UNIVERSITY OF CALGARY

TEAM NOTEBOOK 2013 ACM ICPC

HICHEM ZAKARIA AICHOUR, TORYN QWYLLYN KLASSEN, KENT WILLIAMS-KING

Our Magic Incantations List

When Choosing a Problem

- Find out which balloons are the popular ones!
- Pick one with a nice, ***clean solution*** that you are totally convinced ***will work*** to do first.

Before Designing Your Solution

- Highlight the important information on the problem statement – input bounds, special rules, formatting, etc.
- Look for code in this notebook that you can use!
- Convince yourself that your algorithm will run with time to spare on the biggest input.
- Create several ***test cases*** that you will use, especially for ***special or boundary cases***.

Prior to Submitting

- Check ***maximum*** input, ***zero*** input, and other ***degenerate*** test cases.
- Cross check with team mates' supplementary test cases.
- Read the problem ***output specification*** one more time – your program's output behavior is fresh in your mind.
- Does your program work with ***negative*** numbers?
- Make sure that your program is reading from an appropriate ***input file***.
- Check all ***variable initialization***, ***array bounds***, and ***loop variables*** (`i` vs. `j`, `m` vs. `n`, etc.).
- Finally, run a `diff` on the provided sample output and your program's output.
- And don't forget to submit your solution under the ***correct problem number!***

After Submitting

- Immediately ***print a copy*** of your source.
- Staple the solution to the problem statement and keep them safe. Do not lose them!

If It Doesn't Work…

- Remember that a ***run-time error*** can be ***division by zero***.
- If the solution is not complex, allow a team mate to start the problem afresh.
- Don't waste a lot of time – it's not shameful to ***simply give up!!!***

Data Structure

Fenwick Tree

1-D

```
// Note: it is modified to be zero based
// Intermediate
int N;
VI FT;

void init(){FT.assign(N,0);}

int cum(int pos){
   if(pos<0) return 0;
   if(pos>=N) pos = N-1;
   pos++; int sum=0;
   for( ; pos ; pos -= pos&(-pos)) sum += FT[pos-1];
   return sum;
}

void inc(int pos,int val){
   if(pos<0 || pos>=N) return;
   pos++;
   for( ; pos<=N ; pos += pos&(-pos))  FT[pos-1] += val;
}
```

2-D

```
// Note: it is modified to be zero based
// Intermediate
int NX, NY;
VVI FT;

void init(){FT.assign(NX,VI(NY));}

int cum(int x, int y){
   if(x<0 || y<0) return 0;
   if(x>=NX) x = NX-1;
   if(y>=NY) y = NY-1;
   x++; y++; int sum = 0;
   for(; x ; x -= x & -x) for(int y1=y ; y1 ; y1 -= y1 & -y1)
        sum += FT[x-1][y1-1];
   return sum;
}
```

```
void inc(int x, int y, int val){
   if(x<0 || y<0 || x>=NX || y>=NY) return;
   x++; y++;
   for(; x<=NX ; x += x & -x)
      for(int y1=y ; y1<=NY ; y1 += y1 & -y1)
         FT[x-1][y1-1] += val;
}
```

Segment Tree

Call “initSeg” then call “BuildSeg”

```
int segN;
VI lptr, rptr, segSt, segEn;
void initSeg(int n){ // n = range(0,n-1)
   lptr.clear(); rptr.clear();
   segSt.clear(); segEn.clear();

   lptr.push_back(-1); rptr.push_back(-1);
   segSt.push_back(0); segEn.push_back(n-1);
}
void BuildSeg(int u=0){
   if(segSt[u] == segEn[u]){ /* base case */ return;}
   int mid = (segSt[u]+segEn[u])/2;
   lptr[u] = segN;
   lptr.push_back(-1); rptr.push_back(-1);
   segSt.push_back(segSt[u]); segEn.push_back(mid);
   rptr[u] = segN+1;
   lptr.push_back(-1); rptr.push_back(-1);
   segSt.push_back(mid+1); segEn.push_back(segEn[u]);
   segN += 2;

   Build(lptr[u]); Build(rptr[u]);
   // modify data based on children
}
```

Dynamic Programming

LCM Knowing the Subsequence O(nm)

```
int LCS(VI a, VI b, VI& ret){
   int i,j;
   int N = a.size();
   int M = b.size();

   FOR(i,N+1) FOR(j,M+1)
      {dp[i][j]=0; prex[i][j]=-1; prey[i][j]=-1;}

   REP(i,1,N) REP(j,1,M){
      if(a[i-1] == b[j-1]){
         dp[i][j] = dp[i-1][j-1] + 1;
         prex[i][j] = i-1; prey[i][j] = j-1;
      }
      else if(dp[i-1][j] > dp[i][j-1]){
         dp[i][j] = dp[i-1][j];
         prex[i][j] = i-1; prey[i][j] = j;
      }
      else{
         dp[i][j] = dp[i][j-1];
         prex[i][j] = i;  prey[i][j] = j-1;
      }
   }

   ret.clear();
   int x,y;
   i=N; j=M;
   while(i>0 && j>0){
      if(prex[i][j]==i-1 && prey[i][j]==j-1)
         ret.push_back(a[i-1]);
      x = prex[i][j];
      y = prey[i][j];
      i=x; j=y;
   }
   reverse(All(ret));
   return dp[N][M];
}
```

LIS Knowing the Subsequence O(n lg(k))

```
#define NMAX 1000*1000*10
int N, NA;
int V[NMAX],A[NMAX],pos[NMAX];
int LIS(){ // original(V), ans(A)
   int i; int* it;
   NA=0;
   A[NA++]=V[0];
   fill(pos,pos+N,0); // in algorithm
   int ret = 1;
   REP(i,1,N-1){
      it = lower_bound(A,A+NA,V[i]);
      pos[i] = it-A;
      ret = max(ret,pos[i]);
      if(it==A+NA) A[NA++]=V[i];
      else *it = V[i];
   }
   FORD(i,N) if(pos[i]==ret) A[ret--] = V[i];
   return NA;
}
```

KMP String

```
void kmpProcess(string& w, VI& b){
   b = VI(w.size()+1);
   int i=0, j=-1; b[0]=-1;
   while(i<w.size()){
      while(j>=0 && w[i]!=w[j]) j = b[j];
      i++; j++; b[i] = j;
   }
}

VI kmpSearch(string& s, string& w, VI& b){
   int i=0, j=0;
   VI ret;
   while(i<s.size()){
      while(j>=0 && s[i]!=w[j]) j = b[j];
      i++; j++;
      if(j == w.size()){ret.push_back(i-j); j = b[j];}
   }
   return ret;}
```

Geometry

2D

Point and Vector

```
#define Pt            complex<double>
#define Vec           Pt
#define xx            real()
#define yy            imag()

#define dot(a,b)      (conj(a)*(b)).xx
#define cross(a,b)    (conj(a)*(b)).yy
#define Abs(a)        sqrt(dot(a,a))
#define unit(a)       a/Abs(a)
```

Warning
Parenthesis for #define

```
bool operator< (Pt a, Pt b){
   if(fabs(a.xx-b.xx)>1e-9) return a.xx<b.xx;
   return a.yy+1e-9 < b.yy;
}
bool operator==(Pt a, Pt b){return !(a<b) && !(b<a);}
```

Angles

```
#define rotate(v,t)   (v*Vec(cos(t),sin(t)))
#define rotate(o,p,t) (rotate(p-o,t)+o)
#define reflect(o,p)  (o - (p-o))

#define aSides(a,b,c) acos((b*b + c*c - a*a)/(2*b*c))
#define aSeg(o,a,b)   asin(cross(unit(a-(o)),unit(b-(o))))
```

Lines - Segments - Half Lines

```
#define OnLine(o,u,p) ((p==o) || fabs(cross(p-o,u))<EPS)
#define OnSeg(a,b,p)  (fabs(Abs(a-b)-Abs(p-a)-Abs(p-b)) < EPS)
#define OnRay(o,u,p)  (o==p || unit(p-o)==u)

bool LineIntLine(Pt o1, Vec u1, Pt o2, Vec u2, Pt& r){
   if(fabs(cross(u1,u2)) < EPS) return false;
   r = o1 + (cross(o2-o1,u2)/cross(u1,u2))*u1; return true;
}
```

Circle

```
int CirIntCir(Pt o1,double r1, Pt o2,double r2, Pt& p1, Pt& p2){
   // 0 none - 1 one - 2 two - 3 same
   if(o1==o2 && fabs(r1-r2)<EPS) return 3;
   if(Abs(o2-o1)>r1+r2+EPS||Abs(o2-o1)<fabs(r2-r1)-EPS) return 0;
   p1 = p2 = o1 + unit(o2-o1)*r1;
   double t = aSides(r2,r1,Abs(o2-o1));
   p1 = rotate(o1,p1,t); p2 = rotate(o1,p2,-t);
   return 2;
}
int CirIntLine(Pt o1, double r1, Pt o2, Vec u2, Pt& p1, Pt& p2){
   // 0 none - 1 one - 2 two
   double h = fabs(cross(o1-o2,u2));
   p1 = p2 = u2*dot(u2,o1-o2) + o2;
   if(h > r1 + EPS) return 0; if(h > r1 - EPS) return 1;
   double d = sqrt(r1*r1 - h*h);
   p1 += u2*d; p2 -= u2*d;
}
int CirTanCir(Pt o1, double r1, Pt o2, double r2, vector<Pt>& p1,
vector<Pt>& p2){
   // 0 none - 1 4-pair
   // Requires: rotate   &    r1 <= r2
   p1.resize(4); p2.resize(4);
   double d = Abs(o1-o2);
   double h = r2-r1;
   if(d < r1+r2+EPS) return 0;
   Vec u1 = rotate(unit(o1-o2), acos(h/d));
   Vec u2 = rotate(unit(o1-o2),-acos(h/d));
   p1[0]=o1+u1*r1; p2[0]=o2+u1*r2;
   p1[1]=o1+u2*r1; p2[1]=o2+u2*r2;

   double dd = (r2/(r1+r2)) * d;
   double hh = r2;
   u1 = rotate(unit(o1-o2), acos(hh/dd));
   u2 = rotate(unit(o1-o2),-acos(hh/dd));
   p1[2]=o1-u1*r1; p2[2]=o2+u1*r2; // note (-) with p1
   p1[3]=o1-u2*r1; p2[3]=o2+u2*r2; // note (-) with p1
}

void tanFromPt(Pt o, double r, Pt p, Pt& p1, Pt& p2){
   double d = Abs(p-o);
   double l = sqrt(d*d - r*r);
   double t = aSides(l, r, d);

   p1 = p2 = unit(p-o)*r;
   p1 = o + rotate(p1, t);
   p2 = o + rotate(p2,-t);
}
```

Req: **Pt + rotate + aSides**
Pre: **o, r** → Center and Rad
p → Point
Post: **p1, p2** → p to p1 and p to p2 are tangent to the circle

Triangle

```
double areaTri(double a, double b, double c){
   double s = (a+b+c)/2;
   return sqrt(s*(s-a)*(s-b)*(s-c));}
bool IsTri(double a, double b, double c){
   double s = (a+b+c)/2;
   return s*(s-a)*(s-b)*(s-c) > EPS;}
double radCir(double a, double b, double c){
   return (a*b*c)/sqrt((a+b+c)*(a+b-c)*(a+c-b)*(b+c-a));}
```

Convex

```
#define VPt         vector<Pt>
#define REMOVE_REDUNDANT

double area2(Pt a, Pt b, Pt c) {return cross(a,b) + cross(b,c) +
cross(c,a);}

#ifdef REMOVE_REDUNDANT
bool between(const Pt &a, const Pt &b, const Pt &c) {
  return (fabs(area2(a,b,c)) < EPS && (a.xx-b.xx)*(c.xx-b.xx) <=
0 && (a.yy-b.yy)*(c.yy-b.yy) <= 0);
}
#endif

void ConvexHull(VPt &pts) {
   int i;
   sort(pts.begin(), pts.end());
   pts.erase(unique(pts.begin(), pts.end()), pts.end()); //
remove duplicate
   vector<Pt> up, dn;
   FOR(i,pts.size()){
      while (up.size() > 1 && area2(up[up.size()-2], up.back(),
pts[i]) >= 0) up.pop_back();
      while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(),
pts[i]) <= 0) dn.pop_back();
      up.push_back(pts[i]);
      dn.push_back(pts[i]);
   }
   pts = dn;
   for (int i = (int) up.size() - 2; i >= 1; i--)
pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
   if (pts.size() <= 2) return;
   dn.clear();
   dn.push_back(pts[0]);
   dn.push_back(pts[1]);
   for (int i = 2; i < pts.size(); i++) {
      if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i]))
dn.pop_back();
      dn.push_back(pts[i]);
   }
   if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
      dn[0] = dn.back();
      dn.pop_back();
   }
   pts = dn;
#endif
}
```

Polygons

```
void CCW_Poly(Pt p[], int N){
   int i,n;
   FOR(i,N) if(i==0 || p[i].xx<p[n].xx) n=i;
   rotate(p,p+n,p+N);
   if(fabs(cross(p[1]-p[0], p[N-1]-p[0]))<1e-9
                && p[1].yy>p[N-1].yy)
      reverse(p+1,p+N);
   else if(cross(p[1]-p[0], p[N-1]-p[0])<0)
      reverse(p+1,p+N);
}
```

Req
Pt
Pre
p → points of polygon
N → Size of **p**
Post
p → make **p** CCW

```
bool PointInPolygon(const vector<Pt> &p, Pt q) {
   int i,j;
   bool c = 0;
   FOR(i,p.size()){
      j = (i+1)%p.size();
      if ((p[i].yy <= q.yy && q.yy < p[j].yy ||
         p[j].yy <= q.yy && q.yy < p[i].yy) &&
         q.xx < p[i].xx + (p[j].xx - p[i].xx) * (q.yy - p[i].yy) /
(p[j].yy - p[i].yy))
         c = !c;
   }
   return c;
}
```

2D & 3D

Lines – Segments – Half Lines

```
void pt2line(Pt o, Vec u, Pt p, Pt& c, double& d){
   c = o + u*dot(p-o,u);         d = Abs(p-c);
}
```

Polygons

```
Pt CenPol(VPt& V, double& A){ // Centroid and Area of polygon
   int i,N=V.size();
   Pt C = Pt(3); // 2d: Pt C = Pt(0,0);
   A = 0;
   REP(i,1,N-2){
      double AT = Abs(cross(V[i]-V[0], V[i+1]-V[0]))/2;
      Pt CT = (V[0]+V[i]+V[i+1])/3;
      A += AT;      C += CT*AT;
   }
   C /= A;
   return C;
}
```

Formulas

$$RadOfInner = \frac{2AT}{A+B+C}$$

$$RadOfOutter = \frac{2AT}{A\cos a + B\cos b + C\cos c}$$

Pick's Theorem: $A = I + \frac{B}{2} - 1$

Euler's Theorem: $V - E + F = 2$

3D

Point and Vector

```
#define Vec        valarray<double>
#define Pt         Vec
#define xx         operator[](0)
#define yy         operator[](1)
#define zz         operator[](2)

#define dot(a,b)   ((a)*(b)).sum()
#define Abs(a)     sqrt(dot(a,a))
#define unit(a)    ((a)/Abs(a))
Vec cross(Vec a, Vec b){
return a.cshift(+1)*b.cshift(-1) - a.cshift(-1)*b.cshift(+1);}

Pt NewPt(double a, double b, double c){
   Pt ret(3); ret.xx=a; ret.yy=b; ret.zz=c;
   return ret;
}

bool Less(Pt& a, Pt& b){
   if(fabs(a.xx-b.xx)>EPS) return a.xx<b.xx;
   if(fabs(a.yy-b.yy)>EPS) return a.yy<b.yy;
   return a.zz+EPS < b.zz;
}
```

General

```
void Bases(VPt& pts, Vec& u, Vec& w, Vec& n){
   u = unit(pts[1]-pts[0]);
   w = pts[2]-pts[0];    w = w - u*dot(u,w);   w = unit(w);
   n = cross(u,w);
}

Pt ConvToBases(Pt& p, Pt& o, Vec& u, Vec& w, Vec& n){
   return NewPt(dot(p-o,u), dot(p-o,w), dot(p-o,n));}
Pt ConvFromBases(Pt& p, Pt& o, Vec& u, Vec& w, Vec& n){
   return o + u*p.xx + w*p.yy + n*p.zz;}
```

Convex Polyhedral

```
double VolConvexPolyhedra(VVPt& pts){// vol. of Convex Polyhedra
   int i,j,k;

   double vol = 0;
   FOR(k,pts.size()){
      double A;
      Pt C = CenPol(pts[k],A);

      Pt n =
         unit(cross(pts[k][1]-pts[k][0], pts[k][2]-pts[k][1]));

      FOR(i,pts.size()) {
         FOR(j,pts[i].size())
            if(fabs(dot(pts[i][j]-pts[k][0],n)) > EPS) break;
         if(j != pts[i].size()) break;
      }
      if(dot(pts[i][j]-pts[k][0],n) > 0)
         n = n*-1.0;

      vol += dot(C,n)*A;
   }
   vol /= 3;
   return vol;
}
```

Triangulate Polygon

```
struct triple {
   int i, j, k;
   triple() {}
   triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<double>& x,
vector<double>& y) {
   int i,j,k;
   int n = x.size();
   vector<double> z(n);
   vector<triple> ret;

   FOR(i,n) z[i] = x[i]*x[i] + y[i]*y[i];

   FOR(i,n-2) REP(j,i+1,n-1) REP(k,i+1,n-1){
      if (j == k) continue;
      double xn = (y[j]-y[i])*(z[k]-z[i])-(y[k]-y[i])*(z[j]-z[i]);
      double yn = (x[k]-x[i])*(z[j]-z[i])-(x[j]-x[i])*(z[k]-z[i]);
      double zn = (x[j]-x[i])*(y[k]-y[i])-(x[k]-x[i])*(y[j]-y[i]);
      bool flag = zn < 0;
      for (int m = 0; flag && m < n; m++)
         flag = flag && ((x[m]-x[i])*xn +
         (y[m]-y[i])*yn +
         (z[m]-z[i])*zn <= 0);
      if (flag) ret.push_back(triple(i, j, k));
   }
   return ret;
}
```

Java

```
// compute the area of an Area object containing several disjoint
polygons
static double computeArea(Area area) {
   double totArea = 0;
   PathIterator iter = area.getPathIterator(null);
   ArrayList<Point2D.Double> points =
                                 new ArrayList<Point2D.Double>();

   while (!iter.isDone()) {
      double[] buffer = new double[6];
      switch (iter.currentSegment(buffer)) {
      case PathIterator.SEG_MOVETO:
      case PathIterator.SEG_LINETO:
         points.add(new Point2D.Double(buffer[0], buffer[1]));
         break;
      case PathIterator.SEG_CLOSE:
         totArea += computePolygonArea(points);
         points.clear();
         break;
      }
      iter.next();
   }
   return totArea;
}
```

Matrix

```
// ==================================
// Solve
// ==================================
bool LUP_De(Mat& LU, VI& P){
  int i,j,k,kk;
  int N = LU.size();
  P.clear(); P.resize(N);
  FOR(i,N) P[i] = i;

  FOR(k,N){
    double p = 0;
    REP(i,k,N-1) if(fabs(LU[i][k]) > p){
      p = fabs(LU[i][k]);
      kk = i;
    }
    if(fabs(p) < EPS) return false; // singular
    swap(P[k],P[kk]);
    FOR(i,N) swap(LU[k][i],LU[kk][i]);

    REP(i,k+1,N-1){
      LU[i][k] /= LU[k][k];
      REP(j,k+1,N-1)
                  LU[i][j] -= LU[i][k]*LU[k][j];
    }
  }
  return true;
}
void LUP_Sol(Mat& LU , VI& P , VD& b , VD& x){
  int i,j;
  int N = LU.size();
  VD y(N);
  FOR(i,N){
    y[i] = b[P[i]];
    FOR(j,i) y[i] -= LU[i][j]*y[j];
  }

  x.resize(N);
  FORD(i,N){
    x[i] = y[i];
    REP(j,i+1,N-1) x[i] -= LU[i][j]*x[j];
    x[i] /= LU[i][i];
  }
}
```

```
bool Solve(Mat& A, VD b, VD& x){
  VI P;
  Mat LU = A;
  if(LUP_De(LU,P)==false)
    return false;
  LUP_Sol(LU,P,b,x);
  return true;
}
// ==================================
// Matrix
// ==================================
Mat Inv(Mat M){ // only for N*N Mat
  int i;
  int N=M.size();
  Mat I = Mat(N,VD(N,0));
  FOR(i,N) I[i][i]=1;

  Mat ret = Mat(N,VD(N));
  VI P;
  LUP_De(M,P);
  FOR(i,N) LUP_Sol(M,P,I[i],ret[i]);

  return ret;
}
double Det(Mat& M){ // only for N*N Mat
  int i,j,k;
  Mat LU = M;
  VI P;
  if(!LUP_De(LU,P)) return 0;
  double ret=1;
  FOR(i,LU.size()) ret *= LU[i][i];
  FOR(i,P.size()) while(P[i] != i){
    ret *= -1;
    swap(P[i],P[P[i]]);
  }
  return ret;
}
```

Graph Theory

K-Coloring – χ(G) – Chromatic Number – O(E)

```
int Coloring(){
  int i,j,k;
  int u;
  C.clear();

  int N = Adj.size();
  C.resize(N);
  queue<int> q;
  VVB cnt(N,VB(N));
  VB done(N);
  VB InQ(N);

  FOR(u,N){
    if(done[u]) continue;
    q.push(u); InQ[u] = true;
    while(!q.empty()){
      k = q.front(); q.pop();
      FOR(i,Adj[k].size()){
        j = Adj[k][i];
        cnt[j][C[k]] = true;
      }
      FOR(i,Adj[k].size()){
        j = Adj[k][i];
        if(done[j]) continue;
        while(cnt[j][C[j]]) C[j]++;
        if(!InQ[j]) q.push(j);
        InQ[j] = true;
      }
      done[k] = true;
    }
  }

  int ret = 0;
  FOR(i,N) ret = max(ret,C[i]);
  return ret+1;
}
```

Bellman-Ford O(VE)

```
int bellman(){
  int i,u,v;
  VI d(N,Inf);
  VI level(N);
  queue<int> Q; Q.push(s); d[s] = 0;
  while(!Q.empty()){
    if(d[s] < 0 || level[t] > N) return -Inf;
    u = Q.front(); Q.pop();
    level[u]++;
    if(level[u] > N) continue;
    FOR(i,Adj[u].size()){
      v = Adj[u][i];
      if(d[v] > d[u]+G[u][v]){
        d[v] = d[u]+G[u][v];
        Q.push(v);
      }
    }
  }
  return d[t];
}
```

Euler Tour O(E lg V)

```
#define VsetI   vector<set<int> > // multiset for duplicate

vector<int> tour;
void find_tour(VsetI& Adj, int u){ // n is the number of vertices
  while(!Adj[u].empty()){
    int v = *Adj[u].begin();
    Adj[u].erase(Adj[u].begin()); Adj[v].erase(Adj[v].find(u));
    find_tour(Adj, v);
  }
  tour.push_back(u);
}

void EulerTour(VsetI Adj){
  int i, sum=0, s=0; tour.clear();
  FOR(i,Adj.size()) if(Adj[i].size()%2){sum++; s=i;}
  if(sum > 0 && sum!=2) return;
  find_tour(Adj, s);
}
```

Strongly Connected Components O(E)

```
// input
int N;        VVI Adj;
// Intermediate
VVI AdjRev;   VI order, done;    int NO, currComp;
// output
int NC;       VI compID;

void dfs(int u){
   int i;
   if(done[u]) return;
   done[u]=1;
   FOR(i,Adj[u].size()) dfs(Adj[u][i]);
   NO--; order[NO] = u;
}

void dfsRev(int u){
   int i;
   if(done[u]) return;
   done[u]=1;
   compID[u]=currComp;
   FOR(i,AdjRev[u].size()) dfsRev(Adj[u][i]);
}

void top(){
   int i;
   done.clear(); done.resize(N);
   order.clear(); order.resize(N);
   NO = N;
   FOR(i,N) if(!done[i]) dfs(i);
}

void scc(){
   int u,i;
   NC=0;
   AdjRev.clear(); AdjRev.resize(N);
   FOR(u,N) FOR(i,Adj[u].size()) AdjRev[Adj[u][i]].push_back(u);

   top();
   done.clear(); done.resize(N);
   FOR(i,N){
      u = order[i];
         if(!done[u]){
            currComp = NC++;
            dfsRev(u);
         }
      }
}
```

Maximum Matching Bi-Graph O(VE)

```
// input
#define NMAX 1000
int NL,NR;
int Adj[NMAX][NMAX], deg[NMAX];
// Intermediate
int done[NMAX]
// output
int ML[NMAX], MR[NMAX];
```

Coloring (L (G)) = Max Vertex Degree (G)
Coloring (G) = 2
No C_{Odd}
Max Match = Min Vertex Cover
Max Match = N – Max Ind. Vertex Set
Max Match = N – Min Edge Cover

```
int DFS(int u){
   int i,v;   if(u<0) return 1;  if(done[u]) return 0;
   done[u] = 1;
   FOR(i,deg[u]){
      v = Adj[u][i];
      if(DFS(MR[v])){
         ML[u] = v; MR[v] = u;
         return 1;
      }
   }
   return 0;
}

int Find_Mate(int u){
   fill(done,done+NL,0);
   return DFS(u);
}

int BiMatch(){
   fill(ML,ML+NL,-1);    fill(MR,MR+NR,-1);    int i, ans=0;
   FOR(i,NL) if(ML[i] == -1) ans += Find_Mate(i);
   return ans;
}
```

Warning

It is assumed to have two groups, one for left one for right

Pre:

Adj → elements in right connected to each element in left (i.e: from left to right)

Post:

ML → element in right assigned to left
MR → element in left assigned to right

Min-Cost Max-Matching in BiGraph O(V³)

```
// Requires NL = NR
// Note negate cost[][] for maximization

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
   int n = cost.size(), i,j,k;

   // construct dual feasible solution
   VD u(n);   VD v(n);
   FOR(i,n) {
      u[i] = cost[i][0];
      REP(j,1,n-1) u[i] = min(u[i], cost[i][j]);
   }
   FOR(j,n) {
      v[j] = cost[0][j] - u[0];
      REP(i,1,n-1) v[j] = min(v[j], cost[i][j] - u[i]);
   }

// construct primal solution satisfying complementary slackness
   Lmate = VI(n, -1); Rmate = VI(n, -1);
   int mated = 0;
   FOR(i,n) FOR(j,n){
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
         Lmate[i] = j; Rmate[j] = i; mated++; break;
      }
   }

   VD dist(n); VI dad(n), seen(n);

   // repeat until primal solution is feasible
   while (mated < n) {
      // find an unmatched left node
      int s = 0;
      while (Lmate[s] != -1) s++;

      // initialize Dijkstra
      fill(dad.begin(), dad.end(), -1);
      fill(seen.begin(), seen.end(), 0);
      FOR(k,n) dist[k] = cost[s][k] - u[s] - v[k];

      j = 0;
      while (true) {
         // find closest
         j = -1;
         FOR(k,n) {
            if (seen[k]) continue;
            if (j == -1 || dist[k] < dist[j]) j = k;
         }
         seen[j] = 1;

         // termination condition
         if (Rmate[j] == -1) break;

         // relax neighbors
         const int i = Rmate[j];
         FOR(k,n) {
            if (seen[k]) continue;
            double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
            if (dist[k] > new_dist) {
               dist[k] = new_dist;
               dad[k] = j;
            }
         }
      }

      // update dual variables
      FOR(k,n) {
         if (k == j || !seen[k]) continue;
         const int i = Rmate[k];
         v[k] += dist[k] - dist[j];    u[i] -= dist[k] - dist[j];
      }
      u[s] += dist[j];

      // augment along path
      while (dad[j] >= 0) {
         const int d = dad[j];
         Rmate[j] = Rmate[d];    Lmate[Rmate[j]] = j;    j = d;
      }
      Rmate[j] = s;      Lmate[s] = j;      mated++;
   }

   double value = 0;
   FOR(i,n) value += cost[i][Lmate[i]];

   return value;
}
```

Articulation Point + Bridges O(E)

```
// input
int N;     VVI Adj;
// Intermediate
VI P, dfs_low, dfs_num;    int NCR, root, cnt;
// output
VI IsAP, AP; VII B;

void DFS(int u) {
   int i,j,k;
   dfs_low[u] = dfs_num[u] = cnt++;
   FOR(i,Adj[u].size()){
      int v = Adj[u][i];
      if(dfs_num[v] == -1){
         P[v] = u;
         if(u == root) NCR++;
         DFS(v);
         if(dfs_low[v] >= dfs_num[u] && !IsAP[u] && u!=root){
            IsAP[u] = true; AP.push_back(u);
         }
         if(dfs_low[v] > dfs_num[u]) B.push_back(II(u,v));
         dfs_low[u] = min(dfs_low[u] , dfs_low[v]);
      }
      else if(v != P[u])
         dfs_low[u] = min(dfs_low[u] , dfs_num[v]);
   }
}
int ArtPoint(int s){ // specify the starting node … anything if
the graph is connected
   IsAP.clear(); IsAP.resize(N); AP.clear(); B.clear();
   P.clear(); P.resize(N,-1);
   dfs_low.clear(); dfs_low.resize(N,-1);
   dfs_num.clear(); dfs_num.resize(N,-1);

   NCR = cnt = 0; root = s;
   DFS(root);
   if(NCR > 1){
      IsAP[root] = true;
      AP.push_back(root);
   }
   return AP.size();
}
```

Articulation Point with the Number of disconnected Parts O(V^2+VE)

```
// input
int N;
// Intermediate
VI done;
// output
VI nParts;

void DFS(int u) {
   int i;
   FOR(i,Adj[u].size()){// Adj -> VVI
      int v = Adj[u][i];
      if(!done[v]){
         done[v] = true;
         DFS(v);
      }
   }
}

int ArtPoint(){
   int i,j,k;   int u,v;
   nParts.clear(); nParts.resize(N); // nParts -> VI

   FOR(u,N){
      done.assign(N,0);
      done[u] = true;
      FOR(v,N){
         if(done[v]) continue;
         nParts[u]++;
         done[v] = true; DFS(v);
      }
   }

   int ret = 0;
   FOR(i,N) if(nParts[i] > 1) ret++;
   return ret; // number of articulation points
}
```

Simplex Algorithm

```
// m - number of (less than) inequalities
// n - number of variables
// C - (m+1) by (n+1) array of coefficients:
// row 0 - objective function coefficients
// row 1:m - less-than inequalities
// column 0:n-1 - inequality coefficients
// column n - inequality constants (0 for objective function)
// X[n] - result variables
// return value - maximum value of objective function
// (-inf for infeasible, inf for unbounded)

// input
#define MAXM 400
#define MAXN 400
#define EPS 1e-9
#define INF 1.0/0.0
// Intermediate
double A[MAXM][MAXN];
int basis[MAXM], out[MAXN];

void pivot(int m, int n, int a, int b) {
   int i, j;
   FOR(i,m+1) if(i != a) FOR(j,n+1) if(j != b)
      A[i][j] -= A[a][j] * A[i][b] / A[a][b];
   FOR(j,n+1) if(j != b) A[a][j] /= A[a][b];
   FOR(i,m+1) if(i != a) A[i][b] = -A[i][b]/A[a][b];

   A[a][b] = 1/A[a][b];

   i = basis[a];
   basis[a] = out[b];
   out[b] = i;
}

double simplex(int m, int n, double C[][MAXN], double X[]) {
   int i, j, ii, jj;

   REP(i,1,m) REP(j,0,n) A[i][j] = C[i][j];
   REP(j,0,n) A[0][j] = -C[0][j];
   REP(i,0,m) basis[i] = -i;
   REP(j,0,n) out[j] = j;

   for(;;) {
      for(i = ii = 1; i <= m; i ++) {
         if(A[i][n] < A[ii][n]
            || (A[i][n]==A[ii][n] && basis[i]<basis[ii])) ii = i;
      }

      if(A[ii][n] >= -EPS) break;

      for(j = jj = 0; j < n; j ++) {
         if(A[ii][j] < A[ii][jj]-EPS
            || (A[ii][j] < A[ii][jj]+EPS && out[i]<out[j])) jj=j;
      }

      if(A[ii][jj] >= -EPS) return -INF;
      pivot(m,n,ii,jj);
   }

   for(;;) {
      for(j = jj = 0; j < n; j ++)
         if(A[0][j] < A[0][jj]
            || (A[0][j] == A[0][jj] && out[j] < out[jj])) jj = j;

      if(A[0][jj] > -EPS) break;

      for(i=1,ii=0; i <= m; i ++)
         if(A[i][jj] > EPS &&
            (!ii || A[i][n]/A[i][jj] < A[ii][n]/A[ii][jj]-EPS
            || (A[i][n]/A[i][jj] < A[ii][n]/A[ii][jj]+EPS
               && basis[i]<basis[ii]))) ii = i;

      if(A[ii][jj] <= EPS) return INF;
      pivot(m,n,ii,jj);
   }

   for(j = 0; j < n; j ++) X[j] = 0;
   for(i = 1; i <= m; i ++) if(basis[i] >= 0)
      X[basis[i]] = A[i][n];

   return A[0][n];
}
```

Max Flow (Push-Relabel Algorithm O(V³))

```
// input
VVI Adj, G;
// Intermediate
VI h, e;
VVI AG;
// output
VVI F;

void Init_Preflow(int s){
   int N = Adj.size();
   e.clear(); e.resize(N);
   h.clear(); h.resize(N);
   h[s] = N;

   F.clear(); F.resize(N,VI(N));
   int i;
   FOR(i, Adj[s].size()){
      int u = Adj[s][i];
      F[s][u] = AG[s][u]; F[u][s] = -AG[s][u];

      e[u] = AG[s][u]; e[s] -= AG[s][u];

      AG[u][s] = AG[s][u]; AG[s][u] = 0;
   }
}
void Push(int u,int v){
   int f = min(e[u],AG[u][v]);
   F[u][v] +=    f; F[v][u] = -F[u][v];
   e[u] -= f;    e[v] += f;
   AG[u][v] -= f;   AG[v][u] += f;
}
void Relabel(int u){
   int Min = INT_MAX; // include<climits>
   int i;
   FOR(i, Adj[u].size()){
      int v = Adj[u][i];
      if(AG[u][v]>0) Min = min(Min,h[v]);
   }
   h[u] = Min + 1;
}
int Max_Flow(int s, int t){
   int u,i;
   AG = G;
   Init_Preflow(s);

   bool Still = true;
   while(Still){
      Still = false;
      FOR(u, e.size()){
         if(u==s || u==t || e[u]==0) continue;
         Still = true;
         while(e[u]){
            for(i=0 ; i<Adj[u].size() && e[u] ; i++){
               int v = Adj[u][i];
               if(AG[u][v] > 0 && h[u] == h[v]+1)
                  Push(u,v);
            }
            if(e[u]) Relabel(u);
         }
      }
   }

   int sum = 0;
   FOR(i, Adj[t].size()){
      u = Adj[t][i];
      sum += F[u][t];
   }
   return sum;
}
```

Max Flow Min Cost (Min(O(E lgE f), O(VE lgE fcst))

```
// input
#define NN   1000
#define INF  (INT_MAX/2)
int N, s, t;
int cap[NN][NN], cst[NN][NN];
// Intermediate
#define Pot(u,v) (d[u] + pi[u] - pi[v])
int deg[NN], pre[NN], d[NN], pi[NN];
int Adj[NN][NN];
// output
int fnt[NN][NN];

bool dijkstra(int n){
   int i,u,v,k;
   FOR(i,n) d[i]=INF;
   FOR(i,n) pre[i] = -1;
   d[s] = 0;
   pre[s] = n;
   set<II> PQ; PQ.insert(II(d[s],s));

   while(!PQ.empty()){
      u = PQ.begin()->second;
      k = PQ.begin()->first;
      PQ.erase(PQ.begin());
      if(d[u] != k) continue;
      FOR(i,deg[u]){
         v = Adj[u][i];

         if(fnt[v][u] && d[v]>Pot(u,v)-cst[v][u])
            d[v] = Pot(u,v) - cst[v][pre[v]=u];
         if(fnt[u][v]<cap[u][v] && d[v]>Pot(u,v)+cst[u][v])
            d[v] = Pot(u,v) + cst[pre[v]=u][v];
         if(pre[v] == u)
            PQ.insert(II(d[v],v));
      }
   }
   FOR(i,n) if(pi[i]<INF) pi[i] += d[i];
   return pre[t] >= 0;
}

int MFMC(int n, int& fcst){
   int u,v;
   int flow = fcst = 0;
   FOR(u,n) FOR(v,n) fnt[u][v]=0;
   FOR(u,n) pi[u]=deg[u]=0;

   FOR(u,n) FOR(v,n) if(cap[u][v] || cap[v][u])
      Adj[u][deg[u]++] = v;

   while(dijkstra(n)){
      int bot = INF;
      for(u=pre[v=t] ; v!=s ; u=pre[v=u])
         bot = min(bot, fnt[v][u] ? fnt[v][u] : (cap[u][v]-
fnt[u][v]));
      for(u=pre[v=t] ; v!=s ; u=pre[v=u]){
         if(fnt[v][u]){fnt[v][u] -= bot; fcst -= bot*cst[v][u];}
         else {fnt[u][v] += bot; fcst += bot*cst[u][v];}
      }
      flow += bot;
   }
   return flow;
}
```

2-SAT

```
// - Add clause (v || w) as add_clause(G,VAR(v),VAR(w))
// - To FORCE i to be true: add_clause(G,VAR(i),VAR(i));
// - To implement XOR -- say (i XOR j) :
//    add_clause(G,VAR(i),VAR(j));
//    add_clause(G,NOT(VAR(i)),NOT(VAR(j)));
// NOTE: val[] is indexed by i for var i, not by VAR(i)!!!
// Require: Strongly Connected Component
int VAR(int i) {return 2*i;}     int NOT(int i) {return i^1;}
void add_clause(VVI& Adj, int v, int w) { // adds (v || w)
   if (v == NOT(w)) return;
   Adj[NOT(v)].push_back(w);
   Adj[NOT(w)].push_back(v);
}
bool twoSAT(const VVI& Adj, VI& val) { // assumes graph is built
   val.clear(); val.resize(Adj.size()/2);
   scc();
   for (int i = 0; i < Adj.size(); i += 2) {
      if (compID[i] == compID[i+1]) return false;
      val[i/2] = (compID[i] < compID[i+1]);
   }
   return true;
}
```

MinCut (O(V³))

```
// Input
int G[NN][NN]; // adj-matrix
#define INF (1<<30)
// Intermediate
int v[NN], w[NN], na[NN];     bool a[NN];
long long minCut(int n){
   int i,j;
   FOR(i,n) v[i] = i;

   int best = INF;
   while(n > 1){
      a[v[0]] = true;
      REP(i,1,n-1){
         a[v[i]] = false;
         na[i - 1] = i;
         w[i] = G[v[0]][v[i]];
      }

      int prev = v[0];
      REP(i,1,n-1){
         int zj = -1;
         REP(j,1,n-1) if(!a[v[j]] && (zj < 0 || w[j] > w[zj]))
            zj = j;

         a[v[zj]] = true;

         if(i==n-1){
            best = min(best,w[zj]);

            FOR(j,n)
               G[v[j]][prev] = G[prev][v[j]] += G[v[zj]][v[j]];
            v[zj] = v[--n];
            break;
         }
         prev = v[zj];

         REP(j,1,n-1) if(!a[v[j]]) w[j] += G[v[zj]][v[j]];
      }
   }
   return best;
}
```

Stable Matching O(MW²)

```
// input
int NW, NM;
VVI L; //the list of women in decreasing order [man][i]
VVI R; //Attractivness of wom to man [wom][man]
// Intermediate
VI P;
// output
VI L2R, R2L;

void stableMarriage(){
   int i;
   P.clear(); P.resize(NM);
   L2R.clear(); L2R.resize(NM,-1);
   R2L.clear(); R2L.resize(NW,-1);

   FOR(i,NM){
      int man = i;
      while(man >= 0){
         int wom;
         while(1){
            wom = L[man][P[man]++];
            if(R2L[wom] < 0 || R[wom][man] > R[wom][R2L[wom]])
               break;
         }
         int hubby = R2L[wom]; // divorce
         R2L[L2R[man] = wom] = man; // marry
         man = hubby; // assign bachelor
      }
   }
}
```

Math

Prime, Factoring, & Modular

Sieve

```
bitset<NMAX> IsP; VI P;
void Sieve(){
   long long i,j;
   P.clear(); IsP.set(); // everything 1
   IsP.set(0,false); IsP.set(1,false);
   REP(i,2,NMAX-1){
      if(!IsP.test(i)) continue;
      for(j=i*i ; j<NMAX ; j+=i) IsP.set(j,false);
      P.push_back(i); // P -> VI (all primes)
   }
}
```

Fast Sieve

```
#define NN 20000000
unsigned int prime[NN/64];
#define gP(n) (prime[n>>6]&(1<<((n>>1)&31)))
#define rP(n) (prime[n>>6]&=~(1<<((n>>1)&31)))
void sieve(){
   memset(prime,-1,sizeof(prime));
   unsigned int i;
       for(i=3 ; i*i<=NN ; i+=2) if(gP(i)){
              unsigned int i2 = i + i;
              for(unsigned int j=i*i ; j<NN ; j+=i2) rP(j);
       }
}
```

Is Prime

```
bool IsPrime(int n){
       if(n==2 || n==3 || n==5 || n==7) return true;
       if(n==0 || n==1 || n%2==0 || n%3==0) return false;
       for(int i=5 ; i*i<=n ; i+=6) if(n%i == 0 ||
n%(i+2) == 0) return false;
       return true;
}
```

Relatively Prime

```
bool rPrime(int a, int b){
       int r = a % b;
       while(r != 0){a=b; b=r; r=a%b;}
       return(b == 1);
}
```

Phi (number of rPrime with n and less than it)

```
int phi(int n, VI& pn){ // number and its prime factors
   int i;
   FOR(i,pn.size()) n/=pn[i];
   FOR(i,pn.size()) n*=pn[i]-1;
   return n;
}
```

Prime Factorization

```
// input
VI P
// output
VI p; VI a;
void PF(int n){
   int i,m; p.clear(); a.clear();
   for(i=0 ; P[i]*P[i]<=n ; i++){
      if(n%P[i]) continue;
      m = 0;
      while(n%P[i]==0){m++; n/=P[i];}
      p.push_back(P[i]); a.push_back(m);
   }
   if(n!=1){p.push_back(n); a.push_back(1);}
}
```

Extended Euclidian Optimized

```
// soluation --> (xi,yi) + k(b/gcd(a,b), -a/gcd(a,b))
int EE(int a, int b, int& xi, int& yi){
   if(b==0){xi=1;yi=0; return a;}
   else{
      int ans = EE(b,a%b,xi,yi);
      swap(xi,yi); yi -= (a/b)*xi;
      return ans;
   }
}
```

Chinese Remainder Theorem

```
// solve x = a[i] mod m[i] where gcd(m[i],m[j]) | a[i]-a[j]
// x0 in [0, lcm(m's)], x = x0 + t*lcm(m's) for all t.
int cra(int n, int m[], int a[]) {
   int u = a[0], v = m[0], p, q, r, t;
   for (int i = 1; i < n; i++) {
      r = EE(v, m[i], p, q); t = v;
      v = v/r * m[i]; u = ((a[i]-u)/r * p * t + u) % v;
   }
   if (u < 0) u += v;
   return u;
}
```

Modular Linear Equation Solver

```
// solves ax = b (mod n)
// Requires: Extended Euclid
VI msolve( int a, int b, int n){
   int xi, yi, i;
   VI ret;
   if(n < 0) n = -n;

   int g = EE(a, n, xi, yi);

   if(b % g) return ret;
   int x = (b/g * xi) % n;
   if(x < 0) x += n;
   FOR(i,g) ret.push_back((x + i*n/g) % n);
   return ret;
}
```

Set

```
#define NxtSubSet(cur,S) ((cur-(S)) & S)
int NxtSet(int S){
   int s = S & -S;     int r = S + s;      int ns= r & -r;
   int o = ((ns/s)>>1) - 1;
   return r | o;
}
```

Fraction-less System of Equations

```
int fflinsolve(VVI A, VI b, VI x_star, int n) {
   int k_c, k_r, pivot, sign = 1, d = 1;
   for (k_c = k_r = 0; k_c < n; k_c++) {
      for (pivot = k_r; pivot < n && !A[pivot][k_r]; pivot++);
      if (pivot < n) {
         if (pivot != k_r) {
            for (j = k_c; j < n; j++)
               swap(A[pivot][j], A[k_r][j]);
            swap(b[pivot], b[k_r]); sign *= -1;
         }
         for (int i = k_r +1; i < n; i++) {
            for (int j = k_c; j < n; j++)
   A[i][j] = (A[k_r][ k_c]*A[i][j]-A[i][k_c]*A[k_r][j])/d;
            b[i] = (A[k_r][k_c]*b[i]-A[i][k_c]*b[k_r])/d;
         }
         if (d) d = A[k_r][k_c];
         k_r++;
      } else d = 0;
   }
   if (!d) {
      for (int k = k_r; k < n; k++)
         if (b[k]) return 0; // inconsistent system
      return 0; // multiple solutions
   }
   for (int k = n-1; k >= 0; k--) {
      x_star[k] = sign*d*b[k];
      for (j = k+1; j < n; j++) x_star[k] -= A[k][j]* x_star[j];
      x_star[k] /= A[k][k];
   }
   return sign*d;
}
```

Recursion

Difference Equation “Y(n) = aY(n-1) + bY(n-2)”

```
// Y(n) = aY(n-1) + bY(n-2) : Y(0) = y0 ... Y(1) = y1

double Y0=1;double Y1=1;double A=1;double B=1;
double C1;double C2;double R1;double R2;double R;double Theta;int
Flag;

void init(){ // Set Global Variables
   A/=2; B*=-1;
   if(B == A*A){
      Flag=0; R=A;
      C1 = Y0; C2 = (Y1 - C1*R)/R;
   }
   else if(B < A*A){
      Flag=1; R1=A-sqrt(A*A - B); R2=A+sqrt(A*A - B);
      C1= (Y1 - Y0*R2) / (R1 - R2); C2= Y0 - C1;
   }
   else{
      Flag=-1; R=sqrt(B); Theta=atan(sqrt(B-A*A) / A);
      C1=Y0; C2= (Y1 - C1*R*cos(Theta)) / (R*sin(Theta));
   }
}

double Y(int n){
   if(Flag== 0)return C1*Pow(R,n) + C2*n*Pow(R,n);
   if(Flag== 1)return C1*Pow(R1,n) + C2*Pow(R2,n);
   if(Flag==-1)return C1*Pow(R,n)*cos(Theta*n) +
                      C2*Pow(R,n)*sin(Theta*n);
}
```

```
// bobocel is the 0'th suffix
//  obocel is the 5'th suffix
//   bocel is the 1'st suffix
//    ocel is the 6'th suffix
//     cel is the 2'nd suffix
//      el is the 3'rd suffix
//       l is the 4'th suffix
P.back() = [0,5,1,6,2,3,4]
LCP(0,2) = 2 → “bo”
```

String

KMP (Check Dynamic Programming Section)

Suffix Array O(L lg²L)

```
struct SuffixArray {
   const int L;
   string s;
   VVI P;
   VSt M;

   SuffixArray(const string &s) : L(s.length()), s(s), P(1,
vector<int>(L, 0)), M(L) {
      int i;
      FOR(i,L) P[0][i] = int(s[i]);
      for (int skip = 1, level = 1; skip < L; skip *= 2, level++){
         P.push_back(VI(L, 0));
         FOR(i,L)
            M[i] = St(II(P[level-1][i], i + skip < L ?
                  P[level-1][i + skip] : -1000), i);
         sort(M.begin(), M.end());
         FOR(i,L)
            P[level][M[i].second] = (i > 0 && M[i].first ==
               M[i-1].first) ? P[level][M[i-1].second] : i;
      }
   }

   VI GetSuffixArray() {return P.back();}

   // longest common prefix of s[i...L-1] and s[j...L-1]
   int LongestCommonPrefix(int i, int j) {
      int len = 0;
      if (i == j) return L - i;
      for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
         if (P[k][i] == P[k][j]) {
            i += 1 << k;    j += 1 << k;    len += 1 << k;
         }
      }
      return len;
   }
};
```

Lex Least Rotation + Smallest Period O(N)

```
// Find lex least rotation of a string,
// and smallest period of a string: O(n)
// pos = start of lex least rotation, period = the period
void compute(string s, int &pos, int &period) {
   s += s;
   int len=s.length(), i=0, j=1;
   for (int k = 0; i+k < len && j+k < len; k++) {
      if (s[i+k] > s[j+k]) {i = max(i+k+1, j+1); k = -1;}
      else if (s[i+k] < s[j+k]){j = max(j+k+1, i+1); k = -1;}
   }
   pos = min(i, j);
   period = (i > j) ? i - j : j - i;
}
```

Extra

Hash Function

32-bit

```
int hash_init() {
   return 2166136261;
}

int hash_add(int hash, short c) {
   hash ^= c;
   hash *= 16777619;
   return hash;
}
```

64-bit

```
long long hash_init() {
   return 14695981039346656037ULL;
}

long long hash_add(long long hash, short c) {
   hash ^= c;
   hash *= 1099511628211ULL;
   return hash;
}
```

Knight's Moves

```
int N;
const long long inf = 1e17;
long long Hx[15],Hy[15],Tx[15],Ty[15];
long long dist[15][15];
long long mc[15];
long long cc[15];
long long cccnt;

long long Dis(long long x,long long y){
   long long ret;

   if(x<0) x *= -1;
   if(y<0) y *= -1;
   if(x>y) swap(x,y);

   if(x==0 && y==0) return 0;
   if(x==0 && y==1) return 3;
   if(x==0 && y==2) return 2;
   if(x==0 && y==3) return 3;
   if(x==1 && y==1) return 2;
   if(x==1 && y==2) return 1;
   if(x==2 && y==2) return 4;
   if(x==2 && y==3) return 3;

   if(x<=y/2){
      y -= 2*x;
      ret = (y/4)*2 + x + y%4;
   }
   else{
      ret = x+y;
      ret = ret/3 + ret%3;
   }
   return ret;
}
```

KD Tree

```
// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
   ntype x, y;
   point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

bool operator==(const point &a, const point &b){
   return a.x == b.x && a.y == b.y;}

// sorts points on x-coordinate
bool on_x(const point &a, const point &b){
   return a.x < b.x;}

// sorts points on y-coordinate
bool on_y(const point &a, const point &b){
   return a.y < b.y;}

// squared distance between points
ntype pdist2(const point &a, const point &b){
   ntype dx = a.x-b.x, dy = a.y-b.y;
   return dx*dx + dy*dy;}

// bounding box for a set of points
struct bbox{
   ntype x0, x1, y0, y1;

   bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

   // computes bounding box from a bunch of points
   void compute(const vector<point> &v) {
      for (int i = 0; i < v.size(); ++i) {
         x0 = min(x0, v[i].x);   x1 = max(x1, v[i].x);
         y0 = min(y0, v[i].y);   y1 = max(y1, v[i].y);
      }
   }

   // squared distance between a point and this bbox, 0 if inside
   ntype distance(const point &p) {
      if (p.x < x0) {
         if (p.y < y0)        return pdist2(point(x0, y0), p);
         else if (p.y > y1)   return pdist2(point(x0, y1), p);
         else                 return pdist2(point(x0, p.y), p);
      }
      else if (p.x > x1) {
         if (p.y < y0)        return pdist2(point(x1, y0), p);
         else if (p.y > y1)   return pdist2(point(x1, y1), p);
         else                 return pdist2(point(x1, p.y), p);
      }
      else {
         if (p.y < y0)        return pdist2(point(p.x, y0), p);
         else if (p.y > y1)   return pdist2(point(p.x, y1), p);
         else                 return 0;
      }
   }
};

// stores a single node of the kd-tree, either internal or leaf
struct kdnode{
   bool leaf;      // true if this is a leaf node (has one point)
   point pt;       // the single point of this is a leaf
   bbox bound;     // bounding box for set of points in children

   kdnode *first, *second; // two children of this kd-node

   kdnode() : leaf(false), first(0), second(0) {}
   ~kdnode() { if (first) delete first; if (second) delete
second;}

   // intersect a point with this node (returns squared distance)
   ntype intersect(const point &p) { return bound.distance(p);}

   // recursively builds a kd-tree from a given cloud of points
   void construct(vector<point> &vp)
   {
      // compute bounding box for points at this node
      bound.compute(vp);

      // if we're down to one point, then we're a leaf node
      if (vp.size() == 1) {
         leaf = true;
         pt = vp[0];
      }
```

```
      else {
         // split on x if the bbox is wider than high (not best
heuristic...)
         if (bound.x1-bound.x0 >= bound.y1-bound.y0)
            sort(vp.begin(), vp.end(), on_x);
         // otherwise split on y-coordinate
         else
            sort(vp.begin(), vp.end(), on_y);

         // divide by taking half the array for each child
         // (not best performance if many duplicates in the
middle)
         int half = vp.size()/2;
         vector<point> vl(vp.begin(), vp.begin()+half);
         vector<point> vr(vp.begin()+half, vp.end());
         first = new kdnode();   first->construct(vl);
         second = new kdnode();  second->construct(vr);
      }
   }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree{
   kdnode *root;

   // constructs a kd-tree from a points (copied here, as it
sorts them)
   kdtree(const vector<point> &vp){
      vector<point> v(vp.begin(), vp.end());
      root = new kdnode();
      root->construct(v);
   }
   ~kdtree() { delete root; }

   // recursive search method returns squared distance to nearest
point
   ntype search(kdnode *node, const point &p){
      if (node->leaf) {
      // commented special case tells a point not to find itself
         //          if (p == node->pt) return sentry;
         //          else
         return pdist2(p, node->pt);
      }

      ntype bfirst = node->first->intersect(p);
         ntype bsecond = node->second->intersect(p);

// choose the side with the closest bounding box to search first
// (note that the other side is also searched if needed)
         if (bfirst < bsecond) {
            ntype best = search(node->first, p);
            if (bsecond < best)
               best = min(best, search(node->second, p));
            return best;
         }
         else {
            ntype best = search(node->second, p);
            if (bfirst < best)
               best = min(best, search(node->first, p));
            return best;
         }
      }

      // squared distance to the nearest
      ntype nearest(const point &p){return search(root, p);
      }
};

int main(){
   // generate some random points for a kd-tree
   vector<point> vp;
   for (int i = 0; i < 100000; ++i) {
      vp.push_back(point(rand()%100000, rand()%100000));
   }
   kdtree tree(vp);

   // query some points
   FOR(i,10){
      point q(rand()%100000, rand()%100000);
      cout<<"Closest squared distance to ("<<q.x<<", "<<q.y<< ")"
         << " is " << tree.nearest(q) << endl;
   }

   return 0;
}
```

Die

```
struct Die{
#define T s[0]
#define N s[1]
#define E s[2]
#define W s[3]
#define S s[4]
#define B s[5]
#define CYC( a, b ) t = T; T = a; a = B; B = b; b = t; break;

   string s;

   Die( string ss ) : s( ss ) {}

   void roll( char d ){
      char t;
      switch(d){
      case 'n': CYC( S, N );
      case 'e': CYC( W, E );
      case 'w': CYC( E, W );
      case 's': CYC( N, S );
      }
   }

   char get(char d){
      switch(d){
      case 't': return T;
      case 'n': return N;
      case 'e': return E;
      case 'w': return W;
      case 's': return S;
      case 'b': return B;
      }
      return 0;
   }

#undef T
#undef N
#undef E
#undef W
#undef S
#undef B
#undef CYC
};
```