ICPC World Final

University of Southern California

SCP (Standard Code Paper)

1. Gaussian Elimination

**memset**(used, 0, sizeof used);

for (int i = 1; i <= **min**(n, r); i++){

    for (int j = 1; j <= n; j++){

        if (!used[j] && **sign**(w[j][i])){

            for (int k = 1; k <= r+1; k++)

**swap**(w[j][k], w[i][k]);

            used[j] = true;

**swap**(used[j], used[i]);

            break;

        }

    }

    if (!**sign**(w[i][i])) continue;

    double tmp = w[i][i];

    for (int j = 1; j <= r+1; j++)

        w[i][j] /= tmp;

    for (int j = 1; j <= n; j++){

        if (j != i && **sign**(w[j][i])){

            tmp = w[j][i];

            for (int k = 1; k <= r+1; k++)

                w[j][k] -= tmp \* w[i][k];

        }

    }

}

1. Suffix Array

int sa[maxn], wa[maxn], wb[maxn], cnt[maxn], rank[maxn], height[maxn];

void **getSA**(int \*sz, int len){

    int \*x=wa, \*y=wb, m=27;

    for (int i=1; i<=len; i++)

        cnt[x[i]=sz[i]]++;

    for (int i=1; i<=m; i++)

        cnt[i]+=cnt[i-1];

    for (int i=len; i; i--)

        sa[cnt[x[i]]--]=i;

    for (int h=1; h<=len; h<<=1){

        int pos=0;

        for (int i=len-h+1; i<=len; i++)

            y[++pos]=i;

        for (int i=1; i<=len; i++)

            if (sa[i]>h)

                y[++pos]=sa[i]-h;

**memset**(cnt,0,sizeof cnt);

        for (int i=1; i<=len; i++)

            cnt[x[i]]++;

        for (int i=1; i<=m; i++)

            cnt[i]+=cnt[i-1];

        for (int i=len; i; i--)

            sa[cnt[x[y[i]]]--]=y[i];

**swap**(x, y);

        pos=0;

        x[sa[1]]=++pos;

        for (int i=2; i<=len; i++)

            x[sa[i]]=y[sa[i]]==y[sa[i-1]] &&y[sa[i]+h]==y[sa[i-1]+h]?pos:++pos;

        m=pos;

        if (m==len) break;

    }

}

void **getHeight**(int \*sz, int len){

    for (int i=1; i<=len; i++)

        rank[sa[i]]=i;

    int k=0;

    for (int i=1; i<=len; i++){

        if (k) k--;

        int p=sa[rank[i]-1];

        while (sz[p+k]==sz[i+k]) k++;

        height[rank[i]]=k;

    }

}

1. Suffix Automaton

struct **Node**{

    Node \*par, \*go[26];

    int val;

};

Node sam[maxn<<1];

int sn=1;

Node \*last=&sam[1], \*root=&sam[1];

void **extend**(int c){

    Node \*np=&sam[++sn];

    pos[sn]=c;

    Node \*p=last;

    np->val=p->val+1;

    while (p&&!p->go[c])

        p->go[c]=np, p=p->par;

    if (!p)

        np->par=root;

    else{

        Node \*q=p->go[c];

        if (q->val==p->val+1)

            np->par=q;

        else{

            Node \*nq=&sam[++sn];

            pos[sn]=c;

**memcpy**(nq->go,q->go,sizeof q->go);

            nq->val=p->val+1;

            nq->par=q->par;

            q->par=nq;

            np->par=nq;

            while (p&&p->go[c]==q)

                p->go[c]=nq, p=p->par;

        }

    }

    last=np;

}

1. Cost Flow

struct **Edge**{

    int f,t,w,flow,n;

};

int n, k;

namespace **costflow**{

    int nn=1, head[maxn];

    Edge edge[maxn\*maxn];

    void **addedge**(int f, int t, int flow, int w){

        nn++;

        edge[nn] = **Edge**{f, t, w, flow, head[f]};

        head[f]=nn;

    }

    void **link**(int f, int t, int flow, int w){

**addedge**(f,t,0,w);

**addedge**(t,f,flow,-w);

    }

    queue<int> q;

    int dist[maxn], pre[maxn\*maxn], S, T;

    bool inq[maxn];

    bool **spfa**(){

**memset**(dist,-1,sizeof(dist));

        dist[S]=0;

        q.**push**(S);

        while (!q.**empty**()){

            int nowp=q.**front**();

            q.**pop**();

            inq[nowp]=0;

            for (int e=head[nowp]; e; e=edge[e].n){

                if (!edge[e^1].flow) continue;

                if (dist[edge[e].t]==-1 ||dist[nowp]+edge[e].w<dist[edge[e].t]){

                    dist[edge[e].t]=dist[nowp]+edge[e].w;

                    pre[edge[e].t]=e;

                    if (!inq[edge[e].t])

                    {

                        inq[edge[e].t]=1;

                        q.**push**(edge[e].t);

                    }

                }

            }

        }

        return dist[T]!=-1;

    }

    int cost;

    int **costflow**(){

        cost=0;

        while (**spfa**()){

            int nowp=T;

            cost+=dist[T];

            while (nowp!=S){

                edge[pre[nowp]].flow+=1;

                edge[pre[nowp]^1].flow-=1;

                nowp=edge[pre[nowp]].f;

            }

        }

        return cost;

    }

}

1. Link Cut Tree

struct **LCTNode**{

    LCTNode \*fa, \*ch[2];

    bool tag;

    void **gettag**(){

        tag^=1;

    }

    void **pushdown**(){

        if (tag){

            ch[1]->**gettag**();

            ch[0]->**gettag**();

**swap**(ch[0],ch[1]);

            tag=0;

        }

    }

    void **update**()

    {

    }

} rnil;

LCTNode \*nil=&rnil;

int n;

namespace **LCT**{

    LCTNode tree[maxn];

    void **init**(LCTNode &now){

        now.ch[0]=now.ch[1]=now.fa=nil;

    }

    void **init**(){

        for (int i=1; i<=n; i++)

**init**(tree[i]);

    }

    bool **isRoot**(LCTNode \*now){

        return now->fa->ch[0]!=now&&now->fa->ch[1]!=now;

    }

    void **rotate**(LCTNode \*now){

        LCTNode \*fa=now->fa;

        LCTNode \*gfa=fa->fa;

        bool d=fa->ch[1]==now;

        if (!**isRoot**(fa))

            gfa->ch[gfa->ch[1]==fa]=now;

        now->fa=gfa;

        fa->fa=now;

        fa->ch[d]=now->ch[d^1];

        now->ch[d^1]->fa=fa;

        now->ch[d^1]=fa;

        fa->**update**();

        now->**update**();

    }

    stack<LCTNode\*> sta;

    void **splay**(LCTNode \*now){

        LCTNode \*pre=now;

        sta.**push**(pre);

        while (!**isRoot**(pre)){

            pre=pre->fa;

            sta.**push**(pre);

        }

        while (!sta.**empty**()){

            sta.**top**()->**pushdown**();

            sta.**pop**();

        }

        while (!**isRoot**(now)){

            LCTNode \*fa=now->fa, \*gfa=now->fa->fa;

            if (!**isRoot**(fa)){

                if ((gfa->ch[1]==fa)!=(fa->ch[1]==now))

**rotate**(now);

                else

**rotate**(fa);

            }

**rotate**(now);

        }

    }

    void **access**(LCTNode\* nowp){

        for (LCTNode \*son=nil; nowp!=nil; nowp=nowp->fa){

**splay**(nowp);

            nowp->ch[1]=son;

            son=nowp;

        }

    }

    void **makeRoot**(LCTNode\* nowp){

**access**(nowp);

**splay**(nowp);

        nowp->**gettag**();

    }

    void **cut**(LCTNode\* p1, LCTNode\* p2){

**makeRoot**(p1);

**access**(p2);

**splay**(p2);

        p1->fa=nil;

        p2->ch[0]=nil;

    }

    LCTNode\* **find**(LCTNode \*p1){

**access**(p1);

**splay**(p1);

        while (p1->ch[0]!=nil)

            p1=p1->ch[0];

        return p1;

    }

    bool **find**(LCTNode\* p1, LCTNode\* p2){

        return **find**(p1)==**find**(p2);

    }

    void **link**(LCTNode\* p1, LCTNode\* p2){

**makeRoot**(p1);

        p1->fa=p2;

**access**(p1);

    }

}

1. Extended GCD

LL **ex\_gcd**(LL a, LL b, LL &x, LL &y){

    if (b == 0)

    {

        x = 1, y = 0;

        return a;

    } else {

        LL g = **ex\_gcd**(b, a % b, x, y);

        LL t = x;

        x = y, y = t - a / b \* x;

        return g;

    }

}

ax ≡ **b**(mod n), n > 0

void **modularlinearequationsolver**(int a, int b, int n)

{

    int d, x, y, e, i;

    d = **ex\_gcd**(a, n, x, y);

    if (b % d != 0) cout << "No answer !" ;

    else

    {

        e = x ∗(b / d) % n; *// x=e is a basic solution*

        for (i = 0; i < d; i++)

            cout << (e + i ∗(n / d)) % n << endl;

    }

}

Given bi, wi, i = 0...len − 1 which wi > 0, i = 0...len − 1 and (wi, wj) = 1, i != j Find an x which satisfies : x ≡ **bi**(mod wi), i = 0...len − 1

int **china**(int b[], int w[], int len)

{

    int i, d, x, y, x, m, n;

    x = 0;

    n = 1;

    for (i = 0; i < len; i++)

        n∗= w[i];

    for (i = 0; i < len; i++)

    {

        m = n / w[i];

        d = **ex\_gcd**(w[i], m, x, y);

        x = (x + y∗m∗b[i]) % n;

    }

    return (n + x % n) % n;

}

1. Match

while (!q.**empty**())

{

    int p=q.**front**(); q.**pop**();

    for (; pos[p]<v[p].**size**(); pos[p]++)

    {

        int t=v[p][pos[p]];

        if (level[t][match[t]]<level[t][p])

        {

            if (match[t])

                q.**push**(match[t]);

            match[t]=p;

            break;

        }

    }

}

1. Tarjan (Bi)connected Component

void **tarjan**(int nowp){

    dfsn++;

    dfn[nowp]=dfsn;

    low[nowp]=dfsn;

    ins[nowp]=1;

    sta.**push**(nowp);

    for (int e=head[nowp]; e; e=edge[e].n){

        if (!dfn[edge[e].t]){

**tarjan**(edge[e].t);

            low[nowp]=**min**(low[nowp],low[edge[e].t]);

        }

        else if (ins[edge[e].t]){

            low[nowp]=**min**(low[nowp],dfn[edge[e].t]);

        }

    }

    if (low[nowp]==dfn[nowp]){

        if (sta.**top**()!=nowp) tn++;

        while (sta.**top**()!=nowp){

            belong[sta.**top**()]=nowp;

            ins[sta.**top**()]=0;

            sta.**pop**();

        }

        belong[sta.**top**()]=nowp;

        ins[sta.**top**()]=0;

        sta.**pop**();

    }

}

1. FFT

Find the primitive root modulo some prime x: loop over 2 to x-1 and check.

Common modulo numbers for NTT: 998244353(primitive root 3) 100453580(primitive root 3)

struct **Complex**{

    double x, y;

**Complex** (double x=0, double y=0):**x**(x), **y**(y){}

    Complex operator + (const Complex &an) const{

        return **Complex**(x+an.x,y+an.y);

    }

    Complex operator - (const Complex &an) const{

        return **Complex**(x-an.x,y-an.y);

    }

    Complex operator \* (const Complex &an) const{

        return **Complex**(x\*an.x-y\*an.y,x\*an.y+y\*an.x);

    }

    Complex operator / (const double &an) const{

        return **Complex**(x/an,y/an);

    }

};

const int maxn=300100;

const double pi=**acos**(-1);

Complex a[maxn], b[maxn];

int n, m, len, trans[maxn], po;

void **initTrans**(){

    for (int i=0; i<len; i++)

        trans[i]=(trans[i>>1]>>1)|((i&1)<<(po-1));

}

void **FFT**(Complex \*x, int l, int type){

    for (int i=0; i<l; i++){

        if (trans[i]>i)

**swap**(x[i],x[trans[i]]);

    }

    for (int h=1; h<l; h<<=1){

        Complex w0=**Complex**(**cos**(pi/h),**sin**(pi\*type/h));

        for (int i=0; i<l; i+=h<<1){

            Complex w=**Complex**(1,0);

            for (int j=i; j<i+h; j++){

                Complex f=x[j], t=x[j+h]\*w;

                x[j]=f+t;

                x[j+h]=f-t;

                w=w\*w0;

            }

        }

    }

    if (type==-1){

        for (int i=0; i<l; i++)

            x[i]=x[i]/l;

    }

}

1. FWT

void **fmod**(LL &num){

num %= mod;

if (num<0)

num+=mod;

}

void **fwtand**(LL\* arr, int l, int r, int v){

    if ( l == r ) return;

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

**fwtand**(arr, l, m, v);

**fwtand**(arr, m+1, r, v);

for (int i=0; i<=m-l; i++) {

arr[l+i] += arr[m+1+i]\*v;

**fmod**(arr[l+i]);

}

}

void **fwtxor**(LL\* arr, int l, int r, int v){

    if ( l == r ) return;

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

**fwtxor**(arr, l, m, v);

**fwtxor**(arr, m+1, r, v);

for (int i=0; i<=m-l; i++) {

    int f = arr[l+i], t = arr[m+1+i];

arr[l+i] = f+t;

arr[m+i+1] = f-t;

**fmod**(arr[l+i]);

**fmod**(arr[m+i+1]);

if (v == -1){

    arr[l+i]=rev2\*arr[l+i]%mod;

    arr[m+i+1]=rev2\*arr[m+i+1]%mod;

        }

}

}

1. Split Merge Tree

namespace **SMT**{

    struct **Node**{

        int val, size, data;

        LL lmax, rmax, ans, sum;

        bool tag;

        Node \*ch[2];

        void **get**(int num){

            data=sum=num;

            if (num<0)

                lmax=rmax=ans=0;

            else

                lmax=rmax=ans=num;

        }

        void **gettag**(){

            tag^=1;

**swap**(lmax,rmax);

        }

        void **pushdown**(){

            if (tag){

**swap**(ch[0], ch[1]);

                ch[0]->**gettag**();

                ch[1]->**gettag**();

                tag=0;

            }

        }

        void **update**(){

            size=ch[0]->size+ch[1]->size+1;

            lmax=**max**(ch[0]->lmax, ch[0]->sum+data+ch[1]->lmax);

            rmax=**max**(ch[1]->rmax, ch[1]->sum+data+ch[0]->rmax);

            sum=ch[0]->sum+data+ch[1]->sum;

            ans=**max**(ch[0]->ans, ch[1]->ans);

            ans=**max**(ans, ch[0]->rmax+data+ch[1]->lmax);

        }

**Node**();

    } rnil, tree[maxn];

    Node \*nil=&rnil;

    Node \*root=nil;

**Node::Node**() {

        val=**rand**();

        ch[0]=ch[1]=nil;

        if (this==nil)

            size=0;

        else

            size=1;

    }

    Node \***merge**(Node \*l, Node \*r) {

        if (l==nil) return r;

        if (r==nil) return l;

        if (l->val>r->val) {

            l->**pushdown**();

            l->ch[1]=**merge**(l->ch[1], r);

            l->**update**();

            return l;

        } else {

            r->**pushdown**();

            r->ch[0]=**merge**(l, r->ch[0]);

            r->**update**();

            return r;

        }

    }

    void **split**(Node \*now, int rank, Node\* &l, Node\* &r) {

        now->**pushdown**();

        if (!rank) {

            l=nil;

            r=now;

            return;

        }

        Node \*L, \*R;

        if (now->ch[0]->size>=rank) {

**split**(now->ch[0], rank, L, R);

            now->ch[0]=R; l=L; r=now;

        } else {

**split**(now->ch[1], rank-1-now->ch[0]->size, L, R);

            now->ch[1]=L; l=now; r=R;

        }

        now->**update**();

    }

    void **reverse**(int l, int r){

        Node \*L, \*mid, \*R;

**split**(root, l-1, L, mid);

**split**(mid, r-l+1, mid, R);

        mid->**gettag**();

        mid=**merge**(L, mid);

        root=**merge**(mid, R);

    }

    void **modify**(int pos, int val){

        Node \*L, \*mid, \*R;

**split**(root, pos-1, L, mid);

**split**(mid, 1, mid, R);

        mid->**get**(val);

        mid=**merge**(L, mid);

        root=**merge**(mid, R);

    }

    void **init**(){

        for (int i=1; i<=n; i++){

            int tmp;

**scanf**("%d",&tmp);

            tree[i].**get**(tmp);

            root=**merge**(root, &tree[i]);

        }

    }

}

1. Alpha-Beta Search

int dfs(int round, int alpha, int beta){

    bool player=round&1;

    if (endstate)

        return eval();

    int ret=player?inf-round:round-inf;

    for (moves){

        int back=dfs(round+1,alpha,beta);

        if (player){

            beta=min(beta,back);

            ret=min(ret,back);

        } else {

            alpha=max(alpha,back);

            ret=max(ret,back);

        }

        if (alpha>=beta) break;

    }

    return ret;

}

1. Linear Sieve

void linearSieve(){

    notPrime[1]=1;

    phi[1]=1;

    for (int i=2; i<=limit; i++){

        if (!notPrime[i]){

            phi[i]=i-1;

            prime[++pn]=i;

        }

        for (int j=1; j<=pn; j++){

            if (i\*prime[j]>limit) break;

            notPrime[i\*prime[j]]=1;

            if (i%prime[j]==0){

                phi[i\*prime[j]]=phi[i]\*prime[j];

                break;

            }

            phi[i\*prime[j]]=phi[i]\*(prime[j]-1);

        }

    }

}

1. Palindrome Automaton

char s[maxn];

int ch[maxn][30];

int pn, fail[maxn], len[maxn], num[maxn], last, cnt[maxn];

void init(){

    pn=1;

    len[1]=-1;

    fail[0]=fail[1]=1;

}

int getfail(int pos){

    while (s[n]!=s[n-1-len[pos]]) pos=fail[pos];

    return pos;

}

void extend(int c){

    n++;

    int cur=getfail(last);

    if (!ch[cur][c]){

        ++pn;

        len[pn]=len[cur]+2;

        fail[pn]=ch[getfail(fail[cur])][c];

        ch[cur][c]=pn;

        num[pn]=num[fail[pn]]+1;

    }

    last=ch[cur][c];

    cnt[last]++;

}

void count(){

    for (int i=pn; i; i--)

        cnt[fail[i]]+=cnt[i];

}

1. Computational Geometry

int sign(double x){

    if (x<-EPS) return -1;

    if (x>EPS) return 1;

    return 0;

}

struct Vector{

    double x, y;

    void read(){

        scanf("%lf%lf",&x,&y);

    }

    Vector (double x=0, double y=0):x(x), y(y){}

    Vector operator + (const Vector &an){

        return Vector(x+an.x,y+an.y);

    }

    Vector operator - (const Vector &an){

        return Vector(x-an.x,y-an.y);

    }

    Vector operator \* (const double &an){

        return Vector(x\*an,y\*an);

    }

    Vector operator / (const double &an){

        return Vector(x/an,y/an);

    }

    bool operator < (const Vector &an) const{

        if (x!=an.x) return x<an.x;

        return y<an.y;

    }

    void print(){

        printf("(%f,%f)\n",x,y);

    }

};

#define sqr(x) ((double)(x)\*(x))

double cross(const Vector &a, const Vector &b) {

    return a.x\*b.y-a.y\*b.x;

}

// The following code does not use the previous struct; operations are the same, but syntax differs

// excenter of a triangle

void **Circumcenter**(CPoint p0, CPoint p1, CPoint p2, CPoint &cp){

    double a1 = p1.x - p0.x, b1 = p1.y - p0.y, c1 = (**sqr**(a1) + **sqr**(b1)) / 2;

    double a2 = p2.x - p0.x, b2 = p2.y - p0.y, c2 = (**sqr**(a2) + **sqr**(b2)) / 2;

    double d = a1 \* b2 - a2 \* b1;

    cp.x = p0.x + (c1 \* b2 - c2 \* b1) / d;

    cp.y = p0.y + (a1 \* c2 - a2 \* c1) / d;

}

*// distance from a line segment to a point*

double **disSP**(P p1, P p2, P q)

{

    if (p2.**sub**(p1).**dot**(q.**sub**(p1)) < EPS)

        return q.**sub**(p1).**abs**();

    if (p1.**sub**(p2).**dot**(q.**sub**(p2)) < EPS)

        return q.**sub**(p2).**abs**();

    return **disLP**(p1, p2, q);

}

*// distance from a line to a point*

double **disLP**(P p1, P p2, P q)

{

    return **abs**(p2.**sub**(p1).**det**(q.**sub**(p1))) / p2.**sub**(p1).**abs**();

}

*// Check whether two line segments intersect*

bool **crsSS**(P p1, P p2, P q1, P q2)

{

    if (**max**(p1.x, p2.x) + EPS < **min**(q1.x, q2.x))

        return false;

    if (**max**(q1.x, q2.x) + EPS < **min**(p1.x, p2.x))

        return false;

    if (**max**(p1.y, p2.y) + EPS < **min**(q1.y, q2.y))

        return false;

    if (**max**(q1.y, q2.y) + EPS < **min**(p1.y, p2.y))

        return false;

    return **sig**(p2.**sub**(p1).**det**(q1.**sub**(p1))) \* **sig**(p2.**sub**(p1).**det**(q2.**sub**(p1))) < EPS && **sig**(q2.**sub**(q1).**det**(p1.**sub**(q1))) \* **sig**(q2.**sub**(q1).**det**(p2.**sub**(q1))) < EPS;

}

*// Check whether a circle intersects a line segment*

bool **crsCS**(P c, double r, P p1, P p2)

{

    return **disSP**(p1, p2, c) < r + EPS &&

         (r < c.**sub**(p1).**abs**() + EPS || r < c.**sub**(p2).**abs**() + EPS);

}

*// Check whether two circles intersect*

bool **crsCC**(P c1, double r1, P c2, double r2)

{

    double dis = c1.**sub**(c2).**abs**();

    return dis < r1 + r2 + EPS && **abs**(r1 - r2) < dis + EPS;

}

*// Check whether 4 points are on the circumference of a circle.*

bool **onCir**(P p1, P p2, P p3, P p4)

{

    if (**abs**(p2.**sub**(p1).**det**(p3.**sub**(p1))) < EPS)

        return true;

    P c = **ccenter**(p1, p2, p3);

    return **abs**(c.**sub**(p1).**abs2**() - c.**sub**(p4).**abs2**()) < EPS;

}

*// Foot point of a point on a line*

P **proj**(P p1, P p2, P q)

{

    return p1.**add**(p2.**sub**(p1).**mul**(p2.**sub**(p1).**dot**(q.**sub**(p1)) / p2.**sub**(p1).**abs2**()));

}

*// Intersection of two lines*

P **isLL**(P p1, P p2, P q1, P q2)

{

    double d = q2.**sub**(q1).**det**(p2.**sub**(p1));

    if (**abs**(d) < EPS)

        return null;

    return p1.**add**(p2.**sub**(p1).**mul**(q2.**sub**(q1).**det**(q1.**sub**(p1)) / d));

}

*// Intersections of a circle and a line*

P[] **isCL**(P c, double r, P p1, P p2)

{

    double x = p1.**sub**(c).**dot**(p2.**sub**(p1));

    double y = p2.**sub**(p1).**abs2**();

    double d = x \* x - y \* (p1.**sub**(c).**abs2**() - r \* r);

    if (d < -EPS)

        return new P[0];

    if (d < 0)

        d = 0;

    P q1 = p1.**sub**(p2.**sub**(p1).**mul**(x / y));

    P q2 = p2.**sub**(p1).**mul**(**sqrt**(d) / y);

    return new P[]{q1.**sub**(q2), q1.**add**(q2)};

}

*// Intersection of two circles*

P[] **isCC**(P c1, double r1, P c2, double r2)

{

    double x = c1.**sub**(c2).**abs2**();

    double y = ((r1 \* r1 - r2 \* r2) / x + 1) / 2;

    double d = r1 \* r1 / x - y \* y;

    if (d < -EPS)

        return new P[0];

    if (d < 0)

        d = 0;

    P q1 = c1.**add**(c2.**sub**(c1).**mul**(y));

    P q2 = c2.**sub**(c1).**mul**(**sqrt**(d)).**rot90**();

    return new P[]{q1.**sub**(q2), q1.**add**(q2)};

}

*// Tangent line from point p to circle c*

P[] **tanCP**(P c, double r, P p)

{

    double x = p.**sub**(c).**abs2**();

    double d = x - r \* r;

    if (d < -EPS)

        return new P[0];

    if (d < 0)

        d = 0;

    P q1 = p.**sub**(c).**mul**(r \* r / x);

    P q2 = p.**sub**(c).**mul**(-r \* **sqrt**(d) / x).**rot90**();

    return new P[]{c.**add**(q1.**sub**(q2)), c.**add**(q1.**add**(q2))};

}

*// Co-tangents of two circles*

P[][] **tanCC**(P c1, double r1, P c2, double r2)

{

    List<P[]> list = new ArrayList<P[]>();

    if (**abs**(r1 - r2) < EPS)

    {

        P dir = c2.**sub**(c1);

        dir = dir.**mul**(r1 / dir.**abs**()).**rot90**();

        list.**add**(new P[]{c1.**add**(dir), c2.**add**(dir)});

        list.**add**(new P[]{c1.**sub**(dir), c2.**sub**(dir)});

    }

    else

    {

        P p = c1.**mul**(-r2).**add**(c2.**mul**(r1)).**div**(r1 - r2);

        P[] ps = **tanCP**(c1, r1, p);

        P[] qs = **tanCP**(c2, r2, p);

        for (int i = 0; i < ps.length && i < qs.length; i++)

        {

            list.**add**(new P[]{ps[i], qs[i]});

        }

    }

    P p = c1.**mul**(r2).**add**(c2.**mul**(r1)).**div**(r1 + r2);

    P[] ps = **tanCP**(c1, r1, p);

    P[] qs = **tanCP**(c2, r2, p);

    for (int i = 0; i < ps.length && i < qs.length; i++)

    {

        list.**add**(new P[]{ps[i], qs[i]});

    }

    return list.**toArray**(new P[0][]);

}

*// Area of the intersection of two circles*

double **areaCC**(P c1, double r1, P c2, double r2)

{

    double d = c1.**sub**(c2).**abs**();

    if (r1 + r2 < d + EPS)

        return 0;

    if (d < **abs**(r1 - r2) + EPS)

    {

        double r = **min**(r1, r2);

        return r \* r \* PI;

    }

    double x = (d \* d + r1 \* r1 - r2 \* r2) / (2 \* d);

    double t1 = **acos**(x / r1);

    double t2 = **acos**((d - x) / r2);

    return r1 \* r1 \* t1 + r2 \* r2 \* t2 - d \* r1 \* **sin**(t1);

}

*// Area of the intersection of a circle centered at origin and rectangle (O,p1,p2)*

double **areaCT**(double r, P p1, P p2)

{

    P[] qs = **isCL**(O, r, p1, p2);

    if (qs.length == 0)

        return r \* r \* **rad**(p1, p2) / 2;

    bool b1 = p1.**abs**() > r + EPS, b2 = p2.**abs**() > r + EPS;

    if (b1 && b2)

    {

        if (p1.**sub**(qs[0]).**dot**(p2.**sub**(qs[0])) < EPS &&

            p1.**sub**(qs[1]).**dot**(p2.**sub**(qs[1])) < EPS)

            return (r \* r \* (**rad**(p1, p2) - **rad**(qs[0], qs[1])) + qs[0].**det**(qs[1])) / 2;

        else

            return r \* r \* **rad**(p1, p2) / 2;

    }

    else if (b1)

        return (r \* r \* **rad**(p1, qs[0]) + qs[0].**det**(p2)) / 2;

    else if (b2)

        return (r \* r \* **rad**(qs[1], p2) + p1.**det**(qs[1])) / 2;

    else

        return p1.**det**(p2) / 2;

}

// Whether q is inside polygon ps. Inside: 1, On the edge: 0, Outside: -1.

int **contains**(P[] ps, P q)

{

    int n = ps.length;

    int res = -1;

    for (int i = 0; i < n; i++)

    {

        P a = ps[i].**sub**(q), b = ps[(i + 1) % n].**sub**(q);

        if (a.y > b.y)

        {

            P t = a;

            a = b;

            b = t;

        }

        if (a.y < EPS && b.y > EPS && a.**det**(b) > EPS)

        {

            res = -res;

        }

        if (**abs**(a.**det**(b)) < EPS && a.**dot**(b) < EPS)

            return 0;

    }

    return res;

}

// Distance between a convex polygon and a point. It’s a binary search.

// ps should be given in counterclockwise order

double **disConvexP**(P[] ps, P q)

{

    int n = ps.length;

    int left = 0, right = n;

    while (right - left > 1)

    {

        int mid = (left + right) / 2;

        if (**in**(ps[(left + n - 1) % n], ps[left], ps[mid], ps[(mid + 1) % n], q))

        {

            right = mid;

        }

        else

        {

            left = mid;

        }

    }

    return **disSP**(ps[left], ps[right % n], q);

}

bool **in**(P p1, P p2, P p3, P p4, P q)

{

    P o12 = p1.**sub**(p2).**rot90**();

    P o23 = p2.**sub**(p3).**rot90**();

    P o34 = p3.**sub**(p4).**rot90**();

    return **in**(o12, o23, q.**sub**(p2)) || **in**(o23, o34, q.**sub**(p3)) || **in**(o23, p3.**sub**(p2), q.**sub**(p2)) && **in**(p2.**sub**(p3), o23, q.**sub**(p3));

}

bool **in**(P p1, P p2, P q)

{

    return p1.**det**(q) > -EPS && p2.**det**(q) < EPS;

}

1. BigNum

const int maxn = 1000;

struct bign{

    int d[maxn], len;

    void clean() { while(len > 1 && !d[len-1]) len--; }

    bign()       { memset(d, 0, sizeof(d)); len = 1; }

    bign(int num) { \*this = num; }

    bign(char\* num) { \*this = num; }

    bign operator = (const char\* num){

        memset(d, 0, sizeof(d)); len = strlen(num);

        for(int i = 0; i < len; i++) d[i] = num[len-1-i] - '0';

        clean();

        return \*this;

    }

    bign operator = (int num){

        char s[20]; sprintf(s, "%d", num);

        \*this = s;

        return \*this;

    }

    bign operator + (const bign& b){

        bign c = \*this; int i;

        for (i = 0; i < b.len; i++){

            c.d[i] += b.d[i];

            if (c.d[i] > 9) c.d[i]%=10, c.d[i+1]++;

        }

        while (c.d[i] > 9) c.d[i++]%=10, c.d[i]++;

        c.len = max(len, b.len);

        if (c.d[i] && c.len <= i) c.len = i+1;

        return c;

    }

    bign operator - (const bign& b){

        bign c = \*this; int i;

        for (i = 0; i < b.len; i++){

            c.d[i] -= b.d[i];

            if (c.d[i] < 0) c.d[i]+=10, c.d[i+1]--;

        }

        while (c.d[i] < 0) c.d[i++]+=10, c.d[i]--;

        c.clean();

        return c;

    }

    bign operator \* (const bign& b)const{

        int i, j; bign c; c.len = len + b.len;

        for(j = 0; j < b.len; j++) for(i = 0; i < len; i++)

            c.d[i+j] += d[i] \* b.d[j];

        for(i = 0; i < c.len-1; i++)

            c.d[i+1] += c.d[i]/10, c.d[i] %= 10;

        c.clean();

        return c;

    }

    bign operator / (const bign& b){

        int i, j;

        bign c = \*this, a = 0;

        for (i = len - 1; i >= 0; i--)

        {

            a = a\*10 + d[i];

            for (j = 0; j < 10; j++) if (a < b\*(j+1)) break;

            c.d[i] = j;

            a = a - b\*j;

        }

        c.clean();

        return c;

    }

    bign operator % (const bign& b){

        int i, j;

        bign a = 0;

        for (i = len - 1; i >= 0; i--)

        {

            a = a\*10 + d[i];

            for (j = 0; j < 10; j++) if (a < b\*(j+1)) break;

            a = a - b\*j;

        }

        return a;

    }

    bign operator += (const bign& b){

        \*this = \*this + b;

        return \*this;

    }

    bool operator <(const bign& b) const{

        if(len != b.len) return len < b.len;

        for(int i = len-1; i >= 0; i--)

            if(d[i] != b.d[i]) return d[i] < b.d[i];

        return false;

    }

    bool operator >(const bign& b) const{return b < \*this;}

    bool operator<=(const bign& b) const{return !(b < \*this);}

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

    bool operator!=(const bign& b) const{return b < \*this || \*this < b;}

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

    string str() const{

        char s[maxn]={};

        for(int i = 0; i < len; i++) s[len-1-i] = d[i]+'0';

        return s;

    }

};

istream& operator >> (istream& in, bign& x)

{

    string s;

    in >> s;

    x = s.c\_str();

    return in;

}

ostream& operator << (ostream& out, const bign& x)

{

    out << x.str();

    return out;

}

1. Network Flow

struct Edge{

    int f, t, n;

    double flow;

};

Edge edge[maxn\*1000];

int nn=1, head[maxn], cur[maxn], S, T;

void addedge(int f, int t, double flow){

    nn++;

    edge[nn].f=f;

    edge[nn].t=t;

    edge[nn].flow=flow;

    edge[nn].n=head[f];

    head[f]=nn;

}

void link(int f, int t, double flow){

    addedge(f,t,0);

    addedge(t,f,flow);

}

int dep[maxn];

bool bfs(){

queue<int> q;

    memset(dep,0,sizeof(dep));

    q.push(S);

    dep[S]=1;

    while (!q.empty()){

        int nowp=q.front(); q.pop();

        for (int e=head[nowp]; e; e=edge[e].n){

            if (dep[edge[e].t]||!sign(edge[e^1].flow)) continue;

            dep[edge[e].t]=dep[nowp]+1;

            q.push(edge[e].t);

        }

    }

    return dep[T];

}

double dfs(int nowp, double flow){

    if (nowp==T) return flow;

    double used=0;

    for (int &e=cur[nowp]; e; e=edge[e].n){

        if (dep[edge[e].t]!=dep[nowp]+1||!sign(edge[e^1].flow)) continue;

        double tmp1=dfs(edge[e].t,min(flow,edge[e^1].flow));

        edge[e^1].flow-=tmp1;

        edge[e].flow+=tmp1;

        used+=tmp1;

        flow-=tmp1;

        if (!sign(flow))

            return used;

    }

    return used;

}

double dinic(){

    double nowans=0;

    while (bfs()){

        for (int i=1; i<=T; i++)

            cur[i]=head[i];

        nowans+=dfs(S,inf);

    }

    return nowans;

}

void init(){

    memset(head,0,sizeof(head));

    nn=1;

}

1. Chu–Liu/Edmond's algorithm

int **zhuliu**(int root, int V, int **map**[MAXV + 7][MAXV + 7]){

bool visited[MAXV + 7];

bool flag[MAXV + 7];

int pre[MAXV + 7];

int sum = 0;

int i, j, k;

for(i = 0; i <= V; i++) flag[i] = false, **map**[i][i] = INF;

pre[root] = root;

while(true){

for(i = 1; i <= V; i++){

if(flag[i] || i == root) continue;

pre[i] = i;

for(j = 1; j <= V; j++)

if(!flag[j] && **map**[j][i] < **map**[pre[i]][i])

pre[i] = j;

if(pre[i] == i) return -1;

}

for(i = 1; i <= V; i++){

if(flag[i] || i == root) continue;

for(j = 1; j <= V; j++) visited[j] = false;

visited[root] = true;

j = i;

do{

visited[j] = true;

j = pre[j];

}while(!visited[j]);

if(j == root)continue;

i = j;

do{

sum += **map**[pre[j]][j];

j = pre[j];

}while(j != i);

j = i;

do{

for(k = 1; k <= V; k++)

if(!flag[k] && **map**[k][j] < INF && k != pre[j])

**map**[k][j] -= **map**[pre[j]][j];

j = pre[j];

}while(j != i);

for(j = 1; j <= V; j++){

if(j == i) continue;

for(k = pre[i]; k != i; k = pre[k]){

if(**map**[k][j] < **map**[i][j]) **map**[i][j] = **map**[k][j];

if(**map**[j][k] < **map**[j][i]) **map**[j][i] = **map**[j][k];

}

}

for(j = pre[i]; j != i; j = pre[j]) flag[j] = true;

break;

}

if(i > V){

for(i = 1; i <= V; i++)

if(!flag[i] && i != root) sum += **map**[pre[i]][i];

break;

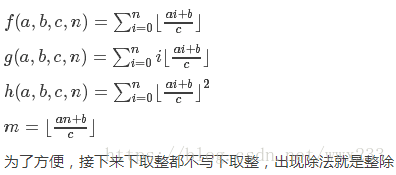
}

}

return sum;

}

1. Euclidean-like Algorithm



struct Data {LL f, g, h;};

Data calc(LL a, LL b, LL c, LL n) {

Data tmp;

if (!a) {

tmp.f=tmp.g=tmp.h=0;

return tmp;

}

if (a>=c || b>=c) {

tmp=calc(a%c,b%c,c,n);

n%=mo;

tmp.h=(tmp.h+

n\*(n+1)%mo\*(2\*n+1)%mo\*inv6%mo\*(a/c)%mo\*(a/c)%mo

+(n+1)\*(b/c)%mo\*(b/c)%mo

+(LL)2\*(a/c)\*tmp.g%mo

+(LL)2\*(b/c)\*tmp.f%mo

+n\*(n+1)%mo\*(a/c)%mo\*(b/c))%mo;

tmp.f=(tmp.f

+n\*(n+1)/2%mo\*(a/c)

+(n+1)\*(b/c))%mo;

tmp.g=(tmp.g

+n\*(n+1)%mo\*(2\*n+1)%mo\*inv6%mo\*(a/c)

+n\*(n+1)/2%mo\*(b/c))%mo;

return tmp;

}

LL m=((LL)a\*n+b)/c;

Data nxt=calc(c,c-b-1,a,m-1);

n%=mo; m%=mo;

tmp.f=((n\*m-nxt.f)%mo+mo)%mo;

tmp.g=(LL)((n\*(n+1)%mo\*m-nxt.f-nxt.h)%mo+mo)\*inv2%mo;

tmp.h=((m\*(m+1)%mo\*n-(LL)2\*(nxt.g+nxt.f)%mo-tmp.f)%mo+mo)%mo;

return tmp;

}

1. Simplex

void **pivot**(int l, int p){

**swap**(idm[l], idn[p]);

double temp = -a[l][p];

a[l][p] = -1;

for (int i = 0; i <= n; ++i)

if (**fabs**(a[l][i]) <= eps)

a[l][i] = 0;

else

a[l][i] /= temp;

for (int i = 0; i <= m; ++i) {

if (i == l)

continue;

if (**fabs**(a[i][p]) <= eps) {

a[i][p] = 0;

continue;

}

temp = a[i][p];

a[i][p] = 0;

for (int j = 0; j <= n; ++j)

a[i][j] += a[l][j] \* temp;

}

}

double **simplex**(){

while (true) {

int l = m + 1, p = n + 1;

idm[l] = idn[p] = n + m + 1;

double mn = inf;

for (int i = 1; i <= n; ++i)

if (a[0][i] > eps && idn[i] < idn[p])

p = i;

if (p == n + 1)

return a[0][0];

for (int i = 1; i <= m; ++i)

if (a[i][p] < -eps) {

double temp = a[i][0] / -a[i][p];

if (temp < mn - eps || (temp < mn + eps && idm[i] < idm[l])) {

l = i;

mn = temp;

}

}

if (l == m + 1) {

cout << "Unbounded" << endl;

**exit**(0);

}

**pivot**(l, p);

}

}

double **solve**(){

double temp = 0;

int p = 0;

for (int i = 1; i <= m; ++i)

if (a[i][0] < temp) {

temp = a[i][0];

p = i;

}

for (int i = 1; i <= n; ++i)

idn[i] = i;

for (int i = 1; i <= m; ++i)

idm[i] = i + n;

if (temp == 0)

return **simplex**();

static double b[MAXN];

for (int i = 0; i <= n; ++i)

b[i] = a[0][i], a[0][i] = 0;

idn[++n] = 0;

a[0][n] = -1;

for (int i = 1; i <= m; ++i)

a[i][n] = 1;

**pivot**(p, n);

if (**simplex**() < -eps) {

cout << "Infeasible" << endl;

**exit**(0);

}

for (int i = 1; i <= m; ++i) {

if (idm[i] != 0)

continue;

for (int j = 1; j <= n; ++j)

if (**fabs**(a[i][j]) > eps) {

**pivot**(i, j);

break;

}

break;

}

p = 0;

for (int i = 1; i <= n; ++i)

if (idn[i] == 0)

p = i;

for (int i = 0; i <= m; ++i)

**swap**(a[i][p], a[i][n]);

**swap**(idn[p], idn[n]);

n--;

for (int i = 1; i <= m; ++i) {

if (idm[i] > n)

continue;

for (int j = 0; j <= n; ++j)

a[0][j] += a[i][j] \* b[idm[i]];

}

for (int i = 1; i <= n; ++i)

if (idn[i] <= n)

a[0][i] += b[idn[i]];

return **simplex**();

}

1. Group theory & Number theory

Burnside: http://img.blog.csdn.net/20170523141748714?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvbGlhbmd6aGFveWFuZzE=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

Polya:

Suppose we’re to compute , which is difficult to compute directly. We can try to find another function. Consider the prefix sum of http://latex.codecogs.com/gif.latex?(f*g)(x):

We get a recurrence relation for S:

If the prefix sum of and can be computed efficiently, then we can do a memorization search. Runtime is. If we preprocess S(x) for all , time complexity is .

In this way, we can compute the prefix sum of and easily (let )

1. Miller-Rabin & rho

LL prime[5] = {2, 3, 5, 233, 331};

LL pow\_mod(LL a, LL n, LL mod)

{

LL ret = 1;

while (n) {

if (n & 1) ret = ret \* a % mod;

a = a \* a % mod;

n >>= 1;

}

return ret;

}

bool miller\_rabin(LL n)

{

if (n < 2 || (n != 2 && !(n & 1))) return 0;

LL s = n - 1;

while (!(s & 1)) s >>= 1;

for (int i = 0; i < 5; ++i) {

if (n == prime[i]) return 1;

LL t = s, m = pow\_mod(prime[i], s, n);

while (t != n - 1 && m != 1 && m != n - 1) {

m = m \* m % n;

t <<= 1;

}

if (m != n - 1 && !(t & 1)) return 0;

}

return 1;

}

int find\_factorplus(int N) {

int a = 2;

int b = a;

do {

a = f(a); // Usually f(x)=x\*x + a0, a0 is rand()

b = f(f(b));

p = gcd( abs( b - a ) , N);

if( p > 1 ) return p; // Found factor: p

} while( b != a );

return 0; // Failed

}

1. KM

LL slack[maxn];

int w[maxn][maxn];

LL ll[maxn], lr[maxn];

int match[maxn];

bool vl[maxn], vr[maxn];

bool hungarian(int p) {

    if (vl[p])

        return false;

    vl[p] = true;

    for (int i=1; i<=n; i++) {

        if (vr[i] || ll[p] + lr[i] != w[p][i])

            continue;

        vr[i] = true;

        if (!match[i] || hungarian(match[i])) {

            match[i] = p;

            return true;

        }

    }

    return false;

}

LL KM() {

    for (int i=1; i<=n; i++)

        for (int j=1; j<=n; j++)

            ll[i] = max(ll[i], (LL)w[i][j]);

    for (int i=1; i<=n; i++) {

        memset(vl, 0, sizeof vl);

        memset(vr, 0, sizeof vr);

        if (hungarian(i))

            continue;

        for (int k=1; k<=n; k++)

            slack[k] = inf;

        for (int j=1; j<=n; j++) {

            if (!vl[j]) continue;

            for (int k=1; k<=n; k++) {

                if (vr[k]) continue;

                slack[k] = min(slack[k], lr[k] + ll[j] - w[j][k]);

            }

        }

        while (1) {

            LL delta = inf;

            int to = -1;

            for (int k=1; k<=n; k++)

                if (!vr[k]) delta = min(delta, slack[k]);

            for (int k=1; k<=n; k++){

                if (vl[k]) ll[k] -= delta;

                if (vr[k]) lr[k] += delta;

                else {

                    slack[k] -= delta;

                    if (slack[k] == 0) {

                        to = k;

                    }

                }

            }

            if (!match[to])

                break;

            int src = match[to];

            vl[src] = vr[to] = true;

            for (int j=1; j<=n; j++)

                if (!vr[j])

                    slack[j] = min(slack[j], ll[src] + lr[j] - w[src][j]);

        }

        memset(vl, 0, sizeof vl);

        memset(vr, 0, sizeof vr);

        hungarian(i);

    }

    LL sum = 0;

    for (int i=1; i<=n; i++)

        sum += lr[i] + ll[match[i]];

    return sum;

}