華中科技大学

课 程 报 告

课程名称： **算法实践**

专业班级：**ACM1501**

学　　号：**U201514582**

姓　　名：朱锦辉

指导教师：王多强

报告日期：**12/24/2017**

计算机科学与技术学院

目录

并查集

一、 涉及问题

并查集，在一些有 N 个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。这一类问题近几年来反复出现在信息学的国际国内赛题中，其特点是看似并不复杂，但数据量极大，若用正常的数据结构来描述的话，往往在空间上过大，计算机无法承受；即使在空间上勉强通过，运行的时间复杂度也极高，根本就不可能在比赛规定的运行时间（1～3 秒）内计算出试题需要的结果，只能用并查集来描述。

并查集是一种树型的数据结构，用于处理一些不相交集合（Disjoint Sets）的合并及查询问题。常常在使用中以森林来表示。

并查集，实际上是常常被用来解决动态连通性一类问题的一种算法。

我们看一张图来了解一下什么是动态连通性：

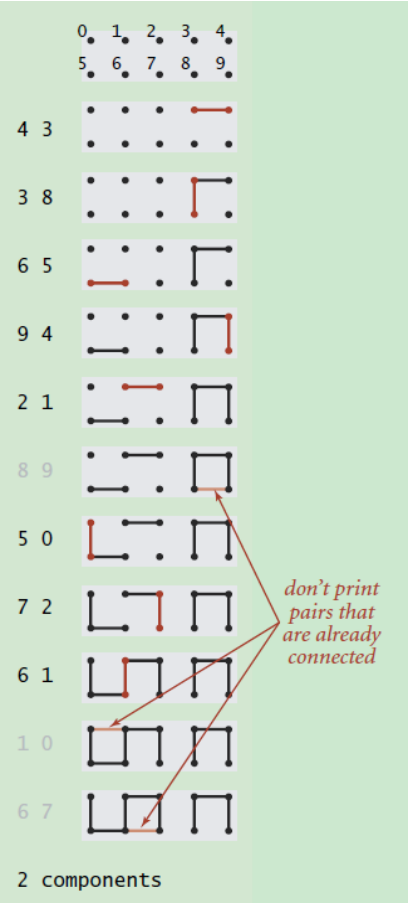


图 1

假设我们输入了一组整数对，即上图中的(4, 3) (3, 8)等等，每对整数代表这两个 points/sites 是连通的。那么随着数据的不断输入，整个图的连通性也会发生变化，从上图中

可以很清晰的发现这一点。同时，对于已经处于连通状态的 points/sites，直接忽略，比如上图中的(8, 9)。

动态连通性的应用场景：

1. 网络连接判断：

如果每个 pair 中的两个整数分别代表一个网络节点，那么该 pair 就是用来表示这两个节点是需要连通的。那么为所有的 pairs 建立了动态连通图后，就能够尽可能少的减少布线的需要，因为已经连通的两个节点会被直接忽略掉。

变量名等同性(类似于指针的概念)：

在程序中，可以声明多个引用来指向同一对象，这个时候就可以通过为程序中声明的引用和实际对象建立动态连通图来判断哪些引用实际上是指向同一对象。

对问题建模：在对问题进行建模的时候，我们应该尽量想清楚需要解决的问题是什么。因为模型中选择的数据结构和算法显然会根据问题的不同而不同，就动态连通性这个场景而言，我们需要解决的问题可能是：

- 给出两个节点，判断它们是否连通，如果连通，不需要给出具体的路径
- 出两个节点，判断它们是否连通，如果连通，需要给出具体的路径

二、 算法介绍

联合-查找算法（union-find algorithm）定义了两个用于此数据结构的操作：

- Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- Union：将两个子集合并成同一个集合。

由于支持这两种操作，一个不相交集也常被称为联合-查找数据结构（union-find data structure）或合并-查找集合（merge-find set）。其他的重要方法，MakeSet，用于建立单元素集合。有了这些方法，许多经典的划分问题可以被解决。

为了更加精确的定义这些方法，需要定义如何表示集合。一种常用的策略是为每个集合选定一个固定的元素，称为代表，以表示整个集合。接着，Find(x) 返回 x 所属集合的代表，而 Union 使用两个集合的代表作为参数。

并查集森林是一种将每一个集合以树表示的数据结构，其中每一个节点保存着到它的父节点的引用（见意大利面条堆栈）。这个数据结构最早由 Bernard A. Galler 和 Michael J. Fischer 于 1964 年提出，但是经过了数年才完成了精确的分析。

在并查集森林中，每个集合的代表即是集合的根节点。“查找”根据其父节点的引用向根行进直到到底树根。“联合”将两棵树合并到一起，这通过将一棵树的根连接到另一棵树的根。实现这样操作的一种方法是

```
function MakeSet(x)
    x.parent := x
function Find(x)
    if x.parent == x
        return x
    else
        return Find(x.parent)
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    xRoot.parent := yRoot
```

这是并查集森林的最基础的表示方法，这个方法不会比链表法好，这是因为创建的树可能会严重不平衡；然而，可以用两种办法优化。

1. 按秩合并

第一种方法，称为“按秩合并”，即总是将更小的树连接至更大的树上。因为影响运行时间的是树的深度，更小的树添加到更深的树的根上将不会增加秩除非它们的秩相同。在这个算法中，术语“秩”替代了“深度”，因为同时应用了路径压缩时（见下文）秩将不会与高度相同。单元素的树的秩定义为 0，当两棵秩同为 r 的树联合时，它们的秩 r+1。只使用这个方法将使最坏的运行时间提高至每个 MakeSet、Union 或 Find 操作 O(log n)}。优化后的 MakeSet 和 Union 伪代码：

```
function MakeSet(x)
    x.parent := x
    x.rank   := 0
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    if xRoot == yRoot
        return

    // x 和 y 不在同一个集合，合并它们。
    if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
    else if xRoot.rank > yRoot.rank
        yRoot.parent := xRoot
    else
        yRoot.parent := xRoot
        xRoot.rank := xRoot.rank + 1
```

2. 路径压缩

第二个优化，称为“路径压缩”，是一种在执行“查找”时扁平化树结构的方法。关键在于在路径上的每个节点都可以直接连接到根上；他们都有同样的表示方法。为了达到这样的效果，Find 递归地经过树，改变每一个节点的引用到根节点。得到的树将更加扁平，为以后直接或者间接引用节点的操作加速。这儿是 Find：

```
function Find(x)
    if x.parent != x
        x.parent := Find(x.parent)
    return x.parent
```

同时使用路径压缩、按秩合（rank）并优化的程序每个操作的平均时间仅为$O(\alpha(n))$，其中$O(\alpha(n))$是$n = f(x) = A(x,x)$的反函数，A 是急速增加的阿克曼函数。因为$\alpha(n)$是其反函数，故$\alpha(n)$在 n 十分巨大时还是小于 5。因此，平均运行时间是一个极小的常数。

实际上，这是渐近最优算法：Fredman 和 Saks 在 1989 年解释了Ω(α(n))的平均时间内可以获得任何并查集。

3. 主要操作

需要注意的是，一开始我们假设元素都是分别属于一个独立的集合里的。

1）合并两个不相交集合

操作很简单：先设置一个数组(阵列)Father[x]，表示 x 的“父亲”的编号。 那么，合并两个不相交集合的方法就是，找到其中一个集合最父亲的父亲（也就是最久远的祖先），将另外一个集合的最久远的祖先的父亲指向它。

C 语言代码表示形式:

```
void Union(int x,int y)
{
      fx = getfather(x);
      fy = getfather(y);
      if(fy!=fx)
          father[fx]=fy;
}
```

2）判断两个元素是否属于同一集合

仍然使用上面的数组。则本操作即可转换为寻找两个元素的最久远祖先是否相同。寻找祖先可以采用递归实现，见后面的路径压缩算法。

C 代码：

```
bool same(int x,int y)
{
    return getfather(x)==getfather(y);
}
/*返回 true  表示相同根结点，返回 false 不相同*/
```

3） 并查集的优化

路径压缩

刚才我们说过，寻找祖先时采用递归，但是一旦元素一多起来，或退化成一条链，每次 GetFather 都将会使用 O(n)的复杂度，这显然不是我们想要的。

对此，我们必须要进行路径压缩，即我们找到最久远的祖先时“顺便”把它的子孙直接连接到它上面。这就是路径压缩了。使用路径压缩的代码如下：

C 语言的实现：

```
int getfather(int v)
{
    if (father[v]==v)
       return v;
    else
    {
        father[v]=getfather(father[v]);//路径压缩
        return father[v];
     }
}
```

Rank 合并

合并时将元素所在深度小的集合合并到元素所在深度大的集合。

C 语言的实现：

```
void judge(int x ,int y)

{
    fx = getfather(x);
    fy = getfather(y);

    if (rank[fx]>rank[fy])
        father[fy] = fx;
    else
    {
        father[fx] = fy;
        if(rank[fx]==rank[fy])
            ++rank[fy]; //重要的是祖先的 rank，所以只用修改祖先的 rank 就可以了，
子节点的 rank 不用管
    }
}
```

初始化：

```
memset(rank,0,sizeof(rank));
```

4. 时间及空间复杂度

时间复杂度

同时使用路径压缩、按秩合（rank）并优化的程序每个操作的平均时间仅为$O(\alpha(n))$，其中$O(\alpha(n))$是$n = f(x) = A(x,x)$的反函数，A 是急速增加的阿克曼函数。因为$\alpha(n)$是其反函数，故$\alpha(n)$在 n 十分巨大时还是小于 5。因此，平均运行时间是一个极小的常数。实际上，这是

渐近最优算法：Fredman 和 Saks 在 1989 年解释了Ω(α(n))的平均时间内可以获得任何并查集。

空间复杂度

O(n)（n 为元素数量）。

三、 解题报告

1. POJ 1417

1）问题描述

一座岛上有 p1 个好人，p2 个坏人，好人永远说真话，坏人永远说假话。现在给定 n 个陈述如下：

- x y yes，即 x 说 y 是好人
- x y no，即 x 说 y 是坏人

如果根据这 n 个陈述能判定出哪些人是好人，按正序输出所有好人的序号，否则输出 no。

输入格式

n p1 p2
xl yl a1
x2 y2 a2
…
xi yi ai
…
xn yn an

输出格式

no
no
x1
x2
end
x1
x2
x3

x4
end

样例输入

2 1 1
1 2 no
2 1 no
3 2 1
1 1 yes
2 2 yes
3 3 yes
2 2 1
1 2 yes
2 3 no
5 4 3
1 2 yes
1 3 no
4 5 yes
5 6 yes
6 7 no
0 0 0

样例输出

no
no
1
2
end
3
4
5
6
end

2）算法设计

这题是并查集在种类问题上的一个应用。

用 r[i]表示 i 与它的根节点的关系，0 代表是同一个群体，1 代表不是一个群体。利用 Union 根据不同的陈述更新节点之间的关系。利用所有的陈述更新完毕后，将所有的节点分成不同的种类，如果在一棵并查集树中好人数和坏人数相等，则直接输出 no。否则利用动

态规划，dp(i, j)表示前 i 个人中好人数为 j 的方案数。如果最终 dp(p1+p2， p1)不等于 1 则输出 no，否则根据 dp 中的信息输出所有好人的序号。

3）程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_P 600

using namespace std;

int n, p1, p2;
int p[MAX_P], ranks[MAX_P], r[MAX_P];
int a[MAX_P], total[MAX_P];
int cat[MAX_P], pa[MAX_P];
bool flag;
int dp[MAX_P][MAX_P];
int divine[MAX_P];

void makeSets()
{
	for(int i = 1; i <= p1 + p2; ++i)
	{
		p[i] = i;
		ranks[i] = 0;
		r[i] = 0;
	}
}

int find(int x)
{
	if(p[x] != x)
	{
		int t = p[x];
		p[x] = find(p[x]);
		r[x] = (r[x] + r[t]) % 2;
	}
	return p[x];
}

void Union(int x, int y, int ans)
```

```
{
	int xp = find(x);
	int yp = find(y);
	if(ranks[xp] < ranks[yp])
	{
		p[xp] = yp;
		r[xp] = (r[x] + ans + r[y]) % 2;
	}
	else
	{
		p[yp] = xp;
		r[yp] = (r[y] + ans + r[x]) % 2;
		if(ranks[xp] == ranks[yp]) ranks[xp]++;
	}
}
int main(int argc, char const *argv[])
{
	while(1)
	{
		scanf("%d%d%d", &n, &p1, &p2);
		if(n == 0 && p1 == 0 && p2 == 0) break;
		memset(p, 0, sizeof(p));
		memset(ranks, 0, sizeof(ranks));
		memset(r, 0, sizeof(r));
		memset(a, 0, sizeof(a));
		memset(total, 0, sizeof(total));
		memset(cat, 0, sizeof(cat));
		memset(pa, 0, sizeof(pa));
		memset(dp, 0, sizeof(dp));
		memset(divine, 0, sizeof(divine));
		makeSets();
		int m = p1 + p2;
		flag = true;
		for(int i = 0; i < n; ++i)
		{
			char str[4];
			int x, y, ans;
			scanf("%d%d%s", &x, &y, str);
			ans = (str[0] == 'n');
			if(find(x) == find(y)) continue;
			Union(x, y, ans);
		}
		for(int i = 1; i <= m; ++i) find(i);
		int cnt = 0;
```

```
for(int i = 1; i <= m; ++i)
{
	int p = find(i);
	if(cat[p] == 0)
	{
		cat[p] = ++cnt;
		pa[cnt] = p;
	}
	total[cat[p]]++;
	if(r[i] == 0) a[cat[p]]++;
}
dp[0][0] = 1;
for(int i = 1; i <= cnt; ++i)
{
	int x = a[i], y = total[i] - a[i];
	if(x == y)
	{
		flag = false;
		break;
	}
	for(int j = m; j >= x; --j) dp[i][j] += dp[i - 1][j - x];
	for(int j = m; j >= y; --j) dp[i][j] += dp[i - 1][j - y];
}
if(dp[cnt][p1] != 1 || !flag) printf("no\n");
else
{
	int cur = p1;
	for(int i = cnt; i >= 1; --i)
	{
		int x = a[i], y = total[i] - a[i];
		if(dp[i - 1][cur - x] == 1)
		{
			divine[i] = 0;
			cur = cur - x;
		}
		else
		{
			divine[i] = 1;
			cur = cur - y;
		}
	}
	for(int i = 1; i <= m; ++i)
	{
		int p = find(i);
```

```
                    if(r[i] == divine[cat[p]]) printf("%d\n", i);
                }
                printf("end\n");
            }
        }
        return 0;
}
```

4）性能分析

程序中对 n 个描述依次使用 Union 操作进行处理，其复杂度是 O(nα(n))，设 M=p1+p2，则 dp 的操作复杂度为 O(n・m)，因此算法的时间复杂度为 O(n・m)，空间复杂度为 O(n)。

2. POJ 1733

1）问题描述

Parity game

Time Limit: 1000MS Memory Limit: 65536K

Total Submissions: 10292 Accepted: 3954

Description

Now and then you play the following game with your friend. Your friend writes down a sequence consisting of zeroes and ones. You choose a continuous subsequence (for example the subsequence from the third to the fifth digit inclusively) and ask him, whether this subsequence contains even or odd number of ones. Your friend answers your question and you can ask him about another subsequence and so on. Your task is to guess the entire sequence of numbers.

You suspect some of your friend's answers may not be correct and you want to

convict him of falsehood. Thus you have decided to write a program to help you in this matter. The program will receive a series of your questions together with the answers you have received from your friend. The aim of this program is to find the first answer which is provably wrong, i.e. that there exists a sequence satisfying answers to all the previous questions, but no such sequence satisfies this answer.

Input

The first line of input contains one number, which is the length of the sequence of zeroes and ones. This length is less or equal to 1000000000. In the second line, there is one positive integer which is the number of questions asked and answers to them. The number of questions and answers is less or equal to 5000. The remaining lines specify questions and answers. Each line contains one question and the answer to this question: two integers (the position of the first and last digit in the chosen subsequence) and one word which is either `even' or `odd' (the answer, i.e. the parity of the number of ones in the chosen subsequence, where `even' means an even number of ones and `odd' means an odd number).

Output

There is only one line in output containing one integer X. Number X says that there exists a sequence of zeroes and ones satisfying first X parity conditions, but there exists none satisfying X+1 conditions. If there exists a sequence of zeroes and ones satisfying all the given conditions, then number X should be the number of all thequestions asked.

2)算法设计

由于数据的范围很大，首先需要对输入的数据进行离散化。

由于题目只给了区间 1 个数的奇偶性，我们可以用一个 rel 数组代表从这个结点到它的根结点这个区间中 1 个数的奇偶性，那么当左右端点的根节点相同时，我们就可以开始进行判断，rel[r]^rel[l]就会等于整个区间的奇偶性(0 为偶，1 为奇）。由于是给出整个闭区间 l，r 的 1 奇偶性，所以让 l--。那么最终结果 rel[r]^rel[l]就可以代表整个闭区间的奇偶性。

3)程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>

using namespace std;

const int max_n = 2e4;

int cnt;
int start[max_n], end[max_n], odd[max_n], temp[max_n];
int p[max_n], ranks[max_n], r[max_n];

int findPos(int x)
{
	int s = 0, e = cnt;
	int mid;
	while(s < e)
	{
		mid = (s + e) / 2;
		if(temp[mid] == x) return mid;
		if(temp[mid] < x) s = mid + 1;
		else e = mid;
	}
	return -1;
}

void makeSets()
{
	for(int i = 0; i <= cnt; i++)
	{
		p[i] = i;
```

```
            ranks[i] = 0;
            r[i] = 0;
        }
}

int find(int x)
{
        if(p[x] != x)
        {
            int t = p[x];
            p[x] = find(p[x]);
            r[x] = r[x] ^ r[t];
        }
        return p[x];
}

void Union(int x, int y, int odd)
{
        int x_parent = find(x);
        int y_parent = find(y);
        if(ranks[x_parent] < ranks[y_parent])
        {
            p[x_parent] = y_parent;
            r[x_parent] = r[x] ^ odd ^ r[y];
        }
        else
        {
            p[y_parent] = x_parent;
            r[y_parent] = r[y] ^ odd ^ r[x];
            if(ranks[x_parent] == ranks[y_parent]) ranks[x_parent]++;
        }
}

int main(int argc, char const *argv[])
{
        int len, n, s, e;
        char str[5];
        cnt = 0;
        scanf("%d%d", &len, &n);
        for(int i = 0; i < n; i++)
        {
            scanf("%d%d%s", &s, &e, str);
            start[i] = --s;
            end[i] = e;
```

```
            odd[i] = (str[0] == 'o');
            temp[cnt++] = s;
            temp[cnt++] = e;
        }
        sort(temp, temp + cnt);
        cnt = unique(temp, temp + cnt) - temp;
        makeSets();
        int X = 0;
        for(int i = 0; i < n; i++)
        {
            int left = findPos(start[i]), right = findPos(end[i]);
            //printf("%d %d\n%d %d\n", start[i], end[i], left, right);
            if(find(left) == find(right))
            {
                //printf("%d %d %d\n", r[left], r[right], odd[i]);
                if((r[left] ^ r[right]) != odd[i]) break;
            }
            else Union(left, right, odd[i]);
            X++;
        }
        printf("%d", X);
        return 0;
}
```

4) 性能分析

初始化及输入：O(n)
离散化：O(nlog n)
并查集更新：O(nα(n))
故总的时间复杂度为O(nlog n)，空间复杂度为O(n)。

5) 编程技术技巧

离散化，将数据范围很大的离散数据映射到一个小整数集上，从而节省空间开销。

树状数组

一、 涉及问题

树状数组或二叉索引树（英语：Binary Indexed Tree），又以其发明者命名为 Fenwick 树，最早由 Peter M. Fenwick 于 1994 年以 A New Data Structure for Cumulative Frequency Tables[1] 为题发表在 SOFTWARE PRACTICE AND EXPERIENCE。其初衷是解决数据压缩里的累积频率（Cumulative Frequency）的计算问题，现多用于高效计算数列的前缀和， 区间和。

二、 算法介绍

假设数组 a[1..n]，那么查询 a[1]+...+a[n]的时间是 log 级别的，而且是一个在线的数据结构，支持随时修改某个元素的值，复杂度也为 log 级别。

来观察这个图：

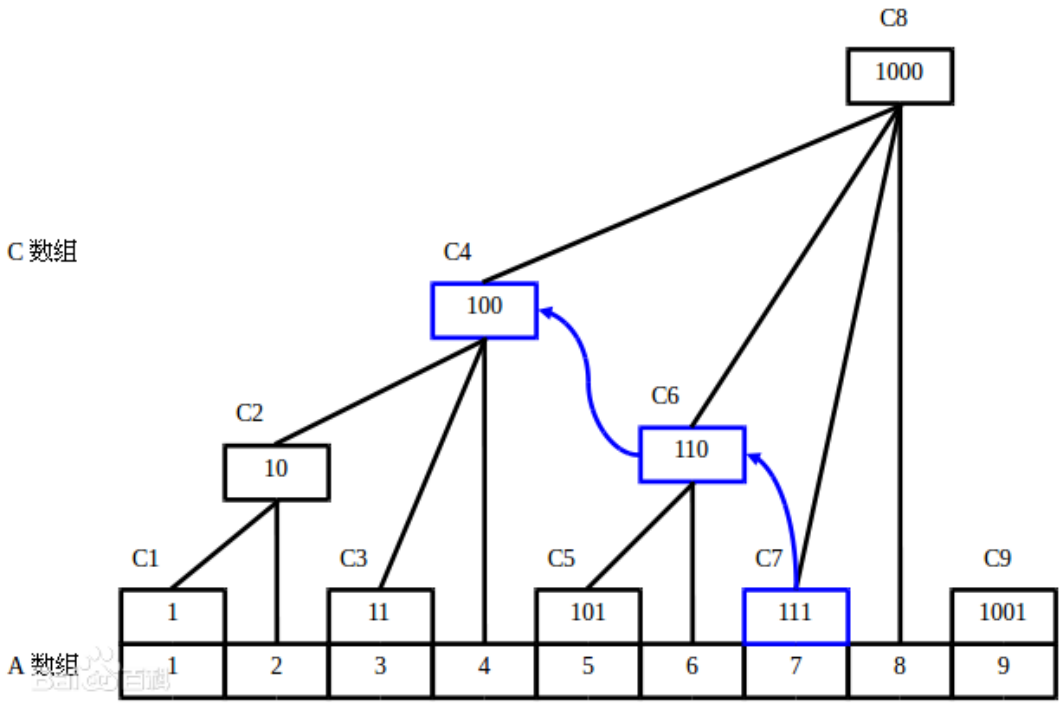


图 2

令这棵树的结点编号为 C1，C2...Cn。令每个结点的值为这棵树的值的总和，那么容易发现：

C1 = A1

C2 = A1 + A2

C3 = A3

C4 = A1 + A2 + A3 + A4

C5 = A5

C6 = A5 + A6

C7 = A7

C8 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8

...

C16 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8 + A9 + A10 + A11 + A12 + A13 + A14 + A15 + A16

这里有一个有趣的性质：

设节点编号为 x，那么这个节点管辖的区间为 2^k（其中 k 为 x 二进制末尾 0 的个数）个元素。因为这个区间最后一个元素必然为 Ax，

所以很明显：Cn = A(n – 2^k + 1) + ... + An

算这个 2^k 有一个快捷的办法，定义一个函数如下即可：

```
1    Int lowbit(int x){
2    returnx&(x^(x–1));
3    }
```

利用机器补码特性，也可以写成：

```
1    Int lowbit(int x){
2    Return x&-x;
3    }
```

当想要查询一个 SUM(n)(求 a[n]的和），可以依据如下算法即可：

step1:令 sum = 0，转第二步；

step2:假如 n <= 0，算法结束，返回 sum 值，否则 sum = sum + Cn，转第三步；

step3: 令 n = n – lowbit(n)，转第二步。

可以看出，这个算法就是将这一个个区间的和全部加起来，为什么是效率是 log(n)的呢？以下给出证明：

n = n – lowbit(n)这一步实际上等价于将 n 的二进制的最后一个 1 减去。而 n 的二进制里最多有 log(n)个 1，所以查询效率是 log(n)的。

那么修改呢，修改一个节点，必须修改其所有祖先，最坏情况下为修改第一个元素，最多有 log(n)的祖先。

所以修改算法如下（给某个结点 i 加上 x）：

step1: 当 i > n 时，算法结束，否则转第二步；

step2: Ci = Ci + x， i = i + lowbit(i)转第一步。

i = i +lowbit(i)这个过程实际上也只是一个把末尾 1 补为 0 的过程。

三、 解题报告

1. POJ 3321

1) 问题描述

Apple Tree

Time Limit: 2000MS Memory Limit: 65536K

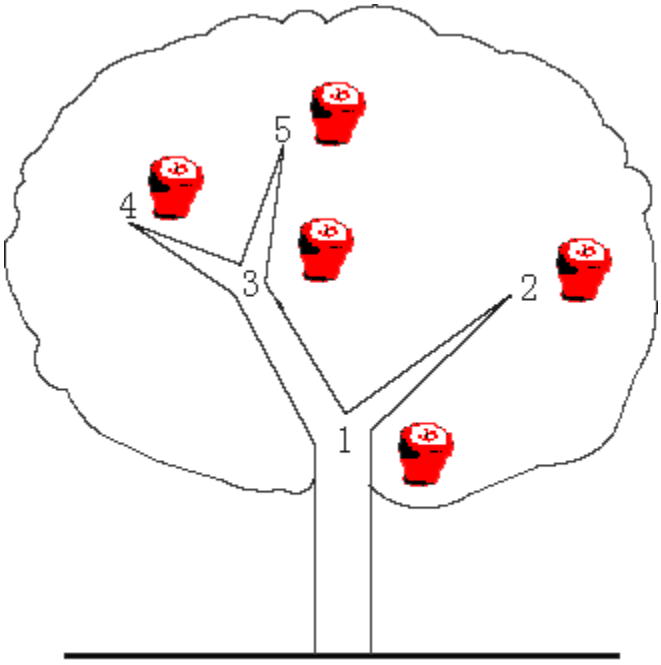
Total Submissions: 32056 Accepted: 9636

Description

There is an apple tree outside of kaka's house. Every autumn, a lot of apples will grow in the tree. Kaka likes apple very much, so he has been carefully nurturing the big apple tree.

The tree has N forks which are connected by branches. Kaka numbers the forks by 1 to N and the root is always numbered by 1. Apples will grow on the forks and two apple won't grow on the same fork. kaka wants to know how many apples are there in a sub-tree, for his study of the produce ability of the apple tree.

The trouble is that a new apple may grow on an empty fork some time and kaka may pick an apple from the tree for his dessert. Can you help kaka?



-

Input

The first line contains an integer N ($N \leq 100,000$) , which is the number of the

forks in the tree.

The following N - 1 lines each contain two integers u and v, which means fork u and fork v are connected by a branch.

The next line contains an integer M (M ≤ 100,000).

The following M lines each contain a message which is either

"C x" which means the existence of the apple on fork x has been changed.

i.e. if there is an apple on the fork, then Kaka pick it; otherwise a new apple has grown on the empty fork.

or

"Q x" which means an inquiry for the number of apples in the sub-tree above the fork x, including the apple (if exists) on the fork x

Note the tree is full of apples at the beginning

Output

For every inquiry, output the correspond answer per line.

2) 算法设计

DFS 遍历整个树，记录每个点遍历的时间戳，即访问的次序，第一次访问的时间戳(用 first 数组记录)和最后一次访问的时间戳（用 last 数组记录）。

这样就将整棵树转化成了一个数组。然后对每一个树枝（即区间的左端点）生成一个苹果，即数组的左端点进行加 1，更新前缀和，并将 picked 标志置否。

对于‘C’，如果 picked 为 true，则将数组对应元素减 1；否则加 1。然后 picked 取反。对于‘Q’，查询树枝对应区间的区间和。

3) 程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
```

```
#define MAX_N 100010
#define MAX_M 100010

using namespace std;

int N, M;
vector<vector<int> > G(MAX_N);
int L[MAX_N], R[MAX_N], bit[MAX_N];
bool picked[MAX_N];

int dfn = 1;
void dfs(int u, int p)
{
	L[u] = dfn++;
	for(int i = 0; i < G[u].size(); ++i)
	{
		int v = G[u][i];
		if(v == p) continue;
		dfs(v, u);
	}
	R[u] = dfn;
}

int lowbit(int i)
{
	return i & (-i);
}

void add(int i, int w)
{
	for(int j = i; j <= N; j += lowbit(j))
		bit[j] += w;
}

int sum(int i)
{
	int ans = 0;
	for(int j = i; j > 0; j -= lowbit(j))
		ans += bit[j];
	return ans;
}

int main(int argc, char const *argv[])
```

```
{
    scanf("%d", &N);
    for(int i = 0; i < N - 1; ++i)
    {
        int u, v;
        scanf("%d%d", &u, &v);
        G[u].push_back(v);
        G[v].push_back(u);
    }

    dfs(1, 0);

    for(int i = 1; i <= N; ++i)
    {
        add(L[i], +1);
        picked[i] = false;
    }

    scanf("%d", &M);
    for(int i = 0; i < M; ++i)
    {
        char cmd[2];
        int x;
        scanf("%s%d", cmd, &x);
        if(cmd[0] == 'C')
        {
            if(picked[x]) add(L[x], +1);
            else add(L[x], -1);
            picked[x] = !picked[x];
        }
        else printf("%d\n", sum(R[x] - 1) - sum(L[x] - 1));
    }
    return 0;
}
```

4) 性能分析

Dfs：O(n)
修改单点值：O(nlog n)
查询前缀和：O(nlog n)
故总的时间复杂度为 O(nlog n)，空间复杂度为 O(n)。

2. POJ 1990

1) 问题描述

MooFest

Time Limit: 1000MS Memory Limit: 30000K

Total Submissions: 8838 Accepted: 3992

Description

Every year, Farmer John's N (1 <= N <= 20,000) cows attend "MooFest",a social gathering of cows from around the world. MooFest involves a variety of events including haybale stacking, fence jumping, pin the tail on the farmer, and of course, mooing. When the cows all stand in line for a particular event, they moo so loudly that the roar is practically deafening. After participating in this event year after year, some of the cows have in fact lost a bit of their hearing.

Each cow i has an associated "hearing" threshold v(i) (in the range 1..20,000). If a cow moos to cow i, she must use a volume of at least v(i) times the distance between the two cows in order to be heard by cow i. If two cows i and j wish to converse, they must speak at a volume level equal to the distance between them times max(v(i),v(j)).

Suppose each of the N cows is standing in a straight line (each cow at some unique x coordinate in the range 1..20,000), and every pair of cows is carrying on a conversation using the smallest possible volume.

Compute the sum of all the volumes produced by all N(N-1)/2 pairs of mooing cows.

Input

* Line 1: A single integer, N

* Lines 2..N+1: Two integers: the volume threshold and x coordinate for a cow. Line 2 represents the first cow; line 3 represents the second cow; and so on. No two cows will stand at the same location.

Output

* Line 1: A single line with a single integer that is the sum of all the volumes of the conversing cows.

2) 算法设计

利用 2 个树状数组，分别统计奶牛 i 的位置 p[i]之前有多少奶牛，和 p[i]之前的奶牛位置和。由权重小的奶牛往权重大的奶牛枚举时，将当前最大权值乘以当前奶牛到剩下的所有奶牛的距离之和，并将这个乘积累加起来，枚举完以后即得答案。

3) 程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <utility>
#include <algorithm>

#define MAX_N 20016
#define MAX_V 20016
#define MAX_X 20016

using namespace std;
typedef long long LL;
```

```
int N;
pair<int, int> cow[MAX_N]; //(v, x)
LL bit_n[MAX_N], bit_x[MAX_N]; //都以奶牛坐标为下标，分别对奶牛数量和坐标计算前缀和

int lowbit(int pos)
{
	return pos & (-pos);
}

void add(LL *bit, int pos, LL w)
{
	for(int i = pos; i <= MAX_X; i += lowbit(i))
		bit[i] += w;
}

LL sum(LL *bit, int pos)
{
	LL ans = 0;
	for(int i = pos; i > 0; i -= lowbit(i))
		ans += bit[i];
	return ans;
}

//重载函数 sum 返回 a[start, end)
LL sum(LL *bit, int start, int end)
{
	return sum(bit, end - 1) - sum(bit, start - 1);
}

int main(int argc, char const *argv[])
{
	scanf("%d", &N);
	for(int i = 0; i < N; ++i)
	{
		scanf("%d%d", &cow[i].first, &cow[i].second);
	}

	sort(cow, cow + N);
	LL total = 0;

	for(int i = 0; i < N; ++i)
	{
```

```
            //v 为当前最大权值
            int v = cow[i].first, x = cow[i].second;
            //x 坐标左边奶牛数量和右边奶牛数量
            LL nleft = sum(bit_n, 1, x), nright = sum(bit_n, x + 1, MAX_X);
            //v 与到剩下所有奶牛的距离之和相乘
            total += v * ((nleft * x - sum(bit_x, 1, x)) + (sum(bit_x, x + 1, MAX_X) - nright * x));
            add(bit_n, x, 1);
            add(bit_x, x, x);
        }
        //！！！ long long 一定要用 lld，否则会 WA！！！
        printf("%lld\n", total);
        return 0;
}
```

4) 性能分析

将奶牛权重排序：O(nlog n)
从小到大枚举所有奶牛的权值：O(nlog n)
总的时间复杂度为 O(nlog n)，空间复杂度为 O(n)。

后缀树

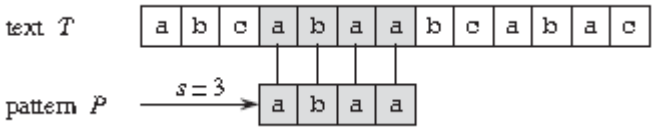
一、 涉及问题

文本（Text）是一个长度为 n 的数组 T[1..n]；

模式（Pattern）是一个长度为 m 且 m≤n 的数组 P[1..m]；

T 和 P 中的元素都属于有限的字母表 Σ 表；

如果 0≤s≤n-m，并且 T[s+1..s+m] = P[1..m]，即对 1≤j≤m，有 T[s+j] = P[j]，则说模式 P 在文本 T 中出现且位移为 s，且称 s 是一个有效位移（Valid Shift）。



比如上图中，目标是找出所有在文本 T = abcabaabcabac 中模式 P = abaa 的所有出现。该模式在此文本中仅出现一次，即在位移 s = 3 处，位移 s = 3 是有效位移。

解决字符串匹配问题的常见算法有：

- 朴素的字符串匹配算法（Naive String Matching Algorithm）
- Knuth-Morris-Pratt 字符串匹配算法（即 KMP 算法）
- Boyer-Moore 字符串匹配算法

字符串匹配算法通常分为两个步骤：预处理（Preprocessing）和匹配（Matching）。所以算法的总运行时间为预处理和匹配的时间的总和。下图描述了常见字符串匹配算法的预处理和匹配时间。

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth Morris Pratt	$\Theta(m)$	$\Theta(n)$

我们知道，上述字符串匹配算法均是通过对模式（Pattern）字符串进行预处理的方式来加快搜索速度。对 Pattern 进行预处理的最优复杂度为 O(m)，其中 m 为 Pattern 字符串的长度。

后缀树即为一种对文本（Text）进行预处理的算法。

后缀树的用途，总结起来大概有如下几种

查找字符串 o 是否在字符串 S 中。

方案：用 S 构造后缀树，按在 trie 中搜索字串的方法搜索 o 即可。

原理：若 o 在 S 中，则 o 必然是 S 的某个后缀的前缀。

例如 S: leconte，查找 o: con 是否在 S 中,则 o(con)必然是 S(leconte)的后缀之一 conte 的前缀.有了这个前提，采用 trie 搜索的方法就不难理解了。

指定字符串 T 在字符串 S 中的重复次数。

方案：用 S+'$'构造后缀树，搜索 T 节点下的叶节点数目即为重复次数

原理：如果 T 在 S 中重复了两次，则 S 应有两个后缀以 T 为前缀，重复次数就自然统计出来了。

字符串 S 中的最长重复子串

方案：原理同 2，具体做法就是找到最深的非叶节点。

这个深是指从 root 所经历过的字符个数，最深非叶节点所经历的字符串起来就是最长重复子串。

为什么要非叶节点呢?因为既然是要重复，当然叶节点个数要>=2。

两个字符串 S1，S2 的最长公共部分

方案：将 S1#S2$作为字符串压入后缀树，找到最深的非叶节点，且该节点的叶节点既有#也有$(无#)。

二、 算法介绍

1. 倍增算法

倍增算法的主要思路是：用倍增的方法对每个字符开始的长度为 2^k 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 2^k 大于 n 以后，每个字符开始的长度为 2^k 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 2^{k-1} 的字符串的 rank 值，那么长度为 2^k 的字符串就可以用两个长度为 2^{k-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 2^k 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 2^k 的字符串的两个关键字。

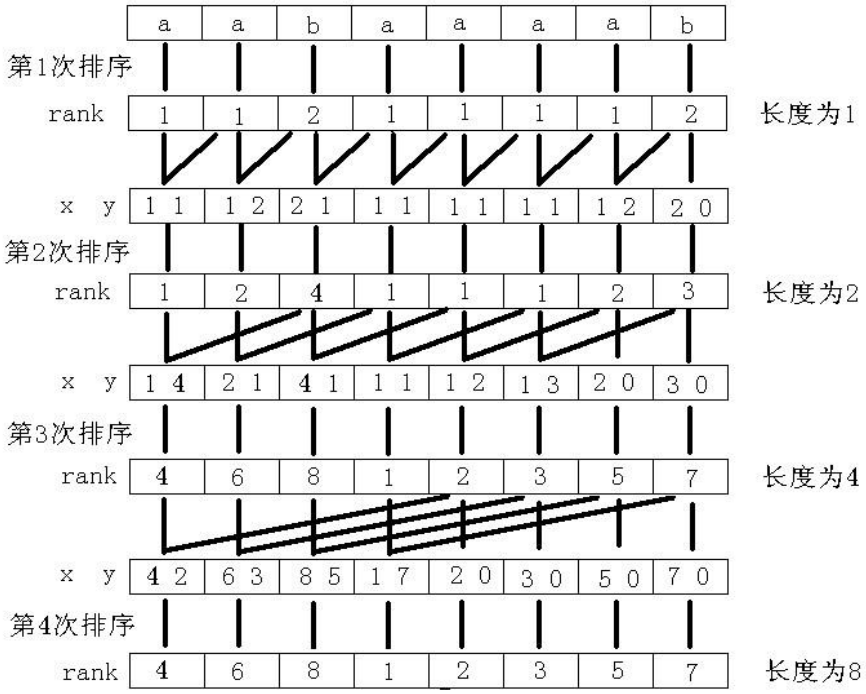


图2

具体实现：

```
int wa[maxn],wb[maxn],wv[maxn],ws[maxn];

int cmp(int *r,int a,int b,int l) {return
r[a]==r[b]&&r[a+l]==r[b+l];} 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++) ws[i]=0; for(i=0;i<n;i++)
    ws[x[i]=r[i]]++; for(i=1;i<m;i++) ws[i]+=ws[i-1]; for(i=n-
    1;i>=0;i--) sa[--ws[x[i]]]=i; for(j=1,p=1;p<n;j*=2,m=p)
    { 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; for(i=0;i<n;i++) wv[i]=x[y[i]]; for(i=0;i<m;i++)
    ws[i]=0; for(i=0;i<n;i++) ws[wv[i]]++; for(i=1;i<m;i++)
    ws[i]+=ws[i-1]; for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
    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++;
    }
    return;
}
```

待排序的字符串放在 r 数组中，从 r[0]到 r[n-1]，长度为 n，且最大值小于 m。为了函数操作的方便，约定除 r[n-1]外所有的 r[i]都大于 0, r[n-1]=0。函数结束后，结果放在 sa 数组中，从 sa[0]到 sa[n-1]。函数的第一步，要对长度为 1 的字符串进行排序。一般来说，在字符串的题目中，r 的最大值不会很大，所以这里使用了基数排序。如果 r 的最大值很大，那么把这段代码改成快速排序。代码：

```
for(i=0;i<m;i++)        ws[i]=0;        for(i=0;i<n;i++)
ws[x[i]=r[i]]++;    for(i=1;i<m;i++)    ws[i]+=ws[i-1];
for(i=n-1;i>=0;i--) sa[--ws[x[i]]]=i;
```

这里 x 数组保存的值相当于是 rank 值。下面的操作只是用 x 数组来比较字符的大小，所以没有必要求出当前真实的 rank 值。

接下来进行若干次基数排序，在实现的时候，这里有一个小优化。基数排序要分两次，第一次是对第二关键字排序，第二次是对第一关键字排序。对第二关键字排序的结果实际上可以利用上一次求得的 sa 直接算出，没有必要再算一次。

代码：

```
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;
```

其中变量 j 是当前字符串的长度，数组 y 保存的是对第二关键字排序的结果。

然后要对第一关键字进行排序，代码：

```
for(i=0;i<n;i++) wv[i]=x[y[i]]; for(i=0;i<m;i++) ws[i]=0;
for(i=0;i<n;i++)        ws[wv[i]]++;        for(i=1;i<m;i++)
ws[i]+=ws[i-1]; for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
```

这样便求出了新的 sa 值。在求出 sa 后，下一步是计算 rank 值。这里要注意的是，可能有多个字符串的 rank 值是相同的，所以必须比较两个字符串是否完全相同，y 数组的值已经没有必要保存，为了节省空间，这里用 y 数组保存 rank 值。这里又有一个小优化，将 x 和 y 定义为指针类型，复制整个数组的操作可以用交换指针的值代替，不必将数组中值一个一个的复制。代码：

```
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++;
```

其中 cmp 函数的代码是：

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

这里可以看到规定 r[n-1]=0 的好处，如果 r[a]=r[b]，说明以 r[a]或 r[b] 开头的长度为 l 的字符串肯定不包括字符 r[n-1]，所以调用变量 r[a+l]和 r[b+l] 不会导致数组下标越界，这样就不需要做特殊判断。执行完上面的代码后，rank 值保存在 x 数组中，而变量 p 的结果实际上就是不同的字符串的个数。这里可以加一个小优化，如果 p 等于 n，那么函数可以结束。因为在当前长度的字符串中，已经没有相同的字符串，接下来的排序不会改变 rank 值。例如图 1 中的第四次排序，实际上是没有必要的。对上面的两段代码，循环的初始赋值和终止条件可以这样写：

for(j=1,p=1;p<n;j*=2,m=p) {……}

在第一次排序以后，rank 数组中的最大值小于 p，所以让 m=p。

整个倍增算法基本写好，代码大约 25 行。

算法分析：

倍增算法的时间复杂度比较容易分析。每次基数排序的时间复杂度为 O(n)，排序的次数决定于最长公共子串的长度，最坏情况下，排序次数为 logn 次，所以总的时间复杂度为 O(nlogn)。

2. DC3 算法

DC3 算法分 3 步：

（1）、先将后缀分成两部分，然后对第一部分的后缀排序。

将后缀分成两部分，第一部分是后缀 k（k 模 3 不等于 0），第二部分是后缀 k（k 模 3 等于 0）。先对所有起始位置模 3 不等于 0 的后缀进行排序，即对 suffix(1), suffix(2), suffix(4), suffix(5), suffix(7)……进行排序。做法是将 suffix(1)和 suffix(2)连接，如果这两个后缀的长度不是 3 的倍数，那先各自在末尾添 0 使得长度都变成 3 的倍数。然后每 3 个字符为一组，进行基数排序，将每组字符“合并”成一个新的字符。然后用递归的方法求这个新的字符串的后缀数组。如图 3 所示。在得到新的字符串的 sa 后，便可以计算出原字符串所有起始位置模 3 不等于 0 的后缀的 sa。要注意的是，原字符串必须以一个最小的且前面没有出现过的字符结尾，这样才能保证结果正确（请读者思考为什么）。

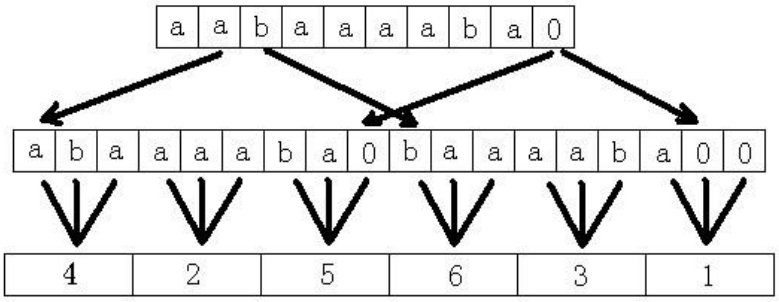


图3

（２）、利用（1）的结果，对第二部分的后缀排序。

剩下的后缀是起始位置模 3 等于 0 的后缀，而这些后缀都可以看成是一个字符加上一个在（1）中已经求出 rank 的后缀，所以只要一次基数排序便可以求出剩下的后缀的 sa。

（３）、将（1）和（2）的结果合并，即完成对所有后缀排序。

这个合并操作跟合并排序中的合并操作一样。每次需要比较两个后缀的大小。分两种情况考虑，第一种情况是 suffix(3*i)和 suffix(3*j+1)的比较，可以把 suffix(3*i)和 suffix(3*j+1)表示成：

suffix(3*i) = r[3*i] + suffix(3*i+1) suffix(3*j+1) = r[3*j+1] + suffix(3*j+2)

其中 suffix(3*i+1)和 suffix(3*j+2)的比较可以利用（2）的结果快速得到。

第二种情况是 suffix(3*i)和 suffix(3*j+2)的比较，可以把 suffix(3*i)和 suffix(3*j+2)表示成：

suffix(3*i) = r[3*i] + r[3*i+1] + suffix(3*i+2)

suffix(3*j+2) = r[3*j+2] + r[3*j+3] + suffix(3*(j+1)+1)

同样的道理，suffix(3*i+2)和 suffix(3*(j+1)+1) 的比较可以利用（2）的结果快速得到。所以每次的比较都可以高效的完成，这也是之前要每 3 个字符合并，而不是每 2 个字符合并的原因。

具体实现：

```
#define F(x) ((x)/3+((x)%3==1?0:tb)) #define G(x)
((x)<tb?(x)*3+1:((x)-tb)*3+2) int
wa[maxn],wb[maxn],wv[maxn],ws[maxn];
int c0(int *r,int a,int b)
{return r[a]==r[b]&&r[a+1]==r[b+1]&&r[a+2]==r[b+2];}
int c12(int k,int *r,int a,int b)
```

```
{if(k==2) return r[a]<r[b]||r[a]==r[b]&&c12(1,r,a+1,b+1); else return
r[a]<r[b]||r[a]==r[b]&&wv[a+1]<wv[b+1];} void sort(int *r,int *a,int *b,int n,int
m)
{
        int i;
        for(i=0;i<n;i++) wv[i]=r[a[i]]; for(i=0;i<m;i++) ws[i]=0;
        for(i=0;i<n;i++)      ws[wv[i]]++;      for(i=1;i<m;i++)
        ws[i]+=ws[i-1]; for(i=n-1;i>=0;i--) b[--ws[wv[i]]]=a[i];
        return;
}
void dc3(int *r,int *sa,int n,int m)
{   int   i,j,*rn=r+n,*san=sa+n,ta=0,tb=(n+1)/3,tbc=0,p;   r[n]=r[n+1]=0;
        for(i=0;i<n;i++) if(i%3!=0) wa[tbc++]=i; sort(r+2,wa,wb,tbc,m);
        sort(r+1,wb,wa,tbc,m);                          sort(r,wa,wb,tbc,m);
        for(p=1,rn[F(wb[0])]=0,i=1;i<tbc;i++)          rn[F(wb[i])]=c0(r,wb[i-
        1],wb[i])?p-1:p++; if(p<tbc) dc3(rn,san,tbc,p);
        else for(i=0;i<tbc;i++) san[rn[i]]=i; for(i=0;i<tbc;i++) if(san[i]<tb)
        wb[ta++]=san[i]*3; if(n%3==1) wb[ta++]=n-1; sort(r,wb,wa,ta,m);
        for(i=0;i<tbc;i++) wv[wb[i]=G(san[i])]=i; for(i=0,j=0,p=0;i<ta &&
        j<tbc;p++)          sa[p]=c12(wb[j]%3,r,wa[i],wb[j])?wa[i++]:wb[j++];
        for(;i<ta;p++) sa[p]=wa[i++]; for(;j<tbc;p++) sa[p]=wb[j++];
        return;
}
```

各个参数的作用和前面的倍增算法一样，不同的地方是 r 数组和 sa 数组的大小都要是 3*n，这为了方便下面的递归处理,不用每次都申请新的内存空间。

函数中用到的变量：

```
int i,j,*rn=r+n,*san=sa+n,ta=0,tb=(n+1)/3,tbc=0,p;
```

rn 数组保存的是（1）中要递归处理的新字符串，san 数组是新字符串的 sa。变量 ta 表示起始位置模 3 为 0 的后缀个数，变量 tb 表示起始位置模 3 为 1 的后缀个数，已经直接算出。变量 tbc 表示起始位置模 3 为 1 或 2 的后缀个数。先按（1）中所说的用基数排序把 3 个

字符“合并”成一个新的字符。为了方便操作，

先将 r[n]和 r[n+1]赋值为 0。

代码：

```
r[n]=r[n+1]=0;          for(i=0;i<n;i++)          if(i%3!=0)

wa[tbc++]=i;                              sort(r+2,wa,wb,tbc,m);

sort(r+1,wb,wa,tbc,m); sort(r,wa,wb,tbc,m);
```

其中 sort 函数的作用是进行基数排序。代码：

```
void sort(int *r,int *a,int *b,int n,int m)
{
        int i;

        for(i=0;i<n;i++) wv[i]=r[a[i]]; for(i=0;i<m;i++) ws[i]=0;

        for(i=0;i<n;i++)        ws[wv[i]]++;        for(i=1;i<m;i++)

        ws[i]+=ws[i-1]; for(i=n-1;i>=0;i--) b[--ws[wv[i]]]=a[i];

        return;
}
```

基数排序结束后，新的字符的排名保存在 wb 数组中。

跟倍增算法一样，在基数排序以后，求新的字符串时要判断两个字符组是否完全相同。代码：

```
for(p=1,rn[F(wb[0])]=0,i=1;i<tbc;i++) rn[F(wb[i])]=c0(r,wb[i-1],wb[i])?p-1:p++;
```

其中 F(x)是计算出原字符串的 suffix(x)在新的字符串中的起始位置，c0 函数是比较是否完全相同，在开头加一段代码：

```
#define  F(x)  ((x)/3+((x)%3==1?0:tb))  inline  int  c0(int

*r,int a,int b)

{return r[a]==r[b]&&r[a+1]==r[b+1]&&r[a+2]==r[b+2];}
```

接下来是递归处理新的字符串，这里和倍增算法一样，可以加一个小优化，如果 p 等于 tbc，那么说明在新的字符串中没有相同的字符，这样可以直接求出 san 数组，并不用递归处理。代码：

```
if(p<tbc)  dc3(rn,san,tbc,p);  else  for(i=0;i<tbc;i++)

san[rn[i]]=i;
```

然后是第（2）步，将所有起始位置模 3 等于 0 的后缀进行排序。其中对第二关键字的排序结果可以由新字符串的 sa 直接计算得到，没有必要再排一次。

代码：

for(i=0;i<tbc;i++) if(san[i]<tb) wb[ta++]=san[i]*3; if(n%3==1) wb[ta++]=n-1;

sort(r,wb,wa,ta,m); for(i=0;i<tbc;i++)

wv[wb[i]=G(san[i])]=i;

要注意的是，如果 n%3==1，要特殊处理 suffix(n-1)，因为在 san 数组里并没有 suffix(n)。G(x)是计算新字符串的 suffix(x)在原字符串中的位置，和 F(x)
为互逆运算。在开头加一段：
#define G(x) ((x)<tb?(x)*3+1:((x)-tb)*3+2)。

最后是第（3）步，合并所有后缀的排序结果，保存在 sa 数组中。代码：

for(i=0,j=0,p=0;i<ta && j<tbc;p++)

sa[p]=c12(wb[j]%3,r,wa[i],wb[j])?wa[i++]:wb[j++]; for(;i<ta;p++)

sa[p]=wa[i++]; for(;j<tbc;p++) sa[p]=wb[j++];

其中 c12 函数是按（3）中所说的比较后缀大小的函数，k=1 是第一种情况， k=2 是第二种情况。代码：

int c12(int k,int *r,int a,int b)

{if(k==2) return r[a]<r[b]||r[a]==r[b]&&c12(1,r,a+1,b+1); else return

r[a]<r[b]||r[a]==r[b]&&wv[a+1]<wv[b+1];}

整个 DC3 算法基本写好，代码大约 40 行。

算法分析：

假设这个算法的时间复杂度为 f(n)。容易看出第（1）步排序的时间为 O(n) （一般来说，m 比较小，这里忽略不计）,新的字符串的长度不超过 2n/3，求新
字符串的 sa 的时间为 f(2n/3)，第（2）和第（3）步的时间都是 O(n)。所以

$$f(n) = O(n) + f(2n/3)\ f(n) \le c \times n + f(2n/3)$$

$$f(n) \le c \times n + c \times (2n/3) + c \times (4n/9) + c \times (8n/27) + \ldots\ldots \le 3c \times n$$

所以 $f(n) = O(n)$

由此看出，DC3 算法是一个优秀的线性算法。

3. 倍增算法和 DC3 算法的比较

从时间复杂度、空间复杂度、编程复杂度和实际效率等方面对倍增算法与 DC3 算法进行比较。

时间复杂度：

倍增算法的时间复杂度为 O(nlogn)，DC3 算法的时间复杂度为 O(n)。从常数上看，

DC3 算法的常数要比倍增算法大。

空间复杂度：

倍增算法和 DC3 算法的空间复杂度都是 O(n)。按前面所讲的实现方法，倍增算法所需数组总大小为 6n，DC3 算法所需数组总大小为 10n。

编程复杂度：

倍增算法的源程序长度为 25 行，DC3 算法的源程序长度为 40 行。

实际效率：

测试环境：NOI-linux Pentium(R) 4 CPU 2.80GHz

N	倍增算法	DC3 算法
200000	192	140
300000	367	244
500000	750	499
1000000	1693	1248

（不包括读入和输出的时间，单位：ms）

从表中可以看出，DC3 算法在实际效率上还是有一定优势的。倍增算法容易实现，DC3 算法效率比较高，但是实现起来比倍增算法复杂一些。对于不同的题目，应当根据数据规模的大小决定使用哪个算法。

三、 解题报告

1. POJ 3294

1) 问题描述

Life Forms

Time Limit: 5000MS Memory Limit: 65536K

Total Submissions: 16833 Accepted: 4946

Description

You may have wondered why most extraterrestrial life forms resemble humans,

differing by superficial traits such as height, colour, wrinkles, ears, eyebrows and the like. A few bear no human resemblance; these typically have geometric or amorphous shapes like cubes, oil slicks or clouds of dust.

The answer is given in the 146th episode of Star Trek - The Next Generation, titled The Chase. It turns out that in the vast majority of the quadrant's life forms ended up with a large fragment of common DNA.

Given the DNA sequences of several life forms represented as strings of letters, you are to find the longest substring that is shared by more than half of them.

Input

Standard input contains several test cases. Each test case begins with 1 ≤ n ≤ 100, the number of life forms. n lines follow; each contains a string of lower case letters representing the DNA sequence of a life form. Each DNA sequence contains at least one and not more than 1000 letters. A line containing 0 follows the last test case.

Output

For each test case, output the longest string or strings shared by more than half of the life forms. If there are many, output all of them in alphabetical order. If there is no solution with at least one letter, output "?".

Leave an empty line between test cases.

2) 算法设计

将所有的字符串通过不同的拼接符相连，作一次后缀数组，二分答案的长

度，然后在 h 数组中分组，判断是否可行，按照 sa 扫描输出长度为 L 的答案即可。注意在一个子串中重复出现答案串的情况。

3) 程序代码

```
#define _CRT_SECURE_NO_WARNINGS

#include<cstdio>

#include<map>

#include<cstdlib>

#include<algorithm>

#include<cmath>

#include <cstring>

#include<vector>

#include<map>

#define MAXNUM 105

#define MAXSIZE 1050

#define MAXNUMSIZE MAXNUM*MAXSIZE

#define    FOREACH(N)    for(i=0;i<N;i++)    using
```

```
namespace std; typedef int ints[MAXNUMSIZE];
int N;

char input_str[MAXNUM][MAXSIZE]; int length[MAXNUM],
ans[MAXNUM], vis[MAXNUM];
ints input_num, sa, myrank, height;
```

2. POJ 3415

1) 问题描述

Common Substrings

Time Limit: 5000MS Memory Limit: 65536K

Total Submissions: 12150 Accepted: 4103

Description

A substring of a string T is defined as:

T(i, k)=TiTi+1...Ti+k-1, 1≤i≤i+k-1≤|T|.

Given two strings A, B and one integer K, we define S, a set of triples (i, j, k):

S = {(i, j, k) | k≥K, A(i, k)=B(j, k)}.

You are to give the value of |S| for specific A, B and K.

Input

The input file contains several blocks of data. For each block, the first line

contains one integer K, followed by two lines containing
strings A and B, respectively. The input file is ended by K=0.

$1 \le |A|, |B| \le 105$

$1 \le K \le \min\{|A|, |B|\}$

Characters of A and B are all Latin letters. Output

For each case, output an integer $|S|$.

2) 算法设计

首先，很容易想到 $O(n^2)$ 的算法，将 A 串和 B 串加拼接符相连，做一遍后缀数组，把分别属于 A 和 B 的所有后缀匹配，LCP-k+1 就是对答案的贡献，但是在这个基础上该如何优化呢。

我们可以发现按照 sa 的顺序下来，每个后缀和前面的串的 LCP 就是区间 LCP 的最小值，那么我们维护一个单调栈，将所有单调递减的 LCP 值合并，保存数量和长度，对每个属于 B 串的后缀更新前面 A 串的后缀的贡献，对属于 A 串的后缀更新属于 B 串的后缀的贡献即可。

3) 程序代码

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>

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

```
#define LL long long

#define N 200005



int n, m, k, ls, la; char a[N], s[N]; int *x, *y, A[N], B[N], c[N],

sa[N], height[N], myrank[N]; int stack[N], cnt[N], top;

LL sum, ans;



void clear()

{    n = m = ls = la = top = 0; sum = ans = 0LL;    memset(A, 0, sizeof(A));

memset(B, 0, sizeof(B)); memset(c, 0,
```

LCA 和 RMQ

一、 涉及问题

LCA：Least Common Ancestors（最近公共祖先）。对于一棵树 T 中的任意两个节点 u，v，算法 x=LCA(T, u, v) 返回的结果为节点 u，v 的最近公共父节点，也即是离树 T 的根节点最远的节点 x，使得 x 同时是 u 和 v 的祖先。

例子：对于 T=(V,E) V={1,2,3,4,5,6,7,8,9,10,11,12,13} E={(1,2),(1,3),(1,4),(3,5),(3,6),(3,7),(6,8),(6,9),(7,10),(7,11),(10,12),(10,13)} 则有：LCA(T,5,2)=1 LCA(T,3,4)=1 LCA(T,5,12)=3，特别的有 LCA(T,1,2)=1，要不然对无向图图来说将会有 LCA(T,1,2)=3 或 LCA(T,1,2)=4。如下图：

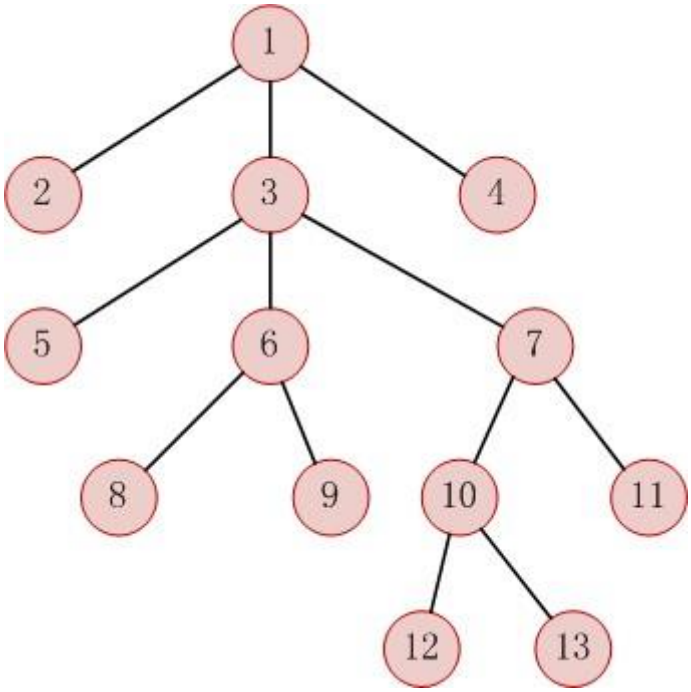


图 1 一颗树的例子

RMQ：Rang Maxmum/Minimun Query（区间最值查询）。对于长度为 n 的数组 A，查询问题 Q(A, i, j)，返回数组 A 中在下标区间[i, j]内的最值（最大值或最小值）下标，（注意这里是最值下标，而不是最值）。 例子：对数列：5,8,1,3,6,4,9,5,7,2 有：RMQ(A,2,4)=3，RMQ(A,6,9)=6。

下标	1	2	3	4	5	6	7	8	9	10
数值	5	8	1	3	6	4	8	5	7	2

表 1　一个数组的例子

二、 算法介绍

1. LCA 和 RMQ 问题的相互转化

1）LCA 问题向 RMQ 问题转换

深度遍历树 T，将数节点的内容存入数组，将该节点的深度也记录下来。同时，访问完每个节点的所有分支后，在退回到该节点的时候，也讲该节点的内容和深度存入数组 S。同时记录节点第一次在标准出现的下标和深度。表 2 是将图 1 深度遍历后得到的表，表 3 是每个节点在深度遍历时第一次出现在整个遍历序列中的下标 R[i]和深度 d(i)。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	2	1	3	5	3	6	8	6	9	6	3	7	10	12	10	13	10	7	11	7	3	1	4	1
1	2	1	2	3	2	3	4	3	4	3	2	3	4	5	4	5	4	3	4	3	2	1	2	1

表 2　S 数组（树节点<访问次序，值，深度>）

R[i]	1	2	4	5	7	8	10	13	14	15	17	20	24
Index[i]	1	2	3	5	6	8	9	7	10	12	13	11	4
D[i]	1	2	2	3	3	4	4	3	4	5	5	4	2

表 3 节点的第一次入栈序列下标和深度

对于树的查询访问题 x=LCA(T,u, v)，其中 u，v 为树节点的标记，由于节点 u,v 的标记是唯一的,u,v 的父节点 x 的深度 d(x)满足 d(x)<=max(d(u), d(v))。从表 2 中可以看出，节点 x， u，v 出现在表中第二行的序列肯定为[x,...u, ..x ...v, ...x] ，相应的表中第三行有序列
[d(x),...d(u),...d(x),...d(v),...d(x)] 这里 x 也可能等于 u 或者 v。

R[u]和 R[v]记录了元素 u 和 v 在表 S 中第一次出现的下标，问题 RMQ[S.d, R[u], R[v]]（S 数组见表 2，其中第三行为访问序列深度数组 S.d）即为求深度数组中给定下标 R[u]，R[v]的对应区间中深度元素最小的元素第一出现的下标。E[RMQ[S.d, R[u], R[v]]]的到的结果即为 u，v 最近公共父节点。

实际上与深度遍历有关，看下图：

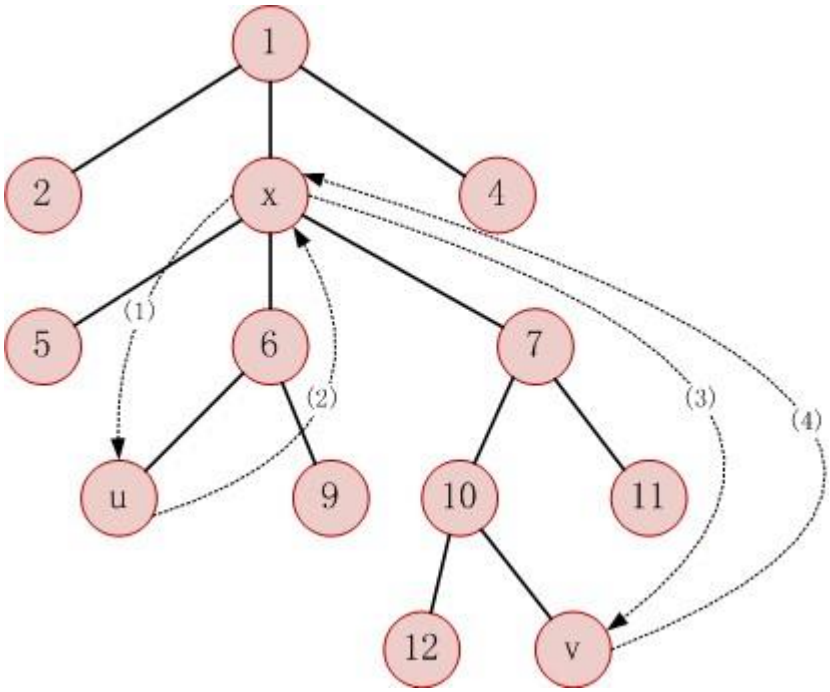


图 2 深度优先遍历轨迹

由于 LCA 问题可以转换成 RMQ 问题，所以我们先讨论 RMQ 问题

2）RMQ 问题向 LCA 问题转换

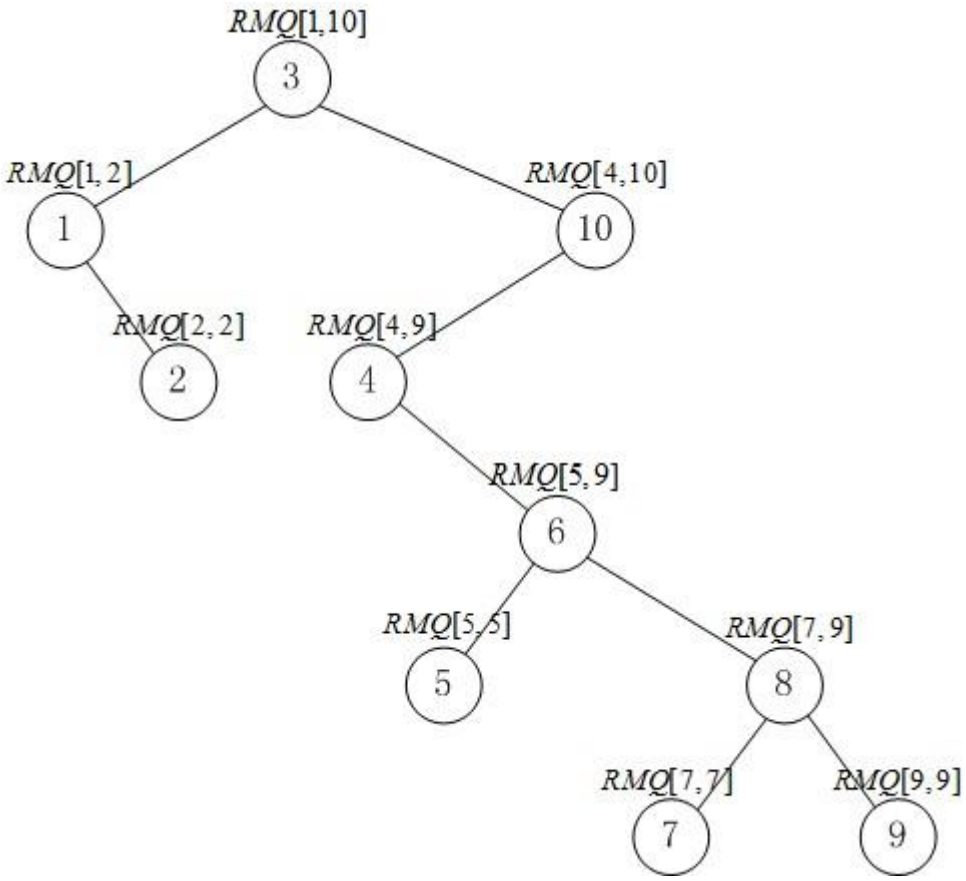
由于 LCA 问题的求解是在树上做遍历，这里需要用到笛卡尔树的结构，将数组转换成一颗笛卡尔树，Cartersian Tree。

树的构造树的主要思想如下。

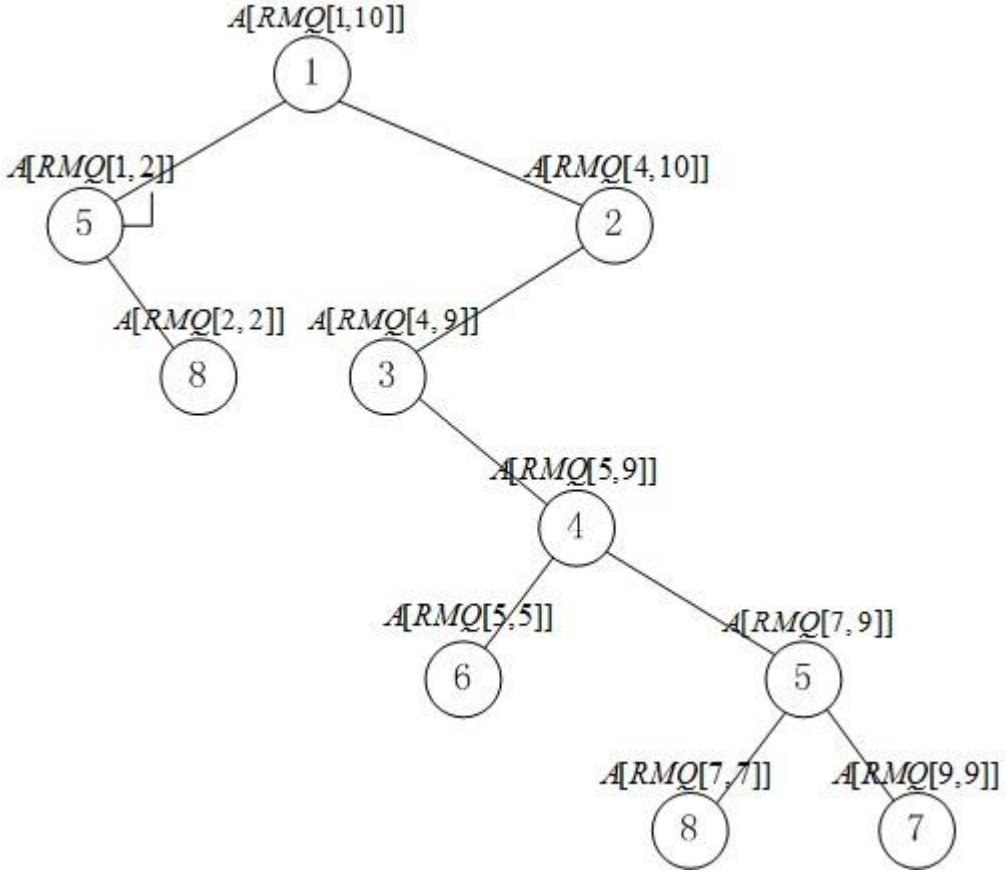
这里的笛卡尔树是一种特殊的最小堆。树中一共有 n 个节点，每个节点的值是数组元素的下标，树的根节点是数组 A 中最小元素的下标 i。根节点的左右孩子分别由数组 A[1..i-1]和数组 A[i+1..n]构造。树的构造是由浅到深，由右到左构造的。

下标	1	2	3	4	5	6	7	8	9	10
数值	5	8	1	3	6	4	8	5	7	2

图 笛卡尔树最终生成结果（中序遍历可得到原数组）



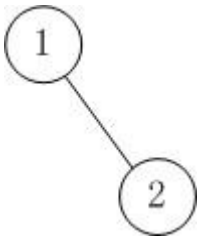
笛卡尔树中的节点的内容[数组元素的下标]



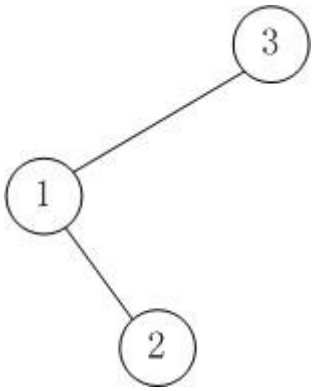
笛卡

尔树中的节点的内容[数组元素]

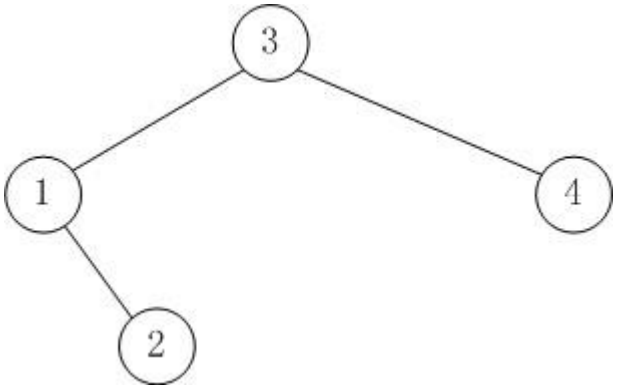
第一个插入树的节点



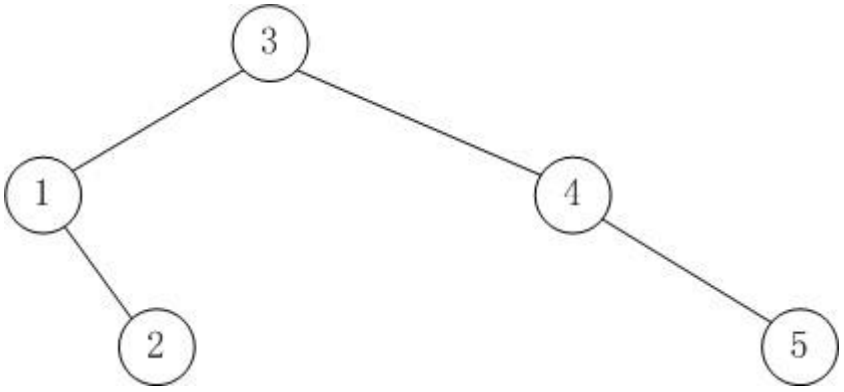
第二个元素比树中最右边路径上的节点 1 的值大，故称为节点 1 的右孩子



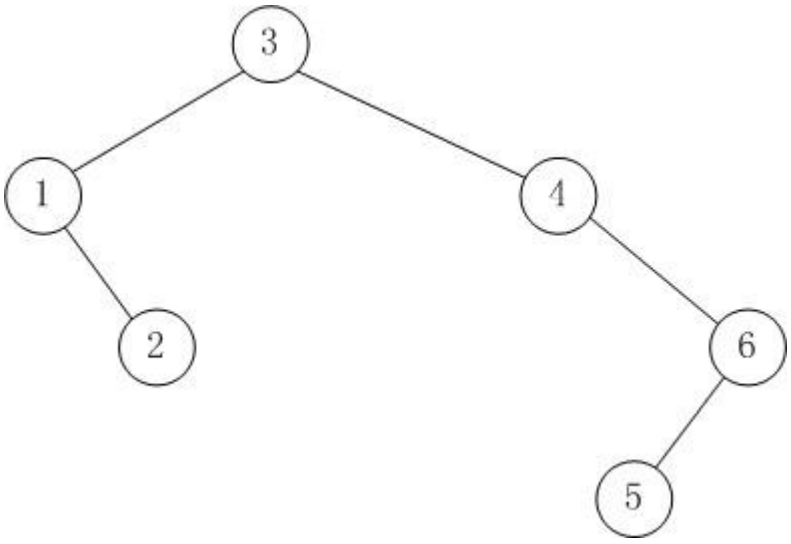
第三个元素的值比树中最右边路径上节点 1，2 的值都小，将节点 1 设为该节点的左孩子



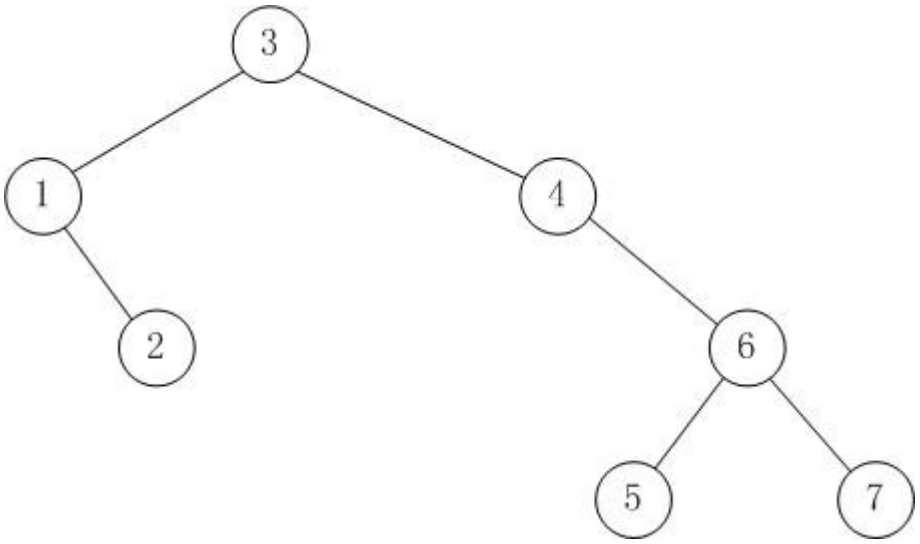
第四个元素的值比树中最右边路径上节点 3 的值小，故成为根节点 3 的右孩子



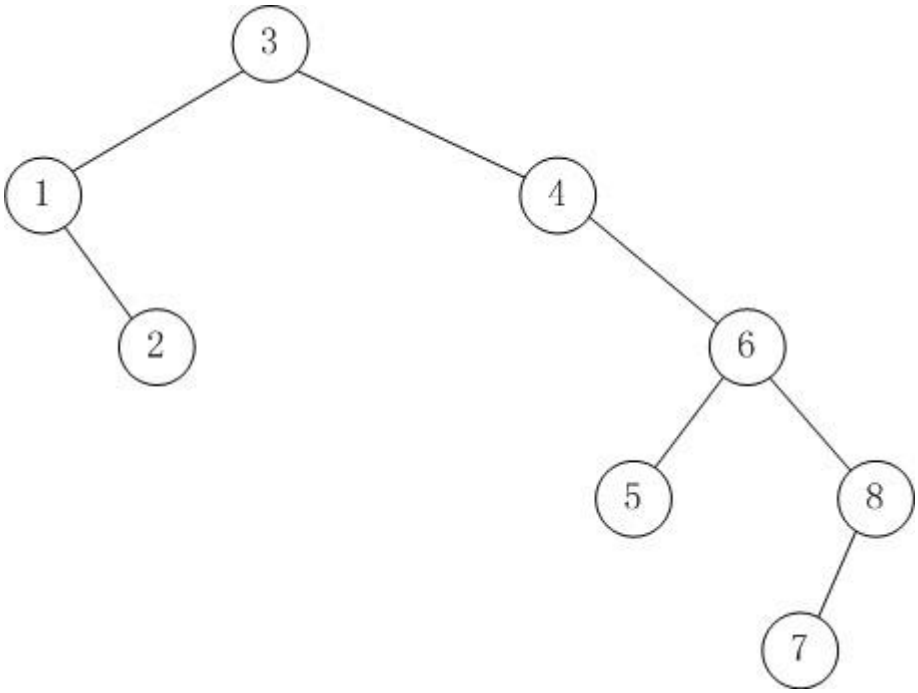
第五个节点元素的值比树中最右边的路径上的节点 3,4 的值都大，故成为节点 4 的右孩子



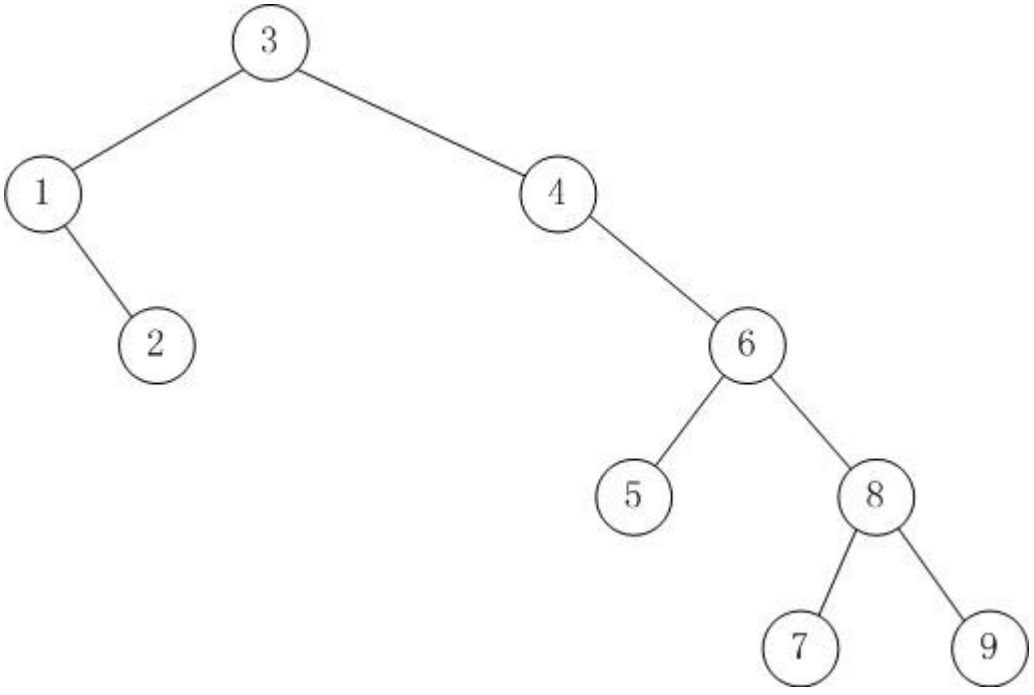
第六个节点元素的值比树中最右边路径上节点 5 的值小，但比节点 4 的值大，故成为节点 4 的
右孩子，将节点 5 设为该节点的左孩子



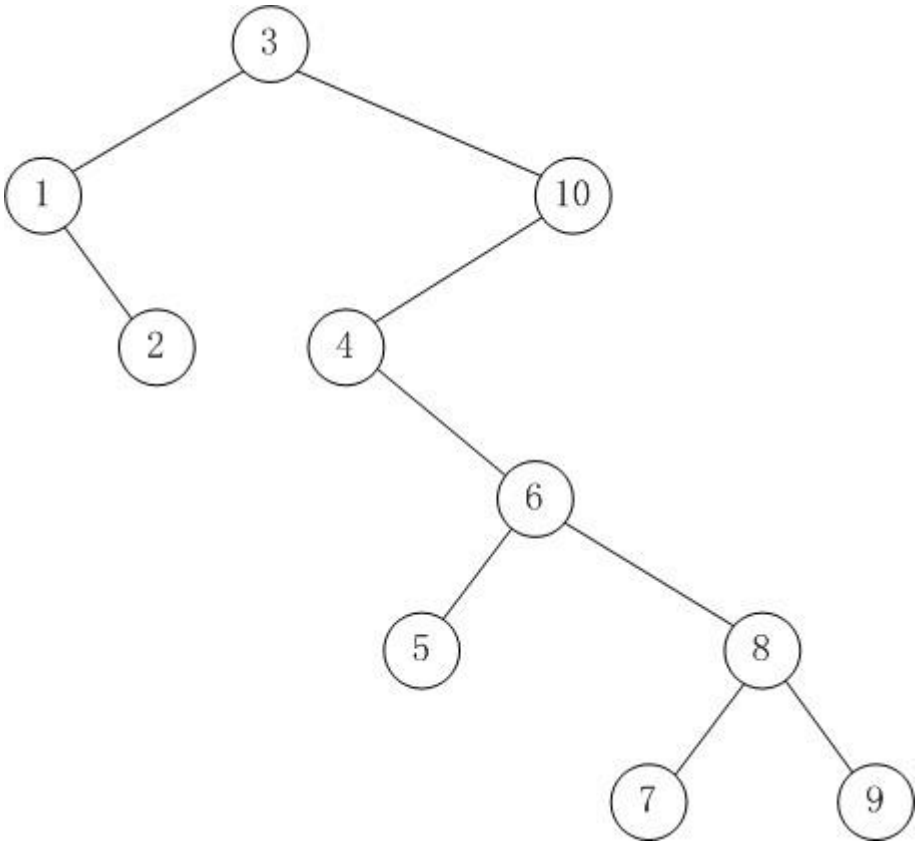
第七个元素的值比树中最右边路径上节点 6 的值大，故成为节点 6 的右孩子



第八个元素的值比树中最右边路径上节点 7 的值小，但比节点 6 的值大，故该节点称为节点 6
的右孩子，节点 7 成为该节点的左孩子



第九个元素的值比树中最右边路径上节点 8 的值大，故成为节点 8 的右孩子



第十个元素的值比树中最右边路径上的节点 9,8,6,4 的值都小，但比节点 3 的值大，故成为节点 3 的右孩子，节点 4 成为该节点的左孩子

构造过程中需要查找一条从根节点出发向树的最右孩子遍历的路径，可以用栈保存。然后用新的节点的值和栈顶元素比较即可。

首先，先从 A[1]开始建立，以后每次加一个数，就修改 Cartesian 数，不难发现，这个数一
定在这棵树的最右边的路径上。而且一定没有右孩子（应为数组为树的中序遍历），所以，只沿着最右路径自底向上把各个节点 p 和 A[i]做比较，如果 p<A[i]，那么 A[i]就为 p 的右孩子，如果 p>A[i]，那么比较 p 的父亲与 A[i]，如果 A[i]>p 的父亲，那么 A[i]为 p 的父亲的右孩子，而 p 则改为 A[i]的左孩子。因为每个节点最多进入和退出最右路径各一次，所以，均摊时间复杂度为 O(n)。

具体算法不描述了：由上面的图例可知。 定理：数组 A 的 Cartesian 数记为 C(A)，则 RMQ(A, i, j) = LCA(C(A), i, j)。

2. ST 算法

ST 算法实际上是一个非常有名的在线处理 RMQ 问题的算法，它可以在 O(nlogn)时间内进行预处理，然后在 O(1)时间内回答每个查询。它利用动态规划的思想实现。

首先是预处理，用动态规划（DP）解决。设 A[i]是要求区间最值的数列，F[i,j]表示从第 i 个

数起连续 2^j 个数中的最大值,即 F[i,j]=max A[i..i+2^j-1]。例如数列 3 2 4 5 6 8 1 2 9 7，

F[1,0]表示第 1 个数起，长度为 2^0=1 的最大值，其实就是第一个数 3。

F[1,1]=Max A[1..2]=3
F[1,2]=Max A[1..4]=5
F[1,3]=Max A[1..8]=8
F[2,0]=Max A[2..2]=2 F[2,1]=Max A[2..3]=4

…… 从这里可以看出初值 F[i,0]=A[i]。

这样，DP 的状态、初值都已经有了，剩下的就是状态转移方程。我们把 F[i,j]平均分成两段（因为

F[i,j]一定是偶数个数字），从 i 到 i+2^(j-1)-1 为一段，i+2^(j-1)到 i+2^j-1 为一段(长度都为 2^(j-1))。用上例说明，当 i=1，j=3 时,这两个段就是 3,2,4,5 和 6,8,1,2 这两段。F[i,j]就是这两段的最大值中的最大值。于是我们得到了动态规划方程

F[i, j] = max(F[i, j-1] , F[i + 2^(j-1), j-1])

然后是查询。若区间 [i,j] 的长度为 2 的幂，那么取 k=[log2(j-i+1)]，k 肯定为整数。则有：RMQ(A, i,

j)=max{F[i,k],F[j-2^k+1,k]}，其实这里有 [i,k]=[j-2^k+1,k] ，这步计算重复了，仅为了和区间 [i,j] 的长度不

是 2 的幂统一。 举例说明：要求区间 [2，9]的最大值，就不需要把该区间分成两个区间，因为区间的长度刚好是 2 的 3 次幂。k=[log2(j-i+1)]=3，RMQ(A,2,9)=max{F(2,3),F(2,3)}。对于区间长度不是 2 的幂，那么需要将区间重叠划分，仍然取 k=[log2(j-i+1)]，这里用到下取整

$x \le \lfloor x \rfloor$。举例说明：要求区间 [2，8]的最大值，就要把它分成[2,5]和[5,8]两个区间，其中区间[2,8]有 7 个元素，区间[2,5]，[5,8]个 4 个元素，元素 5 被重复的包含在了两个区间内。因为这两个区间的最大值我们可以直接由 F[2，2]和 F[5，2]得到，故区间[2,8]的最值也是区间[2,5]和区间[5,8]的最值的最值。算法伪代码：

```
RMQ(A,i,j)
//计算 F[][]
1  //F[i][j]中存的是从 i 开始的 2^j 个数据中的最大值，A 中存有数组的值
2
3  //初始化
4
5
6  fori : 1 to n      F[i,0] = i
   M[i,0]=A[i]
7
8  //滚动数组的计算
9
10 forj : 1 to log(n)/log(2)//利用了换底公式,由于 j 为整数，巧妙的用了下取整函数
11
12
13    fori : 1 to (n+1-2^j)
14
15       F[i,j] = A[F[i,j-1]]>A[F[i+2^(j-1),j-1]] ?F[i,j-
   1] :F[i+2^(j-1),j-1]
16 M[i,j] = max{M[i,j-1],M[i+2^(j-1),j-1]}
17
18
19 //查询
20
21
22 RMQ(i, j)
23
   k = [log(j-i+1) / log(2)]//利用了换底公式,由于 k 为整数，巧妙的用了下取整函数

   returnMAX(F[i,k], F[j-2^k+1,k])
```

上面第一部分的算法利用动态规划构建 :对于区间长度(j)有如下情况： j=0: L=1 F[i,0]=A[i]

j=1: L=2 F[i,1]=max{F[i,0], F[i+1,0]}=max{A[i], A[i+1]}
j=2: L=4 F[i,2]=max{F[i,1], F[i+2,1]}=max{{A[i],A[i+1]}, {A[i+2],A[i+3]}} j=3: L=8 F[i,3]=max{F[i,2], F[i+4,2]}=max{{A[i],A[i+1],A[i+2],A[i+3]}, {A[i+4],A[i+5],A[i+6],A[i+7]}} .....

详细的计算过程如下表：

4										
3	8	8	8							
2	8	8	6	8	8	8	8			
1	8	8	3	6	6	8	8	7	7	
0	5	8	1	3	6	4	8	5	7	2
	1	2	3	4	5	6	7	8	9	10

表 4　数组 M 的计算得到的结果（表 1 中的数据）

4										
3	F[1,3]	F[2,3]	F[3,3]							
2	F[1,2]	F[2,2]	F[3,2]	F4,2]	F[5,2]	F[6,2]	F[7,2]			
1	F[1,1]	F[2,1]	F[3,1]	F[4,1]	F[5,1]	F[6,1]	F[7,1]	F[8,1]	F[9,1]	
0	F[1,0]	F[2,0]	F[3,0]	F[4,0]	F[5,0]	F[6,0]	F[7,0]	F[8,0]	F[9,0]	F[10,0]
	1	2	3	4	5	6	7	8	9	10

表 5　数组的计算得到的结果的下标

4										
3	2/7	2/7	7							
2	2	2	5	7	7	7	7			
1	2	2	4	5	5	7	7	9	9	
0	1	2	3	4	5	6	7	8	9	10
	1	2	3	4	5	6	7	8	9	10

表 6　数组 F 的计算得到的结果

复杂度分析

很明显空间复杂度为 O(n*[log2(n)])=O(n*log(n))

时间复杂度为初始化部分 O(n)，再加上计算数组 F 的部分 O(n*[log2(n)])，故总的

渐近时间复杂度为 O(n*log(n))。

当然，该问题也可以用线段树（也叫区间树）解决，算法复杂度为：O(N)~O(logN)。

3. Tarjan 算法

Tarjan 算法是利用深度优先遍历，也利用了并查集的数据结构。

需要对每一个节点建立一个以该节点为根节点，包括所有孩子节点集合，且每个节点集合有一个父节点。初始化该集合中每个元素的的父节点为该节点自己。在一颗子树访问完后，该子树合并成一个集合，也即是该集合中的所有元素的父节点都指向该子树的根节点。 算法思想

以当前节点 u 进行 DFS，建立一个集合 Set:u={u}，(每个节点 node={index, parent}， index 为节点标记，parent 为公共父节点，初始值为 null)。初始化集合中元素 u.parent=u。

对于 u 的每个孩子 v，回到 1）DFS 递归。从该孩子节点返回后，将孩子节点建立一个集合 Set:v 并入 Set:u，并将该集合中所有元素的父节点设为 u。

当 u 的所有孩子访问完后，设定节点 u 已访问标记。（注意访问标记要在，所有的子树访问完后才设立。）

当 u 的所有孩子访问完后，开始询问：对于任何访问集合 Qurey 中的元素(u, v)，如果 v 已经被访问（说明节点 v 落入节点 u 已经被访问过得子树中了，要么在已访问过的兄弟节点所在的集合中了）如图 4.1 中，1）当节点 10 访问完后如图 4.1，节点集合为{10}，能够询问（10,5），（10,9）但不能够询问（10,6），（10,2）因为 6 和 2 还没有被标记，即是没有从 6，

2 的所有孩子中返回。2）当 6 的所有孩子访问完后如图 4.2，能询问（6,9），（6,10），（6,5），仍然不能询问（6,2）以及其他询问。

算法伪代码

```
LCA (u)
{    creatSet(u.set) and initialize u.set={u}
u.parent=u
for every child v of u
{   LCA(v)
union v.set to u.set
for every e int u.set
e.parent=u
}
```

```
            visit[u] 全局访问标记数组
            for every (u,x) in Query
                    if (    )
                        the least common ancestor of (u,x) is x.parent
                    
                等到x被访问过后，在查询，其实用不到else语句块
}
```

举例如下

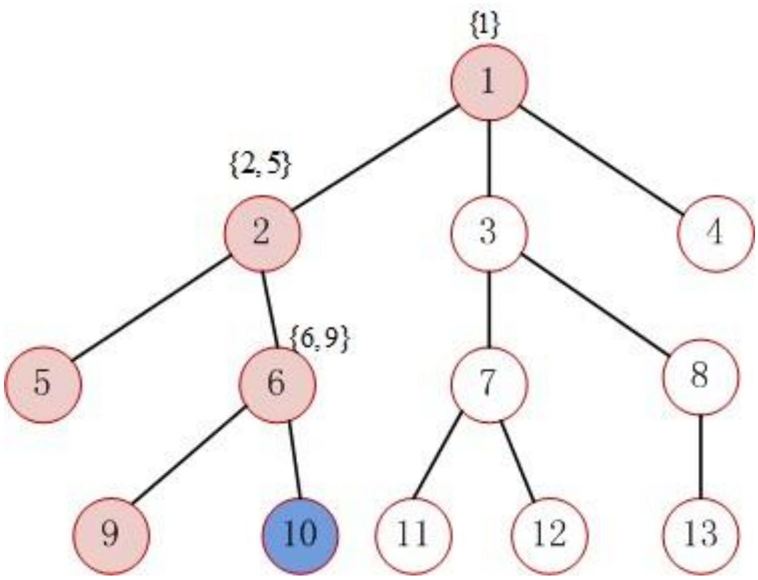


图 4.1

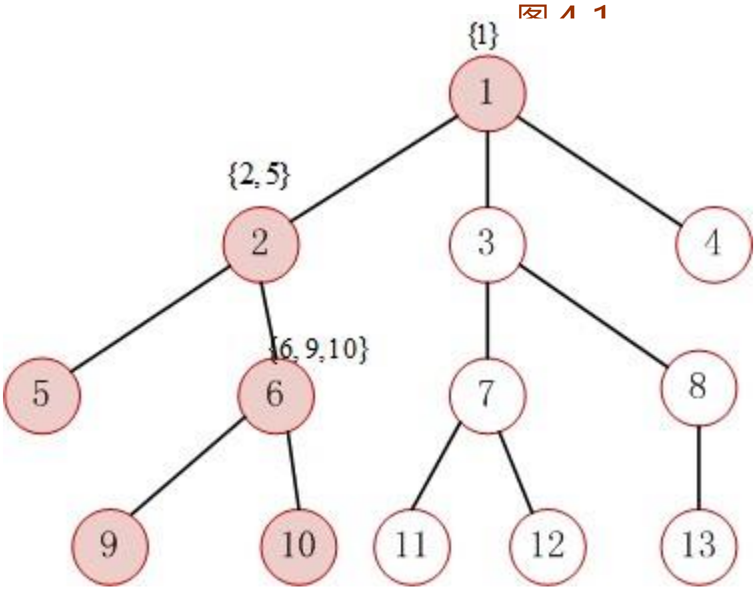


图 4.2

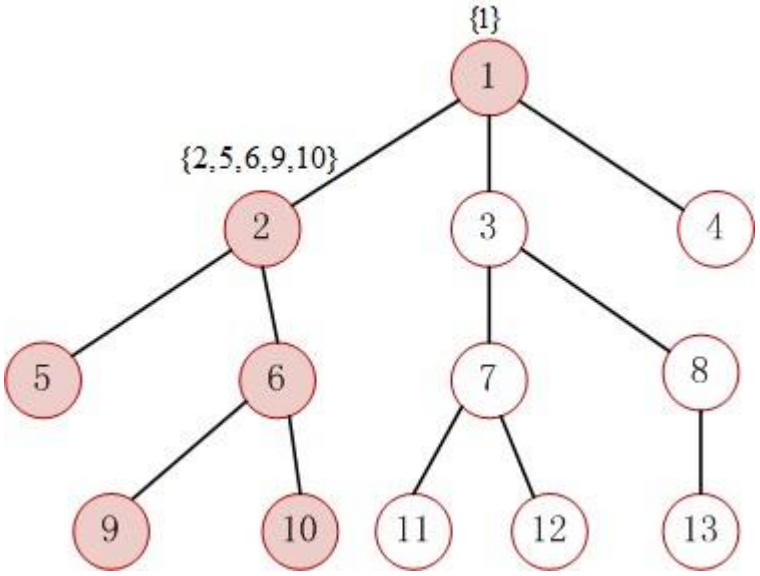


图 4.3

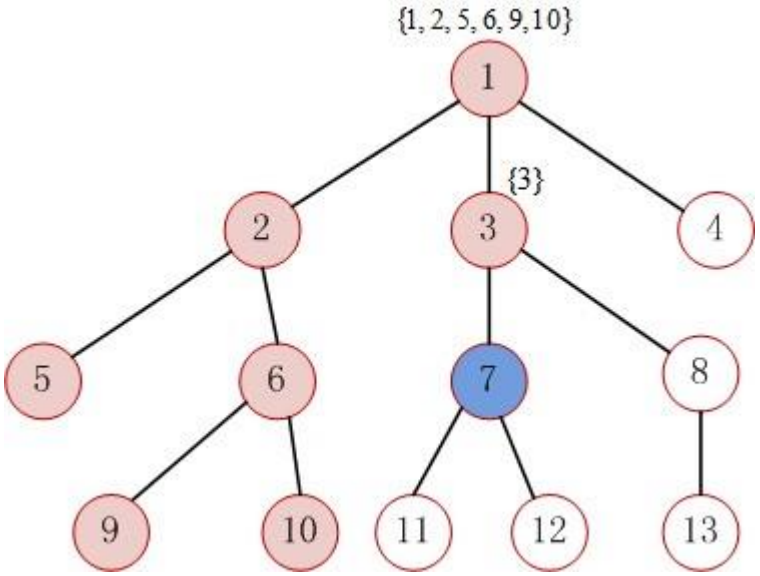


图 4.4

三、 解题报告

1. POJ 1470

1) 问题描述

Closest Common Ancestors

Time Limit: 2000MS Memory Limit: 10000K

Total Submissions: 21450 Accepted: 6798

Description

Write a program that takes as input a rooted tree and a list of pairs of vertices. For each pair (u,v) the program determines the closest common ancestor of u and v in the tree. The closest common ancestor of two nodes u and v is the node w that is an ancestor of both u and v and has the greatest depth in the tree. A node can be its own ancestor (for example in Figure 1 the ancestors of node 2 are 2 and 5)

Input

The data set, which is read from a the std input, starts with the tree description, in the form:

nr_of_vertices vertex:(nr_of_successors) successor1 successor2 ...

successorn

...

where vertices are represented as integers from 1 to n (n <= 900). The tree description is followed by a list of pairs of vertices, in the form:

nr_of_pairs

(u v) (x y) ...

The input file contents several data sets (at least one).

Note that white-spaces (tabs, spaces and line breaks) can be used freely in the input.

Output

For each common ancestor the program prints the ancestor and the number of

pair for which it is an ancestor. The results are printed on the standard output on separate lines, in to the ascending order of the vertices, in the format: ancestor:times

For example, for the following tree:

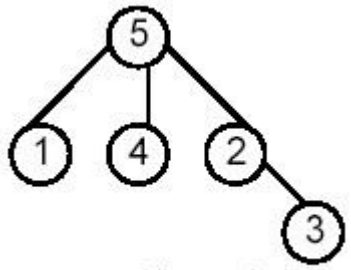


Figure 1

2) 算法设计

这题是 LCA 的模板题。根据输入直接建立树的邻接表，运行离线 Tarjan 算法，查询完毕后按照输入顺序依次输出所查询的最近公共祖先。

3) 程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <vector>
#include <iostream>

#define MAX_N 1000

using namespace std;

int n, m;
vector<int> G[MAX_N]; //G[i]为节点 i 的所有子孙
int parent[MAX_N];
vector<int> Q[MAX_N];
bool visited[MAX_N];
int cnt[MAX_N];
int p[MAX_N]; //并查集

void makeSets() {
	for(int i = 1; i <= n; ++i) {
```

```
            p[i] = i;
        }
}

int find(int u) {
        if(p[u] != u) {
            p[u] = find(p[u]);
        }
        return p[u];
}

void Union(int u, int pa) {
        p[u] = pa;
}

void dfs(int u) {
        p[u] = u;
        for(int i = 0; i < G[u].size(); ++i) {
            int v = G[u][i];
            dfs(v);
        }
        for(int i = 0; i < Q[u].size(); ++i) {
            int v = Q[u][i];
            if(visited[v]) cnt[find(v)]++;
        }
        visited[u] = true;
        p[u] = parent[u];
}

int main(int argc, char const *argv[]) {
        std::ios::sync_with_stdio(false);
#ifdef LOCAL
        freopen("input.txt", "r", stdin);
#endif
        while(scanf("%d", &n) == 1) {
            memset(visited, false, sizeof(visited));
            memset(cnt, 0, sizeof(cnt));
            memset(parent, 0, sizeof(parent));
            makeSets();
            for(int i = 1; i <= n; ++i) {
                int u, num;
                scanf("%d:(%d)", &u, &num);
                for(int j = 0; j < num; ++j) {
                    int v;
```

```
                scanf("%d", &v);
                G[u].push_back(v);
                parent[v] = u;
            }
        }

        int root;
        for(int i = 1; i <= n; ++i) {
            if(parent[i] == 0) {
                parent[i] = i;
                root = i;
                break;
            }
        }

        scanf("%d", &m);
        for(int i = 0; i < m; ++i) {
            int u, v;
            scanf(" (%d %d)", &u, &v);
            if(u == v) {
                cnt[u] += 2;
                continue;
            }
            Q[u].push_back(v);
            Q[v].push_back(u);
        }

        dfs(root);

        for(int i = 1; i <= n; ++i) {
            if(cnt[i] > 0) printf("%d:%d\n", i, cnt[i]);
        }

        for(int i = 1; i <= n; ++i) {
            G[i].clear();
            Q[i].clear();
        }
    }
    return 0;
}
```

4) 性能分析

设树的顶点数为 n，查询数为 q，则程序的运行时间为 Tarjan 算法的运行时间，即为 O(n+q)，空间复杂度为 O(n)。

2. POJ 1986

1) 问题描述

Distance Queries

Time Limit: 2000MS Memory Limit: 30000K

Total Submissions: 15270 Accepted: 5384

Case Time Limit: 1000MS

Description

Farmer John's cows refused to run in his marathon since he chose a path much too long for their leisurely lifestyle. He therefore wants to find a path of a more reasonable length. The input to this problem consists of the same input as in "Navigation Nightmare",followed by a line containing a single integer K, followed by K "distance queries". Each distance query is a line of input containing two integers, giving the numbers of two farms between which FJ is interested in computing distance (measured in the length of the roads along the path between the two farms). Please answer
FJ's distance queries as quickly as possible!

Input

* Lines 1..1+M: Same format as "Navigation Nightmare"

* Line 2+M: A single integer, K. 1 <= K <= 10,000

* Lines 3+M..2+M+K: Each line corresponds to a distance query and contains the indices of two farms.

Output

* Lines 1..K: For each distance query, output on a single line an integer giving the appropriate distance.

2) 算法设计

根据输入的农庄的信息，建立树的邻接表，并记录下所有需要查询的节点对。进行一遍离线 Tarjan 算法，计算所有查询的节点对的 LCA 和他们到 LCA 的距离。最后将这些节点对分别到 LCA 的距离相加，即得这些节点对之间的距离。

3) 程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <utility>
#include <algorithm>
#include <iostream>
#include <vector>
#include <map>

typedef long long ll;
using namespace std;
const int MAXN = 40000 + 10;
const int MAXK = 2 * 10000 + 10;
struct Edge {
	int to, w;
	Edge(int to, int w) : to(to), w(w) {}
};
struct Query {
	int u, v, next;
	ll d;
};
```

```
int n, m, k;
vector<Edge> G[MAXN];
Query Q[MAXK];
int head[MAXN];
int father[MAXN], ranks[MAXN], ancestor[MAXN];
bool visited[MAXN];
ll dist[MAXN];

inline void init() {
	memset(visited, false, sizeof(visited));
	memset(dist, 0, sizeof(dist));
	memset(head, -1, sizeof(head));
	for(int i = 1; i <= n; ++i) {
		father[i] = i;
		ranks[i] = 0;
		G[i].clear();
	}
}

inline void addQuery(int u, int v, int &m) {
	Q[m].u = u; Q[m].v = v; Q[m].d = 0; Q[m].next = head[u]; head[u] = m++;
	Q[m].u = v; Q[m].v = u; Q[m].d = 0; Q[m].next = head[v]; head[v] = m++;
}

int find(int u) {
	if(father[u] == u)
		return u;
	return father[u] = find(father[u]);
}

void Union(int u, int v) {
	u = find(u);
	v = find(v);
	if(u == v) return;
	if(ranks[u] < ranks[v]) father[u] = v;
	else {
		father[v] = u;
		if(ranks[u] == ranks[v]) ranks[u]++;
	}
}

void LCA(int u) {
	ancestor[u] = u;
	visited[u] = true;
```

```
    for(int i = 0; i < G[u].size(); ++i) {
        Edge &e = G[u][i];
        int v = e.to;
        if(!visited[v]) {
            dist[v] = dist[u] + e.w;
            LCA(v);
            Union(u, v);
            ancestor[find(u)] = u;
        }
    }
    for(int i = head[u]; i != -1; i = Q[i].next) {
        Query &query = Q[i];
        if(visited[query.v]) {
            int lca = ancestor[find(query.v)];
            query.d = (dist[query.u] - dist[lca]) + (dist[query.v] - dist[lca]);
        }
    }
}

int main() {
    std::ios::sync_with_stdio(false);
#ifdef LOCAL
    freopen("input.txt", "r", stdin);
#endif
    while(scanf("%d%d", &n, &m) == 2) {
        init();
        for (int i = 0; i < m; ++i) {
            int u, v, w;
            char d[2];
            scanf("%d%d%d%s", &u, &v, &w, d);
            G[u].push_back(Edge(v, w));
            G[v].push_back(Edge(u, w));
        }

        scanf("%d", &k);
        int m = 0;
        for (int i = 0; i < k; ++i) {
            int u, v;
            scanf("%d%d", &u, &v);
            addQuery(u, v, m);
        }

        LCA(1);
```

```
            for (int i = 0; i < k; ++i) {
                bool isolated = G[Q[2 * i].u].empty() || G[Q[2 * i].v].empty();
                if(isolated) Q[2 * i].d = dist[Q[2 * i].u] + dist[Q[2 * i].v];
                printf("%lld\n", Q[2 * i].d);
            }
        }
        return 0;
}
```

4) 性能分析

设树的顶点数为 n，查询数为 q，则程序的运行时间为 Tarjan 算法的运行时间，即为 O(n+q)，空间复杂度为 O(n).

5) 编程技术技巧

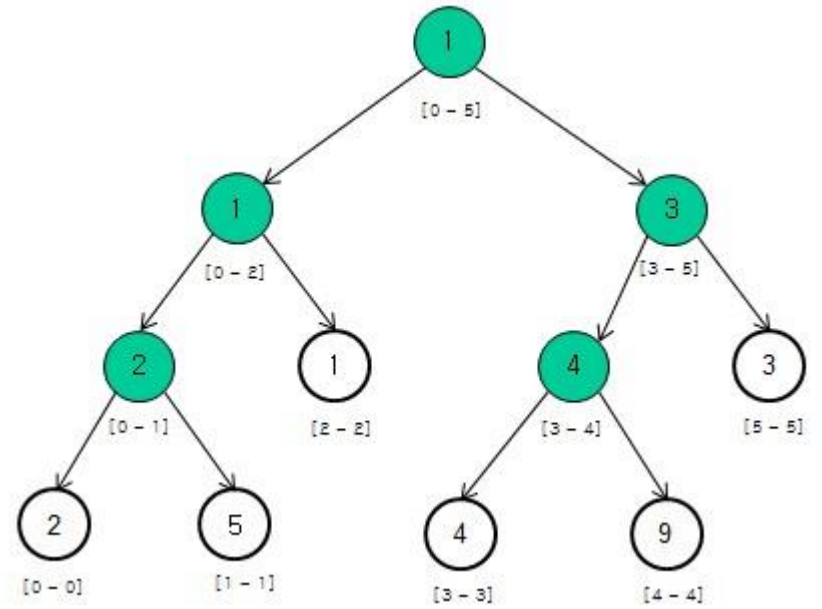
利用链式前向星保存查询的节点对信息。

```
inline void addQuery(int u, int v, int &m) {
    Q[m].u = u; Q[m].v = v; Q[m].d = 0; Q[m].next = head[u]; head[u] = m++;
    Q[m].u = v; Q[m].v = u; Q[m].d = 0; Q[m].next = head[v]; head[v] = m++;
}
```

线段树

一、 涉及问题

下面我们从一个经典的例子来了解线段树，问题描述如下:从数组 arr[0...n-1]中查找某个数组某个区间内的最小值，其中数组大小固定，但是数组中的元素的值可以随时更新。
对这个问题一个简单的解法是：遍历数组区间找到最小值，时间复杂度是 O(n),额外的空间复杂度 O(1)。当数据量特别大，而查询操作很频繁的时候，耗时可能会不满足需求。
另一种解法：使用一个二维数组来保存提前计算好的区间[i,j]内的最小值，那么预处理时间为 O(n^2)，查询耗时 O(1)，但是需要额外的 O(n^2)空间，当数据量很大时，这个空间消耗是庞大的，而且当改变了数组中的某一个值时，更新二维数组中的最小值也很麻烦。
我们可以用线段树来解决这个问题：预处理耗时 O(n)，查询、更新操作 O(logn)，需要额外的空间 O(n)。根据这个问题我们构造如下的二叉树
叶子节点是原始组数 arr 中的元素
非叶子节点代表它的所有子孙叶子节点所在区间的最小值
例如对于数组[2, 5, 1, 4, 9, 3]可以构造如下的二叉树（背景为白色表示叶子节点，非叶子节点的值是其对应数组区间内的最小值，例如根节点表示数组区间 arr[0...5]内的最小值。



由于线段树的父节点区间是平均分割到左右子树，因此线段树是完全二叉树，对于包含 n 个叶子节点的完全二叉树，它一定有 n-1 个非叶节点，总共 2n-1 个节点，因此存储线段是需要的空间复杂度是 O(n)。

二、 算法介绍

线段树的操作包括建树、对点的修改、对区间的修改和区间查询四个方面

建树：

```
//PushUp 函数更新节点信息 ，这里是求和

void PushUp(int rt){Sum[rt]=Sum[rt<<1]+Sum[rt<<1|1];}

//Build 函数建树     void Build(int l,int r,int rt){ //l,r 表示当前节点区间，rt 表示当前节点编号

    if(l==r)    {// 若 到 达 叶 节 点
Sum[rt]=A[l];//储存数组值
        return;

    }

    int m=(l+r)>>1;

    //左右递归

    Build(l,m,rt<<1);

    Build(m+1,r,rt<<1|1);

    //更新信息

    PushUp(rt);
```

```
}
```

对点的修改：假设修改 A[L]+=C

```
void Update(int L,int C,int l,int r,int rt){//l,r 表示当前节点区间，
rt
表示当前节点编号
    if(l==r){//到叶节点，修改
        Sum[rt]+=C;
return;
    }
    int m=(l+r)>>1;
    //根据条件判断往左子树调用还是往右
    if(L <= m) Update(L,C,l,m,rt<<1);
    else       Update(L,C,m+1,r,rt<<1|1);
    PushUp(rt);//子节点更新了，所以本节点
也需要更新信息
}
```

对区间的修改：假设 A[L,R]+=C

```
    void Update(int L,int R,int C,int l,int r,int rt){//L,R 表示操作区
    间，
l,r 表示当前节点区间，rt 表示当前节点编号
        if(L <= l && r <= R){//如果本区间完全在操作区间[L,R]以
内
            Sum[rt]+=C*(r-l+1);//更新数字和，向上保持正确
            Add[rt]+=C;//增加 Add 标记，表示本区间的 Sum 正
    确，子
区间的 Sum 仍需要根据 Add 的值来调整
            return ;
        }
        int m=(l+r)>>1;
        PushDown(rt,m-l+1,r-m);//下推标记
        //这里判断左右子树跟[L,R]有无交集，有交集才递归
        if(L <= m) Update(L,R,C,l,m,rt<<1);
    if(R >   m) Update(L,R,C,m+1,r,rt<<1|1);
        PushUp(rt);//更新本节点信息
```

```
}
```

区间查询：查询 A[L,R]的和

```
int Query(int L,int R,int l,int r,int rt){//L,R 表示操作区间，l,r 表示
```

```
当前节点区间，rt 表示当前节点编号

        if(L <= l && r <= R){

            //在区间内，直接返回

            return Sum[rt];

        }

        int m=(l+r)>>1;

        //下推标记，否则 Sum 可能不正确

        PushDown(rt,m-l+1,r-m);


        //累计答案

        int ANS=0;          if(L <= m)
    ANS+=Query(L,R,l,m,rt<<1);          if(R >
    m) ANS+=Query(L,R,m+1,r,rt<<1|1);
    return ANS;
    }
```

实际上线段树的功能不只是查询区间和，也可以查询区间最值等，可以根据每个非叶节点维护的信息的不同实现不同的功能。

三、 解题报告

1. POJ 2777

1）问题描述

Count Color

Time Limit: 1000MS Memory Limit: 65536K

Total Submissions: 48363 Accepted: 14605

Description

Chosen Problem Solving and Program design as an optional course, you are required to solve all kinds of problems. Here, we get a new problem.

There is a very long board with length L centimeter, L is a positive integer, so we can evenly divide the board into L segments, and they are labeled by 1, 2, ... L from left to right, each is 1 centimeter long. Now we have to color the board - one segment with only one color. We can do following two operations on the board:

1. "C A B C" Color the board from segment A to segment B with color C. 2. "P A B" Output the number of different colors painted between segment
A and segment B (including).

In our daily life, we have very few words to describe a color (red, green, blue, yellow…), so you may assume that the total number of different colors T is very small. To make it simple, we express the names of colors as color 1, color 2, ... color

T. At the beginning, the board was painted in color 1. Now the rest of problem is left to your.

Input

First line of input contains L (1 <= L <= 100000), T (1 <= T <= 30) and O (1 <= O <= 100000). Here O denotes the number of operations. Following O lines, each contains "C A B C" or "P A B" (here A, B, C are integers, and A may be larger than B) as an operation defined previously.

Output

Ouput results of the output operation in order, each line contains a number.

2）算法设计

根据一段木棒要么染了色要么未染色的二值特性，可以使用一个二进制位来表示每一段木棒，最终统计所有这些二进制位中为 1 的位数。利用一棵线段树更新这些二进制位。C l r x 表示将 l 到 r 的颜色更改为 x，p l r 表示询问 l 到 r 有多少种颜色。用线段树来存储下当前的每一段的颜色，更改用数组 lazy 标记，将颜色转化为为 2 进制数，统计一段颜色时，对每一段可以对它的左右子树取|操作，便可以统计这一段中颜色出现的种类。

3）程序代码

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

using namespace std;
const int MAX = 100000 + 10;
int colors[MAX << 2], setv[MAX << 2];
```

```
int l, t, o, a, b, c, _color;

void printb(int x)
{
	int a[32];
	for(int i = 0; i < 32; ++i)
	{
		a[i] = x & 1;
		x >>= 1;
	}
	for(int i = 31; i >=0; --i) printf("%d", a[i]);
}
void build()
{
	for (int i = 4 * l - 1; i >= 1; --i)
	{
		setv[i] = 0;
		colors[i] = 1 << 1;
	}
}
void maintain(int o, int L, int R)
{
	//colors[o] = 0;
	if (R > L)
	{
		colors[o] = colors[o * 2] | colors[o * 2 + 1];
	}
	if (setv[o] > 0)
		colors[o] = setv[o];
}
void pushdown(int o)
{
	if (setv[o] > 0)
	{
		setv[o * 2] = setv[o * 2 + 1] = setv[o];
		setv[o] = 0;
	}
}
void update(int o, int L, int R)
{
	int lc = o * 2, rc = o * 2 + 1;
	if (a <= L && b >= R)
	{
		setv[o] = 1 << c;
```

```
	}
	else
	{
		pushdown(o);
		int M = L + (R - L) / 2;
		if (a <= M)
			update(lc, L, M);
		else
			maintain(lc, L, M);
		if (b > M)
			update(rc, M + 1, R);
		else
			maintain(rc, M + 1, R);
	}
	maintain(o, L, R);
}
void query(int o, int L, int R)
{
	if (setv[o] > 0)
	{
		_color |= setv[o];
		// printf("%d %d ", L, R);
		// printb(setv[o]);
		// printf("\n");
	}
	else if (a <= L && b >= R)
	{
		_color |= colors[o];
		// printf("%d %d ", L, R);
		// printb(colors[o]);
		// printf("\n");
	}
	else
	{
		int M = L + (R - L) / 2;
		if (a <= M)
			query(o * 2, L, M);
		if (b > M)
			query(o * 2 + 1, M + 1, R);
	}
}

int main()
{
```

```
    ios::sync_with_stdio(false);
#ifdef LOCAL
    freopen("input.txt", "r", stdin);
#endif
    while (scanf("%d%d%d", &l, &t, &o) == 3)
    {
        build();
        for (int i = 0; i < o; ++i)
        {
            char op[2];
            scanf("%s", op);
            if (op[0] == 'C')
            {
                scanf("%d%d%d", &a, &b, &c);
                if(a > b) swap(a, b);
                update(1, 1, l);
            }
            else
            {
                scanf("%d%d", &a, &b);
                if(a > b) swap(a, b);
                _color = 0;
                query(1, 1, l);
                int cnt = 0;
                while (_color != 0)
                {
                    if (_color & 1)
                        cnt++;
                    _color >>= 1;
                }
                printf("%d\n", cnt);
            }
        }
    }
    return 0;
}
```

4）性能分析

设操作数为 n，则程序的时间复杂度为 O(nlog 30)，空间复杂度为 O(n)。

2. POJ 3667

1) 问题描述

Hotel

Time Limit: 3000MS Memory Limit: 65536K

Total Submissions: 18761 Accepted: 8169

Description

The cows are journeying north to Thunder Bay in Canada to gain cultural enrichment and enjoy a vacation on the sunny shores of Lake Superior. Bessie, ever the competent travel agent, has named the Bullmoose Hotel on famed Cumberland Street as their vacation residence.

This immense hotel has N ($1 \le N \le 50,000$) rooms all located on the same side of an extremely long hallway (all the better to see the lake, of course).

The cows and other visitors arrive in groups of size Di ($1 \le Di \le N$) and approach the front desk to check in. Each group irequests a set of Di contiguous rooms from Canmuu, the moose staffing the counter. He assigns them some set of consecutive room numbers r..r+Di-1 if they are available or, if no contiguous set of rooms is available, politely suggests alternate lodging. Canmuu always chooses the value of r to be the smallest possible.

Visitors also depart the hotel from groups of contiguous rooms. Checkout i has the parameters Xi and Di which specify the vacating of rooms Xi ..Xi +Di-1 ($1 \le Xi \le N-Di+1$). Some (or all) of those rooms might be empty before the checkout.

Your job is to assist Canmuu by processing M (1 ≤ M < 50,000)

checkin/checkout requests. The hotel is initially unoccupied.

Input

* Line 1: Two space-separated integers: N and M

* Lines 2..M+1: Line i+1 contains request expressed as one of two possible formats:

(a) Two space separated integers representing a check-in request: 1 and Di (b)

Three space-separated integers representing a check-out:
2, Xi, and Di

Output

* Lines 1.....: For each check-in request, output a single line with a single integer r, the first room in the contiguous sequence of rooms to be occupied. If the request cannot be satisfied, output 0.

2)算法设计

题目给出 N 个房间排列成一排，groups 需要 check in 房间，要求房间的编号为连续的 r..r+Di-1 并且 r 是最小的；visitors 同样可能 check out，并且他们每次 check out 都是编号为 Xi ..Xi +Di-1 (1 ≤ Xi ≤ N-Di+1)的房间，题目的输入有两种样式：

1 a : groups 需要 check in a 间编号连续的房间

2 a b : visitors check out 房间，其中房间编号是 a…a+b-1

要求对于每次 request，输出为 groups 分配数目为 a 的房间中编号最小的房间编号我们可以利用线段树建立模型，维护最大连续区间的长度。在线段树

上维护 3 个信息，包括左起连续，右起连续，以及中间连续的空房间个数。每次更新时判断左右两边以及中间的连续房间个数并进行相应操作。

3) 程序代码

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

using namespace std;
const int maxn = 5e4 + 6;
int n, m;
struct node {int setv, lsum, rsum, sum;} T[maxn * 4];

void pushup(int l, int r, int o)
{
	if(T[o].setv == 0)
	{
		T[o].lsum = T[o].rsum = T[o].sum = r - l + 1;
		return;
	}
	if (T[o].setv == 1)
	{
		T[o].lsum = T[o].rsum = T[o].sum = 0;
		return;
	}
	if(r > l) //非叶子节点
	{
		T[o].lsum = T[o * 2].lsum;
		T[o].rsum = T[o * 2 + 1].rsum;
		T[o].sum = max(T[o].lsum, T[o].rsum);
		int mid = (l + r) / 2;
		int len = r - l + 1;
		if (T[o].lsum == len - len / 2)
			T[o].lsum += T[o * 2 + 1].lsum;
		if (T[o].rsum == len / 2)
			T[o].rsum += T[o * 2].rsum;
		T[o].sum = max(T[o].sum, T[o * 2].rsum + T[o * 2 + 1].lsum);
	}
}
void pushdown(int o)
{
	if(T[o].setv != -1)
```

```
	{
		T[o * 2].setv = T[o * 2 + 1].setv = T[o].setv;
		T[o].setv = -1;
	}
}
void build(int l, int r, int o)
{
	T[o].setv = -1;
	if(l == r)
	{
		T[o].lsum = T[o].rsum = T[o].sum = 1;
		return;
	}
	int mid = (l + r) / 2;
	build(l, mid, o * 2);
	build(mid + 1, r, o * 2 + 1);
	pushup(l, r, o);
	//printf("%d %d %d\n", l, r, T[o].sum);
}
void update(int l, int r, int o, int x, int y, int v)
{
	//printf("%d %d\n", l, r);
	if(x <= l && r <= y)
	{
		T[o].setv = v;
		//printf("%d %d %d\n", l, r, v);
	}
	else
	{
		pushdown(o);
		int mid = (l + r) / 2;
		if(x <= mid) update(l, mid, o * 2, x, y, v);
		else pushup(l, mid, o * 2);
		if(mid < y) update(mid + 1, r, o * 2 + 1, x, y, v);
		else pushup(mid + 1, r, o * 2 + 1);
	}
	pushup(l, r, o);
	//printf("%d %d %d %d %d %d\n", l, r, T[o].sum, T[o].lsum, T[o].rsum, T[o].setv);
}
int query(int l, int r, int o, int len)
{
	//printf("%d %d %d    %d %d %d    %d %d %d\n", T[o].lsum, T[o].rsum, T[o].sum, T[o *
2].lsum, T[o * 2].rsum, T[o * 2].sum, T[o * 2 + 1].lsum, T[o * 2 + 1].rsum, T[o * 2 + 1].sum);
	if(T[o].sum < len) return 0;
```

```
    if(T[o].lsum >= len) return l;
    int mid = (l + r) / 2;
    if(T[o * 2].sum >= len) return query(l, mid, o * 2, len);
    else if(T[o * 2].rsum + T[o * 2 + 1].lsum >= len) return mid - T[o * 2].rsum + 1;
    else return query(mid + 1, r, o * 2 + 1, len);
}

int main()
{
    ios::sync_with_stdio(false);
#ifdef LOCAL
    freopen("input.txt", "r", stdin);
#endif
    while(scanf("%d%d", &n, &m) == 2)
    {
        build(1, n, 1);
        int op, x, d;
        for(int i = 1; i <= m; ++i)
        {
            scanf("%d", &op);
            if (op == 1)
            {
                scanf("%d", &d);
                x = query(1, n, 1, d);
                if (x != 0)
                    update(1, n, 1, x, x + d - 1, 1);
                printf("%d\n", x);
            }
            else
            {
                scanf("%d%d", &x, &d);
                update(1, n, 1, x, x + d - 1, 0);
            }
        }
    }
    return 0;
}
```

4) 性能分析

建树：O(n)
更新和查询：O(mlog n)，其中 n 为所有房间的数量，m 为订房和退房的次数。

总的时间复杂度为 O(mlog n)，空间复杂度为 O(n)。

差分约束系统

一、 涉及问题

若一个系统由 n 个变量以及 m 个约束条件组成，且每个约束条件有如下形式：

x[i] – x[j] <= a[k] (0<I,j<n.0<k<m)

则称该系统为一个有 n 个变量和 m 个约束条件的差分约束系统。

二、 算法介绍

差分约束系统可以转化为图，称为约束图，然后用求单源最短路径的算法求解——Bellman-Ford 算法。

差分约束系统利用了单源最短路径问题中的三角形不等式，即对任何一条边 u->v，都有：d(v) <= d(u) + w(u,v)，其中 d(u)和 d(v)是从源点分别到点 u 和点 v 的最短路径的权值，w(u,v)是边 u->b 的权值。

将不等式转化为图：d(v)-d(u)<=w(u,v)正好和差分约束系统中不等式形式相同。所以每一个未知数 x 对应图上的一个顶点 x，将所有的不等式都转化为图中的一条边。

Bellman-Ford 算法可以在最短路存在的情况下求出最短路，并且在存在负权圈的情况下告诉你最短路不存在，前提是起点能够到达这

个负权圈，因为即使图中有负权圈，但是起点到不了负权圈，最短路还是有可能存在的。它是基于这样一个事实：一个图的最短路如果存在，那么最短路中必定不存在圈，所以最短路的顶点数除了起点外最多只有 n-1 个。

Bellman-Ford 同样也是利用了最短路的最优子结构性质，用 d[i]表示起点 s 到 i 的最短路，那么边数上限为 j 的最短路可以通过边数上限为 j-1 的最短路 加入一条边 得到，通过 n-1 次迭代，最后求得 s 到所有点的最短路。

具体算法描述如下：对于图 G = <V, E>，源点为 s，d[i]表示 s 到 i 的最短路。

1) 初始化 所有顶点 d[i] = INF, 令 d[s] = 0，计数器 j = 0；

2) 枚举每条边(u, v)，如果 d[u]不等于 INF 并且 d[u] + w(u, v) < d[v]，则令 d[v] = d[u] + w(u, v)；

3) 计数器 j + +，当 j = n - 1 时算法结束，否则继续重复 2)的步骤；

第 2)步的一次更新称为边的“松弛”操作。

以上算法并没有考虑到负权圈的问题，如果存在负圈权，那么第 2)步操作的更新会永无止境，所以判定负权圈的算法也就出来了，只需要在第 n 次继续进行第 2)步的松弛操作，如果有至少一条边能够被更新，那么必定存在负权圈。

这个算法的时间复杂度为 O(nm)，n 为点数，m 为边数。

三、 解题报告

1. POJ 3159

1) 问题描述

Candies

Time Limit: 1500MS Memory Limit: 131072K

Total Submissions: 34394 Accepted: 9679

Description

During the kindergarten days, flymouse was the monitor of his class.

Occasionally the head-teacher brought the kids of flymouse’s class a large bag of candies and had flymouse distribute them. All the kids loved candies very much and often compared the numbers of candies they got with others. A kid A could had the idea that though it might be the case that another kid B was better than him in some aspect and therefore had a reason for deserving more candies than he did, he should never get a certain number of candies fewer than B did no matter how many candies he actually got, otherwise he would feel dissatisfied and go to the head-teacher to complain about flymouse’s biased distribution. snoopy shared class with flymouse at that time. flymouse always compared the number of his candies with that of snoopy’s. He wanted to make the difference between the numbers as large as possible while keeping every kid satisfied. Now he had just got another bag of candies from the head-teacher, what was the

largest difference he could make out

of it?

Input

The input contains a single test cases. The test cases starts with a line

with two integers N and M not exceeding 30 000 and 150 000 respectively. N is the number of kids in the class and the kids were numbered 1 through N. snoopy and flymouse were always numbered 1 and N. Then follow M lines each holding three integers A, B and c in order, meaning that kid A believed that kid B should never get over c candies more than he did.

Output

Output one line with only the largest difference desired. The difference is guaranteed to be finite.

2) 算法设计

题目给出了 N 个人和 M 个信息，每个信息形如 A B C 表示 B 比 A 多出的糖果不能超过 C 个，求 n 比 1 最多可以多多少个糖果。转化为最短路模型，有 B – A <= C。

在建图的时候将 A 和 B 连接一条有向边，边权值为 C，最后以 1 为起点，以 N 为终点运行 SPFA 算法即可。

3) 程序代码

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

```
#include <string.h>
#include <iostream>
#define INF 0x3f3f3f3f

using namespace std;
const int maxn = 30000 + 10;
const int maxm = 15e4 + 10;
struct Edge
{
	int v, w, next;
} E[maxm];
int head[maxn];		//链式前向星
int d[maxn];		//每个点的最短路径长
int Q[maxn];		//数组实现栈
bool inqueue[maxn]; //记录是否在栈中
int n, m, cnt, top;

inline void init()
{
	memset(head, -1, sizeof(head));
	memset(inqueue, false, sizeof(inqueue));
	memset(d, INF, sizeof(d));
	top = 0;
	cnt = 0;
}
inline void addEdge(int u, int v, int w)
{
	E[cnt].v = v;
	E[cnt].w = w;
	E[cnt].next = head[u];
	head[u] = cnt++;
}
void spfa(int s)
{
	Q[top++] = s;
	inqueue[s] = true;
	d[s] = 0;
	while (top > 0)
	{
		int u = Q[--top];
		inqueue[u] = false;
		for (int i = head[u]; i != -1; i = E[i].next)
		{
			int v = E[i].v;
```

```
                int temp = d[v];
                if(d[u] + E[i].w < d[v])
                    d[v] = d[u] + E[i].w;
                if (temp != d[v] && !inqueue[v])
                {
                    Q[top++] = v;
                    inqueue[v] = true;
                }
            }
        }
}
int main()
{
    ios::sync_with_stdio(false);
#ifdef LOCAL
    freopen("in.txt", "r", stdin);
#endif
    while (scanf("%d%d", &n, &m) == 2)
    {
        int u, v, w;
        init();
        for (int i = 1; i <= m; ++i)
        {
            scanf("%d%d%d", &u, &v, &w);
            addEdge(u, v, w);
        }
        spfa(1);
        printf("%d\n", d[n]);
    }
}
```

4) 性能分析

程序的运行时间即为 SPFA 的运行时间，即 O(kM)，k<=2，k 为一个很小的常数，M 为题目中给定的信息个数，空间复杂度为 O(n)，n 为小朋友个数。

2. POJ 1275

1) 问题描述

Cashier Employment

Time Limit: 1000MS Memory Limit: 10000K

Total Submissions: 8947 Accepted: 3468

Description

A supermarket in Tehran is open 24 hours a day every day and needs a number of cashiers to fit its need. The supermarket manager has hired you to help him, solve his problem. The problem is that the supermarket needs different number of cashiers at different times of each day (for example, a few cashiers after midnight, and many in the afternoon) to provide good service to its customers, and he wants to hire the least number of cashiers for this job.

The manager has provided you with the least number of cashiers needed for every one-hour slot of the day. This data is given as R(0), R(1), ..., R(23): R(0) represents the least number of cashiers needed from midnight to 1:00 A.M., R(1) shows this number for duration of 1:00 A.M. to 2:00 A.M., and so on. Note that these numbers are the same every day. There are N qualified applicants for this job. Each applicant i works non-stop once each 24 hours in a shift of exactly 8 hours starting from a specified hour, say ti (0 <= ti <= 23), exactly from the start of the hour mentioned. That is, if the ith applicant is hired, he/she will work starting from ti o'clock sharp for 8 hours. Cashiers do not replace one another and work exactly as scheduled, and there are enough cash registers and counters for those who are hired.

You are to write a program to read the R(i) 's for i=0..23 and ti 's for i=1..N that

are all, non-negative integer numbers and compute the least number of cashiers needed to be employed to meet the mentioned constraints. Note that there can be more cashiers than the least number needed for a specific
slot.

Input

The first line of input is the number of test cases for this problem (at most 20). Each test case starts with 24 integer numbers representing the R(0), R(1), ..., R(23) in one line (R(i) can be at most 1000). Then there is N, number of applicants in another line (0 <= N <= 1000), after which come N lines each containing one ti (0 <= ti <= 23). There are no blank lines between test cases.

Output

For each test case, the output should be written in one line, which is the least number of cashiers needed.

If there is no solution for the test case, you should write No Solution for that case.

2)算法设计

题目大意为 Tehran 的一家每天 24 小时营业的超市，需要一批出纳员来满足它的需要。超市经理雇佣你来帮他解决他的问题——超市在每天的不同时段需要不同数目的出纳员（例如：午夜时只需一小批，而下午则需要很多）来为顾客提供优质服务。他希望雇佣最少数目的

出纳员。

我们设 num[i] 为来应聘的在第 i 个小时开始工作的人数

r[i] 为第 i 个小时至少需要的人数　　x[i] 为招到的在第 i 个小时开始工作的人数　　根据题意有：

0 <= x[i] <= num[i]

x[i] + x[i-1] + …+ x[i-7] >= r[i]（题目中的连续工作 8 小时）

再设 s[i] = x[1] + … + x[i]　　则有：s[i] - s[i-1] >= 0

s[i-1] - s[i] >= -num[i]

s[i] - s[i-8] >= r[i], 8 <= i <= 24　　s[i] - s[i+16] >= r[i] - s[24], 1<= i <= 7

还需要添加一个隐藏不等式：s[24] - s[0] >= ans（枚举的答案）　　通过枚举 s[24],来检测是否满足条件，题目是求最小值，即求最长路，以 0 为源点。

3)程序代码

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#define INF 0x3f3f3f3f

using namespace std;
const int max = 1000 + 10;
struct Edge
{
    int u, v, w, next;
} E[25 * 3 + 1];
int head[25];
int r[25], t[25]; //r[i]为 i 时刻需要的出纳员的数目，t[i]为 i 时刻应征的申请者数目
int cases, n, cnt, top;
int Q[25], d[25], inqueue[25], count[25];

inline void init()
```

```
{
	memset(head, -1, sizeof(head));
	memset(d, INF, sizeof(d));
	memset(count, 0, sizeof(count));
	memset(inqueue, false, sizeof(inqueue));
	cnt = 0;
	top = 0;
}
inline void addEdge(int u, int v, int w)
{
	E[cnt].u = u;
	E[cnt].v = v;
	E[cnt].w = w;
	E[cnt].next = head[u];
	head[u] = cnt++;
}
void build(int sum)
{
	init();
	addEdge(0, 24, -sum);
	//printf("%d\n", -sum);
	for (int i = 1; i <= 24; ++i)
	{
		addEdge(i - 1, i, 0);
		addEdge(i, i - 1, t[i]);
		//printf("%d %d %d\n", i, i - 1, t[i]);
	}
	//printf("\n");
	for (int i = 1; i <= 16; ++i)
	{
		addEdge(i, i + 8, -r[i + 8]);
		//printf("%d %d %d\n",i, i + 8, -r[i + 8]);
	}
	//printf("\n");
	for (int i = 17; i <= 24; ++i)
	{
		addEdge(i, i - 16, -r[i - 16] + sum);
		//printf("%d %d %d\n",i, i - 16, -r[i - 16] + sum);
	}
	//printf("\n");
}
void spfa(int s, bool &hasCycle)
{
	Q[top++] = s;
```

```
	inqueue[s] = true;
	d[s] = 0;
	while (top > 0)
	{
		int u = Q[--top];
		inqueue[u] = false;
		for (int i = head[u]; i != -1; i = E[i].next)
		{
			int v = E[i].v;
			int temp = d[v];
			if (d[u] + E[i].w < d[v])
				d[v] = d[u] + E[i].w;
			if (temp != d[v] && !inqueue[v])
			{
				count[v]++;
				if(count[v] > 24)
				{
					hasCycle = true;
					return;
				}
				Q[top++] = v;
				inqueue[v] = true;
			}
		}
	}
}
bool isFeasible(int sum)
{
	bool hasCycle = false;
	build(sum);
	spfa(0, hasCycle);
	if (!hasCycle && d[24] == -sum)
		return true;
	return false;
}
int main()
{
	ios::sync_with_stdio(false);
#ifdef LOCAL
	freopen("in.txt", "r", stdin);
#endif
	while (scanf("%d", &cases) == 1)
	{
		while (cases--)
```

```
        {
            memset(t, 0, sizeof(t));
            bool feasible = false;
            for (int i = 1; i <= 24; ++i)
                scanf("%d", &r[i]);
            scanf("%d", &n);
            for (int i = 1; i <= n; ++i)
            {
                int a;
                scanf("%d", &a);
                t[a + 1]++;
            }
            int low = 0, high = n, mid;
            while (low < high)
            {
                mid = (low + high) / 2;
                if (isFeasible(mid))
                    high = mid;
                else
                    low = mid + 1;
            }
            if (low == n && !isFeasible(high))
                printf("No Solution\n");
            else
                printf("%d\n", low);
        }
    }
    return 0;
}
```

4) 性能分析

程序利用二分法进行枚举，每次枚举运行 spfa 算法，则算法的时间复杂度为 O(24*log n)，空间复杂度为 O(1)。

总结

这次的算法实践课程让我能够如同一名真正的 ACM 选手一样系统地进行竞赛算法的学习与练习，体会到了不断提升算法性能和编程技巧以减少程序运行时间、在尽可能短的时间内解决特定的问题，体会到优秀的算法和程序带来的性能的优越性。

更难得的是，通过这门课程，我还能将在算法理论课上学习到的算法理论知识很好地应用于编程实践，体会从算法理论过渡到具体编程实现上所要付出的努力。同时，我还体会到，面对一个需要利用计算机解决的现实问题，不能未经思考就做出鲁莽的编程实践，而需要经过充分的思考，建立合适的模型，考虑合适的数据结构和算法，恰当地分析时间复杂度和空间复杂度，利用给定的资源（时间和空间）正确地解决问题，并利用尽可能好的方式实现算法。在这一过程中，无论是算法的设计过程、算法的实现过程和最终程序的调试过程，都可能花费我们大量的时间，甚至让人望而却步，但是必须要有足够的耐心和洞察力来克服所有这些困难，才能够最终获得性能让人满意的程序。

在本次实践中，我共完成了 24 道题目中的 22 道。Poj 账号为 huieric，题目清单列举如下。

专题一：并查集 4/4

- √ POJ1182 食物链

2628680908	1182	Accepted	2616K	266MS	C++	1665B

- √ POJ1733 Parity game

2628680908	1733	Accepted	492K	157MS	C++	1352B

- √ POJ1417 True Liars

2628680908	1417	Accepted	324K	0MS	C++	3027B

- √ POJ2912 Rochambeau

2628680908	2912	Accepted	304K	735MS	C++	1986B

专题二：树状数组 4/4

- √ POJ3321 Apple Tree

2628680908	3321	Accepted	5680K	438MS	C++	2105B

- √ POJ1990 MooFest

2628680908	1990	Accepted	520K	79MS	C++	1631B

- √ POJ2892 Tunnel Warfare

2628680908	2892	Accepted	960K	407MS	C++	2317B

- √ POJ2481 Cows

2628680908	2481	Accepted	2124K	1094MS	C++	1292B

专题三：后缀数组 2/4

- √ POJ3294 Life Forms

2628680908	3294	Accepted	3448K	1110MS	C++	3812B

- √ POJ3415 Common Substrings

2628680908	3415	Accepted	6696K	1844MS	C++	3110B

专题四：LCA/RMQ 4/4

- √ POJ1470 Closet Common Ancestors

2628680908	1470	Accepted	3008K	610MS	C++	1715B

- √ POJ1986 Distance Queries

2628680908	1986	Accepted	4232K	172MS	C++	2072B

- √ POJ3728 The merchant

2628680908	3728	Accepted	14032K	2750MS	C++	1692B

专题五：线段树 4/4

- √ POJ2777 Count Color

2628680908	2777	Accepted	2256K	313MS	C++	3481B

- √ POJ2761 Feed the dogs

2628680908	2761	Accepted	3932K	3969MS	C++	2112B

- √ POJ3667 Hotel

2628680908	3667	Accepted	3280K	625MS	C++	2857B

专题六：差分约束系统 4/4

- √ POJ3159 Candies

2628680908	3159	Accepted	2104K	719MS	C++	1262B

- √ POJ1275 Cashier Employment

2628680908	1275	Accepted	216K	16MS	C++	1950B

- √ POJ1201 Intervals

2628680908	1201	Accepted	2624K	282MS	C++	1291B

- √ POJ3169 Layout

2628680908	3169	Accepted	360K	110MS	C++	1514B