数据结构与算法
（八）
字符串算法

杜育根

Ygdu@sei.ecnu.edu.cn

字符串算法

- 在很多场景里中，我们经常要处理字符串，解决单个字符串、多个字符串的问题。本课程，我们将介绍 KMP，扩展 KMP 算法，和字典树这些数据结构。并且扩展到多模式串的匹配算法——AC 自动机，以及AC 自动机上的动态规划问题。

KMP 算法

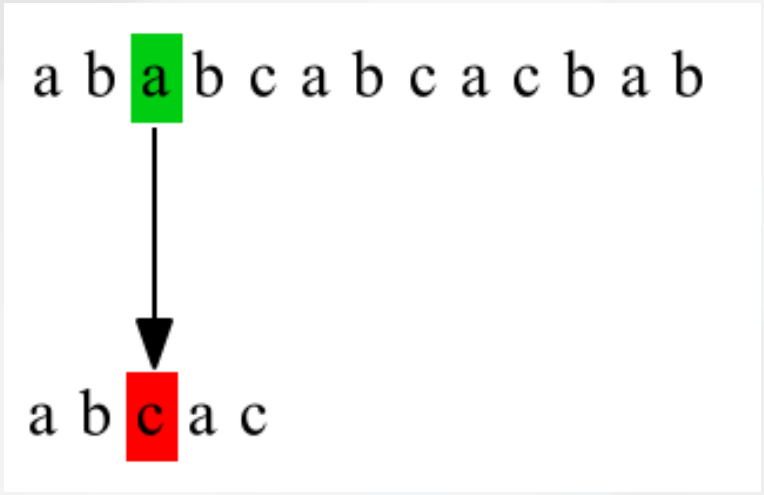
- 概述
- 在计算机科学中，Knuth-Morris-Pratt 字符串查找算法（简称：KMP 算法）要解决的问题是在字符串（也叫主串）中的模式（pattern）定位问题，说简单点就是我们平时常说的关键字搜索。模式串P就是关键字，如果它在一个主串T中出现，就返回它的具体位置，否则返回-1（常用手段）。
- 该算法通过对模式串在不匹配时本身包含的信息来确定下一个匹配将在哪里开始的发现，从而避免重新检查先前匹配的字符,提高程序运行效率。
- 具体来说，KMP 算法在匹配前会预处理模式串 P，得到一个 fail数组。借助fail数组，可以在匹配过程中减少很多冗余的匹配操作，由此提高了算法的效率。KMP 算法的时间复杂度为 O(n+m)(n,m分别是主串、模式串长度）。

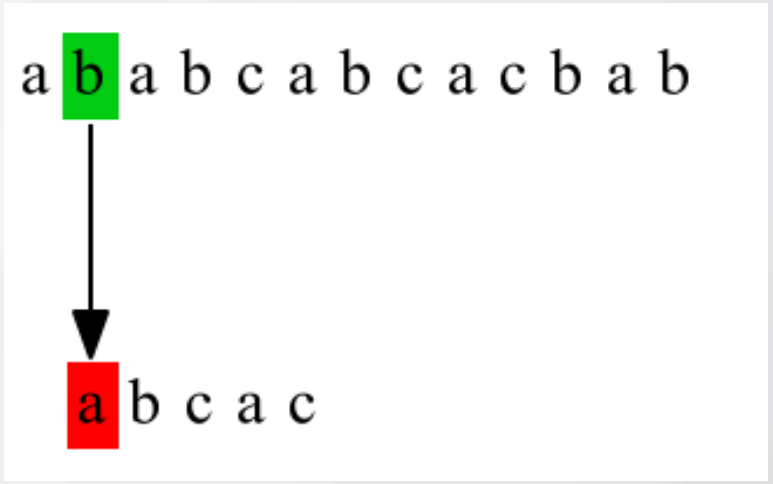
暴力算法BF（Brute Force）算法

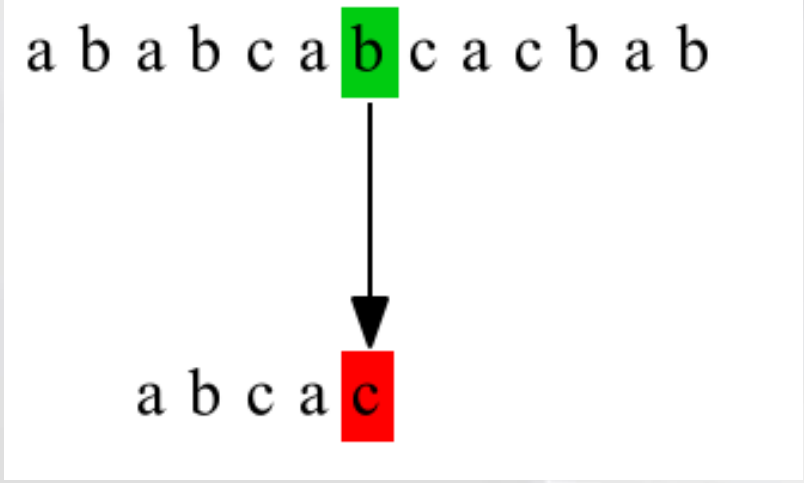
- 暴力匹配方法的思想非常朴素：
- 依次从主串的首字符开始，与模式串逐一进行匹配；
- 遇到失配时，则移到主串的第二个字符，将其与模式串首字符比较，逐一进行匹配；
- 重复上述步骤，直至能匹配上，或剩下主串的长度不足以进行匹配。

暴力匹配的例子

- 主串T="ababcabcacbab"，模式串P="abcac"，
- 第一次匹配失配　　第二次匹配失配：　　第三次匹配失配：





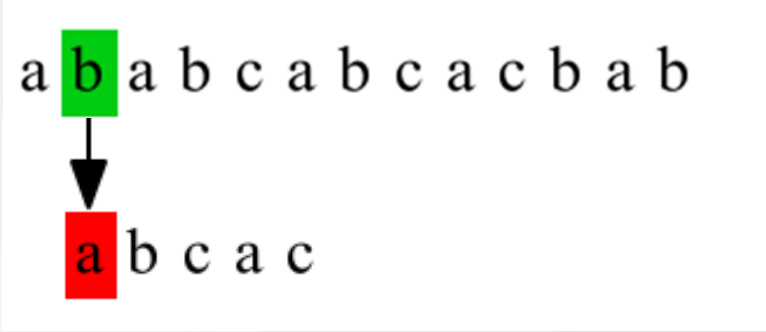


BF算法代码

```
int bf(char *t, char *p) {
    int i, j, tem;
    int tlen = strlen(t), plen = strlen(p);
    for(i = 0, j = 0; i <= tlen - plen; i++, j = 0) {
        tem = i;
        while(t[tem] == p[j] & j < plen) {
            tem++;
            j++;
        }
        // matched
        if(j == plen) {
            return i;
        }
    }
    // [p] is not a substring of [t]
    return -1;
}
```

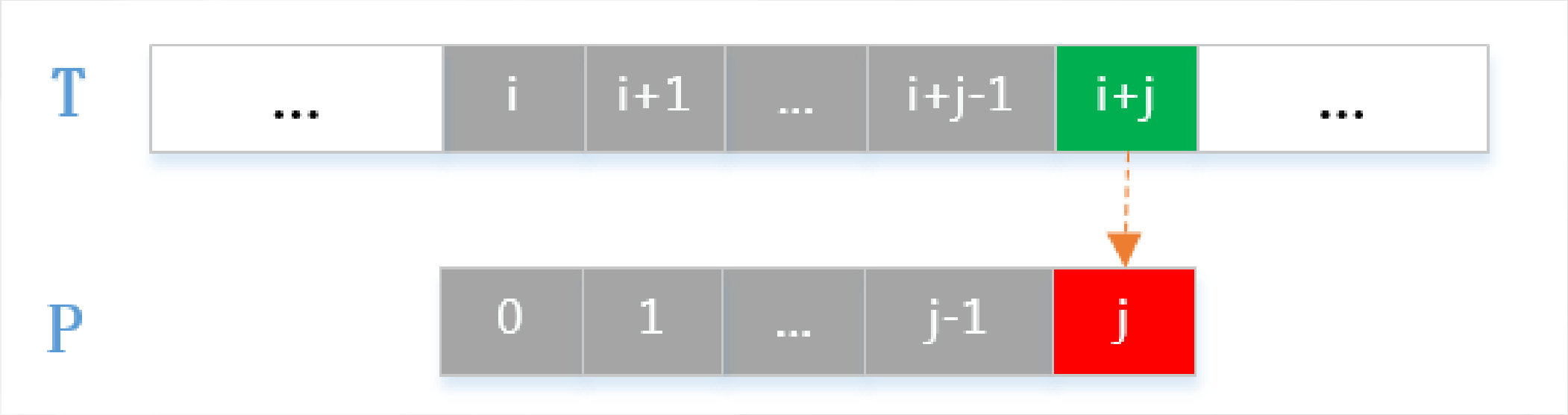
暴力算法的时间复杂度和缺点

- 时间复杂度：i在主串移动次数（外层的for循环）有n－p次，在失配时j移动次数最多有p－1次（最坏情况下）；因此，复杂度为O(n*p)。
- 暴力方法缺点：失配后下一次匹配，主串的起始位置 = 上一轮匹配的起始位置 + 1；模式串的起始位置 = 首字符P[0]。没有利用上次失配已经匹配上的字符的信息，造成了重复匹配。
- 举个例子，比如：第一次匹配失败时，主串、模式串失配位置的字符分别为 a 与 c，下一次匹配时主串、模式串的起始位置分别为T[1]与P[0]；而在模式串中c之前是ab，未有重复字符结构，因此T[1]与P[0]肯定不能匹配上，这样造成了重复匹配。直观上，下一次的匹配应从T[2]与P[0]开始。



KMP算法思想

- 根据暴力方法的缺点，而引出KMP算法的思想。首先，一般化匹配失败，如下图所示：

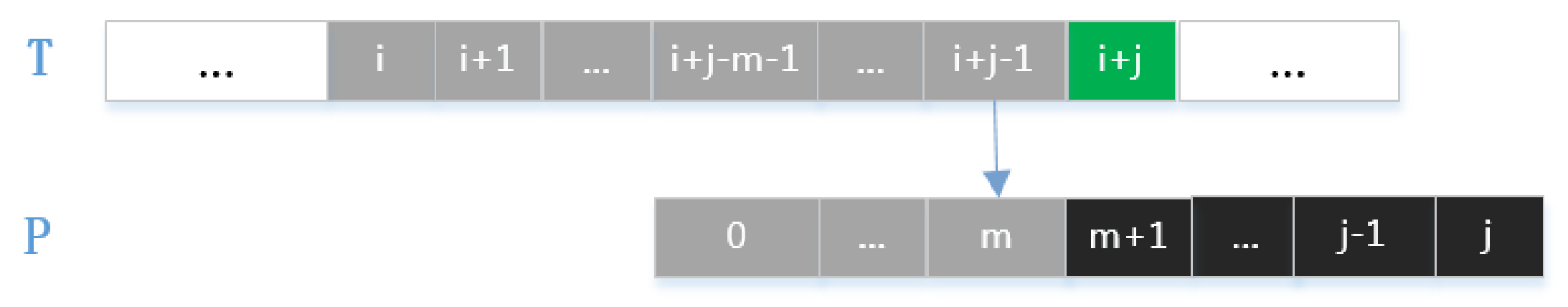


- 在暴力匹配方法中，下一次匹配开始时，主串指针会回溯到i+1，模式串指针会回退到0。那么，如果不让主串指针发生回溯，模式串的指针应回退到哪个位置才能保证正确匹配呢？首先，我们从上图中可以得到已匹配上的字符：

$$T[i \dots i+j-1] = P[0 \dots j-1] \qquad (1)$$

KMP算法思想

- KMP算法思想便是利用已经匹配上的字符信息，使得模式串的指针回退的字符位置能将主串与模式串已经匹配上的字符结构重新对齐。当有重复字符结构时，下一次匹配如下图所示：

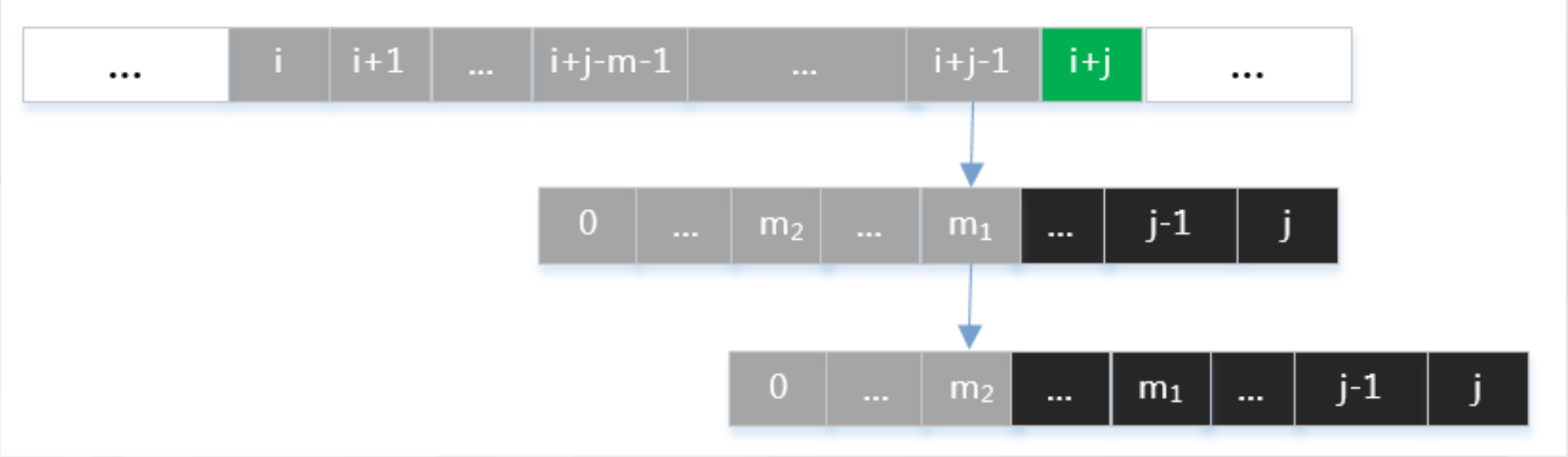


- 从图中可以看出，下一次匹配开始时，主串指针在失配位置i+j，模式串指针回退到m+1；模式串的重复字符结构：

$$P[j-m-1 \dots j-1] = T[i+j-m-1 \dots i+j-1] = P[0 \dots m] \qquad (2)$$

- 且有$T[i+j] \neq P[j] \neq P[m+1]$

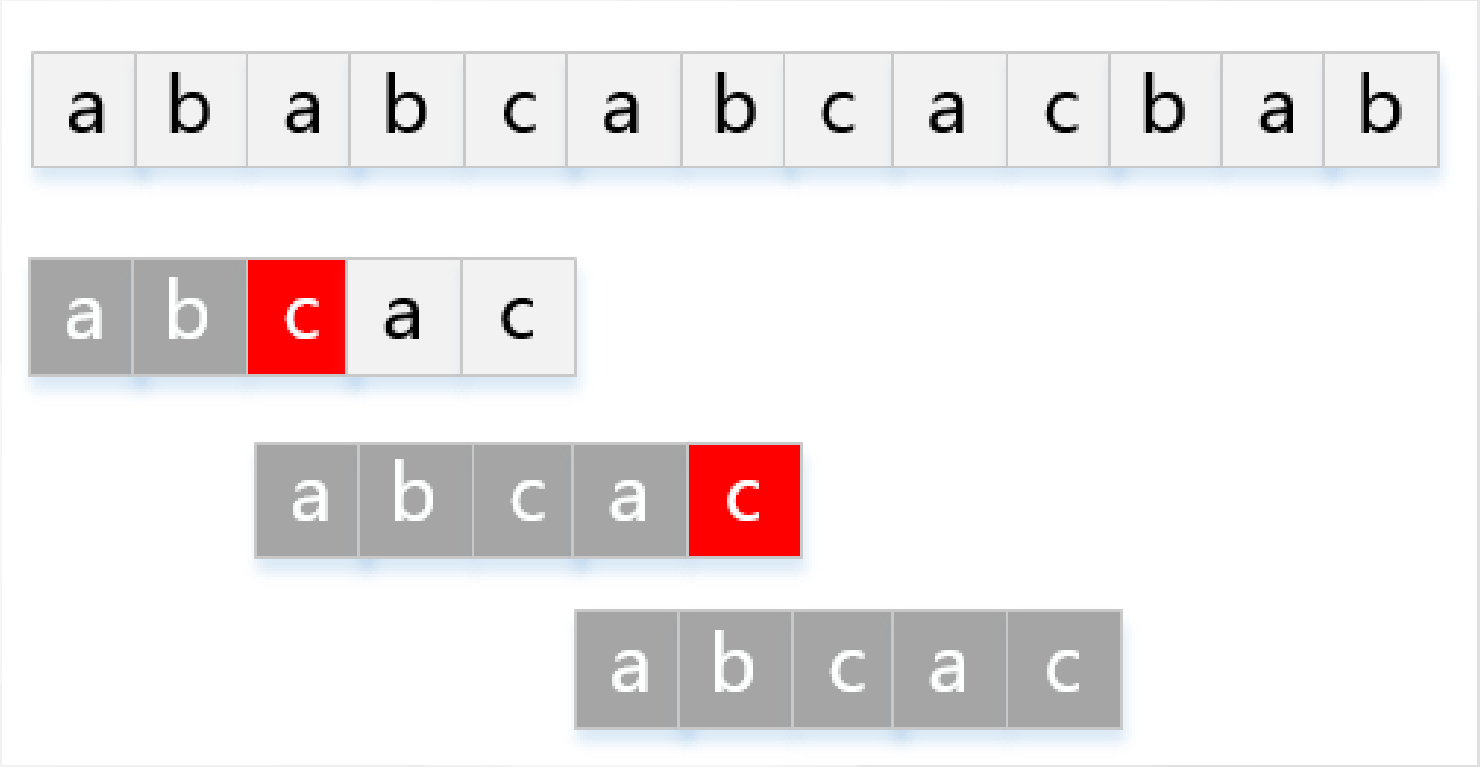
- 那么应如何选取m值呢？假定有满足式子(2)的两个值$m_1 > m_2$，如下图所示：



- 如果选取m= m_2，则会丢失m= m_1的这一种字符匹配情况。由数学归纳法容易知道，应取所有满足式子(2)中最大的m值。

KMP算法中每一次的匹配

- 主串的起始位置 = 上一轮匹配的失配位置；
- 模式串的起始位置 = 重复字符结构的下一位字符（无重复字符结构，则模式串的首字符）
- 模式串P="abcac"匹配主串T="ababcabcacbab"的KMP过程如下图：



部分匹配函数（失配函数）

- 根据上面的讨论，我们定义部分匹配函数（Partial Match，在数据结构书[2]称之为失配函数）：
- $$f(j)=\begin{cases}\max\{m\}, P[0...m]=P[j-m...j],0\le m<j \\ -1, \qquad\qquad else\end{cases}$$
- 其表示字符串P[0...j]的前缀与后缀完全匹配的最大长度，也表示了模式串中重复字符结构信息。
- KMP中的next[j]函数表示对于模式串失配位置j+1，下一轮匹配时模式串的起始位置（即对齐于主串的失配位置）；则

$$next[j]=f(j)+1$$

例子，模式串P=“abababababca”的部分匹配函数即fail函数与next函数如下：

j	0	1	2	3	4	5	6	7	8	9
P[j]	a	b	a	b	a	b	a	b	c	a
f(j)	-1	-1	0	1	2	3	4	5	-1	0
next[j]	0	0	1	2	3	4	5	6	0	1

部分匹配函数的计算公式及C代码

$$f(j)=\begin{cases} f^k(j-1)+1 & \min\limits_k P[f^k(j-1)+1]=P[j] \\ -1 & else \end{cases}$$

```
int *fail(char *p) {
    int len = strlen(p);
    int *f = (int *) malloc(len * sizeof(int));
    f[0] = -1;
    int i, j;
    for(j = 1; j < len; j++) {
        for(i = f[j-1]; ; i = f[i]) {
            if(p[j] == p[i+1]) {
                f[j] = i + 1;
                break;
            }
            else if(i == -1) {
                f[j] = -1;
                break;
            }
        }
    }
    return f;
}
```

KMP的C代码

```
int kmp(char *t, char *p) {
    int *f = fail(p);
    int i, j;
    for(i = 0, j = 0; i < strlen(t) && j < strlen(p); ) {
        if(t[i] == p[j]) {
            i++;
            j++;
        }
        else if(j == 0)
            i++;
        else
            j = f[j-1] + 1;
    }
    return j == strlen(p) ? i - strlen(p) : -1;
}
```

KMP匹配过程

- 理解了fail数组以后，我们先不急着研究怎么计算 fail，而是先看看如何借助fail值快速地进行字符串匹配。我们用两个值match和i来标记模式串P和匹配串T分别匹配到的下标位置。
- 匹配过程：在匹配过程开始之前，模式串的 fail数组已经计算完毕。匹配过程的伪代码如下：

```
1.  KMP(T, P)
2.      fail = getFail(P)
3.      match = -1
4.      for i = 0 to T.length - 1
5.          while match >= 0 and P[match + 1] != T[i]
6.              match = fail[match]
7.          if P[match + 1] == T[i]
8.              match += 1
9.              if match == P.length - 1
10.                 return true
11.     return false
```

KMP 算法的演示

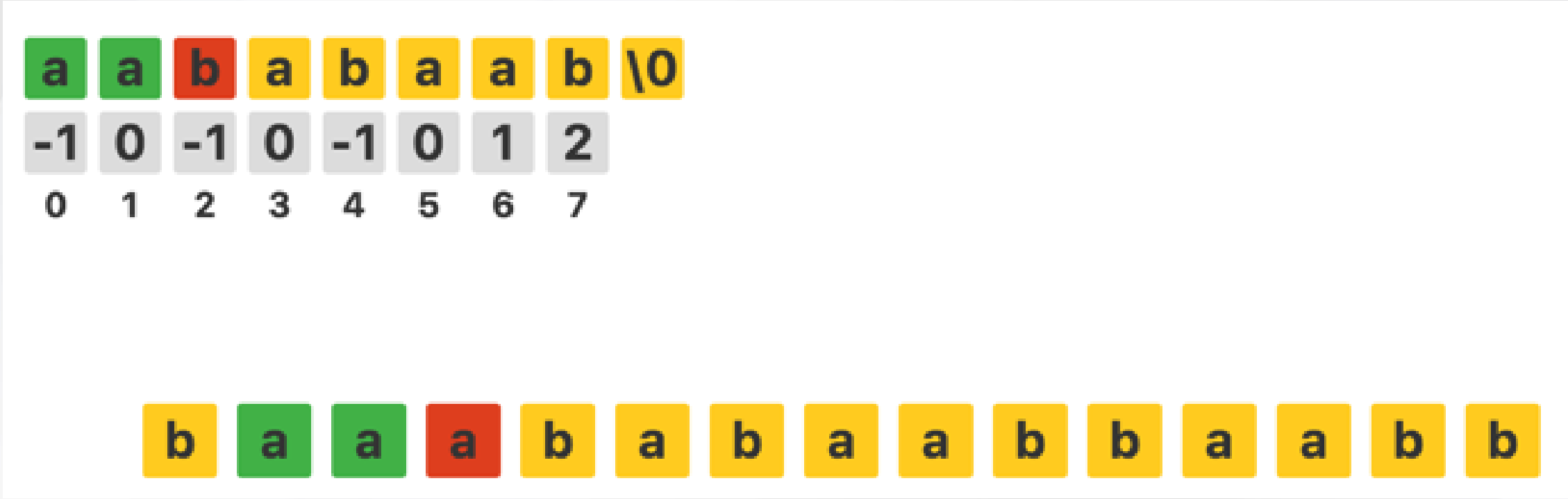
- 我们用两个值match和i来标记模式串P和匹配串T分别匹配到的下标位置，初始状态下 match=−1,i=0。注意，图中上面的串是模式串，标红的位置是P[match + 1]；下面的串是文本串，标红的位置是T[i]。

	a	a	b	a	b	a	a	b
fail	−1	0	−1	0	−1	0	1	2

- 一、判断P[match + 1]是否等于T[i]。如果不相等，则让match = fail[match]，直到相等或 match=–1。
- 此时match == -1，P[match + 1] = ‘a', T[i] = ‘b'，P[match + 1] != T[i]，不对match作调整。

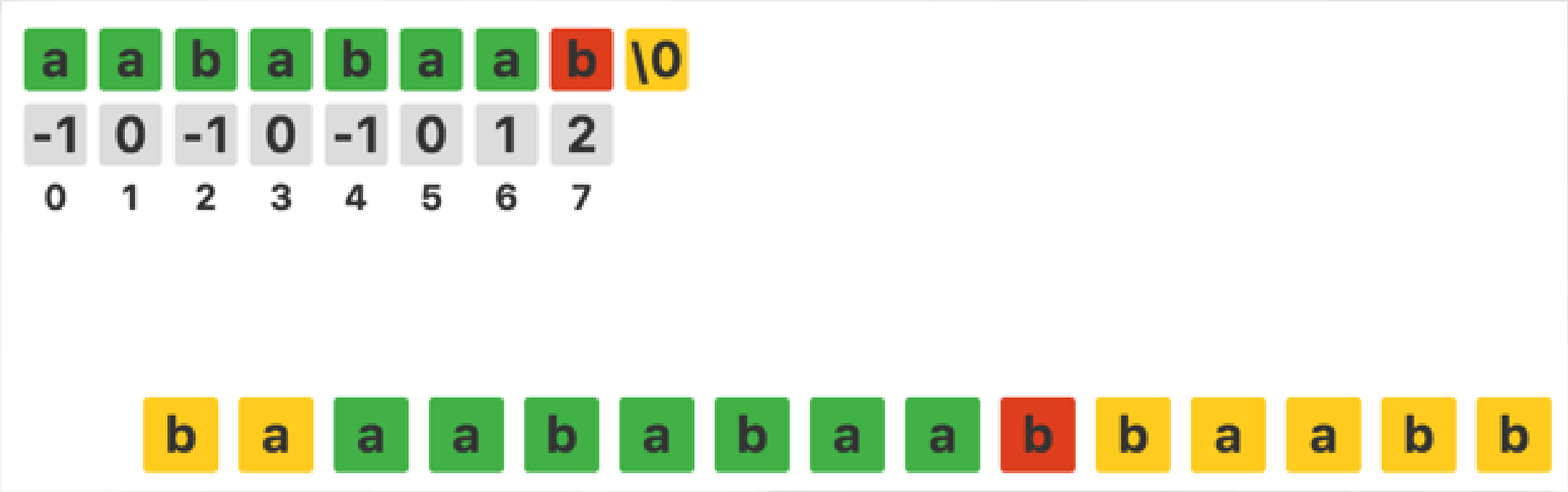
- 二，match=−1,i=1。发现此时P[match + 1] = 'a' = T[i]，将match += 1。

- 三，match=0,i=2。发现此时P[match + 1] = 'a' = T[i]，将match += 1。



- 四，match=1,i=3，发现此时P[match + 1] = ‘b', T[i] = ‘a'，P[match + 1] != T[i]，让match = fail[match] = 0。此时，P[match + 1] = 'a' = T[i]。将match += 1。

- 五，match=1,i=4，发现此时P[match + 1] = 'b' = T[i]，将match += 1。之后省略若干步，直到匹配到match == P.length - 1，说明此时已经在 T 中找到 P，算法结束。



匹配过程深入讨论

- 为什么在匹配的过程中，发现P[match + 1]和T[i]不相等的时候可以让match =fail[match]呢？由fail的定义可知，$P_{0\ldots fail[match]} = P_{match-fail[match]\ldots match}$。并且由$match, i$的定义可知，$P_{0\ldots match} = T_{i-match\ldots i}$，因此必然有$P_{match-fail[match]\ldots match} = T_{i-fail[match]\ldots i}$。通过这两个等式，我们就可以推出：

$$P_{0\ldots fail[match]} = T_{i-fail[match]\ldots i}$$

- 因此，将match = fail[match]，可以确保 match 和 i 符合它们定义的要求，可以继续后面的匹配过程了。
- 整个算法过程中，match最多向后移动|P|次，i最多向后移动|T|次，总体时间复杂度为 O(|P| + |T|)。

计算 fail

- 现在我们已经知道，如何借助 fail快速地求解字符串匹配。那么，要如何求出 fail数组呢？算法的伪代码如下：

```
1.  getFail(P){
2.      fail[0] = -1
3.      match = -1
4.      for i = 1 to P.length - 1
5.          while match >= 0 and P[match + 1] != P[i]
6.              match = fail[match]
7.          if P[match + 1] == P[i]
8.              match += 1
9.          fail[i] = match
10.     return fail}
```

- 是不是看起来很熟悉？没错，求解 fail 的过程，就相当于P串和自己匹配的过程。整个算法过程中，match最多向后移动 |P|次，总体时间复杂度为 O(|P|)。
- 因此，KMP 算法的整体时间复杂度为 O(|P|+|T|)，是一种线性复杂度的串匹配算法。

C++ KMP 示例代码如下：

```
void getFail(char *P, int *fail) {
    int match = -1;
    fail[0] = -1;
    for (int i = 1; P[i]; ++i) {
        while(match>=0  &&  P[match+1]!=P[i])
        {
            match = fail[match];
        }
        if (P[match + 1] == P[i]) {
            match++;
        }
        fail[i] = match;
    }
}
```

```
bool KMP(char *T, char *P) {// T文本串P匹配串
    int fail[strlen(P)],match=-1;//长度|P|的数组fail
    getFail(P, fail);
    for (int i = 0; T[i]; ++i) {
        while(match>=0 && P[match+1]!=T[i]){
            match = fail[match];
        }
        if (P[match + 1] == T[i]) {
            match++;
            if (!P[match + 1]) {
            // P[match + 1] == '\0'，就意味着match
            //是 P 串的最后一位下标，说明匹配成功
                return true;
            }
        }
    }
    return false;
}
```

next 函数

- 令 next[i] 表示后缀 i即 $T_{i\cdots|T|}$与 T 的最长公共前缀。
- 例如

数组索引	1	2	3	4	5	6	7	8	9
字符串数组	a	a	a	b	a	a	a	a	b
next数组	9	2	1	0	3	4	2	1	0

怎样得到 next 函数

假设 next[i]((0<i<x) 的值都已经求出，现在要求 next[x] 。

我们设 p=i+next[i]−1， 当 i=k,0<i<k 时，p取到最大。

根据定义我们可以得到：$T_{k\cdots p} = T_{1\cdots next[k]}$，如右图一所示蓝色部分：

现在我们要求 $T_{x..n}$ 与 $T_{1\cdots n}$ 的最长公共前缀。

由 $T_{k\cdots p} = T_{1\cdots next[k]}$得：

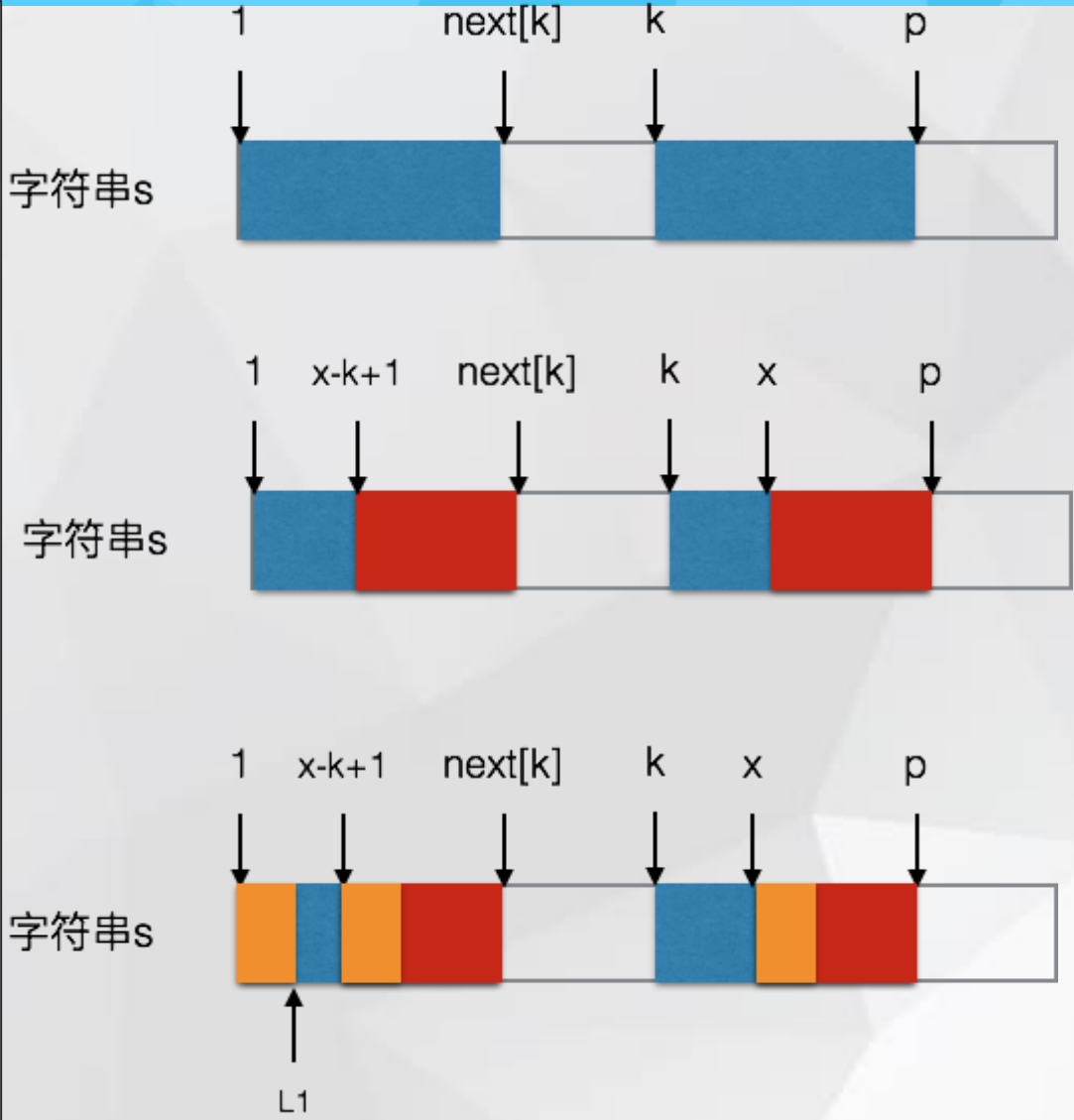
$T_{x\cdots p} = T_{x-k+1\cdots next[k]}$ （如右图二所示红色部分）

设 $L_1 = next[x-k+1]$，根据右图三，可以得到：

$T_{1..L1} = T_{x-k+1\cdots x-k+L1} = T_{x\cdots x+L1-1}$ （图中的黄色部分）

也就是说，如果图中的黄色部分小于红色部分的话，也就是 $L_1 < p-x+1$，那么我们可以确定 $next[x] = L_1$。否则，我们从L_1+1 和 $p+1$ 位置开始逐一比较，从而求出 $next[x]$ 的实际的值。

由于这个过程中，p是一个不降的序列，所以总体的时间复杂度是 O(n)。



得到 next 函数的代码

```
1.  void getnext() {
2.      next[1] = n;
3.      int p = 1;
4.      while(p < n && t[p] == t[p + 1]) p++;
5.      next[2] = p-1;
6.      int k = 2,l;
7.      for(int i = 3; i <= n; i++) {
8.          p = k + next[k] - 1;
9.          l = next[i - k + 1];
10.         if (i + l <= p) next[i] = l;
11.         else {
12.             int j = p - i + 1;
13.             if(j < 0)     j = 0;
14.             while(i + j <= n && t[i + j] == t[j + 1])     j++;
15.             next[i] = j;
16.             k = i;
17.         }
18.     }
19. }
```

习题：重复的密文

- 小明收到了一串密文，但是由于接收器坏了，他不停的重复接收，终于，小明把插头拔了，机器停止了，但是小明已经收到了一个很长字符串，它是由某个原始串不停的重复形成了，因为断电，最后一遍也不一定是完整的。小明现在想知道这个原始串的最短可能长度是多少。
- 输入格式
- 第一行输入一个正整数 $L(1<L\leq 10^6)$，表示这个字符串的长度。
- 第二行输入一个字符串，全部由小写字母组成。
- 输出格式
- 答案输出，输出最短的可能长度。
- 样例输入
- 8
- cabcabca
- 样例输出
- 3

习题：阿里天池的新任务

- 阿里“天池”竞赛平台近日推出了一个新的挑战任务：对于给定的一串碱基序列 t，判断它在另一个根据规则生成的 DNA 碱基序列 s中出现了多少次。
- 定义序列 w:

$$w_i = \begin{cases} b, & i = 0 \\ (w_{i-1} + a) \mod n, & i > 0 \end{cases}$$

- 定义长度为 n 的 DNA 碱基序列 s（下标从 0开始）：

$$s_i = \begin{cases} A, & (L \le w_i \le R) \wedge (w_i \equiv 0 \mod 2) \\ T, & (L \le w_i \le R) \wedge (w_i \equiv 1 \mod 2) \\ G, & ((w_i < L) \vee (w_i > R)) \wedge (w_i \equiv 0 \mod 2) \\ C, & ((w_i < L) \vee (w_i > R)) \wedge (w_i \equiv 1 \mod 2) \end{cases}$$

- 其中 ∧表示“且”关系，∨表示“或”关系。现给定另一个 DNA 碱基序列 t，以及生成 s的参数 n,a,b,L,R，求 t 在 s 中出现了多少次。
- 输入格式: 数据第一行为 5 个整数，分别代表 n,a,b,L,R。第二行为一个仅包含 A,T,G,C 的一个序列代表 t。数据保证 1≤n≤10^6, 0<a<n,0≤b<n, 0≤L≤R < n,0≤L≤R<n, |t|≤10^6; a,n 互质。
- 输入格式: 输出一个整数，为 t在 s中出现的次数。
- 样例说明:
- 对于第一组样例，生成的 s为TTTCGGAAAGGCC。

- 样例输入1
- 13 2 5 4 9
- AGG
- 样例输出1
- 1

- 样例输入2
- 103 51 0 40 60
- ACTG
- 样例输出2
- 5

扩展 KMP

- 概述
- 扩展 KMP是对KMP算法的扩展，处理字符串 S 的所有后缀与字符串 T 的最长公共前缀，它解决如下问题：
- 定义母串S，和字串T，设S的长度为n，T的长度为m，求T与S的每一个后缀的最长公共前缀，也就是说，设extend数组,extend[i]表示T与S[i,n-1]的最长公共前缀，要求出所有extend[i](0<=i<n)。

解决原问题

- 假设 extend[i](0<i<x) 的值都已经求出，现在需要计算 extend[x]。
- 已知：$S_{k\cdots p} = T_{1\cdots extend[k]}$，求 $S_{x\cdots n}$与 $T_{1\cdots m}$的最长公共前缀。解法与上面的问题类似。
- 至此，我们得到 extend 数组，字符串 S的所有后缀与字符串 T 的最长公共前缀。

```
void getextend() {
    int p = 0;
    while (p < m && p < n && s[p + 1] == t[p + 1]) {
        p++;
    }
    extend[1] = p;
    int k = 1, l;
    for (int i = 2; i <= m; i++) {
        p = k + extend[k] - 1;
        l = next[i - k + 1];
        if (i + l <= p) {
            extend[i] = l;
        } else {
            int j = p - i + 1;
            if(j < 0) j = 0;
            while(i + j <= m && j + 1 <=n && s[i + j] == t[j + 1])    j ++;
            extend[i] = j;
            k = i;
        }
    }
}
```

习题：首尾相接

- 小明有两个字符串 S1 和 S2，小明想把 S1 接到 S2 后面。因为 S1 前面有一些字符和 S2 后面的一些字母一样，所以小明在连接的时候就没必要重复了，比如 S1 为cdefgh，S2 为abcde，那么cde这部分就是最长的重复部分，小明可以将这两个串连接为abcdefgh。现在，给你串 S1 和串 S2，请你帮小明找出最长重复部分的长度。
- 输入格式
- 共两行，每行一个字符串，由小写字母构成，第一行表示串 S1，第二行表示串 S2。$(1 \le |S1|, |S2| \le 50000)$
- 输出格式
- 答案输出在一行，先输出重复的字符串，再输出其长度，中间以空格隔开。若该串为空，只需输出 0。
- 样例输入
- riemann
- marjorie
- 样例输出
- rie 3

习题：旋转数字

- 小明发现了一个很好玩的事情，他对一个数作旋转操作，把该数的最后的数字移动到最前面。比如，数 123 可以得到 312,231,123，这样就可以得到很多个数。
- 现在，小明的问题是这些数中，有多少个不同的数小于原数，多少个等于原数，多少个大于原数。
- 旋转中可能会出现前导零，两数比较的时候可以忽略前导零的影响。
- 输入格式
- 输入一个整数 $N(0< N\leq 10^{100000})$。
- 输出格式
- 答案在一行中输出三个整数，分别是小于 N，等于 N，大于 N 的个数，中间以空格隔开。
- 样例输入
- 341
- 样例输出
- 1 1 1

习题：匹配格式

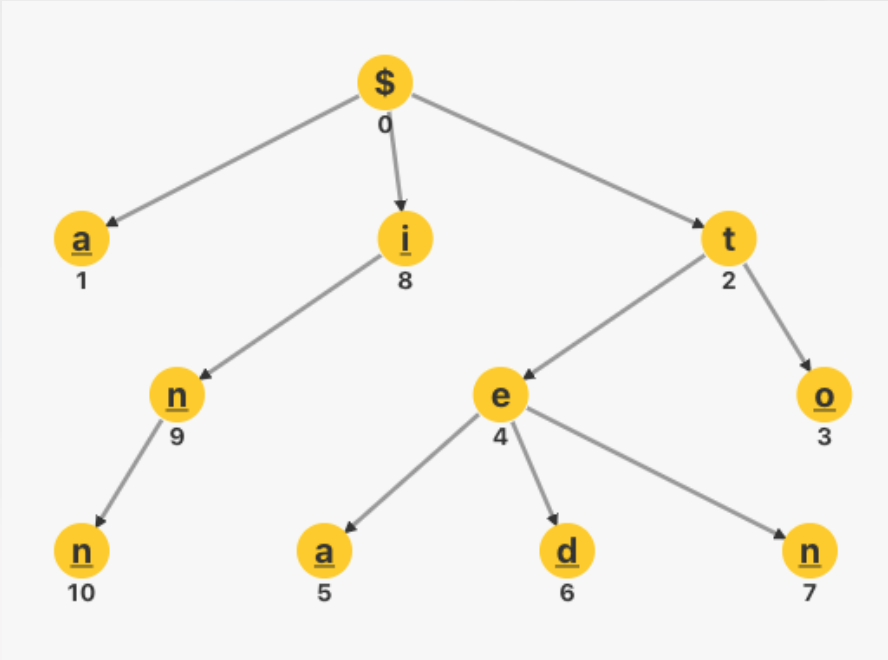
- 有一字符串 S，小明想在 S 中找到最长的子串 E，使得 S 满足格式 "EAEBE"，其中 A,B 可以为任意的 S 子串。也就是说子串 E 既是 S的前缀也是 S的后缀，同时还在 S 中间出现，但不与前缀 E与后缀 E 重叠。
- 输入格式
- 输入一个字符串 S，由小写字母构成，长度不超过 10^6。
- 输出格式
- 答案输出占一行，输出一个整数，表示子串 EEE 的长度。
- 样例输入
- aaxoaaaaa
- 样例输出
- 2

字典树

- 接下来我们来看两个字符串匹配问题：
- 已知有 n个长度不一定相同的母串，以及一个长度为 m的模式串 T，求该模式串是否是其中一个母串的前缀。如果将模式串 T 挨个去比较，则算法复杂度会很高，达到 O(n×m)，是否有高效的方法呢?
- 已知一个长度为 n 的字符串 S，求该字符串有多少个不相同的子串。朴素的做法，可以先枚举出所有的子串，这样时间复杂度为 $O(n^2)$，之后再去重，直接做算法复杂度很高，改成哈希或者用 set、map 处理，整体复杂度仍然会很高。
- 实际上，我们可以用 Trie 树来高效求解这两个问题。我们先来看看什么是 Trie 树吧。

Trie 树

- Trie 树又称字典树或者单词查找树，是一种树形结构的数据结构，支持字符串的插入、和查询操作，常用于大量字符串的检索、去重、排序等。Trie 树利用字符串的公共前缀，逐层建立起一棵多叉树。在检索时类似于在字典中查单词，从第一个字符开始遍历，在 Trie 树中一层一层往下查找，查找效率可以达到 O(n)，n 为查找字符串的长度。
- 下图是一棵以词：a、to、tea、ted、ten、i、in、inn构成的字典树，其中带下划线的结点为 终端结点（从根结点到终端结点的遍历过程就是 Trie 中存储的一个字符串）。



Trie 树特点

- Trie 树有以下特点：
- 1. Trie 树的根结点上不存储字符，其余结点上存且只存一个字符。
- 2. 从根结点出发，到某一结点上经过的字符，即是该结点对应的前缀。
- 3. 每个结点的孩子结点存储的字符各不相同。
- 4. Trie 树牺牲空间来换取时间，当数据量很大时，会占用很大空间。如果字符串均由小写字母组成，则每个结点最多会有 26个孩子结点，则最多会有 26n 个用于存储的结点，n 为字符串的最大长度。
- Trie 树常用于字符串的快速检索，字符串的快速排序与去重，文本的词频统计等。

数据结构

```
1.  const int MAX_N = 10000;  // Trie 树上的最大结点数
2.  struct Trie {
3.      int ch[MAX_N][26];  // ch 保存了每个结点的 26 个可能的子结点编号，26 对应
    //着 26 种小写字母，也就是说，插入的字符串全部由小写字母组成。初始全部为 -1
4.      int tot;  // 总结点个数，初始为 0
5.      int cnt[MAX_N];  // 每个结点

6.      void init() {  // 初始化 Trie 树
7.          tot = 0;
8.          memset(cnt, 0, sizeof(cnt));
9.          memset(ch, -1, sizeof(ch));
10.     }
11. };
```

插入

- 从根结点开始，按字符串中当前字符出边，走到对应的结点。若没有这样的结点，则新开一个结点。

```
1.  void insert(char *str) {
2.      int p = 0; // 从根结点（0）出发
3.      for (int i = 0; str[i]; ++i) {
4.          if (ch[p][str[i] - 'a'] == -1) { // 该子结点不存在
5.              ch[p][str[i] - 'a'] = ++tot; // 新增结点
6.          }
7.          p = ch[p][str[i] - 'a']; // 在 Trie 树上继续插入字符串 str
8.      }
9.      cnt[p]++;
10. }
```

查询

- 从根结点开始，按当前字符出边，走到对应的结点。若没有这样的结点，则不存在；若读完字符串的结点不是终止结点，也不存在。

```
1.  int find(char *str) {  // 返回字符串 str 的出现次数
2.      int p = 0;
3.      for (int i = 0; str[i]; ++i) {
4.          if (ch[p][str[i] - 'a'] == -1) {
5.              return 0;
6.          }
7.          p = ch[p][str[i] - 'a'];
8.      }
9.      return cnt[p];
10. }
```

完整代码

```
const int MAX_N = 10000;  // Trie 树上的最大结点数
const int MAX_C = 26;  // 每个结点的子结点个数上限
struct Trie {
    int ch[MAX_N][MAX_C];  // ch 保存了每个结点的 26
//个可能的子结点编号，26 对应着 26 种小写字母，也就
//是说，插入的字符串全部由小写字母组成。初始全部为 -1
    int tot;  // 总结点个数（不含根结点），初始为 0
    int cnt[MAX_N];
// 以当前结点为终端结点的 Trie 树中的字符串个数
    void init() {  // 初始化 Trie 树，根结点编号始终为 0
        tot = 0;
        memset(cnt, 0, sizeof(cnt));
        memset(ch, -1, sizeof(ch));
    }
    void insert(char *str) {
        int p = 0;  // 从根结点（0）出发
        for (int i = 0; str[i]; ++i) {
            if (ch[p][str[i] - 'a'] == -1) {  // 该子结点不存在
                ch[p][str[i] - 'a'] = ++tot;  // 新增结点
    }
            p = ch[p][str[i] - 'a'];
// 在 Trie 树上继续插入字符串 str
        }
        cnt[p]++;
    }
    int find(char *str) {  // 返回字符
//串 str 的出现次数
        int p = 0;
        for (int i = 0; str[i]; ++i) {
            if (ch[p][str[i] - 'a'] == -1) {
                return 0;
            }
            p = ch[p][str[i] - 'a'];
        }
        return cnt[p];
    }
};
```

内存动态分配

- 在实现字典树时，有时候会发现如果按照题目要求来开辟内存空间，会导致内存超出题目限制。这时，可以使用内存动态分配的小技巧。在之前给出的 Trie 树代码中，无论树上的结点是否是叶子结点，都给它开辟了 MAX_C 这么大的子结点指针数组，这是没有必要的。于是，我们在一个结点一定有子结点的时候再给它开辟大小为 MAX_C的子结点指针数组就可以了，这样能减少很多不必要的内存开销。

代码

```
const int MAX_N = 10000;  // Trie 树上的最大结点数
const int MAX_C = 26;  // 每个结点的子结点个数上限
struct Trie {
    int *ch[MAX_N];  // ch 保存了每个结点的 26 个可能
//的子结点编号，26 对应着 26 种小写字母，也就是说，
//插入的字符串全部由小写字母组成。初始全部为 -1
    int tot;  // 总结点个数（不含根结点），初始为 0
    int cnt[MAX_N];
 // 以当前结点为终端结点的 Trie 树中的字符串个数
     void init() {  // 初始化 Trie 树，根结点编号始终为 0
        tot = 0;
        memset(cnt, 0, sizeof(cnt));
        memset(ch, 0, sizeof(ch));
// 将 ch 中的元素初始化为 NULL
    }
    void insert(char *str) {
       int p = 0;  // 从根结点（0）出发
       for (int i = 0; str[i]; ++i) {
          if (ch[p] == NULL) {
             ch[p] = new int[MAX_C];
// 只有 p 当包含子结点的时候才去开辟 ch[p] 的空间
             memset(ch[p], -1, sizeof(int) * MAX_C);
// 初始化为 -1
          }
if (ch[p][str[i] - 'a'] == -1) { // 该子结点不存在
             ch[p][str[i] - 'a'] = ++tot;
                // 新增结点
       }
       p = ch[p][str[i] - 'a'];
            // 在 Trie 树上继续插入字符串 str
    }
    cnt[p]++;
  }
  int find(char *str) {
             // 返回字符串 str 的出现次数
    int p = 0;
    for (int i = 0; str[i]; ++i) {
       if (ch[p][str[i] - 'a'] == -1) {
          return 0;
       }
       p = ch[p][str[i] - 'a'];
    }
    return cnt[p];
  }
};
```

例题 1：串的快速检索

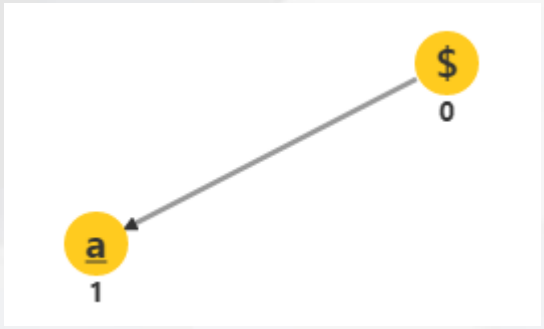
- 给出若干个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。
- 解析
- 我们可以对熟词建一棵 Trie 树。文章中的每个单词都在 Trie 树上查找。

例题 2：串的排序

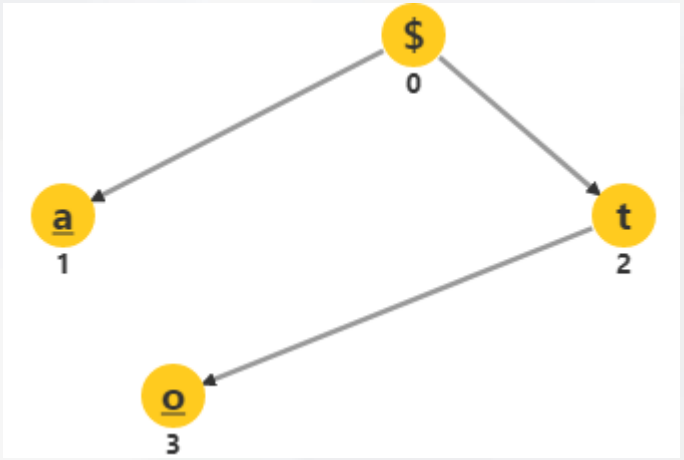
- 给定若干个互不相同的仅由一个单词构成的英文名，按字典序将它们从小到大输出。
- 解析：我们对所有名字（单词）建一棵 Trie 树，再从根开始深度优先搜索 DFS，按照字母从小到大的顺序进行搜索，遇到终端结点时就可以把单词输出（如果是非叶结点的终端结点，则先输出再继续遍历）。这样，所有单词就按字典序都输出一遍。

```
char now[MAX_LEN];
void dfs(int p, int len) {
    if (cnt[p] > 0) {
        now[len] = '\0';
        while (cnt[p] --> 0) {
            cout << now << endl;
        }
    }
    for (int i = 0; i < 26; ++i) {
        if (ch[p][i]) {
            now[len] = 'a' + i;
            dfs(ch[p][i], len + 1);
        }
    }
}
```

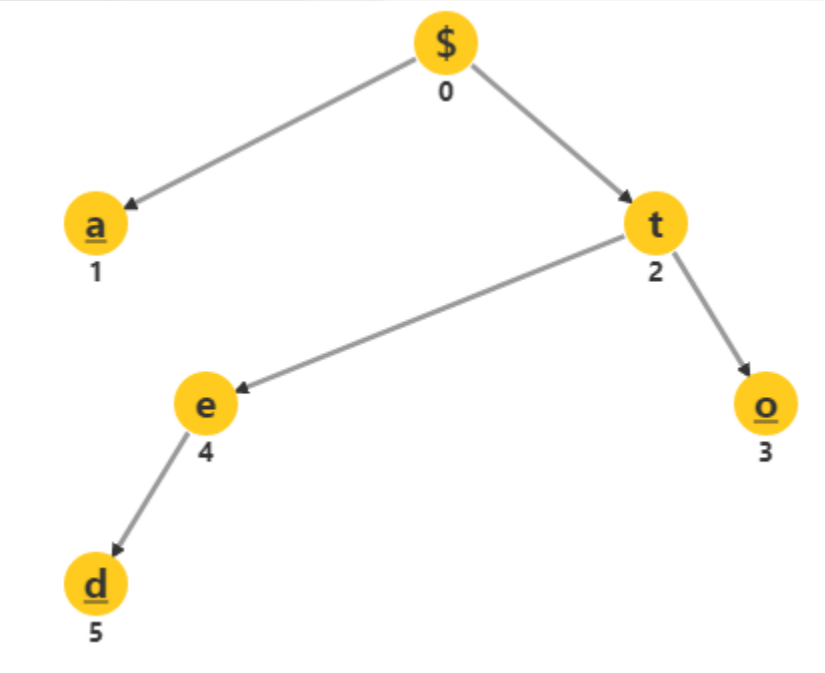
Trie 树的演示



[任务] 插入字符串 a
从根节点开始插入
需要的后继节点不存在，创建新节点
前进到节点 1
标记当前节点，任务完成



[任务] 插入字符串 to
从根节点开始插入
需要的后继节点不存在，创建新节点
前进到节点 2
需要的后继节点不存在，创建新节点
前进到节点 3
标记当前节点，任务完成



[任务] 插入字符串 ted
从根节点开始插入
前进到节点 2
需要的后继节点不存在，创建新节点
前进到节点 4
需要的后继节点不存在，创建新节点
前进到节点 5
标记当前节点，任务完成

习题：糟糕的 Bug

- 小明作为工厂的工程师，在开发网站时不小心写出了一个 Bug：当用户输入密码时，如果既和自己的密码一致，也同时是另一个用户密码的 前缀 时，用户会跳转到 404 页。然而小明坚称：我们的用户那么少，怎么可能触发这个 Bug……
- 机智的你，能不能帮小明确认一下这个 Bug 到底会不会触发呢?
- 样例输入：第一行输入一个整数 n(1≤n≤233333)，表示工厂网站的用户数。接下来一共 n行，每行一个由小写字母a-z组成的字符串，长度不超过 10，表示每个用户的密码。工厂的数据库容量太小，所有密码长度加起来小于 466666。
- 样例输出：如果触发了 Bug 则输出一行Bug!，否则输出一行Good Luck!。
- 样例输入1
- 3
- abc
- abcdef
- cdef
- 样例输出1
- Bug!
- 样例输入2
- 3
- abc
- bcd
- cde
- 样例输出2
- Good Luck!

习题：新年礼物

- 新年了，工厂 BOSS 要给底下人发新年礼物，其中有一份神秘大奖，但却不知道应该发给谁。于是，工厂 BOSS 打算让大家玩一个游戏。
- 一共有 n个字符串排成一排，小明需要从中按顺序选取一部分字符串，使得选出来的字符串顺序和原顺序一致（也就是从中选出一个子序列），且靠前的字符串 xi 和靠后的字符串 xj 之间均同时满足如下要求：
  - xi 是 xj 的前缀
  - xi 是 xj 的后缀
- 小明需要从中按顺序选取最多的字符串，并且满足如上的要求。工厂 BOSS 最后会给选出最多字符串的人平分神秘大奖。你能算出选取的最大个数么？
- 样例输入：第一行输入一个整数 N，紧接着输入 N 行字符串，每个字符串仅包含小写或大写字母。输入数据总共少于 2×10^6个字符。
- 样例输出：输出一个整数，表示最大个数。

提示：需要用到 KMP 预处理，每个字符串中哪些长度的前缀和后缀相同，在向 Trie 插入字符串的时候用动态规划更新到当前字符串为止的最优解。
由于数据量比较大，建议用内存动态分配的写法。

- 样例输入1
- 5
- A
- B
- AA
- BBB
- AAA
- 样例输出1
- 3

- 样例输入2
- 5
- A
- ABA
- BBB
- ABABA
- AAAAAB
- 样例输出2
- 3

AC 自动机

- 从这一节课开始，我们来学习下 AC 自动机算法。
- 首先我们来看一个经典的问题：已知有 n 个长度不一定相同的模式串，以及一个长度为 m 的母串 S，求在母串 S 中出现过多少个模式串。朴素的做法时间复杂度会很高，有没有很高效的方法呢?

AC 自动机三个过程

实际上，这是一道 AC 自动机的经典题。AC 自动机（Aho-Corasick automaton）于 1975 年在贝尔实验室产生，这是一种将 Trie 树和 KMP 相结合的算法。AC 自动机有以下三个过程：

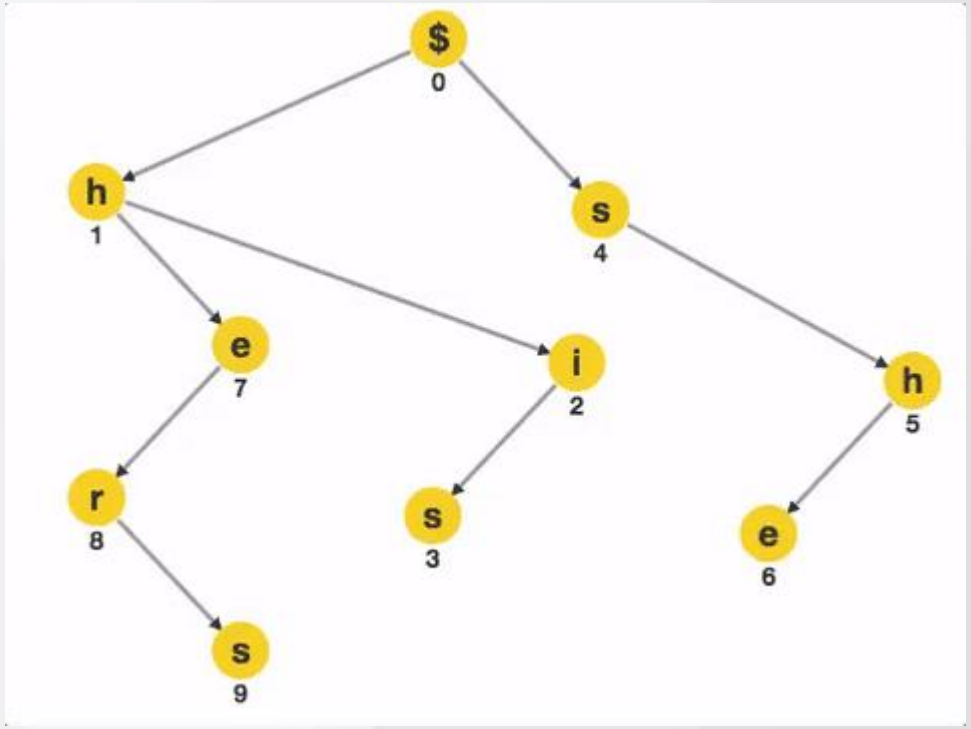
1. 建立 Trie 树。这一步操作和 Trie 树的一样，将若干个模式串建立起一棵 Trie 树。

2. 建立失败指针。这一步类似于 KMP 算法中建立 next 数组，记得 KMP 中提到的失败指针 next 作用么？这是为了方便后续的匹配操作，当第 i 位失配时，则跳转到 next[i] 位进行匹配。在 AC 自动机中也是同样的功能，在某个结点 A上失配时，则通过失败指针指向某结点 B进行下一次的匹配（定义以结点 A 为终点的某个后缀为 s1；以根结点为起点，结点 B为终点的串为 s2，则有 s1=s2，且保证在所有这样的串中， s1 和 s2 的长度是最大的），而不需要回溯到上一层，避免多余的匹配操作。

3. 字符串匹配。从根结点开始，沿树的路径进行匹配，如果当前结点匹配成功则继续往下一个结点匹配，否则跳转到失败指针所指的结点进行匹配。重复上述过程，直到匹配完模式串为止。

广度优先搜索的方法构造失败指针

- 我们可以用广度优先搜索的方法构造失败指针。每个结点都有一个失败指针，首先将根结点的失败指针指向空，根结点的直接子结点的失败指针指向根结点。对 Trie 树进行广度优先搜索，每个结点的失败指针都是由它父结点的失败指针决定的。例如，字符串she是从sh连向she的，sh的失败指针指向h，所以she的失败指针就指向"h" + "e" = "he"。若不存在he，则失败指针也就会指向更上方。



参考代码

```
const int MAX_N = 10000;
const int MAX_C = 26;
struct AC_Automaton {
    int ch[MAX_N][MAX_C], fail[MAX_N], cnt[MAX_N];
        // ch 和 cnt 数组与 Trie 树中的一样；fail 保存的是失败指针。ch 和 fail 默认都为 -1
    int tot;  // Trie 树的总结点（不含根结点）个数

    void init() {
        memset(ch, -1, sizeof(ch));
        memset(fail, 0, sizeof(fail));
        tot = 0;
        memset(cnt, 0, sizeof(cnt));
    }

    void insert(char* str) {
        int p = 0;
        for (int i = 0; str[i]; ++i) {
            if (ch[p][str[i] - 'a'] == -1) {
                ch[p][str[i] - 'a'] = ++tot;
            }
            p = ch[p][str[i] - 'a'];
        }
        cnt[p]++;
    }
```

参考代码（续）

```
23.  void build() {
24.      int l = 0, r = 0, Q[MAX_N];
25.      for (int i = 0; i < MAX_C; i++) {
26.          if (ch[0][i] == -1) {
27.              ch[0][i] = 0;
28.          } else {
29.              Q[r++] = ch[0][i];
30.          }
31.      }
32.      while (l < r) {
33.          int p = Q[l++];
34.          for (int i = 0; i < MAX_C; i++) {
35.              if (ch[p][i] == -1) {
36.                  ch[p][i] = ch[fail[p]][i];
37.              } else {
38.                  fail[ch[p][i]] = ch[fail[p]][i];
39.                  Q[r++] = ch[p][i];
40.              }
41.          }
42.      }
43.  }
```

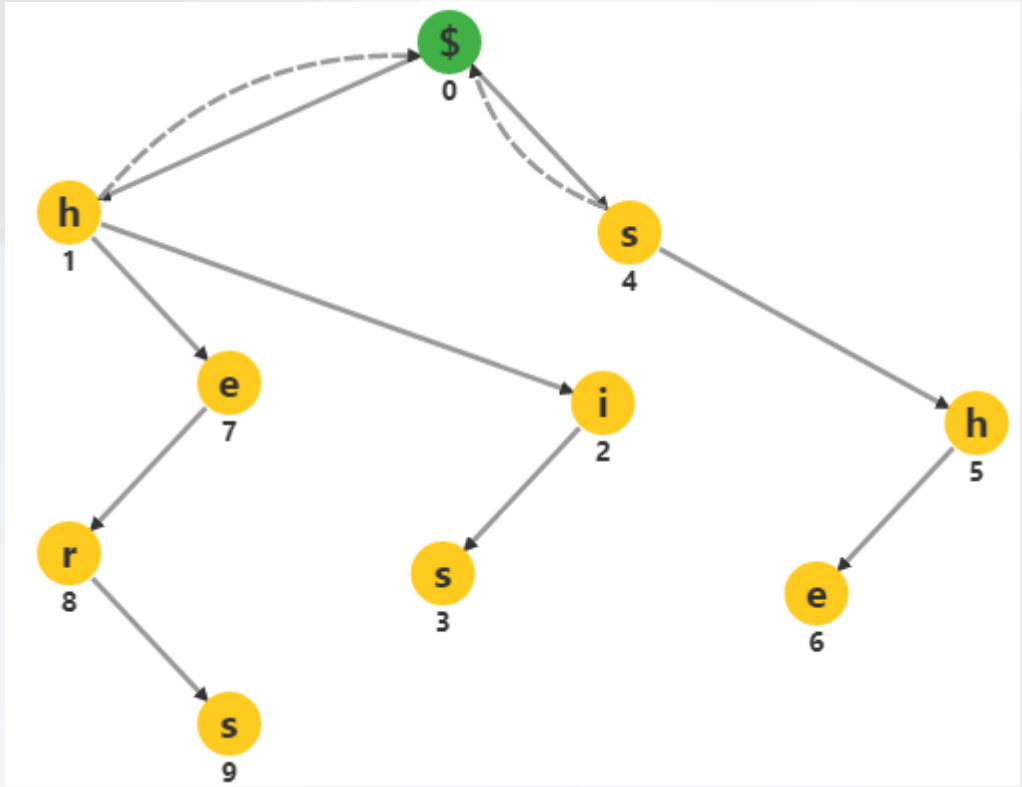
```
44.  int count(char* str) {  // 统计一个字符串在给定
     //的字符串集合（Trie）中出现了多少次
45.      int ret = 0, p = 0;
46.      for (int i = 0; str[i]; ++i) {
47.          p = ch[p][str[i] - 'a'];
48.          int tmp = p;
49.          while (tmp) {
50.              ret += cnt[tmp];
51.              cnt[tmp] = 0;
52.  // 避免重复统计同一字符串
53.              tmp = fail[tmp];
54.          }
55.      }
56.      return ret;
57.  }
58.  };
```

例题：工厂工作手册

- 工厂工作手册，你听说过么？小明把工厂工作手册全部摘抄了下来并把它变成了一个长度不超过 10^5 的字符串 S，小明还有一个包含 n 个单词的列表，列表里的 n 个单词记为 $t_1 \cdots t_N$。他希望从 S 中删除这些单词。
- 小明每次在 S中找到第一个出现的列表中的单词，然后从 S 中删除这个单词。他重复这个操作直到 S 中没有列表里的单词为止。需要注意的是删除一个单词后，后面的紧跟着的字符和前面的字符连接起来可能再次出现列表中出现的单词。并且小明注意到列表中的单词不会出现一个单词是另一个单词子串的情况。
- 请你帮助小明输出删除后的 S。
- 题目解析
- 在进行统计的时候，要记录匹配到字符串中每个位置时 Trie 树上的结点编号，不妨设匹配到字符串下标为 i 时，结点编号为 $match_i$。我们将结点编号依次插入一个栈 s 中，如果发现当前栈顶已经匹配到 Trie 中的终端结点，不妨设其长度为 len，则将栈 s 最顶部的 len 个元素 $match_{top} \ldots match_{top-len+1}$ 都弹出栈，之后将当前搜索用的结点编号设为 $match_{top-len}$。

AC 自动机的演示

初始化



进行初始化过程

新建一个队列 q

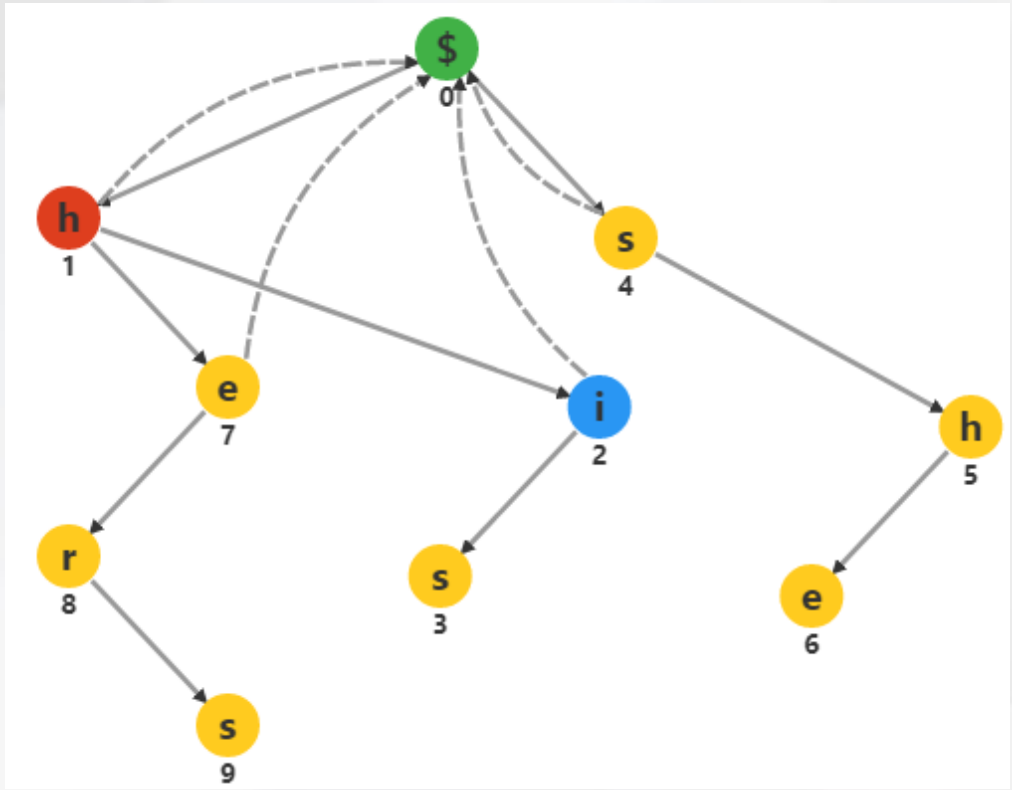
将根节点的孩子加入队列，并将其的 fail 指针指向根节点

完成队列 q 的初始化

BFS 到达节点 1，处理节点 1 的孩子

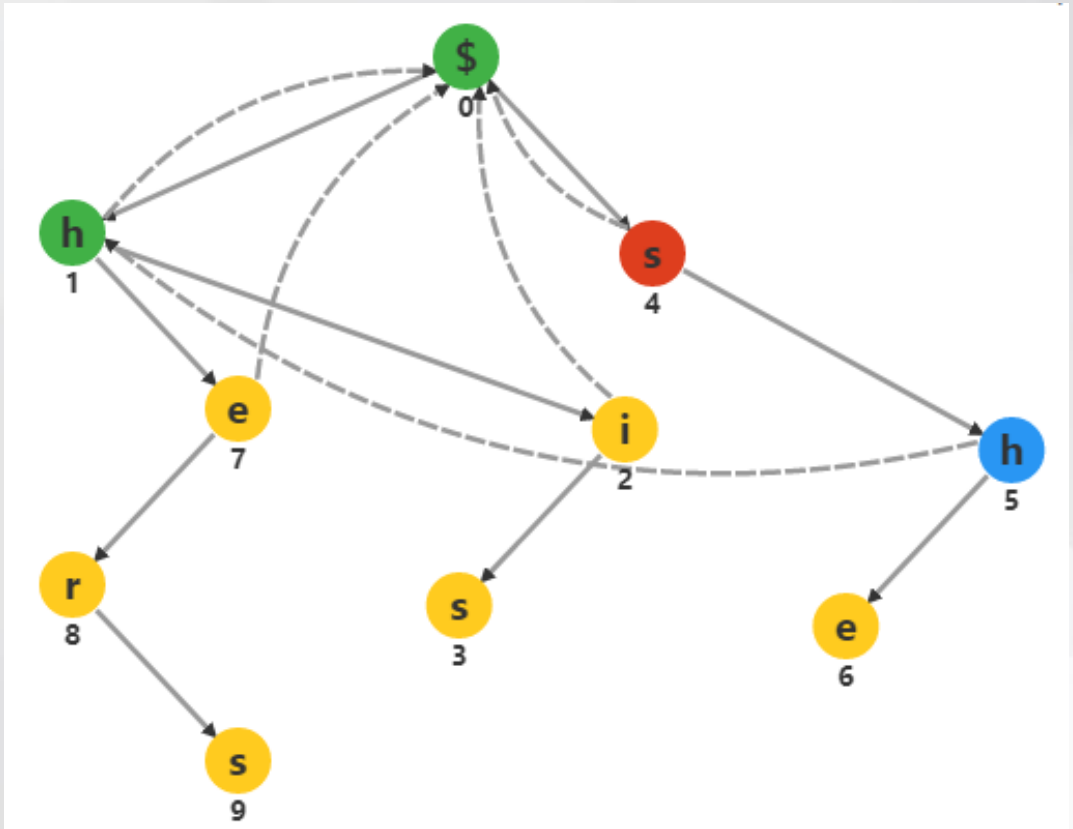
将节点 7 的fail指针指向节点 0，并将其入队

将节点 2 的fail指针指向节点 0，并将其入队



BFS 到达节点 4，处理节点 4 的孩子

将节点 5 的fail指针指向节点 1，并将其入队



BFS 到达节点 7，处理节点 7 的孩子

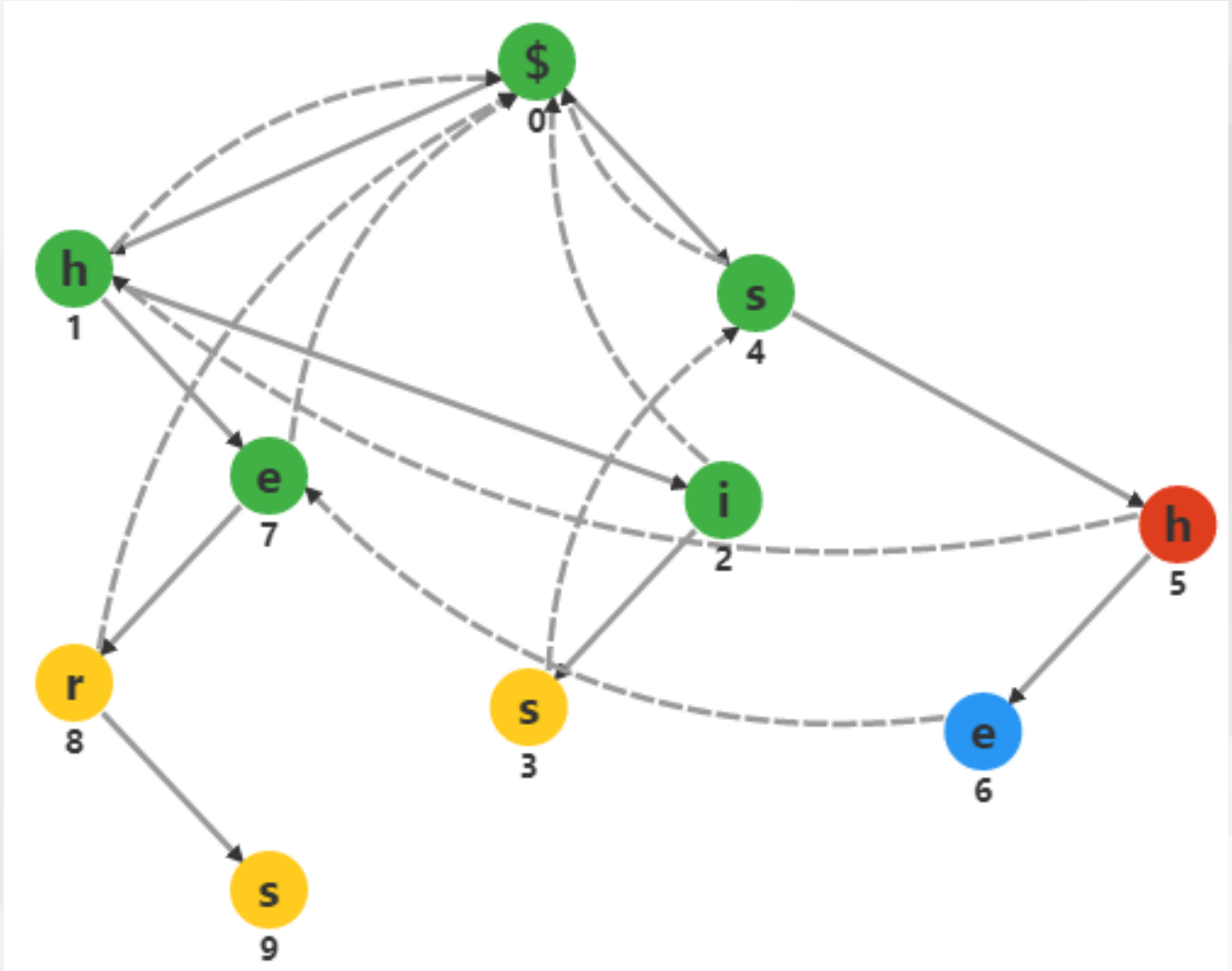
将节点 8 的fail指针指向节点 0，并将其入队

BFS 到达节点 2，处理节点 2 的孩子

将节点 3 的fail指针指向节点 4，并将其入队

BFS 到达节点 5，处理节点 5 的孩子

将节点 6 的fail指针指向节点 7，并将其入队



BFS 到达节点 8，处理节点 8 的孩子

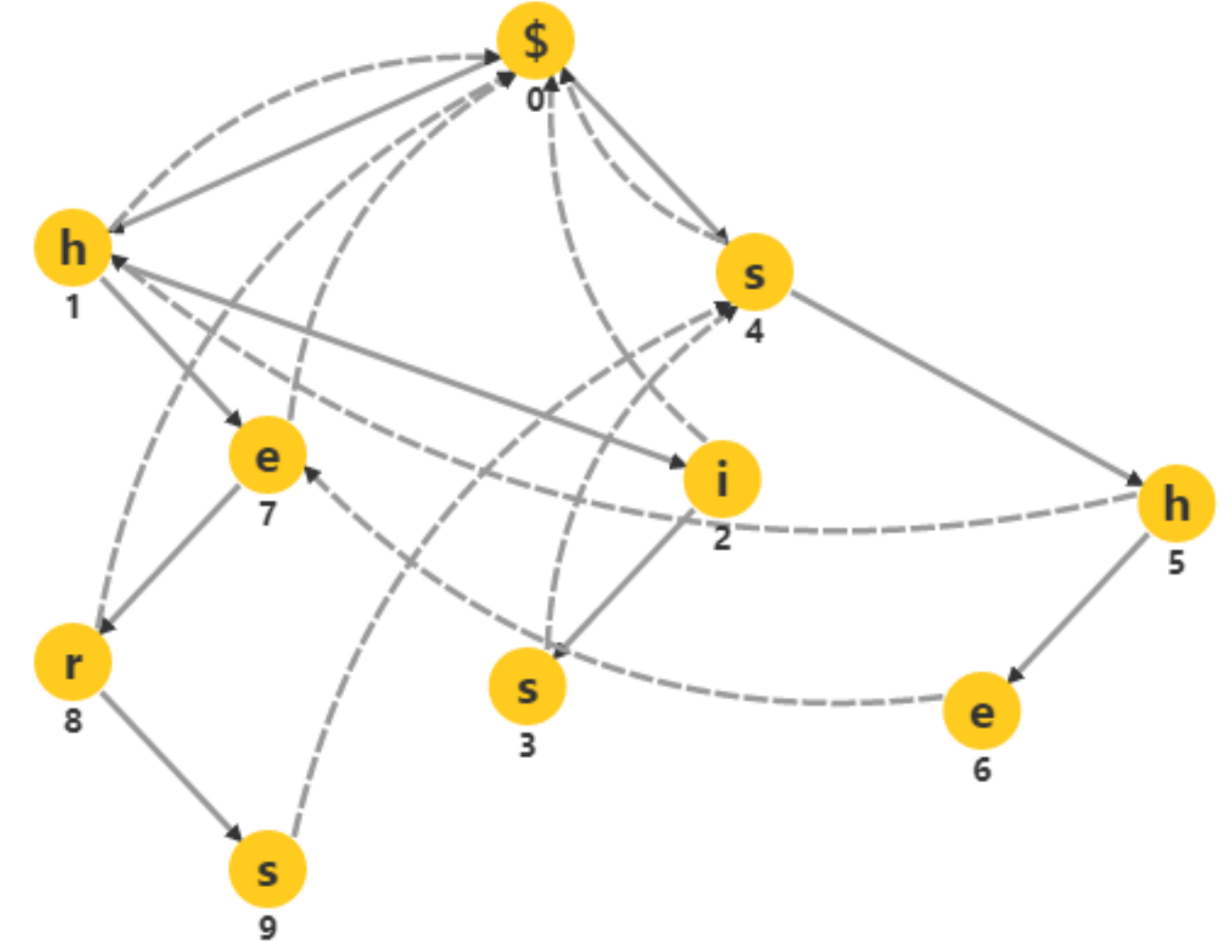
将节点 9 的fail指针指向节点 4，并将其入队

BFS 到达节点 3，处理节点 3 的孩子

BFS 到达节点 6，处理节点 6 的孩子

BFS 到达节点 9，处理节点 9 的孩子

任务完成



习题：猴子打字

- 有一个有趣的定理：无限猴子定理（infinite monkey theorem），它的表述如下：让一只猴子在打字机上随机按键，当按键次数达到无穷时，几乎必然能够打出任何给定的文字。
- 给出一篇猴子打出的“文章”，并给定一个由若干个词组成的词典，问猴子一共打出了多少个在词典中出现的词。
- 输入格式
- 第一行一个整数 n(1≤n≤10000)，表示词典中单词的个数。
- 接下来 n 行，每行一个仅由小写字母组成的单词，长度不超过 50。
- 最后一行是一篇仅由小写字母组成的文章，长度不超过 1000000。
- 输入格式
- 一行一个整数，表示答案。
- 样例输入
- 5
- jsk
- jisuan
- suantou
- love
- program
- jisuantouisprogramming
- 样例输出
- 3

习题：工厂工作手册

- 工厂工作手册，你听说过么？小明把工厂工作手册全部摘抄了下来并把它变成了一个长度不超过 10^5 的字符串 S，小明还有一个包含 n 个单词的列表，列表里的 n 个单词记为 t1…tN。他希望从 S 中删除这些单词。
- 小明每次在 S 中找到第一个出现的列表中的单词，然后从 S 中删除这个单词。他重复这个操作直到 S 中没有列表里的单词为止。需要注意的是删除一个单词后，后面的紧跟着的字符和前面的字符连接起来可能再次出现列表中出现的单词。并且小明注意到列表中的单词不会出现一个单词是另一个单词子串的情况。
- 请你帮助小明输出删除后的 S。
- 输入格式：第一行输入一个字符串 $S(1 \le |S| \le 10^5)$。第二行输入一个整数 $N(1 \le N \le 2000)$。接下来的 N 行，每行输入一个字符串，第 i 行的字符串是 ti。
- N 个字符串的长度和小于 10^5。注意：输入的字符串仅包含小写字母。
- 输出格式：答案输出一行，输出操作后的 S。
- 样例输入
- oorjskorzorzzooorzrzrzr
- 2
- orz
- jsk
- 样例输出
- or

AC 自动机动态规划

- 概述
- AC 自动机相当于在 Trie 树上增加了反向状态转移的边，因而，我们也称 AC 自动机为 Trie 图。Trie 图上每个点都是一个状态，状态之间可以相互转移，进而可以在 Trie 图之上实现动态规划算法，用于求解一类动态规划问题。

例 1

- 给定 n(n≤1000) 个长度不大于 50 的模式串（保证所有的模式串都不相同），一个长度不大于 2000000 的待匹配串，求模式串在待匹配串中的出现次数。
- 解法
- 由于每个病毒串不会完全相同，对于每个病毒串末尾打一个标记。主串在 AC 自动机上跑一遍，最后对模式串末尾的所有状态求和即可。

例 2

- 给定 n (n≤2500) 个长度小于等于 1100 的模式串，求长度不大于 5100000 的目标串 S 中包含的模式串的数目，如果包含了模式串 A 和 B，并且 B 是 A 的子串，那么只记录 A。
- 解法
- 这题与上题类似，需要注意的是，存在包含关系时，只记录 A，不记录 B。
- 对所有模式串建一个 AC 自动机，主串在 AC 自动机上匹配，对所有经过的模式串标记。再用同样的方法，使模式串在 AC 自动机上匹配，对经过的子串模式串消除标记。最后统计一下多少个模式串有标记。

例 3

- 给定 p (p≤10) 个长度不大于 10 的模式串，求长度为 m (m≤50) 的串中不包含任何模式串的串的种类数。
- 解法
- 对所有模式串建一个 AC 自动机，模式串的终止点都是非法的，不可经过的点。
- 用 DP[i,j] 表示长度为 i，状态为 j 的字符串的种类数，枚举所有字符进行状态转移即可，最后 ∑DP[m][i] 即答案。

例 4

- 给定 m（m≤10）个 DNA 片段，每个串长度不超过 10。求长度为 N (N≤2×10^9) 的串中不包括任何给定的 DNA 片段的串的总数。
- 解法
- 对所有模式串建一个 AC 自动机，转移与上题类似。这题的 m 较小，但是长度 N 较大。我们考虑用矩阵乘法加速 DP，时间复杂度 $O((ML)^3 logN)$。

例 5

- 给定 N (N≤10) 个长度不超过 20 的模式串，求一个长度最短的串使得它包含所有的模式串。
- 解法
- 对所有模式串建一个 AC 自动机，串的数量 N 很小，可以状压 DP。二进制状态 $j\&2^k \neq 0$ 表示从根到该结点的路径上有第 k个模式串。用 DP[i,j] 表示状态为 i，二进制状态 j 的最短长度。初始化 DP[0,0]=0，在 Trie 图上求 (0,0) 到 $(i,2^n-1)$点的最短路问题，最后求出来的 $DP[i,2^n-1]$ 就是答案。

习题：情报加密

- 小明是我军的情报专家，现在我军有一份重要情报要发送出去，这是一份特别重要的情报，一旦被敌军截获，里面有一些重要的片段会暴露我们的身份，所以小明需要改变一些字符，这样即使敌军截获了我们的情报，也无法获得正确信息。
- 比如，情报中如果包含AAB,ABC和CAB就会暴露我们的身份，但是情报中的AABCAB把这三个片段都包含了，不过我们只需改变两个字符使其变成ABBCAB，这样就不会暴露任意一个关键片段信息了。注意，我们的情报中只包含A,B,C,D这四个字母，我们修改时也只能使用这四个字母。
- 你要帮助小明通过改变最少的字符来加密情报。
- 输入格式：第一行一个整数 n (n≤50)，表示可能关键的情报段数量。
- 接下来 n 行，每行一个长度不超过 20的字符串，由A，B，C和D构成。
- 最后一行，一个长度不超过 10^3 的字符串是，由A，B，C和D构成。
- 输出格式：一行一个整数，表示最少修改字符的数量。若无解，则输出 –1。
- 样例输入
- 2
- A
- DB
- DBAADB
- 样例输出
- 4

习题：小明的神笔

- 小明在一次郊外出游，捡到一支神笔，神笔能快速的写出一个长度为 m的字符串，字符只由大写字母组成。花椰菜君知道了后，可是相当羡慕嫉妒呀，他给了小明 n个字符串，问小明，一共能写出多少个不同的字符串 S，使得字符串 S 中至少包含 n 个字符串中的一个。
- 输入格式
- 第一行输入两个正整数 n 和 m（1≤n≤60,1≤m≤100），分别表示花椰菜君给的字符串个数以及神笔能写下字符串的长度；
- 接下来输入 n行，每行输入一个字符串，表示花椰菜君给的字符串，长度不超过 100。所有字符串的字符只由大写字母组成，不会包含其他字符。
- 输出格式
- 输出一行，输出一个整数，表示小明一共能写出来的字符串 S 个数。结果可能会很大，输出答案对 10,007 取余的结果皆可。
- 样例输入
- 1 3
- XY
- 样例输出
- 52

习题：工厂抽奖

- 工厂即将举办一个抽奖活动，每个人需要给自己准备一个长度为 M 只包含字母a,b,c,d的字符串，抽奖活动的时候，会宣布一些幸运串，如果某个人的字符串里包含了这个幸运串，那么这个人就可以给自己加相应的分数，每个幸运串最多加一次分数，有趣的是还有一些诅咒串，如果你的字符串中包含了这个诅咒串，也会扣去相应的分数，同样的，每个诅咒串也最多让你扣一次分数。小明也会参加这次抽奖活动，他偷偷地得知了幸运串和诅咒穿有哪些，打算准备一个可以得到最高分的串以确保自己获奖，不过如果所有人的分数都是负分的话，这次抽奖活动就会取消，没有人能获奖。你能帮小明计算出他能获得的最高分数是多少么，又或者这次活动没有人能获奖?
- 输入格式
- 第一行输入两个正整数 N,M(1≤N≤10,1≤M≤100)，分别表示特殊串的个数和每个人可以给自己准备的字符串的长度。接下来 N 行，每行输入一个特殊串和它对应的分数，用空格隔开，输入保证特殊串仅包含字母a,b,c,d，且每个特殊串长度不超过100。
- 输出格式
- 答案输出一行，为小明可以获得的最大的分数，如果这次比赛没人获奖，则输出Unhappy!。
- 样例输入1
- 1 6
- abc 2
- 样例输出1
- 2
- 样例输入2
- 4 3
- a -1
- b -2
- c -3
- d -4
- 样例输出2
- Unhappy!