

# Contents

<b>1 Math</b>	1
1.1 Math Basic . . . . .	1
1.2 Euler Function . . . . .	1
1.3 China Remain Theorem . . . . .	1
1.4 Math Counting . . . . .	2
<b>2 String</b>	
2.1 KMP . . . . .	
2.2 LPS . . . . .	
<b>3 Tree</b>	
3.1 Tree Min Vertex Cover . . . . .	
<b>4 Graph</b>	
4.1 Bipartite: MaxMatch, MinVerCover, MaxIndSet .	
4.2 Min Vertex Cover . . . . .	
<b>5 Flow</b>	
5.1 Dinic . . . . .	
<b>6 Geometry</b>	
6.1 Geometry basic . . . . .	
6.2 Minimal Enclose Disk . . . . .	
6.3 2D Convex Hull . . . . .	

```
set nocompatible
```

```
set enc=utf-8
set fenc=utf-8
```

```
set tabstop=4
set softtabstop=4
set shiftwidth=4
set backspace=2
set autoindent
set cindent
```

```
syntax on
set t_Co=256
set number
set showmatch
set hls
```

```
autocmd FileType cpp noremap <F9> :w <bar> :! g++ % -std
=c++11 -O2 -Wall && ./a.out<CR>
```

## 1 Math

### 1.1 Math Basic

```
1 vector<pii> primeFac(int n) {
2     vector<pii> ret;
3     for(int i=2; n>1; ++i){
4         if( n%i != 0 ) continue;
5         int e = 0;
6         while( n%i == 0 ) ++e , n/=i;
7         ret.push_back({i, e});
8     }
9     return ret;
10 }
11 long long fastPow(long long x, int n, long long m){
12     long long ans = 1LL;
13     while( n ){
14         if( n&1 ) ans = ans * x % m;
15         x = x*x % m;
16         n >>= 1;
17     }
18     return ans;
19 }
20 long long modInv(long long x, long long p){
21     return fastPow(x, p-2, p);
22 }
23 long long modInv_euler(long long x, long long m){
24     // must be gcd(x,m)==1
25     // phi is euler function: O(sqrt(x))
26     return fastPow(x, phi(m)-1, m);
27 }
```

### 1.2 Euler Function

```
int phi(int n){
    // euler function: in [0,n], # of coprime(i, n)
    vector<pii> fac = primeFac(n);
    int num = 1 , m = 1;
    for(auto &p : fac)
        num *= (p.first-1) , m *= p.first;
    return n/m * num;
}
```

### 1.3 China Remain Theorem

```
bool china_solvable(vector<pii> &rule) {
    for(int i=0; i<rule.size(); ++i)
        for(int j=1; j<rule.size(); ++j) {
            int gcd = __gcd(rule[i].second, rule[j].second);
            if( rule[i].first%gcd != rule[j].first%gcd )
                return false;
        }
    return true;
}
long long china(const vector<pii> &rule, int nlt=0){
    // solve x = ai (mod mi)
    // rule should solvable
    long long MM = 1LL;
    for(auto &r : rule)
        MM = lcm(MM, r.second);
    long long x = 0LL;
    for(auto &r : rule){
        long long ai = r.first;
        long long mi = r.second;
        long long Mi = MM / r.second;
        long long Mv = modInv_euler(Mi%mi, mi);
        long long tmp = ai*Mi%MM *Mv %MM;
        x = (x+tmp) % MM;
    }
    if( x>=nlt ) return x;
    long long n = ceil((nlt-x)*1.0/MM);
    return x + n*MM;
}
```

## 1.4 Math Counting

```
const int MaxNum = 1000004;
const int modNum = 1000000009;
long long fac [MaxNum];
long long facIv[MaxNum];
void initFac(){
    fac[0] = facIv[0] = 1LL;
    for(int i=1; i<MaxNum; ++i) {
        fac [i] = fac[i-1]*i % modNum;
        facIv[i] = modInv(fac[i], modNum);
    }
}
long long Cnm(int n, int m){
    if( m==0 || n==m ) return 1LL;
    return fac[n]*facIv[m] % modNum *facIv[n-m] % modNum;
}
long long nBlock_kColor(int n,int k){
    // n different blocks; k different colors
    // use inclusion-exclusion principle
    long long ans = fastPow(k, n, modNum);
    bool del = true;
    for(int i=k-1; i>0; --i, del=!del){
        long long now = Cnm(k, i)*fastPow(i, n, modNum) %
            modNum;
        if( del ) ans = (ans+modNum-now) % modNum;
        else ans = (ans+now) % modNum;
    }
    return ans;
}
```

## 2 String

### 2.1 KMP

```
class kmp{
private:
    int prefix[maxLen];
    char pat[maxLen];
public:
    void setPattern(const char *str){
        strcpy(pat, str);
        prefix[0] = -1;
        int i=1, j=0;
        for( ; str[i]!='\0' ; ++i, ++j ){
            if( str[i]==str[j] )
                prefix[i] = prefix[j];
            else
                prefix[i] = j;
            while( j>=0 && str[j]!=str[i] )
                j = prefix[j];
        }
        prefix[i] = j;
    }
    int search(const char *str){
        // return index of str match pattern
        int i=0, j=0;
        for( ; str[i]!='\0' && pat[j]!='\0' ; ++i, ++j ){
            while( j>=0 && pat[j]!=str[i] )
                j = prefix[j];
        }
        if( pat[j]=='\0' )
            return i-j;
        return -1;
    }
    int countMatched(const char *str){
        // return # of pattern in str
        int cnt = 0;
        int i=0, j=0;
        while( true ){
            if( pat[j]=='\0' ) ++cnt;
            if( str[i]=='\0' ) break;
            while( j>=0 && pat[j]!=str[i] )
                j = prefix[j];
            ++i, ++j;
        }
        return cnt;
    }
};
```

### 2.2 LPS

```
int lps(const char *str){
    // return len of longest palindrom substring
    static char emptyChar = '@';
    static char tmp[maxLen*2];
    static int lprb[maxLen*2];
    // [i-lprb[i], i+lprb[i]] is the lps when mid is i
    for(int i=0, j=-1; true; ++i){
        if( str[i]=='\0' ){
            tmp[++j] = emptyChar;
            tmp[++j] = '\0';
            break;
        }
        tmp[++j] = emptyChar;
        tmp[++j] = str[i];
    }
    lprb[0] = 0;
    int rightBorder = 0, midId = 0;
    for(int i=1; tmp[i]!='\0'; ++i){
        if( i>rightBorder ){
            rightBorder = i;
            midId = i;
            lprb[i] = 0;
        }
    }
}
```

```

int mirId = midId - (i-midId);
if( i+lprb[mirId] > rightBorder )
    lprb[i] = rightBorder - i;
else if( i+lprb[mirId] < rightBorder )
    lprb[i] = lprb[mirId];
else{
    int j=lprb[mirId];
    while( tmp[i+j]!='\0' && i-j>=0 && tmp[i+j]==tmp[i-j] )
        ++j;
    rightBorder = i+j-1;
    midId = i;
    lprb[i] = j-1;
}
}
int ans = 1;
for(int i=0 ; tmp[i]!='\0' ; ++i)
    if( lprb[i]>ans )
        ans = lprb[i];
return ans;
}

```

## 3 Tree

### 3.1 Tree Min Vertex Cover

```

class TreeMinVertexCover {
private:
    static const int maxNum = 100004;
    vector<int> G[maxNum];
    int in[maxNum];
public:
    bool pick[maxNum];
    int MVC; // min vertex cover
    void init() {
        for(int i=0; i<maxNum; ++i)
            G[i].clear();
        memset(in, 0, sizeof(in));
    }
    void addEdge(int u, int v) {
        G[u].emplace_back(v);
        G[v].emplace_back(u);
        ++in[u];
        ++in[v];
    }
    int treeMinVertexCover() {
        memset(pick, 0, sizeof(pick));
        MVC = 0;
        queue<int> myQ;
        for(int i=1; i<=maxNum; ++i)
            if( in[i]==1 ) myQ.push(i);
        while( myQ.size() ) {
            int nowAt = myQ.front();
            myQ.pop();
            if( in[nowAt]==0 ) continue;
            ++MVC;
            int id;
            for(int i=0; i<G[nowAt].size(); ++i)
                if( in[G[nowAt][i]] ) {
                    id = G[nowAt][i];
                    break;
                }
            for(int i=0; i<G[id].size(); ++i)
                if( in[G[id][i]] ) {
                    --in[G[id][i]];
                    --in[id];
                    if( in[G[id][i]]==1 )
                        myQ.push(G[id][i]);
                }
        }
        return MVC;
    }
};

```

## 4 Graph

### 4.1 Bipartite: MaxMatch, Min-VerCover, MaxIndSet

```
class Bipartite {
private:
    static const int MaxNum = 1004;
    vector<int> g[MaxNum];
    bool visited [MaxNum];

    bool bipart(int nowAt, int nowSide) {
        visited[nowAt] = true;
        side[nowAt] = nowSide;
        for(auto &id : g[nowAt])
            if( !visited[id] )
                bipart(id , !nowSide);
            else if( side[id]==nowSide )
                return false;
        return true;
    }

    bool maxMatch(int nowAt) {
        visited[nowAt] = true;
        for(auto &id : g[nowAt])
            if( cp[id]==-1
                || (!visited[cp[id]] && maxMatch(cp[id])) ) {
                cp[id] = nowAt;
                cp[nowAt] = id;
                return true;
            }
        return false;
    }

    void minVertexCover(int nowAt) {
        MVC[nowAt] = 1;
        for(auto &id : g[nowAt])
            if( !MVC[id] ) {
                MVC[id] = 1;
                minVertexCover(cp[id]);
            }
    }

    void maxIndependentSet(int nowAt) {
        MIS[nowAt] = 1;
        for(auto &id : g[nowAt])
            if( !MIS[cp[id]] )
                maxIndependentSet(cp[id]);
    }
public:
    int matchNum; // max match num
    int cp [MaxNum]; // id and cp[id] is couple
    bool side[MaxNum]; // left/right side
    bool MVC [MaxNum]; // min vertex cover
    bool MIS [MaxNum]; // max indepent set
    void addEdge(int u, int v) {
        g[u].emplace_back(v);
        g[v].emplace_back(u);
    }
    void init() {
        for(int i=0; i<MaxNum; ++i)
            g[i].clear();
    }
    bool countAll() {
        // if graph is not bipartite return false

        // bipartite
        memset(side, 0, sizeof(side));
        memset(visited, 0, sizeof(visited));
        for(int i=0; i<MaxNum; ++i)
            if( !visited[i] && !bipart(i, 0) )
                return false;

        // maximum match
        // O(VE), this code can be more optimized
        // alternative: dinic O(V^0.5*E)
```

```
        matchNum = 0;
        memset(cp, -1, sizeof(cp));
        for(int i=0; i<MaxNum; ++i){
            if( cp[i]!=-1 ) continue;
            memset(visited, 0, sizeof(visited));
            if( maxMatch(i) )
                ++matchNum;
        }

        // min vertex cover
        memset(MVC, 0, sizeof(MVC));
        for(int i=0; i<MaxNum; ++i)
            if( side[i]==1 && cp[i]==-1 )
                minVertexCover(i);
        for(int i=0; i<MaxNum; ++i)
            if( side[i]==1 )
                MVC[i] = !MVC[i];

        // max independent set
        memset(MIS, 0, sizeof(MIS));
        for(int i=0; i<MaxNum; ++i)
            if( cp[i]==-1 )
                maxIndependentSet(i);
        for(int i=0; i<MaxNum; ++i)
            if( side[i]==1 && cp[i]!=-1
                && !MIS[i] && !MIS[cp[i]] )
                MIS[i] = 1;

        return true;
    }
};
```

### 4.2 Min Vertex Cover

```
struct MinVertexCover {
private:
    static const int MaxNum = 54;
    vector<int> G[MaxNum];
    int in[MaxNum];

    int undo(vector<int> &record) {
        for(int i=0; i<record.size(); ++i)
            ++in[record[i]];
        record.clear();
    }

    int delNode(int u, vector<int> &record) {
        for(int i=0; i<G[u].size(); ++i)
            if( in[G[u][i]] ) {
                --in[G[u][i]];
                --in[u];
                record.push_back(G[u][i]);
                record.push_back(u);
            }
    }

    int cnt(int from, int *visited, bool type) {
        if( visited[from] ) return 0;
        if( type==1 ) visited[from] = 1;
        for(int i=0; i<G[from].size(); ++i)
            if( in[G[from][i]] && !visited[G[from][i]] )
                return type+cnt(G[from][i], visited, !type);
        return type;
    }

    int cnt(int *visited) {
        int ret = 0;
        for(int i=0; i<MaxNum; ++i)
            if( in[i]==1 && !visited[i] )
                ret += cnt(i, visited, 0);
        for(int i=0; i<MaxNum; ++i)
            if( in[i]==2 && !visited[i] )
                ret += cnt(i, visited, 0);
        return ret;
    }
public:
```

```

int MVCPick[MaxNum];
int MVC; // min vertex cover
void init() {
    for(int i=0; i<MaxNum; ++i)
        G[i].clear();
    memset(in, 0, sizeof(in));
}
void addEdge(int u, int v) {
    G[u].push_back(v);
    G[v].push_back(u);
    ++in[u];
    ++in[v];
}
void minVertexCover(int nowMVC=0, const int *lastPick=
    NULL) {
    //  $O(n^2 * 1.38^n)$ 
    int nowPick[MaxNum] = {};
    if( nowMVC==0 ) {
        MVC = MaxNum;
        memset(MVCPick, 0, sizeof(MVCPick));
    }
    else memcpy(nowPick, lastPick, sizeof(nowPick));

    int maxid = 0;
    for(int i=0; i<MaxNum; ++i)
        if( in[i]>in[maxid] )
            maxid = i;
    if( in[maxid]<=2 ) {
        nowMVC += cnt(nowPick);
        if( nowMVC<MVC ) {
            MVC = nowMVC;
            memcpy(MVCPick, nowPick, sizeof(nowPick));
        }
        return;
    }

    vector<int> record;

    delNode(maxid, record);
    nowPick[maxid] = 1;
    minVertexCover(nowMVC+1, nowPick);
    nowPick[maxid] = 0;
    undo(record);

    int cnt = 0;
    for(int i=0; i<G[maxid].size(); ++i)
        if( in[G[maxid][i]] ) {
            ++cnt;
            delNode(G[maxid][i], record);
            nowPick[G[maxid][i]] = 1;
        }
    minVertexCover(nowMVC+cnt, nowPick);
    undo(record);
}
};

```

## 5 Flow

### 5.1 Dinic

```

struct Dinic{
    enum{
        maxN = 104
    };

    int maxFlow;
    int pipe[maxN][maxN];
    vector<int> g[maxN];

    void init(){
        maxFlow = 0;
        memset(pipe, 0, sizeof(pipe));
        for(int i=0; i<maxN; ++i)
            g[i].clear();
    }

    int level[maxN];
    bool bfsLabeling(int s, int t){
        memset(level, 0, sizeof(level));
        queue<int> myQ;
        myQ.push( s );
        level[s] = 1;
        while( !myQ.empty() ){
            int nowAt = myQ.front();
            myQ.pop();
            for(int i=0; i<g[nowAt].size(); ++i)
                if( !level[g[nowAt][i]] && pipe[nowAt][g[nowAt][i]]
                ){
                    level[g[nowAt][i]] = level[nowAt] + 1;
                    myQ.push( g[nowAt][i] );
                }
        }
        return level[t];
    }

    int dfsFindRoute(const int &nowAt, const int &t, const
        int &maxC){
        if( nowAt==t ){
            maxFlow += maxC;
            return maxC;
        }
        for(int i=0; i<g[nowAt].size(); ++i){
            int next = g[nowAt][i];
            if( level[next] != level[nowAt]+1 ) continue;
            if( !pipe[nowAt][next] ) continue;
            int nowOut = dfsFindRoute(next, t, min(maxC, pipe[
                nowAt][next]));
            if( nowOut==0 )
                continue;
            pipe[nowAt][next] -= nowOut;
            pipe[next][nowAt] += nowOut;
            return nowOut;
        }
        return 0;
    }

    void findMaxFlow(int s, int t){
        while( bfsLabeling(s, t) )
            while( dfsFindRoute(s, t, 1e8) )
                ;
    }

    void findMinCut(int s, int t){
        findMaxFlow(s, t);
        for(int i=0; i<maxN; ++i){
            if( level[i]!=0 )
                continue;
            for(int j=0; j<g[i].size(); ++j)
                if( level[g[i][j]]==0 )
                    printf("%d %d\n", i, g[i][j]);
        }
    }
}

```

```

    }
};

```

## 6 Geometry

### 6.1 Geometry basic

```

struct Point{
    double x,y;
    Point(double xi=0.0,double yi=0.0){
        x = xi , y = yi;
    }
    Point operator - (const Point &r)const{
        return Point(x-r.x , y-r.y);
    }
};
typedef Point Vector;

double angle(const Vector &v,const Vector &u){
    // return rad [0, pi] of two vector
    return acos( dot(v,u)/len(v)/len(u) );
}
Vector rotate(const Vector &v,double rad){
    return Vector(
        v.x*cos(rad) - v.y*sin(rad),
        v.x*sin(rad) + v.y*cos(rad)
    );
}
double pointSegLen(const Point &A,const Point &B,const
    Point &Q){
    if(A==B) return len(Q-A);
    if( dot(B-A , Q-A)<0 ) return len(Q-A);
    if( dot(B-A , Q-B)>0 ) return len(Q-B);
    return fabs( cross(B-A , Q-A) ) / len(B-A);
}
bool pointOnSeg(const Point &A,const Point &B,const Point
    &Q){
    return fabs( len(Q-B)+len(Q-A)-len(A-B) ) < 1e-9;
}

struct Line{
    Point P0;
    Vector v;
    Line(const Point &pi,const Vector &vi):p0(pi) , v(vi)
    {}
};

double pointLineLen(const Line &L,const Point &Q){
    return fabs( cross(L.v , Q-L.P0) ) / len(L.v);
}
Point projectToLine(const Line &L,const Point &Q){
    double t = dot(Q-L.P0 , L.v) / dot(L.v , L.v);
    return L.P0 + L.v * t;
}

Point innerCircle(point &p1, point &p2, point &p3){
    // p1,p2,p3 should not on same line
    double a1 = (-2*p1.x + 2*p2.x);
    double b1 = (-2*p1.y + 2*p2.y);
    double c1 = (p2.x*p2.x + p2.y*p2.y - p1.x*p1.x - p1.y*
        p1.y);
    double a2 = (-2*p1.x + 2*p3.x);
    double b2 = (-2*p1.y + 2*p3.y);
    double c2 = (p3.x*p3.x + p3.y*p3.y - p1.x*p1.x - p1.y*
        p1.y);
    double cx = (c1*b2-c2*b1) / (a1*b2-a2*b1);
    double cy = (a1*c2-a2*c1) / (a1*b2-a2*b1);
    return Point(cx, cy);
}
Point outerCircle(point &p1, point &p2, point &p3) {
    // p1,p2,p3 should not on same line
    double x1 = (p1.x+p2.x)/2.0;
    double y1 = (p1.y+p2.y)/2.0;
    double x2 = (p2.x+p3.x)/2.0;
    double y2 = (p2.y+p3.y)/2.0;
    double vx = p2.x-p1.x;
    double vy = p2.y-p1.y;
    double ux = p3.x-p2.x;

```

```

double uy = p3.y-p2.y;
double A = vx*x1 + vy*y1;
double B = ux*x2 + uy*y2;
double cx = (uy*A - vy*B) / (uy*vx - ux*vy);
double cy = (ux*A - vx*B) / (ux*vy - uy*vx);
return Point(cx, cy);
}

```

## 6.2 Minimal Enclose Disk

```

struct Circle {
    Point c;
    double R2; // square of radius
    Circle() {}
    Circle(const Point &p1, const Point &p2) {
        c.x = (p1.x+p2.x)/2.0;
        c.y = (p1.y+p2.y)/2.0;
        R2 = dot(p1-p2, p1-p2)/4.0;
    }
    Circle(const Point &p1, const Point &p2, const Point &
        p3) {
        // p1, p2, p3 should not on same line
        c = outerCircle(p1, p2, p3);
        double dx = p1.x - c.x;
        double dy = p1.y - c.y;
        R2 = dx*dx + dy*dy;
    }
    bool contain(const Point &p) const {
        double dx = c.x - p.x;
        double dy = c.y - p.y;
        return fdif(dx*dx + dy*dy - R2)<=0;
    }
};

Circle minEncloseDisk(vector<Point> &ps) {
    // Find minimal circal enclose all point
    // worst case O(n^3), expected O(n)
    Circle D;
    if( ps.size()==0 ) return D;
    if( ps.size()==1 ) {
        D.c = ps[0];
        D.R2 = 0.0;
        return D;
    }

    random_shuffle(ps.begin(), ps.end());
    D = Circle(ps[0], ps[1]);
    for(int i=2; i<ps.size(); ++i) {
        if( D.contain(ps[i]) )
            continue;
        D = Circle(ps[i], ps[0]);
        for(int j=1; j<i; ++j) {
            if( D.contain(ps[j]) )
                continue;
            D = Circle(ps[i], ps[j]);
            for(int k=0; k<j; ++k) {
                if( D.contain(ps[k]) )
                    continue;
                D = Circle(ps[i], ps[j], ps[k]);
            }
        }
    }
}

```

## 6.3 2D Convex Hull

```

bool turnLeft(const Vector &v1, const Vector &v2) {
    return fdif(cross(v1, v2)) > 0LL;
}

vector<Point> convexHull(vector<Point> &ps) {
    // return convex hull without redundant point
    sort(ps.begin(), ps.end());

    vector<Point> up;

```

```

for(int i=0; i<ps.size(); ++i) {
    while( up.size()>1
        && !turnLeft(up.back()-up[up.size()-2],
            ps[i]-up.back()) )
        up.pop_back();
    up.emplace_back(ps[i]);
}

```

```

vector<Point> btn;
for(int i=ps.size()-1; i>=0; --i) {
    while( btn.size()>1
        && !turnLeft(btn.back()-btn[btn.size()-2],
            ps[i]-btn.back()) )
        btn.pop_back();
    btn.emplace_back(ps[i]);
}

vector<Point> res(up);
res.insert(res.end(), btn.begin()+1, btn.end());
res.pop_back();
return res;
}

```