

ACM-ICPC-Template

produced by MenYifan



The latest update: 2016.10.10

Table of Contents

| | |
|----|-----|
| 简介 | 1.1 |
|----|-----|

Setting

| | |
|------|-----|
| 头文件 | 2.1 |
| 快速输入 | 2.2 |
| 快速输出 | 2.3 |

PART I - 基本算法

| | |
|------|-----|
| 折半查找 | 3.1 |
| 二分计算 | 3.2 |
| 三分计算 | 3.3 |
| 离散化 | 3.4 |
| 归并排序 | 3.5 |

PART II - 博弈

| | |
|-----------|-----|
| Bash博弈 | 4.1 |
| Wythoff博弈 | 4.2 |
| Nimm博弈 | 4.3 |
| SG函数 | 4.4 |

PART III - 字符串

| | |
|----------|-----|
| KMP | 5.1 |
| Trie前缀树 | 5.2 |
| AC自动机 | 5.3 |
| 后缀数组 | 5.4 |
| Manacher | 5.5 |
| 回文树 | 5.6 |

PART IV - 数据结构

| | |
|--------|-------|
| 并查集 | 6.1 |
| 树状数组 | 6.2 |
| 树状数组 | 6.2.1 |
| 二维树状数组 | 6.2.2 |
| 三维树状数组 | 6.2.3 |
| 线段树 | 6.3 |
| 单点更新 | 6.3.1 |
| 区间更新 | 6.3.2 |
| RMQ | 6.3.3 |
| 树链剖分 | 6.4 |
| 主席树 | 6.5 |

PART V - 动态规划

| | |
|----------------|-------|
| 背包问题 | 7.1 |
| ○一背包 | 7.1.1 |
| 完全背包 | 7.1.2 |
| 最大子段和 | 7.2 |
| 最长上升子序列 | 7.3 |
| 最大公共子序列 | 7.4 |
| 最长公共上升子序列 | 7.5 |
| 矩阵链乘 | 7.6 |
| Sparse－Table算法 | 7.7 |
| 数位DP | 7.8 |

PART VI - 数论

| | |
|--------------|-----|
| 快速幂&乘 | 8.1 |
| 线性筛素数表 | 8.2 |
| Mobius反演 | 8.3 |
| 小型素数判断 | 8.4 |
| Miller_Rabin | 8.5 |
| Ex_gcd | 8.6 |
| 中国剩余定理 | 8.7 |
| 单变元模线性方程 | 8.8 |

| | |
|-------------|--------|
| 逆元 | 8.9 |
| 组合数打表 | 8.10 |
| 逆元求组合数 | 8.11 |
| Lucas定理求组合数 | 8.12 |
| 欧拉函数 | 8.13 |
| 高斯消元 | 8.14 |
| 高斯消元－整数 | 8.14.1 |
| 高斯消元－浮点 | 8.14.2 |
| 高斯消元－异或 | 8.14.3 |
| FFT多项式乘法 | 8.15 |
| Simpson | 8.16 |

PART VII - 图论

| | |
|---------------|-------|
| 拓扑排序 | 9.1 |
| 最短路 | 9.2 |
| Dijkstra | 9.2.1 |
| Dijkstra_Heap | 9.2.2 |
| Bellman-Ford | 9.2.3 |
| Floyd | 9.2.4 |
| SPFA | 9.2.5 |
| 最小生成树 | 9.3 |
| Prim | 9.3.1 |
| Prim_Heap | 9.3.2 |
| Kruskal | 9.3.3 |
| 次小生成树 | 9.4 |
| 最小树形图 | 9.5 |
| LCA | 9.6 |
| LCA -> RMQ | 9.6.1 |
| Tarjan | 9.6.2 |
| 倍增算法 | 9.6.3 |
| 二分图 | 9.7 |
| 匈牙利算法 | 9.7.1 |
| Hopcroft-Karp | 9.7.2 |

| | |
|-------------|-------|
| 网络流 | 9.8 |
| EdmondsKarp | 9.8.1 |
| Dinic | 9.8.2 |
| ISAP | 9.8.3 |
| 最小费用最大流 | 9.8.4 |
| 舞蹈链 | 9.9 |
| 无向图的割顶和割桥 | 9.10 |
| 无向图的双连通分量 | 9.11 |
| 有向图的强连通分量 | 9.12 |
| 2-SAT | 9.13 |
| 交叉染色 | 9.14 |

PART VIII - 其他

| | |
|-----------|-------|
| JAVA输入挂 | 10.1 |
| JAVA大数类 | 10.2 |
| 高精度计算-大数类 | 10.3 |
| double的比较 | 10.4 |
| 矩阵类 | 10.5 |
| 计算几何 | 10.6 |
| STL | 10.7 |
| sscanf | 10.8 |
| sprintf | 10.9 |
| 逆序数 | 10.10 |
| RMQ问题 | 10.11 |

END

| | |
|---------------|------|
| To Do List | 11.1 |
| To Learn List | 11.2 |

简介

鉴于本人水平太渣，估计也没什么人看这个，所以这个省略了，等以后出名了再回来补。

不出名？不出名补简介干什么？

推荐几个ACM模版

- [范神的模版/ACM-ICPC-Template](#)
- [超级神犇Blog](#)
- [金海峰的GitHub](#)
- [神犇一号GitBook](#)
- [神犇二号GitBook](#)
- [神犇三号GitBook](#)
- [神犇四号GitBook](#)
- [神犇五号GitHub](#)里面有一个regional.rar，还有更多神牛的板子

头文件

```
#include <set>
#include <map>
#include <stack>
#include <cmath>
#include <queue>
#include <cstdio>
#include <string>
#include <vector>
#include <iomanip>
#include <bitset>
#include <cstring>
#include <iostream>
#include <iostream>
#define Memset(a, val) memset(a, val, sizeof(a))
#define PI acos(-1.0)
#define PB push_back
#define MP make_pair
#define rt(n)          (i == n ? '\n' : ' ')
#define hi             printf("Hi-----\n")
#define IN freopen("input.txt", "r", stdin);
#define OUT freopen("output.txt", "w", stdout);
#define debug(x) cout<<"Debug : ---"<<x<<"---"<<endl;
#define debug2(x,y) cout<<"Debug : ---"<<x<<" , "<<y<<"---"<<endl;
#pragma comment(linker, "/STACK:1024000000,1024000000")
using namespace std;
typedef pair<int,int> PII;
typedef long long ll;
const int maxn=100000+5;
const int mod=1000000007;
const int INF=0x3f3f3f3f;
const double eps=1e-8;
```

输入

使用方法

- `read(x);` 读取整数 `x`

Tips

- 只能读取整数，包括 `int` , `long long` , `unsigned int` , `unsigned long long`

模版

```
template <class T>
inline bool Read(T &ret) {
    char c; int sgn;
    if(c=getchar(),c==EOF) return 0; //EOF
    while(c!='-'&&(c<'0'||c>'9')) c=getchar();
    sgn=(c=='-') ?-1:1 ;
    ret=(c=='-') ?0:(c-'0');
    while(c=getchar(),c>='0'&&c<='9')
        ret=ret*10+(c-'0');
    ret*=sgn;
    return 1;
}
```

输出

使用方法

- `Out(x);` 输出 `x`

Tips

- `Out(x);` 等价于 `printf("%d", x);`
- 需要自己手动换行 `puts("");`
- 仅限于 `int` 类型变量

模版

```
void Out(int a)
{
    //输出外挂
    if(a < 0)
    {
        putchar('-');
        a = -a;
    }
    if(a >= 10)
        Out(a / 10);
    putchar(a % 10 + '0');
}
```

折半查找

说明

使用二分的思想进行查找

array为被查找数组 l为区间左端点 r为区间右端点 v为目标值

查询时间复杂度O(logn)

使用方法

1. 自己定义 f() 函数
2. binary_search(l, r); 其中 l, r 为对应区间的左右端点值

Tips

- 此模版是求 [l ,r] 闭区间的，而不是求 [l, r) 开区间的
- 推荐使用 lower_bound() 和 upper_bound()

模版

```
int binary_search(int array[], int l,int r, int v)
{
    int left, right, middle;
    left = l, right = r;
    while (left <= right) //此处应注意
    {
        middle = (left + right) / 2;
        if (array[middle] > v)
            right = middle - 1;
        else if (array[middle] < v)
            left = middle + 1;
        else
            return middle; //此处应注意
    }
    return -1;
}
```

```
//金海峰
LL binary_search(LL start, LL end, LL a)
{
    LL l = start;
    LL r = end;
    while (l < r)
    {
        LL mid = (l + r) / 2;
        if (ok(mid, a))
            r = mid;
        else
            l = mid + 1;
    }
    return l;
}
```

二分计算

说明

二分法一般用于计算区间内零点

使用方法

1. 自己定义 `f()` 函数
2. `Binary_Calculate(l, r);` 其中 `l, r` 为对应区间的左右端点值

Tips

- 该模版用于单调递增函数零点，单调递减请改符号

模版

```
double f(double theta) {
    return blablabla;
}

const double eps=1e-8;
double Binary_Calculate(double l, double r) {
    double mid;
    while(fabs(r-l)>eps) {
        mid=(l+r)/2;
        if (f(mid)>0)
            r=mid;
        else
            l=mid;
    }
}
```

三分计算

说明

三分法一般用于计算区间内极值

使用方法

1. 自己定义 `f()` 函数
2. `Ternary_Calculate(l, r);` 其中 `l, r` 为对应区间的左右端点值

Tips

- 该模版用于求极大值，求极小值请改符号

模版

```
double f(double theta) {
    return blablabla;
}

const double eps=1e-8;
double Ternary_Calculate(double l, double r) {
    double mid,midmid,ans;
    while (fabs(r-l)>eps) {
        mid=(l+r)/2;
        midmid=(mid+r)/2;
        if (f(mid)<f(midmid))      //求极大值
            l=mid;
        else
            r=midmid;
    }
    ans=f(l);
    return ans;
}
```

离散化

说明

`num[]` 表示离散化之前的数组，下标从`1~n`

`a[]` 表示离散化之后的数组，下标从`1~n`

`lsh[]` 表示离散化的缓存数组，下标为`1~cnt`

函数返回值为 `cnt`

使用方法

1. 输入`n`
2. `for(i: 1->n) scanf(num[i]);`
3. `ll cnt = discrete()`

Tips

- 考虑内存问题，会爆内存的话换 `int`
- 需要加 `algorithm` 和 `cstring` 头文件

模版

版本一

说明: 此方法为完整的离散化，表示不考虑重复数字`num[i]`为第 `a[i]` 大的数

`num`数组: `1 5 5 5 100`

`a`数组: `1 2 2 2 3`

```

ll num[maxn], lsh[maxn], a[maxn], n;
ll discrete () {
    memcpy(lsh, num, sizeof(lsh));
    stable_sort(lsh+1, lsh+1+n);
    ll cnt=unique(lsh+1, lsh+1+n)-lsh-1;
    for (ll i=1; i<=n; i++)
        a[i]=lower_bound(lsh+1, lsh+cnt+1, num[i])-lsh;
    return cnt;
}

```

版本二

说明: 此方法为特殊的离散化, 表示考虑重复数字num[i]为第 a[i] 大的数

num数组: 1 100 5 5 5

a数组: 1 5 2 2 2

```

ll num[maxn], lsh[maxn], a[maxn], n;
ll discrete () {
    memcpy(lsh, num, sizeof(lsh));
    stable_sort(lsh+1, lsh+1+n);
    ll cnt=n;
    for (ll i=1; i<=n; i++)
        a[i]=lower_bound(lsh+1, lsh+cnt+1, num[i])-lsh;
    return cnt;
}

```

归并排序

说明

通过递归的方式进行排序

时间复杂度: $O(n \log n)$

参数说明:

$A[]$ 为原数组, 归并排序对 $[l, r]$ 区间进行排序,

$T[]$ 为辅助数组, 需要定义这样一个数组, 作为参数传进去

使用方法

1. 定义辅助数组 $T[]$, 求逆序数的话需要将 cnt 置零
2. $\text{merge_sort}(A, l, r, T);$ l 为区间左端点, r 为区间右端点的下一个点
3. 数组 A 中 $[l, r]$ 已排好序

Tips

- 注意边界范围是 $[l, r)$
- 如果求解逆序数, 需要保证 cnt 初始化为 0

模版

归并排序

```
//归并排序
void merge_sort(int *A,int l,int r,int *T) {
    if (r-l>1) {
        int m=l+(r-l)/2;                                //划分
        int p=l,q=m,i=l;
        merge_sort(A, l, m, T);                         //递归求解
        merge_sort(A, m, r, T);                         //递归求解
        while (p<m||q<r)
            if (q>=r|| (p<m&&A[p]<=A[q]))
                T[i++]=A[p++];                          //从左半数组复制到临时空间
            else
                T[i++]=A[q++];                          //从右半数组复制到临时空间
        for (i=l; i<r; i++)
            A[i]=T[i];                                //从辅助空间复制回A数组
    }
}
```

```
//归并排序求逆序数
void merge_sort_inversion(int *A,int l,int r,int *T,int *cnt) {
    if (r-l>1) {
        int m=l+(r-l)/2;                                //划分
        int p=l,q=m,i=l;
        merge_sort_inversion(A, l, m, T,cnt);           //递归求解
        merge_sort_inversion(A, m, r, T,cnt);           //递归求解
        while (p<m||q<r)
            if (q>=r|| (p<m&&A[p]<=A[q]))
                T[i++]=A[p++];                          //从左半数组复制到临时空间
            else{
                T[i++]=A[q++];                          //从右半数组复制到临时空间
                *cnt += m-p;
            }
        for (i=l; i<r; i++)
            A[i]=T[i];                                //从辅助空间复制回A数组
    }
}
```

巴什博弈

说明

有一堆石子，石子个数为n，两人轮流从石子中取石子出来，最少取一个，最多取m个。最后不能取的人输，问你最后的输赢情况。

结果：当 $n \%(m+1) == 0$ 时，先手输，否则先手赢

这种就是可以直接判断必败态的问题。如果这堆石子少于或者等于m个，那么先手赢。如果石子数目为m+1个，那么先手必输，因为无论先手怎么拿石子，后手都可以直接把剩下的石子全部拿走。如果石子数目为 $m+1 < n < 2(m+1)$,那么先手就可以拿走 $n-(m+1)$ 个石子，使得对手面对m+1的必败态，这样先手必赢。所以如此推算下去，我们就知道当 $n \%(m+1) == 0$ 时，先手输，否则先手赢。

使用方法

```
int ans = bash(n, m); n为总棋子数目，m为单次可取的最大石子数量，返回值为0表示先手输，返回1表示先手赢。
```

模版

```
int bash(int n, int m)
{
    if (n % (m + 1) != 0)
        return 1;
    else
        return 0;
}
```

Wythoff博奕

说明

有两堆石子，石子数目分别为n和m,现在两个人轮流从两堆石子中拿石子，每次拿时可以从一堆石子中拿走若干个，也可以从两中拿走相同数量的石子，拿走最后一刻石子的人赢。

结论：如果当前局势未奇异局势则先手必败，否则先手必胜。奇异局势的判断公式为： $ak = \lfloor k(1+\sqrt{5})/2 \rfloor$, $bk = ak + k$ ($k=0, 1, 2, \dots n$ 方括号表示取整函数)

可以发现，前面几组奇异局势为 $(a, b) : (0, 0)、(1, 2)、(3, 5)、(4, 7)、(6, 10)、(8, 13)、(9, 15)、(11, 18)、(12, 20)$

这其中第k组的a为前面没有出现过的最小非负整数，而 $b = a + k$

两个人如果都采用正确操作，那么面对非奇异局势，先拿者必胜；反之，则后拿者取胜。那么任给一个局势 (a, b) ，怎样判断它是不是奇异局势呢？我们有如下公式： $ak = \lfloor k(1+\sqrt{5})/2 \rfloor$, $bk = ak + k$ ($k=0, 1, 2, \dots n$ 方括号表示取整函数)

使用方法

```
int ans = wzf(n, m); n和m分别为两堆石子的个数，如果是奇异局势（必败态）则返回0，否则返回1
```

模版

```
int wzf(int n, int m)
{
    if(n > m)
        swap(n, m);
    int k = m-n;
    int a = (k * (1.0 + sqrt(5.0)) / 2.0);
    if(a == n)
        return 0;
    else
        return 1;
}
```

例题

题目

同样是取石头问题，不过如果先手能赢，要输出第一次取石子后所剩的情况。（如果在任意的一堆中取走石子能胜同时在两堆中同时取走相同数量的石子也能胜，先输出取走相同数量的石子的情况）这道题目要输出威佐夫博奕从非奇异局势到奇异局势的转变方案。比直接判断复杂了很多。

思路

1. 先判断能否从两堆中取出相同的石头，达到目的。
2. 判断从一堆中取出石头，这里其实可以分为三种。假设一开始的时候为 (a, b) 。第一种是从 b 当中取走少量的，变为 $(a, b - k)$ 。第二种是从 b 中取走大量的，变为 $(b - t, a)$ 。第三种是从 a 当中取走一些，变为 $(a - t, b)$

代码

```

int main () {
    int a, b;
    while(scanf("%d%d", &a, &b)) {
        if (a == 0 && b == 0) break;
        int k = b - a;
        int n = (1 + sqrt(5.0)) / 2 * k;
        if (a == n) puts("0");
        else {
            puts("1");
            if (a - n == b - n - k) printf("%d %d\n", n, n + k);
            //两者之间的差值不变
            if (a == 0) puts("0 0");      //如果a等于0了，差值只能为0，即
为 (0, 0)。
            for (int i = 1; i <= b; i++) { //枚举a和b之间的差值，可能从1
一直到b - 1
                n = (1 + sqrt(5.0)) / 2 * i;
                int tmp = n + i;
                if (n == a) printf("%d %d\n", n, tmp);
                if (tmp == a) printf("%d %d\n", n, tmp);
                if (tmp == b) printf("%d %d\n", n, b);
            }
        }
    }
    return 0;
}

```


Nimm博弈

说明

有三堆各若干个物品，两个人轮流从某一堆取任意多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

结论：当每堆石头取异或之后，不为0代表着必胜态。为0代表着必败态。

证明：

1. 最后的状态，全为零，显然成立；
2. 对于某个局面 (a_1, a_2, \dots, a_n) ，若 $a_1 \wedge a_2 \wedge \dots \wedge a_n > 0$ ，一定存在某个合法的移动，将 a_i 改变成 a'_i 后满足 $a_1 \wedge a_2 \wedge \dots \wedge a'_i \wedge \dots \wedge a_n = 0$ 。不妨设 $a_1 \wedge a_2 \wedge \dots \wedge a_n = k$ ，则一定存在某个 a_i ，它的二进制表示在 k 的最高位上是1（否则 k 的最高位那个1是怎么得到的）。这时 $a_i \wedge k < a_i$ 一定成立。则我们可以将 a_i 改变成 $a'_i = a_i \wedge k$ ，此时 $a_1 \wedge a_2 \wedge \dots \wedge a'_i \wedge \dots \wedge a_n = a_1 \wedge a_2 \wedge \dots \wedge a_n \wedge k = 0$ 。
3. 对于某个局面 (a_1, a_2, \dots, a_n) ，若 $a_1 \wedge a_2 \wedge \dots \wedge a_n = 0$ ，一定不存在某个合法的移动，将 a_i 改变成 a'_i 后满足 $a_1 \wedge a_2 \wedge \dots \wedge a'_i \wedge \dots \wedge a_n = 0$ 。因为异或运算满足消去率，由 $a_1 \wedge a_2 \wedge \dots \wedge a_n = a_1 \wedge a_2 \wedge \dots \wedge a'_i \wedge \dots \wedge a_n$ 可以得到 $a_i = a'_i$ 。所以将 a_i 改变成 a'_i 不是一个合法的移动。证毕。

解题

每一堆的石子的数量的异或和不为零时先手必胜，否则先手必败。

SG函数

本文转自：[组合游戏 - SG函数和SG定理](#)

在介绍SG函数和SG定理之前我们先介绍介绍必胜点与必败点吧。

必胜点和必败点的概念：

P点：必败点，换而言之，就是谁处于此位置，则在双方操作正确的情况下必败。

N点：必胜点，处于此情况下，双方操作均正确的情况下必胜。

必胜点和必败点的性质：

1、所有终结点是必败点 P。（我们以此为基本前提进行推理，换句话说，我们以此为假设）

2、从任何必胜点N 操作，至少有一种方式可以进入必败点 P。

3、无论如何操作，必败点P 都只能进入 必胜点 N。

我们研究必胜点和必败点的目的时间为题进行简化，有助于我们的分析。通常我们分析必胜点和必败点都是以终结点进行逆序分析。我们以[hdu 1847 Good Luck in CET-4 Everybody!](#)为例：

当 $n = 0$ 时，显然为必败点，因为此时你已经无法进行操作了

当 $n = 1$ 时，因为你一次就可以拿完所有牌，故此时为必胜点

当 $n = 2$ 时，也是一次就可以拿完，故此时为必胜点

当 $n = 3$ 时，要么就是剩一张要么剩两张，无论怎么取对方都将面对必胜点，故这一点为必败点。

以此类推，最后你就可以得到；

| | | | | | | | | |
|-----------|---|---|---|---|---|---|---|-----|
| n : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
| position: | P | N | N | P | N | N | P | ... |

你发现了什么没有，对，他们就是成有规律，使用了 P/N 来分析，有没有觉得问题变简单了。

现在给你一个稍微复杂一点点的：[hdu 2147 kiki's game](#)

现在我们就来介绍今天的主角吧。组合游戏的和通常是很复杂的，但是有一种新工具，可以使组合问题变得简单——SG函数和SG定理。

Sprague-Grundy定理 (SG定理) :

游戏和的SG函数等于各个游戏SG函数的Nim和。这样就可以将每一个子游戏分而治之，从而简化了问题。而Bouton定理就是Sprague-Grundy定理在Nim游戏中的直接应用，因为单堆的Nim游戏 SG函数满足 $SG(x) = x$ 。不知道Nim游戏的请移步：[这里](#)

SG函数：

首先定义mex(minimal excludant)运算，这是施加于一个集合的运算，表示最小的不属于这个集合的非负整数。例如 $mex\{0,1,2,4\}=3$ 、 $mex\{2,3,5\}=0$ 、 $mex\{\}=0$ 。

对于任意状态 x ，定义 $SG(x) = mex(S)$, 其中 S 是 x 后继状态的SG函数值的集合。如 x 有三个后继状态分别为 $SG(a), SG(b), SG(c)$ ，那么 $SG(x) = mex\{SG(a), SG(b), SG(c)\}$ 。这样集合 S 的终态必然是空集，所以SG函数的终态为 $SG(x) = 0$, 当且仅当 x 为必败点P时。

【实例】取石子问题

有1堆n个的石子，每次只能取{ 1, 3, 4 }个石子，先取完石子者胜利，那么各个数的SG值为多少？

$SG[0]=0, f[]=\{1,3,4\}$,

$x=1$ 时, 可以取走 $1 - f[1]$ 个石子, 剩余 $\{0\}$ 个, 所以 $SG[1] = \text{mex}\{ SG[0] \} = \text{mex}\{0\} = 1$;

$x=2$ 时, 可以取走 $2 - f[1]$ 个石子, 剩余 $\{1\}$ 个, 所以 $SG[2] = \text{mex}\{ SG[1] \} = \text{mex}\{1\} = 0$;

$x=3$ 时, 可以取走 $3 - f[1,3]$ 个石子, 剩余 $\{2,0\}$ 个, 所以 $SG[3] = \text{mex}\{SG[2], SG[0]\} = \text{mex}\{0,0\} = 1$;

$x=4$ 时, 可以取走 $4 - f[1,3,4]$ 个石子, 剩余 $\{3,1,0\}$ 个, 所以 $SG[4] = \text{mex}\{SG[3], SG[1], SG[0]\} = \text{mex}\{1,1,0\} = 2$;

$x=5$ 时, 可以取走 $5 - f[1,3,4]$ 个石子, 剩余 $\{4,2,1\}$ 个, 所以 $SG[5] = \text{mex}\{SG[4], SG[2], SG[1]\} = \text{mex}\{2,0,1\} = 3$;

以此类推.....

$x \quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \dots$

$SG[x] \quad 0 \ 1 \ 0 \ 1 \ 2 \ 3 \ 2 \ 0 \ 1 \dots$

由上述实例我们就可以得到SG函数值求解步骤, 那么计算 $1 \sim n$ 的SG函数值步骤如下:

- 1、使用 数组 f 将 可改变当前状态 的方式记录下来。
- 2、然后我们使用 另一个数组 将当前状态 x 的后继状态标记。
- 3、最后模拟mex运算, 也就是我们在标记值中 搜索 未被标记值 的最小值, 将其赋值给 $SG(x)$ 。
- 4、我们不断的重复 2 - 3 的步骤, 就完成了 计算 $1 \sim n$ 的函数值。

代码实现如下:

```

//f[N] :可改变当前状态的方式, N为方式的种类, f[N]要在getSG之前先预处理
//SG[] :0~n的SG函数值
//S[] :为后继状态的集合
int f[N], SG[MAXN], S[MAXN];
void getSG(int n) {
    int i, j;
    memset(SG, 0, sizeof(SG));
    //因为SG[0]始终等于0, 所以i从1开始
    for(i = 1; i <= n; i++) {
        //每一次都要将上一状态 的 后继集合 重置
        memset(S, 0, sizeof(S));
        for(j = 0; f[j] <= i && j <= N; j++)
            S[SG[i-f[j]]] = 1; //将后继状态的SG函数值进行标记
        for(j = 0;; j++) if(!S[j]) { //查询当前后继状态SG值中最小的非零
            SG[i] = j;
            break;
        }
    }
}

```

现在我们来一个实战演练 ([HDU1848](#)) :

只要按照上面的思路, 解决这个就是分分钟的问题。

代码如下:

```
#include <stdio.h>
#include <string.h>
#define MAXN 1000 + 10
#define N 20
int f[N], SG[MAXN], S[MAXN];
void getSG(int n){
    int i, j;
    memset(SG, 0, sizeof(SG));
    for(i = 1; i <= n; i++){
        memset(S, 0, sizeof(S));
        for(j = 0; f[j] <= i && j <= N; j++)
            S[SG[i-f[j]]] = 1;
        for(j = 0;;j++) if(!S[j]){
            SG[i] = j;
            break;
        }
    }
}
int main(){
    int m, n, k;
    f[0] = f[1] = 1;
    for(int i = 2; i <= 16; i++)
        f[i] = f[i-1] + f[i-2];
    getSG(1000);
    while(scanf("%d%d%d", &m, &n, &k), m||n||k){
        if(SG[n]^SG[m]^SG[k]) printf("Fibo\n");
        else printf("Nacci\n");
    }
    return 0;
}
```

KMP

说明

时间复杂度: $O(n+m)$

- `char T[]` : 被匹配串
- `char P[]` : 子串
- `char f[]` : 存储失败指针, 大小需要大于等于与 `P[]`
- 返回值: 所有匹配上的点的下标

`f[i]` 表示: 如果子串 `P[i]` 匹配原串 `T[j]` 失败, 则使用 `P[f[i]]` 匹配 `T[j]`

使用方法

1. `scanf("%s %s", T, P);` 输入被匹配串T, 匹配子串P
2. `vector<int>ans = KMP(T, P, f);` 需要自行开辟失败指针数组 `f[]`, 返回每一个匹配位置下标

Tips

- `getFail()` 会在 KMP 中自动调用
- 字符串均从 `s[0]` 开始
- KMP 函数中的 `for` 的 `while` 利用了字符串最后一个字符是 `\0` 的特性, 当匹配的是int数组时需要特别注意 (坑题示例[HDU 5918 Sequence I](#))

模版

```

void getFail(char *P, int *f) {
    int lenP = (int)strlen(P);
    f[0] = 0; f[1] = 0; //递推边界初值
    for (int i=1; i<lenP; i++) {
        int j = f[i];
        while (j && P[i] != P[j]) {
            j = f[j];
        }
        f[i+1] = (P[i]==P[j]?j+1:0);
    }
}

vector<int> KMP(char *T, char *P, int *f) {
    int lenT = strlen(T);
    int lenP = strlen(P);
    getFail(P, f);
    int j = 0; //当前结点的编号，初始为0号始点
    vector<int> ans;
    for (int i = 0; i<lenT; i++) { //文本串当前指针
        while (j && P[j] != T[i]) //顺着失败边走，直到可以匹配
            j = f[j];
        if (P[j] == T[i])
            j++;
        if (j==lenP)
            ans.push_back(i-lenP+1);
    }
    return ans;
}

```

Trie前缀树

说明

此模版用于查找对于若干个字符串，查找某字符串作为前缀出现的次数

- `maxNode` : 最大节点的数量
- `sigma_size` : 每个节点的子节点个数
- `sz` : 表示字典树中下一个节点的下标，根节点编号为0，不指向任何字符
- `val[]` : 到当前位置的前缀串出现的次数

使用方法

1. `init()`; 初始化
2. `insert()` 插入所有字符串
3. `query()` 查询

模版

```

const int maxNode=1005*25;
const int sigma_size=26;
int c[maxNode][sigma_size];
//    int val[maxNode]; //val[i]用于记录以下标以i结尾的辅助信息，如权值
int cnt[maxNode];
int sz;
void init()
{
    sz=1;
    memset(c[0], 0, sizeof(c[0]));
    memset(cnt, 0, sizeof(cnt));
}
int idx(char ch) {
    return ch - 'a';
}
void insert(char s[], int v=0)      //v表示该字符串的辅助信息，如权值等，使用
时需要取消对应注释
{
    int u=0;
    for (int i=0; s[i]; i++)
    {

```

```
char ch = idx(s[i]);
if (!c[u][ch])
{
    memset(c[sz], 0, sizeof(c[sz]));
    // val[sz]=0;
    c[u][ch]=sz++;
}
u=c[u][ch];
cnt[u]++;
}
// val[u]=v;
}

int query(char s[])
{
    int u=0,n=strlen(s);
    for (int i=0; i<n; i++)
    {
        char ch = idx(s[i]);
        if (!c[u][ch] || u!=0&&cnt[u]<=1)
            return i;      //查询最大匹配长度
        u=c[u][ch];
    }
    return n;      //查询最大匹配长度
// return cnt[u]; /*查询该字符串出现的次数*/
}
```

Aho-Corasick自动机

使用方法

- 见主函数

模版

```

const int maxn=500010+5;
const int SIGMA_SIZE=26;
struct ACAutomata
{
    int nxt[maxn][SIGMA_SIZE];      //节点
    int fail[maxn]; //失配指针
    int end[maxn]; //end[i]记录以i结尾的字符串个数

    int root,L;
    int newnode()
    {
        for(int i = 0; i < SIGMA_SIZE; i++)
            nxt[L][i] = -1;
        end[L++] = 0;
        return L-1;
    }
    int idx(char ch) {
        return ch - 'a';
    }
    void init()
    {
        L = 0;
        root = newnode();
    }
    void insert(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        for(int i = 0; i < len; i++)
        {
            char ch = idx(buf[i]);
            if(nxt[now][ch] == -1)

```

```

        nxt[now][ch] = newnode();
        now = nxt[now][ch];
    }
    end[now]++;
}
void build()
{
    queue<int> q;
    fail[root] = root;
    for(int i = 0; i < SIGMA_SIZE; i++)
        if(nxt[root][i] == -1)
            nxt[root][i] = root;
        else
    {
        fail[nxt[root][i]] = root;
        q.push(nxt[root][i]);
    }
    while(!q.empty())
    {
        int now = q.front();
        q.pop();
        for(int i = 0; i < SIGMA_SIZE; i++)
            if(nxt[now][i] == -1) //若该点不存在，直接将该位置指向失配

```

指针的下一位

```

                nxt[now][i] = nxt[fail[now]][i];
            else
    {
        fail[nxt[now][i]]=nxt[fail[now]][i];
        q.push(nxt[now][i]);
    }
}
int query(char buf[])
{
    int len = strlen(buf);
    int now = root;
    int res = 0;
    for(int i = 0; i < len; i++)
    {
        now = nxt[now][idx(buf[i])];
        int tmp = now;
        while(tmp != root)
        {

```

```

        res += end[tmp];
        end[tmp] = 0;    //防止重复，如考虑重复情况请注释掉本行，如
Hdu5384
        tmp = fail[tmp];
    }
}
return res;
}
void Debug()
{
    for(int i = 0; i < L; i++)
    {
        printf("id = %3d,fail = %3d,end = %3d,chi =
[",i,fail[i],end[i]);
        for(int j = 0; j < SIGMA_SIZE; j++)
            printf("%2d",nxt[i][j]);
        printf("]\n");
    }
}
char buf[1000010];
ACAutomata ac;
int main()
{
    int t;
    int n;
    scanf("%d",&t);
    while(t--)
    {
        scanf("%d",&n);
        ac.init();
        for(int i = 0; i < n; i++)
        {
            scanf("%s",buf);
            ac.insert(buf);
        }
        ac.build();
        scanf("%s",buf);
        printf("%d\n",ac.query(buf));
    }
    return 0;
}

```


后缀数组

说明

本算法的实现使用：DA算法

将字符串 s 分割为 len 个后缀串，然后将这 len 个字符串按照字典序排放，每一个后缀串可以用第一个字符在原串中的位置表示，用 $sa[]$ 数组存储字典序从小到大的后缀串的首字符下标。 $rk[]$ 数组表示对于当前串，在 sa 数组中的下标位置。 $ht[i]$ 数组表示 $sa[i]$ 和 $sa[i-1]$ 表示的两个后缀串的公共前缀长度。

用法

1. `getSa(s, len+1, 'z'+1);` 对于从0开始的字符串 s ，长度为 len ，最大值为 ' z '
2. `getHet(s, len);` 当获取过 sa 数组后 对于从0开始的字符串，长度为 len

Tips

- 分清下标，见主函数

模版

```
/*
2016.7.31
rk的有效信息从0到n-1
sa,ht的有效信息从1到n
在传参的时候要多传一位*/
const int maxn=100000+5;
int rk[maxn], sa[maxn], ht[maxn], wa[maxn], wb[maxn], wx[maxn],
wv[maxn];

bool isq(int *r, int a, int b, int len) {
    return r[a] == r[b] && r[a + len] == r[b + len];
}

bool isEqual(int *r, int a, int b, int len) {
    return r[a] == r[b] && r[a + len] == r[b + len];
}
// r数组有效信息为0~n-1, m为 (数组中最大值的上界+1)
```

```

void getSa(char r[], int n, int m) {
    int i, j, p, *t, *x = wa, *y = wb;
    for (i = 0; i < m; ++i)
        wx[i] = 0;
    for (i = 0; i < n; ++i)
        ++wx[x[i]] = r[i];
    for (i = 1; i < m; ++i)
        wx[i] += wx[i - 1];
    for (i = n - 1; i >= 0; --i)
        sa[--wx[x[i]]] = i;
    for (j = 1, p = 0; p < n; j <= 1, m = p) {
        for (p = 0, i = n - j; i < n; ++i)
            y[p++] = i;
        for (i = 0; i < n; ++i)
            sa[i] >= j ? y[p++] = sa[i] - j : 0;
        for (i = 0; i < m; ++i)
            wx[i] = 0;
        for (i = 0; i < n; ++i)
            ++wx[wv[i]] = x[y[i]];
        for (i = 1; i < m; ++i)
            wx[i] += wx[i - 1];
        for (i = n - 1; i >= 0; --i)
            sa[--wx[wv[i]]] = y[i];
        p = 1, t = x, x = y, y = t;
        x[sa[0]] = 0;
        for (i = 1; i < n; ++i)
            x[sa[i]] = isEqual(y, sa[i], sa[i - 1], j) ? p - 1 :
    p++;
    }
}

void getHet(char r[], int n) {
    int i, j, k = 0;
    for (i = 1; i <= n; ++i)
        rk[sa[i]] = i;
    for (i = 0; i < n; ht[rk[i++]] = k) {
        k = k > 0 ? k - 1 : 0;
        j = sa[rk[i] - 1];
        while (r[i + k] == r[j + k])
            ++k;
    }
}

int main() {

```

```
char s[maxn] = "banana";
int len = strlen(s);

//用法
getSa(s, len+1, 'z'+1); //对于从0开始的字符串s, 长度为len, 最大值
为'z'
getHet(s, len); //当获取过sa数组后 对于从0开始的字符串, 长度为len

//备注: sa[]和rk符合逆运算 即同时满足sa[i]=j rk[j]=i
for (int i=1; i<=len; i++) {
    printf("%d%c", sa[i], "\n"[i==len]);
} // 输出sa数组 下标从 1~len 值为: 5 3 1 0 4 2

for (int i=0; i<len; i++) {
    printf("%d%c", rk[i], "\n"[i==len-1]);
} // 输出rk数组 下标从 0~len-1 值为: 4 3 6 2 5 1

for (int i=1; i<=len; i++) {
    printf("%d%c", ht[i], "\n"[i==len]);
} // 输出ht数组 下标从 1~len 值为: 0 1 3 0 0 2
}
```

Manacher

说明

manacher算法能在接近线性的时间找到最长的回文串。

推荐一篇博客，看这篇博客看了三四次才把这个算法给搞明白->[Manacher's Algorithm 马拉车算法](#)

使用方法

- `string subStr = Manacher(string s);` 返回值为最长回文串

模版

```

#include <vector>
#include <iostream>
#include <string>
using namespace std;
string Manacher(string s) {
    // Insert '#'
    string t = "$#";
    for (int i = 0; i < s.size(); ++i) {
        t += s[i];
        t += "#";
    }
    // Process t
    vector<int> p(t.size(), 0);
    int mx = 0, id = 0, resLen = 0, resCenter = 0;
    for (int i = 1; i < t.size(); ++i) {
        p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
        while (t[i + p[i]] == t[i - p[i]]) ++p[i];
        if (mx < i + p[i]) {
            mx = i + p[i];
            id = i;
        }
        if (resLen < p[i]) {
            resLen = p[i];
            resCenter = i;
        }
    }
    return s.substr((resCenter - resLen) / 2, resLen - 1);
}

int main() {
    string s1 = "12212";
    cout << Manacher(s1) << endl;
    string s2 = "122122";
    cout << Manacher(s2) << endl;
    string s = "waabwsfd";
    cout << Manacher(s) << endl;
}

```

回文树

说明

原文链接：[Palindromic Tree——回文树【处理一类回文串问题的强力工具】](#)

1. `len[i]` 表示编号为`i`的节点表示的回文串的长度（一个节点表示一个回文串）
2. `next[i][c]` 表示编号为`i`的节点表示的回文串在两边添加字符`c`以后变成的回文串的编号（和字典树类似）。
3. `fail[i]` 表示节点`i`失配以后跳转不等于自身的节点`i`表示的回文串的最长后缀回文串（和AC自动机类似）。
4. `cnt[i]`
~~表示节点*i*表示的本质不同的串的个数（建树时求出的不是完全的，最后count()函数跑一遍以后才是正确的）~~

看了半天没看懂有什么卵用，等我用到了再回来补

5. `num[i]` 表示以节点`i`表示的最长回文串的最右端点为回文串结尾的回文串个数。
6. `last` 指向新添加一个字母后所形成的最长回文串表示的节点。
7. `s[i]` 表示第`i`次添加的字符（一开始设`s[0] = -1`（可以是任意一个在串`S`中不会出现的字符））。
8. `p` 下一个被插入的位置下标
9. `n` 已插入的字符数量

使用方法

- `Palindromic_Tree p;`
- `p.init();` 初始化
- `int tmp = p.add(str[i]);` 依次加入字符串中的每个字符，返回的`tmp`为以当前字符结尾的回文串数量
- 全部插入之后，以插入的第`i`个字符结尾的回文串的个数为 `num[i+1]`，最大长度为 `len[i+1]`，其中`i`的下标在 `[1, Len]` 区间

Tips

- 注意 `maxLen` 和 `SIGMA_SIZE`
- `scanf("%s", str+1);` 下标从 `1` 开始

模版

```

const int maxLen = 100005 ;
const int SIGMA_SIZE = 26 ;
struct Palindromic_Tree {
    int next[maxLen][SIGMA_SIZE] ;//next指针， next指针和字典树类似， 指向的串为当前串两端加上同一个字符构成
    int fail[maxLen] ;//fail指针， 失配后跳转到fail指针指向的节点
    int cnt[maxLen] ;
    int num[maxLen] ;//以该节点结束的回文串的数量
    int len[maxLen] ;//len[i]表示 以i结尾的最大回文串的长度
    int S[maxLen] ;//存放添加的字符
    int last ;//指向一个字符所在的节点， 方便下一次add
    int n ;//已插入的字符数量
    int p ;//下一个被插入的位置下标

    int newnode ( int l ) { //新建节点
        for ( int i = 0 ; i < SIGMA_SIZE ; ++ i ) next[p][i] = 0 ;
        cnt[p] = 0 ;
        num[p] = 0 ;
        len[p] = 1 ;
        return p ++ ;
    }

    void init () { //初始化
        p = 0 ;
        newnode ( 0 ) ;
        newnode ( -1 ) ;
        last = 0 ;
        n = 0 ;
        S[n] = -1 ;//开头放一个字符集中没有的字符， 减少特判
        fail[0] = 1 ;
    }

    int get_fail ( int x ) { //和KMP一样， 失配后找一个尽量最长的
        while ( S[n - len[x] - 1] != S[n] ) x = fail[x] ;
        return x ;
    }

    int add ( int c ) { //返回以当前字符结尾的回文串个数
        c -= 'a' ;
        S[++n] = c ;
        int cur = get_fail ( last ) ;//通过上一个回文串找这个回文串的匹配
       位置
        if ( !next[cur][c] ) { //如果这个回文串没有出现过， 说明出现了一个新
    }

```

的本质不同的回文串

```
int now = newnode ( len[cur] + 2 ) ;//新建节点
fail[now] = next[get_fail ( fail[cur] )][c] ;//和AC自动机
一样建立fail指针，以便失配后跳转
next[cur][c] = now ;
num[now] = num[fail[now]] + 1 ;
}
last = next[cur][c] ;
cnt[last] ++ ;
return num[last];
}

void count () {
for ( int i = p - 1 ; i >= 0 ; -- i ) cnt[fail[i]] += cnt[i]
;
//父亲累加儿子的cnt，因为如果fail[v]=u，则u一定是v的子回文串！
}
};

}
```

并查集

说明

并查集

使用方法

- `init(n);` 初始化 n 为并查集的元素数量
- `findX(x);` 找到 x 的根节点
- `merge(a, b);` 合并 a, b 所在集合
- `zip1(x);` 和 `zip2(x);` 压缩 x 路径上的所有节点

Tips

- 该 `merge` 为按秩合并，不需要时可以直接使用简化版本
- 压缩路径使用 `zip2()` 较好，递归调用较消耗时间

模版

```

const int maxn=2005+5;
int bin[maxn];//bin表示集合树
int sz[maxn];//sz表示每个节点所形成的分支的大小

//初始化bin数组
void init(int size){
    for (int i=0; i<=size; i++) {
        bin[i]=i;
        sz[i]=1;
    }
}

//找到x的根节点
int findX(int x) {
    int r=x;
    while (bin[r]!=r) {
        r=bin[r];
    }
    return r;
}

```

```

}

//路径压缩—递归形式 (效率低)
int zip1(int x) {
    if (x!=bin[x]) {
        bin[x]=zip1(bin[x]);
    }
    return bin[x];
}

//路径压缩—迭代形式 (效率高)
int zip2(int x) {
    int tmp, _x = x;
    while (bin[_x]!=_x)
        _x = bin[_x];
    while (bin[x]!=x) {
        tmp = bin[x];
        bin[x] = _x;
        x = tmp;
    }
    return _x;
}

//合并两个集合
void merge(int a,int b) {
    int aRoot=findX(a);
    int bRoot=findX(b);
    if (aRoot==bRoot) return;

    //这个选择结构只是为了让树更加平衡, 即把小树作为大树的子树
    //如果不考虑树的平衡问题, 只需使bin[aRoot]=bRoot即可
    if (sz[aRoot] < sz[bRoot]) {
        bin[aRoot] = bRoot;
        sz[bRoot] += sz[aRoot];
    }
    else {
        bin[bRoot] = aRoot;
        bin[aRoot] += bin[bRoot];
    }
}

```


树状数组

树状数组

说明

树状数组(Binary Indexed Tree) 总结

`bin[]` 为树状数组 `n` 为数组的下标最大值

时间复杂度分析:

- `update` $O(\log n)$
- `sum` $O(\log n)$

使用方法

1. `memset(bin, 0, sizeof(bin));`
2. 给 `n` 赋值
3. `update(3, 1);` 给数组中下标为3的位置更新 1
4. `sum(i);` 求数组中下标为1~i的值的和

Tips

- 树状数组无法更新下标为 0 的数
- 分清树状数组的操作是更新、修改、取最大/最小值等
- 用树状数组时经常需要离散化

模版

```

int bin[maxn], n;
int lowbit(int x) {
    return x & (-x);
}
void update(int pos, int val) {
    while (pos < maxn) {
        bin[pos] += val;
        pos += lowbit(pos);
    }
}
int sum(int pos) {
    int ans = 0;
    while (pos > 0) {
        ans += bin[pos];
        pos -= lowbit(pos);
    }
    return ans;
}

```

应用

单点更新区间查和

最一般的应用，每次将树状数组对应位置的值更新。

```

for (int i=1; i<=n; i++) {
    update(pos[i], val[i]);
}
int ans = sum(r) - sum(l-1);

```

区间更新单点查询

转化为更新两个点的值来更新区间

```

update(l, 1);
update(r+1, -1);
int val = sum(pos);

```

求解逆序数

一般需要对序列离散化，可以降低更新和查询的复杂度以及数组大小

```
for (int i=1; i<=n; i++) {  
    update(a[i], 1);  
    ans+=i-sum(a[i]);    //插入的数字个数 - 已插入的小于等于a[i]的元素个数  
}  
printf("逆序数个数为: %d", ans);
```

区间更新区间查询

使用方法

1. 给n赋值
2. memset(c, 0, sizeof(c));
3. memset(b, 0, sizeof(b));
4. add(l,r,x); 给 [l,r] 区间增加 x
5. sum(l, r); 查询区间 [l, r] 的和

模版

```

int n;
const int maxn=500005;
long long c[maxn],b[maxn];
inline int lowbit(int t)
{
    return t&(-t);
}
void update(long long c[],int flag,int x,long long v)
{
    if (flag) for (int i=x; i<=n; i+=lowbit(i)) c[i]+=x*v;
    else for (int i=x; i>0; i-=lowbit(i)) c[i]+=v;
}
long long query(long long c[],int flag,int x)
{
    long long ans=0;
    if (flag) for (int i=x; i>0; i-=lowbit(i)) ans+=c[i];
    else for (int i=x; i<=n; i+=lowbit(i)) ans+=c[i];
    return ans;
}
void add(int l,int r,long long v)
{
    update(b,0,r,v);
    update(c,1,r,v);
    if (l>1)
    {
        update(b,0,l-1,-v);
        update(c,1,l-1,-v);
    }
}
long long sum(int x)
{
    if (x) return query(b,0,x)*x+query(c,1,x-1);
    else return 0;
}
long long sum(int l,int r)
{
    return sum(r)-sum(l-1);
}

```

二维树状数组

说明

c[] 为二维树状数组

n 为数组的下标最大值

使用方法

1. `memset(c, 0, sizeof(c));`
2. 给 n 赋值
3. `update(posx, posy, val);` 给数组中下标为[posx][posy]的位置更新 val
4. `sum(x, y, xx, yy);` 求数组中下标为 (x, y) 与 (xx, yy) 为对角线顶点的区域元素之和

Tips

- 树状数组无法更新下标为 0 的数
- 分清树状数组的操作是更新、修改、取最大/最小值等
- 用树状数组时经常需要离散化

模版

二维树状数组

```
int N;
int c[maxn][maxn];
inline int lowbit(int t)
{
    return t&(-t);
}
void update(int x, int y, int v)
{
    for (int i=x; i<=N; i+=lowbit(i))
        for (int j=y; j<=N; j+=lowbit(j))
            c[i][j] += v;
}
int query(int x, int y)
{
    int s=0;
    for (int i=x; i>0; i-=lowbit(i))
        for (int j=y; j>0; j-=lowbit(j))
            s += c[i][j];
    return s;
}
int sum(int x, int y, int xx, int yy)
{
    x--, y--;
    return query(xx, yy) - query(xx, y) - query(x, yy) + query(x, y);
}
```

三维树状数组

说明

c[] 为三维树状数组

n 为数组的下标最大值

使用方法

1. memset(c, 0, sizeof(c));
2. 给 N 赋值
3. update(posx, posy, posz, val); 给数组中下标为[posx][posy][posz]的位置更新 val
4. sum(x, y, z, xx, yy, zz); 求数组中下标为 (x, y, z) 与 (xx, yy, zz) 为对角线顶点的区域元素之和

Tips

- 树状数组无法更新下标为 0 的数
- 分清树状数组的操作是更新、修改、取最大/最小值等
- 用树状数组时经常需要离散化

模版

三维树状数组

```
int N;
long long c[130][130][130] = {};
inline int lowbit(int t)
{
    return t & (-t);
}
void update(int x, int y, int z, long long v)
{
    for (int i=x; i<=N; i+=lowbit(i))
        for (int j=y; j<=N; j+=lowbit(j))
            for (int k=z; k<=N; k+=lowbit(k))
                c[i][j][k] += v;
}
long long query(int x, int y, int z)
{
    long long s=0;
    for (int i=x; i>0; i-=lowbit(i))
        for (int j=y; j>0; j-=lowbit(j))
            for (int k=z; k>0; k-=lowbit(k))
                s += c[i][j][k];
    return s;
}
long long sum(int x, int y, int z, int xx, int yy, int zz)
{
    x--, y--, z--;
    return query(xx, yy, zz)
        -query(x, yy, zz) -query(xx, y, zz) -query(xx, yy, z)
        +query(x, y, zz) +query(xx, y, z) +query(x, yy, z)
        -query(x, y, z);
}
```

线段树

单点更新

说明

单点更新，区间求和（你问我单点求和？？你就不会把区间长度设为0啊？）

- `sum[]` 为线段树，需要开辟四倍的元素数量的空间。
- `build()` 为建树操作
- `update()` 为更新操作
- `query()` 为查询操作

时间复杂度： $O(n\log n)$

使用方法

1. `build(1, n);` 建立一个叶子节点为 n 个的线段树
2. `update(pos, val, 1, n);` 更新树中下标为 pos 的叶子节点值增加 val
3. `query(l, r, 1, n);` 查询 $[l, r]$ 区间值之和

Tips

- 请注意`update`的目的是增减还是替换，根据情况修改`update`函数和`pushup`函数
- 建出来的树为空树，默认每个点值都为0，需要自行将值`update`上去，或者修改`build`中 `sum[rt]=0;` 为输入操作 `scanf("%d", sum+rt);`

模版

```
//无注释版本
const int maxn=2005+5;
#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1
int sum[maxn<<2];
void pushup(int rt) {
    sum[rt]=sum[rt<<1]+sum[rt<<1|1];
}
void build(int l,int r,int rt=1) {
    if (l==r) {
        sum[rt]=0;
        return;
    }
    int m=(l+r)>>1;
    build(lson);
    build(rson);
    pushup(rt);
}
void update(int pos,int val,int l,int r,int rt=1) {
    if (l==r) {
        sum[rt]+=val;
        return;
    }
    int m = ( l + r ) >> 1;
    if (pos<=m)
        update(pos, val, lson);
    else
        update(pos, val, rson);
    pushup(rt);
}
int query(int L,int R,int l,int r,int rt=1) {
    if (L<=l&&r<=R)
        return sum[rt];
    int m=(l+r)>>1;
    int res=0;
    if (L<=m)
        res+=query(L, R, lson);
    if (R>m)
        res+=query(L, R, rson);
    return res;
}
```

```

// 有注释版

const int maxn=2005+5;
#define lson l,m,rt<<1           //预定子左树
#define rson m+1,r,rt<<1|1        //预定右子树
int sum[maxn<<2]; //表示节点, 需要开到最大区间的四倍

void pushup(int rt) {
    //对于编号为rt的节点, 他的左右节点分别为rt<<1和rt<<1|1
    sum[rt]=sum[rt<<1]+sum[rt<<1|1];
}

//造树
void build(int l,int r,int rt=1) {
    //建树操作, 生成一个区间为l~r的完全二叉树

    //如果到底, 则线段长度为0, 表示一个点, 输入该点的值
    if (l==r) {
        sum[rt]=0;
        return;
    }

    //准备子树
    int m=(l+r)>>1;

    //对当前节点建立子树
    build(lson);
    build(rson);

    //由底向上求和
    pushup(rt);
}

//更新点和包含点的枝
void update(int pos,int val,int l,int r,int rt=1) {
    //pos为更新的位置 val为增加的值, 正则加, 负则减
    //l r为区间的两个端点值

    //触底, 为一个点的时候, 该节点值更新
    if (l==r) {
        sum[rt]+=val;
        return;
    }

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

```

```
if (pos<=m)      //pos在左子树的情况下，对左子树进行递归
    update(pos, val, lson);
else            //pos在右子树的情况下，对右子树进行递归
    update(pos, val, rson);

//更新包含该点的一系列区间的值
pushup(rt);
}

//查询点或区间
int query(int L,int R,int l,int r,int rt=1) {
    // L~R为被查询子区间 l~r为“当前”树的全区间
    if (L<=l&&r<=R)      //子区间包含“当前”树全区间
        return sum[rt]; //返回该节点包含的值
    int m=(l+r)>>1;
    int res=0;
    if (L<=m)          //左端点在左子树内
        res+=query(L, R, lson);
    if (R>m)          //右端点在右子树内
        res+=query(L, R, rson);
    return res;
}
```

区间更新

说明

区间更新，区间求和（你问我单点求和？？你就不会把区间长度设为0啊？）

- `sum[]` 为线段树，需要开辟四倍的元素数量的空间。
- `build()` 为建树操作
- `update()` 为更新操作
- `query()` 为查询操作

时间复杂度：O(nlogn)

使用方法

1. `build(1, n);` 建立一个叶子节点为 n 个的线段树
2. `update(l, r, val, 1, n);` 更新线段树中 $[l, r]$ 区间每个值都增加 val
3. `query(l, r, 1, n);` 查询 $[l, r]$ 区间值之和

Tips

- 请注意`update`的目的是增减还是替换，根据情况修改`update`函数和`pushup`函数
- 建出来的树为空树，默认每个点值都为0，需要自行将值`update`上去，或者修改`build`中 `sum[rt]=0;` 为输入操作 `scanf("%d", sum+rt);`

模版

```
#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1
const int maxn = 100005;
int add[maxn<<2],sum[maxn<<2];
void PushUp(int rt)
{
    sum[rt]=sum[rt<<1]+sum[rt<<1|1];
}
void PushDown(int rt,int m)
{
    if (add[rt])
    {
        add[rt<<1] += add[rt];
    }
}
```

区间更新

```
    add[rt<<1|1] += add[rt];
    sum[rt<<1] += add[rt] * (m - (m >> 1));
    sum[rt<<1|1] += add[rt] * (m >> 1);
    add[rt] = 0;
}
}

void build(int l,int r,int rt=1)
{
    add[rt] = 0;
    if (l == r)
    {
        sum[rt]=0;
        return ;
    }
    int m = (l + r) >> 1;
    build(lson);
    build(rson);
    PushUp(rt);
}

void update(int L,int R,int c,int l,int r,int rt=1)
{
    if (L <= l && r <= R)
    {
        add[rt] += c;
        sum[rt] += c * (r - l + 1);
        return ;
    }
    PushDown(rt , r - l + 1);
    int m = (l + r) >> 1;
    if (L <= m) update(L , R , c , lson);
    if (m < R) update(L , R , c , rson);
    PushUp(rt);
}

int query(int L,int R,int l,int r,int rt=1)
{
    if (L <= l && r <= R)
    {
        return sum[rt];
    }
    PushDown(rt , r - l + 1);
    int m = (l + r) >> 1;
    int ret = 0;
    if (L <= m) ret += query(L , R , lson);
```

区间更新

```
if (m < R) ret += query(L, R, rson);
return ret;
}
```

RMQ

说明

RMQ: Range Minimum(Maximum) Query

- `sum[]` 为线段树，需要开辟四倍的元素数量的空间。
- `build()` 为建树操作
- `update()` 为更新操作
- `query()` 为查询操作

使用方法

1. 根据情况修改RMQ的宏定义
2. `build(1, n);` 建立一个叶子节点为 `n` 个的线段树
3. `update(pos, val, 1, n);` 修改树中下标为 `pos` 的叶子节点值为 `val`
4. `query(l, r, 1, n);` 查询 `[l, r]` 区间中的RMQ

Tips

- 建出来的树为空树，默认每个点值都为0，需要自行将值update上去，或者修改build 中 `sum[rt]=0;` 为输入操作 `scanf("%d", sum+rt);`
- RMQ为宏定义，请根据情况自行修改为 `max` 或者 `min`，对应修改 query 中的 `res` 为 `-INF` 或者 `INF`

模版

```

const int maxn=2005+5;
#define RMQ max
#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1
int sum[maxn<<2]={ };
void pushup(int rt) {
    sum[rt]=RMQ(sum[rt<<1],sum[rt<<1|1]);
}
void build(int l,int r,int rt=1) {
    if (l==r) {
        sum[rt]=0;
        return;
    }
    int m=(l+r)>>1;
    build(lson);
    build(rson);
    pushup(rt);
}
void update(int pos,int val,int l,int r,int rt=1) {
    if (l==r) {
        sum[rt]=val;
        return;
    }
    int m=(l+r)>>1;
    if (pos<=m) update(pos, val, lson);
    else update(pos, val, rson);
    pushup(rt);
}
int query(int L,int R,int l,int r,int rt=1) {
    if (L<=l&&r<=R) return sum[rt];
    int m=(l+r)>>1;
    int res=-INF;      //防负数的坑
    if (L<=m) res=RMQ(res,query(L, R, lson));
    if (R>m) res=RMQ(res,query(L, R, rson));
    return res;
}

```

树链剖分

By Tak3n

说明

Tak3n

树链剖分解决在树上进行任意两点间的一类路径更新和路径查询的问题。大体分为点权型和边权型两种。该模板以更新点，单点更新，区间查询，求和为基础，其他种类在后面补充。

Manyfun

示例题目都是使用Tak3n的模版做的，模版很完善，但是细节要注意。分清楚边权型题目和点权型题目的建图区别，线段树方面要注意是使用RMQ/区间更新/单点更新的板子。当边权型题目最后是找 `son[u]` 和 `v`，详解请见本人[SPOJ-QTREE Query on a tree题解](#)

推荐博客

[树链剖分——starszys的博客](#)

使用方法

1. `work(n)` 初始化,注意输入点权和输入树的先后关系, 默认点下标从1开始, 树根为1
2. `find(x, y)` 求x,y路径上的点权和 (或最值, 可通过修改线段树得到)
3. `update(w[x], y, 1, nodenum)` 将x节点的权值更新为y

示例

1. 点权, 区间更新, 单点/区间求和[HDU 3966 Aragorn's Story](#)
2. 边权, 单点更新, 单点/区间求和[POJ 2763 Housewife Wind](#)
3. 点权, 单点更新, 单点/区间求和[LightOJ 1348 Aladdin and the Return Journey](#)
4. 边权, 单点更新, 区间求和[FZU 2082 过路费](#)
5. 点权, 单点更新, 区间求和 + RMQ[HYSBZ 1036 树的统计Count](#)
6. 边权, 单点更新, 区间RMQ[SPOJ-QTREE Query on a tree](#)
7. 点权+边权, 区间更新, 单点查询, 数组维护[HDU 5044 Tree](#)
8. 点权, 单点更新, 线段树变型[HDU 5274 Dylans loves tree](#)

模版

```
// 点权型 单点更新 区间查询
#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1
const int Vmax = 5*1e4 + 5;//点的数量
const int Emax = 2*1e5+5;//边的数量 小于Vmax的两倍
namespace segment_tree{
    int sum[Vmax<<2], add[Vmax<<2];
    inline void pushup(int rt) {
        sum[rt]=sum[rt<<1]+sum[rt<<1|1];
    }
    void update(int L,int c,int l,int r,int rt=1) {
        if (L == l && l == r)
        {
            sum[rt] = c;
            return ;
        }
        int m = (l + r) >> 1;
        if (L <= m) update(L , c , lson);
        else update(L , c , rson);
        pushup(rt);
    }
    int query(int L,int R,int l,int r,int rt=1){
        if (L <= l && r <= R)
            return sum[rt];
        int m = (l + r) >> 1;
        int ret = 0;
        if (L <= m) ret+=query(L , R , lson);
        if (m < R) ret+=query(L , R , rson);
        return ret;
    }
}
namespace poufen{
    using namespace segment_tree;
    int siz[Vmax], son[Vmax], fa[Vmax], dep[Vmax], top[Vmax], w[Vmax];
    int nodenum;

    struct edge{
        int v,next;
    }e[Emax];
    int pre[Vmax],ecnt;
    inline void init() {

```

```

        memset(pre, -1, sizeof(pre));
        ecnt=0;
    }

    inline void add_(int u,int v) {
        e[ecnt].v=v;
        e[ecnt].next=pre[u];
        pre[u]=ecnt++;
    }

    void dfs(int u) {
        siz[u]=1;son[u]=0;//下标从1开始, son[0]初始为0
        for(int i=pre[u];~i;i=e[i].next)
        {
            int v=e[i].v;
            if(fa[u]!=v)
            {
                fa[v]=u;
                dep[v]=dep[u]+1;//初始根节点dep!=0
                dfs(v);
                siz[u]+=siz[v];
                if(siz[v]>siz[son[u]])son[u]=v;
            }
        }
    }

    void build_tree(int u,int tp) {
        top[u]=tp,w[u]=++nodenum;
        if(son[u])build_tree(son[u],tp);
        for(int i=pre[u];~i;i=e[i].next)
            if(e[i].v!=fa[u]&&e[i].v!=son[u])
                build_tree(e[i].v,e[i].v);
    }

    inline int find1(int u,int v) {
        int ret=0;
        int f1=top[u],f2=top[v];
        while(f1!=f2)
        {
            if(dep[f1]<dep[f2])
                swap(f1,f2),swap(u,v);
            ret+=query(w[f1],w[u],1,nodenum);
            u=fa[f1];
            f1=top[u];
        }
        if(dep[u]>dep[v])swap(u,v);
    }
}

```

```

    ret+=query(w[u],w[v],1,nodenum);
    return ret;
}

int a[Vmax],b[Vmax];
int val[Vmax];//  

void work1(int n)
{
    memset(siz, 0, sizeof(siz));
    memset(sum, 0, sizeof(sum));

    init();
    int root=1;
    fa[root]=nodenum=dep[root]=0;
    for(int i=1;i<=n;i++)
        scanf("%d",&val[i]);
    for(int i=1;i<n;i++)
    {
        scanf("%d%d",&a[i],&b[i]);
        add_(a[i],b[i]);
        add_(b[i],a[i]);
    }
    dfs(root);
    build_tree(root,root);
    for(int i=1;i<=n;i++)
        update(w[i],val[i],1,nodenum);
}

using namespace poufen;

```

如果是更新边，`find`函数替换为以下函数。

边权型

```
inline int find2(int u,int v){  
    int f1=top[u],f2=top[v],ret=0;  
    while(f1!=f2)  
    {  
        if(dep[f1]<dep[f2])  
            swap(f1,f2),swap(u,v);  
        ret+=query(w[f1],w[u],1,nodenum);  
        u=fa[f1];  
        f1=top[u];  
    }  
    if(u==v) return ret;  
    if(dep[u]>dep[v]) swap(u,v);  
    return ret+=query(w[son[u]],w[v],1,nodenum);  
}
```

如果是边权型， work函数替换为以下函数。

```

void work2 (int n) {
    memset(siz, 0, sizeof(siz));
    memset(sum, 0, sizeof(sum));

    init();
    int root=1;
    fa[root]=nodenumber=dep[root]=0;
    for(int i=1;i<n;i++)
    {
        scanf("%d%d%d", &a[i], &b[i], &c[i]);
        add_(a[i], b[i]);
        add_(b[i], a[i]);
    }
    dfs(root);

    build_tree(root, root);

    for(int i=1;i<n;i++) {
        int u = a[i];
        int v = b[i];
        if (dep[u] > dep[v]) { //保证v是被指向的点，也就是深度较大的点
            swap(u, v);
            swap(a[i], b[i]);
        }
        update(w[v], c[i], 1, nodenum);
    }
}

```

区间更新

点更新

如果是区间更新，update为 `upd(u, v, c)` 意为将u,v路径上的点权都 $+=c$,注意将线段树模板修改成区间更新模板，直接改为c的操作可以修改线段树模板

```

inline void upd1(int u,int v,int c) {
    int f1=top[u],f2=top[v];
    while(f1!=f2)
    {
        if(dep[f1]<dep[f2])
            swap(f1,f2),swap(u,v);
        update(w[f1],w[u],c,1,nodenum);
        u=fa[f1];
        f1=top[u];
    }
    if(dep[u]>dep[v]) swap(u,v);
    update(w[u],w[v],c,1,nodenum);
}

```

边更新

区间更新： upd2(u,v,c) 意为将u,v路径上的边权+=c

```

inline void upd(int u,int v,int c) {
    int f1=top[u],f2=top[v];
    while(f1!=f2)
    {
        if(dep[f1]<dep[f2])
            swap(f1,f2),swap(u,v);
        update(w[f1],w[u],c,1,nodenum);
        u=fa[f1];
        f1=top[u];
    }
    if(u==v) return;
    if(dep[u]>dep[v]) swap(u,v);
    update(w[son[u]],w[v],c,1,nodenum);
}

```

主席树

说明

修改自[树状结构之主席树](#)

模版

```
// HDU 2665
const int maxn=100000+5;
int a[maxn], b[maxn], sz; //原数据、离散化辅助数组、离散化后的元素数量
int rt[20*maxn], ls[20*maxn], rs[20*maxn], sum[20*maxn]; //主席树数组
int tot; //主席树总顶点数
int n, q;

void build(int& id, int l, int r){
    id = ++ tot;
    sum[id] = 0;
    if(l == r) return;
    int m = (l + r) >> 1;
    build(ls[id], l, m);
    build(rs[id], m + 1, r);
}

void update(int& id, int l, int r, int pre, int rk){
    id = ++ tot;
    ls[id] = ls[pre];
    rs[id] = rs[pre];
    sum[id] = sum[pre] + 1;
    if(l == r) return;
    int m = (l + r) >> 1;
    if(rk <= m) update(ls[id], l, m, ls[pre], rk);
    else update(rs[id], m + 1, r, rs[pre], rk);
}

int query(int ss, int tt, int l, int r, int k){
    if(l == r) return l;
    int m = (l + r) >> 1;
    int cnt = sum[ls[tt]] - sum[ls[ss]];
    if(k <= cnt) return query(ls[ss], ls[tt], l, m, k);
```

```

    else return query(rs[ss], rs[tt], m + 1, r, k - cnt);
}

int main() {
    int T;
    scanf("%d", &T);
    while(T--) {
        tot = 0; // 初始化

        scanf("%d%d", &n, &q);
        for(int i = 1; i <= n; i++) {
            scanf("%d", a + i);
            b[i] = a[i];
        }

        // 离散化
        stable_sort(b+1, b+1+n);
        int sz=unique(b+1, b+1+n)-b-1;
        for (int i=1; i<=n; i++)
            a[i]=lower_bound(b+1, b+sz+1, a[i])-b;

        build(rt[0],1, sz); //建树

        for(int i = 1; i <= n; i++)      //加点
            update(rt[i], 1, sz, rt[i - 1], a[i]);

        while(q--) {
            int L,R,K;
            scanf("%d%d%d", &L, &R, &K);
            int idx = query(rt[L - 1], rt[R], 1, sz, K);
            printf("%d\n", b[idx]); // 输出[L, R]第K大
        }
    }
    return 0;
}

```

背包问题

经典背包问题，[这个网站](#)讲的比较详细了。

网站链接：<http://love-oriented.com/pack/>

○一背包

说明

最简单的入门DP问题

给出一个容积为V的背包，以及 n 个物品，每个物品可以装入背包或者不装入背包，问背包内物品的最大价值是多少

f[]表示背包容积

w[]表示物品价值 下标1~n

c[]表示物品体积 下标1~n

使用方法

1. 输入 w[] , c[] , n , V
2. int ans = work(); 返回值即为最大价值

模版

```
const int maxv = 1005;
const int maxn = 105;
int f[maxv];
int w[maxn], c[maxn];
int n, V;
int work() {
    for (int i = 1; i <= n; ++i)
        for (int v = V; v >= c[i]; --v)
            f[v] = max(f[v], f[v - c[i]] + w[i]);
    return f[V];
}
```

完全背包

说明

最简单的入门DP问题

给出一个容积为V的背包，以及 n 个物品，每个物品可以装入背包任意次（可以为0次），问背包内物品的最大价值是多少

f[]表示背包容积

w[]表示物品价值 下标1~n

c[]表示物品体积 下标1~n

使用方法

1. 输入 w[], c[], n, V
2. int ans = work(); 返回值即为最大价值

模版

```
const int maxv = 1005;
const int maxn = 105;
int f[maxv];
int w[maxn], c[maxn];
int n, V;
int work() {
    for (int i = 1; i <= n; ++i)
        for (int v = c[i]; v <= V; v++)
            f[v] = max(f[v], f[v - c[i]] + w[i]);
    return f[V];
}
```

最大子段和

模版

```

int a[maxn], n;
int pre[maxn];
int dp[maxn] = { };
//输出子段起始点
while (scanf("%d", &n), n)
{
    for (int i=1; i<=n; i++)
        scanf("%d", &a[i]);
    memset(dp, -1, sizeof(dp));
    for (int i=1; i<=n; i++)
    {
        if (dp[i-1]<0)
        {
            pre[i]=i;
            dp[i]=a[i];
        }
        else
        {
            pre[i]=pre[i-1];
            dp[i]=dp[i-1]+a[i];
        }
    }
    int idx, mx=-1;
    for (int i=1; i<=n; i++)
        if (dp[i]>mx)
        {
            mx=dp[i];
            idx=i;
        }
    if (mx<0) printf("%d %d %d\n", 0, a[1], a[n]); //所有值均小于零
    else printf("%d %d %d\n", mx, a[pre[idx]], a[idx]);
}

```

最长上升子序列

模版

LIS1

复杂度: $O(n^2)$

`a[]` 下标为 $1 \sim n$, `dp[]` 表示以该点结束的最长上升子序列的长度

```
int dp[1050]={},cnt;
int LIS(int *a,int n){
    cnt = 0;
    for (int i=1; i<=n; i++) {
        dp[i]=1;
        for (int j=1; j<i; i++)
            if (a[j]<a[i])
                dp[i]=max(dp[i],dp[j]+1);
        cnt=max(cnt,dp[i]);
    }
    return cnt;
}
```

LIS2

复杂度: $O(n \log n)$

`a[]` 下标为 $1 \sim n$, `dp[]` 并不会存储最长上升子序列, 只是长度相等

最长上升子序列

```
const int maxn = 1005;
int dp[maxn];
int LIS(int a[], int n) {
    int len = 1;
    dp[1] = a[1];
    for (int i = 2; i <= n; i++) {
        if (a[i] > dp[len]) {
            dp[++len] = a[i];
        } else {
            int pos = lower_bound(dp + 1, dp + len + 1, a[i]) - dp;
            // 找到插入位置
            dp[pos] = a[i];
        }
    }
    return len;
}
```

最长公共子序列

模版

```
const int LEN=1005;
int dp[LEN][LEN],res;
int a[LEN],b[LEN];
int n,m;
int lcs()
{
    memset(dp,0,sizeof(dp));
    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            if (a[i]==b[j]) dp[i][j]=dp[i-1][j-1]+1;
            else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
    return res=dp[n][m];
}
int ans[LEN];
void getans()
{
    int num=res;
    for (int i=n,j=m; i>=1&&j>=1; )
        if (a[i]==b[j])
        {
            ans[num--]=a[i];
            i--;
            j--;
        }
        else
        {
            if (dp[i-1][j]>dp[i][j-1]) i--;
            else j--;
        }
}
```

最长公共上升子序列

说明

求数组a和数组b的最长公共上升子序列

a[] 的长度为n, 下标为 1~n

b[] 的长度为m, 下标为 1~m

模版

```

const int LEN=1005;
int dp[LEN][LEN],path[LEN][LEN],res;
int a[LEN],b[LEN];
int n,m,ai,aj;
int lcis()
{
    memset(dp,0,sizeof(dp));
    res=0;
    for (int i=1; i<=n; i++)
    {
        int mx=0,t;
        for (int j=1; j<=m; j++)
        {
            dp[i][j]=dp[i-1][j];
            path[i][j]=-1;
            if (a[i]>b[j] && mx<dp[i-1][j]) mx=dp[i-1][j],t=j;
            if (a[i]==b[j]) dp[i][j]=mx+1,path[i][j]=t;
            if(res<dp[i][j])
            {
                res=dp[i][j];
                ai=i;
                aj=j;
            }
        }
    }
    return res;
}
int ans[LEN];
void getans()
{
    int num=res;
    while(num)
    {
        if (path[ai][aj]!=-1)
        {
            ans[num--]=b[aj];
            aj=path[ai][aj];
        }
        ai--;
    }
}

```


矩阵链乘

说明

虽然acm中从来没遇到这个问题，不过这个对dp思想还是有点帮助，故把原来敲的板子放这了。

模版

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

#define N 6      //N表示矩阵的个数
#define MAXVALUE 1000000

void matrix_chain_order(int *p,int len,int m[N+1][N+1],int s[N+1][N+1]);
void print_optimal_parents(int s[N+1][N+1],int i,int j);

int main()
{
    int p[N+1] = {30,35,15,5,10,20,25}; //矩阵1大小为p[0]*p[1]
                                         //矩阵2大小为p[1]*p[2]
                                         //.....
                                         //矩阵n大小为p[n-1]*p[n]
    int m[N+1][N+1]={0}; //m数组用于储存决策代价
    int s[N+1][N+1]={0}; //s数组储存取最优解的时候的k的值
    int i,j;

    matrix_chain_order(p,N+1,m,s);

    cout<<"m value is: "<<endl;

    for(i=0;i<=N;++i)
    {
        for(j=1;j<=N;++j)
            printf("%5d ",m[i][j]);
        cout<<endl;
    }

    cout<<"s value is: "<<endl;
}
```

```

    for(i=1;i<=N;++i)
    {
        for(j=1;j<=N;++j)
            cout<<s[i][j]<<" ";
        cout<<endl;
    }

    cout<<"The result is:"<<endl;

    print_optimal_parents(s,1,N);
    return 0;
}

void matrix_chain_order(int *p,int len,int m[N+1][N+1],int s[N+1]
[N+1])
{
    int i,j,k,t;
    for(i=0;i<=N;++i)
        m[i][i] = 0;
    for(t=2;t<=N;t++) //当前链乘矩阵的长度
    {
        for(i=1;i<=N-t+1;i++) //从第一矩阵开始算起，计算长度为t的最少代价
        {
            j=i+t-1; //长度为t时候的最后一个元素
            m[i][j] = MAXVALUE; //初始化为最大代价
            for(k=i;k<=j-1;k++) //寻找最优的k值，使得分成两部分k在i与j-1之间
            {
                int temp = m[i][k]+m[k+1][j] + p[i-1]*p[k]*p[j];
                if(temp < m[i][j])
                {
                    m[i][j] = temp; //记录下当前的最小代价
                    s[i][j] = k; //记录当前的括号位置，即矩阵的编号
                }
            }
        }
    }

    //s中存放着括号当前的位置
    void print_optimal_parents(int s[N+1][N+1],int i,int j)
    {

```

```
if( i == j )
    cout<<"A"<<i;
else
{
    cout<<"(";
    print_optimal_parents(s,i,s[i][j]);
    print_optimal_parents(s,s[i][j]+1,j);
    cout<<") ";
}

}
```

Sparse – Table算法

说明

复杂度分析

- 预处理 $O(n \log n)$
- 查询 $O(1)$

使用时请灵活处理，模版仅给出了最简单的按照区间最值的查询，还可以灵活的换为下标找下标在数组中的映射，举例：[LCA->RMQ](#)

模版

最大/最小值查询

Tips

- 使用时请注意宏定义 RMQ
- `data[]` 下标从 1~n

代码

```

#define RMQ max
const int maxn = 150005; //数据量
const int NVB = 33; //对于整型而言，其值不会超过 $2^{32}$ ，因此第二维大小为33已经足够
int rmq[maxn][NVB];
void init(int data[], int n) {
    /*data下标从1开始*/
    int k = log2(n);
    for(int i=1; i<=n; i++)
        rmq[i][0] = data[i];
    for(int j=1; j<=k; j++) {
        for(int i=1; i+(1<<j)-1<=n; i++) {
            rmq[i][j] = RMQ(rmq[i][j-1], rmq[i+(1<<(j-1))][j-1]);
        }
    }
}
int query(int l, int r) {
    int k = log2(r-l+1);
    return RMQ(rmq[l][k], rmq[r-(1<<k)+1][k]);
}

```

二合一

Tips

- flag为1表示取最大值 0表示取最小值
- `data[]` 下标从1~n

代码

```

const int maxn = 150005; //数据量
const int NVB = 33; //对于整型而言，其值不会超过 $2^{32}$ ，因此第二维大小为33已经足够
int mx[maxn][NVB], mn[maxn][NVB];
void init(int data[], int n) {
    /*data下标从1开始*/
    int k = log2(n);
    for(int i=1; i<=n; i++) {
        mx[i][0] = mn[i][0] = data[i];
    }
    for(int j=1; j<=k; j++) {
        for(int i=1; i+(1<<j)-1<=n; i++) {
            mx[i][j] = max(mx[i][j-1], mx[i+(1<<(j-1))][j-1]);
            mn[i][j] = min(mn[i][j-1], mn[i+(1<<(j-1))][j-1]);
        }
    }
}
int query(int l, int r, int flag) {
    int k = log2(r-l+1);
    if(flag) return max(mx[l][k], mx[r-(1<<k)+1][k]);
    else return min(mn[l][k], mn[r-(1<<k)+1][k]);
}

```

二维RMQ

Tips

- `data[][]` 下标从1~n行 1~m列

代码

```

#define RMQ max
const int NV = 33; //对于整型而言，其值不会超过 $2^{32}$ ，因此第二维大小为33已经足够
const int maxn = 500; //数据量
int rmq[NV][NV][maxn][maxn];
void init(int data[][maxn], int n, int m) // 1~n行 1~m列
{
    int mx=floor(log(n+0.0)/log(2.0));
    int my=floor(log(m+0.0)/log(2.0));
    for(int i=1; i<=n; i++)
        for(int j=1; j<=m; j++)
            rmq[0][0][i][j]=data[i][j];
}

```

```

for(int i=0;i<=mx;i++) {
    for(int j=0;j<=my;j++) {
        if(i==0&&j==0) continue;
        for(int row=1;row+(1<<i)-1<=n;row++) {
            for(int col=1;col+(1<<j)-1<=m;col++) {
                if(i==0)
                    rmq[i][j][row][col]=RMQ(rmq[i][j-1][row]
[col]
                                         ,rmq[i][j-1][row]
[col+(1<<(j-1))]);
                else
                    rmq[i][j][row][col]=RMQ(rmq[i-1][j][row]
[col]
                                         ,rmq[i-1][j][row+
(1<<(i-1))][col]);
            }
        }
    }
}

int query(int x1,int y1,int x2,int y2)
{
    int mx=floor(log(x2-x1+1.0)/log(2.0));
    int my=floor(log(y2-y1+1.0)/log(2.0));

    int m1=rmq[mx][my][x1][y1];
    int m2=rmq[mx][my][x2-(1<<mx)+1][y2-(1<<my)+1];
    int m3=rmq[mx][my][x1][y2-(1<<my)+1];
    int m4=rmq[mx][my][x2-(1<<mx)+1][y1];

    return RMQ(RMQ(m1,m2),RMQ(m3,m4));
}

```

数位DP

题目描述

给定两个 10^{18} 范围内的数字 l, r ，问在 $[l, r]$ 区间内，有多少单调的数字，单调的数字的定义是：高位恒小于等于低位或低位恒小于等于高位。

解题思路

通过数位DP的方式，求出来所有高位恒小于等于低位的数的个数以及低位恒小于等于高位的数的个数，再减去所有位相等的数的个数

Tips

- 数位DP是通过记忆化搜索的方式，也就是说，求出来的位对所有数进行数位DP都适用，不需要每次都 `memset`，可以大幅度降低时间复杂度。
- 主要难点在于对状态的判断

代码

```
//BNU52325
int num[20], pos;
ll dp1[20][10];//20位表示数的长度 10表示前一位的值
ll dp2[20][10][2];//20位表示数的长度 10表示前一位的值 2表示是否是第一位
ll dfs1(int pos, int pre, bool limit){//统计单调递增 pos表示当前位的下标
    pre表示前一位的状态 limit表示是否受数字上限大小约束
    if (pos < 1) return 1;
    if (!limit && dp1[pos][pre] != -1) return dp1[pos][pre];
    int mx = limit?num[pos]:9;
    ll ret = 0;
    for (int i = pre; i <= mx ; i++) //保证当前位比前一位大
        ret += dfs1(pos-1, i, limit && i == num[pos]);
    if (!limit) dp1[pos][pre] = ret; //仅存储无约束条件的数 保证记忆化搜索的正确性
    return ret;
}
ll dfs2(int pos, int pre, bool limit, bool first){//单调递减 pos表示当前位的下标 pre表示前一位的状态 limit表示是否受数字上限大小约束 first表示是否是最高位
    if (pos < 1) return 1;
    if (!limit && dp2[pos][pre] != -1) return dp2[pos][pre];
    int mx = limit?num[pos]:9;
    ll ret = 0;
    for (int i = mx; i >= pre ; i--) //保证当前位比前一位小
        ret += dfs2(pos-1, i, limit && i == num[pos]);
    if (!limit) dp2[pos][pre] = ret;
    return ret;
}
```

```

    if (pos < 1) return 1;
    if (!limit && dp2[pos][pre][first] != -1) return dp2[pos][pre]
[first];
    int mx;
    if (first && limit) mx = num[pos];
    else if (first && !limit) mx = 9;
    else if (!first && limit) mx = min(num[pos], pre);
    else mx = pre;
    ll ret = 0;
    for (int i = 0; i <= mx; i++) //保证当前位比前一位小
        ret += dfs2(pos-1, i, limit && i == num[pos], first&&i==0);
    if (!limit) dp2[pos][pre][first] = ret; //仅存储无约束条件的数 保证
记忆化搜索的正确性
    return ret;
}
ll solve(ll x){
    ll tmp = x;
    pos = 0;
    while (x) {
        num[++pos] = x % 10;
        x /= 10;
    }
    ll ret = 0;
    ret += dfs1(pos, 0, true) - 1; //去除0
    ret += dfs2(pos, 9, true, true) - 1; //去除0
    ll bas = 1;
    while (tmp >= bas) { //去除所有位相等的情况 如111, 33333, 55, 6666
        ret -= min(9LL, tmp/bas);
        bas = bas * 10 + 1;
    }
    return ret;
}
int main(){
    //初始化记忆标记
    memset(dp1, -1, sizeof(dp1));
    memset(dp2, -1, sizeof(dp2));

    int n;
    scanf("%d", &n);
    while (n--) {
        ll l, r;
        scanf("%lld%lld", &l, &r);
        printf("%lld\n", solve(r) - solve(l-1));
    }
}

```

```
    }  
    return 0;  
}
```

快速乘

直接调用 `mult_mod(a, b, mod)` 即可计算 $a \cdot b \% \text{mod}$

```
ll mult_mod(ll a, ll b, ll mod) {
    return (a * b - (ll)(a / (long double)mod * b + 1e-3) * mod + mod) % mod;
}
```

快速幂

直接调用 `pow_mod(x, n, mod)` 即可计算 $x^n \% \text{mod}$

```
ll mult_mod(ll a, ll b, ll mod) {
    return (a * b - (ll)(a / (long double)mod * b + 1e-3) * mod + mod) % mod;
}

ll pow_mod(ll x, ll n, ll mod) { // x^n % c
    if (n == 1) return x % mod;
    x %= mod;
    ll tmp = x;
    ll ret = 1;
    while (n) {
        if (n & 1) ret = mult_mod(ret, tmp, mod);
        tmp = mult_mod(tmp, tmp, mod);
        n >>= 1;
    }
    return ret;
}
```

线性筛素数表

说明：

pcnt 记录素数的个数+1

prime[] 记录素数表 有效数据范围:1~pcnt-1

factor[i] 记录i的最小质数因子

Tips: 记得要**Init_Prime()**

```
typedef long long ll;
#define maxn 100
ll prime[maxn]={1};
int pcnt=0;//素数下标1~pcnt-1
int factor[maxn]={1,1}; //factor[i]表示i的最小质因子
void Init_Prime () {
    pcnt=1;
    for(ll i = 2 ; i < maxn ; i++) {
        if(!factor[i]) {
            prime[pcnt++]=i;
            factor[i]=i;
        }
        for(ll j = 1 ; j < pcnt && i * prime[j] < maxn ; j++)
        {
            factor[i * prime[j]] = prime[j];
            if( !(i % prime[j] ) )
                break;
        }
    }
    return;
}
```

莫比乌斯反演

莫比乌斯反演 (mobius) 的形式

$$\text{若 } F(n) = \sum_{d|n} f(d) , \text{ 那么 } f(n) = \sum_{d|n} \mu(d) F\left(\frac{n}{d}\right)$$

$$\text{若 } F(n) = \sum_{n|d} f(d) , \text{ 那么 } f(n) = \sum_{n|d} \mu\left(\frac{d}{n}\right) F(d)$$

其中 `mu` 的定义如下

$$\mu(d) = \begin{cases} 1 & d = 1 \\ (-1)^k & d = p_1 * p_2 * \dots * p_k, \text{ 其中 } p_1, p_2, \dots, p_k \text{ 是互异素数} \\ 0 & \text{其他} \end{cases}$$

性质：

1.

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & (n = 1) \\ 0 & (n > 1) \end{cases}$$

2.

$$\sum_{d|n} \frac{\mu(d)}{d} = \frac{\varphi(n)}{n}, \text{ 其中 } \varphi(n) \text{ 是欧拉函数}$$

代码求 `mu` :

```
int normal[maxn];
int mu[maxn];
int prime[maxn];
int pcnt;
void Init()
{
    memset(normal, 0, sizeof(normal));
    mu[1] = 1;
    pcnt = 0;
    for(int i=2; i<maxn; i++)
    {
        if(!normal[i])
        {
            prime[pcnt++] = i;
            mu[i] = -1;
        }
        for(int j=0; j<pcnt&&i*prime[j]<maxn; j++)
        {
            normal[i*prime[j]] = 1;
            if(i%prime[j]) mu[i*prime[j]] = -mu[i];
            else
            {
                mu[i*prime[j]] = 0;
                break;
            }
        }
    }
}
```

小型素数判断

使用方法

1. 打素数表时需要调用 `Init_Prime()`
2. `bool isprime = isPrime(n);`

模版

无素数表时

```
bool isPrime(int n)
{
    if(n==2) return 1;
    if(n%2==0 || n<2) return 0;
    int l=sqrt(n+1);
    for(int i=3;i<=l;i+=2)
        if(n%i==0) return 0;
    return 1;
}
```

有素数表时

说明： 素数表为`prime[i]` 下标由1~pcnt-1

```
ll prime[maxn]={1};  
int pcnt=0;//素数下标1~pcnt-1  
int factor[maxn]={1,1}; //factor[i]表示i的最小质因子  
void Init_Prime () {  
    pcnt=1;  
    for(ll i = 2 ; i < maxn ; i ++){  
        if(!factor[i]) {  
            prime[pcnt++]=i;  
            factor[i]=i;  
        }  
        for(ll j = 1 ; j < pcnt && i * prime[j] < maxn ; j ++)  
        {  
            factor[i * prime[j]] = prime[j];  
            if( !(i % prime[j] ) )  
                break;  
        }  
    }  
    return;  
}  
bool isPrime(int n)  
{  
    if(n==2) return 1;  
    if(n%2==0 || n<2) return 0;  
    int l=sqrt(n+1);  
    for(int i=2;prime[i]<=l;i++)  
        if(n%prime[i]==0) return 0;  
    return 1;  
}
```

Miller_Rabbin

Miller_Rabbin 算法进行素数测试

说明：

速度快，而且可以判断 $<2^{63}$ 的数

复杂度： $O(\log N) * \text{测试次数}$ 由于要处理long long，所以略慢

是素数返回true 合数返回false;

使用方法：

- bool isPrime = Miller_Rabbin(x)

Tips:

- 需要stdlib.h头文件
- 可能是伪素数，但概率极小

```

const int S = 20; //随机算法判定次数, S越大, 判错概率越小
ll mult_mod(ll a,ll b,ll mod) {
    return (a*b-(ll)(a/(long double)mod)*b+1e-3)*mod+mod;
}

//计算 x^n %c
ll pow_mod(ll x, ll n, ll mod) { //x^n%c
    if(n == 1) return x % mod;
    x %= mod;
    ll tmp = x;
    ll ret = 1;
    while(n) {
        if(n & 1) ret = mult_mod(ret, tmp, mod);
        tmp = mult_mod(tmp, tmp, mod);
        n >>= 1;
    }
    return ret;
}

//以a为基, n-1=x*2^t      a^(n-1)=1 (mod n)  验证n是不是合数
//一定是合数返回true, 不一定返回false
bool check(ll a, ll n, ll x, ll t) {
    ll ret = pow_mod(a, x, n);
    ll last = ret;

```

```

    for(int i = 1; i <= t; i++) {
        ret = mult_mod(ret, ret, n);
        if(ret == 1 && last != 1 && last != n - 1) return true; //合数
        last = ret;
    }
    if(ret != 1) return true;
    return false;
}

// Miller_Rabin() 算法素数判定
//

bool Miller_Rabin(ll n) {
    if(n < 2) return false;
    if(n == 2) return true;
    if((n & 1) == 0) return false; //偶数
    ll x = n - 1;
    ll t = 0;
    while((x & 1) == 0) {
        x >>= 1;
        t++;
    }
    for(int i = 0; i < S; i++) {
        ll a = rand() % (n - 1) + 1; //rand()需要stdlib.h头文件
        if(check(a, n, x, t))
            return false; //合数
    }
    return true;
}

```

拓展欧几里得

说明

关于linear_equation()的一些说明: 通过linear_equation()返回值为该线性方程是否有整数解。参数中返回的x,y是 $ax+by=c$ 的一组解

根据方程可知 $\Rightarrow a(X+x) + b(Y+y) = c$
 $\Rightarrow aX + bY = 0$
 $\Rightarrow X/b = -Y/a$
 \Rightarrow 设参数t使 $X=t \cdot b / \gcd(a, b)$
 $Y=-t \cdot a / \gcd(a, b)$
(t为整数)

由此可以构建线性方程 $ax+by=c$ 的所有解

模版

```
//拓展欧几里德求解ax+by=gcd(a,b)的一组解
ll ex_gcd(ll a,ll b,ll &x,ll &y) {
    if (b==0) {
        x=1, y=0;
        return a;
    }
    else {
        ll r=ex_gcd(b,a%b,y,x);
        y-=x*(a/b);
        return r;
    }
}

//返回的bool值表示该方程是否有解
bool linear_equation(int a,int b,int c,int &x,int &y) {
    int d=ex_gcd(a,b,x,y);
    if (c%d) return false;
    int k=c/d;
    x*=k; y*=k;
    return true;
}
```


中国剩余定理

说明

中国剩余定理用于求模方程组 方程组形如 $x \% \text{divisor}[i] = \text{rest}[i]$

使用方法

1. 将n个方程的模域和余数存入divisor[]和rest[]中，下标从0~n-1
2. `x = CRT1(n)` 返回值即为一个符合条件的解，其它符合条件的解为 $x + \prod(\text{divisor}[i])$

Tips

1. 考虑数据范围，如果求模域的时候可能爆long long请使用快速乘
2. 模域是否两两互质下面 `CRT1` 和 `CRT2` 对号入座

```

const int maxn=1000+5;
typedef long long ll;
ll mult_mod(ll a,ll b,ll mod) {
    return (a*b-(ll)(a/(long double)mod)*b+1e-3)*mod+mod;
}

ll ex_gcd(ll a,ll b,ll &x,ll &y) {
    if (b==0) {
        x=1,y=0;
        return a;
    }
    else{
        ll res=ex_gcd(b,a%b,y,x);
        y-=x*(a/b);
        return res;
    }
}

// 中国剩余定理 模域两两互质 x%(divisor[i])=rest[i]
ll CRT1 (ll divisor[],ll rest[], int n){      //n表示有n个方程, 0~n-1
    ll gcd,tmp,product=1,res=0,x,y;
    for (int i=0; i<n; i++)
        product*=divisor[i];
    for (int i=0; i<n; i++) {

```

```

    tmp=product/divisor[i];
    gcd=ex_gcd(divisor[i], tmp, x, y);
    res = (res + mult_mod(mult_mod(y, tmp, product), rest[i],
product))%product;//防爆
//          res=(res+y*tmp%product*rest[i]%product)%product;
}
return (res+ product)%product;//返回值为符合模方程组的解
}

// 中国剩余定理非互质版 有解返回解，无解返回-1 x%(divisor[i])=rest[i]
ll CRT2(ll divisor[], ll rest[], int n) { //n表示有n个方程, 0~n-1
if (n == 1) {
    if (divisor[0] > rest[0]) return rest[0];
    else return -1;
}
ll x, y, d;
for (int i = 1; i < n; i++) {
    if (divisor[i] <= rest[i]) return -1;
    d = ex_gcd(divisor[0], divisor[i], x, y);
    if ((rest[i] - rest[0]) % d != 0) return -1;
    ll t = divisor[i] / d;
    x = ((rest[i] - rest[0]) / d * x % t + t) % t;
    rest[0] = x * divisor[0] + rest[0];
    divisor[0] = divisor[0] * divisor[i] / d;
    rest[0] = (rest[0] % divisor[0] + divisor[0]) % divisor[0];
}
return rest[0];
}

```

单变元模线性方程

说明

已知 a, b, n , 求 x , 使得 $ax \equiv b \pmod{n}$

输入 a, b, n 输出所有 $[0, n)$ 中的解(个数为 $\gcd(a, n)$)

要注意在处理过程中, 如果 b 是负数, 需要转换: $b = ((b \% n) + n) \% n;$

复杂度: $O(\log N)$

使用方法

- `vector<ll> ans = line_mod_equation(a, b, n);`

模版

```

ll ex_gcd(ll a,ll b,ll &x,ll &y){
    if (b==0) {
        x=1,y=0;
        return a;
    }
    else{
        ll r=ex_gcd(b,a%b,y,x);
        y-=x*(a/b);
        return r;
    }
}

vector<ll> line_mod_equation(ll a, ll b, ll n) {
    ll x, y;
    ll d = extend_gcd(a, n, x, y);
    vector<ll> ans;
    ans.clear();
    if (b % d == 0) {
        x = (x % n + n) % n;
        x %= (n / d);
        ans.push_back(x * (b / d) % (n / d));
        for (ll i = 1; i < d; i++) ans.push_back((ans[0] + i * n /
d) % n);
    }
    return ans;
}

```

逆元

说明

有的题目要求结果mod一个大质数，如果原本的结果中有除法，比如除以a，那就就可以乘以a的逆元替代。

费马小定理求逆元 p为素数

说明

费马小定理求逆元 $ax \equiv 1 \pmod{p}$ 其中p为质数

复杂度：O ($\log N$)

使用方法

```
ll ans = inv(a);
```

模版

```
const int mod = 1000000009;
long long quickpow(long long a, long long b) {
    if (b < 0) return 0;
    long long ret = 1;
    a %= mod;
    while (b) {
        if (b & 1) ret = (ret * a) % mod;
        b >>= 1;
        a = (a * a) % mod;
    }
    return ret;
}
long long inv(long long a) {
    return quickpow(a, mod - 2);
}
```

扩展欧几里得算法求逆元 a,n互质

说明

扩展欧几里得算法求逆元 $ax \equiv 1 \pmod{n}$ 其中a,n互质

复杂度: $O(\log N)$

使用方法

```
ll ans = inv(a, mod);
```

模版

```
ll extend_gcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    else {
        ll r = extend_gcd(b, a % b, y, x);
        y -= x * (a / b);
        return r;
    }
}

ll inv(ll a, ll n) {
    ll x, y;
    extend_gcd(a, n, x, y);
    x = (x % n + n) % n;
    return x;
}
```

逆元线性筛 (P为质数)

说明

求 $1, 2, 3, \dots, N$ 关于P的逆元 (P为质数)

复杂度: $O(N)$

使用方法

```
ll ans = inv(a);
```

模版

```
const int mod = 1000000009;
const int maxn = 10005;
int inv[maxn];
inv[1] = 1;
for(int i = 2; i < 10000; i++)
    inv[i] = inv[mod % i] * (mod - mod / i) % mod;
```

组合数打表

Tips

- 请注意是否需要取模

模版

```
const int maxn=20;
long long c[maxn][maxn] = { };
void cinit()
{
    for(int i=0; i<maxn; i++)
    {
        c[i][0]=c[i][i]=1;
        for(int j=1; j<i; j++)
            c[i][j]=(c[i-1][j]+c[i-1][j-1])%mod;
    }
}
```

逆元求组合数

说明

- mod 必须为素数

该方法使用求逆元的方式求组合数

使用方法

1. `getfac();` 打表
2. `ll ans = C(n, m);` 计算组合数 $C(n, m)$;

模版

```
const int fcnt=100005;
const ll mod=1000000007;
ll fac[fcnt];
void getfac()
{
    fac[0]=1;
    for (int i=1; i<fcnt; i++)
        fac[i]=fac[i-1]*i%mod;
}
ll quickpow(ll a, ll b) {
    if (b < 0) return 0;
    ll ret = 1;
    a %= mod;
    while(b) {
        if (b & 1) ret = (ret * a) % mod;
        b >>= 1;
        a = (a * a) % mod;
    }
    return ret;
}
ll inv(ll a) {
    return quickpow(a, mod - 2);
}
ll C(ll n,ll m)
{
    if (n<m)
        return 0;
    return fac[n]*inv(fac[m])%mod*inv(fac[n-m])%mod;
}
```

Lucas求组合数

说明： 1. 数较小且mod较大时求组合数使用逆元，数较大且mod较小时求组合数用Lucas 2. 该模版只可以求对于正数的组合数，如果出现负数的情况则返回0

使用方法 1. init(); 2. Lucas(n, m);//求组合数

```

const long long mod=110119;
const int fcnt=120005;
long long fac[fcnt];
long long inv[fcnt];
ll powMod(ll a, ll b) {
    ll ans = 1;
    for( a%=mod; b; b>>=1, a = a * a % mod)
        if(b&1)    ans = ans * a % mod;
    return ans;
}
void init() {
    fac[0] = 1;
    for(int i = 1; i < mod; i++)
        fac[i] = fac[i-1] * i % mod;
    inv[mod - 1]=powMod( fac[mod - 1], mod - 2);
    for(int i = mod-2; i >= 0 ; i--)
        inv[i] = inv[i+1] * (i+1) % mod;
}
ll C(ll n, ll m) {
    return m > n ? 0 : fac[n] * inv[m] % mod * inv[n-m] % mod;
}
ll Lucas(ll n, ll m){ // n>m
    if(n<0 || m<0 || n<m) return 0;
    return m ? (C(n%mod , m%mod) * Lucas(n/mod, m/mod)) % mod : 1;
}

```

欧拉函数

说明

欧拉函数：求从1到p-1中与p互质的数字个数

应用

$$a^b \% p = a^{\phi(p) + b \% \phi(b)} \% p$$

欧拉函数

使用方法

- `ll ans = euler(x);` 计算x的欧拉函数值

模版

```
ll euler(ll x) {
    ll res = x;
    for (ll i = 2; i <= x / i; i++)
        if (x % i == 0) {
            res = res / i * (i - 1);
            while (x % i == 0) x /= i;
        }
    if (x > 1) res = res / x * (x - 1);
    return res;
}
```

欧拉函数值表

说明

计算1-n所有数的欧拉phi函数值

时间复杂度O(nloglongn)

模版

```
void phi_table(int n, int *phi)
{
    //memset(phi, 0, sizeof(phi)); //指针不能用这个初始化
    for(int i=2; i<=n; i++) phi[i]=0;
    phi[1]=1;
    for(int i=2; i<=n; i++)
    {
        if(!phi[i])
            for(int j=i; j<=n; j+=i)
            {
                if(!phi[j]) phi[j]=j;
                phi[j]=phi[j]/i*(i-1);
            }
    }
}
```

高斯消元

高斯消元－整数

说明

使用高斯消元法求解浮点数方程组

kuangbin神的板子

高斯消元法解方程组(Gauss-Jordan elimination).(-2表示有浮点数解，但无整数解，-1表示无解，0表示唯一解，大于0表示无穷解，并返回自由变元的个数)，有equ个方程，var个变元。增广矩阵行数为equ,分别为0到equ-1,列数为var+1,分别为0到var

使用方法

1. 系数矩阵放在a[][]中，a[0~equ-1][var]存放结果
2. int tmp = Gauss(); 返回值说明：-2表示有浮点数解，但无整数解，-1表示无解，0表示唯一解，大于0表示无穷解，并返回自由变元的个数
3. 答案存放在x[0~var-1]中

模版

```
#include<stdio.h>
#include<algorithm>
#include<iostream>
#include<string.h>
#include<math.h>
using namespace std;
const int maxn=50;
int a[maxn][maxn]; //增广矩阵
int x[maxn]; //解集
bool free_x[maxn]; //标记是否是不确定的变元

inline int gcd(int a,int b)
{
    int t;
    while(b!=0)
    {
        t=b;
        b=a%b;
        a=t;
    }
}
```

```

    a=t;
}
return a;
}

inline int lcm(int a,int b)
{
    return a/gcd(a,b)*b;//先除后乘防溢出
}

int Gauss(int equ,int var)
{
    int i,j,k;
    int max_r;// 当前这列绝对值最大的行.
    int col;//当前处理的列
    int ta,tb;
    int LCM;
    int temp;
    int free_x_num;
    int free_index;

    for(int i=0;i<=var;i++)
    {
        x[i]=0;
        free_x[i]=true;
    }

    //转换为阶梯阵.
    col=0; // 当前处理的列
    for(k = 0;k < equ && col < var;k++,col++)
    {// 枚举当前处理的行.
        // 找到该col列元素绝对值最大的那行与第k行交换.(为了在除法时减小误差)
        max_r=k;
        for(i=k+1;i<equ;i++)
        {
            if(abs(a[i][col])>abs(a[max_r][col])) max_r=i;
        }
        if(max_r!=k)
        {// 与第k行交换.
            for(j=k;j<var+1;j++) swap(a[k][j],a[max_r][j]);
        }
        if(a[k][col]==0)
        {// 说明该col列第k行以下全是0了, 则处理当前行的下一列.
    
```

```

        k--;
        continue;
    }

    for(i=k+1;i<equ;i++)
    { // 枚举要删去的行.
        if(a[i][col]!=0)
        {
            LCM = lcm(abs(a[i][col]),abs(a[k][col]));
            ta = LCM/abs(a[i][col]);
            tb = LCM/abs(a[k][col]);
            if(a[i][col]*a[k][col]<0) tb=-tb; // 异号的情况是相加
            for(j=col;j<var+1;j++)
            {
                a[i][j] = a[i][j]*ta-a[k][j]*tb;
            }
        }
    }

    // Debug();
}

// 1. 无解的情况：化简的增广阵中存在(0, 0, ..., a)这样的行(a != 0).
for (i = k; i < equ; i++)
{ // 对于无穷解来说，如果要判断哪些是自由变元，那么初等行变换中的交换就会影响记录交换.
    if (a[i][col] != 0) return -1;
}

// 2. 无穷解的情况：在var * (var + 1)的增广阵中出现(0, 0, ..., 0)这样
// 即说明没有形成严格的上三角阵。
// 且出现的行数即为自由变元的个数。
if (k < var)
{
    // 首先，自由变元有var - k个，即不确定的变元至少有var - k个。
    for (i = k - 1; i >= 0; i--)
    {
        // 第i行一定不会是(0, 0, ..., 0)的情况，因为这样的行是在第k行到
        // u行。
        // 同样，第i行一定不会是(0, 0, ..., a), a != 0的情况，这样的
        // 的。
        free_x_num = 0; // 用于判断该行中的不确定的变元的个数，如果超过
        // 则无法求解，它们仍然为不确定的变元。
        for (j = 0; j < var; j++)
        {

```

```

        if (a[i][j] != 0 && free_x[j]) free_x_num++,
free_index = j;
    }
    if (free_x_num > 1) continue; // 无法求解出确定的变元.
    // 说明就只有一个不确定的变元free_index, 那么可以求解出该变元,
且该变元是确定的.

    temp = a[i][var];
    for (j = 0; j < var; j++)
    {
        if (a[i][j] != 0 && j != free_index) temp -= a[i][j]
* x[j];
    }
    x[free_index] = temp / a[i][free_index]; // 求出该变元.
    free_x[free_index] = 0; // 该变元是确定的.
}
return var - k; // 自由变元有var - k个.
}

// 3. 唯一解的情况: 在var * (var + 1)的增广阵中形成严格的上三角阵.
// 计算出Xn-1, Xn-2 ... X0.

for (i = var - 1; i >= 0; i--)
{
    temp = a[i][var];
    for (j = i + 1; j < var; j++)
    {
        if (a[i][j] != 0) temp -= a[i][j] * x[j];
    }
    if (temp % a[i][i] != 0) return -2; // 说明有浮点数解, 但无整数
解.
    x[i] = temp / a[i][i];
}
return 0;
}

int main(void)
{
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    int i, j;
    int equ, var;
    while (scanf("%d %d", &equ, &var) != EOF)
    {
        memset(a, 0, sizeof(a));
        for (i = 0; i < equ; i++)
        {

```

```

    for (j = 0; j < var + 1; j++)
    {
        scanf("%d", &a[i][j]);
    }
}

// Debug();

int free_num = Gauss(equ, var);
if (free_num == -1) printf("无解!\n");
else if (free_num == -2) printf("有浮点数解, 无整数解!\n");
else if (free_num > 0)
{
    printf("无穷多解! 自由变元个数为%d\n", free_num);
    for (i = 0; i < var; i++)
    {
        if (free_x[i]) printf("x%d 是不确定的\n", i + 1);
        else printf("x%d: %d\n", i + 1, x[i]);
    }
}
else
{
    for (i = 0; i < var; i++)
    {
        printf("x%d: %d\n", i + 1, x[i]);
    }
    printf("\n");
}
return 0;
}

```

高斯消元－浮点

说明

使用高斯消元法求解浮点数方程组

kuangbin神的板子

使用方法

1. 给 equ 和 var 赋值，分别表示等式个数（行数），变量个数（列数）
2. 系数矩阵放在a[][]中，a[0~equ-1][var]存放结果
3. int tmp = Gauss(); 返回0表示无解,1表示有解
4. 答案存放在x[0~var-1]中

模版

```

const double eps=1e-9;
const int maxn=220;
double a[maxn][maxn],x[maxn];//方程的左边的矩阵和等式右边的值,求解之后x存
的就是结果
int equ,var;//方程数和未知数个数
int Gauss() //返回0表示无解,1表示有解
{
    int i,j,k,col,max_r;
    for(k=0,col=0; k<equ&&col<var; k++,col++)
    {
        max_r=k ;
        for(i=k+1; i<equ; i++)
            if(fabs(a[i][col])>fabs(a[max_r][col]))
                max_r=i ;
        if(fabs(a[max_r][col])<eps)
            return 0;
        if(k!=max_r)
        {
            for(j=col; j< var; j++)
                swap(a[k][j],a[max_r][j]);
            swap(x[k],x[max_r]);
        }
        x[k]/=a[k][col];
        for(j=col+1; j<var; j++)
            a[k][j]/= a[k][col];
        a[k][col]=1;
        for(i=0; i<equ; i++)
            if(i!=k)
            {
                x[i]-=x[k]*a[i][k];
                for(j=col+1; j<var; j++)
                    a[i][j]-=a[k][j]*a[i][col];
                a[i][col]=0;
            }
        }
        return 1;
}

```

高斯消元－异或

模版

```

const int maxn=50;
int equ,var;//等式数,变量数
int a[maxn][maxn],x[maxn];//矩阵(多一列最终状态),答案
int free_x[maxn],free_num;//自由变元
int Gauss()
{
    int max_r,col,k;
    free_num=0;
    for(k=0,col=0;k<equ&&col<var;k++,col++)
    {
        max_r=k;
        for(int i=k+1;i<equ;i++)
        {
            if(abs(a[i][col])>abs(a[max_r][col]))
                max_r=i;
        }
        if(a[max_r][col]==0)
        {
            k--;
            free_x[free_num++]=col;
            continue;
        }
        if(max_r!=k)
        {
            for(int j=col;j<var+1;j++)
                swap(a[k][j],a[max_r][j]);
        }
        for(int i=k+1;i<equ;i++)
        {
            if(a[i][col]!=0)
            {
                for(int j=col;j<var+1;j++)
                    a[i][j]^=a[k][j];
            }
        }
    }
}

```

```
for(int i=k;i<equ;i++)
    if(a[i][col]!=0)
        return -1;
if(k<var) return var-k;
for(int i=var-1;i>=0;i--)
{
    x[i]=a[i][var];
    for(int j=i+1;j<var;j++)
        x[i]^=(a[i][j]&&x[j]);
}
return 0;
```

FFT多项式乘法

By 饭团

说明

转自 [多项式乘法运算初级版——ACdreamer](#)

模版

```
//len长度为原长度的两倍
//while(len < 2*len1 || len < 2*len2) len <<= 1;
struct Virt
{
    double r, i;//个人认为这里是r 是实部 i是虚部
    Virt(double r = 0.0, double i = 0.0)
    {
        this->r = r;
        this->i = i;
    }
    Virt operator + (const Virt &x)
    {
        return Virt(r + x.r, i + x.i);
    }
    Virt operator - (const Virt &x)
    {
        return Virt(r - x.r, i - x.i);
    }
    Virt operator * (const Virt &x)
    {
        return Virt(r * x.r - i * x.i, i * x.r + r * x.i);
    }
};

//雷德算法--倒位序
void Rader(Virt F[], int len)
{
    int j = len >> 1;
    for(int i=1; i<len-1; i++)
    {
        if(i < j) swap(F[i], F[j]);
    }
}
```

```

int k = len >> 1;
while(j >= k)
{
    j -= k;
    k >= 1;
}
if(j < k) j += k;
}

//FFT实现
void FFT(Virt F[], int len, int on)
{
    Rader(F, len);
    for(int h=2; h<=len; h<<=1) //分治后计算长度为h的DFT
    {
        Virt wn(cos(-on*2*PI/h), sin(-on*2*PI/h)); //单位复根
        e^(2*PI/m)用欧拉公式展开
        for(int j=0; j<len; j+=h)
        {
            Virt w(1, 0); //旋转因子
            for(int k=j; k<j+h/2; k++)
            {
                Virt u = F[k];
                Virt t = w * F[k + h / 2];
                F[k] = u + t; //蝴蝶合并操作
                F[k + h / 2] = u - t;
                w = w * wn; //更新旋转因子
            }
        }
        if(on == -1)
            for(int i=0; i<len; i++)
                F[i].r /= len;
    }

    //求卷积
    void Conv(Virt a[], Virt b[], int len)
    {
        FFT(a, len, 1);
        FFT(b, len, 1);
        for(int i=0; i<len; i++)
            a[i] = a[i]*b[i];
        FFT(a, len, -1);
    }
}

```

```
void Work()
{
    Conv(va,vb,len);
    for(int i=0; i<len; i++)
        result[i] = va[i].r+0.5;
}
```

Simpson

模版

```
/*
求某一段的积分
*/

double F(double x)
{
    ///积分函数
}

/// 3-points simpson

double simpson(double a, double b)
{
    double c=(a+b)/2;
    return (F(a)+4*F(c)+F(b)) * (b-a) / 6.;
}

double asr(double a, double b, double eps, double A)
{
    double c=(a+b)/2;
    double L=simpson(a,c), R=simpson(c,b);
    if(fabs(L+R-A)<=15*eps) return L+R+(L+R-A)/15.0;
    return asr(a,c,eps/2,L)+asr(c,b,eps/2,R);
}

double ASR(double a, double b, double eps)
{
    return asr(a,b,eps,simpson(a,b));
}
```

拓扑排序

说明

拓扑排序用于处理有向图顺序的问题，比如在到达一个顶点之前必须要先到达一个顶点。

BFS

说明

将所有入度为0的点全都push进队列，然后对队列中的每一个点u执行两个操作。

1. 将其排入topo数组之中
2. 找对应的临接点v，并将这条边删除(表现为将v的入度减一)，如果此时v的入度为0了，则将v点Push进队列。当队列为空时，topo数组即为排好序的序列。

使用方法

1. init() 初始化
2. head[u].push_back(v); indegree[v]++; 顶点下标从1~n，表示完成v之前必须要完成u
3. bool ok = toposort(); //返回值为false表示有环
4. 得到topo[]即为排好序的数组

Tips

- 下标必须从1开始

模版

```
//下标从1~n
const maxn=1000;
int n,m;//n个点 m条边
vector<int>head[maxn];
int indegree[maxn];//记录入度
int topo[maxn];
void init(){
    for (int i=0; i<=n; i++) {
        head[i].clear();
        indegree[i]=0;
    }
}
```

```

        topo[i]=0;
    }
}

bool toposort() {
    int index=1, cnt=0;
    priority_queue<int>q;
    for (int i=1; i<=n; i++) {
        if (indegree[i]==0) {
            q.push(i);
        }
    }
    if (q.empty()) return false;
    while (!q.empty()) {
        cnt++;
        int u=q.top();
        q.pop();
        topo[index++]=u;
        int sz=head[u].size();
        for (int i=0; i<sz; i++) {
            int v=head[u][i];
            if (--indegree[v]==0) {
                q.push(v);
            }
        }
    }
    if (cnt==n) return true;
    else return false;
}

void showAns() {
    for (int i=1; i<=n; i++) {
        printf("%d%c", topo[i], i==n?'\\n':' ');
    }
}

```

DFS

说明

总共n个点(0~n-1), m组关系。

t: topo数组反向记录时的位置下标。

G数组: 邻接矩阵 G[u][v]=1表示 u到v单向连通。

c数组：0表示未访问，1表示已找过，-1表示在DFS递归中。

topo数组：记录拓扑排序后的结果。

使用举例：[UVa 10305 Ordering Tasks](#)

使用方法

1. `memset(G, 0, sizeof(G));` 清空G数组
2. `adde(u, v);` 加一条从u指向v的边，顶点下标从1~n，但是在 G 中存是从 0~n-1 的
3. `toposort();` 调用该函数进行排序，返回值为false表示有环，否则表示没有环

模版

```
//下标1~n
const maxn=1000;
int G[maxn][maxn];//邻接矩阵
int n,m;
int c[maxn];
int topo[maxn],t;
void adde(int u, int v) {
    G[u-1][v-1]=1;
}
bool dfs(int u) {
    c[u]=-1;//访问标志
    for (int v=0; v<n; v++)
        if (G[u][v])
            if (c[v]<0) return false;
            else if (!c[v] && !dfs(v)) return false;
    c[u]=1;
    topo[--t]=u;
    return true;
}
bool toposort() {
    t=n;
    memset(c,0,sizeof(c));
    for (int u=0; u<n; u++)
        if (!c[u] && !dfs(u)) return false;
    return true;
}
void showAns() {
    for (int i=0; i<n; i++) {
        printf("%d%c",topo[i]+1,i==n-1?\n:' ');
    }
}
```

最短路

Todo List:

Bellman-Ford没有写

朴素版的Dijkstra没有写

Dijkstra

Dijkstra_Heap

说明

Dijkstra_Heap使用优先队列，比Dijkstra快一点，最快的还是SPFA

使用方法

1. Dijkstra d;
2. d.init(n); 顶点为n
3. d.adde(x,y,w); 加边
4. cout<<d.get_ans(int source,int destination)<<endl 输出最短路的值

Tips

待写

模版

```

const int maxn=100000+5; //最大顶点数
const int INF=0x3f3f3f3f;
using namespace std;
struct Edge{
    int from,to, dist;
    Edge(int u,int v,int d):from(u),to(v),dist(d){}
};

struct HeapNode{
    int d,u;
    bool operator < (const HeapNode& rhs) const{
        return d>rhs.d;
    }
};

struct Dijkstra{
    int n,m;
    vector<Edge> edges;
    vector<int> G[maxn];
    bool done[maxn];           //是否已永久标号
    int d[maxn];               //s(源点)到各个点的距离
    int p[maxn];               //最短路中的上一条弧
};

```

```

void init(int n) {
    this->n=n;
    for(int i=0;i<=n;i++) G[i].clear();
    edges.clear();
}

void adde(int from,int to,int dist) {
    edges.push_back(Edge(from-1, to-1, dist));
    m=edges.size();
    G[from-1].push_back(m-1);
}

void dijkstra(int s) {
    priority_queue<HeapNode> q;
    for(int i=0;i<n;i++) d[i]=INF;
    d[s]=0;
    memset(done, 0, sizeof(done));
    q.push((HeapNode){0,s});
    while(!q.empty()) {
        HeapNode x=q.top(); q.pop();
        int u=x.u;
        if(done[u]) continue;
        done[u]=true;
        for(int i=0;i<G[u].size();i++) {
            Edge& e=edges[G[u][i]];
            if(d[e.to]>d[u]+e.dist) {
                d[e.to]=d[u]+e.dist;
                p[e.to]=G[u][i];
                q.push((HeapNode){d[e.to],e.to});
            }
        }
    }
}

int get_ans(int source,int destination) {
    dijkstra(source-1);
    return d[destination-1];
}
};

```


Bellman-Ford

Floyd

SPFA

说明

SPFA是Bellman-Ford的进化版

SPFA在网络流中也有应用

可以用于判环，同样可以判环的还有拓扑排序

使用方法

1. 确定点边的最大数量 Vmax Emax
2. `init()`
3. `adde(x, y, w); adde(y, x, w); 双向加边`
4. `cout<<get_ans(int n, int source, int destination)<<endl` n个点、起点、终点

Tips

- `pre[]` 数组未验证，请使用时注意

模版

```
const int INF=0x3f3f3f3f;
typedef int mytype;
const int Vmax = 500+10;
const int Emax = 1000+10;//未翻倍时
int he[Vmax],ecnt;
struct edge{
    int v,next;
    mytype w;
}e[Emax*2];
int dis[Vmax];
int vcnt[Vmax]; //记录每个点进队次数，用于判断是否出现负环
bool inq[Vmax];
int pre[Vmax]; //记录最短路中的上一个节点
void init() {
    // 邻接表初始化
    ecnt=0;
    memset(he, -1, sizeof(he));
```

```

// SPFA初始化
memset(inq, false, sizeof(inq));
memset(vcnt, 0, sizeof(vcnt));
memset(dis, INF, sizeof(dis)); //即使是dis类型为long long也可赋
int型的INF
}
//***注意双向加边
void adde(int from,int to,mytype w) {
    e[ecnt].v=to;
    e[ecnt].w=w;
    e[ecnt].next=he[from];
    he[from]=ecnt++;
}
bool SPFA(int n,int source){ //n为顶点数 source为起点
    //return true表示无负环, 反之亦然
    dis[source]=0;
    queue<int>q;
    q.push(source);inq[source]=true;

    while (!q.empty()) {
        int tmp=q.front();
        q.pop();inq[tmp]=false;

        //判断负环
        vcnt[tmp]++;
        if (vcnt[tmp]>=n) return false;

        for (int i=he[tmp]; i!= -1; i=e[i].next) {
            int w=e[i].w;
            int v=e[i].v;
            if (dis[tmp]+w<dis[v]) {
                dis[v]=dis[tmp]+w; //松弛操作
                pre[v]=tmp;
                if (!inq[v]) {
                    q.push(v);
                    inq[v]=true;
                }
            }
        }
    }
    return true;
}

```

SPFA

```
mytype get_ans(int n,int source,int destination) {
    SPFA(n,source);
    return dis[destination];
}
```

最小生成树

Prim

说明

找对于到每一个点最近边的长度，存在对应下标的dis数组中

稀疏图较快

使用方法

1. Prim p;
2. p.init(vsz, source); 初始化顶点个数，起始点位置
3. p.adde(u, v, w); 从u指向v权值为w的边，加一次边即可
4. int weight = p.prim(); 输出最小生成树的重量，无法建成则返回-1

Tips

最好用Prim_Heap，相同条件下更快

考虑数据范围，树的重量可能会爆int

模版

```
#define Vmax 999
#define INF 0x3f3f3f3f
struct Prim{
    int n;//n个顶点
    int s;//起点 树的根
    int weight;//树的重量
    int mp[Vmax][Vmax];
    int dis[Vmax]; //dis[i]表示指向i点的最短的边
    bool vis[Vmax];
    int pre[Vmax];//记录前驱

    void init(int vsz,int source=1){//默认起点为1
        n=vsz;
        weight=0;
        for(int i=0;i<=n;i++){
            dis[i]=INF;
            vis[i]=false;
        }
    }
};
```

```
}

dis[source]=0;

}

//1~n的邻接矩阵
void adde(int u,int v,int w) {
    mp[u][v]=w;
    mp[v][u]=w;
}

int prim() {
    int cnt=0;
    for(int i=1;i<=n;i++) {
        int pos=0;
        for(int j=1;j<=n;j++) {
            if(!vis[j]&&dis[j]<dis[pos])
                pos=j;
        }
        vis[pos]=true;cnt++;
        weight+=dis[pos];
        for(int j=1;j<=n;j++) {
            if(!vis[j]&&mp[pos][j]<dis[j]) {
                dis[j]=mp[pos][j];
                pre[j]=pos;
            }
        }
    }
    if(cnt==n) return weight;
    else return -1;
}
};
```

Prim_Heap

说明

找对于到每一个点最近边的长度，存在对应下标的dis数组中

稀疏图较快

使用方法

1. 见 init() 函数
2. mytype ans = prim_heap(s) 得到以s为起点开始找的最小生成树重量
3. judge(n) 判断该树能否生成

Tips

最小生成树无脑情况就用这个

考虑数据范围，树的重量可能会爆int

模版

```

typedef int mytype;
const int NV=105;
const int NE=10005*2;
mytype dis[NV];
int pre[NV],vis[NV],he[NV],ecnt,pcnt;
struct edge
{
    int v,next;
    mytype l;
} E[NE];
void adde(int u,int v,mytype l)
{
    E[++ecnt].v=v;
    E[ecnt].l=l;
    E[ecnt].next=he[u];
    he[u]=ecnt;
}
void init(int n,int m,int s)

```

```

{
    ecnt=0;
    memset(pre,0,sizeof(pre));
    memset(vis,0,sizeof(vis));
    memset(he,-1,sizeof(he));
    for (int i=0; i<=n; i++)
        dis[i]=inf;
    dis[s]=0;
    for (int i=1; i<=m; i++)
    {
        int u,v;
        mytype l;
        scanf("%d%d%d", &u, &v, &l);
        adde(u,v,l);
        adde(v,u,l);
    }
}

struct point
{
    int u;
    mytype l;
    point(int u,mytype l):u(u),l(l) {}
    bool operator<(const point &p) const
    {
        return l>p.l;
    }
};

mytype prim_heap(int s)
{
    priority_queue<point> q;
    q.push(point(s,0));
    mytype ans=0;
    pcnt=0;
    while(!q.empty())
    {
        point p=q.top();
        q.pop();
        int u=p.u;
        if (vis[u])
            continue;
        vis[u]=1;
        ans+=p.l;//==dis[x]
        pcnt++;
    }
}

```

Prim_Heap

```
for (int i=he[u]; i!=-1; i=E[i].next)
    if (!vis[E[i].v]&&E[i].l<dis[E[i].v])
    {
        dis[E[i].v]=E[i].l;
        pre[E[i].v]=u;
        q.push(point(E[i].v,dis[E[i].v]));
    }
}
return ans;
}

bool judge(int n)
{
    return pcnt==n;
}
```

Kruskal

说明

结构体排序 + 并查集——结构体排序找到所有符合条件的边，加入并查集

稀疏图较快

使用方法

1. Kruskal d;
2. d.init(Vsz, Esz); 初始化顶点个数，边的个数
3. d.adde(u, v, w); 从u指向v权值为w的边，单向加边即可
4. int weight = d.kruskal(); 输出最小生成树的重量，无法建成则返回-1

Tips

待写

模版

```
#include <algorithm>
using namespace std;
#define Vmax 300      //最大点数量
#define Emax 1000     //最大边数量
struct Kruskal{
    int n,m;//n个点 m条边
    int rank[Vmax],fa[Vmax];//并查集需要
    int weight;// weight为最小生成树的重量
    int ecnt;

    struct edge{
        int u,v,w;//起点 终点 权值
        bool operator<(const edge e) const{
            return w<e.w;
        }
    }e[Emax];

    void init(int Vs, int Es) {

```

```

n=Vsz;m=Esz;
ecnt=0;
weight=0;
for(int i=0;i<=Vsz;i++) {
    fa[i]=i;
    rank[i]=0;
}
}

//边的编号从0~m-1
void adde(int u,int v,int w) {
    e[ecnt].u=u;
    e[ecnt].v=v;
    e[ecnt++].w=w;
}

int findx(int x) {
    while(x!=fa[x])
        x=fa[x];
    return x;
}

void set_merge(int u,int v) {
    if(rank[u]<rank[v]){
        fa[u]=v;
        rank[v]+=rank[u];
    }
    else{
        fa[v]=u;
        rank[u]+=rank[v];
    }
}

int kruskal() {
    int cnt=0;
    sort(e,e+m);
    for(int i=0;i<m;i++) {
        int uR=findx(e[i].u);
        int vR=findx(e[i].v);
        if(uR!=vR){
            set_merge(uR, vR);
            weight+=e[i].w;
            cnt++;
        }
    }
}

```

Kruskal

```
        }
    }
    if(cnt==n-1) return weight; //最小生成树可建
    else return -1; //最小生成树不可建
}
};
```

次小生成树

说明

求最小生成树时，用数组`maxd[i][j]`来表示MST中i到j最大边权，求完后，直接枚举所有不在MST中的边，替换掉最大边权的边，更新答案

使用方法

1. `SMST::init(n);`
2. `SMST::adde(u, v, w);` 初始化顶点个数，起始点位置
3. `int status = SMST::smst();` 返回-1表示无最小生成树，返回-2表示有最小生成树无次小生成树，否则返回次小生成树的权

Tips

- `SMST::weight` 直接记录了最小生成树的权重
- 对于平行边/自环要进行特殊处理

模版

邻接矩阵

缺点：无法处理平行边

```
// 顶点下标1~n
const int Vmax = 1005;
namespace SMST {
    // MST
    int n; // n个顶点
    int s; // 起点 树的根
    int weight; // MST的重量
    int mp[Vmax][Vmax];
    int dis[Vmax]; // dis[i] 表示指向 i 点的最短的边
    bool vis[Vmax]; // 标记该点是否在树上
    int pre[Vmax]; // 记录前驱

    // SMST
    int subweight; // SMST的重量
}
```

```

bool used[Vmax][Vmax]; //表示当前边有没有被用过
int maxd[Vmax][Vmax]; //maxd[u][v]表示 u-v路径上最大的边权

void init(int Vsz,int source=1){//默认起点为1
    memset(mp, INF, sizeof(mp));
    memset(dis, INF, sizeof(dis));
    memset(used, 0, sizeof(used));
    memset(maxd, 0, sizeof(maxd));
    memset(vis, false, sizeof(vis));
    n=Vsz;
    for (int i=0; i<=n; i++) {
        mp[i][i] = 0;
    }
    weight=0;
    dis[source] = 0;
    pre[source] = source;
}

//1~n的邻接矩阵
void adde(int u,int v,int w) {
    mp[u][v]=w;
    mp[v][u]=w;
}

int prim() {
    /*
    返回值说明:
    -1: 无生成树
    weight: 最小生成树的重量
    */
}

for(int i=1;i<=n;i++) {
    int pos=0;
    int minc = INF;
    for(int j=1;j<=n;j++) {
        if(!vis[j]&&dis[j]<minc) {
            pos=j;
            minc = dis[j];
        }
    }
    if(minc == INF) return -1; //n个点不联通 无生成树
    weight+=dis[pos];
}

```

```

        used[pos][pre[pos]] = true; // for SMST
        used[pre[pos]][pos] = true; // for SMST
        vis[pos]=true;

        for(int j=1;j<=n;j++) {
            if (j == pos) continue;
            if(vis[j]) {
                maxd[j][pos] = maxd[pos][j] = max(dis[pos],
maxd[j][pre[pos]]);
            }
            else if(!vis[j]&&mp[pos][j]<dis[j]){
                dis[j]=mp[pos][j];
                pre[j]=pos;
            }
        }

    }
    return weight;
}

int smst () {
/*
    返回值说明:
    -1: 无生成树
    -2: 有最小生成树 无次小生成树 (比如给出的图即为一棵树)
    subweight: 次小生成树的重量
*/
    if(prim() == -1) return -1; //无生成树
    subweight = INF;
    for(int i = 1; i<= n; i++) {
        for(int j = i + 1; j <= n; j++){
            if(mp[i][j]!=INF && !used[i][j]){
                subweight = min(subweight, weight+mp[i][j]-
maxd[i][j]);
            }
        }
    }
    if(subweight == INF) return -2; //只有唯一生成树 也就是说只有最
小生成树 没有次小生成树
    return subweight;
}
}

```

邻接表

```

// 顶点下标1~n
const int Vmax = 105;
namespace SMST{
    struct Edge{
        int u,v,next,w,vis;
    }e[405];
    int ecnt;
    int he[Vmax];

    // MST
    int n;//n个顶点
    int s;//起点 树的根
    int weight; //MST的重量
    int dis[Vmax]; //dis[i]表示指向i点的最短的边
    bool vis[Vmax]; //标记该点是否在树上
    int pre[Vmax];//记录前驱

    //SMST
    int subweight; //SMST的重量
    int id[Vmax];
    int maxd[Vmax][Vmax]; //maxd[u][v]表示 u-v路径上最大的边权

    void init(int vsz,int source=1){//默认起点为1
        memset(he, -1, sizeof(he));
        ecnt = 0;
        memset(dis, INF, sizeof(dis));
        memset(maxd, 0, sizeof(maxd));
        memset(vis, false, sizeof(vis));
        n=vsz;
        weight=0;
        dis[source] = 0;
        pre[source] = source;
        s = source;
    }

    //1~n的邻接矩阵
    void adde(int u,int v,int w){
        e[ecnt].vis = 0;
        e[ecnt].u = u;
        e[ecnt].v = v;
        e[ecnt].w = w;
    }
}

```

```

e[ecnt].next = he[u];
he[u] = ecnt++;
}

int prim() {
/*
    返回值说明:
    -1: 无生成树
    weight: 最小生成树的重量
*/
for(int i=1;i<=n;i++) {
    int pos=0;
    int minc = INF;
    for(int j=1;j<=n;j++) {
        if(!vis[j]&&dis[j]<minc) {
            pos=j;
            minc = dis[j];
        }
    }
    if(minc == INF) return -1; //n个点不联通 无生成树

    weight+=dis[pos];
    vis[pos]=true;
    if (i!=s) {
        e[id[pos]].vis = 1;
        e[id[pos]^1].vis = 1;
    }

    for(int j=1;j<=n;j++) {
        if (j == pos) continue;
        if(vis[j]) {
            maxd[j][pos] = maxd[pos][j] = max(dis[pos],
maxd[j][pre[pos]]);
        }
    }
    for (int j = he[pos]; j != -1; j = e[j].next) {
        int v = e[j].v;
        if (!vis[v] && e[j].w < dis[v]) {
            dis[v] = e[j].w;
            pre[v] = pos;
            id[v] = j;
        }
    }
}
}

```

```

    }

    return weight;
}

int smst() {
    /*
    返回值说明:
    -1: 无生成树
    -2: 有最小生成树 无次小生成树 (比如给出的图即为一棵树)
    subweight: 次小生成树的重量
    */
    if(prim() == -1) return -1; //无生成树
    subweight = INF;
    for (int i=0; i<ecnt; i+=2) {
        int u = e[i].u;
        int v = e[i].v;
        if (e[i].vis == 0) {
            subweight = min(subweight, weight + e[i].w - maxd[u]
[v]);
        }
    }
    if(subweight == INF) return -2; //只有唯一生成树 也就是说只有最
小生成树 没有次小生成树
    return subweight;
}
}

```

最小树形图

说明

最小树形图就是处理一个确定起点有向有权图的最小生成树问题

解决的方法为[朱刘算法](#)

使用方法

1. 修改mytype为边权的类型
2. `init(m)` 表示初始化 m 条边
3. `ans = Directed_MST(1, n, m);` 起点为 1 , n 个点, m 条边
4. `bool ok = judge(ans);` ok为true表示存在该树, false表示不存在

Tips

待写

模版

```
typedef int mytype;
const int Vmax=105;
const int Emax=Vmax*Vmax;
struct edge//有向边
{
    int u,v;      //起点 终点
    mytype l;     //权值
} e[Emax];
int pre[Vmax],ID[Vmax],vis[Vmax];
mytype In[Vmax];
void init(int m)
{
    for(int i=1; i<=m; i++){
        scanf("%d%d%d", &e[i].u, &e[i].v, &e[i].l);
    }
}
mytype Directed_MST (int root,int NV,int NE)
{
```

最小树形图

```
//      memset(pre, 0, sizeof(pre));
mytype ret = 0;
while(1)
{
    //1.找最小入边
    for(int i=1; i<=NV; i++)
        In[i] = INF;
    for(int i=1; i<=NE; i++)
    {
        int u = e[i].u;
        int v = e[i].v;
        if(e[i].l < In[v] && u != v)
        {
            pre[v] = u;
            In[v] = e[i].l;
        }
    }
    for(int i=1; i<=NV; i++)
    {
        if(i == root)
            continue;
        if(fabs(In[i]-INF)<eps)
            return -1; //除了跟以外有点没有入边，则根无法到达它
    }
    //2.找环
    int cntnode = 0;
    memset(ID,-1,sizeof(ID));
    memset(vis,-1,sizeof(vis));
    In[root] = 0;
    for(int i=1; i<=NV; i++) //标记每个环
    {
        ret += In[i];
        int v = i;
        while(vis[v] != i && ID[v] == -1 && v != root)
        {
            vis[v] = i;
            v = pre[v];
        }
        if(v != root && ID[v] == -1)
        {
            ID[v] = ++cntnode;
            for(int u = pre[v] ; u != v ; u = pre[u])
                ID[u] = cntnode;
        }
    }
}
```

最小树形图

```
        }
    }
    if(cntnode == 0)
        break; //无环
    for(int i=1; i<=NV; i++)
        if(ID[i] == -1)
            ID[i] = ++cntnode;
    //3.缩点,重新标记
    for(int i=1; i<=NE; i++)
    {
        int v = e[i].v;
        e[i].u = ID[e[i].u];
        e[i].v = ID[e[i].v];
        if(e[i].u != e[i].v)
        {
            e[i].l -= In[v];
        }
    }
    NV = cntnode;
    root = ID[root];
}
return ret;
}
bool judge(mytype ans) //判断能否成树
{
    return fabs(ans+1)>eps;
}
```

LCA

最近公共祖先 (Least Common Ancestors)

定义：最近公共祖先是指在一个树或者有向无环图中同时拥有 v 和 w 作为后代的最深的节点。在这里，我们定义一个节点也是其自己的后代，因此如果 v 是 w 的后代，那么 w 就是 v 和 w 的最近公共祖先。

LCA->RMQ

说明

在线算法，使用dfs处理出深度之后进行RMQ找到区间内深度最小的点，该点即为LCA

使用方法

1. 见主函数

Tips

- 不推荐使用该方法查询LCA，**推荐使用倍增法**

模版

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <ctype.h>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <vector>
#define eps 1e-8
#define INF 0x7fffffff
#define maxn 100005
#define PI acos(-1.0)
#define seed 31//131,1313
typedef long long LL;
typedef unsigned long long ULL;
using namespace std;
//DFS + RMQ 在线算法
```

```

struct node
{
    int x;
};

vector<node> V[maxn];
int E[maxn * 2], D[maxn * 2], first[maxn], vis[maxn], dis[maxn], n,
m, top = 1, root[maxn], st;
int dp[30][maxn * 2];
void init()
{
    top=1;
    memset(root, 0, sizeof(root));
    memset(vis, 0, sizeof(vis));
    scanf("%d", &n);
    for(int i=1; i<=n; i++)
        V[i].clear();
    int a, b;
    node tmp;
    for (int i = 0; i < n-1; i++)
    {
        scanf(" %d%d", &a, &b);
        tmp.x = b;
        V[a].push_back(tmp);
        tmp.x = a;
        V[b].push_back(tmp);
        root[b]=1;
    }
    for(int i=1; i<=n; i++)
        if(root[i]==0)
    {
        st=i;
        break;
    }
}
void dfs(int u, int dep)
{
    vis[u] = 1, E[top] = u, D[top] = dep, first[u] = top++;
    for (int i = 0; i < V[u].size(); i++) if (!vis[V[u][i].x])
    {
        int v = V[u][i].x;
        dfs(v, dep + 1);
        E[top] = u, D[top++] = dep;
    }
}

```

```

}

void ST(int num)
{
    for (int i = 1; i <= num; i++) dp[0][i] = i;
    for (int i = 1; i <= log2(num); i++)
        for (int j = 1; j <= num; j++) if (j + (1 << i) - 1 <= num)
    {
        int a = dp[i - 1][j], b = dp[i - 1][j + (1 << i >> 1)];
        if (D[a] < D[b]) dp[i][j] = a;
        else dp[i][j] = b;
    }
}

int RMQ(int x, int y)
{
    int k = (int) log2(y - x + 1.0);
    int a = dp[k][x], b = dp[k][y - (1 << k) + 1];
    if (D[a] < D[b]) return a;
    return b;
}

int main ()
{
    int T;
    scanf("%d", &T);
    while(T--)
    {
        init();
        dfs(st, 0);
        ST(top);
        int x, y;
        scanf("%d%d", &x, &y);
        int a = first[x], b = first[y];
        if (a > b) swap(a, b);
        int pos = RMQ(a, b);
        printf("%d\n", E[pos]);
    }
    return 0;
}

```

Tarjan

说明

离线的方式进行查询，先将所有询问录入，进行DFS搜索时处理询问，使用并查集的思想。

复杂度 $O(n+q)$

使用方法

1. 见主函数

Tips

- 不推荐使用该方法查询LCA，**推荐使用倍增法**

模版

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <ctype.h>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <vector>
#define eps 1e-8
#define INF 0x7fffffff
#define PI acos(-1.0)
#define seed 31//131,1313
typedef long long LL;
typedef unsigned long long ULL;
using namespace std;
const int maxn = 300000+10;
```

```

int pre[maxn], point[maxn], point2[maxn];
bool vis[maxn];
struct Edge
{
    int v; //连接点
    int next; //下一条从此边的出发点发出的边
} edge[maxn*2];
struct Query
{
    int v;
    int w;
    int next;
} query[maxn*2];
int top, top2;
void init()
{
    memset(vis, 0, sizeof(vis));
    memset(point, -1, sizeof(point));
    memset(point2, -1, sizeof(point2));
    top=0;
    top2=0;
}
void add_edge(int u, int v)
{
    edge[top].v=v;
    edge[top].next=point[u]; //上一条边的编号
    point[u]=top++; //u点的第一条边编号变成head
}
void findset(int x) //并查集
{
    if(x!=pre[x])
    {
        pre[x]=findset(pre[x]); //路径压缩
    }
    return pre[x];
}
void add_query(int u, int v)
{
    query[top2].v=v;
    query[top2].w=-1;
    query[top2].next=point2[u]; //上一条边的编号
    point2[u]=top2++; //u点的第一条边编号变成head
    query[top2].v=u;
}

```

```

query[top2].w=-1;
query[top2].next=point2[v];///上一条边的编号
point2[v]=top2++;///u点的第一条边编号变成head
}
int lca(int u,int f) ///当前节点，父节点
{
    pre[u]=u; //设立当前节点的集合
    for(int i=point[u]; i!=-1; i=edge[i].next)
    {
        if(edge[i].v==f)
            continue;
        lca(edge[i].v,u); //搜索子树
        pre[edge[i].v]=u; //合并子树
    }
    vis[u]=1; //以u点为集合的点搜索完毕
    for(int i=point2[u]; i!=-1; i=query[i].next)
    {
        if(vis[query[i].v]==1)
            query[i].w=findset(query[i].v);
    }
    return 0;
}
int main()
{
    int root[maxn];
    int T;
    scanf("%d", &T);
    for(int ii=0; ii<T; ii++)
    {
        init();
        int tot,r=-1,a,b;
        scanf("%d", &tot);
        for(int i=1; i<=tot; i++)
            root[i]=i;
        for(int i=0; i<tot-1; i++)
        {
            scanf("%d%d", &a, &b);
            add_edge(a,b);
            add_edge(b,a);
            root[b]=a;
        }
        for(int i=1; i<=tot; i++)
            if(root[i]==i)

```

```
r=i; //树的根  
scanf("%d%d", &a, &b);  
add_query(a, b);  
lca(r, r);  
for(int i=0; i<top2; i++)  
{  
    if(query[i].w!= -1)  
        printf("%d\n", query[i].w);  
}  
  
}  
return 0;  
}
```

倍增算法

说明

效率较高的一种求LCA的方法，在线查询，通过二进制的方式降低复杂度，推荐使用

原理讲解：[LCA-倍增法（在线）O\(nlogn\)-O\(logn\)](#)

使用方法

1. `init()`
2. `addege()` 需要双向加边
3. DFS/BFS
4. `LCA(x, y)` 返回 x和y的LCA

Tips

1. 推荐使用该方法查询LCA
2. 树的根节点要根据具体情况来定
3. 如使用DFS处理，可以调用 `get_kth_anc(int x, int k)` 来查找x的第k个祖先，但是可能会爆栈

模版

```
#pragma comment(linker, "/STACK:10240000000,10240000000") // 扩栈，要用c++交，用g++交并没有什么卵用。。。。
int n, m, he[maxn], rt[maxn], ecnt;
bool vis[maxn];
struct edge{
    int v, next;
} e[maxn << 1];

int dep[maxn], f[20][maxn], Lev, s[maxn]; // 1<<20 < N, f[j][i] 表示i的第2^j个祖先 最多可容纳100w的节点数量

void dfs(int x, int fa){ // 也可以用bfs，但bfs不能统计节点孩子个数
    dep[x] = dep[fa] + 1;
    f[0][x] = fa, s[x] = 1;
    for (int i = he[x]; ~i; i = e[i].next) {
```

```

int y = e[i].x;
if (y != fa) {
    dfs(y, x);
    s[x] += s[y]; //节点x的孩子个数
}
}

void bfs(int rt) ///不需要求孩子个数，同时防止暴栈
{
    queue<int> q;
    q.push(rt);
    f[0][rt] = 0, dep[rt] = 1, vis[rt] = 1;
    while (!q.empty()) {
        int fa = q.front();
        q.pop();
        for (int i = he[fa] ; ~i ; i = e[i].next) {
            int v = e[i].v;
            if (!vis[v]) {
                dep[v] = dep[fa] + 1;
                f[0][v] = fa, vis[v] = 1;
                q.push(v);
            }
        }
    }
}

int LCA(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    for (int i = Lev ; i >= 0 ; -- i) ///找y的第dep[y] - dep[x]个祖先
        if (dep[y] - dep[x] >> i & 1)
            y = f[i][y];
    if (x == y) return y;
    for (int i = Lev ; i >= 0 ; -- i)
        if (f[i][x] != f[i][y])
            x = f[i][x], y = f[i][y];
    return f[0][x];
}

int get_kth_anc(int x, int k) { ///找x的第k个祖先
    for (int i = 0 ; i <= Lev ; ++ i)
        if (k >> i & 1)
            x = f[i][x];
    return x;
}

void init() {

```

```

    memset(he , -1 , sizeof(he));
    memset(rt, 0, sizeof(rt));
    ecnt = 0;
}

void adde(int u,int v){
    e[ecnt].v = v;
    e[ecnt].next = he[u];
    he[u] = ecnt++;
}

void work(){
    int i , j , x , y , z,anc;
    init();
    scanf("%d",&n);
    for (i = 1 ; i < n ; ++ i){
        scanf("%d%d",&x,&y);
        adde(x,y);
        adde(y,x);
        rt[y] = x;
    }
    for(int i=1; i<=n; i++){
        if(rt[i]==0){
            anc = i;      //确定树的根节点
            break;
        }
    }
    dfs(anc , 0);
    for (j = 1 ; 1 << j < n ; ++ j)
        for (i = 1 ; i <= n ; ++ i)
            f[j][i] = f[j - 1][f[j - 1][i]];
    Lev = j - 1;

    int q;
    scanf("%d",&q);
    while (q--) {
        scanf("%d%d",&x,&y);
        if (dep[x] < dep[y])
            swap(x , y);
        printf("%d\n",LCA(x , y));
    }
}
int main(){
    int T;
    scanf("%d",&T);
}

```

倍增算法

```
while (T--) {  
    work();  
}  
return 0;  
}
```

二分图

二分图主要是说，一个点集A和一个点集B， A与B无公共元素，且各点集内部点无连线

概念

最大独立集：求一个二分图中最大的一个点集，该点集内的点互不相连。

最小顶点覆盖数：在二分图中，用最少的点，让所有的边至少和一个点有关联。换句话说，假如选了一个点就相当于覆盖了以它为端点的所有边，你需要选择最少的点来覆盖所有的边。

最小路径覆盖：找出最小的路径条数，使这些路径覆盖图中所有点。

计算方法

最大独立集 = 顶点数 - 最大匹配数 = $vN + uN - \text{hungary}()$

最小顶点覆盖数 = 最大匹配数 = $\text{hungary}()$

最小路径覆盖 = $|G| - \text{最大匹配数}$

匈牙利算法

说明

匈牙利算法的DFS实现

g[][]两边顶点的划分情况

优点：适用于稠密图，DFS找增广路，实现简洁易于理解

时间复杂度:O(VE)

使用方法

1. 给vN和uN赋值， uN是匹配左边的顶点数， vN是匹配右边的顶点数
2. `memset(g, 0, sizeof(g));`
3. `g[u][v]=1` 表示加一条从u指向v的边，注意单向加边
4. `int ans = hungary();` 得到最大匹配数

Tips

注意下标！

```

const int maxn=1000;
int uN,vN; //u,v数目
int g[maxn][maxn];//编号是1~n的
int linker[maxn];
bool used[maxn];
bool dfs(int u)
{
    int v;
    for(v=1;v<=vN;v++)
        if(g[u][v] && !used[v])
    {
        used[v]=true;
        if(linker[v]==-1 || dfs(linker[v]))
        {
            linker[v]=u;
            return true;
        }
    }
    return false;
}
int hungary()
{
    int res=0;
    int u;
    memset(linker,-1,sizeof(linker));
    for(u=1;u<=uN;u++)
    {
        memset(used,0,sizeof(used));
        if(dfs(u)) res++;
    }
    return res;
}

```

Hopcroft-Karp

说明

二分图最大匹配 Hopcroft-Karp算法

二分图集合X和Y。求X的最大匹配

复杂度: $O(E * \sqrt{V})$

不停的BFS求出距离标号，然后DFS找增广路

适用于数据较大的二分匹配

使用方法

1. nx, ny需要初始化
2. `g[i].clear()` 对于 $0 \leq i \leq nx$
3. `g[x].pb(y);` 表示加一条从x指向y的边，注意单向加边
4. `int ans = match();` 得到最大匹配数

Tips

下标从1~nx以及1~ny

模版

```
const int maxn = 500;
int nx, ny; //nx表示x部的点, ny表示y部的点
vector<int> g[maxn];
int mx[maxn], my[maxn]; //mx表示x部中每个点对应的y值, my表示y部中每个点对应的x值
queue<int> q;
int dx[maxn], dy[maxn];
bool vis[maxn];
bool find (int u) {
    for (int i = 0; i < g[u].size(); i++) if (!vis[g[u][i]] &&
        dy[g[u][i]] == dx[u] + 1) {
        int v = g[u][i];
        vis[v] = true;
        if (!my[v] || find(my[v])) {
```

```

        mx[u] = v;
        my[v] = u;
        return true;
    }
}
return false;
}

int match() {
    memset(mx, 0, sizeof(mx));
    memset(my, 0, sizeof(my));
    int ans = 0;
    while(true) {
        bool flag = false;
        while(!q.empty()) q.pop();
        memset(dx, 0, sizeof(dx));
        memset(dy, 0, sizeof(dy));
        for (int i = 1; i <= nx; i++) if (!mx[i]) q.push(i);
        while(!q.empty()) {
            int u = q.front();
            q.pop();
            for (int i = 0; i < g[u].size(); i++) if (!dy[g[u][i]])
{
                int v = g[u][i];
                dy[v] = dx[u] + 1;
                if (my[v]) {
                    dx[my[v]] = dy[v] + 1;
                    q.push(my[v]);
                }
                else flag = true;
            }
        }
        if (!flag) break;
        memset(vis, 0, sizeof(vis));
        for (int i = 1; i <= nx; i++) if (!mx[i] && find(i)) ans++;
    }
    return ans;
}

```

网络流

最小割 = 最大流

建模文章：[网络流建模汇总][Edelweiss].pdf

EdmondsKarp

说明：

- n为定点数量
- m为边数乘2
- e[]和g[][]存储邻接表
- a[]用于记录通过当前边的最大流量
- p[]记录路径

使用方法：

1. EdmondsKarp X;
2. X.init(n); \$\$n\$\$为顶点数
3. X.addEdge(from, to, cap); 加边
4. int ans = X.Maxflow(start, sink); 求出最大流

Tips:

复杂度较高，慎用

```
#define maxn 10005 //表示节点数量
struct edge{
    int from,to,cap,flow;
    edge(int u,int v,int c,int f):from(u),to(v),cap(c),flow(f){}
};

struct EdmondsKarp{
    int n,m;
    vector<edge>e;
    vector<int>g[maxn];
    int a[maxn];
    int p[maxn];
    void init(int n){
        for(int i=0;i<n;i++)
            g[i].clear();
        e.clear();
    }

    void AddEdge(int from,int to,int cap){
        e.push_back(edge(from,to,cap,0));
        e.push_back(edge(to,from,0,0));
        m=e.size();
        g[from].push_back(m-2);
    }
}
```

```

        g[to].push_back(m-1);
    }

int Maxflow(int s,int t){ //source target
    int flow=0;
    while(1) {
        memset(a, 0, sizeof(a));
        queue<int>q;
        q.push(s);
        a[s]=INF;
        while(!q.empty()) {
            int tmp=q.front();q.pop();
            for (int i=0; i<g[tmp].size(); i++) {
                edge& etmp=e[g[tmp][i]];
                if(!a[etmp.to]&&etmp.cap>etmp.flow) {
                    p[etmp.to]=g[tmp][i];
                    a[etmp.to]=min(a[tmp], etmp.cap-etmp.flow);
                    q.push(etmp.to);
                }
            }
            if(a[t]) break;
        }
        if(!a[t])break;
        for(int u=t;u!=s;u=e[p[u]].from) {
            e[p[u]].flow+=a[t];
            e[p[u]^1].flow-=a[t];
        }
        flow+=a[t];
    }
    return flow;
}
};
```

Dinic

说明

使用方法

1. `Dinic d;`
2. `d.init(start,end);` 初始化
3. `d.adde(x,y,c);` 加边
4. `cout<<d.dinic<<endl` 输出最大流的值

Tips

- 较EdmondsKarp快一点，但是最好用ISAP
- `adde`是加一条有向边，体现在参量网络上是加了正向反向两条边，如果原图中是无向的，应该执行两次加边过程

示例

- 网络流最大权闭合图[POJ 2987 Firing](#)

模版

```
#define vmax 20005 //最大顶点数
#define emax 500005 //最大边数
struct Dinic{
    int src,target,ecnt;//源点, 汇点, 边数
    int head[vmax];//邻接表表头
    int cur[vmax];
    int dis[vmax];//从源点到该点的距离
    struct edge{
        int to,next,cap;
    }e[2*emax];//边可能是双向的, 故乘2

    void init(int start,int end) {
        ecnt=0;
        src=start;
        target=end;
    }
}
```

```

        memset(head, -1, sizeof(head));
    }

void adde(int from, int to, int cap) {
    e[ecnt].to=to;
    e[ecnt].cap=cap;
    e[ecnt].next=head[from];
    head[from]=ecnt++;

    e[ecnt].to=from;
    e[ecnt].cap=0;
    e[ecnt].next=head[to];
    head[to]=ecnt++;
}

bool BFS() {
    memset(dis, -1, sizeof(dis));
    dis[src]=0;
    queue<int>q;
    q.push(src);
    while(!q.empty()) {
        int tmp=q.front();
        q.pop();
        for(int i=head[tmp]; i!=-1; i=e[i].next) {
            int tp=e[i].to;
            if(dis[tp]==-1&&e[i].cap) {
                dis[tp]=dis[tmp]+1;
                q.push(tp);
                if(tp==target)
                    return true;
            }
        }
    }
    return false;
}

int DFS(int u, int delta) {
    if(u==target || delta==0)
        return delta;
    int f, fflow=0;
    for(int& i=cur[u]; i!=-1; i=e[i].next) {
        if(dis[u]+1==dis[e[i].to] &&
           (f=DFS(e[i].to, min(delta, e[i].cap)))) {

```

```
{  
    e[i].cap-=f;  
    e[i^1].cap+=f;  
    flow+=f;  
    delta-=f;  
    if(!delta)break;  
}  
}  
return flow;  
}  
  
int dinic(){  
    int tmp=0,maxflow=0;  
    while(BFS()){  
        for(int i=src;i<=target;i++) cur[i]=head[i];  
        while(tmp=DFS(src, INF))  
            maxflow+=tmp;  
    }  
    return maxflow;  
}  
};
```

ISAP

说明

ISAP + Gap优化

- `node.cap` 残量网络中该边剩余可以走的流量
- `node.flow` 最大流经过这条边的流量

使用方法

1. `ISAP::init(st, ed, n);` 初始化起点、终点、顶点数
2. `ISAP::adde(u, v, cap);` 是一个单向加边的过程，如果拆点的话要注意出点和入点如 `ISAP::adde(u+n, v, cap);`
3. `int ans = ISAP::maxflow();` 得到结果

Tips

1. 该模版只能跑一次网络流
2. 注意Vmax和Emax，拆点的话要把顶点数加倍，尤其是Emax可能为
3. 拆点的话要把顶点数加倍

示例

1. [HDU 4280 Island Transport](#)
2. 最小割最大流: [HDU 4289 Control](#)
3. 最大权闭合图: [HDU 5855 Less Time, More profit](#)
4. 最大权闭合图[POJ 2987 Firing](#)

模版

```
const int INF=0x3f3f3f3f;
namespace ISAP{
    const int Vmax = 420900;
    const int Emax = 420900;
    int n;
    struct Node {
        int v;      // vertex
        int cap;    // capacity
    };
}
```

```

        int flow;    // current flow in this arc
        int nxt;
    } e[Emax * 2];
int g[Vmax], fcnt;
int st, ed;
int dist[Vmax], numbs[Vmax], q[Vmax];
void adde(int u, int v, int c) {      //单向加边的过程
    e[++fcnt].v = v;
    e[fcnt].cap = c;
    e[fcnt].flow = 0;
    e[fcnt].nxt = g[u];
    g[u] = fcnt;
    e[++fcnt].v = u;
    e[fcnt].cap = 0;
    e[fcnt].flow = 0;
    e[fcnt].nxt = g[v];
    g[v] = fcnt;
}
void init(int src,int sink,int n_) {
    memset(g, 0, sizeof(g));
    fcnt = 1;
    n=n_;
    st = src, ed = sink; /*修改*/
}
void rev_bfs() {
    int font = 0, rear = 1;
    for (int i = 0; i <= n; i++) { //n为总点数
        dist[i] = Vmax;
        numbs[i] = 0;
    }
    q[font] = ed;
    dist[ed] = 0;
    numbs[0] = 1;
    while(font != rear) {
        int u = q[font++];
        for (int i = g[u]; i; i = e[i].nxt) {
            if (e[i ^ 1].cap == 0 || dist[e[i].v] < Vmax)
continue;
            dist[e[i].v] = dist[u] + 1;
            ++numbs[dist[e[i].v]];
            q[rear++] = e[i].v;
        }
    }
}

```

```

}

int maxflow() {
    rev_bfs();
    int u, totalflow = 0;
    int curg[Vmax], revpath[Vmax];
    for(int i = 0; i <= n; ++i) curg[i] = g[i];
    u = st;
    while(dist[st] < n) {
        if(u == ed) { // find an augmenting path
            int augflow = INF;
            for(int i = st; i != ed; i = e[curg[i]].v)
                augflow = min(augflow, e[curg[i]].cap);
            for(int i = st; i != ed; i = e[curg[i]].v) {
                e[curg[i]].cap -= augflow;
                e[curg[i] ^ 1].cap += augflow;
                e[curg[i]].flow += augflow;
                e[curg[i] ^ 1].flow -= augflow;
            }
            totalflow += augflow;
            u = st;
        }
        int i;
        for(i = curg[u]; i; i = e[i].nxt)
            if(e[i].cap > 0 && dist[u] == dist[e[i].v] + 1)
break;
        if(i) { // find an admissible arc, then Advance
            curg[u] = i;
            revpath[e[i].v] = i ^ 1;
            u = e[i].v;
        } else { // no admissible arc, then relabel this
vertex
            if(0 == (--numbs[dist[u]])) break; // GAP cut,
Important!
            curg[u] = g[u];
            int mindist = n;
            for(int j = g[u]; j; j = e[j].nxt)
                if(e[j].cap > 0) mindist = min(mindist,
dist[e[j].v]);
            dist[u] = mindist + 1;
            ++numbs[dist[u]];
            if(u != st)
                u = e[revpath[u]].v; // Backtrack
        }
    }
}

```

```
    }
    return totalflow;
}
}
```

最小费用最大流

说明

出处：《算法竞赛入门经典(第二版)》

使用方法

1. MCMF::init(n) n个点
2. MCMF::adde(u,v,cap,cost); 加边
3. long long minCost, maxFlow;
4. maxFlow = MCMF::MincostMaxflow(st, ed, minCost); 执行完的结果 maxFlow 为最大流 minCost 为最小费用

Tips

- 需要保证初始网络中没有负权圈，不过可以有负权边
- 求最大费用最大流：将所有边的费用都加个负号，最后结果再加回来一个负号即可
- n个点的下标可以为 0~n-1 也可以为 1~n
- 拆点的话要把顶点数加倍

示例

1. [POJ 2195 Going Home](#)
2. [UVa 1658 Admiral](#)
3. [HDU 3488 Tour](#)

模版

```
const int Vmax=2005; //需要拆点的话记得加倍
namespace MCMF{
    struct Edge{
        int from,to,cap,flow,cost;
        Edge(int u,int v,int c,int f,int w):from(u),to(v),cap(c),flow(f),cost(w){}
    };
    int n,m;
```

```

vector<Edge>edges;
vector<int>G[Vmax];
int inq[Vmax]; //是否在队列中
int d[Vmax]; //Bellman-Ford
int p[Vmax]; //上一条弧
int a[Vmax]; //可改进量

void init(int _Vsz) {
    n=_Vsz;
    for(int i=0;i<=n;i++) G[i].clear();
    edges.clear();
}

void adde(int from,int to,int cap,int cost) {
    edges.push_back(Edge(from,to,cap,0,cost));
    edges.push_back(Edge(to,from,0,0,-cost));
    m=edges.size();
    G[from].push_back(m-2);
    G[to].push_back(m-1);
}

bool SPFA(int s,int t,int& flow,long long& cost) {
    for(int i=0;i<=n;i++) d[i]=INF;
    memset(inq, 0, sizeof(inq));
    d[s]=0;
    inq[s]=1;
    p[s]=0;
    a[s]=INF;

    queue<int>q;
    q.push(s);
    while(!q.empty()) {
        int u=q.front();
        q.pop();
        inq[u]=0;
        for(int i=0;i<G[u].size();i++) {
            Edge& e=edges[G[u][i]];
            if(e.cap>e.flow&&d[e.to]>d[u]+e.cost) {
                d[e.to]=d[u]+e.cost; //松弛操作
                p[e.to]=G[u][i]; //记录上一条边信息
                a[e.to]=min(a[u], e.cap-e.flow);
                if(!inq[e.to]) {
                    q.push(e.to);
                }
            }
        }
    }
    return flow=t;
}

```

```
       inq[e.to]=1;
    }
}
}
}
if(d[t]==INF) return false; //s-t 不联通, 失败退出
flow+=a[t];
cost+=(long long)d[t]*(long long)a[t];
for(int u=t;u!=s;u=edges[p[u]].from) {
    edges[p[u]].flow+=a[t];
    edges[p[u]^1].flow-=a[t];
}
return true;
}

int MincostMaxflow(int s,int t,long long& cost) {
    int flow=0;
    cost=0;
    while(SPFA(s, t, flow, cost));
    return flow;
}
}
```

舞蹈链

说明

Dancing Links [Kuangbin带你飞] 模版及题解

这方面了解的非常不完善，有待添加

使用方法

Tips

模版

```
// 精确覆盖
const int MAXN = 1010;
const int MAXM = 1010;
const int MAXNODE = MAXN * MAXN;
struct DLX
{
    int n, m, sz;
    int
    U[MAXNODE], D[MAXNODE], R[MAXNODE], L[MAXNODE], Row[MAXNODE], Col[MAXNODE];
};

int H[MAXN], S[MAXM];
int ansd, ans[MAXN];

void init(int _n, int _m)
{
    n = _n;
    m = _m;
    for(int i = 0 ; i <= m ; i++)
    {
        S[i] = 0;
        U[i] = D[i] = i;
        L[i] = i - 1;
        R[i] = i + 1;
    }
    R[m] = 0; L[0] = m;
```

```

sz = m;
for(int i = 1 ; i <= n ; i++) H[i] = -1;
}

void link(int r,int c)
{
    ++S[Col[+sz] = c];
    Row[sz] = r;
    D[sz] = D[c];
    U[D[c]] = sz;
    U[sz] = c;
    D[c] = sz;
    if(H[r] < 0) H[r] = L[sz] = R[sz] = sz;
    else
    {
        R[sz] = R[H[r]];
        L[R[H[r]]] = sz;
        L[sz] = H[r];
        R[H[r]] = sz;
    }
}

void Remove(int c)
{
    L[R[c]] = L[c]; R[L[c]] = R[c];
    for(int i = D[c] ; i != c ; i = D[i])
        for(int j = R[i] ; j != i ; j = R[j])
        {
            U[D[j]] = U[j];
            D[U[j]] = D[j];
            --S[Col[j]];
        }
}

void resume(int c)
{
    for(int i = U[c] ; i != c ; i = U[i])
        for(int j = L[i] ; j != i ; j = L[j])
            ++S[Col[U[D[j]] = D[U[j]] = j]];
    L[R[c]] = R[L[c]] = c;
}

bool Dance(int d)

```

```

{
    if(R[0] == 0)
    {
        ansd = d;
        printf("%d ", d);
        for (int i = 0 ; i < d ; i++)
            printf("%d%c", ans[i], i == d - 1 ? '\n' : ' ');
        //输出选定的行
        return true;
    }
    int c = R[0];
    for(int i = R[0] ; i != 0 ; i = R[i])
        if(S[i] < S[c])
            c = i;
    Remove(c);
    for(int i = D[c] ; i != c ; i = D[i])
    {
        ans[d] = Row[i];
        for(int j = R[i] ; j != i ; j = R[j]) Remove(Col[j]);
        if(Dance(d + 1)) return true;
        for(int j = L[i] ; j != i ; j = L[j]) resume(Col[j]);
    }
    resume(c);
    return false;
}
};

```

无向图的割顶和割桥

说明

学习资料

- 刘汝佳<<训练指南>>P312-314页
- 无向图求割顶与桥
- 图的割点、桥与双连通分支

应用

删除一个无向图中的点，能使得原图增加几个连通分量呢？

- 如果该点是一个孤立的点，那么增加 -1 个。
- 如果该点不是割点，那么增加 0 个。
- 如果该点是割点且非根节点，那么增加该点在dfs树中(无反向边连回早期祖先的)的儿子数。
- 如果该点是割点且是一个dfs树的根节点，那么增加该点在dfs树中(无反向边连回早期祖先的)的儿子数-1 的数目，也就是增加了以该dfs树的儿子数目-1

该部分内容 转自[无向图求割顶与桥](#)

使用方法

1. 见 `work()` 函数

Tips

- 该模版修改完尚未经过测试，可能需要小修小补

模版

vector形式

```
//求无向图的割顶和桥
const int maxn=100000+10;      //顶点数
int n,m;//n个点 m条边 顶点下标0~n-1
int dfs_clock;//时钟，每访问一个节点增1
vector<int> G[maxn];//G[i]表示i节点邻接的所有节点
```

无向图的割顶和割桥

```
int pre[maxn]; //pre[i]表示i节点被第一次访问到的时间戳,若pre[i]==0表示i还未被访问
int low[maxn]; //low[i]表示i节点及其后代能通过反向边连回的最早的祖先的pre值
bool iscut[maxn]; //标记i节点是不是一个割点
int cut[maxn]; //cut[i]表示割i点的时图中联通分量的增加量
vector<pair<int, int>>Bridge;

int dfs(int u, int fa=-1) //求出以u为根节点 (u在DFS树中的父节点是fa) 的树的所有割顶和桥
{
    if (fa == -1) cut[u]--; //如果是根的话, 割时增加的量为“连出去的量减一”
    int lowu=pre[u]=++dfs_clock;
    int child=0; //子节点数目
    for(int i=0; i<G[u].size(); i++)
    {
        int v=G[u][i];
        if(!pre[v]){
            child++; //未访问过的节点才能算是u的孩子
            int lowv=dfs(v,u);
            lowu=min(lowu,lowv);
            if(lowv>=pre[u]){
                cut[u]++;
                iscut[u]=true; //u点是割顶
                if(lowv>pre[u]) //割桥判定
                    Bridge.push_back(make_pair(u, v));
            }
        }
        else if(pre[v]<pre[u] && v!=fa) { //v!=fa确保了(u,v)是从u到v的反向边
            lowu=min(lowu,pre[v]);
        }
    }
    if(fa<0 && child==1)
        iscut[u]=false; //u若是根且孩子数<=1, 那u就不是割顶
    return low[u]=lowu;
}

void work() {
    // input & initialize
    scanf("%d%d", &n, &m);
    dfs_clock=0; //初始化时钟
    memset(pre, 0, sizeof(pre));
    memset(iscut, 0, sizeof(iscut));
    memset(cut, 0, sizeof(cut));
```

```

memset(low, 0, sizeof(low));
Bridge.clear();
for(int i=1; i<=n; i++) G[i].clear();
for(int i=0; i<m; i++) {
    int u, v;
    scanf("%d%d", &u, &v); //下标1~n
    G[u].push_back(v);
    G[v].push_back(u);
}

//Application
int k = 0;
for (int i=1; i<=n; i++) {
    if (!pre[i]) {
        k++;
        dfs(i); //每次遍历一个连通块
    }
}

// output
printf("共有 %d 个连通量\n", k);
for(int i=1; i<=n; i++) {
    if(iscut[i]==true)
        printf("割顶是:%d\n", i);
}
for (int i=0; i<Bridge.size(); i++) {
    printf("割桥是:%d %d\n", Bridge[i].first, Bridge[i].second);
}
for (int i=1; i<=n; i++) {
    printf("当删除 %d 点 时, 会增加 %d个联通量\n", i, cut[i] );
}
}
}

```

邻接表形式

```

//求无向图的割顶和桥
const int Vmax=100000+10;      //顶点数
const int Emax=100000+10;      //顶点数
int dfs_clock; //时钟, 每访问一个节点增1
int pre[Vmax]; //pre[i]表示i节点被第一次访问到的时间戳, 若pre[i]==0表示i还未被访问
int low[Vmax]; //low[i]表示i节点及其后代能通过反向边连回的最早的祖先的pre值
bool iscut[Vmax]; //标记i节点是不是一个割点

```

无向图的割顶和割桥

```
int cut[Vmax]; //cut[i]表示割i点的时图中联通分量的增加量
vector<pair<int, int>>Bridge;
struct Edge{
    int v,next;
}e[Emax*2];
int he[Vmax],ecnt=0;
void adde(int u,int v){
    e[ecnt].v=v;
    e[ecnt].next = he[u];
    he[u] = ecnt++;
}

int dfs(int u,int fa=-1) //求出以u为根节点(u在DFS树中的父节点是fa)的树的所有割顶和桥
{
    if (fa == -1) cut[u]--;
    int lowu=pre[u]=++dfs_clock;
    int child=0;      //子节点数目
    for (int i=he[u]; i!= -1; i=e[i].next) {
        int v=e[i].v;
        if (!pre[v]){
            child++;
            int lowv=dfs(v,u);
            lowu=min(lowu,lowv);
            if (lowv>=pre[u]){
                cut[u]++;
                iscut[u]=true;           //u点是割顶
                if (lowv>pre[u]) //割桥判定
                    Bridge.push_back(make_pair(u, v));
            }
        }
        else if (pre[v]<pre[u] && v!=fa) { //v!=fa确保了(u,v)是从u到v的反向边
            lowu=min(lowu,pre[v]);
        }
    }
    if (fa<0 && child==1)
        iscut[u]=false; //u若是根且孩子数<=1,那u就不是割顶
    return low[u]=lowu;
}
void work(int n,int m){
    // input & initialize
    memset(he, -1, sizeof(he));
```

无向图的割顶和割桥

```
ecnt = 0;
dfs_clock=0;//初始化时钟
memset(pre,0,sizeof(pre));
memset(iscut,0,sizeof(iscut));
memset(cut, 0, sizeof(cut));
Bridge.clear();

for(int i=0;i<m;i++) {
    int u,v;
    scanf("%d%d", &u, &v); //下标1~n
    adde(u, v);
    adde(v, u);
}

//Application
int k = 0;
for (int i=1; i<=n; i++) {
    if (!pre[i]) {
        k++;
        dfs(i); //每次遍历一个连通块
    }
}

// output
printf("共有 %d 个连通量\n", k);
for(int i=1;i<=n;i++)
    if(iscut[i]==true)
        printf("割顶是:%d\n",i);
for (int i=0; i<Bridge.size(); i++) {
    printf("割桥是:%d %d\n",Bridge[i].first,Bridge[i].second);
}
for (int i=1; i<=n; i++) {
    printf("当删除 %d点 时, 会增加 %d个联通量\n",i,cut[i] );
}
}
```

无向图的双连通分量

说明

学习资料：

- 刘汝佳<<训练指南>>
- 无向图的双连通分量——饶齐

学习笔记

图的点-双连通定义：如果任意两点至少存在两条“点不重复”的路径，则说这个图是点-双连通的。

这个要求等价于任意两条边都在同一个简单环中，即内部无割点。也就是说，不含割点的图即为点-双连通图

图的边-双连通定义：如果任意两点之间至少存在两条“边不重复”的路径。

这个要求等价于每条边都至少在一个简单环中，即所有边都不是桥。也就是说，不含桥的图即为边-双连通图

特殊地，孤立点所形成的图是点-双连通的，也是边-双连通的，因为内部无割点、无割桥。

特殊地，孤立边是点-双连通的，而不是边-双连通的。

割桥将每一个边-双连通分量分开，`low[i]`的意义就是`low[i]`所连的块能返回的最早的祖先，也就是说，`low[i]`相同的点即为一个边连通分量。

使用方法

- 见 `work()` 函数

Tips

- 点的下标是从 0~n-1
- 使用该算法需要保证无重边
- 跑完模版之后，所有`low[i]`值相同的点的集合分别为一个边-双连通分量

模版

```
//点-双连通分量  
const int maxn=5000+10; //点数
```

```

struct Edge{
    int u,v;
    Edge(int _u,int _v) {
        u = _u, v = _v;
    }
};

int pre[maxn]; //第一次访问的dfs_clock时间戳
int low[maxn];
int iscut[maxn]; //割点判断
int bccno[maxn]; // bccno[i]表示i所在最早访问的点-双联通分量的下标 即
//bcc[bccno[i]]这个连通分量集合中含有i这个点 对于割顶来讲没有意义，因为他属于多
//个点-双联通分量
vector<int> belong[maxn]; //belong[i]表示i所在的双连通分量的下标的集合
int dfs_clock;
int bcc_cnt; // 双连通分量个数
vector<int> G[maxn]; // 顶点, 下标0~n-1
vector<int> bcc[maxn]; //点双连通分量存储结果 下标1~bcnt
stack<Edge> S;
int dfs(int u,int fa) {
    int lowu = pre[u] = ++dfs_clock;
    int child = 0;
    for (int i=0; i<G[u].size(); i++) {
        int v = G[u][i];
        Edge e = Edge(u, v);
        if (!pre[v]) {
            S.push(e);
            child++;
            int lowv = dfs(v, u);
            lowu = min(lowu, lowv);
            if (lowv >= pre[u]) {
                iscut[u] = true;
                bcc_cnt++;
                bcc[bcc_cnt].clear();
                while (true) {
                    Edge x = S.top();
                    S.pop();
                    if (bccno[x.u] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(x.u);
                        bccno[x.u] = bcc_cnt;
                        belong[x.u].push_back(bcc_cnt);
                    }
                    if (bccno[x.v] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(x.v);
                    }
                }
            }
        }
    }
}

```

```

        bccno[x.v] = bcc_cnt;
        belong[x.v].push_back(bcc_cnt);
    }
    if (x.u == u && x.v == v) {
        break;
    }
}

}

else if (pre[v] < pre[u] && v != fa) {
    S.push(e);
    lowu = min(lowu, pre[v]);
}
}

if (fa < 0 && child == 1) {
    iscut[u] = 0;
}
return low[u]=lowu;
}

int find_bcc(int n){//n个顶点
//栈无需清空，每次跑完必然为空
//bcc[]无需清空，组建连通分量时已清空
memset(pre, 0, sizeof(pre));
memset(iscut, 0, sizeof(iscut));
memset(bccno, 0, sizeof(bccno));
memset(low, 0, sizeof(low));
dfs_clock = bcc_cnt = 0;
int cnt = 0;
for (int i=1; i<=n; i++) {
    if (!pre[i]) {
        dfs(i, -1);
        cnt++;
    }
}
return cnt;
}

void work(int n,int m){// n个点 m条边
// input and initialize
for (int i=1; i<=n; i++) {
    G[i].clear();
    belong[i].clear();
}
}

```

无向图的双连通分量

```
}

for (int i=0; i<m; i++) {
    int u,v;
    scanf("%d%d", &u, &v); //index range: 1~n
    G[u].push_back(v);
    G[v].push_back(u);
}

// find biconnected component
int cnt = find_bcc(n);

// output
printf("共计%d个连通块\n", cnt);
printf("共计%d个点-双连通分量\n", bcc_cnt);
for (int i=1; i<=bcc_cnt; i++) {
    printf("第%d个点-双连通分量所含的点有: ", i);
    for (int j = 0; j<bcc[i].size(); j++) {
        printf("%d ", bcc[i][j]);
    }
    printf("\n");
}
}

int main() {
    int n,m; //n个点 m条边
    while (scanf("%d%d", &n, &m) !=EOF) {
        work(n, m);
    }
}
```

有向图的强连通分量

说明

学习资料：

- 刘汝佳 白书
- 有向图的强连通分量——饶齐

概念：

有向图强连通分量在有向图G中，如果两个顶点 v_i, v_j 间 ($v_i > v_j$) 有一条从 v_i 到 v_j 的有向路径，同时还有一条从 v_j 到 v_i 的有向路径，则称两个顶点强连通(strongly connected)。如果有向图G的每两个顶点都强连通，称G是一个强连通图。有向图的极大强连通子图，称为强连通分量(strongly connected components)。

使用方法

- 见 `work()` 函数

Tips

- 自行挑需要的东西写，如果附加的信息太多可能会MLE
- 一般都要缩点成为DAG，然后再做一些操作
- 对于 `scc_cnt == 1` 的情况偶尔需要特判

模版

```
const int maxn=5000+10;
int dfs_clock;//时钟
int scc_cnt;//强连通分量总数
vector<int> G[maxn];//G[i]表示i节点指向的所有点
vector<int> belong[maxn];//belong[i]表示第i个强联通分量包含的所有元素 下
标1~scc_cnt
int pre[maxn]; //时间戳
int low[maxn]; //u以及u的子孙能到达的祖先pre值
int sccno[maxn];//sccno[i]==j表示i节点属于j连通分量 sccno[i]的区间为
1~scc_cnt
int cnt[maxn];
stack<int> S;
```

有向图的强连通分量

```
void dfs(int u) {
    pre[u]=low[u]=++dfs_clock;
    S.push(u);
    for(int i=0;i<G[u].size();i++) {
        int v=G[u][i];
        if(!pre[v]) {
            dfs(v);
            low[u]=min(low[u],low[v]);
        }
        else if(!sccno[v]) {
            low[u]=min(low[u],pre[v]);
        }
    }
    if(low[u] == pre[u]){//u为当前强连通分量的入口
        scc_cnt++;
        belong[scc_cnt].clear();
        while(true) {
            int x=S.top(); S.pop();
            sccno[x]=scc_cnt;
            cnt[scc_cnt]++;
            belong[scc_cnt].push_back(x);
            if(x==u) break;
        }
    }
}

//求出有向图所有连通分量
void find_scc(int n) {
    scc_cnt=dfs_clock=0;
    memset(sccno,0,sizeof(sccno));
    memset(pre,0,sizeof(pre));
    for(int i=1;i<=n;i++)
        if(!pre[i]) dfs(i);
}

void work(int n,int m) {
    for(int i=1;i<=n;i++) G[i].clear();
    while(m--) {
        int u,v;
        scanf("%d%d",&u,&v); //index range: 1~n
        G[u].push_back(v);
    }
    find_scc(n);
    for(int i=1;i<=n;i++)
        printf("%d号点属于%d分量\n",i,sccno[i]);
}
```

```
}
```

```
int main() {
```

```
    int n,m;
```

```
    while (scanf("%d%d", &n, &m) !=EOF) {
```

```
        work(n,m);
```

```
    }
```

```
}
```

2-SAT

说明

学习资料

- 2-SAT问题——饶齐
- 2-SAT总结——kuangbin
- 白书

个人理解

使用方法

Tips

模版

```

const int maxn=10000+10;
struct TwoSAT
{
    int n; //原始图的节点数(未翻倍)
    vector<int> G[maxn*2]; //G[i]==j表示如果mark[i]=true,那么mark[j]也要=true
    bool mark[maxn*2]; //标记
    int S[maxn*2], c; //S和c用来记录一次dfs遍历的所有节点编号

    void init(int n)
    {
        this->n=n;
        for(int i=0;i<2*n;i++) G[i].clear();
        memset(mark, 0, sizeof(mark));
    }

    //加入(x, xval)或(y, yval)条件
    //xval=0表示假, yval=1表示真
    void add_clause(int x, int xval, int y, int yval)
    {

```

```

x=x*2+xval;
y=y*2+yval;
G[x^1].push_back(y);
G[y^1].push_back(x);
}

//从x执行dfs遍历，途径的所有点都标记
//如果不能标记，那么返回false
bool dfs(int x)
{
    if(mark[x^1]) return false;//这两句的位置不能调换
    if(mark[x]) return true;
    mark[x]=true;
    S[c++]=x;
    for(int i=0;i<G[x].size();i++)
        if(!dfs(G[x][i])) return false;
    return true;
}

//判断当前2-SAT问题是否有解
bool solve()
{
    for(int i=0;i<2*n;i+=2)
        if(!mark[i] && !mark[i+1])
    {
        c=0;
        if(!dfs(i))
        {
            while(c>0) mark[S[--c]]=false;
            if(!dfs(i+1)) return false;
        }
    }
    return true;
}
};

```

交叉染色

说明

- 刘汝佳<<训练指南>>P311页

二分图又称作二部图，是图论中的一种特殊模型。设 $G=(V,E)$ 是一个无向图，如果顶点 V 可分割为两个互不相交的子集 (A,B) ，并且图中的每条边 (i, j) 所关联的两个顶点 i 和 j 分别属于这两个不同的顶点集($i \text{ in } A, j \text{ in } B$)，则称图 G 为一个二分图。

二分图的另一种等价的说法是，可以把每个节点着以黑色和白色之一，使得每条边的两个端点颜色不同。不难发现，非连通的图是二分图当且仅当每个连通分量都是二分图，因此我们只考虑无向连通图。

使用方法

1. 见 `work()` 函数

Tips

- 待写

模版

DFS

```

//无向图的二分图判断
const int maxn=1000+5;
int n;//图节点数
vector<int> G[maxn];//G[i]表示i节点邻接的点
int color[maxn];//color[i]=0,1,2 表i节点 不涂颜色 涂白色 涂黑色
//判断无向图是否可二分
bool bipartite(int u)
{
    for(int i=0;i<G[u].size();i++)
    {
        int v=G[u][i];
        if(color[v]==color[u]) return false;
        if(!color[v])
        {
            color[v]=3-color[u];
            if(!bipartite(v)) return false;
        }
    }
    return true;
}

```

BFS

```

//HDU5285
vector<int> G[maxn];
int c[maxn];
int main() {
    int T;
    scanf("%d", &T);
    while(T--) {
        int n,m;
        scanf("%d%d", &n, &m);
        for (int i = 1 ; i <= n ; i++) G[i].clear();
        for (int i = 1 ; i <= m ; i++) {
            int u,v;
            scanf("%d%d", &u, &v);
            G[u].push_back(v);
            G[v].push_back(u);
        }
        if (n < 2) {
            puts("Poor why");
        }
    }
}
```

```

        continue;
    }

    if (m == 0) {
        printf("%d %d\n", n-1, 1);
        continue;
    }

/* --START---交叉染色---START--- */
memset(c, -1, sizeof(c)); //标记所有颜色
bool flag = true; //判断是否可以完成染色
queue<int>q;
int mx = 0, mn = 0;
bool flag = true;
for (int i = 1; i <= n && flag ; i++) {
    if (c[i] == -1) {
        int tmp1 = 0, tmp2 = 0;
        q.push(i);
        c[i] = 1;
        while (!q.empty() && flag) {
            int u = q.front(); q.pop();
            if (c[u] == 1) tmp1++;
            else tmp2++;
            for (int i = 0; i < G[u].size() && flag ; i++) {
                int v = G[u][i];
                if (c[v] == -1) {
                    c[v] = c[u]^1;
                    q.push(v);
                }
                else if (c[v] != c[u]^1)
                    flag = false;
            }
        }
        mx += max(tmp1, tmp2);
        mn += min(tmp2, tmp1);
    }
}

if (flag) printf("%d %d\n", mx,mn); //输出单色出现次数最多的 单色
出现次数最少的
else puts("Poor wyh"); //无法完成染色
/* --END---交叉染色---END--- */
}
}

```


JAVA输入挂

模版

```
// uwi输入挂

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.math.BigInteger;
import java.util.Arrays;
import java.util.InputMismatchException;

public class Main {

    static InputStream is;
    static PrintWriter out;
    static String INPUT = "";

    static void solve()
    {
        //        while (!eof()) {
        //
        //        }

        int T = ni();
        for(int cas = 1 ; cas <= T; cas++) {
            // 代码开始
            char[] s = ns().toCharArray();
            Arrays.sort(s);
            int n = s.length;
            BigInteger can = null;
            int nz = 0;
            for(int i = 0;i < n;i++) {
                if(s[i] != '0')nz++;
            }
            if(nz <= 1){
                out.println("Uncertain");
            }else{
                char[] ret = new char[n];
                Arrays.fill(ret, '0');
                char sel = 0;
```

```

        int p = 0;
        for(int i = 0;i < n;i++) {
            if(s[i] != '0' && sel == 0) {
                sel = s[i];
            }else{
                ret[p++] = s[i];
            }
        }
        ret[0] += sel-'0';
        for(int i = 0;i < n;i++) {
            if(ret[i] > '9'){
                ret[i] = (char)(ret[i]-10);
                ret[i+1]++;
            }else{
                break;
            }
        }
        boolean z = true;
        for(int i = n-1;i >= 0;i--) {
            if(ret[i] == '0' && z)continue;
            z = false;
            out.print(ret[i]);
        }
        out.println();
    }
    // 代码结束
}

public static void main(String[] args) throws Exception
{
//    long S = System.currentTimeMillis(); //初始时间
//    is =System.in : new
ByteArrayInputStream(INPUT.getBytes());
is = System.in;
out = new PrintWriter(System.out);

solve();
out.flush(); //输出
//    long G = System.currentTimeMillis(); //结束时间
//    tr(G-S+"ms");
}

```

```

private static boolean eof()
{
    if(lenbuf == -1) return true;
    int lptr = ptrbuf;
    while(lptr < lenbuf)if(!isSpaceChar(inbuf[lptr++]))return
false;

    try {
        is.mark(1000);
        while(true){
            int b = is.read();
            if(b == -1){
                is.reset();
                return true;
            }else if(!isSpaceChar(b)){
                is.reset();
                return false;
            }
        }
    } catch (IOException e) {
        return true;
    }
}

private static byte[] inbuf = new byte[1024];
static int lenbuf = 0, ptrbuf = 0;

private static int readByte()
{
    if(lenbuf == -1)throw new InputMismatchException();
    if(ptrbuf >= lenbuf){
        ptrbuf = 0;
        try { lenbuf = is.read(inbuf); } catch (IOException e) {
throw new InputMismatchException(); }
        if(lenbuf <= 0)return -1;
    }
    return inbuf[ptrbuf++];
}

private static boolean isSpaceChar(int c) { return !(c >= 33 &&
c <= 126); }

private static int skip() { int b; while((b = readByte()) != -1
&& isSpaceChar(b)); return b; }

```

```

//输入double
private static double nd() { return Double.parseDouble(ns()); }

//输入char
private static char nc() { return (char)skip(); }

//输入一个String
private static String ns()
{
    int b = skip();
    StringBuilder sb = new StringBuilder();
    while(!isSpaceChar(b)){ // when nextLine, (isSpaceChar(b)
&& b != ' ')
        sb.appendCodePoint(b);
        b = readByte();
    }
    return sb.toString();
}

//输入一个char数组
private static char[] ns(int n)
{
    char[] buf = new char[n];
    int b = skip(), p = 0;
    while(p < n && !(isSpaceChar(b))){
        buf[p++] = (char)b;
        b = readByte();
    }
    return n == p ? buf : Arrays.copyOf(buf, p);
}

//输入一个矩阵
private static char[][] nm(int n, int m)
{
    char[][] map = new char[n][];
    for(int i = 0;i < n;i++)map[i] = ns(m);
    return map;
}

//输入一个输入n整数
private static int[] na(int n)
{
    int[] a = new int[n];

```

```

        for(int i = 0;i < n;i++)a[i] = ni();
        return a;
    }

//输入一个整数
private static int ni()
{
    int num = 0, b;
    boolean minus = false;
    while((b = readByte()) != -1 && !((b >= '0' && b <= '9') ||
b == '-'));
        if(b == '-'){
            minus = true;
            b = readByte();
        }

    while(true){
        if(b >= '0' && b <= '9'){
            num = num * 10 + (b - '0');
        }else{
            return minus ? -num : num;
        }
        b = readByte();
    }
}

//输入一个long
private static long nl()
{
    long num = 0;
    int b;
    boolean minus = false;
    while((b = readByte()) != -1 && !((b >= '0' && b <= '9') ||
b == '-'));
        if(b == '-'){
            minus = true;
            b = readByte();
        }

    while(true){
        if(b >= '0' && b <= '9'){
            num = num * 10 + (b - '0');
        }else{

```

```
        return minus ? -num : num;
    }
    b = readByte();
}
}

private static void tr(Object... o) { if(INPUT.length() != 0) System.out.println(Arrays.deepToString(o)); }
```

JAVA大数类

Java大数方法

不可变的任意精度的整数。所有操作中，都以二进制补码形式表示 BigInteger（如 Java 的基本整数类型）。BigInteger 提供所有 Java 的基本整数操作符的对应物，并提供 java.lang.Math 的所有相关方法。另外，BigInteger 还提供以下运算：模算术、GCD 计算、质数测试、素数生成、位操作以及一些其他操作。

算术运算的语义完全模仿 Java 整数算术运算符的语义，如 The Java Language Specification 中所定义的。例如，以零作为除数的除法抛出

ArithmException，而负数除以正数的除法则产生一个负（或零）的余数。Spec 中关于溢出的细节都被忽略了，因为 BigIntegers 所设置的实际大小能适应操作结果的需要。

位移操作的语义扩展了 Java 的位移操作符的语义以允许产生负位移距离。带有负位移距离的右移操作会导致左移操作，反之亦然。忽略无符号的右位移运算符 (>>>)，因为该操作与由此类提供的“无穷大的词大小”抽象结合使用时毫无意义。

逐位逻辑运算的语义完全模仿 Java 的逐位整数运算符的语义。在执行操作之前，二进制运算符 (and、or、xor) 对两个操作数中的较短操作数隐式执行符号扩展。

比较操作执行有符号的整数比较，类似于 Java 的关系运算符和相等性运算符执行的比较。

提供的模算术操作用来计算余数、求幂和乘法可逆元。这些方法始终返回非负结果，范围在 0 和 (modulus - 1)（包括）之间。

位操作对其操作数的二进制补码表示形式的单个位进行操作。如有必要，操作数会通过扩展符号来包含指定的位。单一位操作不能产生与正在被操作的 BigInteger 符号不同的 BigInteger，因为它们仅仅影响单个位，并且此类提供的“无穷大词大小”抽象可保证在每个 BigInteger 前存在无穷多的“虚拟符号位”数。

为了简洁明了，在整个 BigInteger 方法的描述中都使用了伪代码。伪代码表达式 ($i + j$) 是“其值为 BigInteger i 加 BigInteger j 的 BigInteger”的简写。伪代码表达式 ($i == j$) 是“当且仅当 BigIntegeri 表示与 BigIntegerj 相同的值时，才为true”的简写。可以类似地解释其他伪代码表达式。

当为任何输入参数传递 null 对象引用时，此类中的所有方法和构造方法都将抛出 NullPointerException。

字段摘要

```

static BigInteger ONE
    BigInteger 的常量 1。
static BigInteger TEN
    BigInteger 的常量 10。
static BigInteger ZERO
    BigInteger 的常量 0。

```

构造方法摘要

```

BigInteger(byte[] val)
    将包含 BigInteger 的二进制补码表示形式的字节数组转换为
    BigInteger。
BigInteger(int signum, byte[] magnitude)
    将 BigInteger 的符号-数量表示形式转换为 BigInteger。
BigInteger(int bitLength, int certainty, Random rnd)
    构造一个随机生成的正 BigInteger，它可能是一个具有指定 bitLength
    的素数。
BigInteger(int numBits, Random rnd)
    构造一个随机生成的 BigInteger，它是在 0 到 ( $2^{numBits} - 1$ ) (包
    括) 范围内均匀分布的值。
BigInteger(String val)
    将 BigInteger 的十进制字符串表示形式转换为 BigInteger。
BigInteger(String val, int radix)
    将指定基数的 BigInteger 的字符串表示形式转换为 BigInteger。

```

方法摘要

读入：

```

public BigInteger nextBigInteger(int radix)
    输入以radix为进制数的BigInteger

```

```

BigInteger abs()
    返回其值是此 BigInteger 的绝对值的 BigInteger。
BigInteger add(BigInteger val)
    返回其值为 (this + val) 的 BigInteger。
BigInteger and(BigInteger val)
    返回其值为 (this & val) 的 BigInteger。
BigInteger andNot(BigInteger val)
    返回其值为 (this & ~val) 的 BigInteger。
int bitCount()
    返回此 BigInteger 的二进制补码表示形式中与符号不同的位的数量。
int bitLength()

```

返回此 BigInteger 的最小的二进制补码表示形式的位数，不包括 符号位。

```
BigInteger clearBit(int n)
```

返回其值与清除了指定位的此 BigInteger 等效的 BigInteger。

```
int compareTo(BigInteger val)
```

将此 BigInteger 与指定的 BigInteger 进行比较。

```
BigInteger divide(BigInteger val)
```

返回其值为 $(this / val)$ 的 BigInteger。

```
BigInteger[] divideAndRemainder(BigInteger val)
```

返回包含 $(this / val)$ 后跟 $(this \% val)$ 的两个 BigInteger 的数组。

```
double doubleValue()
```

将此 BigInteger 转换为 double。

```
boolean equals(Object x)
```

比较此 BigInteger 与指定的 Object 的相等性。

```
BigInteger flipBit(int n)
```

返回其值与对此 BigInteger 进行指定位翻转后的值等效的 BigInteger。

```
float floatValue()
```

将此 BigInteger 转换为 float。

```
BigInteger gcd(BigInteger val)
```

返回一个 BigInteger，其值是 $\text{abs}(\text{this})$ 和 $\text{abs}(\text{val})$ 的最大公约数。

```
int getLowestSetBit()
```

返回此 BigInteger 最右端（最低位）1 比特的索引（即从此字节的右端开始到本字节中最右端 1 比特之间的 0 比特的位数）。

```
int hashCode()
```

返回此 BigInteger 的哈希码。

```
int intValue()
```

将此 BigInteger 转换为 int。

```
boolean isProbablePrime(int certainty)
```

如果此 BigInteger 可能为素数，则返回 true，如果它一定为合数，则返回 false。

```
long longValue()
```

将此 BigInteger 转换为 long。

```
BigInteger max(BigInteger val)
```

返回此 BigInteger 和 val 的最大值。

```
BigInteger min(BigInteger val)
```

返回此 BigInteger 和 val 的最小值。

```
BigInteger mod(BigInteger m)
```

返回其值为 $(\text{this} \bmod m)$ 的 BigInteger。

```
BigInteger modInverse(BigInteger m)
```

返回其值为 $(\text{this}^{-1} \bmod m)$ 的 BigInteger。

```

BigInteger modPow(BigInteger exponent, BigInteger m)
    返回其值为 (thisexponent mod m) 的 BigInteger。
BigInteger multiply(BigInteger val)
    返回其值为 (this * val) 的 BigInteger。
BigInteger negate()
    返回其值是 (-this) 的 BigInteger。
BigInteger nextProbablePrime()
    返回大于此 BigInteger 的可能为素数的第一个整数。
BigInteger not()
    返回其值为 (~this) 的 BigInteger。
BigInteger or(BigInteger val)
    返回其值为 (this | val) 的 BigInteger。
BigInteger pow(int exponent)
    返回其值为 (thisexponent) 的 BigInteger。
static BigInteger probablePrime(int bitLength, Random rnd)
    返回有可能是素数的、具有指定长度的正 BigInteger。
BigInteger remainder(BigInteger val)
    返回其值为 (this % val) 的 BigInteger。
BigInteger setBit(int n)
    返回其值与设置了指定位的此 BigInteger 等效的 BigInteger。
BigInteger shiftLeft(int n)
    返回其值为 (this << n) 的 BigInteger。
BigInteger shiftRight(int n)
    返回其值为 (this >> n) 的 BigInteger。
int signum()
    返回此 BigInteger 的正负号函数。
BigInteger subtract(BigInteger val)
    返回其值为 (this - val) 的 BigInteger。
boolean testBit(int n)
    当且仅当设置了指定的位时，返回 true。
byte[] toByteArray()
    返回一个字节数组，该数组包含此 BigInteger 的二进制补码表示形式。
String toString()
    返回此 BigInteger 的十进制字符串表示形式。
String toString(int radix)
    返回此 BigInteger 的给定基数的字符串表示形式。
static BigInteger valueOf(long val)
    返回其值等于指定 long 的值的 BigInteger。
BigInteger xor(BigInteger val)
    返回其值为 (this ^ val) 的 BigInteger。

```

使用举例

```
//大数输入及相乘
import java.math.*;
import java.util.*;
public class Main{
    static BigInteger zero = BigInteger.valueOf(0);
    static BigInteger one = BigInteger.valueOf(1);
    static BigInteger two = BigInteger.valueOf(2);
    public static void main(String[] args)
    {
        Scanner input=new Scanner(System.in);
        int T,cas=1;
        T = input.nextInt();
        String s1,s2;

        while (T-->0) {
            BigInteger a = input.nextBigInteger(2); //以二进制方式
            输入a
            BigInteger b = input.nextBigInteger(2);
            System.out.println("Case #"+cas+":");
            "+a.gcd(b).toString(2));
            cas++;
        }
    }
}
```

其他

执行到EOF—— `while(input.hasNext()){...}`

高精度计算-大数类

说明

转自：[大整数类BIGN的设计与实现 C++高精度模板](#)

使用时可以选择需要的部分使用

Tips

- 注意最大长度，一般2000足够用了

模版

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

#define MAX_L 2005 //最大长度，可以修改
class bign
{
public:
    int len, s[MAX_L]; //数的长度，记录数组
//构造函数
    bign();
    bign(const char* );
    bign(int);
    bool sign; //符号 1正数 0负数
    string toStr() const; //转化为字符串，主要是便于输出
    friend istream& operator>>(istream &, bign &); //重载输入流
    friend ostream& operator<<(ostream &, bign &); //重载输出流
//重载复制
    bign operator=(const char* );
    bign operator=(int );
```

```

bign operator=(const string);
//重载各种比较
    bool operator>(const bign &) const;
    bool operator>=(const bign &) const;
    bool operator<(const bign &) const;
    bool operator<=(const bign &) const;
    bool operator==(const bign &) const;
    bool operator!=(const bign &) const;
//重载四则运算
    bign operator+(const bign &) const;
    bign operator++();
    bign operator++(int);
    bign operator+=(const bign&);
    bign operator-(const bign &) const;
    bign operator--();
    bign operator--(int);
    bign operator.=(const bign&);
    bign operator*(const bign &) const;
    bign operator*(const int num) const;
    bign operator*=(const bign&);
    bign operator/(const bign&) const;
    bign operator/=(const bign&);
//四则运算的衍生运算
    bign operator%(const bign&) const;//取模 (余数)
    bign factorial() const;//阶乘
    bign Sqrt() const;//整数开根 (向下取整)
    bign pow(const bign&) const;//次方
//一些乱乱的函数
    void clean();
    ~bign();
};

#define max(a,b) a>b ? a : b
#define min(a,b) a<b ? a : b

bign::bign()
{
    memset(s, 0, sizeof(s));
    len = 1;
    sign = 1;
}

bign::bign(const char *num)
{

```

```

        *this = num;
    }

bign::bign(int num)
{
    *this = num;
}

string bign::toString() const
{
    string res;
    res = "";
    for (int i = 0; i < len; i++)
        res = (char)(s[i] + '0') + res;
    if (res == "")
        res = "0";
    if (!sign&&res != "0")
        res = "-" + res;
    return res;
}

istream &operator>>(istream &in, bign &num)
{
    string str;
    in>>str;
    num=str;
    return in;
}

ostream &operator<<(ostream &out, bign &num)
{
    out<<num.toString();
    return out;
}

bign bign::operator=(const char *num)
{
    memset(s, 0, sizeof(s));
    char a[MAX_L] = "";
    if (num[0] != '-')
        strcpy(a, num);
    else
        for (int i = 1; i < strlen(num); i++)

```

```

        a[i - 1] = num[i];
    sign = !(num[0] == '-');
    len = strlen(a);
    for (int i = 0; i < strlen(a); i++)
        s[i] = a[len - i - 1] - 48;
    return *this;
}

bign bign::operator=(int num)
{
    char temp[MAX_L];
    sprintf(temp, "%d", num);
    *this = temp;
    return *this;
}

bign bign::operator=(const string num)
{
    const char *tmp;
    tmp = num.c_str();
    *this = tmp;
    return *this;
}

bool bign::operator<(const bign &num) const
{
    if (sign^num.sign)
        return num.sign;
    if (len != num.len)
        return len < num.len;
    for (int i = len - 1; i >= 0; i--)
        if (s[i] != num.s[i])
            return sign ? (s[i] < num.s[i]) : (!(s[i] < num.s[i]));
    return !sign;
}

bool bign::operator>(const bign&num) const
{
    return num < *this;
}

bool bign::operator<=(const bign&num) const
{
}

```

```

        return !(*this>num);
    }

bool bign::operator>=(const bign&num) const
{
    return !(*this<num);
}

bool bign::operator!=(const bign&num) const
{
    return *this > num || *this < num;
}

bool bign::operator==(const bign&num) const
{
    return !(num != *this);
}

bign bign::operator+(const bign &num) const
{
    if (sign^num.sign)
    {
        bign tmp = sign ? num : *this;
        tmp.sign = 1;
        return sign ? *this - tmp : num - tmp;
    }
    bign result;
    result.len = 0;
    int temp = 0;
    for (int i = 0; temp || i < (max(len, num.len)); i++)
    {
        int t = s[i] + num.s[i] + temp;
        result.s[result.len++] = t % 10;
        temp = t / 10;
    }
    result.sign = sign;
    return result;
}

bign bign::operator++()
{
    *this = *this + 1;
    return *this;
}

```

```

}

bign bign::operator++(int)
{
    bign old = *this;
    ++(*this);
    return old;
}

bign bign::operator+=(const bign &num)
{
    *this = *this + num;
    return *this;
}

bign bign::operator-(const bign &num) const
{
    bign b=num, a=*this;
    if (!num.sign && !sign)
    {
        b.sign=1;
        a.sign=1;
        return b-a;
    }
    if (!b.sign)
    {
        b.sign=1;
        return a+b;
    }
    if (!a.sign)
    {
        a.sign=1;
        b=bign(0)-(a+b);
        return b;
    }
    if (a<b)
    {
        bign c=(b-a);
        c.sign=false;
        return c;
    }
    bign result;
    result.len = 0;
}

```

```

    for (int i = 0, g = 0; i < a.len; i++)
    {
        int x = a.s[i] - g;
        if (i < b.len) x -= b.s[i];
        if (x >= 0) g = 0;
        else
        {
            g = 1;
            x += 10;
        }
        result.s[result.len++] = x;
    }
    result.clean();
    return result;
}

bign bign::operator * (const bign &num) const
{
    bign result;
    result.len = len + num.len;

    for (int i = 0; i < len; i++)
        for (int j = 0; j < num.len; j++)
            result.s[i + j] += s[i] * num.s[j];

    for (int i = 0; i < result.len; i++)
    {
        result.s[i + 1] += result.s[i] / 10;
        result.s[i] %= 10;
    }
    result.clean();
    result.sign = !(sign^num.sign);
    return result;
}

bign bign::operator*(const int num) const
{
    bign x = num;
    bign z = *this;
    return x*z;
}

bign bign::operator*=(const bign&num)
{

```

```

        *this = *this * num;
        return *this;
    }

bign bign::operator /(const bign&num) const
{
    bign ans;
    ans.len = len - num.len + 1;
    if (ans.len < 0)
    {
        ans.len = 1;
        return ans;
    }

    bign divisor = *this, divid = num;
    divisor.sign = divid.sign = 1;
    int k = ans.len - 1;
    int j = len - 1;
    while (k >= 0)
    {
        while (divisor.s[j] == 0) j--;
        if (k > j) k = j;
        char z[MAX_L];
        memset(z, 0, sizeof(z));
        for (int i = j; i >= k; i--)
            z[j - i] = divisor.s[i] + '0';
        bign dividend = z;
        if (dividend < divid) { k--; continue; }
        int key = 0;
        while (divid*key <= dividend) key++;
        key--;
        ans.s[k] = key;
        bign temp = divid*key;
        for (int i = 0; i < k; i++)
            temp = temp * 10;
        divisor = divisor - temp;
        k--;
    }

    ans.clean();
    ans.sign = !(sign^num.sign);
    return ans;
}

```

```

bign bign::operator/=(const bign&num)
{
    *this = *this / num;
    return *this;
}

bign bign::operator%(const bign& num) const
{
    bign a = *this, b = num;
    a.sign = b.sign = 1;
    bign result, temp = a / b*b;
    result = a - temp;
    result.sign = sign;
    return result;
}

bign bign::pow(const bign& num) const
{
    bign result = 1;
    for (bign i = 0; i < num; i++)
        result = result*(*this);
    return result;
}

bign bign::factorial() const
{
    bign result = 1;
    for (bign i = 1; i <= *this; i++)
        result *= i;
    return result;
}

void bign::clean()
{
    if (len == 0) len++;
    while (len > 1 && s[len - 1] == '\0')
        len--;
}

bign bign::Sqrt() const
{
    if (*this<0) return -1;
    if (*this<=1) return *this;
}

```

```
bign l=0, r=*this, mid;
while(r-l>1)
{
    mid=(l+r)/2;
    if(mid*mid>*this)
        r=mid;
    else
        l=mid;
}
return l;

bign::~bign()
{
}

bign num0, num1, res;

int main()
{
    num0 = 1, num1 = 2;
    res=num0-num1;
    cout << res << endl;
    return 0;
}
```

double的比较

模版

```
inline bool operator == (const double &a, const double &b) {
    return fabs(a - b) < eps;
}

inline bool operator != (const double &a, const double &b) {
    return fabs(a - b) > eps;
}

inline bool operator > (const double &a, const double &b) {
    return a - b > eps;
}

inline bool operator >= (const double &a, const double &b) {
    return a - b > -eps;
}

inline bool operator < (const double &a, const double &b) {
    return a - b < -eps;
}

inline bool operator <= (const double &a, const double &b) {
    return a - b < eps;
}
```

矩阵类

改编自范神: ACM-ICPC-Template/模板/9_数学/5_矩阵类.cpp

Tips

请注意 `unit()` 方法, 该方法返回值为一个单位矩阵

模版

```

typedef long long mytype;
const int SZ=105;
const long long mod=1000000007;
long long quickpow(long long a, long long b)
{
    if(b < 0) return 0;
    long long ret = 1;
    a %= mod;
    for (; b; b >= 1, a = (a * a) % mod)
        if (b & 1)
            ret = (ret * a) % mod;
    return ret;
}
long long inv(long long a)
{
    return quickpow(a,mod-2);
}
struct mat
{
    int n,m;
    mytype a[SZ][SZ];
    //构造函数 n行m列
    mat(int n=SZ,int m=SZ):n(n),m(m){}

    // 初始化
    void init(); //清零
    mat unit(); //该函数的返回值为一个单位矩阵

    // 输入输出

```

```

void in();
void out();

//基本运算
mytype *operator [](int n);
mat operator +(const mat &b);
mat operator -(const mat &b);
mat operator *(const mat &b);
mat operator *(const mytype &b);
mat operator /(const mytype &b);
mat operator !(); //矩阵的转置

//矩阵快速幂
friend mat quickpow(mat a, mytype b);
};

void mat::init()
{
    memset(a, 0, sizeof(a));
}

mat mat::unit()
{
    mat t(n, n);
    t.init();
    for (int i=0; i<n; i++)
        t.a[i][i]=1;
    return t;
}

void mat::in()
{
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            scanf("%lld", &a[i][j]);
}

void mat::out()
{
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            printf("%lld%c", a[i][j], " \n"[j==m-1]);
}

mytype *mat::operator [](int n)
{
    return * (a+n);
}

```

矩阵类

```
mat mat::operator +(const mat &b)
{
    mat t(n,m);
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            t.a[i][j]=(a[i][j]+b.a[i][j]+mod)%mod;
    return t;
}

mat mat::operator -(const mat &b)
{
    mat t(n,m);
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            t.a[i][j]=(a[i][j]-b.a[i][j]+mod)%mod;
    return t;
}

mat mat::operator *(const mat &b)
{
    mat t(n,b.m);
    for(int i=0; i<n; i++)
        for(int j=0; j<b.m; j++)
    {
        t.a[i][j]=0;
        for(int k=0; k<m; k++)
            t.a[i][j]=(t.a[i][j]+(a[i][k]*b.a[k][j])%mod)%mod;
    }
    return t;
}

mat mat::operator *(const mytype &b)
{
    mat t(n,m);
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            t.a[i][j]=a[i][j]*b%mod;
    return t;
}

mat mat::operator /(const mytype &b)
{
    mat t(n,m);
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            t.a[i][j]=a[i][j]*inv(b)%mod;
    return t;
}
```

矩阵类

```
}

mat mat::operator !()
{
    mat t(m, n);
    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++)
            t.a[i][j]=a[j][i];
    return t;
}

mat quickpow(mat a, mytype b)
{
    if(b<0) return a.unit();
    mat ret=a.unit();
    for (; b; b>>=1, a=a*a)
        if (b&1)
            ret=ret*a;
    return ret;
}
```

计算几何

转自qscquesze: [计算几何模板](#)

用前先膜

模版

目录

(一) 点的基本运算

1. 平面上两点之间距离 1
2. 判断两点是否重合 1
3. 矢量叉乘 1
4. 矢量点乘 2
5. 判断点是否在线段上 2
6. 求一点绕某点旋转后的坐标 2
7. 求矢量夹角 2

(二) 线段及直线的基本运算

1. 点与线段的关系 3
2. 求点到线段所在直线垂线的垂足 4
3. 点到线段的最近点 4
4. 点到线段所在直线的距离 4
5. 点到折线集的最近距离 4
6. 判断圆是否在多边形内 5
7. 求矢量夹角余弦 5
8. 求线段之间的夹角 5
9. 判断线段是否相交 6
10. 判断线段是否相交但不交在端点处 6
11. 求线段所在直线的方程 6
12. 求直线的斜率 7
13. 求直线的倾斜角 7
14. 求点关于某直线的对称点 7
15. 判断两条直线是否相交及求直线交点 7
16. 判断线段是否相交, 如果相交返回交点 7

(三) 多边形常用算法模块

1. 判断多边形是否简单多边形 8
2. 检查多边形顶点的凸凹性 9
3. 判断多边形是否凸多边形 9
4. 求多边形面积 9

- 5. 判断多边形顶点的排列方向, 方法一 10
- 6. 判断多边形顶点的排列方向, 方法二 10
- 7. 射线法判断点是否在多边形内 10
- 8. 判断点是否在凸多边形内 11
- 9. 寻找点集的graham算法 12
- 10. 寻找点集凸包的卷包裹法 13
- 11. 判断线段是否在多边形内 14
- 12. 求简单多边形的重心 15
- 13. 求凸多边形的重心 17
- 14. 求肯定在给定多边形内的一个点 17
- 15. 求从多边形外一点出发到该多边形的切线 18
- 16. 判断多边形的核是否存在 19

(四) 圆的基本运算

- 1. 点是否在圆内 20
- 2. 求不共线的三点所确定的圆 21

(五) 矩形的基本运算

- 1. 已知矩形三点坐标, 求第4点坐标 22

(六) 常用算法的描述 22

(七) 补充

- 1. 两圆关系: 24
- 2. 判断圆是否在矩形内: 24
- 3. 点到平面的距离: 25
- 4. 点是否在直线同侧: 25
- 5. 镜面反射线: 25
- 6. 矩形包含: 26
- 7. 两圆交点: 27
- 8. 两圆公共面积: 28
- 9. 圆和直线关系: 29
- 10. 内切圆: 30
- 11. 求切点: 31
- 12. 线段的左右旋: 31
- 13. 公式: 32

*/

/* 需要包含的头文件 */

```
#include <cmath>
```

/* 常用的常量定义 */

```
const double INF = 1E200
const double EP = 1E-10
```

```

const int MAXV = 300
const double PI = 3.14159265

/* 基本几何结构 */
struct POINT
{
    double x;
    double y;
    POINT(double a=0, double b=0) { x=a; y=b; } //constructor
};

struct LINESEG
{
    POINT s;
    POINT e;
    LINESEG(POINT a, POINT b) { s=a; e=b; }
    LINESEG() {}
};

struct LINE           // 直线的解析方程 a*x+b*y+c=0 为统一表示, 约定 a
>= 0
{
    double a;
    double b;
    double c;
    LINE(double d1=1, double d2=-1, double d3=0) { a=d1; b=d2; c=d3; }
};

/*****************
 *      *
 * 点的基本运算      *
 *      *
*****************/
double dist(POINT p1,POINT p2)           // 返回两点之间欧氏距离
{
    return( sqrt( (p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y) ) )
}

bool equal_point(POINT p1,POINT p2)         // 判断两个点是否重合
{
    return ( (abs(p1.x-p2.x)<EP) && (abs(p1.y-p2.y)<EP) );
}

/*****************
 *      *

```

```

r=multiply(sp,ep,op),得到(sp-op)和(ep-op)的叉积
r>0: ep在矢量opsp的逆时针方向;
r=0: opsp三点共线;
r<0: ep在矢量opsp的顺时针方向
*****
*****/
double multiply(POINT sp,POINT ep,POINT op)
{
    return((sp.x-op.x)*(ep.y-op.y)-(ep.x-op.x)*(sp.y-op.y));
}
/*
r=dotmultiply(p1,p2,op),得到矢量(p1-op)和(p2-op)的点积,如果两个矢量都非
零矢量
r<0: 两矢量夹角为钝角;
r=0: 两矢量夹角为直角;
r>0: 两矢量夹角为锐角
*****
*****/
double dotmultiply(POINT p1,POINT p2,POINT p0)
{
    return ((p1.x-p0.x)*(p2.x-p0.x)+(p1.y-p0.y)*(p2.y-p0.y));
}
/*
判断点p是否在线段l上
条件: (p在线段l所在的直线上) && (点p在以线段l为对角线的矩形内)
*****
*****/
bool online(LINESEG l,POINT p)
{
    return( (multiply(l.e,p,l.s)==0) && ( ( (p.x-l.s.x)*(p.x-l.e.x)
<=0 ) && ( (p.y-l.s.y)*(p.y-l.e.y)<=0 ) ) );
}
// 返回点p以点o为圆心逆时针旋转alpha(单位: 弧度)后所在的位置
POINT rotate(POINT o,double alpha,POINT p)
{
    POINT tp;
    p.x-=o.x;
    p.y-=o.y;
    tp.x=p.x*cos(alpha)-p.y*sin(alpha)+o.x;
    tp.y=p.y*cos(alpha)+p.x*sin(alpha)+o.y;
    return tp;
}

```

```

/* 返回顶角在o点, 起始边为os, 终止边为oe的夹角(单位: 弧度)
角度小于pi, 返回正值
角度大于pi, 返回负值
可以用于求线段之间的夹角
原理:
r = dotmultiply(s,e,o) / (dist(o,s)*dist(o,e))
r'= multiply(s,e,o)

r >= 1 angle = 0;
r <= -1 angle = -PI
-1<r<1 && r'>0 angle = arccos(r)
-1<r<1 && r'<=0 angle = -arccos(r)
*/
double angle(POINT o,POINT s,POINT e)
{
    double cosfi,fi,norm;
    double dsx = s.x - o.x;
    double dsy = s.y - o.y;
    double dex = e.x - o.x;
    double dey = e.y - o.y;

    cosfi=dsx*dex+dsy*dey;
    norm=(dsx*dsx+dsy*dsy)*(dex*dex+dey*dey);
    cosfi /= sqrt( norm );

    if (cosfi >= 1.0 ) return 0;
    if (cosfi <= -1.0 ) return -3.1415926;

    fi=acos(cosfi);
    if (dsx*dey-dsy*dex>0) return fi;           // 说明矢量os 在矢量 oe的顺
时针方向
    return -fi;
}
*****\*
*          *
*      线段及直线的基本运算      *
*          *
\*****/
/* 判断点与线段的关系,用途很广泛
本函数是根据下面的公式写的, P是点C到线段AB所在直线的垂足
AC dot AB

```

```

r = -----
| |AB| |^2
(Cx-Ax) (Bx-Ax) + (Cy-Ay) (By-Ay)
= -----
L^2

r has the following meaning:

r=0      P = A
r=1      P = B
r<0    P is on the backward extension of AB
r>1    P is on the forward extension of AB
0<r<1  P is interior to AB
*/
double relation(POINT p,LINESEG l)
{
    LINESEG tl;
    tl.s=l.s;
    tl.e=p;
    return dotmultiply(tl.e,l.e,l.s)/(dist(l.s,l.e)*dist(l.s,l.e));
}
// 求点C到线段AB所在直线的垂足 P
POINT perpendicular(POINT p,LINESEG l)
{
    double r=relation(p,l);
    POINT tp;
    tp.x=l.s.x+r*(l.e.x-l.s.x);
    tp.y=l.s.y+r*(l.e.y-l.s.y);
    return tp;
}
/* 求点p到线段l的最短距离，并返回线段上距该点最近的点np
注意： np是线段l上到点p最近的点， 不一定是垂足 */
double ptolinesegdist(POINT p,LINESEG l,POINT &np)
{
    double r=relation(p,l);
    if(r<0)
    {
        np=l.s;
        return dist(p,l.s);
    }
    if(r>1)
    {
        np=l.e;
    }
}

```

```

        return dist(p,l.e);
    }
    np=perpendicular(p,l);
    return dist(p,np);
}

// 求点p到线段l所在直线的距离,请注意本函数与上个函数的区别
double ptoldist(POINT p,LINESEG l)
{
    return abs(multiply(p,l.e,l.s))/dist(l.s,l.e);
}

/* 计算点到折线集的最近距离,并返回最近点.
注意: 调用的是ptolineseg()函数 */
double ptopointset(int vcount,POINT pointset[],POINT p,POINT &q)
{
    int i;
    double cd=double(INF),td;
    LINESEG l;
    POINT tq,cq;

    for(i=0;i<vcount-1;i++)
    {
        l.s=pointset[i];

        l.e=pointset[i+1];
        td=ptolinesegdist(p,l,tq);
        if(td<cd)
        {
            cd=td;
            cq=tq;
        }
    }
    q=cq;
    return cd;
}

/* 判断圆是否在多边形内.ptolineseg()函数的应用2 */
bool CircleInsidePolygon(int vcount,POINT center,double radius,POINT polygon[])
{
    POINT q;
    double d;
    q.x=0;
    q.y=0;
    d=ptopointset(vcount,polygon,center,q);
}

```

```

    if (d<radius || fabs(d-radius)<EP)
        return true;
    else
        return false;
}

/* 返回两个矢量l1和l2的夹角的余弦 (-1 --- 1) 注意：如果想从余弦求夹角的话，注意
反余弦函数的定义域是从 0到pi */
double cosine(LINESEG l1,LINESEG l2)
{
    return (((l1.e.x-l1.s.x)*(l2.e.x-l2.s.x) +
            (l1.e.y-l1.s.y)*(l2.e.y-
l2.s.y))/(dist(l1.e,l1.s)*dist(l2.e,l2.s))) ;
}

// 返回线段l1与l2之间的夹角 单位：弧度 范围(-pi, pi)
double lsangle(LINESEG l1,LINESEG l2)
{
    POINT o,s,e;
    o.x=o.y=0;
    s.x=l1.e.x-l1.s.x;
    s.y=l1.e.y-l1.s.y;
    e.x=l2.e.x-l2.s.x;
    e.y=l2.e.y-l2.s.y;
    return angle(o,s,e);
}

// 如果线段u和v相交(包括相交在端点处)时，返回true
//
// 判断P1P2跨立Q1Q2的依据是：( P1 - Q1 ) × ( Q2 - Q1 ) * ( Q2 - Q1 ) × ( P2 - Q1 ) >= 0。
// 判断Q1Q2跨立P1P2的依据是：( Q1 - P1 ) × ( P2 - P1 ) * ( P2 - P1 ) × ( Q2 - P1 ) >= 0。
bool intersect(LINESEG u,LINESEG v)
{
    return( (max(u.s.x,u.e.x)>=min(v.s.x,v.e.x)) &&
//排斥实验
        (max(v.s.x,v.e.x)>=min(u.s.x,u.e.x)) &&
        (max(u.s.y,u.e.y)>=min(v.s.y,v.e.y)) &&
        (max(v.s.y,v.e.y)>=min(u.s.y,u.e.y)) &&
        (multiply(v.s,u.e,u.s)*multiply(u.e,v.e,u.s)>=0) &&
//跨立实验
        (multiply(u.s,v.e,v.s)*multiply(v.e,u.e,v.s)>=0));
}

// (线段u和v相交) && (交点不是双方的端点) 时返回true
bool intersect_A(LINESEG u,LINESEG v)

```

```

{
    return ((intersect(u,v)) &&
            (!online(u,v.s)) &&
            (!online(u,v.e)) &&
            (!online(v,u.e)) &&
            (!online(v,u.s)));
}

// 线段v所在直线与线段u相交时返回true; 方法: 判断线段u是否跨立线段v
bool intersect_l(LINESEG u,LINESEG v)
{
    return multiply(u.s,v.e,v.s)*multiply(v.e,u.e,v.s)>=0;
}

// 根据已知两点坐标, 求过这两点的直线解析方程: a*x+b*y+c = 0 (a >= 0)
LINE makeline(POINT p1,POINT p2)
{
    LINE tl;
    int sign = 1;
    tl.a=p2.y-p1.y;
    if(tl.a<0)
    {
        sign = -1;
        tl.a=sign*tl.a;
    }
    tl.b=sign*(p1.x-p2.x);
    tl.c=sign*(p1.y*p2.x-p1.x*p2.y);
    return tl;
}

// 根据直线解析方程返回直线的斜率k, 水平线返回 0, 坚直线返回 1e200
double slope(LINE l)
{
    if(abs(l.a) < 1e-20)
        return 0;
    if(abs(l.b) < 1e-20)
        return INF;
    return -(l.a/l.b);
}

// 返回直线的倾斜角alpha ( 0 - pi)
double alpha(LINE l)
{
    if(abs(l.a)< EP)
        return 0;
    if(abs(l.b)< EP)
        return PI/2;
}

```

```

double k=slope(l);
if(k>0)
    return atan(k);
else
    return PI+atan(k);
}

// 求点p关于直线l的对称点
POINT symmetry(LINE l,POINT p)
{
    POINT tp;
    tp.x=((l.b*l.b-l.a*l.a)*p.x-2*l.a*l.b*p.y-
2*l.a*l.c)/(l.a*l.a+l.b*l.b);
    tp.y=((l.a*l.a-l.b*l.b)*p.y-2*l.a*l.b*p.x-
2*l.b*l.c)/(l.a*l.a+l.b*l.b);
    return tp;
}

// 如果两条直线 l1(a1*x+b1*y+c1 = 0), l2(a2*x+b2*y+c2 = 0)相交, 返回
true, 且返回交点p
bool lineintersect(LINE l1,LINE l2,POINT &p) // 是 L1, L2
{
    double d=l1.a*l2.b-l2.a*l1.b;
    if(abs(d)<EP) // 不相交
        return false;
    p.x = (l2.c*l1.b-l1.c*l2.b)/d;
    p.y = (l2.a*l1.c-l1.a*l2.c)/d;
    return true;
}

// 如果线段l1和l2相交, 返回true且交点由(inter)返回, 否则返回false
bool intersection(LINESEG l1,LINESEG l2,POINT &inter)
{
    LINE l11,l12;
    l11=makeline(l1.s,l1.e);
    l12=makeline(l2.s,l2.e);
    if(lineintersect(l11,l12,inter))
        return online(l1,inter) && online(l2,inter);
    else
        return false;
}

/*******************\

*          *
* 多边形常用算法模块      *
*          *

```

```
\*****/*
```

// 如果无特别说明，输入多边形顶点要求按逆时针排列

/*
 返回值：输入的多边形是简单多边形，返回true
 要求：输入顶点序列按逆时针排序
 说明：简单多边形定义：
 1：循环排序中相邻线段对的交是他们之间共有的单个点
 2：不相邻的线段不相交
 本程序默认第一个条件已经满足
*/

```
bool issimple(int vcount,POINT polygon[])
{
    int i,cn;
    LINESEG l1,l2;
    for(i=0;i<vcount;i++)
    {
        l1.s=polygon[i];
        l1.e=polygon[(i+1)%vcount];
        cn=vcount-3;
        while(cn)
        {
            l2.s=polygon[(i+2)%vcount];
            l2.e=polygon[(i+3)%vcount];
            if(intersect(l1,l2))
                break;
            cn--;
        }
        if(cn)
            return false;
    }
    return true;
}

// 返回值：按输入顺序返回多边形顶点的凸凹性判断，bc[i]=1,iff:第i个顶点是凸顶点
void checkconvex(int vcount,POINT polygon[],bool bc[])
{
    int i,index=0;
    POINT tp=polygon[0];
    for(i=1;i<vcount;i++) // 寻找第一个凸顶点
    {
        if(polygon[i].y<tp.y||(polygon[i].y ==
tp.y&&polygon[i].x<tp.x))
```

```

    {
        tp=polygon[i];
        index=i;
    }
    int count=vcount-1;
    bc[index]=1;
    while(count) // 判断凸凹性
    {
        if(multiply(polygon[(index+1)%vcount],polygon[(index+2)%vcount],poly
gon[index])>=0 )
            bc[(index+1)%vcount]=1;
        else
            bc[(index+1)%vcount]=0;
        index++;
        count--;
    }
}
// 返回值：多边形polygon是凸多边形时，返回true
bool isconvex(int vcount,POINT polygon[])
{
    bool bc[MAXV];
    checkconvex(vcount,polygon,bc);
    for(int i=0;i<vcount;i++) // 逐一检查顶点，是否全部是凸顶点
        if(!bc[i])
            return false;
    return true;
}
// 返回多边形面积(signed)；输入顶点按逆时针排列时，返回正值；否则返回负值
double area_of_polygon(int vcount,POINT polygon[])
{
    int i;
    double s;
    if (vcount<3)
        return 0;
    s=polygon[0].y*(polygon[vcount-1].x-polygon[1].x);
    for (i=1;i<vcount;i++)
        s+=polygon[i].y*(polygon[(i-1)].x-polygon[(i+1)%vcount].x);
    return s/2;
}
// 如果输入顶点按逆时针排列，返回true
bool isconterclock(int vcount,POINT polygon[])

```

```

{
    return area_of_polygon(vcount,polygon)>0;
}
// 另一种判断多边形顶点排列方向的方法
bool isccwize(int vcount,POINT polygon[])
{
    int i,index;
    POINT a,b,v;
    v=polygon[0];
    index=0;
    for(i=1;i<vcount;i++) // 找到最低且最左顶点，肯定是凸顶点
    {
        if(polygon[i].y<v.y||polygon[i].y == v.y &&
        polygon[i].x<v.x)
        {
            index=i;
        }
    }
    a=polygon[(index-1+vcount)%vcount]; // 顶点v的前一顶点
    b=polygon[(index+1)%vcount]; // 顶点v的后一顶点
    return multiply(v,b,a)>0;
}
*****
***** 射线法判断点q与多边形polygon的位置关系，要求polygon为简单多边形，顶点逆时针排列
如果点在多边形内： 返回0
如果点在多边形边上： 返回1
如果点在多边形外： 返回2
*****
int insidepolygon(int vcount,POINT Polygon[],POINT q)
{
    int c=0,i,n;
    LINESEG l1,l2;
    bool bintersect_a,bonline1,bonline2,bonline3;
    double r1,r2;

    l1.s=q;
    l1.e=q;
    l1.e.x=double(INF);
    n=vcount;
    for (i=0;i<vcount;i++)

```

```

{
    l2.s=Polygon[i];
    l2.e=Polygon[(i+1)%n];
    if(online(l2,q))
        return 1; // 如果点在边上, 返回1
    if ( (bintersect_a=intersect_A(l1,l2))|| // 相交且不在端点
        ( (bonline1=online(l1,Polygon[(i+1)%n]))&& // 第二个端点在
射线上
        ( (! (bonline2=online(l1,Polygon[(i+2)%n])))&& /* 前一个
端点和后一个端点在射线两侧 */
        ((r1=multiply(Polygon[i],Polygon[(i+1)%n],l1.s)*multiply(Polygon[(i+
1)%n],Polygon[(i+2)%n],l1.s))>0) ||
        (bonline3=online(l1,Polygon[(i+2)%n]))&& /* 下一条
边是水平线, 前一个端点和后一个端点在射线两侧 */
        ((r2=multiply(Polygon[i],Polygon[(i+2)%n],l1.s)*multiply(Polygon[(i+
2)%n],
Polygon[(i+3)%n],l1.s))>0)
        )
        )
        ) c++;
}
if(c%2 == 1)
    return 0;
else
    return 2;
}

//点q是凸多边形polygon内时, 返回true; 注意: 多边形polygon一定要是凸多边形
bool InsideConvexPolygon(int vcount,POINT polygon[],POINT q) // 可用于三角形!
{
    POINT p;
    LINESEG l;
    int i;
    p.x=0;p.y=0;
    for(i=0;i<vcount;i++) // 寻找一个肯定在多边形polygon内的点p: 多边形顶
点平均值
    {
        p.x+=polygon[i].x;
        p.y+=polygon[i].y;
    }
}

```

```

    p.x /= vcount;
    p.y /= vcount;

    for(i=0;i<vcount;i++)
    {
        l.s=polygon[i];l.e=polygon[(i+1)%vcount];
        if(multiply(p,l.e,l.s)*multiply(q,l.e,l.s)<0) /* 点p和点q在边
        l的两侧, 说明点q肯定在多边形外 */
            break;
    }
    return (i==vcount);
}

/*****************
寻找凸包的graham 扫描法
PointSet为输入的点集;
ch为输出的凸包上的点集, 按照逆时针方向排列;
n为PointSet中的点的数目
len为输出的凸包上的点的个数
*****************/
void Graham_scan(POINT PointSet[],POINT ch[],int n,int &len)
{
    int i,j,k=0,top=2;
    POINT tmp;
    // 选取PointSet中y坐标最小的点PointSet[k], 如果这样的点有多个, 则取最左
    边的一个
    for(i=1;i<n;i++)
        if ( PointSet[i].y<PointSet[k].y ||
    (PointSet[i].y==PointSet[k].y) && (PointSet[i].x<PointSet[k].x) )
            k=i;
    tmp=PointSet[0];
    PointSet[0]=PointSet[k];
    PointSet[k]=tmp; // 现在PointSet中y坐标最小的点在PointSet[0]
    for (i=1;i<n-1;i++) /* 对顶点按照相对PointSet[0]的极角从小到大进行排
    序, 极角相同的按照距离PointSet[0]从近到远进行排序 */
    {
        k=i;
        for (j=i+1;j<n;j++)
            if ( multiply(PointSet[j],PointSet[k],PointSet[0])>0 ||
// 极角更小
                (multiply(PointSet[j],PointSet[k],PointSet[0])==0) &&
                /* 极角相等, 距离更短 */
                dist(PointSet[0],PointSet[j])
            <dist(PointSet[0],PointSet[k]))

```

```

        )
        k=j;
    tmp=PointSet[i];
    PointSet[i]=PointSet[k];
    PointSet[k]=tmp;
}

ch[0]=PointSet[0];
ch[1]=PointSet[1];
ch[2]=PointSet[2];
for (i=3;i<n;i++)
{
    while (multiply(PointSet[i],ch[top],ch[top-1])>=0)
        top--;
    ch[++top]=PointSet[i];
}
len=top+1;
}

// 卷包裹法求点集凸壳，参数说明同graham算法
void ConvexClosure(POINT PointSet[],POINT ch[],int n,int &len)
{
    int top=0,i,index,first;
    double curmax,curcos,curdis;
    POINT tmp;
    LINESEG l1,l2;
    bool use[MAXV];
    tmp=PointSet[0];
    index=0;
    // 选取y最小点，如果多于一个，则选取最左点
    for(i=1;i<n;i++)
    {
        if(PointSet[i].y<tmp.y||PointSet[i].y ==
tmp.y&&PointSet[i].x<tmp.x)
        {
            index=i;
        }
        use[i]=false;
    }
    tmp=PointSet[index];
    first=index;
    use[index]=true;

    index=-1;
    ch[top++]=tmp;
}

```

```

tmp.x-=100;
l1.s=tmp;
l1.e=ch[0];
l2.s=ch[0];

while(index!=first)
{
    curmax=-100;
    curdis=0;
    // 选取与最后一条确定边夹角最小的点，即余弦值最大者
    for(i=0;i<n;i++)
    {
        if(use[i])continue;
        l2.e=PointSet[i];
        curcos=cosine(l1,l2); // 根据cos值求夹角余弦，范围在 (-1 --
1 )
        if(curcos>curmax || fabs(curcos-curmax)<1e-6 &&
dist(l2.s,l2.e)>curdis)
        {
            curmax=curcos;
            index=i;
            curdis=dist(l2.s,l2.e);
        }
    }
    use[first]=false; //清空第first个顶点标志，使最后能形
成封闭的hull
    use[index]=true;
    ch[top++]=PointSet[index];
    l1.s=ch[top-2];
    l1.e=ch[top-1];
    l2.s=ch[top-1];
}
len=top-1;
}

*****
*****判断线段是否在简单多边形内(注意：如果多边形是凸多边形，下面的算法可以化简)
*****必要条件一：线段的两个端点都在多边形内；
*****必要条件二：线段和多边形的所有边都不交；
*****用途： 1. 判断折线是否在简单多边形内
*****2. 判断简单多边形是否在另一个简单多边形内
***** /

```

```

bool LinesegInsidePolygon(int vcount,POINT polygon[],LINESEG l)
{
    // 判断线段l的端点是否都不在多边形内

    if(!insidepolygon(vcount,polygon,l.s)||!insidepolygon(vcount,polygon
    ,l.e))
        return false;
    int top=0,i,j;
    POINT PointSet[MAXV],tmp;
    LINESEG s;

    for(i=0;i<vcount;i++)
    {
        s.s=polygon[i];
        s.e=polygon[(i+1)%vcount];
        if(online(s,l.s)) //线段l的起始端点在线段s上
            PointSet[top++]=l.s;
        else if(online(s,l.e)) //线段l的终止端点在线段s上
            PointSet[top++]=l.e;
        else
        {
            if(online(l,s.s)) //线段s的起始端点在线段l上
                PointSet[top++]=s.s;
            else if(online(l,s.e)) // 线段s的终止端点在线段l上
                PointSet[top++]=s.e;
            else
            {
                if(intersect(l,s)) // 这个时候如果相交，肯定是内交，返回
false
                    return false;
            }
        }
    }

    for(i=0;i<top-1;i++) /* 冒泡排序，x坐标小的排在前面；x坐标相同者，y坐标
小的排在前面 */
    {
        for(j=i+1;j<top;j++)
        {
            if( PointSet[i].x>PointSet[j].x || fabs(PointSet[i].x-
PointSet[j].x)<EP && PointSet[i].y>PointSet[j].y )
            {
                tmp=PointSet[i];

```

```

        PointSet[i]=PointSet[j];
        PointSet[j]=tmp;
    }
}

for(i=0;i<top-1;i++)
{
    tmp.x=(PointSet[i].x+PointSet[i+1].x)/2; //得到两个相邻交点的中
点
    tmp.y=(PointSet[i].y+PointSet[i+1].y)/2;
    if(!insidepolygon(vcount,polygon,tmp))
        return false;
}
return true;
}
*****
*****求任意简单多边形polygon的重心
需要调用下面几个函数:
void AddPosPart(); 增加右边区域的面积
void AddNegPart(); 增加左边区域的面积
void AddRegion(); 增加区域面积
在使用该程序时, 如果把xtr,ytr,wtr,xtl,ytl,wtl设成全局变量就可以使这些函数的形式得到化简,
但要注意函数的声明和调用要做相应变化
*****
```

```

*****/
void AddPosPart(double x, double y, double w, double &xtr, double
&ytr, double &wtr)
{
    if (abs(wtr + w)<1e-10 ) return; // detect zero regions
    xtr = ( wtr*xtr + w*x ) / ( wtr + w );
    ytr = ( wtr*ytr + w*y ) / ( wtr + w );
    wtr = w + wtr;
    return;
}
void AddNegPart(double x, double y, double w, double &xtl, double
&yt1, double &wt1)
{
    if ( abs(wt1 + w)<1e-10 )
        return; // detect zero regions
```

```

    xtl = ( wtl*xtl + w*x ) / ( wtl + w );
    ytl = ( wtl*ytl + w*y ) / ( wtl + w );
    wtl = w + wtl;
    return;
}

void AddRegion ( double x1, double y1, double x2, double y2, double
&xtr, double &ytr,
                 double &wtr, double &xtl, double &ytl, double &wtl )
{
    if ( abs (x1 - x2) < 1e-10 )
        return;

    if ( x2 > x1 )
    {
        AddPosPart ((x2+x1)/2, y1/2, (x2-x1) * y1,xtr,ytr,wtr); /* rectangle 全局变量变化处 */
        AddPosPart ((x1+x2+x2)/3, (y1+y1+y2)/3, (x2-x1)*(y2-
y1)/2,xtr,ytr,wtr);
        // triangle 全局变量变化处
    }
    else
    {
        AddNegPart ((x2+x1)/2, y1/2, (x2-x1) * y1,xtl,ytl,wtl);
        // rectangle 全局变量变化处
        AddNegPart ((x1+x2+x2)/3, (y1+y1+y2)/3, (x2-x1)*(y2-
y1)/2,xtl,ytl,wtl);
        // triangle 全局变量变化处
    }
}

POINT cg_simple(int vcount,POINT polygon[])
{
    double xtr,ytr,wtr,xtl,ytl,wtl;
    //注意： 如果把xtr,ytr,wtr,xtl,ytl,wtl改成全局变量后这里要删去
    POINT p1,p2,tp;
    xtr = ytr = wtr = 0.0;
    xtl = ytl = wtl = 0.0;
    for(int i=0;i<vcount;i++)
    {
        p1=polygon[i];
        p2=polygon[(i+1)%vcount];
        AddRegion(p1.x,p1.y,p2.x,p2.y,xtr,ytr,wtr,xtl,ytl,wtl); //全局变量变化处
    }
}

```

```

    tp.x = (wtr*xtr + wtl*xtl) / (wtr + wtl);
    tp.y = (wtr*ytr + wtl*ytl) / (wtr + wtl);
    return tp;
}

// 求凸多边形的重心, 要求输入多边形按逆时针排序
POINT gravitycenter(int vcount,POINT polygon[])
{
    POINT tp;
    double x,y,s,x0,y0,cs,k;
    x=0;y=0;s=0;
    for(int i=1;i<vcount-1;i++)
    {
        x0=(polygon[0].x+polygon[i].x+polygon[i+1].x)/3;
        y0=(polygon[0].y+polygon[i].y+polygon[i+1].y)/3; //求当前三角
        形的重心
        cs=multiply(polygon[i],polygon[i+1],polygon[0])/2;
        //三角形面积可以直接利用该公式求解
        if(abs(s)<1e-20)
        {
            x=x0;y=y0;s+=cs;continue;
        }
        k=cs/s; //求面积比例
        x=(x+k*x0)/(1+k);
        y=(y+k*y0)/(1+k);
        s += cs;
    }
    tp.x=x;
    tp.y=y;
    return tp;
}

```

给定一简单多边形, 找出一个肯定在该多边形内的点

定理1 : 每个多边形至少有一个凸顶点

定理2 : 顶点数 ≥ 4 的简单多边形至少有一条对角线

结论 : x 坐标最大, 最小的点肯定是凸顶点

y 坐标最大, 最小的点肯定是凸顶点

```

POINT a_point_insidepoly(int vcount,POINT polygon[])
{
    POINT v,a,b,r;
    int i,index;
    v=polygon[0];

```

```

index=0;
for(i=1;i<vcount;i++) //寻找一个凸顶点
{
    if(polygon[i].y<v.y)
    {
        v=polygon[i];
        index=i;
    }
}
a=polygon[(index-1+vcount)%vcount]; //得到v的前一个顶点
b=polygon[(index+1)%vcount]; //得到v的后一个顶点
POINT tri[3],q;
tri[0]=a;tri[1]=v;tri[2]=b;
double md=INF;
int in1=index;
bool bin=false;
for(i=0;i<vcount;i++) //寻找在三角形avb内且离顶点v最近的顶点q
{
    if(i == index)continue;
    if(i == (index-1+vcount)%vcount)continue;
    if(i == (index+1)%vcount)continue;
    if(!InsideConvexPolygon(3,tri,polygon[i]))continue;
    bin=true;
    if(dist(v,polygon[i])<md)
    {
        q=polygon[i];
        md=dist(v,q);
    }
}
if(!bin) //没有顶点在三角形avb内, 返回线段ab中点
{
    r.x=(a.x+b.x)/2;
    r.y=(a.y+b.y)/2;
    return r;
}
r.x=(v.x+q.x)/2; //返回线段vq的中点
r.y=(v.y+q.y)/2;
return r;
}
*****
*****求从多边形外一点p出发到一个简单多边形的切线,如果存在返回切点,其中rp点是右切点,lp是左切点

```

注意： p点一定要在多边形外，输入顶点序列是逆时针排列

原 理： 如果点在多边形内肯定无切线；凸多边形有唯一的两个切点，凹多边形就可能有多于两个的切点；

如果polygon是凸多边形，切点只有两个只要找到就可以，可以化简此算法

如果是凹多边形还有一种算法可以求解：先求凹多边形的凸包，然后求凸包的切线

```
*****
*****
void pointtangentpoly(int vcount,POINT polygon[],POINT p,POINT
&rp,POINT &lp)
{
    LINESEG ep,en;
    bool blp,bln;
    rp=polygon[0];
    lp=polygon[0];
    for(int i=1;i<vcount;i++)
    {
        ep.s=polygon[(i+vcount-1)%vcount];
        ep.e=polygon[i];
        en.s=polygon[i];
        en.e=polygon[(i+1)%vcount];
        blp=multiply(ep.e,p,ep.s)>=0;           // p is to the
left of pre edge
        bln=multiply(en.e,p,en.s)>=0;           // p is to the
left of next edge
        if(!blp&&bln)
        {
            if(multiply(polygon[i],rp,p)>0)      // polygon[i]
is above rp
                rp=polygon[i];
        }
        if(blp&&!bln)
        {
            if(multiply(lp,polygon[i],p)>0)      // polygon[i]
is below lp
                lp=polygon[i];
        }
    }
    return ;
}
// 如果多边形polygon的核存在，返回true，返回核上的一点p.顶点按逆时针方向输入
bool core_exist(int vcount,POINT polygon[],POINT &p)
{
    int i,j,k;
```

```

LINESEG l;
LINE lineset[MAXV];
for(i=0;i<vcount;i++)
{
    lineset[i]=makeline(polygon[i],polygon[(i+1)%vcount]);
}
for(i=0;i<vcount;i++)
{
    for(j=0;j<vcount;j++)
    {
        if(i == j) continue;
        if(lineintersect(lineset[i],lineset[j],p))
        {
            for(k=0;k<vcount;k++)
            {
                l.s=polygon[k];
                l.e=polygon[(k+1)%vcount];
                if(multiply(p,l.e,l.s)>0)
                    //多边形顶点按逆时针方向排列，核肯定在每条边的左侧或
边
                    //找到了一个核上的点
                    break;
            }
            if(k == vcount)
                break;
        }
        if(j<vcount) break;
    }
    if(i<vcount)
        return true;
    else
        return false;
}
/****************************************\
*          *
* 圆的基本运算          *
*          *
\****************************************/
/*********************************************
*****
```

返回值：点 p 在圆内(包括边界)时，返回true

用途：因为圆为凸集，所以判断点集，折线，多边形是否在圆内时，只需要逐一判断点是否在圆内即可。

```
*****
*/
bool point_in_circle(POINT o, double r, POINT p)
{
    double d2=(p.x-o.x)*(p.x-o.x)+(p.y-o.y)*(p.y-o.y);
    double r2=r*r;
    return d2<r2||abs(d2-r2)<EP;
}
/*****
用 途 : 求不共线的三点确定一个圆
输入 : 三个点p1,p2,p3
返回值 : 如果三点共线, 返回false; 反之, 返回true。圆心由q返回, 半径由r返回
*****
*/
bool cocircle(POINT p1,POINT p2,POINT p3,POINT &q,double &r)
{
    double x12=p2.x-p1.x;
    double y12=p2.y-p1.y;
    double x13=p3.x-p1.x;
    double y13=p3.y-p1.y;
    double z2=x12*(p1.x+p2.x)+y12*(p1.y+p2.y);
    double z3=x13*(p1.x+p3.x)+y13*(p1.y+p3.y);
    double d=2.0*(x12*(p3.y-p2.y)-y12*(p3.x-p2.x));
    if(abs(d)<EP) //共线, 圆不存在
        return false;
    q.x=(y13*z2-y12*z3)/d;
    q.y=(x12*z3-x13*z2)/d;
    r=dist(p1,q);
    return true;
}
int line_circle(LINE l,POINT o,double r,POINT &p1,POINT &p2)
{
    return true;
}

/*
*      *
* 矩形的基本运算      *
*      *
\*****\*/
/*
说明: 因为矩形的特殊性, 常用算法可以化简:
```

1. 判断矩形是否包含点

只要判断该点的横坐标和纵坐标是否夹在矩形的左右边和上下边之间。

2. 判断线段、折线、多边形是否在矩形中

因为矩形是个凸集，所以只要判断所有端点是否都在矩形中就可以了。

3. 判断圆是否在矩形中

圆在矩形中的充要条件是：圆心在矩形中且圆的半径小于等于圆心到矩形四边的距离的最小值。

*/

// 已知矩形的三个顶点 (a, b, c)，计算第四个顶点d的坐标。注意：已知的三个顶点可以是无序的

```
POINT rect4th(POINT a, POINT b, POINT c)
{
    POINT d;
    if (abs(dotmultiply(a, b, c)) < EP) // 说明c点是直角拐角处
    {
        d.x = a.x + b.x - c.x;
        d.y = a.y + b.y - c.y;
    }
    if (abs(dotmultiply(a, c, b)) < EP) // 说明b点是直角拐角处
    {
        d.x = a.x + c.x - b.x;
        d.y = a.y + c.y - b.y;
    }
    if (abs(dotmultiply(c, b, a)) < EP) // 说明a点是直角拐角处
    {
        d.x = c.x + b.x - a.x;
        d.y = c.y + b.y - a.y;
    }
    return d;
}
```

```
/*
 * 常用算法的描述
 */
/* 尚未实现的算法：
1. 求包含点集的最小圆
2. 求多边形的交
3. 简单多边形的三角剖分
4. 寻找包含点集的最小矩形
5. 折线的化简
*/
```

- 6. 判断矩形是否在矩形中
 - 7. 判断矩形能否放在矩形中
 - 8. 矩形并的面积与周长
 - 9. 矩形并的轮廓
 - 10. 矩形并的闭包
 - 11. 矩形的交
 - 12. 点集中的最近点对
 - 13. 多边形的并
 - 14. 圆的交与并
 - 15. 直线与圆的关系
 - 16. 线段与圆的关系
 - 17. 求多边形的核监视摄像机
 - 18. 求点集中不相交点对 railwai
- *//*

寻找包含点集的最小矩形

原理: 该矩形至少一条边与点集的凸壳的某条边共线

First take the convex hull of the points. Let the resulting convex

polygon be P. It has been known for some time that the minimum area rectangle enclosing P must have one rectangle side flush with

(i.e., collinear with and overlapping) one edge of P. This geometric

fact was used by Godfried Toussaint to develop the "rotating calipers"

algorithm in a hard-to-find 1983 paper, "Solving Geometric Problems

with the Rotating Calipers" (Proc. IEEE MELECON). The algorithm rotates a surrounding rectangle from one flush edge to the next, keeping track of the minimum area for each edge. It achieves $O(n)$

time (after hull computation). See the "Rotating Calipers Homepage"

<http://www.cs.mcgill.ca/~orm/rotcal.frame.html> for a description and applet.

//

折线的化简 伪码如下:

Input: tol = the approximation tolerance
 $L = \{V_0, V_1, \dots, V_{n-1}\}$ is any n-vertex polyline

```
Set start = 0;
Set k = 0;
Set w0 = v0;
```

```

for each vertex Vi (i=1,n-1)
{
    if Vi is within tol from Vstart
        then ignore it, and continue with the next vertex

    Vi is further than tol away from Vstart
    so add it as a new vertex of the reduced polyline
    Increment k++;
    Set Wk = Vi;
    Set start = i; as the new initial vertex
}

Output: W = {W0,W1,,Wk-1} = the k-vertex simplified polyline
*/
/******\

*      *
* 补充      *
*      *
\*****/

```

//两圆关系:

```

/* 两圆:
相离: return 1;
外切: return 2;
相交: return 3;
内切: return 4;
内含: return 5;
*/
int CircleRelation(POINT p1, double r1, POINT p2, double r2)
{
    double d = sqrt( (p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y)
);

    if( fabs(d-r1-r2) < EP ) // 必须保证前两个if先被判定!
        return 2;
    if( fabs(d-fabs(r1-r2)) < EP )
        return 4;
    if( d > r1+r2 )
        return 1;
    if( d < fabs(r1-r2) )
        return 5;
    if( fabs(r1-r2) < d && d < r1+r2 )
        return 3;
}

```

```

        return 0; // indicate an error!
    }

//判断圆是否在矩形内:
// 判定圆是否在矩形内, 是就返回true (设矩形水平, 且其四个顶点由左上开始按顺时针
排列)
// 调用ptoldist函数, 在第4页
bool CircleRecRelation(POINT pc, double r, POINT pr1, POINT pr2,
POINT pr3, POINT pr4)
{
    if( pr1.x < pc.x && pc.x < pr2.x && pr3.y < pc.y && pc.y < pr2.y
)
    {
        LINESEG line1(pr1, pr2);
        LINESEG line2(pr2, pr3);
        LINESEG line3(pr3, pr4);
        LINESEG line4(pr4, pr1);
        if( r<ptoldist(pc,line1) && r<ptoldist(pc,line2) &&
r<ptoldist(pc,line3) && r<ptoldist(pc,line4) )
            return true;
    }
    return false;
}

//点到平面的距离:
//点到平面的距离, 平面用一般式表示ax+by+cz+d=0
double P2planeDist(double x, double y, double z, double a, double b,
double c, double d)
{
    return fabs(a*x+b*y+c*z+d) / sqrt(a*a+b*b+c*c);
}

//点是否在直线同侧:
//两个点是否在直线同侧, 是则返回true
bool SameSide(POINT p1, POINT p2, LINE line)
{
    return (line.a * p1.x + line.b * p1.y + line.c) *
    (line.a * p2.x + line.b * p2.y + line.c) > 0;
}

//镜面反射线:
// 已知入射线、镜面, 求反射线。
// a1,b1,c1为镜面直线方程( $a_1 x + b_1 y + c_1 = 0$ , 下同)系数;
// a2,b2,c2为入射光直线方程系数;
// a,b,c为反射光直线方程系数.
// 光是有方向的, 使用时注意: 入射光向量: $\langle -b_2, a_2 \rangle$ ; 反射光向量: $\langle b, -a \rangle$ .
// 不要忘记结果中可能会有"negative zeros"

```

```

void reflect(double a1,double b1,double c1,double a2,double
b2,double c2,double &a,double &b,double &c)
{
    double n,m;
    double tpb,tpa;
    tpb=b1*b2+a1*a2;
    tpa=a2*b1-a1*b2;
    m=(tpb*b1+tpa*a1)/(b1*b1+a1*a1);
    n=(tpa*b1-tpb*a1)/(b1*b1+a1*a1);
    if(fabs(a1*b2-a2*b1)<1e-20)
    {
        a=a2;b=b2;c=c2;
        return;
    }
    double xx,yy; // (xx,yy) 是入射线与镜面的交点。
    xx=(b1*c2-b2*c1)/(a1*b2-a2*b1);
    yy=(a2*c1-a1*c2)/(a1*b2-a2*b1);
    a=n;
    b=-m;
    c=m*yy-xx*n;
}
//矩形包含：
// 矩形2 (C, D) 是否在1 (A, B) 内
bool r2inr1(double A,double B,double C,double D)
{
    double X,Y,L,K,DMax;
    if (A < B)
    {
        double tmp = A;
        A = B;
        B = tmp;
    }
    if (C < D)
    {
        double tmp = C;
        C = D;
        D = tmp;
    }
    if (A > C && B > D)           // trivial case
        return true;
    else
        if (D >= B)
            return false;
}

```

```

    else
    {
        X = sqrt(A * A + B * B);           // outer rectangle's
diagonal
        Y = sqrt(C * C + D * D);           // inner rectangle's
diagonal
        if (Y < B) // check for marginal conditions
            return true; // the inner rectangle can freely
rotate inside
        else
            if (Y > X)
                return false;
            else
            {
                L = (B - sqrt(Y * Y - A * A)) / 2;
                K = (A - sqrt(Y * Y - B * B)) / 2;
                DMax = sqrt(L * L + K * K);
                if (D >= DMax)
                    return false;
                else
                    return true;
            }
        }
    }
//两圆交点:
// 两圆已经相交 (相切)
void c2point(POINT p1,double r1,POINT p2,double r2,POINT &rp1,POINT
&rp2)
{
    double a,b,r;
    a=p2.x-p1.x;
    b=p2.y-p1.y;
    r=(a*a+b*b+r1*r1-r2*r2)/2;
    if(a==0&&b!=0)
    {
        rp1.y=rp2.y=r/b;
        rp1.x=sqrt(r1*r1-rp1.y*rp1.y);
        rp2.x=-rp1.x;
    }
    else if(a!=0&&b==0)
    {
        rp1.x=rp2.x=r/a;
        rp1.y=sqrt(r1*r1-rp1.x*rp2.x);
    }
}

```

```

    rp2.y=-rp1.y;
}
else if(a!=0&&b!=0)
{
    double delta;
    delta=b*b*r*r-(a*a+b*b)*(r*r-r1*r1*a*a);
    rp1.y=(b*r+sqrt(delta))/(a*a+b*b);
    rp2.y=(b*r-sqrt(delta))/(a*a+b*b);
    rp1.x=(r-b*rp1.y)/a;
    rp2.x=(r-b*rp2.y)/a;
}

rp1.x+=p1.x;
rp1.y+=p1.y;
rp2.x+=p1.x;
rp2.y+=p1.y;
}

//两圆公共面积:
// 必须保证相交
double c2area(POINT p1,double r1,POINT p2,double r2)
{
    POINT rp1, rp2;
    c2point(p1,r1,p2,r2,rp1, rp2);

    if(r1>r2) //保证r2>r1
    {
        swap(p1,p2);
        swap(r1,r2);
    }
    double a,b,rr;
    a=p1.x-p2.x;
    b=p1.y-p2.y;
    rr=sqrt(a*a+b*b);

    double dx1,dy1,dx2,dy2;
    double sita1,sita2;
    dx1=rp1.x-p1.x;
    dy1=rp1.y-p1.y;
    dx2=rp2.x-p1.x;
    dy2=rp2.y-p1.y;
    sita1=acos((dx1*dx2+dy1*dy2)/r1/r1);

    dx1=rp1.x-p2.x;
}

```

```

dy1=rp1.y-p2.y;
dx2=rp2.x-p2.x;
dy2=rp2.y-p2.y;
sita2=acos((dx1*dx2+dy1*dy2)/r2/r2);
double s=0;
if(rr<r2)//相交弧为优弧
    s=r1*r1*(PI-sita1/2+sin(sita1)/2)+r2*r2*(sita2-
sin(sita2))/2;
else//相交弧为劣弧
    s=(r1*r1*(sita1-sin(sita1))+r2*r2*(sita2-sin(sita2)))/2;

return s;
}

//圆和直线关系:
//0----相离 1----相切 2----相交
int clpoint(POINT p,double r,double a,double b,POINT c,POINT &rp1,POINT &rp2)
{
    int res=0;

    c=c+a*p.x+b*p.y;
    double tmp;
    if(a==0&&b!=0)
    {
        tmp=-c/b;
        if(r*r<tmp*tmp)
            res=0;
        else if(r*r==tmp*tmp)
        {
            res=1;
            rp1.y=tmp;
            rp1.x=0;
        }
        else
        {
            res=2;
            rp1.y=rp2.y=tmp;
            rp1.x=sqrt(r*r-tmp*tmp);
            rp2.x=-rp1.x;
        }
    }
    else if(a!=0&&b==0)
    {

```

```

tmp=-c/a;
if(r*r<tmp*tmp)
    res=0;
else if(r*r==tmp*tmp)
{
    res=1;
    rp1.x=tmp;
    rp1.y=0;
}
else
{
    res=2;
    rp1.x=rp2.x=tmp;
    rp1.y=sqrt(r*r-tmp*tmp);
    rp2.y=-rp1.y;
}
else if(a!=0&&b!=0)
{
    double delta;
    delta=b*b*c*c-(a*a+b*b)*(c*c-a*a*r*r);
    if(delta<0)
        res=0;
    else if(delta==0)
    {
        res=1;
        rp1.y=-b*c/(a*a+b*b);
        rp1.x=(-c-b*rp1.y)/a;
    }
    else
    {
        res=2;
        rp1.y=(-b*c+sqrt(delta))/(a*a+b*b);
        rp2.y=(-b*c-sqrt(delta))/(a*a+b*b);
        rp1.x=(-c-b*rp1.y)/a;
        rp2.x=(-c-b*rp2.y)/a;
    }
}
rp1.x+=p.x;
rp1.y+=p.y;
rp2.x+=p.x;
rp2.y+=p.y;
return res;

```

```

}

//内切圆:

void incircle(POINT p1,POINT p2,POINT p3,POINT &rp,double &r)
{
    double dx31,dy31,dx21,dy21,d31,d21,a1,b1,c1;
    dx31=p3.x-p1.x;
    dy31=p3.y-p1.y;
    dx21=p2.x-p1.x;
    dy21=p2.y-p1.y;

    d31=sqrt(dx31*dx31+dy31*dy31);
    d21=sqrt(dx21*dx21+dy21*dy21);
    a1=dx31*d21-dx21*d31;
    b1=dy31*d21-dy21*d31;
    c1=a1*p1.x+b1*p1.y;

    double dx32,dy32,dx12,dy12,d32,d12,a2,b2,c2;
    dx32=p3.x-p2.x;
    dy32=p3.y-p2.y;
    dx12=-dx21;
    dy12=-dy21;

    d32=sqrt(dx32*dx32+dy32*dy32);
    d12=d21;
    a2=dx12*d32-dx32*d12;
    b2=dy12*d32-dy32*d12;
    c2=a2*p2.x+b2*p2.y;

    rp.x=(c1*b2-c2*b1)/(a1*b2-a2*b1);
    rp.y=(c2*a1-c1*a2)/(a1*b2-a2*b1);
    r=fabs(dy21*rp.x-dx21*rp.y+dx21*p1.y-dy21*p1.x)/d21;
}

//求切点:

// p---圆心坐标, r---圆半径, sp---圆外一点, rp1, rp2---切点坐标
void cutpoint(POINT p,double r,POINT sp,POINT &rp1,POINT &rp2)
{
    POINT p2;
    p2.x=(p.x+sp.x)/2;
    p2.y=(p.y+sp.y)/2;

    double dx2,dy2,r2;
    dx2=p2.x-p.x;
    dy2=p2.y-p.y;

```

```

r2=sqrt(dx2*dx2+dy2*dy2);
c2point(p,r,p2,r2,rp1,rp2);
}

//线段的左右旋:
/* 12在l1的左/右方向 (l1为基准线)
返回 0 : 重合;
返回 1 : 右旋;
返回 -1 : 左旋;
*/
int rotat(LINESEG l1,LINESEG l2)
{
    double dx1,dx2,dy1,dy2;
    dx1=l1.s.x-l1.e.x;
    dy1=l1.s.y-l1.e.y;
    dx2=l2.s.x-l2.e.x;
    dy2=l2.s.y-l2.e.y;

    double d;
    d=dx1*dy2-dx2*dy1;
    if(d==0)
        return 0;
    else if(d>0)
        return -1;
    else
        return 1;
}
/*
公式:

```

球坐标公式:

直角坐标为 $P(x, y, z)$ 时, 对应的球坐标是 $(r \sin\varphi \cos\theta, r \sin\varphi \sin\theta, r \cos\varphi)$, 其中 φ 是向量 OP 与 z 轴的夹角, 范围 $[0, \pi]$; θ 是 OP 在 XOY 面上的投影到 x 轴的旋角, 范围 $[0, 2\pi]$

直线的一般方程转化成向量方程:

$$\frac{ax+by+c}{m} = \frac{y-y_0}{n} // (x_0, y_0) 为直线上一点, m, n 为向量$$

转换关系:

$$\begin{aligned} a &= n; b = -m; c = m \cdot y_0 - n \cdot x_0; \\ m &= -b; n = a; \end{aligned}$$

三点平面方程：

三点为 P_1, P_2, P_3

设向量 $M_1 = P_2 - P_1; M_2 = P_3 - P_1;$

平面法向量： $M = M_1 \times M_2$ ()

平面方程： $M.i(x - P_1.x) + M.j(y - P_1.y) + M.k(z - P_1.z) = 0$

STL

Vector

以 `vector<int>v;` 为例

1. `v.push_back(a);`
2. `v.clear();` 无法真正释放空间，可能会导致MLE。推荐使用 `vector<int>().swap(v);`
3. `v.erase();`
 - `v.erase(it);` 或 `v.erase(v.begin() + 2);`
 - `b.erase(b.begin(), b.begin() + 3);` // 将(b.begin(), b.begin() + 3)之间的元素删除
4. `v.size();` 返回 `v` 的元素个数
5. `v.back();` 返回最后一个元素的引用
6. `v.front();` 返回第一个元素的引用
7. `vector<int> myvector(8, 10);` // myvector: 10 10 10 10 10 10 10 10
8. `fill_n(myvector.begin(), 4, 20);` // myvector: 20 20 20 20 10 10 10 10
9. 存储方式，下标范围 `[0, v.size() - 1]`
10. `v.empty();`
11. `v.insert();`

List

Map

Bitset

Stack

Queue

Priority_queue

String

Algorithm

sscanf

sprintf

逆序数

RMQ问题

To Do List

2016.8.11

1. [这个博客](#)有一些平时不常用的位运算知识，值得收纳
2. 发现一个野生的[ACM模版](#)，有空研究抄袭一下
3. 发现另一个野生的[ACM模版](#)，有空研究抄袭一下
4. 还有区间DP要学

2016.8.10

1. 二维线段树还没有加
2. 范神的DP模版里有数位DP，回头研究学习一下
3. 计算几何模版内容太多，还需要斟酌

2016.8.10

1. 当前能做的是把这套版子给搞定
2. 组合数部分还没有加入
3. DP还没有细分状压DP、树形DP、数位DP、插头DP、概率DP
4. 容斥原理还没想好怎么加入
5. 莫队算法还没想好要咋搞

To Learn List

DP

1. 区间DP
2. 数位DP
3. 状压DP
4. 树形DP
5. 概率DP

图论

1. ~~最小树形图 朱刘算法~~
2. 二分图KM算法
3. 网络流ISAP
4. 欧拉回路
5. 割顶割桥
6. 双联通分量(点、边)
7. 强联通分量
8. 染色
9. 舞蹈链
10. 2-SAT

数学/数论

1. 质因数分解及pollard_rho大数质因数分解
2. Simpson积分法
3. 指数循环节
4. FFT
5. Romberg积分法

数据结构

1. 二维线段树相关
2. 可持久化线段树
3. ~~树链剖分~~
4. 左偏树
5. 划分树
6. SBT
7. 伸展树Splay

8. 主席树

字符串

1. 回文自动机

其他

1. 莫队算法
2. cdq分治
3. zkw线段树