“十二五”普通高等教育本科国家级规划教材

国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

◎ 王晓东　编著

非外借

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

“十二五”普通高等教育本科国家级规划教材
国家精品课程教材

计算机算法设计与分析习题解答

（第5版）

王晓东 编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

计算机算法设计与分析习题解答/王晓东编著. —5版. —北京：电子工业出版社，2018.10

ISBN 978-7-121-34438-1

Ⅰ. ① 计… Ⅱ. ① 王… Ⅲ. ① 电子计算机－算法设计－高等学校－题解 ② 电子计算机－算法分析－高等学校－题解 Ⅳ. ① TP301.6-44

中国版本图书馆CIP数据核字（2018）第120711号

策划编辑：章海涛
责任编辑：章海涛
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编　100036
开　　本：787×1092　1/16　　印张：22.75　　字数：580千字
版　　次：2005年8月第1版
　　2018年10月第5版
印　　次：2018年10月第1次印刷
定　　价：56.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：192910558（QQ群）。

前　言

一些著名的计算机科学家在有关计算机科学教育的论述中认为，计算机科学是一种创造性思维活动，其教育必须面向设计。“计算机算法设计与分析”正是一门面向设计，且处于计算机学科核心地位的教育课程。通过对计算机算法系统的学习与研究，理解掌握算法设计的主要方法，培养对算法的计算复杂性正确分析的能力，为独立设计算法和对算法进行复杂性分析奠定坚实的理论基础，对每一位从事计算机系统结构、系统软件和应用软件研究与开发的科技工作者都是非常重要和必不可少的。课程结合我国高等学校教育工作的现状，追踪国际计算机科学技术的发展水平，更新了教学内容和教学方法，以算法设计策略为知识单元，在内容选材、深度把握、系统性和可用性方面进行了精心设计，力图适合高校本科生教学对学时数和知识结构的要求。

本书是“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》（ISBN 978-7-121-34439-8）配套的辅助教材，对《计算机算法设计与分析（第5版）》一书中的全部习题做了详尽的解答，旨在让使用该书的教师更容易教，学生更容易学。为了便于对照阅读，本书的章序与《计算机算法设计与分析（第5版）》一书的章序保持一致，且一一对应。

本书的内容是对《计算机算法设计与分析（第5版）》的较深入的扩展，许多教材中无法讲述的较深入的主题通过习题的形式展现出来。为了加强学生灵活运用算法设计策略解决实际问题的能力，本书将主教材中的许多习题改造成算法实现题，要求学生不仅设计出解决具体问题的算法，而且能上机实现。作者的教学实践反映出这类算法实现题的教学效果非常好。作者还结合国家精品课程建设，建立了“算法设计与分析”教学网站。国家精品资源共享课地址：http://www.icourses.cn/sCourse/course_2535.html。欢迎广大读者访问作者的教学网站并提出宝贵意见。

在本书编写过程中，福州大学“211工程”计算机与信息工程重点学科实验室为本书的写作提供了优良的设备与工作环境。电子工业出版社负责本书编辑出版工作的全体同仁为本书的出版付出了大量辛勤劳动，他们认真细致、一丝不苟的工作精神保证了本书的出版质量。在此，谨向每位曾经关心和支持本书编写工作的各方面人士表示衷心的谢意！

作　者

目　录

第 9 章　串与序列的算法

算法分析题 9

9-1　简单子串搜索算法最坏情况复杂性。

试说明简单子串搜索算法在最坏情况下的计算时间复杂性为 $O(m(n-m+1))$。

分析与解答：考察一个特例。

设 $t=a^n$（由连续 n 个 a 组成的串），$p=a^{m-1}b$。

在算法 naive 中，对于第 1 个循环的每个 i，都需要对 p 做 m 次比较。因此，总比较次数为 $m(n-m+1)$。由此可见，简单子串搜索算法在最坏情况下的计算时间复杂性为 $O(m(n-m+1))$。当 $m=n/2$ 时，所需计算时间为 $O(n^2)$。

9-2　后缀重叠问题。

设 x，y 和 z 是 3 个串，且满足 $x \sqsupset z$ 和 $y \sqsupset z$。试证明：

（1）若 $|x| \leqslant |y|$，则 $x \sqsupset y$。

（2）若 $|x| \geqslant |y|$，则 $y \sqsupset x$。

（3）若 $|x| = |y|$，则 $x=y$。

分析与解答：从图 9-1 容易看出结论的正确性。

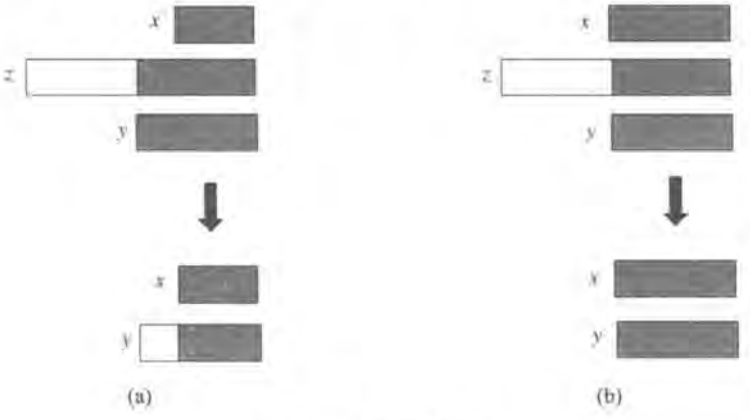

图 9-1　后缀重叠问题

9-3　改进前缀函数。

KMP 算法通过模式串的前缀函数，较好地利用了搜索过程中的部分匹配信息，从而提高了效率。然而在某些情况下，还可以更好地利用部分匹配信息。例如，考察图 9-2 中，KMP 算法对主串 aaabaaaab 和模式串 aaaab 的搜索过程。

在图 9-2(a)中匹配失败后，按前缀函数指示继续作了图(b)～(d)的比较后，最后在图(e)找到一个匹配。事实上，图(b)～(d)的比较都是多余的。因为模式串在位置 0、1、2 处的字符和位置 3 处的字符都相等，因此不需要再和主串中位置 3 处的字符比较，而可以将模式一次

向右滑动 4 个字符，直接进入图(e)的比较。这就是说，在 KMP 算法中遇到 p[j+1]≠t[i]，且 p[j+1]=p[next[j]+1]时，可一次向右滑动 j-next[next[j]]个字符，而不是 j-next[j]个字符。根据此观察，设计一个改进的前缀函数，使得遇到上述特殊情况时效率更高。

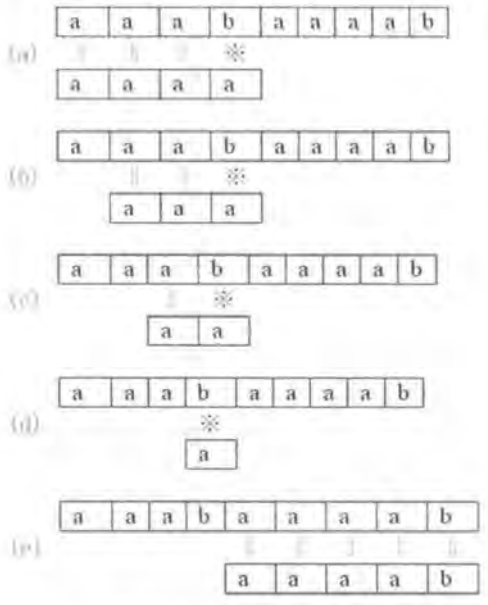

图 9-2　改进前缀函数

分析与解答：根据对此特例的观察，可将前缀函数修正为 next' 如下。

$$next'[q]=\begin{cases}next'[next[q]] & p[next[q]+1]=p[q+1],\ next[q]>-1\\ next[q] & 其他\end{cases}$$

相应的计算模式串的前缀函数的算法可修改如下。

```
1   void mbuild(const string& p, int *next) {                    // 改进的前缀函数
2     int  *f = new int[m];
3     build(p, f);
4     next[0] = -1;
5     for(int i=1; i <= m-1; i++) {
6       int  j = f[i];
7       if(j<0 || p[j] != p[i])
8         next[i] = j;
9       else
10        next[i] = next[j];
11    }
12    delete []f;
13  }
```

将算法 KMP-Matcher 的第 5 行的 build 换作 mbuild，就可用修正后的前缀函数来搜索子串，从而得到一个改进的 KMP 算法。

9-4　确定所有匹配位置的 KMP 算法。

修改算法 KMP-Matcher，使其能找到模式串 p 在主串 t 中的所有匹配位置。

分析与解答：找到一个匹配位置后，可以利用前缀函数 next 的性质，继续比较 t[i]与 p[next[j]+1]。修改后的算法如下。

```
void KMP-Matcher (const string& t, const string& p) {        // 改进的KMP算法
   int  *next = new int[m];
   build(p, next);
   int  j = -1;
   for(int i=0; i < n; i++) {
      while(j > -1 && p[j+1] != t[i])
         j = next[j];
      if(p[j+1] == t[i])
         j++;
      if(j == m-1) {
         output(i-m+1);
         j = next[j];
      }
   }
}
```

9-5　特殊情况下简单子串搜索算法的改进。

假设模式串 p 中所有的字符均不相同，说明如何修改简单子串搜索算法，使其计算时间为 $O(n)$，其中 n 为主串 t 的长度。

分析与解答：显而易见，如果 p 在 t 中多次出现，这些子串均不重叠。因此，当 t[i+j]≠p[j]时，可以从 t[i+j+1]开始与 p 进行比较。这相当于指针 i 不再回溯。因为主串 t 中每个字符最多只比较 2 次，故总比较次数小于 $2n$，即计算时间为 $O(n)$。

修改后的算法如下。

```
int naive(const string& t, const string& p) {       // 特殊情况下简单子串搜索算法
   n = t.length();
   m = p.length();
   int  i = 0;
   while(i <= n-m) {
      int  j = 0;
      while(j < m && t[i+j] == p[j])
         j++;
      if(j == m)
         return i;
      i = i+j+1;
   }
   return -1;
}
```

9-6　简单子串搜索算法的平均性能。

设主串 t 和模式串 p 分别是由 d（$d \geqslant 2$）元字符集 $\sum_d = \{0,1,\cdots,d-1\}$ 中随机字符组成的长度为 n 和 m 的字符串。试证明简单子串搜索算法所做比较次数的期望值为

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leqslant 2(n-m+1)$$

由此可见，对于随机选取的字符串，简单子串搜索算法还是十分有效的。

分析与解答：在简单子串搜索算法中，对每个不同的 i（$0 \leqslant i \leqslant n-m$）有如下结论。p[$j$]需

要与 t[i+j]进行比较的概率是 $p[0..j-1]=t[i..i+j-1]$ 的概率，即 $1/d^j$。也就是说，对每个 i，算法的期望比较次数均为 $\sum_{j=0}^{m-1}\frac{1}{d^j}$。因此算法的总期望比较次数为 $(n-m+1)\sum_{j=0}^{m-1}\frac{1}{d^j}$。由此可知：

$$(n-m+1)\sum_{j=0}^{m-1}\frac{1}{d^j}=(n-m+1)\frac{d^{-m}-1}{d^{-1}-1}$$

$$\leqslant (n-m+1)\frac{1}{1-d^{-1}} \leqslant (n-m+1)\frac{1}{1-\frac{1}{2}}$$

$$= 2(n-m+1)$$

9-7　带间隙字符的模式串搜索。

假设允许模式串 p 中可以出现能与任意字符串（包括长度为 0 的空串）匹配的间隙字符◇，如模式串 ab◇ba◇c 可在主串 cabccbacbacab 中产生如图 9-3 所示的匹配。间隙字符◇可在模式串中出现任意多次，但不允许在主串中出现。

图 9-3　带间隙字符的模式串

试设计一个多项式时间算法，确定在主串中能否找到与模式串 p 匹配的子串，并分析算法的计算时间复杂性。

分析与解答：由于◇可与主串中任意多个字符匹配，因此只要考察 p 中被◇分隔开的各个子串能否按序在主串 t 中找到匹配即可。这就转化成有限个 p 中的非◇字符子串在主串中的匹配问题。

假设 p 被分解成 $a_1◇a_2\cdots◇a_k$，先找到 a_1 在主串中的最左端，再从其右端开始找 a_2 的最左端……直到找到 a_k。如果用简单子串搜索算法实现这个过程，则需要 $O(nm)$ 计算时间。其中，n 是主串 t 的长度，m 是模式串 p 的长度。如果用 KMP 算法实现，则只要 $O(n+m)$ 计算时间。

9-8　串接的前缀函数。

设模式串 p 和主串 t 的串接为 pt。试说明，如何利用 pt 的前缀函数来计算模式串 p 在主串 t 中出现的位置。

分析与解答：如果 p 在 t 中出现，则存在 $k\geqslant 0$，使得 $t[k..k+m-1]=p[0..m-1]$。其中，m 为串 p 的长度，这说明 p 是 pt[0..$k+2m-1$] 的真前缀，又是真后缀。因此，只要查找 pt 的前缀函数 next 的值。若存在 $i\geqslant 2m-1$ 且 next[i] $\geqslant m-1$，则 p 在位置 i-next[i]出现，即在 t 中的 i-next[i]-m 位置出现。

```
1   int conc(const string& t, const string& p) {          // 串接的前缀函数
2      int  *next = new int[n+m];
3      string  pt = p+t;
4      build(pt, next);
```

```
5     for(int i=0; i < n+m; i++)
6       if(i >= 2*m-1 && next[i] >= m-1)
7         return i-next[i]-m;
8     return -1;
9   }
```

另一个类似的方法是用一个在 p 和 t 中均未出现的字符 c 连接 p 和 t 为 pct。pct 的前缀函数 next 的值最大为 $m-1$。在任何一处出现 next[i] = $m-1$ 的位置 i，就是 p 出现的右端位置。

```
1   int conc(const string& t, const string& p) {      // 串接的前缀函数
2     int  *next = new int[n+m+1];
3     string  pt = p+"*"+t;
4     build(pt, next);
5     for (int i=0; i < n+m+1; i++)
6       if(next[i] == m-1)
7         return i-next[i]-m-1;
8     return -1;
9   }
```

9-9　串的循环旋转。

试设计一个线性时间算法确定一个串 t 是否为另一串 t'的循环旋转。例如，arc 与 car 互为循环旋转。

分析与解答：容易证明，t 是 t'的循环旋转，当且仅当 t'是 tt 的子串。

用 KMP 算法可在线性时间内计算如下。

```
1   bool cyclic(const string& t,const string& p) {    // 串的循环旋转
2     string  t2 = t+t;
3     n = t2.length();
4     m = p.length();
5     return (n/2 == m && KMP-Matcher(t2, p) > -1);
6   }
```

9-10　失败函数性质。

在字符串集合 P 的 AC 自动机 T 中，状态结点 s 所表示的字符串是从根结点到 s 的路径上各边的字符依次连接组成的字符串 α(s)。设 s 和 t 是 T 中两个结点，且 u=α(s)，v=α(t)。试证明，f(s)=t 当且仅当 v 是字符串 p_i（$0 \leqslant i < k$）的所有前缀中 u 的最长真后缀。

分析与解答：对 u 的长度 $|u|$ 用数学归纳法。当 $|u|=1$ 时，s 是 T 中第一层结点。对于 T 中所有第一层结点 s'均有 f(s')=0，因而 f(s)=0，即 t=0，v=e。因此当 $|u|=1$ 时结论成立。

设结论对所有长度小于 j 的字符串成立。

当 $|u|=j$ 时，设 $u=a_1a_2\cdots a_j$，且 v 是字符串 p_i（$0 \leqslant i < k$）的所有前缀中 u 的最长真后缀。进一步设状态结点 r 表示的字符串是 $a_1a_2\cdots a_{j-1}$，s 表示的字符串是 u。r_1，r_2，…，r_q 是满足如下条件的结点序列。

$$\begin{cases} r_1 = f(r) & \\ g(r_i, a_j) = -1 & 1 \leqslant i < q \\ r_{i+1} = f(r_i) & 1 \leqslant i < q \\ g(r_q, a_j) = t & \end{cases}$$

此序列实际上是算法 build_failure 计算失败函数时产生的结点序列。算法在 while 循环结束后取 $f(s)=t$。

设 t 表示的字符串为 v，则 v 就是字符串 p_i（$0\leqslant i<k$）的所有前缀中 u 的最长真后缀。

事实上，设 r_i 表示的字符串为 v_i（$0\leqslant i\leqslant q$）。由归纳假设知，v_1 是 $a_1a_2\cdots a_{j-1}$ 的最长真后缀；v_2 是 v_1 的最长真后缀……v_q 是 v_{q-1} 的最长真后缀。因此，v_q 是 $a_1a_2\cdots a_{j-1}$ 的最长真后缀，且 $v_q a_j$ 是 P 中某个字符串的前缀，从而 $v_q a_j$ 是字符串 p_i（$0\leqslant i<k$）的所有前缀中 u 的最长真后缀。算法 build_failure 中取 $f(s)=g(r_q,a_j)=t$。

由数学归纳法即知结论成立。

9-11 输出函数性质。

设 s 是字符串集合 P 的 AC 自动机中的状态结点，且 $u=\alpha(s)$。试证明，$v\in output(s)$ 当且仅当 $v\in P$ 且 v 是 u 的后缀。

分析与解答：从算法 insert 容易看出，输入字符串 p_i 后，在相应的叶结点 s_i 处，$output(s_i)=p_i$。因此，在算法计算转向函数 g 结束时，结论成立。

在算法计算失败函数阶段，可以对结点层数用数学归纳法证明结论也成立。

在根结点处已经证明了结论成立。假设对所有层数小于 d 的结点结论成立。

设 s 是 d 层结点且表示的字符串是 u。对任一 $v\in output(s)$，如果 v 是算法计算转向函数时加入 output(s)，则 $v\in P$。如果 v 是算法计算失败函数时加入 output(s)，则 $v\in output(f(s))$。由习题 9-10 的结论知，v 是 u 的后缀。

反之，设 $v\in P$ 是 u 的后缀。由于 $v\in P$，因此存在状态结点 t，使得 t 表示的字符串是 v。由算法 insert 知，$v\in output(t)$。因此，当 v=s 时，有 t=s，自然 $v\in output(s)$。当 v 是 u 的真后缀时，由归纳假设可知 $v\in output(f(s))$。在算法 build_failure 的第 15 行，$output(f(s))$ 合并到 output(s)。因此，$v\in output(s)$。由数学归纳法即知结论成立。

9-12 后缀数组类。

试设计一个后缀数组类。用倍前缀算法构造后缀数组，并支持以下运算：

（1）length()　　返回后缀数组长度。

（2）select(int i)　　返回 sa[i]。

（3）index(int i)　　返回 rank[i]。

（4）llcp(int i)　　返回 lcp[i]。

分析与解答：用倍前缀算法构造后缀数组类的描述如下。

```
1  class suffix {                          // 倍前缀算法构造后缀数组
2    #define maxn 33003
3  private:
4    int  m, n;
5    int  a[maxn], b[maxn], cnt[maxn], t[maxn];
6    int  cmp(int *t, int u, int v, int l);
7    void radix(int *x, int *y, int *z, int n, int m);
8    void sort2(int *x, int *y, int h);
9    void doubling(int *t);
10   void kasai(int *t, int n);
11   int lce(int l, int r, int n);
12 public:
```

```
13    int sa[maxn], rank[maxn], lcp[maxn];
14    suffix(string txt) {
15       m = 128, n = txt.length();
16       for(int i=0; i < n; i++)
17          t[i] = txt[i];
18       doubling(t);
19       kasai(t, n);
20    }
21    int length() {  return n;  }
22    int select(int i) {  return sa[i];  }
23    int index(int i) {  return rank[i];  }
24    int llcp(int i) {  return lcp[i];  }
25 };
```

9-13　最长公共扩展查询。

试说明如何对最长公共前缀数组 lcp 做适当预处理，使得最长公共扩展查询在最坏情况下需要 $O(1)$时间。

分析与解答：如果事先计算好最长公共前缀数组 lcp 的所有查询 rmq(i,j)，并存入一个二维数组 dp，就可以做到 $O(1)$时间响应。例如：

```
1  class rmq {                    // 区间最小查询
2     typedef vector<int> vct;
3     int  n, *a;
4     vector<vct> dp;
5     void build() {
6     for(int i=0; i < n; i++)
7        dp[i][i] = i;
8     for(int i=0; i < n; i++) {
9        for(int j=i+1; j < n; j++) {
10          if(a[dp[i][j-1]] > a[j])
11             dp[i][j] = j;
12          else
13             dp[i][j] = dp[i][j-1];
14       }
15    }
16 public:
17    rmq(int *x, int len) {
18       n = len;
19       a = x;
20       dp.resize(n);
21       for(int i=0; i < n; i++)
22          dp[i].resize(n);
23          build();
24    }
25    int query(int l,int r) {
26       return a[dp[l][r]];
27    }
28 };
```

其中，build()函数用动态规划算法将所有查询 rmq(i,j)保存到数组 dp 中，查询时直接返回查询值。预处理时间为 $O(n^2)$，需要空间也是 $O(n^2)$。

用稀疏表（Sparse Table）算法可以将预处理时间和空间降为 $O(n\log n)$。其基本思想是，对$(i,j)(0\leqslant i<n-1, 0\leqslant j<\log n)$，将区间$[i,2^j]$的最小值保存在 dp[$j$][$i$]中。数组 dp 需要 $O(n\log n)$空间。用动态规划算法可以在 $O(n\log n)$时间计算数组 dp。

$$dp[j][i]=\begin{cases}dp[j-1][i] & lcp[dp[j-1][i]\leqslant dp[j-1][i+2^{j-1}-1]\\ dp[j-1][i+2^{j-1}-1] & 其他\end{cases}$$

在响应查询 rmq(i,j)时，选择覆盖查询区间$[i,j]$的两个已经计算最小值的区间如下。

设 k=[log($j-i$)]，则区间$[i, i+2^k-1]$和$[j-2^k+1, j]$覆盖查询区间$[i,j]$，因此 min{dp[k][i], dp[k][$j-2^k+1$]}就是 rmq(i,j)的值。

稀疏表（Sparse Table）算法可以实现如下。

```
class rmq {                          // 区间最小查询稀疏表（Sparse Table）算法
   typedef vector<int> vct;
private:
   int  n;
   vct  a, ad, dp[20];
   void prep() {
      int  k = (int)floor(log(n)/log(2.0));
      for(int i=1; i <= k; i++)
         for(int j=0; j+(1<<i)-1 < n; j++)
            dp[i][j] = min(dp[i-1][j], dp[i-1][j+(1<<(i-1))]);
   }
public:
   rmq(int *x, int len) {
      n = len;
      a.resize(n);
      ad.resize(n+1);
      for(int i=0; i < 20; i++)
         dp[i].resize(n);
         for(int i=0; i < n; i++)
            dp[0][i] = x[i];
               prep();
   }
   int query(int l, int r) {
      int  k = (int)floor(log(r-l+1)/log(2.0));
      return min(dp[k][l], dp[k][r-(1<<k)+1]);
   }
};
```

用区间块分割算法可以将预处理时间和空间降为 $O(n)$，但查询响应时间增加为 $O(\sqrt{n})$。其基本思想是将数组划分为 $\sqrt{n}$ 块，每块中有 $\sqrt{n}$ 个元素。事先计算好每块的最小值。查询时先按块查询，然后再块内查询。

```
class rmq {                          // 区间最小查询区间块分割算法
   typedef vector<int> vct;
   int  n, *a;
```

```
    vector<int> sp;
    void build() {
      int  size_m = 0;
      for(int i=0; i < n; ) {
        int  minindex = i;
        for(int j=0; j < (int)sqrt(n) && i < n; j++) {
          if(a[i] < a[minindex])
            minindex = i;
          i++;
        }
        sp[size_m++] = minindex;
      }
    }
public:
    rmq(int *x, int len) {
      n = len;
      a = x;
      sp.resize(n);
      build();
    }
    int query(int l, int r) {
      if(l == r)
        return a[l];
      int  b = l/(int)sqrt(n), e = r/(int)sqrt(n);
      int  ans = a[l];
      for(int i=l; i < (b+1)*sqrt(n); i++)
        if(a[i]<ans)
          ans = a[i];
      for(int i=b+1; i < e; i++)
        if(a[sp[i]] < ans)
          ans = a[sp[i]];
      for(int i=e*sqrt(n); i <= r; i++)
        if(a[i] < ans)
          ans = a[i];
      return ans;
    }
};
```

用线段树来表示查询区间可以将预处理时间和空间降为 $O(n)$，但查询响应时间为 $O(\log n)$。

```
class rmq {                    // 区间最小查询线段树算法
private:
    int  n, *a;
    vector<int> seg;
    void build(int node, int start, int end) {
      if(start == end)
        seg[node] = start;
      else {
        build(2*node+1, start, (start+end)/2);
```

```
        build(2*node+2, (start+end)/2+1, end);
        if(a[seg[2*node+1]] < a[seg[2*node+2]])
          seg[node] = seg[2*node+1];
        else
          seg[node] = seg[2*node+2];
      }
    }
    int search(int node, int start, int end, int s,int e) {
      if(s <= start && e >= end)
        return seg[node];
      else if(s > end || e < start)
        return -1;
      int  q1 = search(2*node+1, start, (start+end)/2, s, e);
      int  q2 = search(2*node+2, (start+end)/2+1, end, s, e);
      if(q1 == -1)
        return q2;
      else if(q2 == -1)
        return q1;
      if(a[q1] < a[q2])
        return q1;
      return q2;
    }
  public:
    rmq(int *x, int len) {
      n = len;
      a = x;
      seg.resize(2*n+1);
      build(0, 0, n-1);
    }
    int query(int s, int e) {
      int  ret = search(0, 0, n-1, s, e);
      if(ret >= 0)
        return a[search(0, 0, n-1, s, e)];
      else
        return INT_MAX;
    }
};
```

在计算最长公共扩展时，先计算后缀数组和最长公共前缀数组 lcp，然后建立 lcp 的 rmq 类，就可以在 $O(1)$ 时间内响应最长公共扩展查询。

9-14 最长公共扩展性质。

设字符串 t 的后缀数组和最长公共前缀数组分别为 sa 和 lcp。对于非负整数 $0 \leqslant l \leqslant r$，t 的后缀 S_l 和 S_r 的最长前缀的长度为 $\mathrm{lce}(l,r)$。设 $x=\mathrm{sa}^{-1}[l]$，$z=\mathrm{sa}^{-1}[r]$，则 $\mathrm{sa}[x]=l$，$\mathrm{sa}[z]=r$。不失一般性，可设 $x<z$。试证明 $\mathrm{lce}(l,r)$ 具有如下性质。

$$\mathrm{lce}(l,r)=\min_{x\leqslant y<z}\{\mathrm{lce}(\mathrm{sa}[y],\mathrm{sa}[y+1])\}=\min_{x\leqslant y<z}\{\mathrm{lcp}[y]\} \tag{9.1}$$

分析与解答：首先，对于任意 $0\leqslant x<y<z$，有

$$\mathrm{lce}(\mathrm{sa}[z],\mathrm{sa}[z])=\min\{\mathrm{lce}(\mathrm{sa}[x],\mathrm{sa}[y]),\mathrm{lce}(\mathrm{sa}[y],\mathrm{sa}[z])\}=\delta \tag{9.2}$$

事实上：

（1）按照δ的定义，有

$$\begin{cases} \mathrm{t[sa}[x]..\mathrm{sa}[x]+\delta-1]=\mathrm{t[sa}[y]..\mathrm{sa}[y]+\delta-1] \\ \mathrm{t[sa}[y]..\mathrm{sa}[y]+\delta-1]=\mathrm{t[sa}[z]..\mathrm{sa}[z]+\delta-1] \end{cases} \tag{9.3}$$

从而$\mathrm{t[sa}[x]..\mathrm{sa}[x]+\delta-1]=\mathrm{t[sa}[z]..\mathrm{sa}[z]+\delta-1]$，即

$$\mathrm{lce(sa}[x],\mathrm{sa}[z])\geqslant\delta \tag{9.4}$$

（2）由$0\leqslant x<y<z$可知，$S_{\mathrm{sa}[x]}<S_{\mathrm{sa}[y]}<S_{\mathrm{sa}[z]}$，因此$\mathrm{t[sa}[x]+\delta]\leqslant\mathrm{t[sa}[y]+\delta]\leqslant\mathrm{t[sa}[z]+\delta]$。

如果$\mathrm{t[sa}[x]+\delta]=[\mathrm{sa}[z]+\delta]$，则必有$\mathrm{t[sa}[x]+\delta]=[\mathrm{sa}[y]+\delta]=[\mathrm{sa}[z]+\delta]$，由此可得$\mathrm{lce(sa}[x],\mathrm{sa}[y])>\delta$且$\mathrm{lce(sa}[y],\mathrm{sa}[z])>\delta$。这与$\delta$的定义矛盾。

由此可见，$t[\mathrm{sa}[x]+\delta]<[\mathrm{sa}[z]+\delta]$，也就是说

$$\mathrm{lce(sa}[x],\mathrm{sa}[z])\leqslant\delta \tag{9.5}$$

结合式（9.4）和式（9.5）即知$\mathrm{lce(sa}[x],\mathrm{sa}[z])=\delta$。

据此对$z-x$用数学归纳法就可以证明式（9.1）成立。事实上，当$z-x=1$时，式（9.1）显然成立。假设当$z-x\leqslant m$时式（9.1）成立。当$z-x=m+1$时，任取$x<p<z$，则有$z-p\leqslant m$且$p-x\leqslant m$。由式（9.2）和归纳假设即知

$$\begin{aligned} \mathrm{sa}[z])\} &= \min\{\min_{x\leqslant y<p}\{\mathrm{lce(sa}[y],\mathrm{sa}[y+1])\},\min_{p\leqslant y<z}\{\mathrm{lce(sa}[y],\mathrm{sa}[y+1])\}\} \\ &= \min_{x\leqslant y<p}\{\mathrm{lce(sa}[y],\mathrm{sa}[y+1])\} \end{aligned}$$

由数学归纳法即知式（9.1）成立。

9-15 后缀数组性质。

设字符串 t 的后缀数组和最长公共前缀数组分别为 sa 和 lcp。数组 h 定义为$\mathrm{h}[i]=\mathrm{lcp}[\mathrm{sa}^{-1}[i]]$。试证明，如果$h[i]>1$，则

$$\mathrm{h}[i+1]\geqslant\mathrm{h}[i]-1 \tag{9.6}$$

分析与解答： 设$p=\mathrm{rank}[i]$，$q=\mathrm{rank}[i+1]$，$j=\mathrm{sa}[p+1]$，$k=\mathrm{sa}[q+1]$。

（1）

$$h[i+1]\geqslant h[i]-1\Leftrightarrow\mathrm{lce}(i+1,k)\geqslant\mathrm{lce}(i,j)-1 \tag{9.7}$$

事实上，由h和lcp的定义知

$$\begin{cases} h[i]=\mathrm{lcp}[p]=\mathrm{lce(sa}[p],\mathrm{sa}[p+1])=\mathrm{lce}(i,j) \\ h[i+1]=\mathrm{lcp}[q]=\mathrm{lce(sa}[q],\mathrm{sa}[q+1])=\mathrm{lce}(i+1,k) \end{cases} \tag{9.8}$$

（2）从式（9.1）可看出

$$x<y\leqslant z\Rightarrow\mathrm{lce(sa}[x],\mathrm{sa}[z])\leqslant\mathrm{lce(sa}[y],\mathrm{sa}[z]) \tag{9.9}$$

（3）如果$\mathrm{lce}(i,j)>1$，则

$$\mathrm{lce}(i+1,k)\geqslant\mathrm{lce}(i+1,j+1) \tag{9.10}$$

事实上，由$p=\mathrm{rank}[i]<p+1=\mathrm{rank}[j]$可知，$S_i<S_j$。又由于$\mathrm{lce}(i,j)>1$知$t[i]=t[j]$，因此$S_{i+1}<S_{j+1}$，即$\mathrm{rank}[i+1]<\mathrm{rank}[j+1]$，等价于$\mathrm{rank}[i+1]+1\leqslant\mathrm{rank}[j+1]$。

因为$\mathrm{rank}[i+1]+1=\mathrm{rank}[k]$，所以

$$\mathrm{rank}[i+1]<\mathrm{rank}[k]\leqslant\mathrm{rank}[j+1] \tag{9.11}$$

根据式（9.9）的结论就有$\mathrm{lce}(i+1,k)\geqslant\mathrm{lce}(i+1,j+1)$。

（4）根据式（9.8）的结论可知

$$\begin{cases} h[i] = \mathrm{lce}(i,j) > 1 \\ h[i+1] = \mathrm{lce}(i+1,k) \end{cases} \tag{9.12}$$

根据式(9.10)的结论有 $\mathrm{lce}(i+1,k) \geqslant \mathrm{lce}(i+1,j+1) = \mathrm{lce}(i,j)-1$。换句话说，$h[i+1] \geqslant h[i]-1$。

9-16 后缀数组搜索。

字符串 t 和 p 的长度分别为 m 和 n。t 的后缀数组为 sa。请说明如何利用 t 的后缀数组搜索给定字符串 p 在 t 中出现的所有位置。要求算法在最坏情况下的时间复杂性为 $O(m\log n)$。

分析与解答：由于 t 的后缀数组 sa 是排好序的，字符串 p 在 t 中出现的所有位置在后缀数组 sa 中是连续排列的，因此可以对后缀数组 sa 用二分搜索算法找到这个连续排列的开始位置和结束位置。

首先，用二分搜索算法找到 sa 的最小位置 i，使得 p 是 sa[i]的前缀。如果找不到最小位置 i，则说明 t 不含字符串 p。否则从最小位置 i 开始用二分搜索算法找到 sa 的最大位置 j，使得 p 是 sa[j]的前缀。这样就找到了 p 在 t 中出现的所有位置 sa[$i..j$]。

具体算法描述如下。

```
1  int lower(string text,int *sa,string p) {                  // 后缀数组最小位置搜索
2     int  lft = 0, len = text.size();
3     while(len > 0) {
4          int  half = len>>1, mid = lft+half;
5          int  res = cmp(text, p, mid);
6         if(res < 0)
7              lft = mid+1, len = len-half-1;
8          else
9              len = half;
10    }
11    return lft;
12 }
13 int upper(string text, int *sa, string p, int lft) {     // 后缀数组最大位置搜索
14    int  len = text.size()-lft-1;
15    while(len > 0) {
16       int  half = len>>1, mid = lft+half;
17       int  res = cmp(text, p, mid);
18       if(res > 0)
19          len = half;
20       else
21          lft = mid+1, len = len-half-1;
22    }
23    return lft;
24 }
```

算法 lower 和 upper 分别用于寻找 sa 的最小位置 i 和最大位置 j，与第 2 章中介绍的二分搜索算法如出一辙。其中的比较函数 cmp()用于比较 $S_{\mathrm{sa[mid]}}$ 和 p。

```
1  int cmp(string text,string p,int j) {                      // 比较函数
2     return text.compare(sa[j], p.size(),p);
3  }
```

结合 lower()和 upper()函数就可以找到 p 在 t 中出现的所有位置。

```
1  int search(string text,string p) {        // 后缀数组搜索
2    int  l = lower(text,sa,p);
3    if(text.compare(sa[l], p.size(),p) != 0)
4      return -1;
5    int  r = upper(text, sa, p, l);
6    if(r-l > 0)
7      out(l, r);
8    return r-l;
9  }
```

算法的第 3 行判断 t 是否含字符串 p。

由于在最坏情况下比较函数 cmp 需要 $O(m)$ 时间，因此整个算法需要的计算时间是 $O(m\log n)$。

9-17 后缀数组快速搜索。

设字符串 t 和 p 的长度分别为 m 和 n。t 的后缀数组和最长公共前缀数组分别为 sa 和 lcp。请说明如何利用 t 的后缀数组和最长公共前缀数组搜索给定字符串 p 在 t 中出现的所有位置。要求算法在最坏情况下的时间复杂性为 $O(m+\log n)$。

分析与解答： 在一般情况下，设要搜索的后缀数组区间是 sa$[L,R]$。开始时 $L=0$，$R=n-1$。在搜索过程中始终有

$$S_{\text{sa}[L]} \leqslant p \leqslant S_{\text{sa}[R]} \tag{9.13}$$

仍然用习题 9-15 中的二分搜索算法。关键运算是比较 p 和 $S_{\text{sa}[M]}$ 的大小，即计算 $\delta(p,S_{sa[M]})$ 的值，其中 $M=L+R$ 是搜索区间的中点。

$$\delta(p,S_{\text{sa}[M]})=\begin{cases}-1 & p>S_{\text{sa}[M]}\\ 0 & p=S_{\text{sa}[M]}\\ 1 & p<S_{\text{sa}[M]}\end{cases} \tag{9.14}$$

在搜索过程中，设

$$\begin{cases}l=\text{lcp}(p,S_{\text{sa}[L]})\\ r=\text{lcp}(p,S_{\text{sa}[R]})\\ \text{mid}=\text{lcp}(p,S_{\text{sa}[M]})\\ \alpha=\text{lcp}(S_{\text{sa}[L]},S_{\text{sa}[M]})\\ \beta=\text{lcp}(S_{\text{sa}[M]},S_{\text{sa}[R]})\end{cases} \tag{9.15}$$

其中，lcp(p, q)是字符串 p 和 q 的最长公共前缀的长度。

由于后缀数组 sa 是按照 t 的所有后缀的字典序排好序的，因此 $S_{\text{sa}[M]}$ 和 p 的前 $\min(l,r)$ 个字符相同。

当 $l=r$ 时，只要从 $S_{\text{sa}[M]}$ 和 p 的第 $l+1$ 个字符开始比较。当 $l>r$ 时，有如下 3 种情形。

（1）$l<\alpha$

此时有 p$[l]>$ t[sa$[L]+l]=$ t[sa$[M]+l]$，因此

$$\begin{cases}p>S_{\text{sa}[M]}\\ \text{mid}=l\end{cases}$$

搜索区间改变为 sa[M,R]，不需比较字符。

（2）$l>\alpha$

此时有 p[α] = t[sa[L]+α] < t[sa[M]+α]，因此

$$\begin{cases} p < S_{\text{sa}[M]} \\ \text{mid} = \alpha \end{cases}$$

搜索区间改变为 sa[L,M]，不需比较字符。

（3）$l=\alpha$

此时 $S_{\text{sa}[M]}$ 和 p 的前 l 个字符相同。只要从 $S_{\text{sa}[M]}$ 和 p 的第 $l+1$ 个字符开始比较。

当 $r>l$ 时，同样有如下 3 种情形。

（1）$r<\beta$

此时有 p[r] < t[sa[R]+r] = t[sa[M]+r]，因此

$$\begin{cases} p < S_{\text{sa}[M]} \\ \text{mid} = r \end{cases}$$

搜索区间改变为 sa[L,M]，不需比较字符。

（2）$r>\beta$

此时有 p[β] = t[sa[R]+β] > t[sa[M]+β]，因此

$$\begin{cases} p > S_{\text{sa}[M]} \\ \text{mid} = \beta \end{cases}$$

搜索区间改变为 sa[M,R]，不需比较字符。

（3）$r=\beta$

此时 $S_{\text{sa}[M]}$ 和 p 的前 r 个字符相同。只要从 $S_{\text{sa}[M]}$ 和 p 的第 $r+1$ 个字符开始比较。

综合以上分析可知 $\delta(p,S_{\text{sa}[M]})$，可以根据参数 l,r,α,β,M 来计算。

设

$$\eta = \delta(\text{p}[\max(l,r),m-1],\text{t}[\text{sa}[M]+\max(l,r),n-1]) \tag{9.16}$$

则

$$\delta(p,S_{\text{sa}[M]}) = \begin{cases} -1 & r<l<\alpha \ \vee \ r>\max(l,\beta) \\ 1 & l<r<\beta \ \vee \ l>\max(r,\alpha) \\ \eta & l=r \ \vee \ l<r=\beta \ \vee \ r<l=\alpha \end{cases} \tag{9.17}$$

将习题 9-16 中找下界和上界的算法 lower 和 upper 的比较函数 cmp()换成式（9.17）所表示的比较函数如下。

```
1  int cmp(string s, string p, int lft, int rht, int mid, int &l, int &r) {    // 改进的比较函数
2     int  al = lce(lft, mid), be = lce(mid, rht);
3     if(r < l && l < al || r > max(l, be))
4        return -1;
5     else if(l < r && r < be || l > max(r, al))
6        return 1;
7     else {
8        int  k = max(l, r);
9        int  ret = cp(s, p, k, sa[mid]);
10       if(ret > 0)
```

```
11          r = k;
12        else
13          l = k;
14        return ret;
15    }
16 }
```

其中，变量 lft、rht、mid 分别对应搜索区间的左右端点和中点。变量 l、r、al、be 分别对应式（9.15）中的 l、r、α、β。

算法的第 2 行用函数 lce()来计算 α、β。第 9 行按照公式（9.16）用 cp()函数来计算 η。

```
1  int lce(int l, int r) {                          // 计算α和β的值
2    if(l == r)
3      return(n-sa[l]);
4    if(r > n-1)
5      return 0;
6    return rq.query(l, r-1);
7  }
```

lce()函数先要用数组 lcp 进行预处理，建立对数组 lcp 做区域最小查询的类 rq。rq.query(l, r)函数可以在 $O(1)$ 时间内计算 lcp[l, r]中的最小值，而这个最小值就是 $S_{sa[l]}$ 和 $S_{sa[r]}$ 的最长公共前缀。

```
1  int cp(string text, string p, int &k, int j) {          // 计算η值
2    while(k < m && k+j < n && text[k+j] == p[k])
3      k++;
4    if(k == m)
5      return (n > k+j);
6    else if(k+j == n)
7      return -1;
8    else
9      return text[k+j]-p[k];
10 }
```

cp()函数在计算 η 时，从第 k 个字符开始比较字符串 p 和 $S_{sa[j]}$，结束后返回 k 的值。

要用改进的比较函数 cmp()来计算，还需要对算法 lower 和 upper 做一些改变。

```
1  int lower(string text,int *sa,string p) {              // 后缀数组最小位置搜索
2    int  lft=0, rht=n-1, len=n, l=0, r=0;
3    if(cp(text, p, 1, sa[0]) >0 || cp(text, p, r, sa[n-1]) < 0)
4      return 0;
5    while(len > 0) {
6      int  half = len>>1, mid = lft+half;
7      int  res = cmp(text, p, lft == 0 ? lft : lft-1, len == n ? rht : rht+1, mid, l, r);
8      if(res == 0)
9        return mid;
10     if(res < 0)
11       lft = mid+1, len = len-half-1;
12     else
13       len = half, rht = lft+len-1;
14   }
```

```
15    return lft;
16 }
```

第 3 行计算 $S_{sa[0]}$ 和 $S_{sa[n-1]}$ 与 p 的最长前缀长度，并判断式（9.13）的条件是否满足。这个条件是整个算法中必须满足的不变性条件。只有满足这个条件，才能保证公式（9.17）的正确性。

第 7 行没有直接用 lft 和 rht 的值来计算 cmp 的值。这是因为算法在改变搜索区间时，并不是用 mid 来替换 lft 或 rht，而是用 mid+1 来替换 lft，或用 mid−1 来替换 rht。这样可能破坏了不变性条件式（9.13）。因而在计算 cmp 的值时，lft 需要向左退一步，rht 需要向右退一步。当 $t[n-m,n-1]=p$ 时，出现 cmp 的值为 0。继续搜索就可能破坏了不变性条件式(9.13)。不过此时已经找到下界，可以终止搜索，所以在第 7 行直接返回其值。

算法其他部分不变。

```
1  int upper(string text, int *sa, string p, int fst) {        // 后缀数组最大位置搜索
2     int  lft = fst, rht = n-1, len = n-lft-1, l = 0, r = 0;
3     while(len > 0) {
4        int  half = len>>1, mid = lft+half;
5        int  lft1 = lft == fst ? lft : lft-1, rht1 = len == n ? rht : rht+1;
6        int  lm = lce(lft1, mid);
7        int  res = cmp(text, p, lft1, rht1, mid, l, r) && (lm < m);
8        if(res)
9           len = half, rht = lft+len-1;
10       else
11          lft = mid+1, len = len-half-1;
12    }
13    return lft;
14 }
```

算法 upper 的主要改变在第 6～7 行。由于要找的是后缀数组 sa 的上界，即返回值 lft 使得 sa[lft]大于 p 的最小元素，因此 $S_{sa[lft]}$ 与 p 的最长公共前缀的长度小于 m。此条件在 cmp() 返回的对于 0 的值中还不能体现，但是可以通过第 6 行中计算的值来判断 $S_{sa[mid]}$ 与 p 的最长公共前缀的长度是否小于 m，因此在第 7 行中还要加此条件。

改进后的算法在 lower()函数的第 3～4 行最多做了 $2m$ 次比较。将改进的比较函数 cmp() 所做的比较分为两类，即相等比较和不相等比较。算法的每次迭代最多有 1 次不相等比较，而迭代次数不超过 $\log n$ 次。所以，不相等比较次数不超过 $\log n$。对于相等比较，每次比较都从 p 的第 $k=\max\{l,r\}+1$ 个字符开始比较，而且每次相等比较都使下一轮迭代的 $\max\{l,r\}$ 值增加。由此可见，p 的每个字符最多只参加 1 次相等比较。也就是说，相等比较次数不超过 m。由此可知，改进的比较函数 cmp()所做的比较次数不超过 $m+\log n$。算法 lower 和 upper 循环体内的其他运算均需 $O(1)$ 时间。因此，算法需要的总时间是 $O(m+\log n)$。

算法实现题 9

9-1　安全基因序列问题。

问题描述：基因序列是用字符串表示的携带基因信息的 DNA 分子的一级结构。基因序

列的字符集是Σ={A,C,G,T}。其中字符分别代表组成 DNA 的 4 种核苷酸：腺嘌呤、胞嘧啶、鸟嘌呤、胸腺嘧啶。许多疾病往往是由基因突变引起的。这种基因突变是从一个正常的基因序列通过几代人的遗传而产生的。对于基因片段的分析有助于了解基因突变导致的遗传疾病。例如，如果一个基因序列中含有基因片段 ATG，则可能含有某种遗传疾病。生物科学家们已经发现许多这类基因片段。对于已知的不安全的基因片段集合 P，如果一个基因序列中含有 P 中基因片段，则称该基因序列为不安全的基因序列，否则称该基因序列为安全的基因序列。

算法设计：对于给定的不安全的基因片段集合 P 和一个正整数 n，计算长度为 n 的安全的基因序列个数。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行有两个正整数 n（$1 \leqslant n \leqslant 2\times10^9$）和 m（$0 \leqslant m \leqslant 10$）。$n$ 是基因序列长度，m 是不安全的基因片段个数。接下来的 m 行中，每行是一个长度不超过 10 的不安全的基因片段。每个文件可能有多个测试数据。

结果输出：将计算出的长度为 n 的安全的基因序列个数 mod 100000，输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
3 4	36
AT	
AC	
AG	
AA	

分析与解答：先建立不安全的基因片段集合 P 的 AC 自动机 T。如果一个基因序列 t 含有 P 中不安全的基因片段，则用 AC 自动机 T 做关于基因序列 t 的多子串搜索就可以找出它所含的所有不安全的基因片段。反之，如果搜索不到不安全的基因片段，则基因序列 t 是安全的基因序列。

基于这个基本思想，在 T 的基础上构造一个有向无环图 T'如下。在 T 中删去所有 output 非空，即 cnt>0 的结点，得到 T'。由此可知，从根结点到 T'中任一结点的路径组成的字符串都是安全的基因序列。设 T'有 m 个结点，且 T'的邻接矩阵为 $a^2[0..3][0..m-1]$。邻接矩阵为 a 的第 1 列中 4 个元素 $a[0..3][0]$分别表示从根结点出发经过边 A、C、G、T 的长度为 1 的字符串个数。a^2 的第 1 列中 4 个元素 $a^2[0..3][0]$分别表示从根结点出发经过边 A、C、G、T 的长度为 2 的字符串个数。以此类推，a^n 的第 1 列中 4 个元素 $a^n[0..3][0]$分别表示从根结点出发经过边 A、C、G、T 的长度为 n 的字符串个数。

由此可知，$\sum_{i=0}^{3} a^n[i][0]$ 就是长度为 n 的安全的基因序列个数。按此思路，解题需要以下 3 个步骤。

（1）建立不安全的基因片段集合 P 的 AC 自动机 T。

（2）根据 AC 自动机 T 建立 T'的邻接矩阵 a。

（3）计算 a^n，并输出 $\sum_{i=0}^{3} a^n[i][0]$。

由于本题只要输出安全的基因序列个数，在不安全的基因片段集合 P 的 AC 自动机 T 中

可以不必存储输出函数 output，因而可以用简化版的 AC 自动机 T 如下。

```
typedef struct node {                    // AC 自动机结点
   int  cnt;
   int  state;
   node  *fail;
   node  *go[dsize];
} tnode;
```

其中，cnt 存储输出函数 output()中的字符串个数。当 cnt>0 时有输出，否则无输出。

在 AC 自动机插入字符串的算法实现如下。

```
void insert(const string& word) {            // AC 自动机插入字符串
   tnode*  cur = root;
   for(int i=0; i < word.length(); ++i) {
      if(!cur->go[idx[word[i]]])
         cur->go[idx[word[i]]] = newnode();
      cur = cur->go[idx[word[i]]];
   }
   cur->cnt = 1;
}
```

其中，字符索引 idx 由 index()函数建立。

```
void index() {                               // 建立字符索引
   idx['A'] = 0;
   idx['C'] = 1;
   idx['G'] = 2;
   idx['T'] = 3;
}
```

计算失败函数的算法可具体描述如下。

```
void build_failure() {                   // 计算失败函数
   queue<tnode*> q;
   root->fail = NULL;
   q.push(root);
   while(!q.empty()) {
      tnode*  cur = q.front();
      q.pop();
      for(int i=0; i < dsize; ++i) {
         if(cur->go[i]) {
            tnode*  p = cur->fail;
            while(p && !p->go[i])
               p = p->fail;
            if(p) {
               cur->go[i]->fail = p->go[i];
               cur->go[i]->cnt = addout(cur->go[i], p->go[i]);
            }
            q.push(cur->go[i]);
         }
         else
```

```
            cur->go[i] = cur == root ? root : cur->fail->go[i];
        }
    }
}
```

根据 AC 自动机 T 建立 T'的邻接矩阵 a 如下。

```
void buildadj() {                                          // 建立邻接矩阵
    memset(adj, 0, sizeof(adj));
    for(int i=0; i < size; i++) {
        for(int j=0; j < dsize; j++) {
            tnode  *tmp = nmap[i].go[j];
            if(!nmap[i].cnt && !tmp->cnt)
                adj[i][tmp->state]++;
        }
    }
}
```

用二分法计算矩阵幂的算法如下。

```
void pow() {                                               // 计算矩阵幂
    for(int i=0; i < size; i++)
        for(int j=0; j < size; j++)
            pw[i][j] = i == j ? 1 : 0;
    binpow(pw, adj, size, n);
}
void binpow(int t[][nsize], int a[][nsize], int sz, int n) {      // 用二分法计算矩阵幂
    while(n > 0) {
        if(1 & n)
            mmult(t, a, sz);
        mmult(a, a, sz);
        n >>= 1;
    }
}
```

其中，mmult()函数用于计算两个矩阵的乘积。

综合以上步骤，计算安全的基因序列个数 $\sum_{i=0}^{3} a^n[i][0]$ 的算法描述如下。

```
void comp() {                                              // 计算安全的基因序列个数
    string  kwd;
    vector<string>  keywords;
    while(cin>>n>>m) {
        for(int i=0; i < m; i++) {
            cin>>kwd;
            keywords.push_back(kwd);
        }
        cons(keywords);
        buildadj();
        pow();
        out();
    }
```

```
14 }
```

其中，cons()函数构造不安全的基因片段集合 P 的 AC 自动机 T。

```
1  void cons(vector<string>& keywords) {          // 构造AC自动机
2     index();
3     memset(nmap, 0, sizeof(nmap));
4     root = NULL;
5     size = 0;
6     root = newnode();
7     for(int i=0; i < keywords.size(); i++)
8        insert(keywords[i]);
9     build_failure();
10 }
```

out 输出计算结果。

```
1  void out() {                                   // 输出计算结果
2     int  ans = 0;
3     for(int i=0; i < size; i++)
4        ans = (pw[0][i]+ans) % mod;
5     cout<<ans<<endl;
6  }
```

9-2　最长重复子串问题。

问题描述：最长重复子串问题在分子生物学和模式识别中有广泛应用，可以具体表述如下。给定 1 个长度为 n 的 DNA 序列 X，最长重复子串问题就是要找出在 X 中出现 2 次以上且长度最长的子串。例如，给定的 DNA 序列为 X=AGCATGCATGCAT，则子串 GCATGCAT 是 X 的一个最长重复子串，它在 X 的位置 1 和 5 处出现（第 1 个字符的位置为 0）。

算法设计：设计一个算法，找出给定字符串 X 的最长重复子串。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中给出字符串 X。

结果输出：将计算出的字符串 X 的最长重复子串输出到文件 output.txt 中。

文件的第 1 行是最长重复子串的长度。文件的第 2 行是最长重复子串。

输入文件示例	输出文件示例
input.txt	output.txt
AGCATGCATGCAT	8
	GCATGCAT

分析与解答：此题是后缀数组的一个简单应用。先计算出 X 的后缀数组 sa 和最长公共前缀数组 lcp。

最长重复子串的长度实际上就是 X 的两个后缀的最长公共前缀的最大值，所以最长公共前缀数组 lcp 中的最大值就是 X 的最长重复子串的长度。如果在 lcp[i]中取得最大值，则最长重复子串是 X 从位置 sa[i]开始的长度为 lcp[i]的子串。

```
1  pair<int,int> lp(int n) {                      // 最长重复子串
2     pair<int, int>r;
3     int  len = 0, j = -1;
4     for(int i=0; i < n-1; i++)
5        if(lcp[i] > len)
6           len = lcp[i], j = i;
```

```
7       r.first = len, r.second = j;
8       return r;
9   }
```

计算 X 的后缀数组 sa 和最长公共前缀数组 lcp 需要 $O(n)$ 时间。计算最长公共前缀数组 lcp 中最大值显然只要 $O(n)$ 时间。

由此可知，上述最长重复子串算法需要 $O(n)$ 时间。

9-3 最长回文子串问题。

问题描述：如果一个字符串正读和反读相同，则称此字符串为回文。如果字符串 X 的一个子串 Y 是回文，则称子串 Y 是字符串 X 的一个回文子串。最长回文子串问题可以具体表述如下。

给定 1 个长度为 n 的字符串 X，最长回文子串问题就是要找出 X 中长度最长的回文子串。例如，如果给定的字符串 X=bbacababa，则子串 bacab 是 X 的一个最长的回文子串，它的长度是 5。

算法设计：设计一个算法，找出给定字符串 X 的最长回文子串。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中给出字符串 X。

结果输出：将计算出的字符串 X 的最长回文子串输出到文件 output.txt 中。文件的第 1 行是最长回文子串的长度。文件的第 2 行是最长回文子串。

输入文件示例	输出文件示例
input.txt	output.txt
bbacababa	5
	bacab

分析与解答：当回文串的长度是奇数 $2k+1$ 时，第 $k+1$ 个字符就称为该回文串的中心。当回文串的长度是偶数 $2k$ 时，它的中心定义为第 k 和第 $k+1$ 个字符之间的位置。这两种情形都定义回文串的半径为 k。注意，每个不同的最长回文子串的中心也不同。因此，最长回文子串的中心最多出现在 $2n-1$ 个不同位置。设给定字符串 s 的逆串为 $s^{\sim}$，构造一个新字符串 t=s\$$s^{\sim}$，其中 \$ 是不在 s 中的特殊的字符。计算新字符串 t 的后缀数组 sa 和最长公共前缀数组 lcp。这样就把问题变成计算新字符串 t 的最长公共扩展问题。如果最长回文子串的长度是奇数，且其中心位置是 i，则其长度是 $s[i+1,n-1]$ 与 $s^{\sim}[n-i+2,n-1]$ 的最长公共前缀的长度。类似地，如果最长回文子串的长度是偶数且其中心位置是 i，则其长度是 $s[i+1,n-1]$ 与 $s^{\sim}[n-i+1,n-1]$ 的最长公共前缀的长度。用一次线性扫描就可以找到最长回文子串。

先将输入字符串 s 变换为新字符串 t。

```
1   void change(string &s) {                          // 字符串变换
2       s.resize(2*n+2);
3       s[n] = 1, s[2*n+1] = 0;
4       for(int i=n+1; i <= 2*n; i++)
5           s[i] = s[2*n-i];
6   }
```

然后计算新字符串 t 的后缀数组 sa 和最长公共前缀数组 lcp，并建立一个类 rmq 来支持 t 的最长公共扩展查询。

```
1   int lce(rmq rq, int l, int r) {                   // 最长公共扩展查询
2       return rq.query(min(rank[l], rank[r]), max(rank[l], rank[r])-1);
```

```
3  }
```

借助 t 的最长公共扩展查询，对输入字符串做一次线性扫描，就可以找到它的最长回文子串。

```
1  int palin(string s, rmq rq, int n) {          // 最长回文子串
2    int  ans = 1, pos = 0;
3    for(int i=0; i < n; i++) {
4      int  cp = lce(rq, i, 2*n-i);
5      if(cp*2-1 > ans)
6        ans = cp*2-1, pos = i-cp+1;
7      cp = lce(rq, i, 2*n-i+1);
8      if(cp*2 > ans)
9        ans = cp*2, pos = i-cp;
10   }
11   return ans;
12 }
```

计算新字符串 t 的后缀数组和最长公共前缀数组并建立一个类 rmq 需要的计算时间是 $O(n)$，t 的最长公共扩展查询 lce 的响应时间是 $O(1)$，所以一次线性扫描需要的计算时间是 $O(n)$。由此可见，整个算法需要的计算时间是 $O(n)$。

9-4　相似基因序列性问题。

问题描述：最长公共子序列问题是生物信息学中序列比对问题的一个特例。这类问题在分子生物学和模式识别中有广泛应用。其中最主要的应用是测量基因序列的相似性。在演化分子生物学的研究中发现，某个重要的 DNA 序列片段常出现在不同的物种中。在测量基因序列的相似性时，如果需要特别关注一个具体的 DNA 序列片段，就要考察带有子串排斥约束的最长公共子序列问题。这个问题可以具体表述如下。

给定两个长度分别为 n 和 m 的序列 x[0..n−1]和 y[0..m−1]，以及一个长度为 p 的约束字符串 s[0..p−1]。带有子串排斥约束的最长公共子序列问题就是要找出 x 和 y 的不包含 s 为其子串的最长公共子序列。例如，如果给定的序列 x 和 y 分别为 AATGCCTAGGC 和 CGATCTGGAC，字符串 s=TG 时，子序列 ATCTGGC 是 x 和 y 的一个无约束的最长公共子序列，而不包含 s 为其子串的最长公共子序列是 ATCGGC。

算法设计：设计一个算法，找出给定序列 x 和 y 的不包含 s 为其子串的最长公共子序列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中给出正整数，分别表示给定序列 x 和 y 及约束字符串 s 的长度。接下来的 3 行分别给出序列 x、y 和约束字符串 s。

结果输出：将计算出的 x 和 y 的不包含 s 为其子串的最长公共子序列的长度输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
11 10 2	6
AATGCCTAGGC	
CGATCTGGAC	
TG	

分析与解答：按照主教材中式（9.29）可以设计求序列 x 和 y 的不包含 s 为其子串的最长公共子序列的长度的算法如下。首先为了有效计算函数 σ，对字符集中每个字符 ch $\in \Sigma$ 和

$1 \leqslant k \leqslant p$，预先计算$\sigma(s[1..k]\text{ch})$，并保存在表$\pi$中。

```
int cp(int i, int j) {                                  // 预处理
  if(i && !pi[i][j])
    pi[i][j] = cp(kmp[i],j);
  return pi[i][j];
}
```

用预先计算好的表π，就可以在$O(1)$时间内计算$\sigma(s[1..k]\text{ch})$的值。

```
int r(char ch,int k) {                                  // 计算σ值
  return pi[k][di(ch)];
}
int dyna() {                                            // 动态规划算法
  for(int i=n; i > 0; i--) {
    for(int j=m; j > 0; j--) {
      for(int k=0; k < p; k++) {
        if(x[i] == y[j]) {
          f[i][j][k] = f[i+1][j+1][k];
          int  q = r(x[i],k);
          if(q < p && f[i][j][k] < 1+f[i+1][j+1][q])
            f[i][j][k] = 1+f[i+1][j+1][q];
        }
        else
          f[i][j][k] = max(f[i+1][j][k], f[i][j+1][k]);
      }
    }
  }
  return f[1][1][0];
}
```

9-5　计算机病毒问题。

问题描述：计算机病毒是黑客在计算机程序中插入的破坏计算机功能或者数据的一组计算机指令或者程序代码。计算机病毒不仅能影响计算机使用，还能自我复制。就像生物病毒一样，它具有自我繁殖、互相传染及激活再生等生物病毒特征。计算机病毒的独特的复制能力，使它们能够快速蔓延，又常常难以根除。它们能把自身附着在各种类型的文件上，当文件被复制或从一个用户传送到另一个用户时，它们随文件一起蔓延。杀除计算机病毒的一个有效方法是找出特定计算机病毒的代码特征。对于给定的带有某种病毒的程序代码段集合，通过寻找程序代码段集合中所包含的公共特征，可以快速确定计算机病毒的代码特征。

算法设计：给定带有某种病毒的程序代码段集合，寻找程序代码段集合中每个代码段都包含的最长字符串。

数据输入：由文件 input.txt 提供输入数据。文件第一行有一个正整数 n（$1 \leqslant n \leqslant 100$），表示程序代码段集合中代码段数。接下来的 n 行中，每行是一个程序代码段。每个程序代码段已经转换成由英文大小写字母组成的长度不超过 1000 的字符串。

结果输出：将找到的程序代码段集合中最长公共字符串输出到文件 output.txt 中。

文件的第 1 行输出最长公共字符串的长度。文件的第 2 行输出最长公共字符串。

输入文件示例　　输出文件示例

input.txt	output.txt
3	6
abcdefgi	bcdefg
cbcdefghc	
cdefgibcdefghe	

分析与解答：此题要求给定字符串集合的最长公共子串。借助于后缀数组 sa 和最长公共前缀数组 lcp 设计的算法如下。

```
1  int lcs() {                                    // 最长公共子串
2     readin();
3     build();
4     int  max = search();
5     return max;
6  }
```

第 2 行读入 n 个字符串，并将它们用不在输入串中出现的 n 个不同字符连接成一个新字符串 r。

```
1  void readin() {                          // 读入 n 个字符串
2     string  s;
3     r.resize(maxn);
4     cin>>n;
5     for(int i=1; i <= n; i++){
6        cin>>s;
7        int  k = s.size();
8        if(k < up)
9           up = k;
10       for(int j=0; j < k; j++) {
11          r[j+len] = s[j];
12          idx[j+len] = i;
13       }
14       r[len+k] = i+300, idx[len+k] = 0, len += k+1;
15    }
16    len--, r[len] = 0, r.resize(len+1);
17 }
```

前面程序第 3 行的 build()函数建立新字符串 r 的后缀数组 sa 和最长公共前缀数组 lcp。

```
1  void build() {                                 // 建立后缀数组
2     suffix  suf(r);
3     sa = suf.sa;
4     lcp = suf.lcp;
5     lcp[len] = -1;
6  }
```

前面程序第 4 行的 search()函数利用后缀数组 sa 和最长公共前缀数组 lcp 做二分搜索，来寻找最长公共子串。

```
1  int search() {                           // 搜索公共子串
2     int  first = 1, last = up;
3     while(first <= last) {
4        int  mid = (first+last)>>1;
```

```
5       if(check(mid))
6         first = mid+1;
7       else
8         last = mid-1;
9     }
10    return last;
11  }
```

其中，check(mid)函数用于判断 r 中是否存在长度为 mid 的公共子串。

```
1   bool check(int mid) {                          // 长度为mid的公共子串
2     int  j, t, s;
3     for(int i=0; i < len; i = j+1) {
4       for(; lcp[i] < mid && i <= len; i++)  ;
5       for(j=i; lcp[j] >= mid; j++)  ;
6       if(j-i+1 < n)
7         continue;
8       cnt++;
9       s = 0;
10      for(int k=i; k <= j; k++)
11        if((t=idx[sa[k]]) != 0 && lab[t] != cnt)
12          lab[t] = cnt, s++;
13      if(s == n) {
14        ans = sa[i];
15        return true;
16      }
17    }
18    return false;
19  }
```

如果输入字符串集合中所有字符串长度总和为 l，其中最短字符串长度为 m，则上述算法需要的计算时间为 $O(l\log m)$。因为建立后缀数组 sa 和最长公共前缀数组 lcp 需要 $O(l)$ 时间。算法 check 需要 $O(l)$ 时间。二分搜索算法 search 最多调用 check 算法 $\log m$ 次。

9-6 带有子串包含约束的最长公共子序列问题。

问题描述：给定 2 个长度分别为 n 和 m 的序列 x[0..n−1]和 y[0..m−1]，以及一个长度为 p 的约束字符串 s[0..p−1]。带有子串包含约束的最长公共子序列问题就是要找出 x 和 y 的包含 s 为其子串的最长公共子序列。例如，如果给定的序列如果给定的序列 x 和 y 分别为 AATGCCTAGGC 和 CGATCTGGAC，字符串 s=GTA 时，子序列 ATCTGGC 是 x 和 y 的一个无约束的最长公共子序列，而包含 s 为其子串的最长公共子序列是 GTAC。

算法设计：设计一个算法，找出给定序列 x 和 y 的包含 s 为其子串的最长公共子序列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中给出正整数，分别表示给定序列 x、y 和约束字符串 s 的长度。接下来的 3 行分别给出序列 x、y 和约束字符串 s。

结果输出：将计算出的 x 和 y 的包含 s 为其子串的最长公共子序列的长度输出到文件 output.txt 中。

输入文件示例	输出文件示例
input.txt	output.txt
11 10 3	4

AATGCCTAGGC
CGATCTGGAC
GTA

分析与解答：按照主教材中的式（9.24）和式（9.25）可以设计求 x 和 y 的包含 s 为其子串的最长公共子序列的长度的算法如下。

```
int comp() {                    // 带有子串包含约束的最长公共子序列
  suffix();
  lcsr();
  int  i1, j1, tmp = 0;
  for(int i=1; i <= n; i++) {
    for(int j=1; j <= m; j++) {
      int  sum = f[i][j][p];
      if(i < n && j < m)
        sum += g[i+1][j+1];
      if(tmp < sum) {
        tmp = sum;
        i1 = i;
        j1 = j;
      }
    }
  }
  return tmp;
}
```

其中，suffix()函数用于计算 $f(i,j,k)$，lcsr()函数用于计算 $g(i,j)$。

```
int suffix() { // 计算f(i,j,k)
  for(int i=0; i <= n; i++)
    f[i][0][0] = 0;
  for(int j=0; j <= m; j++)
    f[0][j][0] = 0;
  for(int k=1; k <= p; k++) {
    for(int i=0; i <= n; i++)
      f[i][0][k] = INT_MIN;
    for(int j=0; j <= m; j++)
      f[0][j][k] = INT_MIN;
  }
  for(int i=1; i <= n; i++) {
    for(int j=1; j <= m; j++) {
      for(int k=0; k <= p; k++) {
        int  &z = f[i][j][k];
        if(x[i-1] != y[j-1])
          z = max(f[i][j-1][k], f[i-1][j][k]);
        else {
          if(k == 0 && x[i-1] == y[j-1])
            z = f[i-1][j-1][k]+1;
          if(k>0 && x[i-1] != s[k-1])
            z = f[i-1][j-1][k];
          else if(k>0 && x[i-1] == s[k-1])
```

```
                z = 1+f[i-1][j-1][k-1];
            }
        }
    }
    return max(-1, f[n][m][p]);
}
int lcsr() {                                    // 计算g(i,j,k)
    int  i, j;
    g[n+1][m+1] = 0;
    for(i = 1; i <= n; i++)
        g[i][n+1] = 0;
    for(i = 1; i <= m; i++)
        g[m+1][i] = 0;
    for(int i=n; i; i--) {
        for(int j=m; j; j--) {
            if(x[i-1] == y[j-1])
                g[i][j] = g[i+1][j+1]+1;
            else if(g[i+1][j] > g[i][j+1])
                g[i][j] = g[i+1][j];
            else
                g[i][j] = g[i][j+1];
        }
    }
    return g[1][1];
}
```

9-7　多子串排斥约束的最长公共子序列问题。

问题描述：给定 2 个长度分别为 n 和 m 的序列 x[0..n−1]和 y[0..m−1]，以及 d 个约束字符串 $s_1, s_2, \cdots, s_d$。多子串排斥约束的最长公共子序列问题就是要找出 x 和 y 的不含 $s_1, s_2, \cdots, s_d$ 为其子串的最长公共子序列。

算法设计：设计一个算法，找出给定序列 x 和 y 的不含 $s_1, s_2, \cdots, s_d$ 为其子串的最长公共子序列。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中给出正整数 d，表示约束字符串个数。接下来的 2 行分别给出序列 x 和 y。最后 d 行的每行给出一个约束字符串。

结果输出：将计算出的 x 和 y 的不含 $s_1, s_2, \cdots, s_d$ 为其子串的最长公共子序列输出到文件 output.txt 中。文件的第 1 行输出最长公共子序列。第 2 行输出最长公共子序列的长度。

输入文件示例	输出文件示例
input.txt	output.txt
4	CCC
AATGCCTAGGC	3
TG	
A	
G	
TC	

分析与解答：给定输入序列 x[0..n−1]、y[0..m−1]和 d 个约束字符串 $s_1, s_2, \cdots, s_d$，其总长

度是 r。问题求解目标是找到 x 和 y 的最长公共子序列，不含任何 $s_1, s_2, \cdots, s_d$ 为其子串。这个问题实际上是有子串排除约束的最长公共子序列在多子串情况下的推广。算法思想是类似的。在多子串的情形下，需要用到 AC 自动机。首先建立字符串约束集 $s_1, s_2, \cdots, s_d$ 的 AC 自动机 T。设 T 中非叶结点编号为 $0,1,\cdots,t-1$。根结点编号为 0。在 AC 自动机 T 中，从结点根 0 到任一状态结点 state 的路径上各边的标号字符连接组成的字符串，即结点 state 的标号为 $\alpha(\text{state})$。对任一字符串 q，它在 AC 自动机 T 中的最长后缀结点记为 $\sigma(q)$，即

$$|\alpha(\sigma(q))| = \max_{0 \leqslant i < t} \{|\alpha(i)| \,|\, \alpha(i) \sqsupset q\}$$

设 $Z(i,j,k)$ 是 x[0..i] 和 y[0..j] 的不含任何 $s_1, s_2, \cdots, s_d$ 中字符串为其子串的最长公共子序列组成的集合，且对任一 $z \in Z(i,j,k)$ 有 $\sigma(z) = k$（$0 \leqslant i \leqslant n-1$，$0 \leqslant j \leqslant m-1$，$0 \leqslant k < t$）。

$Z(i,j,k)$ 中任一最长公共子序列的长度记为 $f(i,j,k)$。

如果能有效计算出 $f(i,j,k)$，则 x 和 y 的不含任何 $s_1, s_2, \cdots, s_d$ 为其子串的最长公共子序列的长度就是 $\max\limits_{0 \leqslant k < t} \{f(n,m,k)\}$。

用动态规划算法计算 $f(i,j,k)$ 的递归式如下：

$$f(i,j,k) = \begin{cases} \max\{f(i-1,j,k), f(i,j-1,k)\} & x_i \neq y_j \\ \max\{f(i-1,j-1,k), 1 + \beta(i,j,k)\} & x_i = y_j \end{cases}$$

其中

$$\beta(i,j,k) = \max_{0 \leqslant q < t} \{f(i-1,j-1,q) \,|\, \sigma(\alpha(q)x_i) = k\}$$

用字符串约束集 $s_1, s_2, \cdots, s_d$ 的 AC 自动机 T，可以在 $O(1)$ 时间内计算 $\sigma(q)$，因此上述动态规划算法所需计算时间为 $O(nmr)$。

参考文献

[1] 王晓东，傅清祥，叶东毅. 算法与数据结构学习指导与习题解析. 北京：电子工业出版社，2000.

[2] 王晓东. 数据结构（C 语言描述）. 北京：电子工业出版社，2011.

[3] 王晓东. 算法设计与实验题解. 北京：电子工业出版社，2006.

[4] 王晓东. 计算机算法设计与分析（第 4 版）. 北京：电子工业出版社，2012.

[5] 王晓东. 计算机算法设计与分析（第 5 版）. 北京：电子工业出版社，2018.

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为，歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，本社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail：dbqq@phei.com.cn
通信地址：北京市海淀区万寿路 173 信箱
　　电子工业出版社总编办公室
邮　　编：100036

[General Information]
书名=14493193
SS号=14493193

欢迎登录 免费 获取本书教学资源
http://www.hxedu.com.cn

计算机算法设计与分析习题解答（第5版）

本书是与“十二五”普通高等教育本科国家级规划教材《计算机算法设计与分析（第5版）》配套的辅助教材和国家精品课程教材，分别对主教材中的算法分析题和算法实现题给出了解答或解题思路提示。为了提高学生灵活运用算法设计策略解决实际问题的能力，本书还将主教材中的许多习题改造成算法实现题，要求学生设计出求解算法并上机实现。本书教学资料包含各章算法实现题、测试数据和答案，可在华信教育资源网免费注册下载。

本书内容丰富，理论联系实际，可作为高等学校计算机科学与技术、软件工程、信息安全、信息与计算科学等专业本科生和研究生学习计算机算法设计的辅助教材，也是工程技术人员和自学者的参考书。

提升学生“知识—能力—素质”	体现“基础—技术—应用”内容
把握教学“难度—深度—强度”	提供“教材—教辅—课件”支持

相关图书：《计算机算法设计与分析（第5版）》 ISBN 978-7-121-34439-8

策划编辑：章海涛
责任编辑：章海涛
封面设计：张 昱

ISBN 978-7-121-34438-1

定价：56.00 元