Maximum Matching in Bipartite Graph

```
const int MAXN1 = 50000; const int MAXN2 = 50000;const int MAXM = 150000;
int n1, n2, edges, last[MAXN1], prev[MAXM], head[MAXM];
int matching[MAXN2], dist[MAXN1], Q[MAXN1];
bool used[MAXN1], vis[MAXN1];
void init(int _n1, int _n2) {
	n1 = _n1;
	n2 = _n2;
	edges = 0;
	fill(last, last + n1, -1);
}
void addEdge(int u, int v) { //use this in main function to build the graph
	head[edges] = v;
	prev[edges] = last[u];
	last[u] = edges++;
}
void bfs() {
	fill(dist, dist + n1, -1);
	int sizeQ = 0;
	for (int u = 0; u < n1; ++u) {
		if (!used[u]) {
			Q[sizeQ++] = u;
			dist[u] = 0;
		}
	}
	for (int i = 0; i < sizeQ; i++) {
		int u1 = Q[i];
		for (int e = last[u1]; e >= 0 ; e = prev[e]) {
			int u2 = matching[head[e]];
			if (u2 >= 0 && dist[u2] < 0) {
				dist[u2] = dist[u1] + 1;
				Q[sizeQ++] = u2;
			}
		}
	}
}
bool dfs(int u1) {
	vis[u1] = true;
	for (int e = last[u1]; e >= 0; e = prev[e]) {
		int v = head[e];
		int u2 = matching[v];
		if (u2 < 0 || !vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2)) {
			matching[v] = u1;
			used[u1] = true;
			return true;
		}
	}
	return false;
}
int maxMatching() {
	fill(used, used + n1, false);
	fill(matching, matching + n2, -1);
	for (int res = 0;;) {
		bfs();
		fill(vis, vis + n1, false);
		int f = 0;
		for (int u = 0; u < n1; ++u)
			if (!used[u] && dfs(u))
				++f;
		if (!f)
			return res;
		res += f;
	}
}
```

Dijkstra (fast)

```
typedef pair<int,int> PII;
```

```
int main(){

  int N, s, t;
  scanf ("%d%d%d", &N, &s, &t);
  vector<vector<PII> > edges(N);
  for (int i = 0; i < N; i++){
    int M;
    scanf ("%d", &M);
    for (int j = 0; j < M; j++){
      int vertex, dist;
      scanf ("%d%d", &vertex, &dist);
      edges[i].push_back (make_pair (dist, vertex)); // note order of arguments here
    }
  }
  // use priority queue in which top element has the "smallest" priority
  priority_queue<PII, vector<PII>, greater<PII> > Q;
  vector<int> dist(N, INF), dad(N, -1);
  Q.push (make_pair (0, s));
  dist[s] = 0;
  while (!Q.empty()){
    PII p = Q.top();
    if (p.second == t) break;
    Q.pop();

    int here = p.second;
    for (vector<PII>::iterator it=edges[here].begin(); it!=edges[here].end(); it++){
      if (dist[here] + it->first < dist[it->second]){
        dist[it->second] = dist[here] + it->first;
        dad[it->second] = here;
        Q.push (make_pair (dist[it->second], it->second));
      }
    }
  }

  printf ("%d\n", dist[t]);
  if (dist[t] < INF)
    for(int i=t;i!=-1;i=dad[i])
      printf ("%d%c", i, (i==s?'\n':' '));

  return 0;
}
```

Miller Rabin Primality Test

```
const int maxIter = 10;
bool isPrime(unsigned long long N)
{
        if(N < 2) return false;
        if( N % 2 == 0) return N == 2;
        if( N % 3 == 0) return N == 3;
        if( N % 5 == 0) return N == 5;
        if( N % 7 == 0) return N == 7;
        int d = 0;
        long long  odd = N - 1;
        while( (odd & 1) == 0)
         {
                d++;
                odd>>= 1;
         }

        for(int i = 0; i < maxIter; i++)
         {
                long long a = rand() % ( N - 1) + 1;          // a is random number from
1 to N -1
                long long mod = modulo( a, odd, N);           //(a^odd)%N
                bool passes = ( mod == 1 || mod == N -1 );
```

```
                for(int r = 1; r < d && !passes; r ++)
                  {
                        mod = mulmod( mod, mod, N);    //(a*b)%N
                        passes = passes || mod == N - 1;
                  }

                if(!passes)
                        return false;
         }
        return true;
}
```

Modular Multiplicative Inverse

```
int modmulinverse(int a,int m)
{
    int x = 0,y = 1,u = 1,v = 0;
    int e = m,f = a;
    int c,d,q,r;
    while(f != 1)
    {
        q = e/f;
        r = e%f;
        c = x-q*u;        d = y-q*v;
        x = u;        y = v;
        u = c;        v = d;
        e = f;        f = r;
    }
    u = (u+m)%m;
    return u;
}
```

Binary Index Tree

`int read(int idx){`
`        int sum = 0;`
`        while (idx > 0){`
`                sum += tree[idx];`
`                idx -= (idx & -idx);`
`        }`
`        return sum;`
`}`
	`void update(int idx ,int val){`
	`        while (idx <= MaxVal){`
	`                tree[idx] += val;`
	`                idx += (idx & -idx);`
	`        }`
	`}`

Stable Marriage

```
int m[501][501];
int w[501][501];
int m_explored[501];
int w_engaged[501];
queue<int> unmarried;
int main() {
        int T;
        T=read_int();
        while(T) {
                int n=read_int();
                int i=1;

                while(i<=n) {
                        w_engaged[i]=0;
                        int j=1;
                        read_int();
                        while(j<=n) {
                                w[i][read_int()]=j;
                                j++;
                        }
                        i++;
                }
                i=1;
                while(i<=n) {
                        unmarried.push(i);
```

```
				m_explored[i]=0;
				int j=1;
				read_int();
				while(j<=n) {
					m[i][j]=read_int();
					j++;
				}
				i++;
			}
			while(!unmarried.empty()) {
				int man=unmarried.front();
				unmarried.pop();

				while(1) {
					m_explored[man]++;
					int next_w=m[man][m_explored[man]];
					if(w_engaged[next_w]) {
						if(w[next_w][w_engaged[next_w]]<w[next_w][man]) {
							continue;
						} else {
							unmarried.push(w_engaged[next_w]);

							w_engaged[next_w]=man;
							break;
						}
					} else {
						w_engaged[next_w]=man;
						break;
					}
				}
			}
			i=1;
			while(i<=n) {
				printf("%d %d\n", w_engaged[i], i);

				i++;
			}
			T--;
		}
		return 0;
}
```

Maximum flow

```
#define MAX 200
int edge[MAX][MAX];
int parent[MAX];

bool search_path(int N,int s, int t)
{
bool visited[N];
int i;
for(i=0 ; i< N ; i++)
visited[i] = false;
queue<int> myqueue;
myqueue.push(s);
parent[s] = -1;
visited[s] = true;
while(!(myqueue.empty()))
{
int f = myqueue.front();
myqueue.pop();
for(int i = 0; i< N ; i++)
{
if(edge[f][i] > 0 && visited[i] == false)
{
myqueue.push(i);
```

```
parent[i] = f;
visited[i] = true;
}
}
}
return visited[t];
}

int ford_fulkerson(int N, int flow,int s, int t)
{
int min_wt;
while(search_path(N,s,t))
{
   min_wt = 99999;
int tt = t;
while(tt != s)
{
int temp = parent[tt];
if(min_wt > edge[temp][tt])
min_wt = edge[temp][tt];
tt = temp;
}
tt = t;
while(tt != s)
{
int temp = parent[tt];
edge[temp][tt] -= min_wt ;
edge[tt][temp] += min_wt ;
tt = temp;
}
flow = flow + min_wt ;
}
return flow;
}
```

ConvexHull.cc 7/27

```
struct Point {
    long long x;
    long long y;
    double angle;
} p [100 + 5], hull [100 + 5];

double angle (long long x, long long y)
{
    double theta = atan2 (fabs (y), fabs (x));
    if ( x >= 0 ) return theta;
    return pi - theta;
}

bool cmp (Point a, Point b)
{
    if (fabs (a.angle - b.angle) < 1e-6 )
        return (a.x * a.x + a.y * a.y) < (b.x * b.x + b.y * b.y);
    return a.angle < b.angle;
}

bool isInRight (Point a, Point b, Point c)
{
    if ( c.x * (a.y - b.y) + c.y * (b.x - a.x) + (a.x * b.y - a.y * b.x) < 0 )
        return true;
    return false;
}

double square (double a)
{
```

```
    return a * a;
}

int main ()
{
    int testCase;
    scanf ("%d", &testCase);
    while ( testCase-- ) {
        long long initialLength=0;
        int totalPoints;
        cin >>totalPoints;
        for ( int i = 0; i < totalPoints; i++ )
            cin >> p [i].x >> p [i].y;

        // special case if only 1 totalPoints
        if ( totalPoints == 1 ) {
            printf ("%.5lf\n", (double) initialLength);
            continue;
        }

        int id = 0;

        for ( int i = 1; i < totalPoints; i++ )
            if ( p [i].y < p [id].y ) id = i;

        swap (p [id], p [0]);

        // scaling down respective to id
        for ( int i = totalPoints - 1; i >= 0; i-- ) {
            p [i].x -= p [0].x;
            p [i].y -= p [0].y;
        }

        // measure angle according to id
        for ( int i = 1; i < totalPoints; i++ )
            p [i].angle = angle (p [i].x, p [i].y);

        sort (p + 1, p + totalPoints, cmp);

        hull [0] = p [0];
        hull [1] = p [1];
        int top = 1;

        for ( int i = 2; i < totalPoints; i++ ) {
            while (isInRight (hull [top - 1], hull [top], p [i]))
                top--;
            hull [++top] = p [i];
        }

        double finalLength = 0;

        // perimeter of convex hull
        for ( int i = 1; i <= top; i++ )
            finalLength += sqrt (square (hull [i - 1].x - hull [i].x) + square (hull [i
- 1].y - hull [i].y));

        finalLength += sqrt (square (hull [top].x - hull [0].x) + square (hull [top].y
- hull [0].y));

        if ( finalLength < initialLength ) finalLength = initialLength;

        printf ("%.2lf\n", finalLength);

        for(int i=0;i<totalPoints;i++)
                       cout<<hull[i].x<<" "<<hull[i].y<<"   ";
    }
```

```
        return 0;
}
```

Geometry.cc 8/27

```
double INF = 1e100;
double EPS = 1e-12;
struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,   y*c  ); }
  PT operator / (double c)     const { return PT(x/c,   y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")"; }

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}
```

```
// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b with line passing through c and
d, assuming unique intersection exists; for segment intersection, check if segments
intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);}

PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}
// compute intersection of line through points a and b with circle centered at c with
rad r >0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
```

```
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}
// compute intersection of circle centered at a with rad r with circle centered at b
with rad R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}
// This code computes the area or centroid of a (possibly nonconvex)polygon, assuming
that the coordinates are listed in a clockwise or counterclockwise fashion.  Note that
the centroid is often known as the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}
// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

int main() {
    cerr << RotateCCW90(PT(2,5)) << endl;  // expected: (-5,2)
  cerr << RotateCW90(PT(2,5)) << endl;// expected: (5,-2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;// expected: (-5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;// expected: (5,2)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;
  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;
```

```
// expected: 1 0 1
cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
     << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
     << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

// expected: 0 0 1
cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
     << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
     << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

// expected: 1 1 1 0
cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
     << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
     << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
     << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

// expected: (1,2)
cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

// expected: (1,1)
cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

vector<PT> v;
v.push_back(PT(0,0));
v.push_back(PT(5,0));
v.push_back(PT(5,5));
v.push_back(PT(0,5));

// expected: 1 1 1 0 0
cerr << PointInPolygon(v, PT(2,2)) << " "
     << PointInPolygon(v, PT(2,0)) << " "
     << PointInPolygon(v, PT(0,2)) << " "
     << PointInPolygon(v, PT(5,2)) << " "
     << PointInPolygon(v, PT(2,5)) << endl;

// expected: 0 1 1 1 1
cerr << PointOnPolygon(v, PT(2,2)) << " "
     << PointOnPolygon(v, PT(2,0)) << " "
     << PointOnPolygon(v, PT(0,2)) << " "
     << PointOnPolygon(v, PT(5,2)) << " "
     << PointOnPolygon(v, PT(2,5)) << endl;

// expected: (1,6)
//           (5,4) (4,5)
//           blank line
//           (4,5) (5,4)
//           blank line
//           (4,5) (5,4)
vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

// area should be 5.0
// centroid should be (1.1666666, 1.166666)
PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
vector<PT> p(pa, pa+4);
PT c = ComputeCentroid(p);
cerr << "Area: " << ComputeArea(p) << endl;
```

```
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

Euclid.cc 12/27

```
// This is a collection of useful code for solving problems that
// involve modular linear equations.  Note that all of the// returns d = gcd(a,b);
finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}
// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}
// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}
// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[i]).  Return (z,M).  On
// failure, M = -1.  Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.first, ret.second, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}
// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

GaussJordan.cc 13/27

```
// Gauss-Jordan elimination with full pivoting.

//
```

```
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
        if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}

int main() {
  const int n = 4;
  const int m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++) {
```

```
    a[i] = VT(A[i], A[i] + n);
    b[i] = VT(B[i], B[i] + m);
  }

  double det = GaussJordan(a, b);

  // expected: 60
  cout << "Determinant: " << det << endl;
  cout << "Inverse: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

  // expected: 1.63333 1.3
  //           -0.166667 0.5
  //           2.36667 1.7
  //           -1.85 -1.35
  cout << "Solution: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
      cout << b[i][j] << ' ';
    cout << endl;
  }
}
```

SCC.cc 18/27

```
#include<memory.h>
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x)
{
  int i;
  v[x]=true;
  for(i=sp[x];i;i=e[i].nxt) if(!v[e[i].e]) fill_forward(e[i].e);
  stk[++stk[0]]=x;
}
void fill_backward(int x)
{
  int i;
  v[x]=false;
  group_num[x]=group_cnt;
  for(i=spr[x];i;i=er[i].nxt) if(v[er[i].e]) fill_backward(er[i].e);
}
void add_edge(int v1, int v2) //add edge v1->v2
{
  e [++E].e=v2; e [E].nxt=sp [v1]; sp [v1]=E;
  er[  E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
}
void SCC()
{
  int i;
  stk[0]=0;
  memset(v, false, sizeof(v));
  for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
  group_cnt=0;
  for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++; fill_backward(stk[i]);}
}
```

SuffixArray.cc 19/27

```
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

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

  vector<int> GetSuffixArray() { return P.back(); }

  // returns the length of the 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;
  }
};

}
```

UnionFind.cc 21/27

```
//union-find set: the vector/array contains the parent of each node
int find(vector <int>& C, int x){return (C[x]==x) ? x : C[x]=find(C, C[x]);} //C++
int find(int x){return (C[x]==x)?x:C[x]=find(C[x]);} //C
```

KMP

```
function build_failure_function(pattern[])
{
  // let m be the length of the pattern
  F[0] = F[1] = 0; // always true
  for(i = 2; i <= m; i++) {
      // j is the index of the largest next partial match
      // (the largest suffix/prefix) of the string under
      // index i - 1
      j = F[i - 1];
      for( ; ; ) {
          // check to see if the last character of string i -
          // - pattern[i - 1] "expands" the current "candidate"
          // best partial match - the prefix under index j
          if(pattern[j] == pattern[i - 1]) {
```

```
            F[i] = j + 1; break;
         }
         // if we cannot "expand" even the empty string
         if(j == 0) { F[i] = 0; break; }
         // else go to the next best "candidate" partial match
         j = F[j];
      }
   }
}
function Knuth_Morris_Pratt(text[], pattern[])
{
   // let n be the size of the text, m the
   // size of the pattern, and F[] - the
   // "failure function"

   build_failure_function(pattern[]);

   i = 0; // the initial state of the automaton is
   // the empty string

   j = 0; // the first character of the text

   for( ; ; ) {
      if(j == n) break; // we reached the end of the text

      // if the current character of the text "expands" the
      // current match
      if(text[j] == pattern[i]) {
         i++; // change the state of the automaton
         j++; // get the next character from the text
         if(i == m) // match found
      }

      // if the current state is not zero (we have not
      // reached the empty string yet) we try to
      // "expand" the next best (largest) match
      else if(i > 0) i = F[i];

      // if we reached the empty string and failed to
      // "expand" even it; we go to the next
      // character from the text, the state of the
      // automaton remains zero
      else j++;
   }
}
```

KRUSKAL

```
using namespace std;
vector< pair<int, edge > > Graph,MST;
int n,e,parent[MAX];
void initialize(int n)
{
    for(int i=1; i<=n; i++)
    {
        parent[i] = i;
    }
    MST.clear();
    Graph.clear();
}
```

```
//finding parent with path compression
int findSet(int x, int *parent)
{
    if(x!=parent[x])
    {
        parent[x] = findSet(parent[x],parent);
    }
    return parent[x];
}
void kruskal()
{
    sort(Graph.begin(),Graph.end());
    int total = 0;
    for(int i=0; i<e; i++)
    {
        int pu = findSet(Graph[i].second.first,parent);
        int pv = findSet(Graph[i].second.second,parent);
        if(pu!=pv)
        {
            total+=Graph[i].first;
            MST.push_back(Graph[i]);
            parent[pu] = parent[pv];
        }
    }
    cout<<total<<"\n";
}
int main()
{
    cin>>n>>e;
    initialize(n);
    int u,v,w;
    for(int i=0; i<e; i++)
    {
        cin>>u>>v>>w;
        Graph.push_back(pair<int, edge>(w,edge(u,v)));
    }
    kruskal();
    return 0;
}
```

Baby step Gaint Step Algorithm:

To calculate value of x such that

Output: A value *x* satisfying (mod N)

1. m ← Ceiling(√n)
2. For all j where $0 \le j < m$:
    1. Compute αj and store the pair (j, αj) in a table. (See section "In practice")
3. Compute $\alpha -m$.
4. γ ← β. (set γ = β)
5. For i = 0 to (m − 1):
    1. Check to see if γ is the second component (αj) of any pair in the table.
    2. If so, return $im + j$.

If not, γ ← γ • α−m

Pollard Factorization

def pollardRho(N):

    if N%2==0:

```
        return 2
    x = random.randint(1, N-1)
    y = x
    c = random.randint(1, N-1)
    g = 1
    while g==1:
        x = ((x*x)%N+c)%N
        y = ((y*y)%N+c)%N
        y = ((y*y)%N+c)%N
        g = gcd(abs(x-y),N)
    return g
```

Method for Finding the Lowest Common Ancestor of 2 nodes in a tree.

```
void process(int N) //N nodes
      memset(Root,-1,sizeof(Root)); //Root[N][log N]
      for(int i=1;i<=N;i++) Root[i][0]=parent[i]; //stores the 2i
      for(int i=1;(1<<i) <= N; i++)
      for(int j=1;j<=N;j++)
      if(Root[j][i-1]!=-1)
      Root[j][i]=Root[Root[j][i-1]][i-1];

int lca(int p,int q) //Fnds the LCA of 2 nodes
      int temp;
      if(depth[p]>depth[q])
            int steps=store[depth[q]];
      for(int i=steps;i>=0;i--) //putting p & q on same level
            swap(p,q);
      if(depth[q]-(1<<i) >= depth[p])
      q=Root[q][i];
      if(p==q) return p;
      for(int i=steps;i>=0;i--)
            if(Root[p][i]!=Root[q][i])
                  p=Root[p][i],q=Root[q][i];
      return parent[p];

void init() //store[i]= int( log2i )
      store[0]=0; store[1]=0; store[2]=1;
      int cmp=4;
      for(int i=3;i<1008;i++)
```

```
		if(cmp>i) store[i]=store[i-1];
		else
			store[i]=store[i-1]+1;
			cmp<<=1;
```

```
//Maximum cost weighted matching
//Lifted from topcoder, tested on SPOJ:GREED
# define MAXN 500
# define MAXD 1000
# define INF 1000000000000ll

int N,R;
int matched;

long long adjmat[MAXN][MAXN];
long long l1[MAXN],l2[MAXN],sl[MAXN];
int m1[MAXN],m2[MAXN],bfsq[MAXN],par[MAXN],sx[MAXN];
char S[MAXN],T[MAXN];

void updatetree(int v,int prev)
{
  S[v]=true;
  par[v]=prev;
  for(int i=0;i<N;i++)
  {
      if(l1[v]+l2[i]-adjmat[v][i]<sl[i])
      {
          sl[i]=l1[v]+l2[i]-adjmat[v][i];
          sx[i]=v;
      }
  }
}

void updatelabels()
{
  long long delta=1ll<<62;
  for(int i=0;i<N;i++)
      if((!T[i])&&(delta>sl[i]))
          delta=sl[i];
  for(int i=0;i<N;i++)
  {
      if(S[i])l1[i]-=delta;
      if(T[i])l2[i]+=delta;
      else sl[i]-=delta;
  }
}

void augment()
{
  if(matched==N)return;
  int qmax=0,qpos=0,x,y,root;
  memset(S,0,N);
  memset(T,0,N);
  memset(par,-1,N<<2);
  for(x=0;x<N;x++)
      if(m1[x]==-1)
      {
```

```
            bfsq[qmax++]=root=x;
            S[x]=1;
            par[x]=-2;
            break;
        }
    for(y=0;y<N;y++)
    {
        sl[y]=l1[root]+l2[y]-adjmat[root][y];
        sx[y]=root;
    }
    while(1)
    {
        while(qpos<qmax)
        {
            x=bfsq[qpos++];
            for(y=0;y<N;y++)
            {
                if((adjmat[x][y]==l1[x]+l2[y])&&(!T[y]))
                {
                    if(m2[y]==-1)
                        break;
                    T[y]=1;
                    bfsq[qmax++]=m2[y];
                    updatetree(m2[y],x);
                }
            }
            if(y<N)break;
        }
        if(y<N)break;
        updatelabels();
        qmax=qpos=0;
        for(y=0;y<N;y++)
        {
            if((!T[y])&&(sl[y]==0))
            {
                if(m2[y]==-1)
                {
                    x=sx[y];
                    break;
                }
                else
                {
                    T[y]=1;
                    if(!S[m2[y]])
                    {
                        bfsq[qmax++]=m2[y];
                        updatetree(m2[y],sx[y]);
                    }
                }
            }
        }
        if(y<N)
            break;
    }
    if(y<N)
    {
        matched++;
        for(int cx=x,cy=y,ty;cx!=-2;cx=par[cx],cy=ty)
        {
            ty = m1[cx];
```

```
            m2[cy] = cx;
            m1[cx] = cy;
        }
        augment();
    }
}

//Way shorter but slower Hungarian Algorithm Use if Number of vertices is small
//Warning: Vertices to be 1-indexed Tested on SPOJ: GREED
const int INF=(1<<30)-1;
int a[505][505];
int Hungarian(int n,int m){
  vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
  for (int i=1; i<=n; ++i) {
      p[0] = i;
      int j0 = 0;
      vector<int> minv (m+1, INF);
      vector<char> used (m+1, false);
      do {
          used[j0] = true;
          int i0 = p[j0], delta = INF, j1;
          for (int j=1; j<=m; ++j)
              if (!used[j]) {
                  int cur = a[i0][j]-u[i0]-v[j];
                  if (cur < minv[j])
                      minv[j] = cur, way[j] = j0;
                  if (minv[j] < delta)
                      delta = minv[j], j1 = j;
              }
          for (int j=0; j<=m; ++j)
              if (used[j])
                  u[p[j]] += delta, v[j] -= delta;
              else
                  minv[j] -= delta;
          j0 = j1;
      } while (p[j0] != 0);
      do {
          int j1 = way[j0];
          p[j0] = p[j1];
          j0 = j1;
      } while (j0);
  }
  return -v[0];
}
```

Closest pair

```
typedef pair<long long, long long> pairll;

int n;

pairll pnts [100000];

set<pairll> box;

double best;

int compx(pairll a, pairll b) { return a.px<b.px; }

int main () {

      scanf("%d", &n);

      for (int i=0;i<n;++i) scanf("%lld %lld", &pnts[i].px, &pnts[i].py);

      sort(pnts, pnts+n, compx);
```

```
	best = 1500000000; // INF
	box.insert(pnts[0]);
	int left = 0;
	for (int i=1;i<n;++i) {
		while (left<i && pnts[i].px-pnts[left].px > best) box.erase(pnts[left++]);
		for (typeof(box.begin()) it=box.lower_bound(make_pair(pnts[i].py-best,
pnts[i].px-best));
		it!=box.end() && pnts[i].py+best>=it->py; it++)
		best = min(best, sqrt(pow(pnts[i].py - it->py, 2.0)+pow(pnts[i].px - it-
>px, 2.0)));
	box.insert(pnts[i]);
	printf("%.2f\n", best);
	}
	return 0;
}
```

Articulation Point

```
#define NIL -1
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // A dynamic array of adjacency lists
 void APUtil(int v, bool visited[], int disc[], int low[],
 int parent[], bool ap[]);
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // function to add an edge to graph
 void AP(); // prints articulation points
};
Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w);
 adj[w].push_back(v); // Note: the graph is undirected
}

// u --> The vertex to be visited next
```

```
// disc[] --> Stores discovery times of visited vertices

// ap[] --> Store articulation points

void Graph::APUtil(int u, bool visited[], int disc[], int low[], int parent[], bool
ap[])

{

 // A static variable is used for simplicity, we can avoid use of static // variable by
passing a pointer.


 static int time = 0;

 // Count of children in DFS Tree

 int children = 0;

 // Mark the current node as visited

  visited[u] = true;

 disc[u] = low[u] = ++time;

 // Go through all vertices aadjacent to this

 list<int>::iterator i;


 for (i = adj[u].begin(); i != adj[u].end(); ++i)

 {

 int v = *i; // v is current adjacent of u // If v is not visited yet, then make it a
child of u // in DFS tree and recur for it


 if (!visited[v])

 {

 children++;

 parent[v] = u;

 APUtil(v, visited, disc, low, parent, ap);

 // Check if the subtree rooted with v has a connection to // one of the ancestors of u

 low[u] = min(low[u], low[v]);

 // u is an articulation point in following cases // (1) u is root of DFS tree and has
two or more chilren.

 if (parent[u] == NIL && children > 1)

 ap[u] = true;

 // (2) If u is not root and low value of one of its child is more // than discovery
value of u.

 if (parent[u] != NIL && low[v] >= disc[u])

 ap[u] = true;

 }

 // Update low value of u for parent function calls.

 else if (v != parent[u])

 low[u] = min(low[u], disc[v]);

 }
```

```
}


// The function to do DFS traversal. It uses recursive function APUtil()


void Graph::AP()
{
 bool *visited = new bool[V];
 int *disc = new int[V];
 int *low = new int[V];
 int *parent = new int[V];
 bool *ap = new bool[V]; // To store articulation points
 // Initialize parent and visited, and ap(articulation point) arrays
 for (int i = 0; i < V; i++)
 {
 parent[i] = NIL;
 visited[i] = false;
 ap[i] = false;
 }


 // Call the recursive helper function to find articulation points // in DFS tree
rooted with vertex 'i'


 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 APUtil(i, visited, disc, low, parent, ap); // Now ap[] contains articulation points,
print them
 for (int i = 0; i < V; i++)
 if (ap[i] == true)
 cout << i << " ";
}


// Driver program to test above function


int main()
{
 // Create graphs given in above diagrams
 cout << "\nArticulation points in first graph \n";Graph g1(5);
 g1.addEdge(1, 0);g1.addEdge(0, 2);g1.addEdge(2, 1);g1.addEdge(0, 3);g1.addEdge(3,
4);g1.AP();
 cout << "\nArticulation points in second graph \n";
 Graph g2(4);g2.addEdge(0, 1);g2.addEdge(1, 2);g2.addEdge(2, 3);g2.AP();
```

```
 cout << "\nArticulation points in third graph \n";Graph g3(7);
 g3.addEdge(0, 1);g3.addEdge(1, 2);g3.addEdge(2, 0);g3.addEdge(1, 3);g3.addEdge(1, 4);
 g3.addEdge(1, 6);g3.addEdge(3, 5);g3.addEdge(4, 5);g3.AP();
 return 0;}
```

HIST2

```
int main(){
for(i=0 ; i< (1<<N) ; i++)
for(j=0 ; j< N ; j++)
      count[i][j]=fact[i][j] = 0;
      for (i= 0 ; i<N ; i++)
      {
            count[(1<<i)][i] = 1;
            fact[(1<<i)][i] = 2*A[i]+2;
      }
      for(i= 3 ; i< (1<<N)  ; i++)
      {
      for(j = 0 ; j<N ; j++)
      {
            if(i&(1<<j))
            {
                  long long int temp_max = 2*A[j]+2;
                  long long int c = 0;
                  for(k= 0 ; k < N ; k++)
                  {
                        if(((i - (1<<j))&(1<<k)) != 0)
                        {//do something}                    }
            fact[i][j] = temp_max;}}}}
```

JAVA Fast IO

```
BufferedReader fin = new BufferedReader (new InputStreamReader ( System . in ) ) ;
StringTokenizer toker ;
String line ;
while ( ( line = fin . readLine ( ) . trim ( ) ) != null )
      toker = new StringTokenizer ( line ) ;
      int m = Integer . parseInt ( toker . nextToken ( ) ) ;
      int n = Integer . parseInt ( toker . NextToken ( ) ) ;
```

Longest Increasing Subsequence

```
int _lis( int arr[], int n, int *max_ref)
{
```

```
    if(n == 1)
        return 1;
    int res, max_ending_here = 1; // length of LIS ending with arr[n-1]
    /* Recursively get all LIS ending with arr[0], arr[1] ... ar[n-2]. If
       arr[i-1] is smaller than arr[n-1], and max ending with arr[n-1] needs
       to be updated, then update it */
    for(int i = 1; i < n; i++)
    {        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }
    // Compare max_ending_here with the overall max. And update the    // overall max
if needed
    if (*max_ref < max_ending_here)
       *max_ref = max_ending_here;
    // Return length of LIS ending with arr[n-1]
    return max_ending_here;}
// The wrapper function for _lis()
int lis(int arr[], int n)
{    // The max variable holds the result
    int max = 1;
    // The function _lis() stores its result in max
    _lis( arr, n, &max );
    return max;
```