之间以空格分隔。

输出

在一行内输出每个整型关键字的在散列表中的位置。数字间以空格分隔。

样例输入

```
1 | 5 11
2 | 24 13 35 15 14
```

样例输出

```
1 | 2 3 1 4 7
```

提示

探查增量序列依次为： 1^2 ， -1^2 ， 2^2 ， -2^2 ， \dots ， i^2 表示平方

需要用这样接收数据。因为输入数据可能分行了，不是题面描述的形式。OJ上面有的题目是给C++设计的，细节考虑不周全。

```
1 | import sys
2 | input = sys.stdin.read
3 | data = input().split()
4 | index = 0
5 | n = int(data[index])
6 | index += 1
7 | m = int(data[index])
8 | index += 1
9 | num_list = [int(i) for i in data[index:index+n]]
```

```
1 | # 2200015507 王一粟
```

```

2  # n, m = map(int, input().split())
3  # num_list = [int(i) for i in input().split()]
4  import sys
5  input = sys.stdin.read
6  data = input().split()
7  index = 0
8  n = int(data[index])
9  index += 1
10 m = int(data[index])
11 index += 1
12 num_list = [int(i) for i in data[index:index+n]]
13
14 mylist = [0.5] * m
15
16 def generate_result():
17     for num in num_list:
18         pos = num % m
19         current = mylist[pos]
20         if current == 0.5 or current == num:
21             mylist[pos] = num
22             yield pos
23         else:
24             sign = 1
25             cnt = 1
26             while True:
27                 now = pos + sign * (cnt ** 2)
28                 current = mylist[now % m]
29                 if current == 0.5 or current == num:
30                     mylist[now % m] = num
31                     yield now % m
32                     break
33                 sign *= -1
34                 if sign == 1:
35                     cnt += 1
36
37 result = generate_result()
38 print(*result)

```

二、KMP

https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

在计算机科学中，**Knuth-Morris-Pratt 算法**（简称 **KMP 算法**）是一种字符串查找算法。该算法通过观察到这样一个关键点来查找主文本字符串 **s** 中是否存在一个“单词”或子字符串 **w**：当发生字符不匹配时，单词 **w** 本身已经包含了足够的信息，可以确定下一次可能的匹配位置，从而跳过对先前已匹配字符的重复检查。

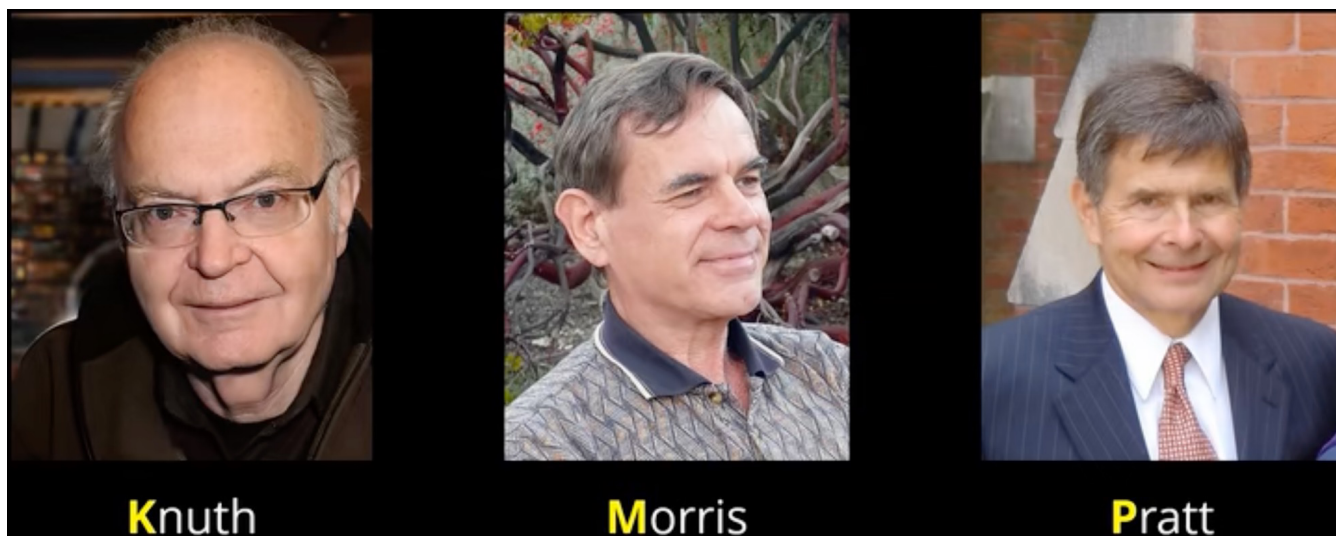
KMP 利用已经匹配的信息避免无谓的重复比较，实现了 $O(n + m)$ 的字符串匹配效率。

该算法最初由 James H. Morris 构思，并在几周后被 Donald Knuth 从自动机理论的角度独立发现。Morris 和 Vaughan Pratt 于 1970 年发表了一篇技术报告。三人于 1977 年联合发表了这一算法。与此同时，在 1969 年，Matiyasevich 在研究一个基于二元字母表的字符串模式匹配识别问题时，利用二维图灵机设计出了一个类似的算法。这是第一个实现线性时间复杂度的字符串匹配算法。

In computer science, the **Knuth-Morris-Pratt algorithm** (or **KMP algorithm**) is a string-searching algorithm that searches for occurrences of a "word" **w** within a main "text string" **s** by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

The algorithm was conceived by James H. Morris and independently discovered by Donald Knuth "a few weeks later" from automata theory. Morris and Vaughan Pratt published a technical report in 1970. The three also published the algorithm jointly in 1977. Independently, in 1969, Matiyasevich discovered a similar algorithm, coded by a two-dimensional Turing machine, while studying a string-pattern-matching recognition problem over a binary alphabet. This was the first linear-time algorithm for string matching.

KMP Algorithm for Pattern Searching



FAST PATTERN MATCHING IN STRINGS*

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Key words. pattern, string, text-editing, pattern-matching, trie memory, searching, period of a string, palindrome, optimum algorithm, Fibonacci string, regular expression

Text-editing programs are often required to search through a string of characters looking for instances of a given “pattern” string; we wish to find all positions, or perhaps only the leftmost position, in which the pattern occurs as a contiguous substring of the text. For example, *catenary* contains the pattern *ten*, but we do not regard *canary* as a substring.

Generative AI is experimental. Info quality may vary.

Knuth-Morris-Pratt (KMP) 算法是一种用于在文本字符串中查找单词的计算机科学算法。该算法从左到右依次比较字符。

当出现字符不匹配时，算法会使用一个预处理表（称为“前缀表”）来跳过不必要的字符比较。

KMP 算法的工作原理

- 该算法会在模式串中寻找被称为 LPS（最长前缀后缀）的重复子串，并将这些 LPS 信息存储在一个数组中。
- 算法从左到右逐个比较字符。
- 当发生不匹配时，算法使用一个预处理好的表（称为“前缀表”）来跳过字符比较。
- 算法预先计算一个前缀函数，帮助确定每次发生不匹配时可以在模式串中跳过多少字符。
- 相比暴力搜索方法，KMP 算法利用先前比较的信息，避免了不必要的字符比较，从而提高了效率。

KMP 算法的优势

- KMP 算法可以高效地帮助你在大量文本中找到特定的模式串。
- KMP 算法可以使你的文本编辑任务更快、更高效。
- KMP 算法保证了**100%**的可靠性。

The Knuth–Morris–Pratt (KMP) algorithm is **a computer science algorithm that searches for words in a text string**. The algorithm compares characters from left to right.

When a mismatch occurs, the algorithm uses a preprocessed table called a "Prefix Table" to skip character comparisons.

How the KMP algorithm works

- The algorithm finds repeated substrings called LPS in the pattern and stores LPS information in an array.
- The algorithm compares characters from left to right.
- When a mismatch occurs, the algorithm uses a preprocessed table called a "Prefix Table" to skip character comparisons.
- The algorithm precomputes a prefix function that helps determine the number of characters to skip in the pattern whenever a mismatch occurs.
- The algorithm improves upon the brute force method by utilizing information from previous comparisons to avoid unnecessary character comparisons.

Benefits of the KMP algorithm

- The KMP algorithm efficiently helps you find a specific pattern within a large body of text.
- The KMP algorithm makes your text editing tasks quicker and more efficient.
- The KMP algorithm guarantees 100% reliability.

Preprocessing Overview:

- KMP algorithm preprocesses `pat[]` and constructs an auxiliary **`lps[]`** of size **`m`** (same as the size of the pattern) which is used to skip characters while matching.
- Name **`lps`** indicates the **longest proper prefix** which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC", and "ABC". 真前缀 (proper prefix) 是一个串除该串自身外的其他前缀。
- We search for `lps` in subpatterns. More clearly we **focus on sub-strings of patterns that are both prefix and suffix**.
- For each sub-pattern `pat[0..i]` where `i = 0 to m-1`, `lps[i]` stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern `pat[0..i]`.

LPS表是一个数组，其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。

`lps[i]` = the longest proper prefix of `pat[0..i]` which is also a suffix of `pat[0..i]`.

核心概念：最长前缀后缀 (LPS 表)

- **LPS (Longest Prefix which is also Suffix)** 表：对模式串 `pattern` 的每个前缀子串，记录它的“最长相等前后缀”的长度。
- 它的作用是：当匹配失败时，指针无需回退主串的位置，只需调整模式串的位置即可继续匹配。

Note: `lps[i]` could also be defined as the longest prefix which is also a proper suffix. We need to use it properly in one place to make sure that the whole substring is not considered.

Examples of `lps[]` construction:

For the pattern "AAAA", `lps[]` is [0, 1, 2, 3]

For the pattern "ABCDE", `lps[]` is [0, 0, 0, 0, 0]

For the pattern "AABAACAABAA", `lps[]` is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern "AAACAAAAC", `lps[]` is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern "AAABAAA", `lps[]` is [0, 1, 2, 0, 1, 2, 3]

KMP (Knuth-Morris-Pratt) 算法是一种利用双指针和动态规划的字符串匹配算法。

```
1  """
2  compute_lps 函数用于计算模式字符串的LPS表。LPS表是一个数组，
3  其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。
4  该函数使用了两个指针 length 和 i，从模式字符串的第二个字符开始遍历。
5  """
6  def compute_lps(pattern):
7      """
8      计算pattern字符串的最长前缀后缀 (Longest Proper Prefix which is also Suffix) 表
9      :param pattern: 模式字符串
10     :return: lps表
11     """
12
13     m = len(pattern)
14     lps = [0] * m # 初始化lps数组
15     length = 0 # 当前最长前后缀长度
16     for i in range(1, m): # 注意i从1开始, lps[0]永远是0
17         while length > 0 and pattern[i] != pattern[length]:
18             length = lps[length - 1] # 回退到上一个有效前后缀长度
19         if pattern[i] == pattern[length]:
20             length += 1
21         lps[i] = length
22
23     return lps
24
25 def kmp_search(text, pattern):
26     n = len(text)
27     m = len(pattern)
28     if m == 0:
29         return 0
30     lps = compute_lps(pattern)
31     matches = []
32
33     # 在 text 中查找 pattern
34     j = 0 # 模式串指针
35     for i in range(n): # 主串指针
```

```

36         while j > 0 and text[i] != pattern[j]:
37             j = lps[j - 1] # 模式串回退
38         if text[i] == pattern[j]:
39             j += 1
40         if j == m:
41             matches.append(i - j + 1) # 匹配成功
42             j = lps[j - 1] # 查找下一个匹配
43
44     return matches
45
46
47 text = "ABABABABCABABABABCABABABABC"
48 pattern = "ABABCABAB"
49 index = kmp_search(text, pattern)
50 print("pos matched: ", index)
51 # pos matched: [4, 13]
52

```

KMP 是一种利用双指针和动态规划的字符串匹配算法。

- **双指针**：确实存在，一个指针在主串 `text` 上 (`i`)，另一个在模式串 `pattern` 上 (`j`)。
- **动态规划**：广义上讲，LPS 的构造有递推性质，有人将其类比为动态规划的表构建过程（类似于状态转移），但它并不是真正的 DP 算法，只是一个预处理表。

✅ 总结

项目	说明
主要目标	在主串中高效地查找子串
核心工具	LPS 表：用于跳过无效比较
优点	匹配失败时主串不回退，时间复杂度 $O(n + m)$
适用场景	文本搜索、DNA序列分析、代码查重等

关于 kmp 算法中 next 数组的周期性质

参考：<https://www.acwing.com/solution/content/4614/>

引理：

对于某一字符串 $S[1 \sim i]$ ，在它众多的 $next[i]$ 的“候选项”中，如果存在某一个 $next[i]$ ，使得： $i \% (i - next[i]) == 0$ ，那么 $S[1 \sim (i - next[i])]$ 可以为 $S[1 \sim i]$ 的循环元而 $i / (i - next[i])$ 即是它的循环次数 K 。

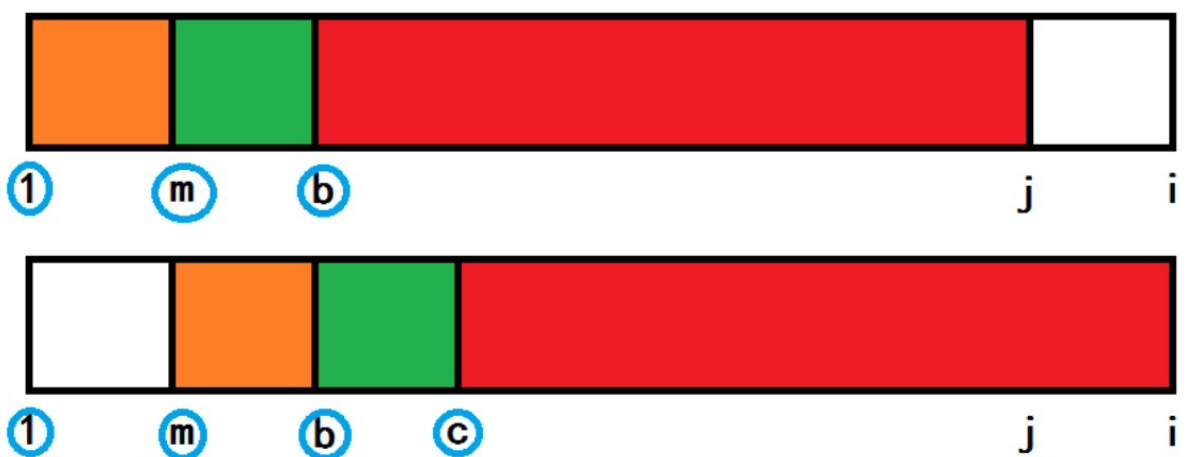
证明如下：



如图, $\text{next}[i] = j$, 由定义得红色部分两个子串完全相同。
 那么有 $S[1 \sim j] = S[m \sim i]$ ($m = i - \text{next}[i]$)。



如果我们在两个子串的前面框选一个长度为 m 的小子串(橙色部分)。
 可以得到: $S[1 \sim m] = S[m \sim b]$ 。

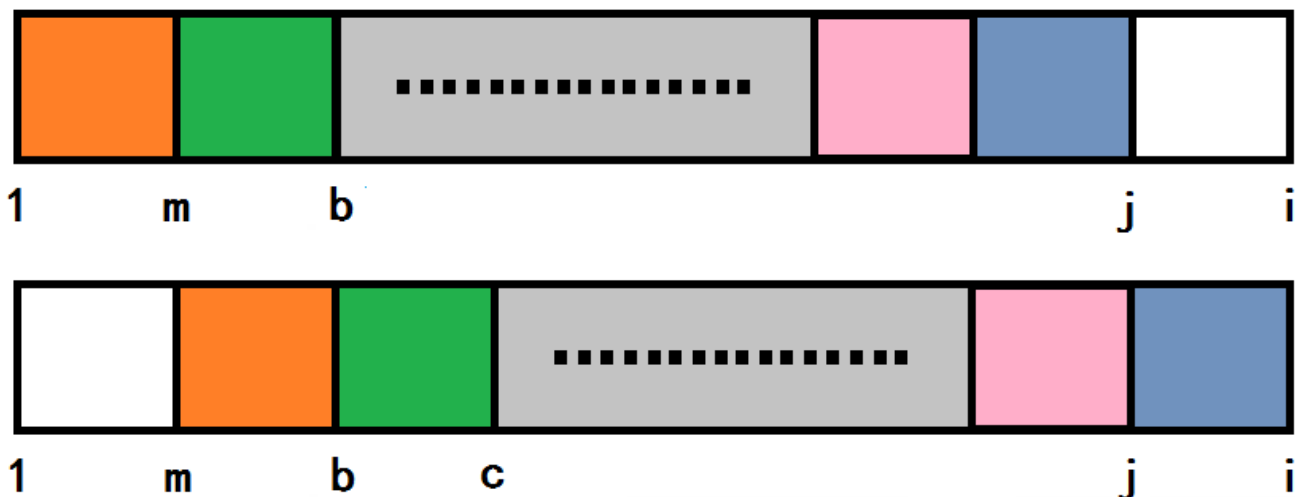


如果在紧挨着之前框选的子串后面再框选一个长度为 m 的小子串(绿色部分), 同样的道理,

可以得到: $S[m \sim b] = S[b \sim c]$

又因为: $S[1 \sim m] = S[m \sim b]$

所以: $S[1 \sim m] = S[m \sim b] = S[b \sim c]$



如果一直这样框选下去，无限推进，总会有一个尽头。当满足 $i \% m == 0$ 时，刚好可以分出 K 个这样的小子串，且形成循环($K=i/m$)。

练习02406: 字符串乘方

<http://cs101.openjudge.cn/practice/02406/>

给定两个字符串a和b,我们定义 $a*b$ 为他们的连接。例如，如果a="abc" 而b="def"，则 $a*b="abcdef"$ 。如果我们将连接考虑成乘法，一个非负整数的乘方将用一种通常的方式定义： $a^0=""$ (空字符串)， $a^{(n+1)}=a*(a^n)$ 。

输入

每一个测试样例是一行可打印的字符作为输入，用s表示。s的长度至少为1，且不会超过一百万。最后的测试样例后面将是一个点号作为一行。

输出

对于每一个s，你应该打印最大的n，使得存在一个a，让 $s = a^n$

样例输入

```
1 | abcd
2 | aaaa
3 | ababab
4 | .
```

样例输出

```
1 | 1
2 | 4
3 | 3
```

提示: 本问题输入量很大，请用scanf代替cin，从而避免超时。

来源: Waterloo local 2002.07.01

```

1  '''
2  gpt
3  使用kmp算法的部分知识，当字符串的长度能被提取的"base字符串"的长度整除时，
4  即可判断s可以被表示为a^n的形式，此时的n就是s的长度除以"base字符串"的长度。
5
6  '''
7
8  import sys
9  while True:
10     s = sys.stdin.readline().strip()
11     if s == '.':
12         break
13     len_s = len(s)
14     next = [0] * len(s)
15     j = 0
16     for i in range(1, len_s):
17         while j > 0 and s[i] != s[j]:
18             j = next[j - 1]
19         if s[i] == s[j]:
20             j += 1
21         next[i] = j
22     base_len = len(s) - next[-1]
23     if len(s) % base_len == 0:
24         print(len_s // base_len)
25     else:
26         print(1)
27

```

练习01961: 前缀中的周期

<http://cs101.openjudge.cn/practice/01961/>

<http://poj.org/problem?id=1961>

For each prefix of a given string S with N characters (each character has an ASCII code between 97 and 126, inclusive), we want to know whether the prefix is a periodic string. That is, for each i ($2 \leq i \leq N$) we want to know the largest $K > 1$ (if there is one) such that the prefix of S with length i can be written as A^K , that is A concatenated K times, for some string A . Of course, we also want to know the period K .

一个字符串的前缀是从第一个字符开始的连续若干个字符，例如"abaab"共有5个前缀，分别是a, ab, aba, abaa, abaab。

我们希望知道一个 N 位字符串 S 的前缀是否具有循环节。换言之，对于每一个从头开始的长度为 i （ i 大于1）的前缀，是否由重复出现的子串 A 组成，即 $AAA...A$ （ A 重复出现 K 次， K 大于1）。如果存在，请找出最短的循环节对应的 K 值（也就是这个前缀串的所有可能重复节中，最大的 K 值）。

输入

输入包括多组测试数据。每组测试数据包括两行。

第一行包括字符串S的长度N ($2 \leq N \leq 1\,000\,000$)。

第二行包括字符串S。

输入数据以只包括一个0的行作为结尾。

输出

对于每组测试数据，第一行输出 "Test case #" 和测试数据的编号。

接下来的每一行，输出前缀长度i和重复次数K，中间用一个空格隔开。前缀长度需要升序排列。

在每组测试数据的最后输出一个空行。

样例输入

```
1 3
2 aaa
3 12
4 aabaabaabaab
5 0
```

样例输出

```
1 Test case #1
2 2 2
3 3 3
4
5 Test case #2
6 2 2
7 6 2
8 9 3
9 12 4
```

【POJ1961】period, <https://www.cnblogs.com/ve-2021/p/9744139.html>

如果一个字符串S是由一个字符串T重复K次构成的，则称T是S的**循环元**。使K出现最大的字符串T称为S的最小循环元，此时的K称为最大循环次数。

现在给定一个长度为N的字符串S，对S的每一个前缀 $S[1 \sim i]$ ，如果它的最大循环次数大于1，则输出该循环的最小循环元长度和最大循环次数。

题解思路：

- 1) 与自己的前缀进行匹配，与KMP中的next数组的定义相同。next数组的定义是：字符串中以i结尾的子串与该字符串的前缀能匹配的最大长度。
- 2) 将字符串S与自身进行匹配，对于每个前缀，能匹配的条件即是： $S[i - \text{next}[i] + 1 \sim i]$ 与 $S[1 \sim \text{next}[i]]$ 是相等的，并且不存在更大的next满足条件。
- 3) 当 $i - \text{next}[i]$ 能整除i时， $S[1 \sim i - \text{next}[i]]$ 就是 $S[1 \sim i]$ 的最小循环元。它的最大循环次数就是 $i / (i - \text{next}[i])$ 。

这是刘汝佳《算法竞赛入门经典训练指南》上的原题（p213），用KMP构造状态转移表。在3.3.2 KMP算法。

```
1  '''
2  gpt
3  这是一个字符串匹配问题，通常使用KMP算法（Knuth-Morris-Pratt算法）来解决。
4  使用了 Knuth-Morris-Pratt 算法来寻找字符串的所有前缀，并检查它们是否由重复的子串组成，
5  如果是的话，就打印出前缀的长度和最大重复次数。
6  '''
7
8  # 得到字符串s的前缀值列表
9  def kmp_next(s):
10     # kmp算法计算最长相等前后缀
11     next = [0] * len(s)
12     j = 0
13     for i in range(1, len(s)):
14         while s[i] != s[j] and j > 0:
15             j = next[j - 1]
16         if s[i] == s[j]:
17             j += 1
18         next[i] = j
19     return next
20
21
22 def main():
23     case = 0
24     while True:
25         n = int(input().strip())
26         if n == 0:
27             break
28         s = input().strip()
29         case += 1
30         print("Test case #{}".format(case))
31         next = kmp_next(s)
32         for i in range(2, len(s) + 1):
33             k = i - next[i - 1] # 可能的重复子串的长度
34             if (i % k == 0) and i // k > 1:
35                 print(i, i // k)
36         print()
37
38
39 if __name__ == "__main__":
40     main()
41
```