Bellman Ford:

```
/*
   Detects in O(VE) if there is a negative cycle reachable from given
source.
   Run bellman taking every node as source to find if there exists any
negative cycle in the graph in O(V^2*E).
   Also finds shortest distance to each node from the given source.
*/
#define MAX         ?
const double INF = 1e80;

struct edgeData{
   int u, v;
   double cost;
};
vector<edgeData>edge;
double dist[MAX+10];

/// cycleNodes will contain at least one node from a negative cycle if
there is any after bellman finishes.

bool bellmanFord(int n, int source, vector<edgeData>&edge,
vector<int>&cycleNodes)
{
   for(int i = 1; i<=n; i++)
       dist[i] = INF;
   dist[source] = 0;
   bool ret = false;
   for(int i = 1; i<=n; i++)
   {
       for(auto e: edge)
       {
           int u = e.u, v = e.v;
           LL cost = e.cost;
           if(dist[v] > dist[u] + e.cost + eps)
           {
               if(i == n)
               {
                   ret = true;
                   cycleNodes.push_back(v);
               }
               dist[v] = dist[u] + e.cost;

           }
       }
   }
   return ret;
}
```

BigInteger:

```
struct Bigint {
    // representations and structures
    string a; // to store the digits
    int sign; // sign = -1 for negative numbers, sign = 1 otherwise

    // constructors
    Bigint() {} // default constructor
    Bigint( string b ) { (*this) = b; } // constructor for string

    // some helpful methods
    int size() { // returns number of digits
        return a.size();
    }
    Bigint inverseSign() { // changes the sign
        sign *= -1;
        return (*this);
    }
    Bigint normalize( int newSign ) { // removes leading 0, fixes sign
        for( int i = a.size() - 1; i > 0 && a[i] == '0'; i-- )
            a.erase(a.begin() + i);
        sign = ( a.size() == 1 && a[0] == '0' ) ? 1 : newSign;
        return (*this);
    }

    // assignment operator
    void operator = ( string b ) { // assigns a string to Bigint
        a = b[0] == '-' ? b.substr(1) : b;
        reverse( a.begin(), a.end() );
        this->normalize( b[0] == '-' ? -1 : 1 );
    }

    // conditional operators
    bool operator < ( const Bigint &b ) const { // less than operator
        if( sign != b.sign ) return sign < b.sign;
        if( a.size() != b.a.size() )
            return sign == 1 ? a.size() < b.a.size() : a.size() >
b.a.size();
        for( int i = a.size() - 1; i >= 0; i-- ) if( a[i] != b.a[i] )
            return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
        return false;
    }
    bool operator == ( const Bigint &b ) const { // operator for
equality
        return a == b.a && sign == b.sign;
    }
```

```
    // mathematical operators
    Bigint operator + ( Bigint b ) { // addition operator overloading
        if( sign != b.sign ) return (*this) - b.inverseSign();
        Bigint c;
        for(int i = 0, carry = 0; i<a.size() || i<b.size() || carry;
i++ ) {
            carry+=(i<a.size() ? a[i]-48 : 0)+(i<b.a.size() ? b.a[i]-
48 : 0);
            c.a += (carry % 10 + 48);
            carry /= 10;
        }
        return c.normalize(sign);
    }
    Bigint operator - ( Bigint b ) { // subtraction operator
overloading
        if( sign != b.sign ) return (*this) + b.inverseSign();
        int s = sign; sign = b.sign = 1;
        if( (*this) < b ) return ((b -
(*this)).inverseSign()).normalize(-s);
        Bigint c;
        for( int i = 0, borrow = 0; i < a.size(); i++ ) {
            borrow = a[i] - borrow - (i < b.size() ? b.a[i] : 48);
            c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
            borrow = borrow >= 0 ? 0 : 1;
        }
        return c.normalize(s);
    }
    Bigint operator * ( Bigint b ) { // multiplication operator
overloading
        Bigint c("0");
        for( int i = 0, k = a[i] - 48; i < a.size(); i++, k = a[i] -
48 ) {
            while(k--) c = c + b; // ith digit is k, so, we add k
times
            b.a.insert(b.a.begin(), '0'); // multiplied by 10
        }
        return c.normalize(sign * b.sign);
    }
    Bigint operator / ( Bigint b ) { // division operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48
);
        Bigint c("0"), d;
        for( int j = 0; j < a.size(); j++ ) d.a += "0";
        int dSign = sign * b.sign; b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b, d.a[i]++;
        }
        return d.normalize(dSign);
    }
    Bigint operator % ( Bigint b ) { // modulo operator overloading
```

```
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48
);
        Bigint c("0");
        b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b;
        }
        return c.normalize(sign);
    }



    // output method
    void print() {
        if( sign == -1 ) putchar('-');
        for( int i = a.size() - 1; i >= 0; i-- ) putchar(a[i]);
    }
};
```

Binary Tree Transformation:

```
void dfs(int node, int parent)
{
    if(E[node].size() == 1)
        return;
    int newChild;
    int i;
    for(i = 0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(v != parent)
        {
            binary[node].pb(v);
            break;
        }
    }
    if(E[node].size() != 2)
    {
        newChild = ++n;
        binary[node].pb(newChild);
        i++;
        for(; i<E[node].size(); i++)
        {
            int v = E[node][i];
            if(v != parent)
            {
                i++;
                binary[newChild].pb(v);
```

```
                break;
            }
        }

        while(i < E[node].size())
        {
            int nxtChild = ++n;
            binary[newChild].pb(nxtChild);
            newChild = nxtChild;
            binary[newChild].pb(E[node][i]);
            i++;
        }
    }

    for(int i = 0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(v != parent)
            dfs(v, node);
    }
}
void makeBinary()
{
    if(E[1].size() <= 2)
    {
        binary[1] = E[1];
        for(int i = 0; i<E[1].size(); i++)
            dfs(E[1][i], 1);
    }
    else
        dfs(1,0);
}
```

Convex Hull Angular:

```
/*
ConvexHull : Graham's Scan O(n lg n), integer implementation
P[]: holds all the points, C[]: holds points on the hull
np: number of points in P[], nc: number of points in C[]
to handle duplicate, call makeUnique() before calling convexHull()
call convexHull() if you have np >= 3
to remove co-linear points on hull, call compress() after convexHull()
*/
point P[MAX], C[MAX], P0;
inline int triArea2(const point &a, const point &b, const point &c)
{
    return (a.x*(b.y-c.y) + b.x*(c.y-a.y) + c.x*(a.y-b.y));
}
inline int sqDist(const point &a, const point &b)
```

```
{
    return ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}
inline bool comp(const point &a, const point &b)
{
    int d = triArea2(P0, a, b);
    if(d < 0) return false;
    if(!d && sqDist(P0, a) > sqDist(P0, b)) return false;
    return true;
}
inline bool normal(const point &a, const point &b)
{
    return ((a.x==b.x) ? a.y < b.y : a.x < b.x);
}
inline bool issame(const point &a, const point &b)
{
    return (a.x == b.x && a.y == b.y);
}
inline void makeUnique(int &np)
{
    sort(&P[0], &P[np], normal);
    np = unique(&P[0], &P[np], issame) - P;
}
void convexHull(int &np, int &nc)
{
    makeUnique(np);
    int i, j, pos = 0;
    for(i = 1; i < np; i++)
        if(P[i].y<P[pos].y || (P[i].y==P[pos].y && P[i].x<P[pos].x))
            pos = i;
    swap(P[0], P[pos]);
    P0 = P[0];
    sort(&P[1], &P[np], comp);
    for(i = 0; i < 3; i++) C[i] = P[i];
    for(i = j = 3; i < np; i++)
    {
        while(triArea2(C[j-2], C[j-1], P[i]) < 0) j--;
        C[j++] = P[i];
    }
    nc = j;
}
void compress(int &nc)
{
    int i, j, d;
    C[nc] = C[0];
    for(i=j=1; i < nc; i++)
    {
        d = triArea2(C[j-1], C[i], C[i+1]);
        if(d || (!d && issame(C[j-1], C[i+1]))) C[j++] = C[i];
    }
    nc = j;
}
```

Convex Hull Trick:

```
struct Line{
        LL m,c;
        Line(LL _m = 0, LL _c = 0):m(_m), c(_c){};
};

bool operator < (const Line &u, const Line &v){
    return u.m > v.m;
}

struct ConvexHullTrick{ //works with long long integers.

    vector<Line> Q; //Fast -> Slow -> Slower -> Slowest
    vector<Line> can;
    bool minFlag;

    ConvexHullTrick(bool flg = true):minFlag(flg){};

    LL getX(Line u, Line v){ // Fast vrs Slow *ORDER MATTERS*
        LL difC = v.c - u.c, difM = u.m - v.m;
        if(difC % difM == 0) return difC/difM;
        if(difC < 0) return difC/difM;
        return difC/difM + 1;
    }

    bool isBad(Line L1, Line L2, Line L3)
    {
        if(minFlag == false) return (L3.c - L1.c) / (long double)
(L1.m - L3.m) > (L2.c-L1.c) / (long double) (L1.m - L2.m);
        else return (L3.c - L1.c) / (long double) (L1.m - L3.m) <
(L2.c-L1.c) / (long double) (L1.m - L2.m);
    }

    void addLine(Line L){ //Has to be slower than then the slowest in
the Q
        while(Q.empty() == false)
        {
            if(Q.back().m < L.m) __builtin_trap();
            else if(minFlag == false && Q.back().m == L.m && L.c >
Q.back().c) Q.pop_back();
            else if(minFlag == true && Q.back().m == L.m && L.c <
Q.back().c) Q.pop_back();
            else if(Q.back().m == L.m) return;
            else if(Q.size() <= 1) break;
            else if(isBad(Q[Q.size()-2], Q.back(), L)) Q.pop_back();
            else break;
        }
```

```
        Q.push_back(L);
    }

    LL query(LL pos, bool f = false){
        int lo = 0, hi = (int) Q.size() - 1, n = hi, mid;
        LL L, R;

        while(true)
        {
            mid = (lo+hi)/2;
            if(minFlag)
            {
                if(mid == 0) L = -5e18;
                else L = getX(Q[mid-1], Q[mid]);

                if(mid == n) R = 5e18;
                else R = getX(Q[mid], Q[mid+1]);


                if(L <= pos && pos < R) return Q[mid].m * pos +
Q[mid].c;
                if(pos < L) hi = mid-1;
                else lo = mid+1;
            }

            else
            {
                if(mid == n) L = -5e18;
                else L = getX(Q[mid], Q[mid+1]);

                if(mid == 0) R = 5e18;
                else R = getX(Q[mid-1], Q[mid]);

                if(L <= pos && pos < R) return Q[mid].m * pos +
Q[mid].c;
                if(pos < L) lo = mid+1;
                else hi = mid-1;
            }
        }
    }
}
```

DSU On Tree:

```
vector<int>E[MAX+10];
int subtreeSize[MAX+10];

int getSize(int node, int par);

void add(int node, int par, int x, int bigChild = -1)
```

```
{
    /// Do whatever you have to do for this node with the command x
    for(auto v: E[node])
    {
        if(v == par || v == bigChild)
            continue;
        add(v, node, x);
    }
}

void dfs(int node, int par, bool keep)
{
    int bigChild = -1;
    for(auto v: E[node])
    {
        if(v == par)
            continue;
        if(bigChild == -1 || (subtreeSize[bigChild] < subtreeSize[v]))
            bigChild = v;
    }
    for(auto v: E[node])
    {
        if(v == par || v == bigChild)
            continue;
        dfs(v, node, 0);
    }
    if(bigChild != -1)
        dfs(bigChild, node, 1);
    add(node, par, 1, bigChild);
    /// my needed array is ready. Handle query next

    /// Clear solution as I'm not a bigchild. :(
    if(keep == 0)
    {
        add(node, par, -1);
    }
}
Strongly Connected Component:
/*
    Topologically sort the nodes of the main graph.
    Start running dfs from nodes in sorted order in the reverse graph.
    Nodes visited in a single dfs form a SCC.
*/
```

Suffix Array (Nlognlogn):

```
char text[MAX+10];
struct data{
    int tupleRank[2];
    int idx;
}A[MAX+10];
int sparseTable[MAXLG+2][MAX+10];
```

```
bool operator < (data a, data b)
{
    return (a.tupleRank[0] == b.tupleRank[0]) ? (a.tupleRank[1] <
b.tupleRank[1]) : (a.tupleRank[0] < b.tupleRank[0]);
}
int step, SA[MAX+10], n;
void buildSA()
{
    n = strlen(text);
    if(n == 1)
    {
        sparseTable[0][0] = 0;
        SA[0] = 0;
        return;
    }
    for(int i = 0; i<n; i++)
        sparseTable[0][i] = text[i];
    step = 1;
    for(int jump = 1; jump < n; jump <<= 1, step++)
    {
        for(int i = 0; i<n; i++)
        {
            A[i].idx = i;
            A[i].tupleRank[0] = sparseTable[step-1][i];
            A[i].tupleRank[1] = (jump+i < n) ? (sparseTable[step-
1][jump+i]) : -1;
        }
        sort(A,A+n);
        sparseTable[step][A[0].idx] = 0;
        for(int i = 1; i<n; i++)
        {
            if(A[i-1].tupleRank[0] == A[i].tupleRank[0] && A[i-
1].tupleRank[1] == A[i].tupleRank[1])
                sparseTable[step][A[i].idx] = sparseTable[step][A[i-
1].idx];
            else
                sparseTable[step][A[i].idx] = i;
        }
    }
    for(int i = 0; i<n; i++)
        SA[sparseTable[step-1][i]] = i;
}

int getLCP(int a, int b)
{
    int ret = 0;
    for(int i = step-1; i>=0; i--)
    {
        if(sparseTable[i][a] == sparseTable[i][b])
        {
            ret += (1<<i);
            a += (1<<i);
```

```
            b += (1<<i);
            if(a>=n || b>=n)
                break;
        }
    }
    return ret;
}
```

TwoSat:

```
/*
    MAX must be equal to the maximum number of variables.
    n passed in init() is the number of variables.
    O(V+E)
    !a is represented as -a.
    example xor:
    |a|b|
    |0|0| x  or(a,b)
    |0|1|
    |1|0|
    |1|1| x  or(-a,-b)
    do OR of negation of values of variables for each undesired
situation to make it impossible.
*/
#define MAX         ?
struct twoSat{
    int n;
    vector<int> E[MAX*2+10], V, Rev[MAX*2+10], sortedNodes;
    bool state[MAX*2+10], vis[MAX*2+10];
    int compId[MAX*2+10];

    void init(int _n)
    {
        n = _n;
        for(int i = 0; i<=2*n; i++)
            E[i].clear(), Rev[i].clear();
        V.clear();
        sortedNodes.clear();
        mem(state, 0);
    }

    inline int actual(int a)
    {
        if(a < 0)
            return n - a;
        else
            return a;
    }

    inline int neg(int a)
```

```
{
    if(a > n)
        return a-n;
    else
        return n+a;
}

void dfs(int node)
{
    vis[node] = true;
    for(auto v: E[node])
    {
        if(!vis[v])
            dfs(v);
    }
    V.push_back(node);
}

void dfsRev(int node, int id)
{
    sortedNodes.push_back(node);
    vis[node] = true;
    compId[node] = id;
    for(auto v: Rev[node])
    {
        if(!vis[v])
            dfsRev(v, id);
    }
}

void buildSCC()
{
    int i;
    V.clear();
    mem(vis,0);
    for(int i = 1; i<=2*n; i++)
    {
        if(!vis[i])
            dfs(i);
    }
    mem(vis,0);
    reverse(all(V));
    int cnt = 0;
    for(auto u: V)
    {
        if(!vis[u])
            cnt++,dfsRev(u, cnt);
    }
}

bool topologicalOrder(int a, int b)
{
```

```
        return compId[a] < compId[b];
    }
    bool satisfy()
    {
        buildSCC();
        /// if leader of i and -i is the same, then they are in the
same component
        /// 2-sat is impossible, return 0
        for(int i = 1; i<=n; i++)
        {
            if(compId[i]==compId[i+n])
                return 0;
        }
        /// topologically sort the components

        /// start from the back end of topologically sorted order and
try to give everyone true state in that component
        /// if someone's opposite has true state, then let him have
false state.
        for(int i = (int)sortedNodes.size()-1; i>=0; i--)
        {
            int u = sortedNodes[i];
            if( state[neg(u)] == 0)
                state[u]=1;
        }
        return 1;
    }

    void addEdge(int u, int v)
    {
        u = actual(u);
        v = actual(v);
        E[u].pb(v);
        Rev[v].pb(u);
    }
    void addOr(int u, int v)
    {
        addEdge(-u, v);
        addEdge(-v, u);
    }

    void addXor(int u, int v)
    {
        addOr(u,v);
        addOr(!u,!v);
    }

    void forceTrue(int u)
    {
        addEdge(-u, u);
    }
    void forceFalse(int u)
```

```
    {
        addEdge(u,-u);
    }

    void addOriginalImplication(int u, int v)
    {
        addOr(-u,v);
    }
}solver;
```

Template:

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

typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair <int,int> PII;
typedef pair <long long,long long> PLL;

typedef unsigned long long int ULL;
typedef long long int LL;
typedef pair<int,int> pii;
typedef pair<LL,LL> pll;

#define mem(t, v)       memset ((t) , v, sizeof(t))
#define un(x)           x.erase(unique(all(x)), x.end())
#define sf(n)           scanf("%d", &n)
#define sff(a,b)        scanf("%d %d", &a, &b)
#define sfff(a,b,c)     scanf("%d %d %d", &a, &b, &c)
#define xx              first
#define yy              second

#define si(a)           scanf("%d",&a)
#define sii(a,b)        scanf("%d %d",&a,&b)
#define siii(a,b,c)     scanf("%d %d %d",&a,&b,&c)

#define sl(a)           scanf("%lld",&a)
#define sll(a,b)        scanf("%lld %lld",&a,&b)
#define slll(a,b,c)     scanf("%lld %lld %lld",&a,&b,&c)

#define pb              push_back
#define mp              make_pair
#define all(v)          v.begin(),v.end()
#define D(x)            cerr << #x " = " << x << '\n'
#define DBG             cerr << "Hi!" << '\n'

#define CLR(a)          memset(a,0,sizeof(a))
#define SET(a)          memset(a,-1,sizeof(a))
```

```
#define eps             1e-9
#define PI              acos(-1.0)

int setBit(int n,int pos){ return n = n | (1 << pos); }
int resetBit(int n,int pos){ return n = n & ~(1 << pos); }
bool checkBit(int n,int pos){ return (bool)(n & (1 << pos)); }

//int fx[] = {+0, +0, +1, -1, -1, +1, -1, +1};
//int fy[] = {-1, +1, +0, +0, +1, +1, -1, -1}; //Four & Eight
Direction

///------------------------------------------------------------------
----------///


int main()
{
//    freopen("in.txt","r",stdin);
//    freopen("out.txt","w",stdout);

//    ios_base::sync_with_stdio(false);
//    cin.tie(NULL);

    return 0;
}
```