ACM-ICPC 培训资料汇编

（5）

字符串处理、搜索分册

（版本号 **1.0.0**）

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

序

2012 年 5 月，哈尔滨理工大学承办了 ACM-ICPC 黑龙江省第七届大学生程序设计竞赛。做为本次竞赛的主要组织者，我还是很在意本校学生是否能在此次竞赛中取得较好成绩，毕竟这也是学校的脸面。因此，当 2011 年 10 月确定学校承办本届竞赛后，我就给齐达拉图同学很大压力，希望他能认真训练参赛学生，严格要求受训队员。当然，齐达拉图同学半年多的工作还是很有成效，不仅带着黄李龙、姜喜鹏、程宪庆、卢俊达等队员开发了我校的 OJ 主站和竞赛现场版 OJ，还集体带出了几个比较像样的新队员，使得今年省赛我校取得了很好的成绩（当然，也承蒙哈工大和哈工程关照，没有派出全部大牛来参赛）。

在 2011 年 9 月之前，我对 ACM-ICPC 关心甚少。但是，我注意到我校队员学习、训练没有统一的资料，也没有按照竞赛所需知识体系全面系统培训新队员。2011-2012 年度的学生教练们做了一个较详细的培训计划，每周都会给 2011 级新队员上课，也会对老队员进行训练，辛辛苦苦忙活了一年——但是这些知识是根据他们个人所掌握情况来给新生讲解的，新生也是杂七杂八看些资料和做题。在培训的规范性上欠缺很多，当然这个责任不在学生教练。2011 年 9 月，我曾给老队员提出编写培训资料这个任务，一是老队员人数少，有的还要去百度等企业实习；二是老队员要开发、改造 OJ；三是培训新队员也很耗费精力，因此这项工作虽很重要，但却不是那时最迫切的事情，只好被搁置下来。

2012 年 8 月底，2012 级新生满怀梦想和憧憬来到学校，部分同学也被 ACM-ICPC 深深吸引。面对这个新群体的培训，如何提高效率和质量这个老问题又浮现出来。市面现在已经有了各种各样的 ACM-ICPC 培训教材，主要算法和解题思路都有了广泛深入的分析和讨论。同时，互联网博客、BBS 等中也隐藏着诸多大牛对某些算法的精彩论述和参赛感悟。我想，做一个资料汇编，采撷各家言论之精要，对新生学习应该会有较大帮助，至少一可以减少他们上网盲目搜索的时间，二可以给他们构造一个相对完整的知识体系。

感谢 ACM-ICPC 先辈们作出的杰出工作和贡献，使得我们这些后继者们可以站在巨人的肩膀上前行。

感谢校集训队各位队员的无私、真诚和抱负的崇高使命感、责任感，能够任劳任怨、以苦为乐的做好这件我校的开创性工作。

唐远新

2012 年 10 月

编写说明

本资料为哈尔滨理工大学 ACM-ICPC 集训队自编自用的内部资料，不作为商业销售目的，也不用于商业培训，因此请各参与学习的同学不要外传。

本分册大纲由黄李龙编写，内容由卢俊达、黄李龙等分别编写和校核。

本分册内容大部分采编自各 OJ、互联网和部分书籍。在此，对所有引用文献和试题的原作者表示诚挚的谢意！

由于时间仓促，本资料难免存在表述不当和错误之处，格式也不是很规范，请各位同学对发现的错误或不当之处向acm@hrbust.edu.cn邮箱反馈，以便尽快完善本文档。在此对各位同学的积极参与表示感谢！

哈尔滨理工大学在线评测系统（Hrbust-OJ）网址：http://acm.hrbust.edu.cn，欢迎各位同学积极参与AC。

国内部分知名 OJ：

杭州电子科技大学：http://acm.hdu.edu.cn

北京大学：http://poj.org

浙江大学：http://acm.zju.edu.cn

以下百度空间列出了比较全的国内外知名 OJ:
http://hi.baidu.com/leo_xxx/item/6719a5ffe25755713c198b50

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

目　录

第4章 字符串处理

4.1 BM 算法

参考文献：

《BM 算法详细图解》 Weisteven

扩展阅读：

南柯一喵：http://www.cnblogs.com/a180285/archive/2011/12/15/BM_algorithm.html

编写：卢俊达　　校核：黄李龙

4.1.1 基本原理

BM 算法主要思想描述如下

(1)模式字符串的匹配顺序是从右向左:

(a)首先将 P 和 T 对齐,即 p1 和 t1 对齐；

(b)然后匹配从模式字符串 P 的最右端字符开始，即判断

mp 和 mt 是否匹配：如果匹配成功,则向左移动判断。pm-1 和 tm-1 是否匹配，如此循环下去；如果匹配不成功，则进行字符串滑移。

(2)字符串滑移启发式策略：

(a)坏字符移动启发式策略

(b)好后缀移动启发式策略

两种策略的使用：如果同时满足两种策略使用条件时，选两者中较大的作为模式串向右滑移的距离。

BM 算法被认为是亚线性串匹配算法，它在最坏情况下找到模式所有出现的时间复杂度为 O(mn)，在最好情况下执行匹配找到模式所有出现的时间复杂度为 O(n/m)。

4.1.2 模板代码

```
#define T_SIZE 1000000
#define P_SIZE 10000
char T[T_SIZE+1],P[P_SIZE+1];

int last(char *p, char c)
{
    //找到 c 在 p 中最后匹配的位置,没有就返回－1
    int length = strlen(p), count   = 0;
    char *pp = p + length -1;
    while (pp >= p)
    {
        if (*pp == c)
            return length - count - 1;
        pp--;
        count++;
    }
   return -1;
}

int min(int a, int b)
{
   return (a <= b) ? a : b;
}
```

```
int BM_index(char *T, char *p)
//返回第一次开始匹配的位置
{
    int n = strlen(T);
    int m = strlen(p);
    int i = m-1, j = m-1;
    while (i <= n-1)
        if (T[i]==p[j])
            if (j==0)
                return i;
            else
                i--, j--;
        else
          {
            i = i + m - min(j, 1+last(p, T[i]) );
               //往后跳，取决于最后一次匹配的字符的位置
            j = m - 1;
        }
    return -1;
}

int sum(char* T,char* P,int s)
//输出母串 T 中包含 P 的数量
{
     int e=BM_index(T+s,P);
     return e==-1?0:1+sum(T,P,s+e+1);
}
```

4.1.3 经典题目

1．题目出处/来源

[HerbustOJ][1551][BM] 基础数据结构——字符串 2 病毒 II

2．题目描述

自从计算机病毒的概念被提出之后，病毒的种类可以说是层出不穷。现在，单纯的病毒是逃不过杀毒软件的。因此现在的病毒往往隐藏一些字符之中来达到蒙混过关的目的。已知连续的字符串"bkpstor"是一段病毒编码，请分析给出的一段字符串中是否包含病毒编码。

3．分析

直接套用模板即可。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;

#define T_SIZE 1000000
#define P_SIZE 10000
char T[T_SIZE+1],P[P_SIZE+1];

int last(char *p, char c)
{
    //找到 c 在 p 中最后匹配的位置,没有就返回－1
    int length = strlen(p), count    = 0;
    char *pp = p + length -1;
    while (pp >= p)
```

```
    {
        if (*pp == c)
            return length - count - 1;
        pp--;
        count++;
    }
    return -1;
}

int min(int a, int b)
{
    return (a <= b) ? a : b;
}

int BM_index(char *T, char *p)
//返回第一次开始匹配的位置
{
    int n = strlen(T);
    int m = strlen(p);
    int i = m-1, j = m-1;
    while (i <= n-1)
        if (T[i]==p[j])
            if (j==0)
                return i;
            else
                i--, j--;
        else
          {
            i = i + m - min(j, 1+last(p, T[i]) );
              //往后跳，取决于最后一次匹配的字符的位置
            j = m - 1;
        }
    return -1;
}

int sum(char* T,char* P,int s)
//输出母串 T 中包含 P 的数量
{
    int e=BM_index(T+s,P);
    return e==-1?0:1+sum(T,P,s+e+1);
}

int main()
{
    while(gets(T))
        printf("%s\n",sum(T,"bkpstor",0)?"Warning":"Safe");
}
```

4.2 前缀函数（Prefix function）KMP 的 next 函数

参考文献：

《算法导论》

编写：卢俊达　　校核：黄李龙

4.2.1 基本原理

前缀函数是 KMP 思想的精髓，其代码实现仅有短短 13 行（算法导论上的标准写法）。

前缀函数的功能是求出模式串 next[]数组。什么是模式串？什么是 next[]数组？

模式串的概念很简单。举个例子：“给出一个字符串 T，再给出 n 个字符串 S1、S2...Sn，问 S1、S2...Sn 中有哪些是 T 的子串？”在这个例子中，S1、S2...Sn 便是 n 个模式串，T 便是被匹配串。模式串是用来与被匹配串匹配的。

那么 next[]数组又是什么？这个 next[]的概念便是前缀函数的精髓，KMP 思想中精髓的精髓。抛去网络中繁杂的讲解，用一句话概括“若模式串 P 的前 i 个字符组成的子串为 S，那么‘S 的前 next[i]个字符’与‘S 的后 next[i]个字符’相同。”

举例：模式串 P 为 ababaaab。next[5]的值为 3。这说明 P 的前 5 个字符组成的字符串 ababa 的“前 3 个字符”与“后 3 个字符”相同，均为 aba。

重要的是理解 next[]数组的含义，在理解的基础上进行题目练习才有意义。

4.2.2 模板代码

```
//T 是被匹配的串。
//P 是模式串。
//字符串都是从下标1开始的。
void COMPUTE_PREFIX_FUNCTION(char P[])
{
    int m=strlen(P+1);
    next[1]=0;
    for(int k=0,q=2;q<=m;q++)
    {
        while(k>0&&P[k+1]!=P[q])
            k=next[k];
        if(P[k+1]==P[q])
            k++;
        next[q]=k;
    }
}
```

4.2.3 经典题目

4.2.3.1 题目 1

1．题目出处/来源

[POJ][2406][KMP] Power Strings

2．题目描述

给出一个字符串，问其是由多少个相同的子串组成的。

3．分析

考验对 next[]数组的理解。

next[i]含义是模式串 P 的前 i 个字符组成的字符串 Pi 与其子串（即 Pi 的子串）匹配的最大长度。例如模式串 P=”ababab”，P4 就是模式串 P 的前 4 个字符组成的字符串”abab”，P4 与其子串匹配的最大长度是 2（即”abab”与”ab”匹配），则 next[4]=2。

由 next[i]的理论可以推知，len(p)-next[len(p)]是模式串 P 的“循环节”长度。为什么

循环节长度上要加引号？因为模式串不一定恰好是“循环节”的整数倍，而是其整数倍的子串。在 POJ 的 board 里看到了一个非常好的样例，当模式串为”aabaabaa”时，循环节为 3。所以在输出答案的时候，要判断模式串 P 的长度能否整除“循环节”，若不能整除，则输出 1。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define P_SIZE 1000000
int next[P_SIZE+2];
char P[P_SIZE+2];
//P 是模式串。
void COMPUTE_PREFIX_FUNCTION(char P[])
//对 next[]进行赋值。
{
    int m=strlen(P+1);
    next[1]=0;
    for(int k=0,q=2;q<=m;q++)
    {
        while(k>0&&P[k+1]!=P[q])
            k=next[k];
        if(P[k+1]==P[q])
            k++;
        next[q]=k;
    }
}
int main()
{
    while(scanf("%s",P+1)&&strcmp(".",P+1))
    {
        COMPUTE_PREFIX_FUNCTION(P);
        int len=strlen(P+1),cycle=len-next[len];
        //len 是模式串 P 的长度。
        //cycle 是“循环节”。
        printf("%d\n",len%cycle?1:len/cycle);
        //若 len 不能整除 cycle 时，输出 1。
    }
}
```

4.2.3.2 题目 2

1．题目出处/来源

[POJ][2752][KMP] Seek the Name, Seek the Fame

2．题目描述

给出一个字符串，输出所有既是前缀又是后缀的子串的长度。

3．分析

本题考验对 next[]数组的理解。

根据题意，首先模式串 strlen(P)一定是满足条件的。根据 next[]数组的含义，P 的左 next[strlen(P)]个字符与 next[strlen(P)]个字符一定是相同的，P 的左 next[next[strlen(P)]]个字符与 next[next[strlen(P)]]个字符也一定是相同的，next[next[next[strlen(P)]]]...同理。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
```

```
using namespace std;
#define P_SIZE 1000000
int next[P_SIZE+2],res[P_SIZE];
char P[P_SIZE+2];
//P 是模式串。
void COMPUTE_PREFIX_FUNCTION(char P[])
//对 next[]进行赋值。
{
    int m=strlen(P+1);
    next[1]=0;
    for(int k=0,q=2;q<=m;q++)
    {
        while(k>0&&P[k+1]!=P[q])
            k=next[k];
        if(P[k+1]==P[q])
            k++;
        next[q]=k;
    }
}
int main()
{
    while(~scanf("%s",P+1))
    {
        COMPUTE_PREFIX_FUNCTION(P);
        int total=0,len=strlen(P+1);
        res[total++]=len;
        for(int i=len;next[i]!=0;i=next[i])
            res[total++]=next[i];
        for(int i=total-1;i>=0;i--)
            printf("%d%c",res[i],i?' ':'\n');
    }
}
```

4.3 KMP 算法

参考文献：

《算法导论》

编写：卢俊达　　校核：黄李龙

4.3.1 基本原理

KMP 算法主要由两部分组成，前缀函数部分和匹配部分。

前缀函数部分是 KMP 的精髓，在前一部分已经概要的叙述了其原理。在本部分主要讲解匹配部分。

匹配部分主要依托 next[]数组进行。其原理还是力求用一句话来概括“当被匹配串的第 i 个字符 T[i]与模式串的第 j 个字符不匹配时，将转而与模式串的第 next[j-1]+1 个字符匹配。”

这样提高了效率，避免从模式串开头重新匹配的情况出现，有效降低了时间复杂度。

4.3.2 模板代码

```
int next[P_SIZE+2];
char T[T_SIZE+2],P[P_SIZE+2];
//T 是被匹配的串。
//P 是模式串。
void COMPUTE_PREFIX_FUNCTION(char P[])
{
	int m=strlen(P+1);
	next[1]=0;
	for(int k=0,q=2;q<=m;q++)
	{
		while(k>0&&P[k+1]!=P[q])
			k=next[k];
		if(P[k+1]==P[q])
			k++;
		next[q]=k;
	}
}
int KMP_MATCHER(char T[],char P[])
{
	int n=strlen(T+1),m=strlen(P+1);
	COMPUTE_PREFIX_FUNCTION(P);
	int sum=0;
	for(int i=1,q=0;i<=n;i++)
	{
		while(q>0&&P[q+1]!=T[i])
			q=next[q];
		if(P[q+1]==T[i])
			q++;
		if(q==m)
		{
			sum++;
			q=next[q];
		}
	}
	return sum;
}
```

4.3.3 经典题目

1．题目出处/来源

[POJ][3461][KMP] Oulipo

2．题目描述

给出两个字符串，问第二个字符串中存在多少个字串能与第一个字符串匹配。

3．分析

KMP 的模板题，直接套用模板即可。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define T_SIZE 1000000
#define P_SIZE 10000
int next[P_SIZE+2];
```

```
char T[T_SIZE+2],P[P_SIZE+2];
//T 是被匹配的串。
//P 是模式串。
void COMPUTE_PREFIX_FUNCTION(char P[])
{
	int m=strlen(P+1);
	next[1]=0;
	for(int k=0,q=2;q<=m;q++)
	{
		while(k>0&&P[k+1]!=P[q])
			k=next[k];
		if(P[k+1]==P[q])
			k++;
		next[q]=k;
	}
}
int KMP_MATCHER(char T[],char P[])
{
	int n=strlen(T+1),m=strlen(P+1);
	COMPUTE_PREFIX_FUNCTION(P);
	int sum=0;
	for(int i=1,q=0;i<=n;i++)
	{
		while(q>0&&P[q+1]!=T[i])
			q=next[q];
		if(P[q+1]==T[i])
			q++;
		if(q==m)
		{
			sum++;
			q=next[q];
		}
	}
	return sum;
}
int main()
{
	int t;
	for(scanf("%d",&t);t>0;t--)
	{
		scanf("%s%s",P+1,T+1);
		//字符串都是从下标1开始的。
		printf("%d\n",KMP_MATCHER(T,P));
	}
}
```

4.4 RK 字符串匹配算法（RK-hash）

参考文献：

《算法导论》

扩展阅读：

CobbLiu的博客：http://www.cnblogs.com/cobbliu/archive/2012/05/24/2517151.html

编写：卢俊达　　校核：黄李龙

4.4.1 基本原理

如果两个字符串 hash 后的值不相同，则它们肯定不相同；如果它们 hash 后的值相同，它们不一定相同。

RK 算法的基本思想就是：将模式串 P 的 hash 值跟主串 S 中的每一个长度为|P|的子串的 hash 值比较。如果不同，则它们肯定不相等；如果相同，则再诸位比较之。

4.4.2 模板代码

```
#define T_SIZE 1000000
#define P_SIZE 10000
char T[T_SIZE+1],P[P_SIZE+1];

bool matcher(char* T,char * P,int s,int m)
{
	for(int i=0;i<m;i++)
		if(T[s+i]!=P[i])
			return false;
	return true;
}

long long mod(long long a, long long b)
{
	if(a<0)
		return (a%b+b)%b;
	if(a>=b)
		return a%b;
	return a;
}

int RABIN_KARP_MATCHAR(char* T,char* P,long long q)
{
	int n=strlen(T);
	int m=strlen(P);
	long long p=0,t=0,h=1;
	for(int i=0;i<m-1;i++)
		h=(h*26)%q;
	for(int i=0;i<m;i++)
	{
		p=mod(26*p+P[i]-'A',q);
		t=mod(26*t+T[i]-'A',q);
	}
	int sum=0;
	for(int s=0;s<=n-m;s++)
	{
		if(p==t&&matcher(T,P,s,m))
			sum++;
		t=mod(26*(t-h*(T[s]-'A'))+(T[s+m]-'A'),q);
	}
	return sum;
}
```

4.4.3 经典题目

1．题目出处/来源

[HerbustOJ][1551][BM] 基础数据结构——字符串 2 病毒 II

2．题目描述

自从计算机病毒的概念被提出之后，病毒的种类可以说是层出不穷。现在，单纯的病毒是逃不过杀毒软件的。因此现在的病毒往往隐藏一些字符之中来达到蒙混过关的目的。已知连续的字符串"bkpstor"是一段病毒编码，请分析给出的一段字符串中是否包含病毒编码。

3．分析

直接套用模板即可。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<stdio.h>
#include<iostream>
#include<cstring>
using namespace std;

#define T_SIZE 1000000
#define P_SIZE 10000
char T[T_SIZE+1],P[P_SIZE+1];

bool matcher(char* T,char * P,int s,int m)
{
	for(int i=0;i<m;i++)
		if(T[s+i]!=P[i])
			return false;
	return true;
}

long long mod(long long a, long long b)
{
	if(a<0)
		return (a%b+b)%b;
	if(a>=b)
		return a%b;
	return a;
}

int RABIN_KARP_MATCHAR(char* T,char* P,long long q)
{
	int n=strlen(T);
	int m=strlen(P);
	long long p=0,t=0,h=1;
	for(int i=0;i<m-1;i++)
		h=(h*26)%q;
	for(int i=0;i<m;i++)
	{
		p=mod(26*p+P[i]-'A',q);
		t=mod(26*t+T[i]-'A',q);
	}
	int sum=0;
	for(int s=0;s<=n-m;s++)
	{
		if(p==t&&matcher(T,P,s,m))
			sum++;
		t=mod(26*(t-h*(T[s]-'A'))+(T[s+m]-'A'),q);
	}
	return sum;
}

int main()
{
```

```
        while(gets(T))
        {
            int sum=RABIN_KARP_MATCHAR(T,"bkpstor",16381*4733+1);
            printf("%s\n",sum?"Warning":"Safe");
        }
}
```

4.5 扩展 KMP（Z function，Extended KMP）

参考文献：

《扩展 KMP 算法》 刘雅琼

扩展阅读：

LTangd的博客：http://www.cnblogs.com/ltang/archive/2010/11/22/1884581.html

编写：卢俊达　　校核：黄李龙

4.5.1 基本原理

扩展 KMP 算法的主要功能是对指定的一对被匹配串 T 和模式串 P，求出一个 extend[]数组。一句话概括 extend[]的作用，“从 T[i]开始可以与模式串 P 匹配 extend[i]个字符”。

求取 extend[]数组的过程需要利用到一个 next[]数组，next[]数组和 extend 数组的求取过程十分相似。求取方法见模板。

4.5.2 模板代码

```
#define T_SIZE 1000000
#define P_SIZE 10000

int extend[T_SIZE],next[P_SIZE];
char str[T_SIZE+1],mode[P_SIZE+1];

void GetExtendNext(const char* mode, int* next, const int strlen)
{
    int i, a, p, j = -1;
    a = p = next[0] = 0;
    for (i = 1; i < strlen; i++, j--)
    {
        if (j < 0 || i + next[i - a] >= p)
        //j==p-i
        {
            if (j < 0) j = 0, p = i;
            while (p < strlen && mode[p] == mode[j])
                ++p, ++j;
            next[i] = j, a = i;
        }
        //mode[a...p]==mode[0...p-a]
        //mode[i..p]==mode[i-a...p-a]
        else next[i] = next[i - a];
    }
}

void GetExtend(const char* str, const int strlen, int* extend, const char* mode, const int modeLen)
{
    GetExtendNext(mode, next, modeLen);

    int i, a, p, j = -1;
    for (a = p = i = 0; i < strlen; i++,--j)
    {
        if (j < 0 || i + next[i - a] >= p)
        {
            if (j < 0) j =0, p = i;
            while (p < strlen && j < modeLen && str[p] == mode[j])
                ++p, ++j;
```

```
                extend[i] = j, a = i;
            }
            else extend[i] = next[i - a];
        }
    }
```

4.5.3 经典题目

1．题目出处/来源

[POJ][3461][KMP] Oulipo

2．题目描述

给出两个字符串，问第二个字符串中存在多少个字串能与第一个字符串匹配。

3．分析

根据被匹配串和模式串求出 extend[]数组，然后遍历 extend[]数组，记录下数组中值与模式串长度相等的元素的个数。

extend[i]==modeLen，说明被匹配串从第 i+1 个字符开始可以与模式串完全匹配。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include <iostream>
using namespace std;

#define T_SIZE 1000000
#define P_SIZE 10000

int extend[T_SIZE],next[P_SIZE];
char str[T_SIZE+1],mode[P_SIZE+1];

void GetExtendNext(const char* mode, int* next, const int strlen)
{
    int i, a, p, j = -1;
    a = p = next[0] = 0;
    for (i = 1; i < strlen; i++, j--)
    {
        if (j < 0 || i + next[i - a] >= p)
        //j==p-i
        {
            if (j < 0) j = 0, p = i;
            while (p < strlen && mode[p] == mode[j])
                ++p, ++j;
            next[i] = j, a = i;
        }
        //mode[a...p]==mode[0...p-a]
        //mode[i..p]==mode[i-a...p-a]
        else next[i] = next[i - a];
    }
}

void GetExtend(const char* str, const int strlen, int* extend, const char* mode, const int modeLen)
{
    GetExtendNext(mode, next, modeLen);

    int i, a, p, j = -1;
    for (a = p = i = 0; i < strlen; i++,--j)
    {
        if (j < 0 || i + next[i - a] >= p)
        {
            if (j < 0) j =0, p = i;
            while (p < strlen && j < modeLen && str[p] == mode[j])
                ++p, ++j;
            extend[i] = j, a = i;
        }
        else extend[i] = next[i - a];
    }
```

```
}

int main()
{
	int t;
	for(scanf("%d",&t);t>0;t--)
	{
		scanf("%s%s",mode,str);
		int strLen=strlen(str),modeLen=strlen(mode);
		GetExtend(str, strLen, extend, mode, modeLen);
		int sum=0;
		for(int i=0;i<strLen;i++)
			if(extend[i]==modeLen)
				sum++;
		printf("%d\n",sum);
	}
	return 0;
}
```

4.6 字典树（Trie 树）

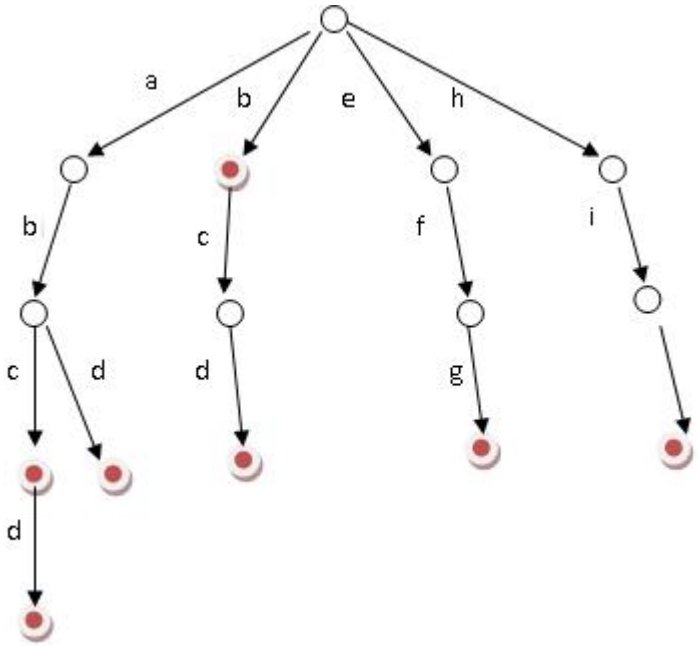
参考文献：

《字典树》 刘春英

编写：卢俊达　　校核：黄李龙

4.6.1 基本原理

字典树，又称 Trie 树，是一种树形结构。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串）。主要思想是利用字符串的公共前缀来节约存储空间。



上图便是一棵字典树，“从根节点到任意一个红色节点的路径上的字母组成一个单词”。

字典树主要包含两种操作，插入和查找。也有一说亦含有删除操作，但这基本属于查找操作的变种，初步学习需要重点掌握插入和查找操作。

4.6.2 模板代码

```
struct trie
//利用结构体来封装字典树的节点。
{
    trie* next[26];
    int num;
    trie()
    //构造函数。
    {
        for(int i=0;i<26;i++)
            next[i]=NULL;
        num=0;
    }
}root;
void insert(char* s)
//将字符串 s 所表示的单词插入到字典树中。
{
    trie *p=&root;
    for(int i=0;s[i]!='\0';i++)
    {
        if(p->next[s[i]-'a']==NULL)
            p->next[s[i]-'a']=new trie;
        p=p->next[s[i]-'a'];
        p->num++;
    }
}
int find(char *s)
//返回值是以 s 为前缀的单词的数量。
{
    trie *p=&root;
    for(int i=0;s[i]!='\0';i++)
        if(p->next[s[i]-'a']==NULL)
            return 0;
        else
            p=p->next[s[i]-'a'];
    return p->num;
}
```

4.6.3 经典题目

4.6.3.1 题目 1

1．题目出处/来源

[HDU][1251][字典树] 统计难题

2．题目描述

Ignatius 最近遇到一个难题，老师交给他很多单词（只有小写字母组成，不会有重复的单词出现），现在老师要他统计出以某个字符串为前缀的单词数量（单词本身也是自己的前缀）。

3．分析

将所有单词插入字典树中，用 num 记录以该节点结尾的单词的数量。

给出指定单词，只要执行查找操作并输出单词结尾节点的 num 参数即可。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;
struct trie
//利用结构体来封装字典树的节点。
{
    trie* next[26];
    int num;
```

```
        trie()
        //构造函数。
        {
                for(int i=0;i<26;i++)
                        next[i]=NULL;
                num=0;
        }
}root;
void insert(char* s)
//将字符串 s 所表示的单词插入到字典树中。
{
        trie *p=&root;
        for(int i=0;s[i]!='\0';i++)
        {
                if(p->next[s[i]-'a']==NULL)
                        p->next[s[i]-'a']=new trie;
                p=p->next[s[i]-'a'];
                p->num++;
        }
}
int find(char *s)
//返回值是以 s 为前缀的单词的数量。
{
        trie *p=&root;
        for(int i=0;s[i]!='\0';i++)
                if(p->next[s[i]-'a']==NULL)
                        return 0;
                else
                        p=p->next[s[i]-'a'];
        return p->num;
}
int main()
{
        char str[11];
        while(gets(str)&&str[0]!='\0')
                insert(str);
        while(~scanf("%s",str))
                printf("%d\n",find(str));
}
```

4.6.3.2 题目 2

1．题目出处/来源

[POJ][1451][字典树] T9

2．题目描述

模拟手机 T9 输入法。先给出 n 个单词和其出现概率，根据用户按下的数字键，选择相应概率最大的单词输出。

3．分析

首先根据给出的单词建树，然后根据按下的数字对字典树按层进行广搜。

每搜完一层节点，就输出该层以 total（出现概率）最大的节点结尾的单词。输出单词利用回溯来完成。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
#include<queue>
using namespace std;
struct trie
//利用结构体来封装字典树的节点。
{
        trie* next[26];
        trie* pre;
```

```
    //指针指向父节点，用于回溯输出单词。
    int total;
    //total 用于记录概率。
    char letter;
    trie()
    //构造函数。
    {
        for(int i=0;i<26;i++)
            next[i]=NULL;
        pre=NULL;
        total=0;
        letter='\0';
    }
};
void insert(char* s,int n,trie* root)
//将字符串 s 所表示的单词插入到字典树中。
{
    trie *p=root;
    for(int i=0;s[i]!='\0';i++)
    {
        if(p->next[s[i]-'a']==NULL)
        {
            p->next[s[i]-'a']=new trie;
            p->next[s[i]-'a']->pre=p;
            p->next[s[i]-'a']->letter=s[i];
        }
        p=p->next[s[i]-'a'];
        p->total+=n;
    }
}
void output(trie* point)
//从 point 开始回溯至 root，输出途径节点的字母，达到输出单词的目的。
{
    if(point->pre->pre!=NULL)
        output(point->pre);
    printf("%c",point->letter);
}
char T9[10][5]={"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
void bfs(char* num,trie* root)
//根据按键广搜
{
    queue<trie*> q;
    q.push(root);
    for(int i=0;num[i]!='1';i++)
    {
        int size=q.size(),max=0;
        //max 记录最大概率。
        trie* now=NULL;
        //now 指向表示概率最大的词的最后一个字母的节点。
        while(size--)
        {
            trie *head=q.front();
            q.pop();
            for(int j=0;j<strlen(T9[num[i]-'0']);j++)
            //遍历 num[i]所表示的几个字母。
            {
                trie *temp=head->next[T9[num[i]-'0'][j]-'a'];
                if(temp!=NULL)
                //若树中存在此单词
                {
                    if(temp->total>max)
                    {
                        max=temp->total;
                        now=temp;
                    }
                    q.push(temp);
                }
```

```
                }
            }
            if(now==NULL)
            //now==NULL 说明没有单词对应现在的按键序列
                printf("MANUALLY");
            else
                output(now);
            printf("\n");
        }
        printf("\n");
    }
    int main()
    {
        int t;
        scanf("%d",&t);
        for(int i=1;i<=t;i++)
        {
            printf("Scenario #%d:\n",i);
            trie root;
            //字典树的根节点。
            int w,m;
            for(scanf("%d",&w);w>0;w--)
            {
                char str[101];
                int p;
                scanf("%s%d",str,&p);
                insert(str,p,&root);
            }
            for(scanf("%d",&m);m>0;m--)
            {
                char num[101];
                scanf("%s",num);
                bfs(num,&root);
                //对每个按键序列进行广搜。
            }
            printf("\n");
        }
    }
```

4.7 后缀数组

参考文献：

《后缀数组——处理字符串的有力工具》 罗穗骞

编写：卢俊达　　校核：黄李龙

4.7.1 基本原理

后缀数组拥有两种常见的实现方式，倍增算法和 DC3 算法。两种算法的时间复杂度分别为 O(nlogn)和 O(n)，空间复杂度为同一级别 O(n)。模板代码为倍增法，以下原理讲解也以被增发为主。

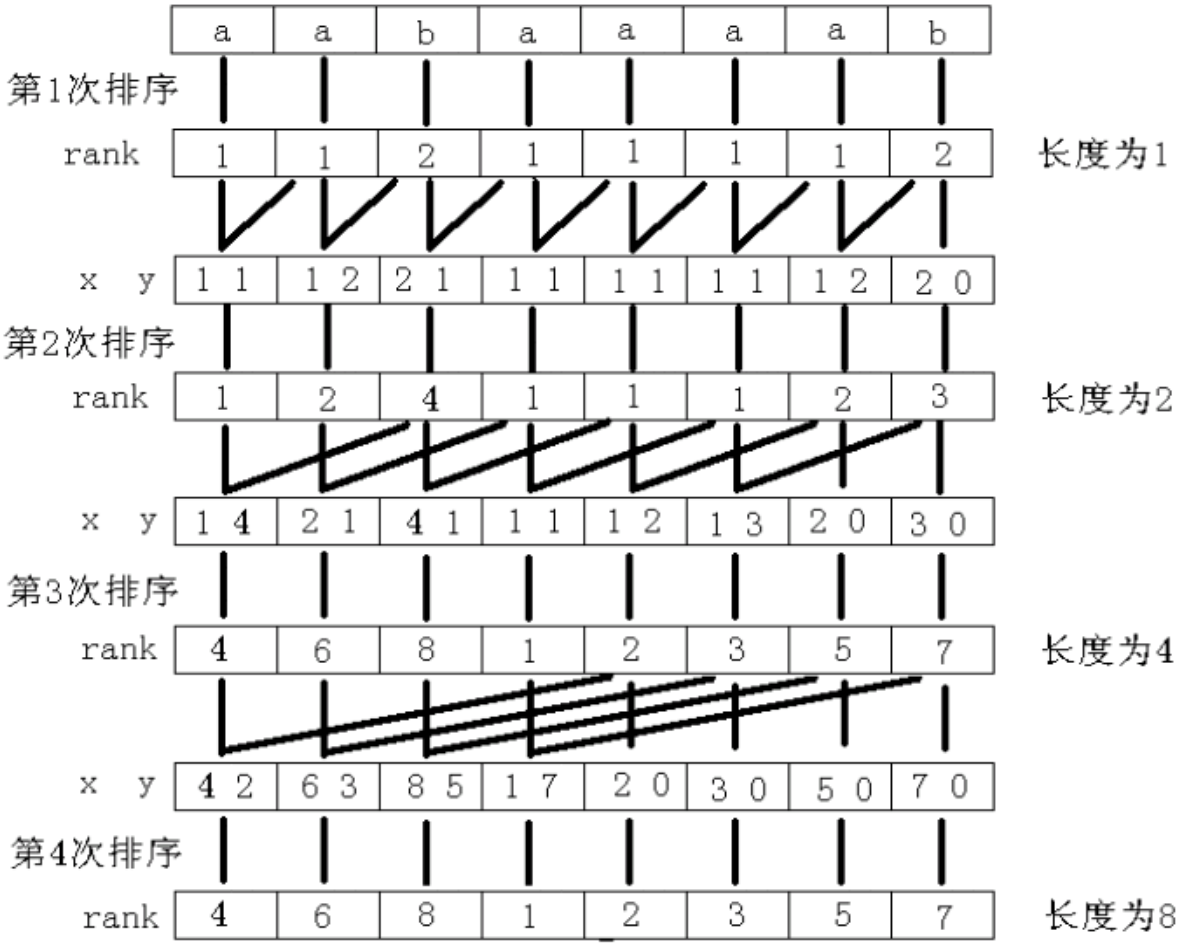
首先明确两个概念，“后缀数组”和“名次数组”。

后缀数组：母串 S 的 n 个后缀从小到大进行排序之后把排好序的后缀的开头位置顺次放入一个数组 sa[]中，该数组便是对应母船 S 的后缀数组。

名次数组：名次数组 rank[]保存的是母串 S 的各个后缀从小到大排列的名次。

简单的说，后缀数组是“排第几的是谁？”，名次数组是“你排第几？”。容易看出，只要知道名次数组或后缀数组其中一个，就可以据此就出对方。

下面讲解倍增法的实现原理。



用倍增的方法对每个字符开始的长度为 2k 的子字符串进行排序，求出排名，即 rank 值。k 从 0 开始，每次加 1，当 2k 大于 n 以后，每个字符开始的长度为 2k 的子字符串便相当于所有的后缀。并且这些子字符串都一定已经比较出大小，即 rank 值中没有相同的值，那么此时的 rank 值就是最后的结果。每一次排序都利用上次长度为 2k-1 的字符串的 rank 值，那么长度为 2k 的字符串就可以用两个长度为 2k-1 的字符串的排名作为关键字表示，然后进行基数排序，便得出了长度为 2k 的字符串的 rank 值。以字符串“aabaaaab”为例，整个过程如图 2 所示。其中 x、y 是表示长度为 2k 的字符串的两个关键字。

这里先介绍后缀数组的一些性质。

height 数组：定义 height[i]=suffix(sa[i-1])和 suffix(sa[i])的最长公共前缀，也就是排名相邻的两个后缀的最长公共前缀。那么对于 j 和 k，不妨设 rank[j]<rank[k]，则有以下性质：

suffix(j) 和 suffix(k) 的最长公共前缀为 height[rank[j]+1]，height[rank[j]+2]，height[rank[j]+3]，……，height[rank[k]]中的最小值。

例如，字符串为“aabaaaab”，求后缀“abaaaab”和后缀“aaab”的最长公共前缀，如图 4 所示：

那么应该如何高效的求出 height 值呢？

如果按 height[2]，height[3]，……，height[n]的顺序计算，最坏情况下时间复杂度为 O(n2)。这样做并没有利用字符串的性质。定义 h[i]=height[rank[i]]，也就是 suffix(i)和在它前一名的后缀的最长公共前缀。

h 数组有以下性质：

h[i]≥h[i-1]-1

证明：

设 suffix(k)是排在 suffix(i-1)前一名的后缀，则它们的最长公共前缀是 h[i-1]。那么 suffix(k+1)将排在 suffix(i)的前面（这里要求 h[i-1]>1，如果 h[i-1]≤1，原式显然成立）并且 suffix(k+1)和 suffix(i)的最长公共前缀是 h[i-1]-1，所以 suffix(i)和在它前一名的后缀的最长公共前缀至少是 h[i-1]-1。按照 h[1]，h[2]，……，h[n]的顺序计算，并利用 h 数组的性质，时间复杂度可以降为 O(n)。

具体实现：

实现的时候其实没有必要保存 h 数组，只须按照 h[1]，h[2]，……，h[n]的顺序计算即可。代码：

```
int rank[maxn],height[maxn];
void calheight(int *r,int *sa,int n)
{
    int i,j,k=0;
    for(i=1;i<=n;i++) rank[sa[i]]=i;
    for(i=0;i<n;height[rank[i++]]=k)
    for(k?k--:0,j=sa[rank[i]-1];r[i+k]==r[j+k];k++);
    return;
}
```

4.7.2 模板代码

```
#define maxn 1000001
#define maxm 256

int wa[maxn],wb[maxn],wv[maxn],wss[maxm];

int cmp(int *r,int a,int b,int l)
{
    return r[a]==r[b]&&r[a+1]==r[b+1];
}

void da(int *r,int *sa,int n,int m)
{
    int i,j,p,*x=wa,*y=wb,*t;
    for(i=0;i<m;i++)
        wss[i]=0;
    for(i=0;i<n;i++)
        wss[x[i]=r[i]]++;
    for(i=1;i<m;i++)
        wss[i]+=wss[i-1];
    for(i=n-1;i>=0;i--)
        sa[--wss[x[i]]]=i;
        //上面 6 行是计数排序，求出母串中长度为 1 的后缀的后缀数组。
    for(j=1,p=1;p<n;j*=2,m=p)
    //j 为当前字符串长度
    {
        for(p=0,i=n-j;i<n;i++)
            y[p++]=i;
        for(i=0;i<n;i++)
            if(sa[i]>=j)
                y[p++]=sa[i]-j;
                //y[]保存的是对第二关键字的排序结果，也是后缀数组。
                //y[i]的具体含义是：母串中所有长度为 j 的后缀，对其按第二关键字大小排序，将排名为 i 的后缀的起始位置存入 y[i]中。
        for(i=0;i<m;i++)
            wss[i]=0;
        for(i=0;i<n;i++)
            wss[ wv[i] = x[ y[i] ] ]++;
        for(i=1;i<m;i++)
            wss[i]+=wss[i-1];
        for(i=n-1;i>=0;i--)
            sa[--wss[wv[i]]]=y[i];
            //上面 6 行是计数排序，在第二关键字排序完毕的基础上对第一关键字进行排序。
        for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
            x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
```

```
            //上面两行是根据后缀数组求出名次数组。
        }
    }
```

4.7.3 经典题目

1．题目出处/来源

[POJ][1743][后缀数组] Musical Theme

2．题目描述

求不可重叠最长重复字串

3．分析

求出 height 后，二分答案，把 height 数组 分成若干份， 每份内的 height 都大于二分的数，观察每份内，是否存在解即可。

当然，这道题规定，如果一段数同时减去一个数后， 与另一段数相同，也算作相同，这里就要用差分思想解决了。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
# include <cstdlib>
# include <cstdio>
# include <cmath>
# include <cstring>

using namespace std;

const int maxn = 100000, oo = 1073741819;

int cmp(int *r,int a,int b,int l)
{
    return r[a]==r[b]&&r[a+l]==r[b+l];
}

int wa[maxn],wb[maxn],wv[maxn],wss[maxn],n;
void da(int *r,int *sa,int n,int m)
{
    int i,j,p,*x=wa,*y=wb,*t;
    for(i=0;i<m;i++)
        wss[i]=0;
    for(i=0;i<n;i++)
        wss[x[i]=r[i]]++;
    for(i=1;i<m;i++)
        wss[i]+=wss[i-1];
    for(i=n-1;i>=0;i--)
        sa[--wss[x[i]]]=i;
        //上面 6 行是计数排序，求出母串中长度为 1 的后缀的后缀数组。
    for(j=1,p=1;p<n;j*=2,m=p)
    //j 为当前字符串长度
    {
        for(p=0,i=n-j;i<n;i++)
            y[p++]=i;
        for(i=0;i<n;i++)
            if(sa[i]>=j)
                y[p++]=sa[i]-j;
                //y[]保存的是对第二关键字的排序结果，也是后缀数组。
                //y[i]的具体含义是：母串中所有长度为 j 的后缀，对其按第二关键字大小排序，将排名为 i 的后缀的起始位置存入 y[i]中。
        for(i=0;i<m;i++)
            wss[i]=0;
        for(i=0;i<n;i++)
            wss[ wv[i] = x[ y[i] ] ]++;
        for(i=1;i<m;i++)
            wss[i]+=wss[i-1];
        for(i=n-1;i>=0;i--)
            sa[--wss[wv[i]]]=y[i];
```

```
            //上面 6 行是计数排序，在第二关键字排序完毕的基础上对第一关键字进行排序。
        for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
            x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
            //上面两行是根据后缀数组求出名次数组。
    }
}

int rank[maxn],height[maxn];
void calheight(int *r,int *sa,int n)
{
    int i,j,k=0;
    for(i=1;i<=n;i++) rank[sa[i]]=i;
    for(i=1;i<n;height[rank[i++]]=k)
    for(k?k--:0,j=sa[rank[i]-1];j+k <=n+1 && i+k <=n+1 && r[j+k]==r[i+k]; k++);
    return;
}

int sa[maxn];
bool check(int mid)
{
    int ll,rr;
    for (int i = 1; i <= n; i++)
    {
        if (height[i] < mid) ll = oo, rr = -oo;
        ll = ll > sa[i] ? sa[i]:ll;
        rr = rr < sa[i] ? sa[i]:rr;
        if (rr- ll >= mid) return true;
    }
    return false;
}

int r[maxn];
void input()
{
    int i;
    memset(r, 0, sizeof(r));
    memset(sa, 0, sizeof(sa));
    scanf("%d", &n);
    for (i = 1; i<= n; i++)
        scanf("%d", &r[i]);
    for (i = 1; i< n; i++)
        r[i] = r[i+1]-r[i];
    r[n] = 0; n--;
    for (i = 1; i<= n; i++)
        r[i] += 89;
}

int main()
{
    while(true)
    {
        input();
        if (n == -1) break;
        r[++n]= 0;
        da(r,sa,n, 1000);
        calheight(r,sa,n);
        int l,r,mid;
        for (l = 0,r = n; l < r;)
        {
            if (check(mid = (l+r+1 >>1))) l = mid;
            else r = mid-1;
        }
        l++;
        printf("%d\n", l >= 5? l:0);
    }
    return 0;
}
```

4.8 AC 自动机

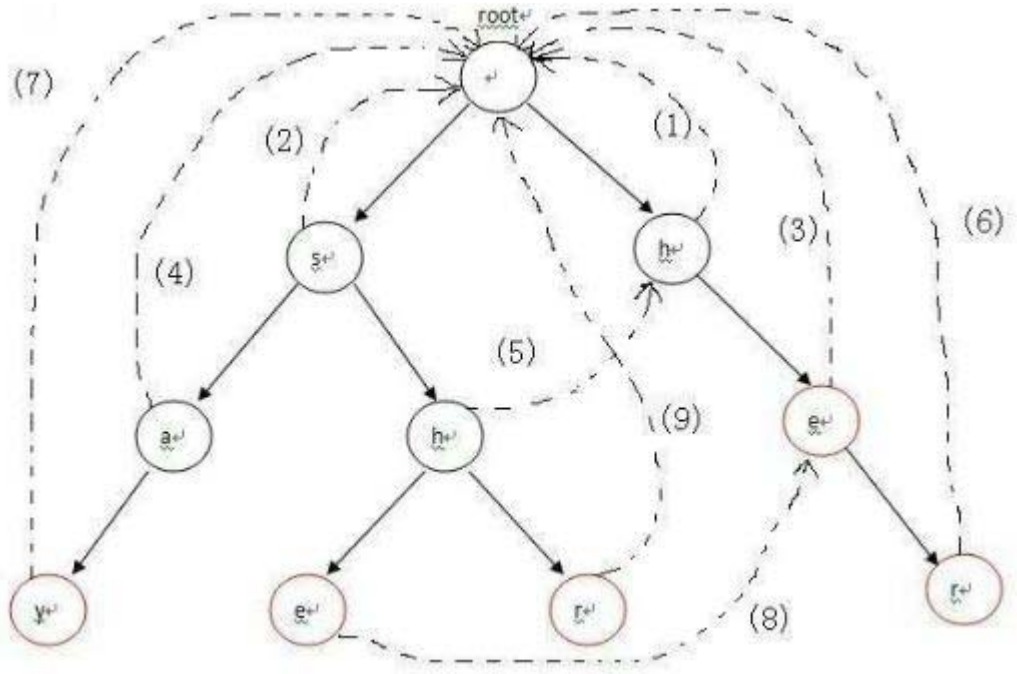
扩展阅读：

DarkRaven的博客：http://hi.baidu.com/nialv7/item/ce1ce015d44a6ba7feded52d

极限定律的博客：http://www.cppblog.com/mythit/archive/2009/04/21/80633.html

编写：卢俊达　　校核：黄李龙

4.8.1 基本原理



首先简要介绍一下 AC 自动机：Aho-Corasick automation，该算法在 1975 年产生于贝尔实验室，是著名的多模匹配算法之一。一个常见的例子就是给出 n 个单词，再给出一段包含 m 个字符的文章，让你找出有多少个单词在文章里出现过。要搞懂 AC 自动机，先得有模式树（字典树）Trie 和 KMP 模式匹配算法的基础知识。AC 自动机算法分为 3 步：构造一棵 Trie 树，构造失败指针和模式匹配过程。

失败指针类似于 KMP 中的 next[]数组。上图是由“say”、“she”、“shr”、“he”、“her”五个单词的字典树，其中虚线表示的就是失败指针。

构造失败指针的过程概括起来就一句话：“设这个节点上的字母为 C，沿着他父亲的失败指针走，直到走到一个节点，他的儿子中也有字母为 C 的节点。然后把当前节点的失败指针指向那个字母也为 C 的儿子。如果一直走到了 root 都没找到，那就把失败指针指向 root。”

失败指针便是 AC 自动机算法中的精髓。究其根本失败指针与 KMP 中的 next[]数组的功能是一样的。

4.8.2 模板代码

```
struct trie
//利用结构体来封装字典树的节点。
{
	trie* next[26];
	trie* fail;
	int num;
	trie()
	//构造函数。
	{
		for(int i=0;i<26;i++)
			next[i]=NULL;
		fail=NULL;
		num=0;
	}
};

char T[T_SIZE+1];
char P[P_SIZE+1];
trie* q[TOTAL_P*P_SIZE];

void insert(trie* root,char* s)
//将字符串 s 所表示的单词插入到字典树中。
{
	trie* p=root;
	for(int i=0;s[i]!='\0';i++)
	{
		if(p->next[s[i]-'a']==NULL)
			p->next[s[i]-'a']=new trie;
		p=p->next[s[i]-'a'];
	}
	p->num++;
}

void	build_ac_automation(trie* root)
//利用广搜构建失败指针
{
	int head=0,tail=0;
	q[tail++]=root;
	while(head!=tail)
	{
		trie* front=q[head++];
		//front 为队头元素
		for(int i=0;i<26;i++)
			if(front->next[i]!=NULL)
			//遍历队头元素的子节点
			{
				trie* p=front->fail;
				while(p!=NULL)
				//只有根节点的失败指针为 NULL
				{
					if(p->next[i]!=NULL)
					//顺着失败指针往回走，直至某个节点，其拥有一个字母为'a'+i 的子节点。
					{
						front->next[i]->fail=p->next[i];
						break;
					}
					p=p->fail;
				}
				if(p==NULL)
				//p==NULL  说明顺着失败指针往回走的过程中没有找到合适的节点，所以将失败指针指向根节点。
					front->next[i]->fail=root;
				q[tail++]=front->next[i];
			}
	}
```

```
}

int ac_find(trie* root,char* T)
//返回值为被匹配串 T 中包含模式串 P 的数量
{
    trie* p=root;
    int sum=0;
    for(int i=0,len=strlen(T);i<len;i++)
    {
        while(p->next[T[i]-'a']==NULL&&p!=root)
        //若当前节点的没有一个字符为 T[i]的儿子且当前节点不是根节点
        //通俗的讲，就是顺着失败指针往回走，直至找到合适的节点或根节点为止。
            p=p->fail;
        if(p->next[T[i]-'a']!=NULL)
        //p->next[T[i]-'a']==NULL 说明没找到合适的节点，p 指针指在根节点上。
            p=p->next[T[i]-'a'];
        trie* temp=p;
        while(temp!=root&&temp->num!=-1)
        //顺着失败指针往回走，一直到根节点。
        {
            sum+=temp->num;
            //若当前节点的 num 不为 0，则说明以当前节点字母结尾的单词出现过一次。
            //此单词是以上一次循环的节点单词为结尾的单词的子集。
            temp->num=-1;
            //标记 num 为-1，避免重复计算。
            temp=temp->fail;
        }
    }
    return sum;
}
```

4.8.3 经典题目

4.8.3.1 题目 1

1．题目出处/来源

[HDU][2222][AC 自动机] Keywords Search

2．题目描述

给出 n 个模式串和一个被匹配串，问有多少个模式串可以成功匹配。

3．分析

将给出的模式串插入字典树中，并构建失败指针。对于给定的被匹配串 T，将其与字典树中的模式串进行匹配，输出匹配数量。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;

#define T_SIZE 1000000
#define P_SIZE 50
#define TOTAL_P 10000

struct trie
//利用结构体来封装字典树的节点。
{
    trie* next[26];
    trie* fail;
    int num;
    trie()
    //构造函数。
    {
```

```
            for(int i=0;i<26;i++)
                next[i]=NULL;
            fail=NULL;
            num=0;
        }
    };

    char T[T_SIZE+1];
    char P[P_SIZE+1];
    trie* q[TOTAL_P*P_SIZE];

    void insert(trie* root,char* s)
    //将字符串 s 所表示的单词插入到字典树中。
    {
        trie* p=root;
        for(int i=0;s[i]!='\0';i++)
        {
            if(p->next[s[i]-'a']==NULL)
                p->next[s[i]-'a']=new trie;
            p=p->next[s[i]-'a'];
        }
        p->num++;
    }

    void   build_ac_automation(trie* root)
    //利用广搜构建失败指针
    {
        int head=0,tail=0;
        q[tail++]=root;
        while(head!=tail)
        {
            trie* front=q[head++];
            //front 为队头元素
            for(int i=0;i<26;i++)
                if(front->next[i]!=NULL)
                //遍历队头元素的子节点
                {
                    trie* p=front->fail;
                    while(p!=NULL)
                    //只有根节点的失败指针为 NULL
                    {
                        if(p->next[i]!=NULL)
                        //顺着失败指针往回走，直至某个节点，其拥有一个字母为'a'+i 的子节点。
                        {
                            front->next[i]->fail=p->next[i];
                            break;
                        }
                        p=p->fail;
                    }
                    if(p==NULL)
                    //p==NULL  说明顺着失败指针往回走的过程中没有找到合适的节点，所以将失败指针指向根
节点。
                        front->next[i]->fail=root;
                    q[tail++]=front->next[i];
                }
        }
    }

    int ac_find(trie* root,char* T)
    {
        trie* p=root;
        int sum=0;
        for(int i=0,len=strlen(T);i<len;i++)
        {
            while(p->next[T[i]-'a']==NULL&&p!=root)
            //若当前节点的没有一个字符为 T[i]的儿子且当前节点不是根节点
            //通俗的讲，就是顺着失败指针往回走，直至找到合适的节点或根节点为止。
```

```
                p=p->fail;
            if(p->next[T[i]-'a']!=NULL)
            //p->next[T[i]-'a']==NULL 说明没找到合适的节点，p 指针指在根节点上。
                p=p->next[T[i]-'a'];
            trie* temp=p;
            while(temp!=root&&temp->num!=-1)
            //顺着失败指针往回走，一直到根节点。
            {
                sum+=temp->num;
                //若当前节点的 num 不为 0，则说明以当前节点字母结尾的单词出现过一次。
                //此单词是以上一次循环的节点单词为结尾的单词的子集。
                temp->num=-1;
                //标记 num 为-1，避免重复计算。
                temp=temp->fail;
            }
        }
        return sum;
    }

    int main()
    {
        int t;
        for(scanf("%d",&t);t>0;t--)
        {
            trie* root=new trie;
            int n;
            scanf("%d",&n);
            getchar();
            for(int i=0;i<n;i++)
            {
                gets(P);
                insert(root,P);
            }
            build_ac_automation(root);
            gets(T);
            printf("%d\n",ac_find(root,T));
        }
    }
```

4.8.3.2 题目 2

1．题目出处/来源

[POJ][1204][AC 自动机] Word Puzzles

2．题目描述

给出一个矩阵和一些字符串，求字符串在矩阵中出现的位置及其方向。

3．分析

遍历四条边，用以边上每个点为起点向三个方向伸展的字符串作为被匹配串并进行匹配。

4．代码（包含必要注释，采用最适宜阅读的 `Courier New` 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;

#define MAP_SIZE 1000
#define P_SIZE 1000
#define TOTAL_P 1000

struct trie
//利用结构体来封装字典树的节点。
{
    trie* next[26];
    trie* fail;
```

```
        int num;
        int len;
        trie()
        //构造函数。
        {
                for(int i=0;i<26;i++)
                        next[i]=NULL;
                fail=NULL;
                len=0;
                num=0;
        }
};

struct point
{
        int x;
        int y;
        char dir;
};

char P[P_SIZE+1];
char map[MAP_SIZE+1][MAP_SIZE+1];
trie* q[TOTAL_P*P_SIZE];
point res[TOTAL_P];
int L,C,W;

void insert(trie* root,char* s,int num)
//将字符串 s 所表示的单词插入到字典树中。
{
        trie* p=root;
        for(int i=0;s[i]!='\0';i++)
        {
                if(p->next[s[i]-'A']==NULL)
                        p->next[s[i]-'A']=new trie;
                p=p->next[s[i]-'A'];
        }
        p->num=num;
        p->len=strlen(s);
}

void    build_ac_automation(trie* root)
//利用广搜构建失败指针
{
        int head=0,tail=0;
        q[tail++]=root;
        while(head!=tail)
        {
                trie* front=q[head++];
                //front 为队头元素
                for(int i=0;i<26;i++)
                        if(front->next[i]!=NULL)
                        //遍历队头元素的子节点
                        {
                                trie* p=front->fail;
                                while(p!=NULL)
                                //只有根节点的失败指针为 NULL
                                {
                                        if(p->next[i]!=NULL)
                                        //顺着失败指针往回走，直至某个节点，其拥有一个字母为'A'+i 的子节点。
                                        {
                                                front->next[i]->fail=p->next[i];
                                                break;
                                        }
                                        p=p->fail;
                                }
                                if(p==NULL)
                                //p==NULL  说明顺着失败指针往回走的过程中没有找到合适的节点，所以将失败指针指向根
```

节点。

```
                                front->next[i]->fail=root;
                            q[tail++]=front->next[i];
                        }
                }
        }

        bool range(int i,int j)
        //判断坐标（i，j）是否在字谜矩阵范围内。
        {
                return i>=0&&j>=0&&i<L&&j<C?true:false;
        }

        int dir[8][2]={{-1,0},{-1,1},{0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1}};
        void ac_find(trie* root,int i,int j,int d)
        //用以（i，j）为起点，向 d 方向伸展的字符串作为被匹配串，并与模式串进行匹配。
        {
                trie* p=root;
                for(;range(i,j);i+=dir[d][0],j+=dir[d][1])
                {
                        while(p->next[map[i][j]-'A']==NULL&&p!=root)
                        //若当前节点的没有一个字符为 map[i][j]的儿子且当前节点不是根节点
                        //通俗的讲，就是顺着失败指针往回走，直至找到合适的节点或根节点为止。
                                p=p->fail;
                        if(p->next[map[i][j]-'A']!=NULL)
                        //p->next[map[i][j]-'A']==NULL 说明没找到合适的节点，p 指针指在根节点上。
                                p=p->next[map[i][j]-'A'];
                        trie* temp=p;
                        while(temp!=root&&temp->num!=-1)
                        //顺着失败指针往回走，一直到根节点。
                        {
                                if(temp->len!=0)
                                {
                                        res[temp->num].x=i-dir[d][0]*(temp->len-1);
                                        res[temp->num].y=j-dir[d][1]*(temp->len-1);
                                        res[temp->num].dir='A'+d;
                                }
                                temp->num=-1;
                                //标记 num 为-1，避免重复计算。
                                temp=temp->fail;
                        }
                }
        }

        int main()
        {
                trie* root=new trie;
                scanf("%d%d%d",&L,&C,&W);
                getchar();
                for(int i=0;i<L;i++)
                        gets(map[i]);
                for(int i=0;i<W;i++)
                {
                        gets(P);
                        insert(root,P,i);
                }
                build_ac_automation(root);
                for(int i=0;i<L;i++)
                //以字谜矩阵的左右两条边上的点为起点向三个方向伸展的字符串作被匹配串。
                {
                        ac_find(root,i,C-1,5);
                        ac_find(root,i,C-1,6);
                        ac_find(root,i,C-1,7);
                        ac_find(root,i,0,1);
                        ac_find(root,i,0,2);
                        ac_find(root,i,0,3);
                }
```

```
        for(int i=0;i<C;i++)
        //同理，遍历上下两条边。
        {
                ac_find(root,L-1,i,0);
                ac_find(root,L-1,i,1);
                ac_find(root,L-1,i,7);
                ac_find(root,0,i,3);
                ac_find(root,0,i,4);
                ac_find(root,0,i,5);
        }
        for(int i=0;i<W;i++)
                printf("%d %d %c\n",res[i].x,res[i].y,res[i].dir);
}
```

4.8.3.3 题目 3

1．题目出处/来源

[HDU][3965][AC 自动机] Computer Virus on Planet Pandora

2．题目描述

给出一些模式串和一个被匹配串，若被匹配串包含模式串或其反转，则称该模式串感染了被匹配串。问有多少个模式串感染了被匹配串。

3．分析

用模式串去匹配被匹配串及其反转。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;

#define T_SIZE 5100000
#define P_SIZE 1000
#define TOTAL_P 250

struct trie
//利用结构体来封装字典树的节点。
{
        trie* next[26];
        trie* fail;
        int num;
        trie()
        //构造函数。
        {
                for(int i=0;i<26;i++)
                        next[i]=NULL;
                fail=NULL;
                num=0;
        }
};

char T[T_SIZE+1];
char temp_T[T_SIZE+1];
char P[P_SIZE+1];
trie* q[TOTAL_P*P_SIZE];

void insert(trie* root,char* s)
//将字符串 s 所表示的单词插入到字典树中。
{
        trie* p=root;
        for(int i=0;s[i]!='\0';i++)
        {
                if(p->next[s[i]-'A']==NULL)
                        p->next[s[i]-'A']=new trie;
                p=p->next[s[i]-'A'];
        }
```

```
        p->num++;
    }

    void    build_ac_automation(trie* root)
    //利用广搜构建失败指针
    {
        int head=0,tail=0;
        q[tail++]=root;
        while(head!=tail)
        {
            trie* front=q[head++];
            //front 为队头元素
            for(int i=0;i<26;i++)
                if(front->next[i]!=NULL)
                //遍历队头元素的子节点
                {
                    trie* p=front->fail;
                    while(p!=NULL)
                    //只有根节点的失败指针为 NULL
                    {
                        if(p->next[i]!=NULL)
                        //顺着失败指针往回走，直至某个节点，其拥有一个字母为'A'+i 的子节点。
                        {
                            front->next[i]->fail=p->next[i];
                            break;
                        }
                        p=p->fail;
                    }
                    if(p==NULL)
                    //p==NULL  说明顺着失败指针往回走的过程中没有找到合适的节点，所以将失败指针指向根节点。
                        front->next[i]->fail=root;
                    q[tail++]=front->next[i];
                }
        }
    }

    int ac_find(trie* root,char* T)
    {
        trie* p=root;
        int sum=0;
        for(int i=0,len=strlen(T);i<len;i++)
        {
            while(p->next[T[i]-'A']==NULL&&p!=root)
            //若当前节点的没有一个字符为 T[i]的儿子且当前节点不是根节点
            //通俗的讲，就是顺着失败指针往回走，直至找到合适的节点或根节点为止。
                p=p->fail;
            if(p->next[T[i]-'A']!=NULL)
            //p->next[T[i]-'A']==NULL 说明没找到合适的节点，p 指针指在根节点上。
                p=p->next[T[i]-'A'];
            trie* temp=p;
            while(temp!=root&&temp->num!=-1)
            //顺着失败指针往回走，一直到根节点。
            {
                sum+=temp->num;
                //若当前节点的 num 不为 0，则说明以当前节点字母结尾的单词出现过一次。
                //此单词是以上一次循环的节点单词为结尾的单词的子集。
                temp->num=-1;
                //标记 num 为-1，避免重复计算。
                temp=temp->fail;
            }
        }
        return sum;
    }

    void get_T(char* temp_T)
    //解压母串
```

```
{
	int len=strlen(temp_T),len_T=0;
	for(int i=0,num;i<len;i++)
		if(temp_T[i]>='A'&&temp_T[i]<='Z')
			T[len_T++]=temp_T[i];
		else if(temp_T[i]=='[')
			num=0;
		else if(temp_T[i]>='0'&&temp_T[i]<='9')
			num=(num+(temp_T[i]-'0'))*10;
		else if(temp_T[i]==']')
		{
			num/=10;
			for(int k=0;k<num-1;k++)
				T[len_T++]=temp_T[i-1];
		}
	T[len_T]='\0';
}

void reverse(char* str)
//反转字符串
{
	int len=strlen(str);
	for(int i=0;i<len/2;i++)
		swap(str[i],str[len-i-1]);
}

int main()
{
	int t;
	for(scanf("%d",&t);t>0;t--)
	{
		trie* root=new trie;
		int n;
		scanf("%d",&n);
		getchar();
		for(int i=0;i<n;i++)
		{
			gets(P);
			insert(root,P);
		}
		build_ac_automation(root);
		gets(temp_T);
		get_T(temp_T);
		int res=0;
		res=ac_find(root,T);
		reverse(T);
		res+=ac_find(root,T);
		printf("%d\n",res);
	}
}
```

4.9 后缀自动机

参考文献：

杭州外国语学校 陈立杰 《后缀自动机 Suffix Automaton》ppt

后缀自动机(FHQ+Neroysq 补完)
http://hi.baidu.com/myidea/item/142c5cd45901a51820e25039

[SPOJ]后缀自动机应用 http://neroysq.blogcn.com/?p=50

后缀自动机初探 http://neroysq.blogcn.com/articles/后缀自动机初探.html

后缀自动机的应用http://blog.sina.com.cn/s/blog_7812e98601012dfv.html

编写：黄李龙　　校核：黄李龙

4.9.1 后缀自动机介绍

学习这部分内容时，假设你已经学会了字典树和 KMP 关于 next 函数的含义，next 函数也可以成为 fail 函数，有些文章把 fail 函数叫做 fail 指针，或者用 parent 替代，代表的含义是一样的。最好也能把 AC 自动机看一遍，虽然后缀自动机的代码很短，比 AC 自动机还短，但是不是很好理解。如果你已经掌握了以上内容，那么可以开始了。

在看后缀自动机之前，请在百度文库里搜索后缀自动的资料，找到杭州外国语学校陈立杰《后缀自动机 Suffix Automaton》的 ppt 资料，把这份 ppt 看一遍。

下面摘自该 ppt：

什么是自动机

- 有限状态自动机的功能是识别字符串，令一个自动机 A，若它能识别字符串 S，就记为 A(S)=True，否则 A(S)=False。
- 自动机由五个部分组成，alpha：字符集，state：状态集合，init：初始状态，end：结束状态集合，trans：状态转移函数。
- 不妨令 trans(s,ch)表示当前状态是 s，在读入字符 ch 之后，所到达的状态。
- 如果 trans(s,ch)这个转移不存在，为了方便，不妨设其为 null，同时 null 只能转移到 null。
- null 表示不存在的状态。

- 同时令 trans(s,str)表示当前状态是 s，在读入字符串 str 之后，所到达的状态。有如下伪代码：

```
Cur = s;
For i = 0 to Length(str)-1
    Cur = trans(Cur,str[i]);
trans(s,str) 就是最后的 Cur。
```

后缀自动机的定义

给定字符串 S，S 的后缀自动机 Suffix Automaton(以后简记为 SAM)是一个能够识别 S 的所有后缀的自动机。即 SAM(x) = True，当且仅当 x 是 S 的后缀。同时，后缀自动机也能用来识别 S 所有的子串。

一些性质和定理证明请参考陈立杰的 ppt。

算法流程：

当你看完这份 ppt 之后，可能还会有不理解的地方，当初我自己在看的时候也不理解，上网搜了一下，找到关于算法运行过程的解释。我找到的一个比较好的解释就是下面

关于后缀自动机的解释。

来源：http://hi.baidu.com/myidea/item/142c5cd45901a51820e25039（有部分改动）

下面以字符串“aabbabd”构造后缀自动机(请耐心看，因为前面几步没有体现算法)

有几点要记住：

1.由一个接受态沿 Parent 往前走所到的状态也是接受态。

（接受态可以理解为可以接受的合法状态，这里所说的 Parent 其实是 fail 指针的指向）

2.一个节点及其父辈的代表的串有相同的后缀。

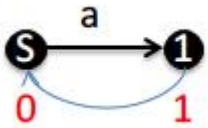
（这个很好理解，可以结合下面图示去理解）

1.首先初始为一个空串，表示根节点。

此时后缀只有一个，就是空串。

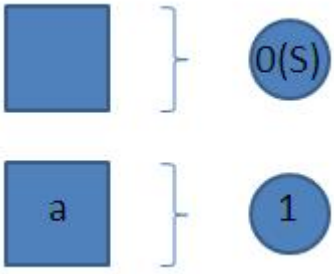
红字表示此节点代表最长串的长度，一个节点可能代表多个串。

2.现在构建"a"的自动机，就是下面这样：



蓝色的边就是 fail 指向，可以结合 AC 自动机里的 fail 指针理解。

现在有这些后缀：

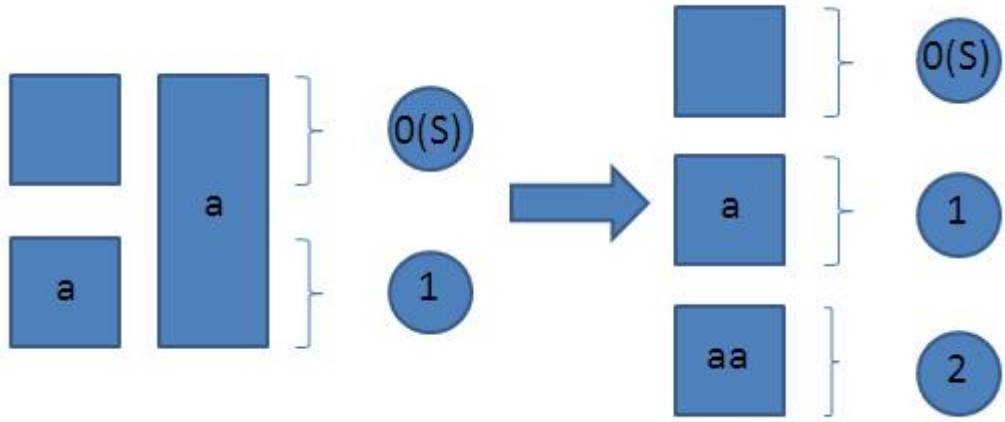


3.然后以上面的后缀为基础构建"aa"的自动机

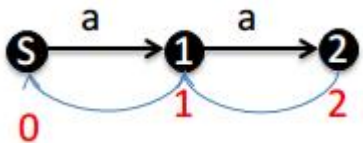
现在想一下，由 S 或者说 0 号节点可以接受的后缀为空串和"a"这两个，那么现在要将"aa"和"a"这两个后缀更新到后缀自动机中，1 号节点的后缀"a"就要加入一个字符"a"，而空串也要加入字符"a"，也就是所有之前的后缀都要在后面加入一个字符"a".

但是由于 1 号节点之前所代表的后缀"a"和 1 的 Parent 所代表的后缀(空串)+"a"代表的一样，所以无需更新 1 及之前的可接受态。

如下图：

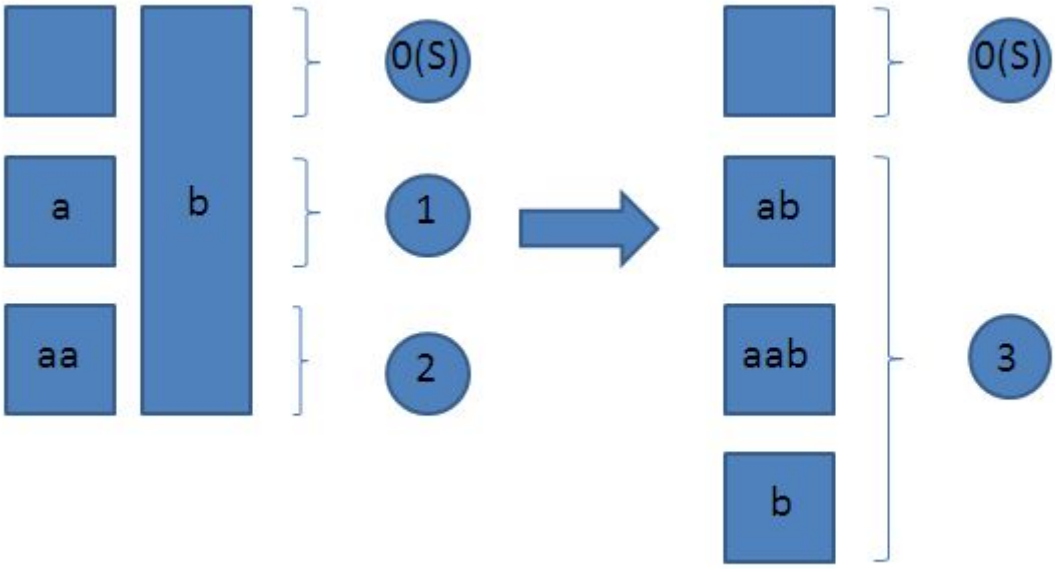


自动机就变成了如下形式：



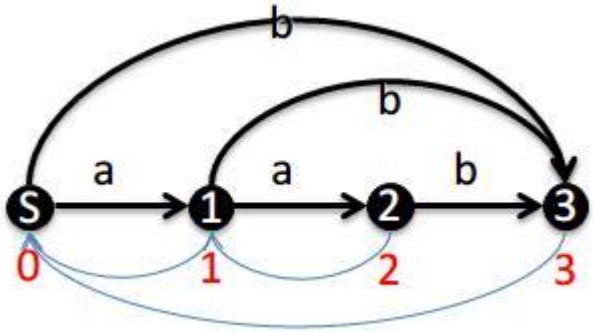
3.更新自动机变成"aab"自动机

同上加所有接受态也要调整,就是在后面加上"b"字符:



由于 1，2 节点无法代表三个后缀的任意一个，所以除空串的所有后缀都由 3 代替，这时 3 号节点和 0 号节点为接受态。

自动机的形式如下:



这一步具体过程：

S1：新建节点 3；

S2：找到最后一个后缀也就是最后一个接受态是节点 2；

S3：2 号节点直接连向 3，表示插入后缀"aab"；

S4：向上找 2 的 Parent，1 号节点，向 3 连边，表示插入后缀"ab"；

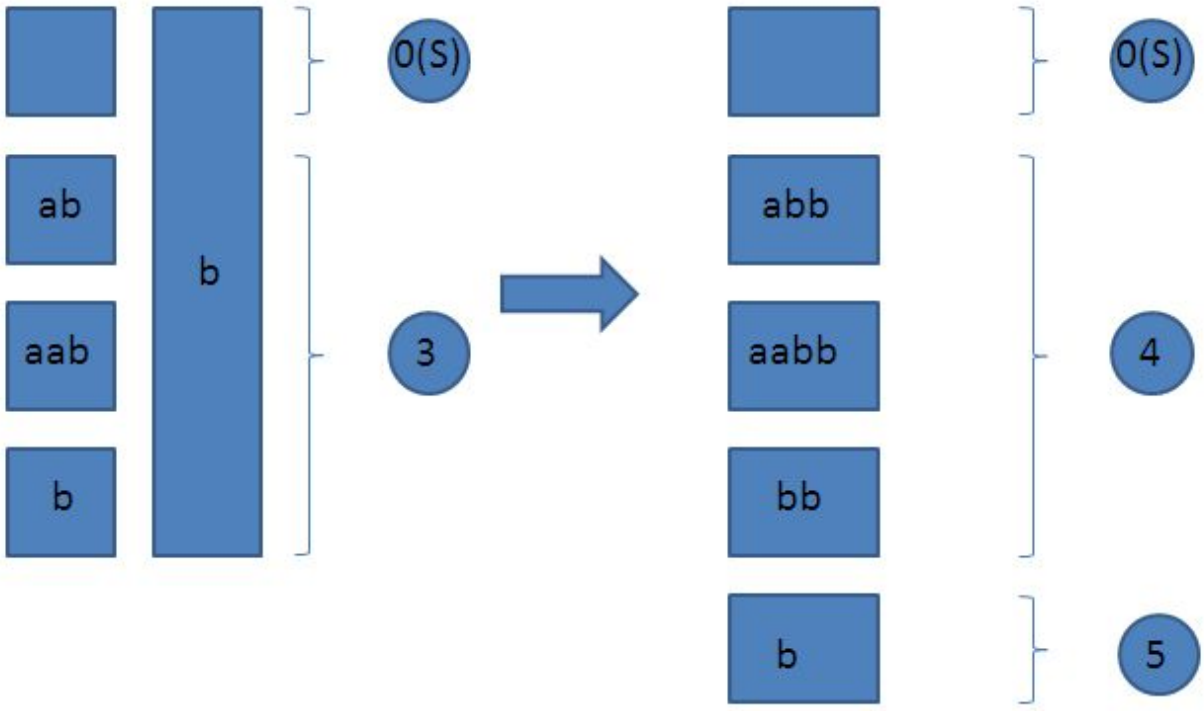
S5：找到 S,连边,表示插入后缀"b"；

S6：没有其他接受态了，那么 3 的上一个接受态为 S，Parent[3]=S

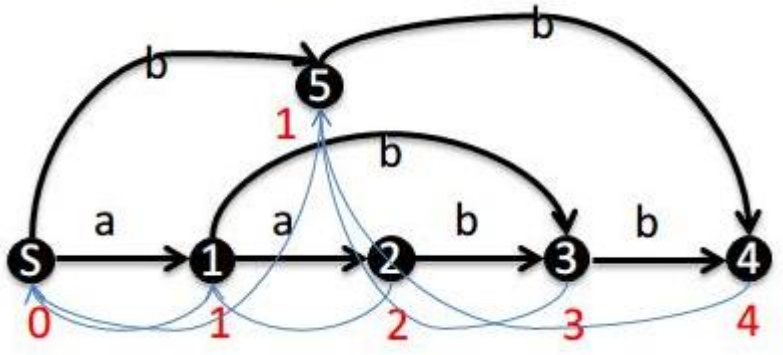
4.更新成"aabb"的自动机

同理，在所有接受态后加上字符"b"。

不过由于接受态 0(S)的转移"b"已经存在，那么,由于不能破坏原来的中间态的转移，只能新建一个节点，来代替接受态 0(S)的转移节点。



自动机成了这样:



找到 S 根节点时，发现转移"b"已经有节点占了，所以新建节点 5，将 3 号所有信息 Copy(包括 Parent)，然后更新 len 值，就是 node[5]->len=node[5]->parent->len+1，所以 5 号节点可以代表后缀空串(0 号代表的串)+字符"b"=后缀"b"，**节点 3 成了中间态，所以将所有孩子指针指向 3 节点的原接受态的孩子指针改为指向 3 的转移改为指向 5**，这时，我们发现孩子指针指向 3 节点的原接受态节点一定是当前节点 S 及当前未访问的原接受态节点，所以可以直接沿着 Parent 往上更新。

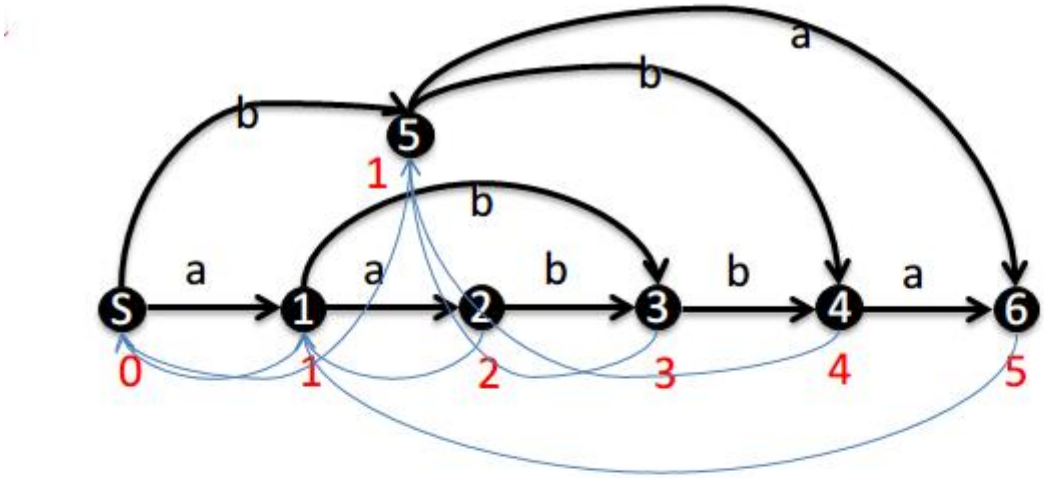
然后节点 5 的 Parent 及祖先加入了现在的接受态。

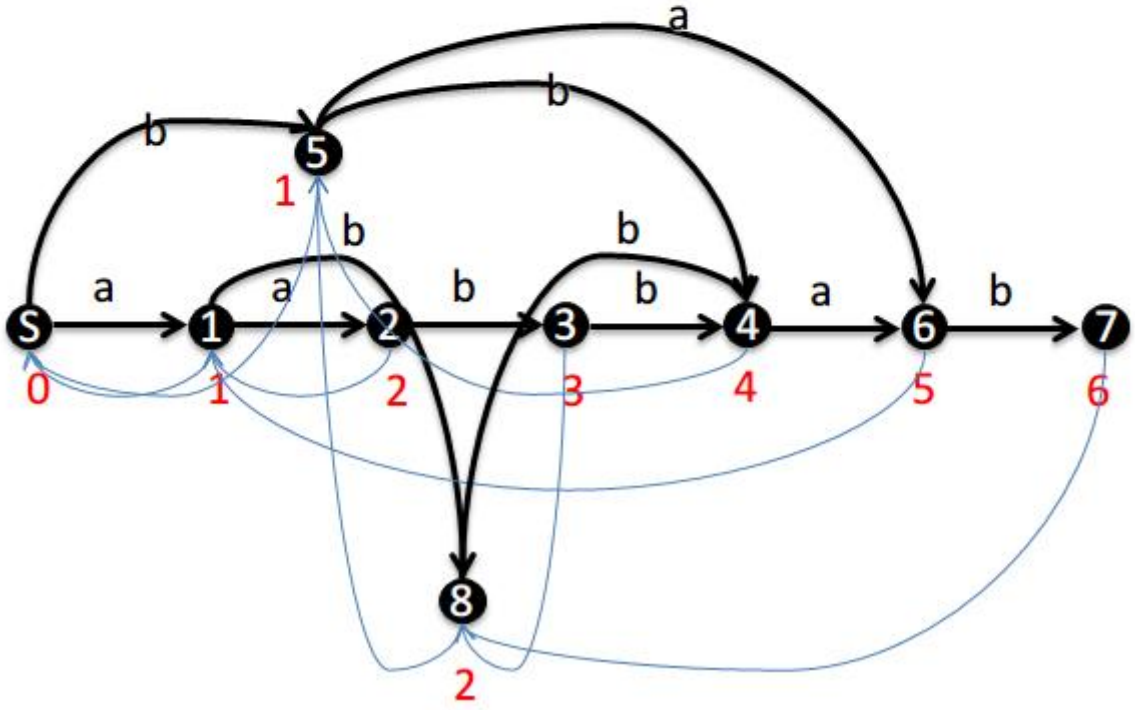
再次重申一点：**一个节点及其父辈的代表的串有相同的后缀，且代表串长度递减，由于 5 号节点是接受态，所以他的父辈也是接受态，同时反过来也一样，与任意接受态拥有相同后缀的长度小于当前节点的未访问节点一定是当前节点的父辈，如与 5 号节点有相同后缀的长度小于 5 号节点的未访问的节点一定是 5 号的父辈，一定可以作为接受态。**

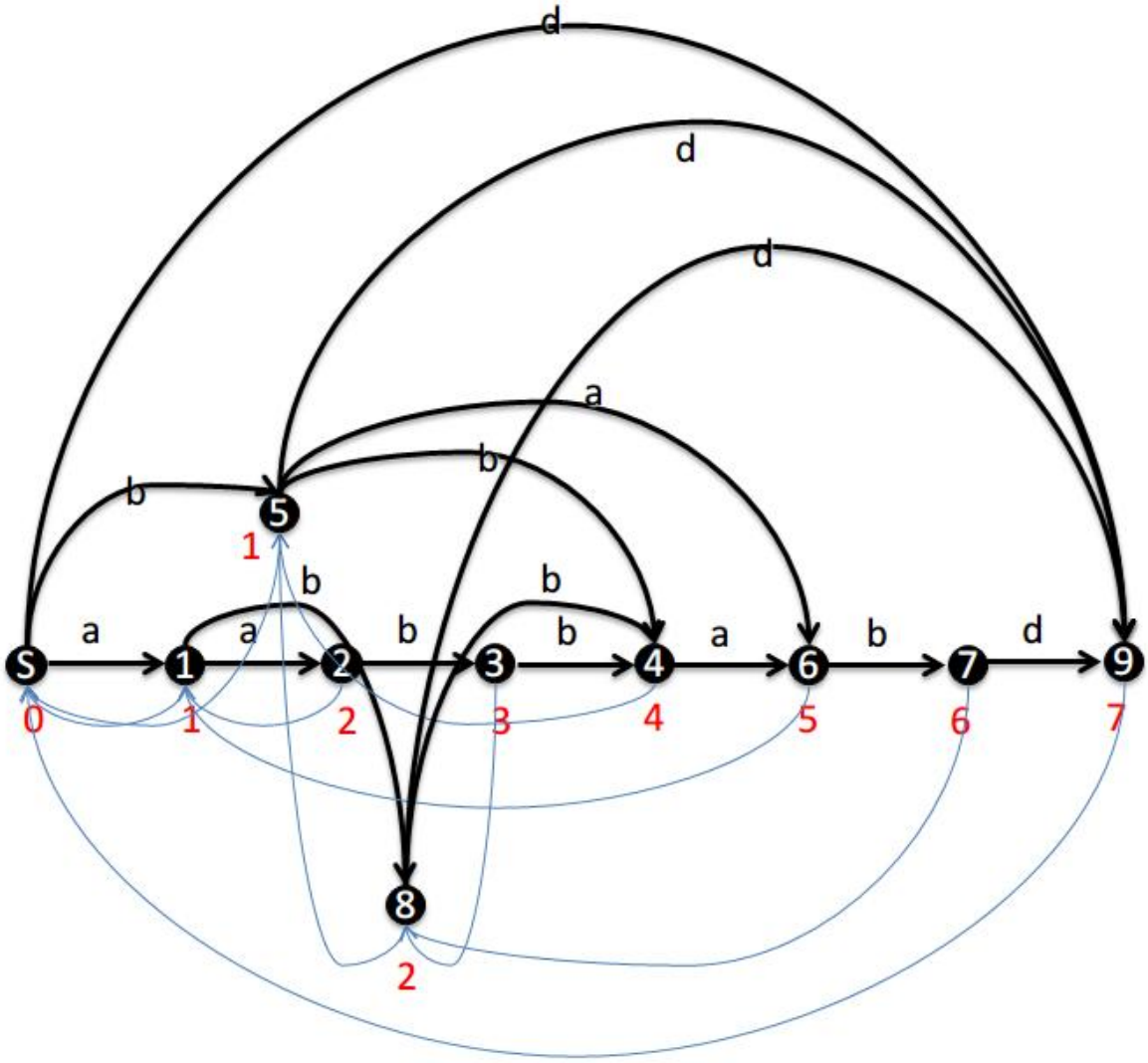
因此为了维护这个性质，**我们应该将 3 号节点的父亲重定义为 5。**

到这里基本上应该明白了。

就将剩下的构造过程放出来：


a
b
b
5
1
b
a
a
b
b
a
S
1
2
3
4
6
0
1
2
3
4
5


a
b
b
5
1
b
b
a
a
b
b
a
b
S
1
2
3
4
6
7
0
1
2
3
4
5
6
8
2



代码什么的如下：
插入一个节点：

```
void add(int c, int len)
{
    sanode *p = tail, *np = &pool[++tot];
    np->ml = len;
    for (; p && !p->ch[c]; p = p->f) p->ch[c] = np;
    tail = np;
    if (!p) np->f = init;
    else
        if (p->ch[c]->ml == p->ml + 1) np->f = p->ch[c];
        else {
            sanode *q = p->ch[c], *r = &pool[++tot];
            *r = *q;
            r->ml = p->ml + 1;
            q->f = np->f = r;
            for (; p && p->ch[c] == q; p = p->f) p->ch[c] = r;
        }
}
```

意义：

当前新建节点为 np,最后的接受态为 tail，接受态指针 p，pool 是内存池，就是一个很大的数组，开 2*n 的空间即可。

init 就是 0 号节点

Step 1：建立节点 np,p 指向 tail,np 的 len 更新；

Step 2：沿着 Parent 寻找第一个有转移冲突的接受态节点并沿途更新转移；

Step 3：

1. 如果找不到,np 的 Parent 更新为 init；
2. 如果正好冲突的节点长度为当前接受态节点那么 np 的 Parent 赋为冲突的节点。
3. 否则，新建节点 r，Copy 冲突节点所有信息，更新 r->len=p->len+1，将冲突节点的 Parent 和 np 的 Parent 赋为 r，再往前更新与冲突节点有关的转移。

时间复杂和空间复杂度：

时间复杂度其实我也不太明白，但是论文里都证明了时间复杂度是 O(N)级别的，就把这个结论记下吧。

在 add 函数里，每次加入一个字符，最多会多新建一个节点，很容易知道空间复杂度为 O(N)级别的。

看完这些可能还是不理解，请继续上网查资料。

4.9.2 例题讲解

4.9.2.1 SPOJ NSUBSTR Substrings

http://www.spoj.pl/problems/NSUBSTR

题意：给一个字符串 S，令 F(x)表示 S 的所有长度为 x 的子串中，出现次数的最大值。求 *F(1)..F(Length(S))* 。

字符串长度小于等于 250000。

分析：

这个是求每种长度的子串出现的最大次数，问的是子串，怎么和后缀联系呢？每一个子串都曾经是后缀，我们来研究研究后缀自动机的性质，对于一个节点 a，我们可以知道长度 len[a]，那就是它可以接收的最大长度的后缀，虽然这里我们称它为后缀，但是，当我们加入字符后，它可能就不是后缀了，于是我们干脆叫它可以接收的最大长度的子串。另外，这只是它可以接收的最大长度的后缀，我们还要解决它可能接收的更短长度的后缀，这个只要我们知道，当 x<y 时 F[x]>=F[y]，因为每个长串都包含各种长度的短串，于是我们按长串到短串的顺序扫描，对于每个点 i，维护 c[i]值，代表这个点到根的字符串的作为要匹配的串在整个字符串中出现次数。扫到一个点 i 的时候，可以用当前点的 c[i]值累加到他父亲（parent 或者 fail 指向的节点）的 c 值上去。为什么这样呢？参考陈立杰后缀自动机中 Right 集合的性质。

C++代码：将后缀自动机封装成了结构体，方便使用。

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 250005;
```

```
struct SuffixAutomaton {
 struct Node {
     int len;
     Node *f, *ch[26];
     Node() {}
     Node(int l) :len(l), f(NULL) {
         memset(ch, 0, sizeof(ch));
     }
 };
 Node *root, *last;
 Node pool[MAXN*2];
 int cnt;

 void init() {
     root = last = pool;
     memset(root, 0, sizeof(Node));
     cnt = 1;
 }
 Node *new_node(int l = 0) {
     Node *ptr = pool + cnt++;
     memset(ptr, 0, sizeof(Node));
     if (l != 0) ptr->len = l;
     return ptr;
 }
 void append(char ch) {
     int c = ch - 'a';
     Node *p = last, *np = new_node(last->len+1);
     last = np;
     for (; p && NULL==p->ch[c]; p = p->f) p->ch[c] = np;
     if (p == NULL) {
         np->f = root;
     } else {
         if (p->ch[c]->len == p->len+1) {
             np->f = p->ch[c];
         } else {
             Node *q = p->ch[c], *nq = new_node();
             *nq = *q;
             nq->len = p->len + 1;
             q->f = np->f = nq;
             for (; p && p->ch[c] == q; p = p->f) p->ch[c] = nq;
         }
     }
 }
};

template <class T>
void check_max(T &a, const T &b) {
 if (a < b) a = b;
}

SuffixAutomaton sam;
char str[MAXN];
int buc[MAXN];
int sorted[MAXN*2];
int c[MAXN*2];
int F[MAXN];

int main() {
 while (EOF != scanf("%s", str)) {
     int ls = strlen(str);
     sam.init();
     for (int i = 0; str[i]; ++i) {
         sam.append(str[i]);
     }
     // 基数排序，求出拓扑序列,按照长度从短到长即可
     memset(buc, 0, sizeof(buc));
     for (int i = 0; i < sam.cnt; ++i) ++buc[sam.pool[i].len];
     for (int i = 1; i <= ls; ++i) buc[i] += buc[i-1];
     for (int i = 0; i < sam.cnt; ++i) sorted[ --buc[ sam.pool[i].len ] ] = i;
```

```
        SuffixAutomaton::Node *ptr = sam.root;
        c[0] = 0;
        for (int i = 0; i < ls; ++i) c[ (ptr=ptr->ch[str[i]-'a']) - sam.pool ] = 1; //
包含自身的算一个
        memset(F, 0, sizeof(F));
        for (int i = sam.cnt-1; i >= 0; --i) {
            check_max(F[ sam.pool[ sorted[i] ].len], c[ sorted[i] ]);
            // 从 fail 指针找到一个
            if (NULL != sam.pool[ sorted[i] ].f) c[sam.pool[ sorted[i] ].f - sam.pool]
+= c[ sorted[i] ];
        }
        for (int i = 1; i <= ls; ++i) printf("%d\n", F[i]);
    }
    return 0;
   }
```

4.9.2.2 SPOJ LCS Longest Common Substring

http://www.spoj.pl/problems/LCS/

题意：给两个字符串 a 和 b，求 a 和 b 的最长公共子串长度。注意：这里的子串要求是连续的子串。

字符串 a 和 b 的长度不超过 250000。

分析：

先以 a 为母串建立后缀自动机，考虑以 b[i]结尾的 b 的子串最长可以和 a 串匹配的长度，设 b[i]为字符 x，S 为一个字符串，如果 Sx 可以被后缀自动机匹配，那么 S 一定也可以匹配，反之 S 不可匹配，那么 xSx 这一类的一定都不可以匹配，因此，如果我们求出了对于 b[i]结尾的字符串可以匹配的最长长度 L，那么 b[i+1]最多可以匹配的长度为 L+1，所以我们在枚举 b 的子串的结尾的过程中维护这个 L 即可。

C++代码：

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 250005;
struct SuffixAutomaton {
 struct Node {
     int len;
     Node *f, *ch[26];
     Node() {}
     Node(int l) : len(l), f(NULL) {
         memset(ch, 0, sizeof(ch));
     }
 };
 Node *root, *last;
 Node pool[MAXN*2];
 int cnt;

 void init() {
     root = last = pool;
     memset(root, 0, sizeof(Node));
     cnt = 1;
 }
 Node* new_node(int l = 0) {
     Node *x = pool + cnt++;
     memset(x, 0, sizeof(Node));
     if (l != 0) x->len = l;
     return x;
 }
 void append(char ch) {
     int c = ch - 'a';
     Node *p = last, *np = new_node(last->len+1);
     last = np;
```

```
        for (; NULL != p && NULL == p->ch[c]; p = p->f) p->ch[c] = np;
        if (NULL == p) {
            np->f = root;
        } else {
            if (p->ch[c]->len == p->len+1) {
                np->f = p->ch[c];
            } else {
                Node *q = p->ch[c], *nq = new_node();
                *nq = *q;
                nq->len = p->len + 1;
                q->f = np->f = nq;
                for (; NULL != p && p->ch[c] == q; p = p->f) p->ch[c] = nq;
            }
        }
    }
};
SuffixAutomaton sam;
char str[MAXN];

int main() {
    while (EOF != scanf("%s", str)) {
        sam.init();
        for (int i = 0; str[i]; ++i) sam.append(str[i]);
        scanf("%s", str);
        int ans = 0, nl = 0;
        SuffixAutomaton::Node *p = sam.root;
        for (int i = 0; str[i]; ++i) {
            int c = str[i] - 'a';
            if (p->ch[c] != NULL) {
                p = p->ch[c];
                ++nl;
            } else {
                for (; NULL != p && NULL == p->ch[c]; p = p->f) ;
                if (p == NULL) {
                    nl = 0;
                    p = sam.root;
                } else {
                    nl = p->len + 1;
                    p = p->ch[c];
                }
            }
            ans < nl ? ans = nl : 0;
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

4.9.2.3 Dynamic Lover　2012 ACM/ICPC 长春赛区网络赛 D 题

题目链接：HDU 4270

参考自：http://www.acforfun.com/?p=98

题意：初始给一个字符串 str，然后又给出三种操作：

1 读取一个字符串 str2 并加到当前串 str 后；

2 读取一个整数 len，问所有长度为 len 的子串和长度不超过 len 的后缀子串中，最小字典序的子串的起始下标是什么，下标从 1 开始计算。

3 读取一个整数 len，删除当前字符串 str 的后 len 个字符。

输入数据中的 str 和 str2 的总长度不超过哦 100000，每一次询问的长度不会超过 1000，并且所有询问综合的长度不会超过 100000。

分析：因为后缀自动机能识别一个合法的子串，那么考虑在后缀自动机上进行深搜，从根节点开始，每次都往字典序小的字符扩展，假如搜到的当前节点 x 时，搜过的长度为 len，或者搜到末尾节点 last，节点 x 代表的字符在字符串 str 中的位置是 pos，答案就是 pos-len+1；有一点要注意，因为也可能会有长度小于 len 的后缀是字符串的子串，所以，

从 last 的 fail 指针到达的节点，也是字符串 str 的合法后缀，所以在深搜之前，需要先标记一下这些末尾节点。

还有就是删除的操作，在构造后缀自动机的时候，会复制一些节点，表示表示不同的转移状态，那么在删除其中一个节点时，它的复制节点也应该删除，所以可以考虑在节点信息里再加一个域，表示是否删除，这个域是一个指针，复制的节点共用一个指针，删除的时候可以直接标记这个域，表示节点是否被删除。注意删除的时候更新末尾节点 last。

最后就是添加字符串了，直接一个一个字符加入到后缀自动机即可。

C++代码：

```
#include <cstdio>
#include <cstring>

#define noDEBUG

const int MAXN = 200005;

struct SAMNode {
 SAMNode *f, *ch[26];
 int len;
 bool *d;
 int pos; /* 节点在字符串的位置 */
 int v;
#ifdef DEBUG
 char c;
#endif
};
SAMNode *root, *last;
int p_cnt, d_cnt, seq_len, v_id;
SAMNode pool[MAXN*2];
bool is_d[MAXN*2];
char str[MAXN];
SAMNode *seq[MAXN];

bool is_del(SAMNode *x) {
 return x == NULL || *x->d;
}

void append(char ch) {
 int c = ch - 'a';
 SAMNode *p = last, *np = pool + p_cnt++;
 memset(np, 0, sizeof(*np));
 np->pos = np->len = p->len + 1;
 np->d = is_d + d_cnt++;
 *np->d = false;
#ifdef DEBUG
 np->c = ch;
#endif
 seq[seq_len = np->len] = np;
 last = np;
 for (; NULL != p && is_del(p->ch[c]); p = p->f) p->ch[c] = np;
 if (NULL == p) {
     np->f = root;
 } else {
     if (p->ch[c]->len == p->len + 1) {
         np->f = p->ch[c];
     } else {
         SAMNode *q = p->ch[c], *nq = pool + p_cnt++;
         *nq = *q;
         nq->len = p->len + 1;
         q->f = np->f = nq;
         for (; NULL != p && p->ch[c] == q; p = p->f) p->ch[c] = nq;
     }
 }
}
```

```
void del(int len) {
 while (len--) *seq[seq_len--]->d = true;
 last = seq[seq_len];
}

int dfs_len;
int dfs(SAMNode *x, int l) {
#ifdef DEBUG
 if (x->c) putchar(x->c);
#endif
 if (l == dfs_len) return x->pos - l  + 1;
 if (x->v == v_id) return seq_len - l + 1;
 for (int i = 0; i < 26; ++i) {
     if (!is_del(x->ch[i])) {
         return dfs(x->ch[i], l+1);
     }
 }
 return x->len - l + 1;
}

int query(int len) {
#ifdef DEBUG
 printf("%s %d:", str, len);
#endif
 ++v_id;
 for (SAMNode *p = last; p != root; p = p->f) p->v = v_id;
 dfs_len = len;
 int ret = dfs(root, 0);
#ifdef DEBUG
 printf("\n");
#endif
 return ret;
}

int main() {
 while (EOF != scanf("%s", str)) {
     root = last = pool;
     memset(root, 0, sizeof(*root));
     root->d = is_d;
     is_d[0] = false;
     p_cnt = 1;
     d_cnt = 1;
     v_id = 0;
     seq[0] = root;
     for (char *c = str; *c; ++c) append(*c);

     int m, q, len;
     scanf("%d", &m);
     while (m--) {
         scanf("%d", &q);
         if (1 == q) {
             scanf("%s", str);
             for (char *c = str; *c; ++c) append(*c);
         } else if (2 == q) {
             scanf("%d", &len);
             printf("%d\n", query(len));
         } else if (3 == q) {
             scanf("%d", &len);
             del(len);
         } else {
             for(;;);
             fprintf(stderr, "error cmd %d\n", q);
         }
     }
 }
 return 0;
 }
```

4.9.2.4 str2int 2012 ACM/ICPC 天津赛区现场赛 F 题

题目链接：HDU 4436

参考自：陈伟杰的解题报告 http://edward-mj.com/?p=835

题意：给 N 个数字串，每个串的所有子串各自转成一个数字，问这些数字都去重后它们的和模 2012 是多少。

分析：在一个子串后面加一个字符，就能形成另外一个字符串，是不是有点像有向图。如果我们把这些子串的按照这些关系建立一个有向无环图(DAG)，尝试用递推的方法解决。

把所有字符串通过用 10 这个不属于任意一个数字的元素串连在一起，用连接起来的字符串构建后缀自动机。然后把结点按节点表示的子串长度 len 从小到大排序一下，从前往后递推统计答案。

递推的时候记录两个值，到达这个结点的方案数 way 以及到达这个状态的数字之和 sum。

每次接收一个字符 c 转移显然就应该给下一个结点的 sum 加上

$$\sum_{i=1}^{way} (number_i * 10 + c)$$

把这个式子的常数 c 拉出来就是

$$way * c + 10 * \sum_{i=1}^{way} number_i$$

亦即 way * c + 10 * sum。

就这样从前往后推一下就好了，最后把每个结点的 sum 值都加起来就是答案。

推的时候要注意的就只有两点：

1. 根节点不应该接收 0 开头的串，因为这会弄出有前导 0 的串从而导致重复，这个很显然吧。

2. 所有的转移都无视 10 这个分支，因为那本身就是不应该走的，这个也很显然把。

【时间复杂度】

O(N*11 + NlogN)

其中 N 为输入字符总数。后面的 N lg N 是因为我贪方便直接按 val sort 了，这一步用了链表跳一下很容易做到 O(n)的。 11 是字符集大小。

【空间复杂度】

O(N)

C++代码：

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <queue>
using namespace std;

const int MAXN = 110005;
const int MOD = 2012;

struct Node {
 int way, sum;
 int len;
 Node *f, *ch[11];
};
```

```
Node *root, *last;
Node pool[MAXN*2];
int cnt;

void init() {
 root = last = pool;
 memset(root, 0, sizeof(Node));
 cnt = 1;
}
void append(char ch) {
/*添加一个字符到末尾，last 为一个后缀终止态，沿着 last->f 走，经过的都是终止态 */
 int c = ch - '0';
 Node *p = last, *np = pool + cnt++;
 memset(np, 0, sizeof(Node));
 np->len = last->len + 1;
 last = np;
 for (; NULL != p && NULL == p->ch[c]; p = p->f) p->ch[c] = np;
 if (p == NULL) {
     np->f = root;
 } else {
     if (p->ch[c]->len == p->len+1) {
         np->f = p->ch[c];
     } else { /* 新建一个节点，避免接受不存在的后缀 */
         Node *q = p->ch[c], *nq = pool + cnt++;
         *nq = *q;
         nq->len = p->len + 1;
         q->f = np->f = nq;
         for (; NULL != p && p->ch[c] == q; p = p->f) p->ch[c] = nq;
     }
 }
}

char str[MAXN];
int idx[MAXN*2];

bool cmp_idx(const int &i, const int &j) {
 if (pool[i].len != pool[j].len) return pool[i].len < pool[j].len;
 return i < j;
}

int main() {
 int n;
 while (EOF != scanf("%d", &n)) {
     init();
     for (int i = 0; i < n; ++i) {
         if (i > 0) append('9'+1);
         scanf("%s", str);
         for (int j = 0; str[j]; ++j) {
             append(str[j]);
         }
     }
     for (int i = 0; i < cnt; ++i) {
         idx[i] = i;
     }
     sort(idx, idx+cnt, cmp_idx);
     int ans = 0;
     for (int i = 1; i <= 9; ++i) {
         if (pool[0].ch[i] == NULL) continue;
         int k = pool[0].ch[i] - pool;
         pool[k].sum += i;
         pool[k].way++;
     }
     for (int id = 1; id < cnt; ++id) {
         int i = idx[id];
         ans += pool[i].sum;
         if (ans >= MOD) ans %= MOD;
         for (int j = 0; j < 10; ++j) {
             if (pool[i].ch[j] == NULL) continue;
             int k = pool[i].ch[j] - pool;
```

```
                pool[k].way += pool[i].way;
                pool[k].sum += pool[i].sum * 10 + pool[i].way * j;
                if (pool[k].sum >= MOD) pool[k].sum %= MOD;
            }
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

4.9.3 其他应用后缀自动机的题目

SPOJ SUBLEX　Lexicographical Substring Search
SPOJ LCS2　Longest Common Substring II

最后要说明的是，这本汇编资料里能讲的后缀自动机内容太少了，一定要在网上找到其他资料进行学习，并且要把汇编资料里的提到的题目都做了，还要把网上推荐的题目也做了。

第5章 搜索

5.1 深度优先搜索

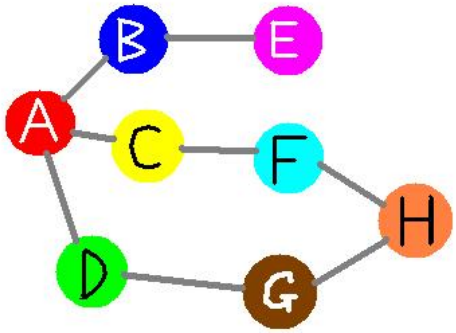
参考文献：

《算法导论》

编写：卢俊达　　校核：黄李龙

深度优先搜索属于图算法的一种，英文缩写为 DFS 即 Depth First Search.其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次.

举例说明之：下图是一个无向图，如果我们从 A 点发起深度优先搜索（以下的访问次序并不是唯一的，第二个点既可以是 B 也可以是 C,D），则我们可能得到如下的一个访问过程：A->B->E（没有路了！回溯到 A)->C->F->H->G->D（没有路，最终回溯到 A,A 也没有未访问的相邻节点，本次搜索结束）.



简要说明深度优先搜索的特点：每次深度优先搜索的结果必然是图的一个连通分量.深度优先搜索可以从多点发起.如果将每个节点在深度优先搜索过程中的"结束时间"排序（具体做法是创建一个 list，然后在每个节点的相邻节点都已被访问的情况下，将该节点加入 list 结尾，然后逆转整个链表)，则我们可以得到所谓的"拓扑排序",即 topological sort.

5.1.1 经典题目 1

1．题目出处/来源

[HrbustOJ][1179][搜索] 下山

2．题目描述

把矩阵想象成鸟瞰一座山，矩阵内的数据可以想象成山的高度。

可以从任意一点开始下山。每一步的都可以朝“上下左右”4 个方向行走，前提是下一步所在的点比当前所在点的数值小。

问题是，对于这种的矩阵，请计算出最长的下山路径。

3．分析

基础深搜题，对于矩阵中的每个元素，先判断它是否访问过，若没访问过，则判断他四周是否其他元素、是否小于它、此点下山的步数相对其他点是否是最大的。最终将结果记录下。

若访问过，则在记录中查找。

记录形式也是矩阵，其元素与已知矩阵一一对应。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为

固定值 12 磅）

```
#include<iostream>
using namespace std;
int a[100][100],used[100][100],n,m;
int dfs(int i,int j)
{
	if(used[i][j])
	return used[i][j];
	int max = 0;
	if(i-1>=0&&a[i][j]>a[i-1][j])
		max = dfs(i-1,j);
	if(i+1<n&&a[i][j]>a[i+1][j]&&max<dfs(i+1,j))
		max = dfs(i+1,j);
	if(j-1>=0&&a[i][j]>a[i][j-1]&&max<dfs(i,j-1))
		max = dfs(i,j-1);
	if(j+1<m&&a[i][j]>a[i][j+1]&&max<dfs(i,j+1))
		max = dfs(i,j+1);
	return used[i][j] = max + 1;
}
int main()
{
	while(cin>>n>>m)
	{
		for(int i=0;i<n;i++)
			for(int j=0;j<m;j++)
			{
				cin>>a[i][j];
				used[i][j] = 0;
			}
		int max = 0;
		for(int i=0;i<n;i++)
			for(int j=0;j<m;j++)
				max = max < dfs(i,j) ? dfs(i,j) : max;
		cout<<max<<endl;
	}
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

5.1.2 经典题目 2

1．题目出处/来源

[USACO][Section 1.4][搜索] The Clocks

2．题目描述

给出 9 个表盘和 9 种表针操作方法，要求找出一个能使所有表针都指向 12 点的方案。

3．分析

用深搜遍历所有可能方案即可，具体思路见代码。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
/*
ID: lujunda1
LANG: C++
PROG: clocks
*/
#include<iostream>
#include<stdio.h>
#include<queue>
#include<cstring>
#include<map>
using namespace std;
struct state
//定义结构体来表示当前时钟的状态。mark[]记录各方法使用的数量。
```

```
{
    int clock[3][3];
    int mark[9];
};
bool check(state temp)
//通过 temp 的 clock[]判断 temp 状态下的表针是否全部指向 12 点。
{
    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
            if(temp.clock[i][j]%12)
                return false;
    return true;
}
void output(state temp)
//输出 temp 状态下的方法序列。
{
    bool check=true;
    for(int i=0;i<9;i++)
        for(int j=0;j<temp.mark[i];j++)
            if(check)
            {
                printf("%d",i+1);
                check=false;
            }
            else
                printf(" %d",i+1);
    printf("\n");
}
int operation[9][9]=
//operation[][]是 9 种方法的具体定义。
{
    {1,1,0,1,1,0,0,0,0},
    {1,1,1,0,0,0,0,0,0},
    {0,1,1,0,1,1,0,0,0},
    {1,0,0,1,0,0,1,0,0},
    {0,1,0,1,1,1,0,1,0},
    {0,0,1,0,0,1,0,0,1},
    {0,0,0,1,1,0,1,1,0},
    {0,0,0,0,0,0,1,1,1},
    {0,0,0,0,1,1,0,1,1}
};
state op(state temp,int n)
//op()方法的含义是，计算 temp 状态在 i 方法作用后的结果并返回。
{
    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
            if(operation[n][i*3+j])
                temp.clock[i][j]+=3;
    temp.mark[n]++;
    return temp;
}
int pos(state temp)
//这实际是为避免重复运算而定义的步骤。
//当方法序列为 111222334，保证其递归下的序列为 1112223344 而不是 1112223334。
//因为 1112223334 在 111222334 之前就已经出现过。
{
    for(int i=8;i>=0;i--)
    //返回使用次数不为 0 且序号最大的方法的序号。
        if(temp.mark[i]>0)
            return i;
    return 0;
}
bool dfs(state temp)
//深搜
{
    if(check(temp))
    //优先检查 temp 状态
```

```
        {
            output(temp);
            return true;
        }
        for(int i=pos(temp);i<9;i++)
            if(temp.mark[i]<3&&dfs(op(temp,i)))
            //temp.mark[i]<3 保证了每个方法使用不超过 4 次，超过 4 次无意义（指针转了超过一圈）。
                return true;
        return false;
    }
    int main()
    {
        freopen("clocks.in","r",stdin);
        freopen("clocks.out","w",stdout);
        state original;
        for(int i=0;i<3;i++)
            for(int j=0;j<3;j++)
                scanf("%d",&original.clock[i][j]);
        for(int i=0;i<9;i++)
            original.mark[i]=0;
        dfs(original);
    }
```

5.1.3 经典题目 3

1．题目出处/来源

[USACO][Section 1.3][搜索] Prime Cryptarithm

2．题目描述

题目给出一个由“*”构成的乘法式子。已知一个范围为[0,9]的整数集合，用集合中的数字替换“*”可能使式子成立，要求编程求出使式子成立的方案的数量。

3．分析

用深搜遍历所有可能方案即可，具体思路见代码注释。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
/*
ID: lujunda1
LANG: C++
PROG: crypt1
*/
#include<iostream>
#include<stdio.h>
#include<cstring>
#include<cstdio>
#include<stdlib.h>
using namespace std;
//num[]用于存储两个乘数，例如 123*45，那么 num[]={1,2,3,4,5}。
int num[5];
//mark[]用于判断某数字是否属于所给集合。
bool mark[10];
//输入
void input()
{
    int n;
    scanf("%d",&n);
    for(int i=0;i<10;i++)
        mark[i]=false;
    for(int i=0,temp;i<n;i++)
    {
        scanf("%d",&temp);
        mark[temp]=true;
    }
}
```

```
//判断整数 n 的各位数字是否都属于集合
int check_num(int n)
{
    //length 用于记录整数 n 的长度。
    int length=0;
    for(;n;n/=10,length++)
        //若某位数字不属于集合，则返回-1。
        if(!mark[n%10])
            return -1;
    //若整数 n 的各位数字都属于集合，则返回其长度。
    //返回长度是因为题目所给乘法式子对不同位置的整数有格式（长度）限制，要判断整数 n 是否符合格式。
    return length;
}
//判断当前 num[]所表示的乘数组合是否符合乘法式子与格式的要求。
bool check()
{
    //a 是被乘数，b 是乘数，d 是乘数的个位，c 是乘数的十位。
    int a=num[0]*100+num[1]*10+num[2],b=num[3]*10+num[4],c=num[4],d=num[3];
    //a*c 和 a*d 分别是乘法式子第 4 行和第 3 行上的整数。它们的长度为 3。
    //a*b 是结果，长度应该为 4。
    if(check_num(a*c)==3&&check_num(a*d)==3&&check_num(a*b)==4)
        return true;
    return false;
}
//利用深搜，遍历所有可能的乘数组合。
//order 表示 num[]中的第 order 位，从 0 开始。
int dfs(int order)
{
    if(order==5)
        return check()?1:0;
    int sum=0;
    for(int i=0;i<10;i++)
        if(mark[i])
        {
            //不能有前导 0
            if((order==0||order==3)&&i==0)
                continue;
            num[order]=i;
            sum+=dfs(order+1);
        }
    return sum;
}
int main()
{
    freopen("crypt1.in","r",stdin);
    freopen("crypt1.out","w",stdout);
    input();
    printf("%d\n",dfs(0));
}
```

5.1.4 经典题目 4

1．题目出处/来源

[USACO][Section 1.4][搜索] Mother’s Milk

2．题目描述

给出三个容量分别为 A、B、C 的桶。开始时只有容量为 C 的桶中有奶且装满，可以将一个桶中的奶倒入另一个桶，问当容量为 A 的桶为空时，容量为 C 的桶内可能有多少奶。

3．分析

通过深搜遍历所有状态，具体思路见代码注释。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
/*
ID: lujunda1
LANG: C++
PROG: milk3
*/
#include<iostream>
#include<stdio.h>
using namespace std;
int cap[3];
//三个桶的容量。
bool map[21][21][21],res[21];
//map[]用来记录状态。
//res[]记录当 A 桶空时，C 桶内的牛奶的数量。
void pour(int i,int j,int state[])
//将第 i 桶倒入第 j 桶中。
//state[]是表示各桶内牛奶的数量。
{
	if(cap[j]-state[j]>=state[i])
	//i 桶可以全部倒入 j 桶
	{
		state[j]+=state[i];
		state[i]=0;
	}
	else
	//i 桶只能部分倒入 j 桶
	{
		state[i]-=(cap[j]-state[j]);
		state[j]=cap[j];
	}
}
void dfs(int a,int b,int c)
//使用深搜遍历所有状态。
{
	map[a][b]1=true;
	if(a==0)
		res1=true;
	for(int i=0;i<3;i++)
		for(int j=0;j<3;j++)
			if(i!=j)
			{
				int state[3]={a,b,c};
				pour(i,j,state);
				if(!map[state[0]][state[1]][state[2]])
					dfs(state[0],state[1],state[2]);
			}
}
void restore()
//对 res[]和 map[]进行初始化。
{
	for(int i=0;i<21;i++)
	{
		res[i]=false;
		for(int j=0;j<21;j++)
			for(int k=0;k<21;k++)
				map[i][j][k]=false;
	}
}
void output()
//根据 res[]输出结果。
{
	bool check=false;
	for(int i=0;i<=cap[2];i++)
	{
		if(res[i]&&!check)
		{
			printf("%d",i);
			check=true;
```

```
                }
                else if(res[i])
                        printf(" %d",i);
        }
        printf("\n");
}
int main()
{
        freopen("milk3.in","r",stdin);
        freopen("milk3.out","w",stdout);
        for(int i=0;i<3;i++)
                scanf("%d",&cap[i]);
        restore();
        dfs(0,0,cap[2]);
        output();
}
```

5.2 广度优先搜索

参考文献：

《算法导论》

编写：卢俊达　　校核：黄李龙

5.2.1 基本原理

已知图 G=(V,E)和一个源顶点 s，广度优先搜索以一种系统的方式探寻 G 的边，从而“发现”s 所能到达的所有顶点，并计算 s 到所有这些顶点的距离(最少边数)，该算法同时能生成一棵根为 s 且包括所有可达顶点的广度优先树。对从 s 可达的任意顶点 v，广度优先树中从 s 到 v 的路径对应于图 G 中从 s 到 v 的最短路径，即包含最小边数的路径。该算法对有向图和无向图同样适用。

之所以称之为广度优先算法，是因为算法自始至终一直通过已找到和未找到顶点之间的边界向外扩展，就是说，算法首先搜索和 s 距离为 k 的所有顶点，然后再去搜索和 S 距离为 k+l 的其他顶点。

为了保持搜索的轨迹，广度优先搜索为每个顶点着色：白色、灰色或黑色。算法开始前所有顶点都是白色，随着搜索的进行，各顶点会逐渐变成灰色，然后成为黑色。在搜索中第一次碰到一顶点时，我们说该顶点被发现，此时该顶点变为非白色顶点。因此，灰色和黑色顶点都已被发现，但是，广度优先搜索算法对它们加以区分以保证搜索以广度优先的方式执行。若(u,v)∈E 且顶点 u 为黑色，那么顶点 v 要么是灰色，要么是黑色，就是说，所有和黑色顶点邻接的顶点都已被发现。灰色顶点可以与一些白色顶点相邻接，它们代表着已找到和未找到顶点之间的边界。

在广度优先搜索过程中建立了一棵广度优先树，起始时只包含根节点，即源顶点 s.在扫描已发现顶点 u 的邻接表的过程中每发现一个白色顶点 v，该顶点 v 及边(u,v)就被添加到树中。在广度优先树中，我们称结点 u 是结点 v 的先辈或父母结点。因为一个结点至多只能被发现一次，因此它最多只能有--个父母结点。相对根结点来说祖先和后裔关系的定义和通常一样：如果 u 处于树中从根 s 到结点 v 的路径中，那么 u 称为 v 的祖先，v 是 u 的后裔。

5.2.2 经典题目

5.2.2.1 题目 1

1．题目出处/来源

[POJ][3278][搜索] Catch That Cow

2．题目描述

一头牛在水平坐标轴上的坐标 K 处，农夫在坐标 N 处。

农夫走一步可以向前移动一个单位或向后移动一个单位已经移动到 2*X 处（X 为当前坐标），为从 N 到 K 至少需要走几步。

3．分析

广搜模板题。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<queue>
#include<cstring>
using namespace std;
bool used[200001];//判断点是否被访问过
int BFS(int n,int m)
{
    memset(used,false,sizeof(used));
    int s=0;//记录步数
    queue<int> a;
    a.push(n);//将起始点 n 压入队列
    used[n]=true;
    while(true)
    {
        int temp=a.size();//a.size()返回队列元素个数
        while(temp--)
        {
            int head=a.front();//将对头元素赋值给 head
            if(head==m)
                return s;
            if(head-1>=0&&!used[head-1])
            {
                a.push(head-1);
                used[head-1]=true;
            }
            if(head+1<200001&&!used[head+1])
            {
                a.push(head+1);
                used[head+1]=true;
            }
            if(head*2<200001&&!used[head*2])
            {
                a.push(head*2);
                used[head*2]=true;
            }
            a.pop();//出队
        }
        s++;
    }
}
int main()
{
    int n,m;
    while(cin>>n>>m)
    cout<<BFS(n,m)<<endl;
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

5.2.2.2 题目 2

1．题目出处/来源

[USACO][Section 1.5][搜索] Superprime Rib

2．题目描述

给出一个整数 n，要求输出所有 n 位超级质数。超级质数的定义为，一个“依次去掉

最右边一位后依然为质数”的质数。

3．分析

一个 n 位质数去掉最右边一位依然为质数，说明其一定从 n-1 位质数演变而来。通过广搜按位数从小到大的顺序搜索到第 n 位即可。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
/*
ID: lujunda1
LANG: C++
PROG: sprime
*/
#include<iostream>
#include<stdio.h>
#include<cmath>
#include<queue>
using namespace std;
bool prime(int n)
//判断参数 n 是否为素数。
{
    if(n<2)
        return false;
    for(int i=2;i<=sqrt(double(n));i++)
        if(n%i==0)
            return false;
    return true;
}
int bfs(int n)
//搜索 n 位超级质数。
{
    queue<int> q;
    q.push(0);
    int temp=0;
    while(temp++<n)
    {
        int size=q.size();
        while(size--)
        {
            int head=q.front();
            q.pop();
            if(n==1)
                printf("2\n");
            if(temp==1&&n!=1)
                q.push(2);
            for(int i=1,son;i<10;i+=2)
            //除 2 以外，所有质数都是奇数，所以只搜奇数并对 2 做特殊处理即可。
            {
                son=head*10+i;
                if(prime(son))
                    if(temp==n)
                    //若是 n 位数便直接输出。
                        printf("%d\n",son);
                    else
                    //不是 n 位数便压入队列。
                        q.push(son);
            }
        }
    }
}
int main()
{
    freopen("sprime.in","r",stdin);
    freopen("sprime.out","w",stdout);
    int n;
    scanf("%d",&n);
    bfs(n);
```

```
}
```

5.3 分支定界法

扩展阅读：

红脸书生的博客：http://www.cnblogs.com/steven_oyj/archive/2010/05/22/1741378.html

编写：卢俊达　　校核：黄李龙

5.3.1 基本原理

类似于回溯法，也是一种在问题的解空间树 T 上搜索问题解的算法。但在一般情况下，分支限界法与回溯法的求解目标不同。回溯法的求解目标是找出 T 中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

由于求解目标不同，导致分支限界法与回溯法在解空间树 T 上的搜索方式也不相同。回溯法以深度优先的方式搜索解空间树 T，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树 T。

分支限界法的一般过程：

分支限界法的搜索策略是：在扩展结点处，先生成其所有的儿子结点（分支），然后再从当前的活结点表中选择下一个扩展对点。为了有效地选择下一扩展结点，以加速搜索的进程，在每一活结点处，计算一个函数值（限界），并根据这些已计算出的函数值，从当前活结点表中选择一个最有利的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。

分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。问题的解空间树是表示问题解空间的一棵有序树，常见的有子集树和排列树。在搜索问题的解空间树时，分支限界法与回溯法对当前扩展结点所使用的扩展方式不同。在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，那些导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被子加入活结点表中。此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所求的解或活结点表为空时为止。

5.3.2 经典题目

1．题目出处/来源

[POJ][1451][字典树] T9

2．题目描述

模拟手机 T9 输入法。先给出 n 个单词和其出现概率，根据用户按下的数字键，选择相应概率最大的单词输出。

3．分析

首先根据给出的单词建树，然后根据按下的数字进行广搜。

广搜的时候是根据按键顺序与字典树中的单词做比较，若字典树中的某个单词或其前缀符合当前按键顺序，则将其最后一个字母压入队列中。通过与字典树比较达到分支定界的目的。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
```

```
#include<stdio.h>
#include<queue>
using namespace std;
struct trie
//利用结构体来封装字典树的节点。
{
	trie* next[26];
	trie* pre;
	//指针指向父节点，用于回溯输出单词。
	int total;
	//total 用于记录概率。
	char letter;
	trie()
	//构造函数。
	{
		for(int i=0;i<26;i++)
			next[i]=NULL;
		pre=NULL;
		total=0;
		letter='\0';
	}
};
void insert(char* s,int n,trie* root)
//将字符串 s 所表示的单词插入到字典树中。
{
	trie *p=root;
	for(int i=0;s[i]!='\0';i++)
	{
		if(p->next[s[i]-'a']==NULL)
		{
			p->next[s[i]-'a']=new trie;
			p->next[s[i]-'a']->pre=p;
			p->next[s[i]-'a']->letter=s[i];
		}
		p=p->next[s[i]-'a'];
		p->total+=n;
	}
}
void output(trie* point)
//从 point 开始回溯至 root，输出途径节点的字母，达到输出单词的目的。
{
	if(point->pre->pre!=NULL)
		output(point->pre);
	printf("%c",point->letter);
}
char T9[10][5]={"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
void bfs(char* num,trie* root)
//根据按键广搜
{
	queue<trie*> q;
	q.push(root);
	for(int i=0;num[i]!='1';i++)
	{
		int size=q.size(),max=0;
		//max 记录最大概率。
		trie* now=NULL;
		//now 指向表示概率最大的词的最后一个字母的节点。
		while(size--)
		{
			trie *head=q.front();
			q.pop();
			for(int j=0;j<strlen(T9[num[i]-'0']);j++)
			//遍历 num[i]所表示的几个字母。
			{
				trie *temp=head->next[T9[num[i]-'0'][j]-'a'];
				if(temp!=NULL)
				//若树中存在此单词
```

```
                        {
                                if(temp->total>max)
                                {
                                        max=temp->total;
                                        now=temp;
                                }
                                q.push(temp);
                        }
                }
        }
        if(now==NULL)
        //now==NULL 说明没有单词对应现在的按键序列
                printf("MANUALLY");
        else
                output(now);
        printf("\n");
    }
    printf("\n");
}
int main()
{
    int t;
    scanf("%d",&t);
    for(int i=1;i<=t;i++)
    {
        printf("Scenario #%d:\n",i);
        trie root;
        //字典树的根节点。
        int w,m;
        for(scanf("%d",&w);w>0;w--)
        {
                char str[101];
                int p;
                scanf("%s%d",str,&p);
                insert(str,p,&root);
        }
        for(scanf("%d",&m);m>0;m--)
        {
                char num[101];
                scanf("%s",num);
                bfs(num,&root);
                //对每个按键序列进行广搜。
        }
        printf("\n");
    }
}
```