Contents

1 Mathematics

1.1 Extended Euclidean Algorithm

```
int ext_gcd(int a, int b, int *x, int *y) {
    if(!b) {
        *x = 1, *y = 0;
        return a;
    }
    int r = ext_gcd(b, a%b, x, y);
    int t = *x;
    *x = *y;
    *y = t-a/b**y;
    return r;
}
```

1.2 Euler's Totient Function ($\phi(n)$)

```
// int p[pn] = primes
int phi(int n) {
    int l = sqrt(n), ans = n;
    F(pn) {
        if (p[i] > l) break;
        int e = 0;
        while (n%p[i] == 0) n /= p[i], ++e;
        if (e) ans = ans/p[i]*(p[i]-1), l = sqrt(n);
    }
    if (n > 1) ans = ans/n*(n - 1)
    return ans;
}
```

1.3 Modular Multiplicative Inverse

1.3.1 $ext\_gcd(a, b, x, y)$

```
int mod_inv(a, m) {
    int x, g = ext_gcd(a, m, x, int y);
    if (g == 1) return x%m;
    return 0; // Inverse doesn't exist.
}
```

1.3.2 $\phi(n)$

```
int mod_inv(a, m) {
    if (__gcd(a, m) == 1) return pow(a, phi(m)-1)%m;
    return 0; // Inverse doesn't exist.
}
```

1.4 Chinese Remainder Theorem

```
// Overflowed?
// int a[r] = remainders, m[r] = dividers
int crt() {
    int mod = 1, ans = 0, im;
    F(r) mod *= m[i];
    F(r) ans = (ans+(((a[i]*mod/m[i])%mod)
                *mod_inv(mod/m[i], m[i]))%mod)%mod;
    return ans; // ans == 0 iff no solution.
}
```

1.5 Miller–Rabin Primality Test

```
srand(7122);
inline int power(int x,int p,int mod) {
    int s=1,m=x;
    while(p) {
        if(p&1) s=(long long)s*m%mod;
        p>>=1;
        m=(long long)m*m%mod;
    }
    return s;
}
// suppose n-1 = u*2^t
int _u,_t;
inline bool witness(int a,int n) {
    int x,nx,i;
    x=power(a,_u,n);
    for(i=0;i<_t;i++) {
        nx=(long long)x*x%n;
        if(nx==1&&x!=1&&x!=n-1) return 1;
        x=nx;
    }
    return x!=1;
}
inline bool miller_rabin(int n,int s=50) {
    // iterate s times of witness on n
    // return 1 if prime, 0 otherwise
    int a;
    if(n<2) return 0;
    if(!(n&1)) return n==2;
    _u=n-1;
    _t=0;
    while(_u&1) {
        _u>>=1;
        _t++;
    }
    while(s--) {
        a=rand()%(n-1)+1;
        if(witness(a,n)) return 0;
    }
    return 1;
}
```

2 Graph

2.1 Trees

2.1.1 Kirchhoff's Matrix Tree Theorem

```
int kirchhoff() {
    int det = 0, tmp, Q[N][2*N];
    F(N) Fi(j, N) if (adj[i][j])
        ++Q[i][i], Q[i][j] = -1;
    F(N-1) Fi(j, N-2) Q[i][N-1+j] = Q[i][j];
    F(N-1) {
        tmp = 1;
        Fi(j, N-1) tmp *= Q[j][i+j];
        det += tmp;
        tmp = 1;
        Fi(j, N-1) tmp *= Q[N-2-j][i+j];
        det -= tmp;
```

```
    }
    return abs(det);
}
```

2.1.2 Prüfer Code

```
int n;
vector<int> G[1000000], P;
void GtoP() {
    int tmp, deg[n];
    priority_queue<int, vector<int>, greater<int>> Q;
    P.clear();
    memset(deg, 0, sizeof(deg));
    F(n) if ((deg[i] = G[i].size()) == 1) Q.push(i);
    while (!Q.empty()) {
        tmp = Q.top();
        Q.pop();
        --deg[tmp];
        F(G[tmp].size()) if (deg[G[tmp][i]]) {
            P.push_back(G[tmp][i]);
            if (--deg[G[tmp][i]] == 1) Q.push(G[tmp][i]);
            break;
        }
        if (P.size() == n-2) break;
    }
}
void PtoG() {
    F(n) G[i].clear();
    int v[n], j = 0, r[2];
    memset(v, 0, sizeof(v));
    F(n-2) ++v[P[i]];
    F(n) if (!v[i]) {
        G[i].push_back(P[j]);
        G[P[j]].push_back(i);
        --v[i], --v[P[j++]];
        if (j == n-2) break;
    }
    j = 0;
    F(n) if (~v[i]) r[j++] = i;
    G[r[0]].push_back(r[1]);
    G[r[1]].push_back(r[0]);
}
```

2.1.3 Cayley's formula: the number of labeled spanning trees

$$N = \begin{cases} V^{V-2}, \quad when \quad G \quad is \quad complete. \\ V_1^{V_2-1}V_2^{V_1-1}, \quad V_1+V_2=V, \quad when \quad G \quad is \quad bipartite. \end{cases}$$

2.2 Connectivity

2.2.1 Articulation Points and Bridges

```
struct Arc {
    int ed, id;
    Arc( int n_ed, int n_id ): ed(n_ed), id(n_id) {}
};
int V, E;
int edge[MAXE][2];//st, ed
vector<Arc> conn[MAXV];
int prev[MAXV], dep[MAXV], low[MAXV], chi[MAXV];
bool vis[MAXV];
set<int> AP, Bridge;
int stk[MAXE], top;
vector<Arc>::iterator stk_it[MAXE];
void DFS( const int &st ) {
    int curr;
    vector<Arc>::iterator it;
    top = -1 + 1;
    stk[top] = st; stk_it[top] = conn[st].begin();
    prev[st] = st; dep[st] = -1;
    while ( top > -1 ) {
        curr = stk[top]; it = stk_it[top]; top--;
        if ( !vis[curr] ) {
            vis[curr] = true;
            dep[curr] = low[curr] = dep[ prev[curr] ] + 1;
            chi[curr] = 0;
        } else {
            low[curr] = min( low[curr], low[it->ed] );
            if ( curr != st && low[it->ed] >= dep[curr] )
                AP.insert( curr );
            if ( low[it->ed] > dep[curr] ) Bridge.insert( it->id );
            if ( ++it == conn[curr].end() ) continue;
        }
        for ( ; it!=conn[curr].end(); ++it ) {
            if ( !vis[it->ed] ) {
                ++top; stk[top] = curr; stk_it[top] = it;
                ++top; stk[top] = it->ed; stk_it[top] =
                conn[it->ed].begin();
                prev[it->ed] = curr;
                chi[curr]++;
                break;
            } else {
                if ( it->ed != prev[curr] ) low[curr] = min( low[curr],
                                                    dep[it->ed] );
            }
        }
    }
    if ( chi[st] > 1 ) AP.insert( st );
}
void Find_AP_Bridge() {
    memset( vis, false, V + 1 );
    for ( int i=0; i<V; i++ ) {
        if ( !vis[i] ) DFS( i );
    }
}
```

2.3 Flow

2.3.1 Maximum Flows

```
struct Arc {
    int ed, cap, dual;
    Arc() {}
    Arc( int ed_, int cap_, int dual_ ):ed(ed_), cap(cap_),
                                        dual(dual_){}
};
int V, S, T, E;
Arc arc[MAXE];
vector<int> ArcList[MAXV];
int lbl[MAXV], lblV[MAXV];
inline void Add_arc( int st, int ed, int ca ) {
    arc[E] = Arc( ed, ca, E+1 );
    ArcList[st].push_back( E );
    arc[E+1] = Arc( st, 0, E );
    ArcList[ed].push_back( E+1 );
    E += 2;
}
int Augment( const int &curr, const int &prec ) {
    int augc = prec, minlbl = V-1, ext = 0;
    vector<int>::iterator it;
    if ( curr == T ) return prec;
    for ( it=ArcList[curr].begin(); it!=ArcList[curr].end();
    ++it ) {
        if ( arc[*it].cap > 0 && lbl[curr] == lbl[ arc[*it].ed ]
        + 1 ) {
            ext = Augment( arc[*it].ed, Min( augc, arc[*it].cap ) );
            if ( ext ) {
                arc[*it].cap -= ext;
                arc[ arc[*it].dual ].cap += ext;
                augc -= ext;
            }
            if ( lbl[S] >= V || augc == 0 ) return prec - augc;
        }
    }
    for ( it=ArcList[curr].begin(); it!=ArcList[curr].end();
    ++it ) {
        if ( arc[*it].cap > 0 ) {
            minlbl = min( minlbl, lbl[ arc[*it].ed ] );
        }
    }
    if ( --lblV[ lbl[curr] ] == 0 ) lbl[S] = V;
    lbl[curr] = minlbl + 1;
    lblV[ lbl[curr] ]++;
    return prec - augc;
}
int Max_Flow() {
    int flow = 0;
    memset( lbl, 0, sizeof(int)*V );
    memset( lblV, 0, sizeof(int)*V );
    lblV[0] = V;
    while ( lbl[S] < V ) {
        flow += Augment( S, INF );
    }
    return flow;
}
void Init_Network() {
    for ( int i=0; i<V; i++ ) {
        ArcList[i].clear();
    }
    E = 0;
}
```

2.3.2 Minimum Cost Maximum Flow

```
struct Arc {
    int ed, cap, cost, dual;
    Arc() {}
    Arc( int ed_, int cap_, int cost_, int dual_ ):
    ed( ed_ ), cap( cap_ ),
    cost( cost_ ), dual( dual_ ){}
};
int S, T, V, E;
Arc arc[MAXE];
```

```
int ArcList[MAXV][MAXE];
int deg[MAXV];
int Queue[MAXE*MAXV], fr, re;
bool inq[MAXV];
int curr_cost_sum;
inline void Add_arc( int st, int ed, int ca, int co ) {
    arc[E] = Arc( ed, ca, co, E+1 );
    ArcList[st][ deg[st]++ ] = E;
    arc[E+1] = Arc( st, 0, -co, E );
    ArcList[ed][ deg[ed]++ ] = E+1;
    E += 2;
}
inline int Min( int a, int b ) {
    return a < b ? a : b;
}
bool Augment() {
    int dist[MAXV], prev[MAXV];
    int curr, ext, i, it;
    for ( i=0; i<V; i++ ) {
        dist[i] = INF;
        prev[i] = -1;
    }
    dist[S] = 0;
    fr = re = 0;
    Queue[re++] = S;
    inq[S] = true;
    while ( fr < re ) {
        curr = Queue[fr++];
        inq[curr] = false;
        for ( i=0; i<deg[curr]; i++ ) {
            it = ArcList[curr][i];
            if ( arc[it].cap > 0 ) {
                if ( dist[curr]+arc[it].cost
                < dist[ arc[it].ed ] ) {
                    dist[ arc[it].ed ] = dist[curr]+arc[it].cost;
                    prev[ arc[it].ed ] = it;
                    if ( !inq[ arc[it].ed ] ) {
                        Queue[re++] = arc[it].ed;
                        inq[ arc[it].ed ] = true;
                    }
                }
            }
        }
    }
    if ( dist[T] == INF ) return false;
    curr = T;
    ext = INF;
    while ( curr != S ) {
        ext = Min( ext, arc[ prev[curr] ].cap );
        curr = arc[ arc[ prev[curr] ].dual ].ed;
    }
    curr = T;
    curr_cost_sum = 0;
    while ( curr != S ) {
        arc[ prev[curr] ].cap -= ext;
        curr_cost_sum += ext * arc[ prev[curr] ].cost;
        arc[ arc[ prev[curr] ].dual ].cap += ext;
        curr = arc[ arc[ prev[curr] ].dual ].ed;
    }
    return true;
}
int Solve() {
    int cost_sum = 0;
    while ( Augment() ) {
        cost_sum += curr_cost_sum;
    }
    return cost_sum;
}
```

2.4 Matching

2.4.1 Maximum Weighted Bipartite Matching: Kuhn-Munkres Algorithm

```
#define MAXN 100
#define INF INT_MAX
int g[MAXN][MAXN],lx[MAXN],ly[MAXN],slack_y[MAXN];
int px[MAXN],py[MAXN],match_y[MAXN],par[MAXN];
int n;
void adjust(int y){ // Inverse all the edges on Augmented path.
    match_y[y]=py[y];
    if(px[match_y[y]]!=-2)
        adjust(px[match_y[y]]);
}
bool dfs(int x){ // DFS to find out Augmented path.
    for(int y=0;y<n;++y){
        if(py[y]!=-1)continue;
        int t=lx[x]+ly[y]-g[x][y];
        if(t==0){
            py[y]=x;
            if(match_y[y]==-1){
                adjust(y);
                return 1;
            }
            if(px[match_y[y]]!=-1)continue;
            px[match_y[y]]=y;
            if(dfs(match_y[y]))return 1;
        }else if(slack_y[y]>t){
            slack_y[y]=t;
            par[y]=x;
        }
    }
    return 0;
}
inline int km(){
    memset(ly,0,sizeof(int)*n);
    memset(match_y,-1,sizeof(int)*n);
    for(int x=0;x<n;++x){
        lx[x]=-INF;
        for(int y=0;y<n;++y){
            lx[x]=max(lx[x],g[x][y]);
        }
    }
    for(int x=0;x<n;++x){
        for(int y=0;y<n;++y)slack_y[y]=INF;
        memset(px,-1,sizeof(int)*n);
        memset(py,-1,sizeof(int)*n);
        px[x]=-2;
        if(dfs(x))continue;
        bool flag=1;
        while(flag){
            int cut=INF;
            for(int y=0;y<n;++y)
                if(py[y]==-1&&cut>slack_y[y])cut=slack_y[y];
            for(int j=0;j<n;++j){
                if(px[j]!=-1)lx[j]-=cut;
                if(py[j]!=-1)ly[j]+=cut;
                else slack_y[j]-=cut;
            }
            for(int y=0;y<n;++y){
                if(py[y]==-1&&slack_y[y]==0){
                    py[y]=par[y];
                    if(match_y[y]==-1){
                        adjust(y);
                        flag=0;
                        break;
                    }
                    px[match_y[y]]=y;
                    if(dfs(match_y[y])){
                        flag=0;
                        break;
                    }
                }
            }
        }
    }
    int ans=0;
    for(int y=0;y<n;++y)if(g[match_y[y]][y]!=-INF)ans+=g[match_y[y]][y];
    return ans;
}
```

2.4.2 Stable Marriage

```
int n;
int xpr[MAXNUM][MAXNUM]; // priority[xpurposer][rank]
int yrk[MAXNUM][MAXNUM]; // rank[yreviewer][xpurposer]
int xid[MAXNUM]; // x's next purpose
int xy[MAXNUM],yx[MAXNUM]; // matches
int sn,st[MAXNUM*MAXNUM]; // free men
inline void stable_marriage() {
    int i,x,y;
    sn=0;
    for(i=0;i<n;i++) {
        xid[i]=0;
        st[sn++]=i;
        yx[i]=NIL;
    }
    while(sn) {
        x=st[--sn];
        y=xpr[x][xid[x]++];
        if(yx[y]==NIL) {
            yx[y]=x;
        } else {
            if(yrk[y][yx[y]]>yrk[y][x]) {
                st[sn++]=yx[y];
                yx[y]=x;
            } else {
                st[sn++]=x;
            }
        }
    }
    for(i=0;i<n;i++) xy[yx[i]]=i;
}
```

3 Computational Geometry

3.1 Implementation

```
const double EPS = 1e-9;
typedef complex<double> Point;
typedef complex<double> Vector;
#define x real()
#define y imag()
//conj -> reflecting about y=0(x-axis)
//<< and >> is work, format is (x,y)
struct Segment {
    Point p1, p2;
    Segment() {}
    Segment( const Point &np1, const Point &np2 ):
        p1(np1), p2(np2){}
};
struct Line {
    double a, b, c; //ax+by=c
    Line() {}
    Line( double na, double nb, double nc ):
        a(na), b(nb), c(nc){}
};
struct Circle {
     Point O;
     double r;
     Circle() {}
     Circle( const Point &nO, const double &nr ):
        O(nO), r(nr) {}
};
inline double Dot( const Vector &A, const Vector &B ) {
    return A.x * B.x + A.y * B.y;
}
inline double Cross( const Vector &A, const Vector &B ) {
    return A.x * B.y - A.y * B.x;
}
inline double Dist( const Point &A, const Point &B ) {
    return abs( A - B );
}
inline double Dist2( const Point &A, const Point &B ) {
    return norm( A - B );
}
inline double Squ( const double &a ) {
    return a * a;
}
inline double Fabs( const double a ) {
    return a >= 0.00 ? a : a * -1.00;
}
inline int Cmp( const double a, const double b ) {
    if ( Fabs( a - b ) < EPS ) return 0;
    else return a < b ? -1 : 1;
}
struct Point_Cmp {
    bool operator()( const Point &A, const Point &B ) {
        return Cmp( A.x, B.x ) < 0 ||
        ( Cmp( A.x, B.x ) == 0 && Cmp( A.y, B.y ) < 0 );
    }
};
inline Line P_Gen_L( const Point &A, const Point &B ) {
    return Line( A.y-B.y, B.x-A.x,
    (A.y-B.y)*A.x + (B.x-A.x)*A.y );
}
pair<int, Point> L_Intersect_L( const Line &A, const Line &B ) {
    //1: point, 2: line, 0: none
    double delta = ( A.a*B.b - A.b*B.a );
    double ra, rb;
    ra = ( A.c*B.b - A.b*B.c );
    rb = ( A.a*B.c - A.c*B.a );
    if ( Cmp( delta, 0.00 ) == 0 ) {
        if ( Cmp( ra, 0.00 ) == 0 && Cmp( rb, 0.00 ) == 0 )
            return make_pair( 2, Point() );
        else return make_pair( 0, Point() );
    } else {
        return make_pair( 1,
        Point( ra/delta + EPS, rb/delta + EPS ) );
    }
}
bool operator==( const Point &A, const Point &B ) {
    return Cmp( A.x, B.x ) == 0 && Cmp( A.y, B.y ) == 0;
}
inline pair<int, Point> S_Intersect_S
( const Segment &A, const Segment &B ) {
    //2: normal, 1: restrictly, -1: segment, 0: none
    double A1 = Cross( B.p1-A.p1, B.p2-A.p1 ),
    A2 = Cross( B.p1-A.p2, B.p2-A.p2 );
    double B1 = Cross( A.p1-B.p1, A.p2-B.p1 ),
    B2 = Cross( A.p1-B.p2, A.p2-B.p2 );
    double area1 = Cross( A.p2-A.p1, B.p2-B.p1 ),
    area2 = Cross( B.p1-A.p1, B.p2-B.p1 );
    if ( Cmp( A1 * A2, 0.00 ) < 0 && Cmp( B1 * B2, 0.00 ) < 0 ) {
        return make_pair( 1, A.p1 + ( area2/area1 )*( A.p2-A.p1 ) );
    } else if ( Cmp( A1 * A2, 0.00 ) <= 0 &&
            Cmp( B1 * B2, 0.00 ) <= 0 ) {
        if ( Cmp( area1, 0.00 ) == 0 ) {
            if ( Cmp( Dot( A.p1-B.p1, A.p2-B.p1 ), 0.00 ) < 0 ||
            Cmp( Dot( B.p1-A.p1, B.p2-A.p1 ), 0.00 ) < 0
            || Cmp( Dot( A.p1-B.p2, A.p2-B.p2 ), 0.00 ) < 0 ||
            Cmp( Dot( B.p1-A.p2, B.p2-A.p2 ), 0.00 ) < 0 ) {
                return make_pair( -1, Point() );
            }
            if ( ( A.p1 == B.p1 && A.p2 == B.p2 ) || ( A.p1 == B.p2
            && A.p2 == B.p1 ) ) return make_pair( -1, Point() );
            if ( A.p1 == B.p1 || A.p1 == B.p2 )
                return make_pair( 2, A.p1 );
            if ( A.p2 == B.p1 || A.p2 == B.p2 )
                return make_pair( 2, A.p2 );
            return make_pair( 0, Point() );
        } else return make_pair( 2,
            A.p1 +( area2/area1 )*( A.p2-A.p1 ) );
    } else return make_pair( 0, Point() );
}
inline bool In_Circle( const Point &A, const Circle &C ) {
    return Cmp( Dist( A, C.O ), C.r ) <= 0;
}
inline Circle Outer_Circle( const Point &A, const Point &B ) {
    if ( A == B ) return Circle( A, 0.00 );
    Circle res;
    res.O = 0.5 * ( A+B );
    res.r = Dist( res.O, A );
    return res;
}
inline Circle Outer_Circle( const Point &A, const Point &B,
const Point &C ) {
    if ( A == B ) return Outer_Circle( A, C );
    if ( A == C ) return Outer_Circle( A, B );
    if ( B == C ) return Outer_Circle( A, B );
    double a, b, c, q = Squ( Cross( A-B, B-C ) ) * 2.00;
    Circle res;
    a = ( norm( B-C ) * Dot( A-B, A-C ) ) / q;
    b = ( norm( A-C ) * Dot( B-A, B-C ) ) / q;
    c = ( norm( A-B ) * Dot( C-A, C-B ) ) / q;
    res.O = a*A + b*B + c*C;
    res.r = Dist( res.O, A );
    return res;
}
inline int P_Quadrand( const Point &a ) {
    if ( Cmp( a.x, 0.00 ) == 0 && Cmp( a.y, 0.00 ) == 0 )
        return -1;
    else if ( Cmp( a.y, 0.00 ) == 0 )
        return Cmp( a.x, 0.00 ) >= 0 ? 0 : 4;
    else if ( Cmp( a.x, 0.00 ) == 0 )
        return Cmp( a.y, 0.00 ) >= 0 ? 2 : 6;
    else if ( Cmp( a.x, 0.00 ) > 0 && Cmp( a.y, 0.00 ) > 0 )
        return 1;
    else if ( Cmp( a.x, 0.00 ) < 0 && Cmp( a.y, 0.00 ) > 0 )
        return 3;
    else if ( Cmp( a.x, 0.00 ) < 0 && Cmp( a.y, 0.00 ) < 0 )
        return 5;
    else return 7;
}
struct PolarAngle_Cmp {
    inline bool operator()( const Point &A, const Point &B ) {
        int p = P_Quadrand( A ), q = P_Quadrand( B );
        double cr = Cross( A, B );
        if ( p != q ) return p < q;
        else return Cmp( cr, 0.00 ) ?
            Cmp( cr, 0.00 ) > 0 : norm( A ) < norm( B );
    }
};
```

3.2 Closest Pair Problem

```
struct Point {
    double x, y, d;
    bool operator<( const Point &k ) const {
        return d < k.d;
    }
};
int N;
Point p[MAXN];
inline double Dist( const Point &A, const Point &B ) {
    return sqrt( ( A.x-B.x )*( A.x-B.x ) +
    ( A.y-B.y )*( A.y-B.y ) );
}
double Closest_Pair() {
    int i, j, ori = rand() % N;
    double d, dmin = 1e+100;
    for ( i=0; i<N; i++ ) p[i].d = Dist( p[ori], p[i] );
    sort( p, p + N );
    for ( i=0; i<N; i++ ) {
        for ( j=i-1; j>=0 && p[i].d-p[j].d<dmin; j-- ) {
            d = Dist( p[i], p[j] );
            if ( d < dmin ) dmin = d;
        }
    }
    return dmin;
```

```
}
```

3.3 Farthest Pair Problem

```
double Farthest_Pair( int n, Point p[] ) {
    int M, i, j, h, cnti, cntj;
    double maxx, maxy, res;
    Point vi, vj;
    static Point CH[50005];
    Convex_Hull( n, p, CH, M );
    if ( M == 2 ) return sqrt( Dist2( CH[0], CH[1] ) );
    i = j = 0;
    maxy = -10000000.00;
    for ( i=0; i<M; i++ ) {
        if ( CH[i].y > maxy ||
        ( maxy == CH[i].y && CH[i].x > maxx ) ) {
            h = i;
            maxx = CH[i].x;
            maxy = CH[i].y;
        }
    }
    j = h;
    res = 0.00;
    cnti = cntj = 0;
    while ( cnti <= M && cntj <= M ) {
        if ( Dist2( CH[i], CH[j] ) > res )
            res = Dist2( CH[i], CH[j] );
        vi = make_pair( CH[i+1].x-CH[i].x, CH[i+1].y-CH[i].y );
        vj = make_pair( CH[j+1].x-CH[j].x, CH[j+1].y-CH[j].y );
        if ( Cross( vi, vj ) < -1e-9 ) {
            j++;
            if ( j >= M ) j = 0;
            cntj++;
        } else {
            i++;
            if ( i >= M ) i = 0;
            cnti++;
        }
    }
    return sqrt( res );
}
```

3.4 Minimum Enclosing Circle

```
Point p[MAXN];
Circle MEC( int n, int lvl ) {
    int i;
    Circle curr;
    static Point sta[3];
    if ( lvl == 0 ) curr = Outer_Circle( p[0], p[1] );
    else if ( lvl == 1 ) curr = Outer_Circle( p[0], sta[0] );
    else if ( lvl == 2 ) curr = Outer_Circle( sta[0], sta[1] );
    else if ( lvl == 3 ) return Outer_Circle( sta[0], sta[1],
                                              sta[2] );
    for ( i=2-lvl; i<n; i++ ) {
        if ( In_Circle( p[i], curr ) ) continue;
        else {
            sta[lvl] = p[i];
            curr = MEC( i, lvl + 1 );
        }
    }
    return curr;
}
```

4 String

4.1 Extended Knuth-Morris-Pratt Algorithm

```
void ext_kmp( char S[], char T[], int Slen, int Tlen, int
Next[], int Ext[] ) {
    int i, p, A;
    Next[0] = Tlen;
    p = 1;
    while ( p < Tlen && T[p] == T[p-1] ) p++;
    Next[1] = p - 1;
    A = 1;
    for ( i=2; i<Tlen; i++ ) {
        if ( i+Next[i-A] < p ) Next[i] = Next[i-A];
        else {
            if ( p < i ) p = i;
            while ( p < Tlen && T[p] == T[p-i] ) p++;
            Next[i] = p - i;
            A = i;
        }
    }
    p = 0;
    while ( p < Tlen && S[p] == T[p] ) p++;
    Ext[0] = p;
    A = 0;
    for ( i=1; i<Slen; i++ ) {
        if ( i+Next[i-A] < p ) Ext[i] = Next[i-A];
        else {
            if ( p < i ) p = i;
            while ( p < Slen && p-i < Tlen && S[p] == T[p-i] )
                p++;
            Ext[i] = p - i;
            A = i;
        }
    }
}
```

4.2 Suffix Array

```
int n;
char s[MAXLEN];
int sa[MAXLEN],rank[MAXLEN*2];
int cnt[MAXLEN],tmp[MAXLEN];
int da[MAXLEN];
inline void radix_pass
(int maxrank,int *from,int *to,int *w) {
    int i;
    for(i=0;i<=maxrank;i++) cnt[i]=0;
    for(i=0;i<n;i++) cnt[w[from[i]]+1]++;
    for(i=0;i<=maxrank;i++) cnt[i+1]+=cnt[i];
    for(i=0;i<n;i++) to[cnt[w[from[i]]]++]=from[i];
}
inline void find_sa() {
    int i,k,maxrank='z'-'a'+1;
    for(i=0;i<n;i++) sa[i]=i;
    for(i=0;i<n;i++) {
        rank[i]=s[i]-'a'+1;
        rank[i+n]=0;
    }
    for(k=1;k<n;k<<=1) {
        radix_pass(maxrank,sa,tmp,rank+k);
        radix_pass(maxrank,tmp,sa,rank);
        for(i=1,tmp[0]=maxrank=1;i<n;i++) {
            if(rank[sa[i]]>rank[sa[i-1]]
            ||rank[sa[i]]==rank[sa[i-1]]
            &&rank[sa[i]+k]>rank[sa[i-1]+k]) ++maxrank;
            tmp[i]=maxrank;
        }
        for(i=0;i<n;i++) rank[sa[i]]=tmp[i];
    }
    for(i=0;i<n;i++) rank[i]--;
}
inline void find_da() {
    int i,j,d=0;
    for(i=0;i<n;i++) {
        if(rank[i]==n-1) {
            d--;
            continue;
        }
        if(d<0) d=0;
        j=sa[rank[i]+1];
        while(s[i+d]==s[j+d]) d++;
        da[rank[i]]=d--;
    }
}
```