Maths - Numbers:

```
#include <bits/stdc++.h>
using namespace std;
#define INF (ll)(1e18+7)
#define INF1 (int)(1e9+7)
typedef long long ll;
typedef vector<int> vi;
typedef map<int, int> mii;
```

// Generate Primes < upperbound:

```
ll _sieve_size;
bitset<100000100> bs;
vi primes;

void sieve(ll upperbound) {
    _sieve_size = upperbound + 1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
        for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
        if (i!=2)
            primes.push_back((int)i);
    }

}

bool isPrime(ll N) {
    if (N <= _sieve_size) return bs[N];
    for (int i = 0; i < (int)primes.size()&&primes[i]*primes[i]<=N; i++)
        if (N % primes[i] == 0) return false;
    return true;
}

ex:     sieve((200000010/5)/3);
```

// Matrix Multiply/Power:`

```
ll** matrixMultiply(ll** m1, ll** m2, int n){
    ll **res = new ll*[n];
    for(int i(0); i < n; i++){
        res[i] = new ll[n];
    }

    for(int i(0); i < n; i++){
        for(int j(0); j < n; j++){
            res[i][j] = 0;
            for(int k(0); k < n; k++){
                res[i][j] += ((m1[i][k] + INF1) % INF1 * (m2[k][j] + INF1) % INF1) % INF1;
                res[i][j] = (res[i][j] + INF1) % INF1;
            }
        }
    }
    return res;
}
```

```
ll** matrixPower(ll** m, int n, ll pow){
    ll **ret = new ll*[n];
    for(int i(0); i < n; i++){
        ret[i] = new ll[n];
    }
    for(int i(0); i < n; i++){
        for (int j = 0; j < n; j++)
        {
            if ( i == j) ret[i][i] = 1;
            else ret[i][j] = 0;
        }
    }

    while (pow){
        if (pow & 1){
            ret = matrixMultiply(ret, m, n);
        }
        pow >>=1;
        m = matrixMultiply(m, m, n);
    }

    return ret;
}
```

Graphe:

// DFS:

```
#include <bits/stdc++.h>

using namespace std ;
int n,m;
int x,y;
vector < vector < int > > graph ;
vector <bool> visited;

void dfs(int pos){
    visited[pos] = true ;
    for(int i = 0 ; i < graph[pos].size() ; i++){
        int v = graph[pos][i] ;
        if(visited[v] == false)dfs(v) ;
    }
    return ;
}

int main(){
    cin >> n >> m ;
    graph.resize(n) ;

    while(m--){
        cin >> x >> y ;
        graph[x].push_back(y);
        graph[y].push_back(x); /// if the edges are directed then you simply remove this line
    }

    visited.resize(n,false) ; /// initialise all the edges as unvisited

    for(int i = 0 ; i < n ; i++){
        if(visited[i] == false)dfs(i) ;
    }
    return 0 ;
}
```

// Dijkstra :

```
#include <queue>
#include <stdio.h>

using namespace std;
const int INF = 2000000000;
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;
}
```

DisjV2:

```
# define INF 1e18
typedef pair<int,int> PII;
int main(){
  /*
   int n,m;
   cin>>n>>m;
```

```
 Graph g(n+1);
 for(int i=0;i<m;i++){
 int s,d,c;
 scanf("%d %d %d",&s,&d,&c);
 g.addEdge(s-1, d-1, c);
 }
 g.shortestPath(0, n-1);*/


int N, M;
int s=0;

scanf ("%d%d", &N, &M);
int t = N-1;
vector<vector<PII> > edges(N+1);
for (int j = 0; j < M; j++){
    int src, dest,dist;
    scanf ("%d%d%d", &src,&dest, &dist);
    edges[src-1].push_back (make_pair (dist, dest-1));
    edges[dest-1].push_back(make_pair(dist,src-1));// 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<long long> dist(N+2, INF);
vector<int> dad(N+2, -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 (int i=0;i<edges[here].size();i++){
        PII it = edges[here][i];

        if ((long long)(dist[here] + it.first) < dist[it.second]){
            dist[it.second] = (long long)(dist[here] + it.first);
            dad[it.second] = here;
            Q.push (make_pair(dist[it.second], it.second));
        }
    }
}
// path print maliku
vector<int> path;
if (dist[t] < INF){
    for(int i=t;i!=-1;i=dad[i])
        path.push_back(i+1);
    for(int i=path.size();i>0;i--)
        cout<<path[i-1]<<" ";
}
else cout<<"-1";



}
```

Géometrie:

```
#include <bits/stdc++.h>
using namespace std;

typedef complex<double> point;
#define sz(a) ((int)(a).size())
#define all(n) (n).begin(),(n).end()
#define EPS 1e-9
```

```
#define OO 1e9
#define X real()
#define Y imag()
#define vec(a,b) ((b)-(a))
#define polar(r,t) ((r)*exp(point(0,(t))))
#define angle(v) (atan2((v).Y,(v).X))
#define length(v) ((double)hypot((v).Y,(v).X))
#define lengthSqr(v) (dot(v,v))
#define dot(a,b) ((conj(a)*(b)).real())
#define cross(a,b) ((conj(a)*(b)).imag())
#define rotate(v,t) (polar(v,t))
#define rotateabout(v,t,a)  (rotate(vec(a,v),t)+(a))
#define reflect(p,m) ((conj((p)/(m)))*(m))
#define normalize(p) ((p)/length(p))
#define same(a,b) (lengthSqr(vec(a,b))<EPS)
#define mid(a,b) (((a)+(b))/point(2,0))
#define perp(a) (point(-(a).Y,(a).X))
#define colliner pointOnLine

enum STATE
{
    IN, OUT, BOUNDRY
};

bool intersect(const point &a, const point &b, const point &p,
const point &q,
               point &ret)
{

    //handle degenerate cases (2 parallel lines, 2 identical
lines,   line is 1 point)

    double d1 = cross(p - a, b - a);
    double d2 = cross(q - a, b - a);
    ret = (d1 * q - d2 * p) / (d1 - d2);
    if(fabs(d1 - d2) > EPS) return 1;
    return 0;
}

bool pointOnLine(const point& a, const point& b, const point& p)
{
    // degenerate case: line is a point
    return fabs(cross(vec(a,b),vec(a,p))) < EPS;
}

bool pointOnRay(const point& a, const point& b, const point& p)
{
    //IMP NOTE: a,b,p must be collinear
    //chech if it's in the same direction as the [a,b)
    return dot(vec(a,p), vec(a,b)) > -EPS;
}
```

```
bool pointOnSegment(const point& a, const point& b, const point&
p)
{
    if (same(a,b))
        return same(a,p);
    if(!colliner(a,b,p)) return 0;
    return pointOnRay(a, b, p) && pointOnRay(b, a, p);
}

double pointLineDist(const point& a, const point& b, const point&
p)
{
    // handle degenrate case: (a,b) is point

    return fabs(cross(vec(a,b),vec(a,p)) / length(vec(a,b)));
}

double pointSegmentDist(const point& a, const point& b, const
point& p)
{
    if (dot(vec(a,b),vec(a,p)) < EPS)
        return length(vec(a,p));
    if (dot(vec(b,a),vec(b,p)) < EPS)
        return length(vec(b,p));
    return pointLineDist(a, b, p);
}

int segmentLatticePointsCount(int x1, int y1, int x2, int y2)
{
    return abs(__gcd(x1 - x2, y1 - y2)) + 1;
}

double triangleAreaBH(double b, double h)
{
    return b * h / 2;
}

double triangleArea2sidesAngle(double a, double b, double t)
{
    return fabs(a * b * sin(t) / 2);
}

double triangleArea2anglesSide(double t1, double t2,
                               double s)
{
    return fabs(s * s * sin(t1) * sin(t2) / (2 * sin(t1 + t2)));
}

double triangleArea3sides(double a, double b, double c)
{
    double s((a + b + c) / 2);
    return sqrt(s * (s - a) * (s - b) * (s - c));
```

```
}

double triangleArea3points(const point& a, const point& b, const
point& c)
{
    return fabs(cross(a,b) + cross(b,c) + cross(c,a)) / 2;
}

//count interior Lattice points inside polygon (corner are already
lattice points)
int picksTheorm(int a, int b)
{
    // a area
    // b nbr of lattice points on boundary
    return a - b / 2 + 1;
}

//get angle opposite to side a
double cosRule(double a, double b, double c)
{
    // Handle denom = 0
    double res = (b * b + c * c - a * a) / (2 * b * c);
    if ( fabs(res-1)<EPS)
        res = 1;
    if (fabs(res+1)<EPS)
        res = -1;
    return acos(res);
}

double sinRuleAngle(double s1, double s2, double a1)
{
    // Handle denom = 0
    double res = s2 * sin(a1) / s1;
    if ( fabs(res-1)<EPS)
        res = 1;
    if ( fabs(res+1)<EPS)
        res = -1;
    return asin(res);
}

double sinRuleSide(double s1, double a1, double a2)
{
    // Handle denom = 0
    double res = s1 * sin(a2) / sin(a1);
    return fabs(res);
}

int circleLineIntersection(const point& p0, const point& p1, const
point& cen,
                           double rad, point& r1, point & r2)
{
```

```
    // handle degenerate case if p0 == p1
    double a, b, c, t1, t2;
    a = dot(p1-p0,p1-p0);
    b = 2 * dot(p1-p0,p0-cen);
    c = dot(p0-cen,p0-cen) - rad * rad;
    double det = b * b - 4 * a * c;
    int res;
    if (fabs(det) < EPS)
        det = 0, res = 1;
    else if (det < 0)
        res = 0;
    else
        res = 2;
    det = sqrt(det);
    t1 = (-b + det) / (2 * a);
    t2 = (-b - det) / (2 * a);
    r1 = p0 + t1 * (p1 - p0);
    r2 = p0 + t2 * (p1 - p0);
    return res;
}

int circleCircleIntersection(const point &c1, const double&r1,
                             const point &c2, const double&r2,
point &res1, point &res2)
{
    if (same(c1,c2) && fabs(r1 - r2) < EPS)
    {
        res1 = res2 = c1;
        return fabs(r1) < EPS ? 1 : OO;
    }
    double len = length(vec(c1,c2));
    if (fabs(len - (r1 + r2)) < EPS || fabs(fabs(r1 - r2) - len) <
EPS)
    {
        point d, c;
        double r;
        if (r1 > r2)
            d = vec(c1,c2), c = c1, r = r1;
        else
            d = vec(c2,c1), c = c2, r = r2;
        res1 = res2 = normalize(d) * r + c;
        return 1;
    }
    if (len > r1 + r2 || len < fabs(r1 - r2))
        return 0;
    double a = cosRule(r2, r1, len);
    point c1c2 = normalize(vec(c1,c2)) * r1;
    res1 = rotate(c1c2,a) + c1;
    res2 = rotate(c1c2,-a) + c1;
    return 2;
}
// P1P2 diameter
```

```
void circle2(const point& p1, const point& p2, point& cen, double&
r)
{
    cen = mid(p1,p2);
    r = length(vec(p1,p2)) / 2;
}
// cercle circonscrit
bool circle3(const point& p1, const point& p2, const point& p3,
point& cen,
             double& r)
{
    point m1 = mid(p1,p2);
    point m2 = mid(p2,p3);
    point perp1 = perp(vec(p1,p2));
    point perp2 = perp(vec(p2,p3));
    bool res = intersect(m1, m1 + perp1, m2, m2 + perp2, cen);
    r = length(vec(cen,p1));
    return res;
}
// point % cercle
STATE circlePoint(const point & cen, const double & r, const
point& p)
{
    double lensqr = lengthSqr(vec(cen,p));
    if (fabs(lensqr - r * r) < EPS)
        return BOUNDRY;
    if (lensqr < r * r)
        return IN;
    return OUT;
}
//p is outside the circle
int tangentPoints(const point & cen, const double & r, const
point& p,
                  point &r1, point &r2)
{
    STATE s = circlePoint(cen, r, p);
    if (s != OUT)
    {
        r1 = r2 = p;
        return s == BOUNDRY;
    }
    point cp = vec(cen,p);
    double h = length(cp);
    double a = acos(r / h);
    cp = normalize(cp) * r;
    r1 = rotate(cp,a) + cen;
    r2 = rotate(cp,-a) + cen;
    return 2;
}

typedef pair<point, point> segment;
// tangentes communes Ã?  deux cercles
```

```
void getCommonTangents(point c1, double r1, point c2, double r2,
vector&lt;segment&gt; &res)
{
    if (r1 < r2) swap(r1, r2), swap(c1, c2);
    double d = length(c1 - c2);
    double theta = acos((r1 - r2) / d);
    point v = c2 - c1;
    v = v / hypot(v.imag(), v.real());
    point v1 = v * exp(point(0, theta));
    point v2 = v * exp(point(0, -theta));
    res.clear();
    res.push_back(segment(c1 + v1 * r1, c2 + v1 * r2));
    res.push_back(segment(c1 + v2 * r1, c2 + v2 * r2));
    theta = acos((r1 + r2) / d);
    v1 = v * exp(point(0, theta));
    v2 = v * exp(point(0, -theta));
    res.push_back(segment(c1 + v1 * r1, c2 - v1 * r2));
    res.push_back(segment(c1 + v2 * r1, c2 - v2 * r2));
}

// minimum enclosing circle
//init p array with the points and ps with the number of points
//cen and rad are result circle
//you must call random_shuffle(p,p+ps); before you call mec
#define MAXPOINTS 100000
point p[MAXPOINTS], r[3], cen;
int ps, rs;
double rad;

void mec()
{
    if (rs == 3)
    {
        circle3(r[0], r[1], r[2], cen, rad);
        return;
    }
    if (rs == 2 && ps == 0)
    {
        circle2(r[0], r[1], cen, rad);
        return;
    }
    if (!ps)
    {
        cen = r[0];
        rad = 0;
        return;
    }
    ps--;
    mec();
    if (circlePoint(cen, rad, p[ps]) == OUT)
    {
        r[rs++] = p[ps];
```

```
        mec();
        rs--;
    }
    ps++;

}

//to check if the points are sorted anti-clockwise or clockwise
//remove the fabs at the end and it will return -ve value if
clockwise
double polygonArea(const vector<point>&p)
{
    double res = 0;
    for (int i = 0; i < sz(p); i++)
    {
        int j = (i + 1) % sz(p);
        res += cross(p[i],p[j]);
    }
    return fabs(res) / 2;
}

// return the centroid point of the polygon
// The centroid is also known as the "centre of gravity" or the
"center of mass". The position of the centroid
// assuming the polygon to be made of a material of uniform
density.
point polyginCentroid(vector<point> &polygon)
{

    double a = 0;
    double x=0.0,y=0.0;
    for (int i = 0; i < (int) polygon.size(); i++)
    {
        int j = (i + 1) % polygon.size();

        x += (polygon[i].X + polygon[j].X) * (polygon[i].X *
polygon[j].Y
                                              - polygon[j].X *
polygon[i].Y);

        y += (polygon[i].Y + polygon[j].Y) * (polygon[i].X *
polygon[j].Y
                                              - polygon[j].X *
polygon[i].Y);

        a += polygon[i].X * polygon[j].Y - polygon[i].Y *
polygon[j].X;
    }

    a *= 0.5;
    x /= 6 * a;
    y /= 6 * a;
```

```
    return point(x,y);
}

int picksTheorm(vector<point>& p)
{
    double area = 0;
    int bound = 0;
    for (int i = 0; i < sz(p); i++)
    {
        int j = (i + 1) % sz(p);
        area += cross(p[i],p[j]);
        point v = vec(p[i],p[j]);
        bound += abs(__gcd((int) v.X, (int) v.Y));
    }
    area /= 2;
    area = fabs(area);
    return round(area - bound / 2 + 1);
}
//convex polygon [a , b) sens trigonomÃ?Â©trique
void polygonCut(const vector<point>& p, const point&a, const
point&b, vector<
                point>& res)
{
    res.clear();
    for (int i = 0; i < sz(p); i++)
    {
        int j = (i + 1) % sz(p);
        bool in1 = cross(vec(a,b),vec(a,p[i])) > EPS;
        bool in2 = cross(vec(a,b),vec(a,p[j])) > EPS;
        if (in1)
            res.push_back(p[i]);
        if (in1 ^ in2)
        {
            point r;
            intersect(a, b, p[i], p[j], r);
            res.push_back(r);
        }
    }
}

//assume that both are anti-clockwise
void convexPolygonIntersect(const vector<point>& p, const
vector<point>& q,
                            vector<point>& res)
{
    res = q;
    for (int i = 0; i < sz(p); i++)
    {
        int j = (i + 1) % sz(p);
        vector<point> temp;
        polygonCut(res, p[i], p[j], temp);
```

```
        res = temp;
        if (res.empty())
            return;
    }
}

void voronoi(const vector<point> &pnts, const vector<point>& rect,
vector<
             vector<point> > &res)
{
    res.clear();
    for (int i = 0; i < sz(pnts); i++)
    {
        res.push_back(rect);
        for (int j = 0; j < sz(pnts); j++)
        {
            if (j == i)
                continue;
            point p = perp(vec(pnts[i],pnts[j]));
            point m = mid(pnts[i],pnts[j]);
            vector<point> temp;
            polygonCut(res.back(), m, m + p, temp);
            res.back() = temp;
        }
    }
}

STATE pointInPolygon(const vector<point>& p, const point &pnt)
{
    point p2 = pnt + point(1, 0);
    int cnt = 0;
    for (int i = 0; i < sz(p); i++)
    {
        int j = (i + 1) % sz(p);
        if (same(p[i],p[j]))
            continue;
        if (pointOnSegment(p[i], p[j], pnt))
            return BOUNDRY;
        point r;
        if(!intersect(pnt, p2, p[i], p[j], r))
            continue;
        if (!pointOnRay(pnt, p2, r))
            continue;
        if (same(r,p[i]) || same(r,p[j]))
            if (fabs(r.Y - min(p[i].Y, p[j].Y)) < EPS)
                continue;
        if (!pointOnSegment(p[i], p[j], r))
            continue;
        cnt++;
    }
    return cnt & 1 ? IN : OUT;
}
```

```
struct cmp
{
    point about;
    cmp(point c)
    {
        about = c;
    }
    bool operator()(const point& p, const point& q) const
    {
        double cr = cross(vec(about, p), vec(about, q));
        if (fabs(cr) < EPS)
            return make_pair(p.Y, p.X) < make_pair(q.Y, q.X);
        return cr > 0;
    }
};

void sortAntiClockWise(vector<point>& pnts)
{
    point mn(1 / 0.0, 1 / 0.0);
    for (int i = 0; i < sz(pnts); i++)
        if (make_pair(pnts[i].Y, pnts[i].X) < make_pair(mn.Y,
mn.X))
            mn = pnts[i];

    sort(all(pnts), cmp(mn));
}

void convexHull(vector<point> pnts, vector<point> &convex)
{
    sortAntiClockWise(pnts);
    convex.clear();
    convex.push_back(pnts[0]);
    if (sz(pnts) == 1)
        return;
    convex.push_back(pnts[1]);
    if (sz(pnts) == 2)
    {
        if (same(pnts[0], pnts[1]))
            convex.pop_back();
        return;
    }
    for (int i = 2; i <= sz(pnts); i++)
    {
        point c = pnts[i % sz(pnts)];
        while (sz(convex) > 1)
        {
            point b = convex.back();
            point a = convex[sz(convex) - 2];
            if (cross(vec(b, a), vec(b, c)) < -EPS)
                break;
            convex.pop_back();
```

```
        }
        if (i < sz(pnts))
            convex.push_back(pnts[i]);
    }
}
vector<point> holes;
point pnts[101];
int db_cmp (double a,double b)
{
    if (fabs(a-b)<EPS)
        return 0;
    return (a>b)*2-1;
}
double getx(const segment &s,const double &y)
{
    /// (x-s.first.X)/(y-s.first.Y)=(s.second.X-s.first.X)/
(s.second.Y-s.first.Y)

    return (s.first.X+(((s.second.X-s.first.X)/(s.second.Y-
s.first.Y))*(y-s.first.Y)));
    //return (fabs((s.second.Y-s.first.Y))>EPS &&
db_cmp(x,min(s.first.X, s.second.X))>0 && db_cmp(max(s.first.X,
s.second.X),x)>0);
}
bool cmps(const segment &a,const segment &b)
{
    double x1=(a.first.X+a.second.X)*0.5;
    double x2=(b.first.X+b.second.X)*0.5;
    return db_cmp(x1,x2)<0;
}
struct trapeze
{
    int id;
    vector<point> v;
    trapeze(const segment &l,const segment &r,int id):id(id)
    {
        v.push_back(l.first);
        v.push_back(r.first);
        v.push_back(r.second);
        v.push_back(l.second);
    }
//    bool pointOntrap(const point &p) const
//    {
//        for (int i=0;i<(int)v.size();i++)
//        {
//            int j=((i+1)%(int)v.size());
//            if (pointOnSegment(v[i],v[j],p))
//                return true;
//        }
//        return false;
//    }
};
```

```
const int N = 1024, E = 2000;    //N must be a power of 2
int head[N],to[E],nxt[E],cost[E],n;
int e;
void init()
{
    e=0;
    memset(head,-1,n*(sizeof head[0]));
}
void addEdge(int f, int t, int w)
{
    nxt[e]=head[f];
    head[f]=e;
    cost[e]=w;
    to[e++]=t;
    //cout << f << " " << t << " " << w << endl;
}
bool isIntersecting(const trapeze &t1,const trapeze &t2)
{
    const vector<point> &v1=t1.v;
    const vector<point> &v2=t2.v;
    const double &x1L=v1[3].X;
    const double &x1R=v1[2].X;
    const double &x2L=v2[0].X;
    const double &x2R=v2[1].X;
    return (db_cmp(max(x1L,x2L),min(x1R,x2R))<0);
}
```