Convex Hull
struct pt {
    double x, y;
};
 	
int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}
 	
bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}
 	
void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1)
        return;
 	
    sort(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
	Line intersection:
	struct pt {
	    long long x, y;
	    pt() {}
	    pt(long long _x, long long _y) : x(_x), y(_y) {}
	    pt operator-(const pt& p) const { return pt(x - p.x, y - p.y); }
	    long long cross(const pt& p) const { return x * p.y - y * p.x; }
	    long long cross(const pt& a, const pt& b) const { return (a - *this).cross(b - *this); }
	};
	
	int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1; }
	
	bool inter1(long long a, long long b, long long c, long long d) {
	    if (a > b)
	        swap(a, b);
	    if (c > d)
	        swap(c, d);
	    return max(a, c) <= min(b, d);
	}
	
	bool check_inter(const pt& a, const pt& b, const pt& c, const pt& d) {
	    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
	        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);
	    return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
	           sgn(c.cross(d, a)) != sgn(c.cross(d, b));
	}
	
	FFT:
	using cd = complex<double>;
	const double PI = acos(-1);

```
        while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], a[i], include_collinear))
            up.pop_back();
        up.push_back(a[i]);
      }
      if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
        while (down.size() >= 2 && !ccw(down[down.size()-2], down[down.size()-1], a[i], include_collinear))
            down.pop_back();
        down.push_back(a[i]);
      }
   }

   if (include_collinear && up.size() == a.size()) {
      reverse(a.begin(), a.end());
      return;
   }
   a.clear();
   for (int i = 0; i < (int)up.size(); i++)
      a.push_back(up[i]);
   for (int i = down.size() - 2; i > 0; i--)
      a.push_back(down[i]);
}
```

```
int reverse(int num, int lg_n) {
   int res = 0;
   for (int i = 0; i < lg_n; i++) {
      if (num & (1 << i))
         res |= 1 << (lg_n - 1 - i);
   }
   return res;
}

void fft(vector<cd> & a, bool invert) {
   int n = a.size();
   int lg_n = 0;
   while ((1 << lg_n) < n)
      lg_n++;

   for (int i = 0; i < n; i++) {
      if (i < reverse(i, lg_n))
         swap(a[i], a[reverse(i, lg_n)]);
   }
   for (int len = 2; len <= n; len <<= 1) {
      double ang = 2 * PI / len * (invert ? -1 : 1);
      cd wlen(cos(ang), sin(ang));
      for (int i = 0; i < n; i += len) {
         cd w(1);
         for (int j = 0; j < len / 2; j++) {
            cd u = a[i+j], v = a[i+j+len/2] * w;
            a[i+j] = u + v;
            a[i+j+len/2] = u - v;
            w *= wlen;
         }
      }
   }
   if (invert) {
      for (cd & x : a)
         x /= n;
   }
}
```

Game Theory:

NIM: n piles of objs. One can take any number of objs from any pile (i.e. set of possible moves for the i-th pile is $M = [pile_i]$, $[x] := \{1,2,...,bxc\}$).	. Strategy: ¶ make the Nim-Sum 0 by *decreasing* a heap; · the same, except when the normal move would only leave heaps of size 1. In that case, leave an odd number of 1's. The result of · is the same as ¶, opposite if all piles are 1's. Many games are essentially NIM.
NIM (powers) $M = \{a^m \mid m \geq 0\}$	If a odd: $SG_n = n\%2$ If a even: $SG_n = 2$, if $n \equiv a\%(a + 1)$; $SG_n = n\%(a + 1)\%2$, else.
NIM (half)	$\neg SG2n = n$, $SG2n{+}1 = SGn$ $SG_0 = 0$, $SG_n = [\log_2 n] + 1$
NIM (divisors) $M_\neg$ = divisors of $pile_i$ M = proper divisors of $pile_i$	$\neg SG_0 = 0$, $SG_n = SG_{,n} + 1$ $SG_1 = 0$, SG_n = number of 0's at the end of n_{binary}
Subtraction Game $M_\neg = [k]$ $M = S$ (finite) $M_® = S \cup \{pile_i\}$	$SG_{\neg,n} = n \bmod (k{+}1)$. ¶lose if $SG = 0$; ·lose if $SG = 1$. $SG®_{,n} = SG_{,n} + 1$ For any finite M, SG of one pile is eventually periodic.
Moore's NIM_k One can take any number of objs from at most k piles.	¶Write $pile_i$ in binary, sum up in base $k + 1$ without carry. Losing if the result is 0. · If all piles are 1's, losing iff $n \equiv 1\%(k + 1)$. Otherwise the result is the same as ¶.
Staircase NIM n piles in a line. One can take any number of objs from $pile_i$, $i > 0$ to $pile_{i-1}$	Losing if the NIM formed by the odd-indexed piles is losing(i.e.
Lasker's NIM Two possible moves: 1.take any number of objs; 2.split a pile into two (no obj removed)	$SG_n = n$, if $n \equiv 1,2(\%4)$ $SG_n = n + 1$, if $n \equiv 3(\%4)$ $SG_n = n - 1$, if $n \equiv 0(\%4)$

Kayles Two possible moves: 1.take 1 or 2 objs; 2.split a pile into two (after removing objs)	SG_n for small n can be computed recursively. SG_n for $n \in [72, 83]$: 4 1 2 8 1 4 7 2 1 8 2 7 SG_n becomes periodic from the 72-th item with period length 12.
Dawson's Chess n boxes in a line. One can occupy a box if its neighbours are not occupied.	SG_n for $n \in [1, 18]$: 1 1 2 0 3 1 1 0 3 3 2 2 4 0 5 2 2 3 Period = 34 from the 52-th item.
Wythoff's Game **Two** piles of objs. One can take any number of objs from either pile, or take the *same* number from *both* piles.	$n_k = \lfloor k\varphi \rfloor = \lfloor m_k\varphi \rfloor - m_k$ $m_k = \lfloor k\varphi^2 \rfloor = \lceil n_k\varphi \rceil = n_k + k$ is the k-th losing position. n_k and m_k form a pair of complementary Beatty Sequences (since). Every $x > 0$ appears either in n_k or in m_k.
Mock Turtles n coins in a line. One can turn over 1, 2 or 3 coins, with the rightmost from head to tail.	$SG_n = 2n$, if ones($2n$) odd; $SG_n = 2n + 1$, else. ones(x): the number of 1's in x_{binary} SG_n for $n \in [0, 10]$ (leftmost position is 0): 1 2 4 7 8 11 13 14 16 19 21
Ruler n coins in a line. One can turn over any *consecutive* coins, with the rightmost from head to tail.	SG_n = the largest power of 2 dividing n. This is implemented as $n\&{-n}$(lowbit) SG_n for $n \in [1, 10]$: 1 2 1 4 1 2 1 8 1 2
Hackenbush-tree Given a forest of rooted trees, one can take an edge and remove the part which becomes unrooted.	At every branch, one can replace the branches by a nonbranching stalk of length equal to their nim-sum.
Hackenbush-graph	Vertices on any circuit can be *fused* without changing SG of the graph. Fusion: two neighbouring vertices into one, and bend the edge into a loop.

const ll FACTORIAL_SIZE = 1.1e6; ll fact[FACTORIAL_SIZE], ifact[FACTORIAL_SIZE]; void gen_factorial(ll n) { fact[0] = fact[1] = ifact[0] = ifact[1] = 1;
for (ll i = 2; i <= n; i++) { fact[i] = (i * fact[i - 1]) % mod;
} ifact[n] = inv(fact[n]); for (ll i = n - 1; i >= 2; i--) { ifact[i] = ((i + 1) * ifact[i + 1])
% mod;
}
}
 	
ll nck(ll n, ll k) { ll den = (ifact[k] * ifact[n - k]) % mod; return (den * fact[n]) %
 	
4) Catalan numbers
 	
const int MOD = ... const int
MAX = ... int
catalan[MAX]; void init() {
catalan[0] = catalan[1] = 1;
for (int i=2; i<=n; i++) {
catalan[i] = 0; for (int j=0; j < i; j++) { catalan[i] += (catalan[j] *
catalan[i-j-1]) % MOD; if
(catalan[i] >= MOD) {
catalan[i] -= MOD;
}
}
}
}
	ll gcdExtended(ll a, ll b, ll *x, ll *y)
	{
	if (a == 0)
	{
	*x = 0, *y = 1;
	return b;
	}
	ll x1, y1;
	ll gcd = gcdExtended(b%a, a, &x1, &y1);
	*x = y1 - (b/a) * x1;
	*y = x1;
	
	return gcd;
	}
	ll modInverse(ll b, ll m)
	{
	ll x, y;
	ll g = gcdExtended(b, m, &x, &y);
	
	if (g != 1)
	return -1;
	
	return (x%m + m) % m;
	}
	ll mdDiv(ll a, ll b, ll m)
	{
	a = a % m;
	ll inv = modInverse(b, m);
	if (inv == -1)
	re -1; //cout << "Division not defined";
	else
	re (inv * a) % m;
	}

Convex Hull

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1)
        return;

    sort(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
```

Line intersection:

```
struct pt {
    long long x, y;
    pt() {}
    pt(long long _x, long long _y) : x(_x), y(_y) {}
    pt operator-(const pt& p) const { return pt(x - p.x, y - p.y); }
    long long cross(const pt& p) const { return x * p.y - y * p.x; }
    long long cross(const pt& a, const pt& b) const { return (a - *this).cross(b - *this); }
};

int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1; }

bool inter1(long long a, long long b, long long c, long long d) {
    if (a > b)
        swap(a, b);
    if (c > d)
        swap(c, d);
    return max(a, c) <= min(b, d);
}

bool check_inter(const pt& a, const pt& b, const pt& c, const pt& d) {
    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);
    return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
           sgn(c.cross(d, a)) != sgn(c.cross(d, b));
}
```

FFT:

```
using cd = complex<double>;
const double PI = acos(-1);
```

Fenwick 2d:
struct Query
{
int x1, y1; // x and y co-ordinates of bottom left
int x2, y2; // x and y co-ordinates of top right
};
// A function to update the 2D BIT
void updateBIT(int BIT[][N+1], int x, int y, int val)
{
for (; x <= N; x += (x & -x))
{
// This loop update all the 1D BIT inside the
// array of 1D BIT = BIT[x]
for (int yy=y; yy <= N; yy += (yy & -yy))
BIT[x][yy] += val;
}
return;
}
// A function to get sum from (0, 0) to (x, y)
int getSum(int BIT[][N+1], int x, int y)
{
int sum = 0;
 	
for(; x > 0; x -= x&-x)
{
// This loop sum through all the 1D BIT
// inside the array of 1D BIT = BIT[x]
for(int yy=y; yy > 0; yy -= yy&-yy)
{
sum += BIT[x][yy];
}
}
return sum;
}
// A function to create an auxiliary matrix
// from the given input matrix
void constructAux(int mat[][N], int aux[][N+1])
	{
	// Initialise Auxiliary array to 0
	for (int i=0; i<=N; i++)
	for (int j=0; j<=N; j++)
	aux[i][j] = 0;
	// Construct the Auxiliary Matrix
	for (int j=1; j<=N; j++)
	for (int i=1; i<=N; i++)
	aux[i][j] = mat[N-j][i-1];
	return;
	}
	// A function to construct a 2D BIT
	void construct2DBIT(int mat[][N], int BIT[][N+1])
	{
	// Create an auxiliary matrix
	int aux[N+1][N+1];
	constructAux(mat, aux);
	// Initialise the BIT to 0
	for (int i=1; i<=N; i++)
	for (int j=1; j<=N; j++)
	BIT[i][j] = 0;
	for (int j=1; j<=N; j++)
	{
	for (int i=1; i<=N; i++)
	{
	// Creating a 2D-BIT using update function
	// everytime we/ encounter a value inthe
	// input 2D-array
	int v1 = getSum(BIT, i, j);
	int v2 = getSum(BIT, i, j-1);
	int v3 = getSum(BIT, i-1, j-1);
	int v4 = getSum(BIT, i-1, j);
	// Assigning a value to a particular element
	// of 2D BIT
	updateBIT(BIT, i, j, aux[i][j]-(v1-v2-v4+v3));
	}
	}
	return;
	}

Fenwick 1d:

```
int getSum(int BITree[], int index)
{
   int sum = 0; // Initialize result

   // index in BITree[] is 1 more than the
index in arr[]
   index = index + 1;

   // Traverse ancestors of BITree[index]
   while (index>0)
   {
      // Add current element of BITree to sum
      sum += BITree[index];

      // Move index to parent node in getSum
View
      index -= index & (-index);
   }
   return sum;
}

// Updates a node in Binary Index Tree
(BITree) at given index
// in BITree. The given value 'val' is added to
BITree[i] and
// all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index,
int val)
{
   // index in BITree[] is 1 more than the
index in arr[]
   index = index + 1;

   // Traverse all ancestors and add 'val'
   while (index <= n)
   {
   // Add 'val' to current node of BI Tree
   BITree[index] += val;

   // Update index to that of parent in update
View
  index += index & (-index);
   }

}

// Constructs and returns a Binary Indexed
Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
   // Create and initialize BITree[] as 0
   int *BITree = new int[n+1];
   for (int i=1; i<=n; i++)
      BITree[i] = 0;

   // Store the actual values in BITree[] using
update()
   for (int i=0; i<n; i++)
      updateBIT(BITree, n, i, arr[i]);

   // Uncomment below lines to see contents
of BITree[]
   //for (int i=1; i<=n; i++)
   //     cout << BITree[i] << " ";

   return BITree;
}
```

1) Sieve	2) Prime Factorisation sieve

bool prime[N+1]; vector<ll> p; void sieve() { for(ll i=1;i<=N;i++) prime[i]=true;
 	
for (int i=2; i*i<=N; i++) {
if (prime[i] == true) { for (int j=i*i;j<=N;j+=i) prime[j] = false;
} } for(ll i=1;i<=N;i++){
if(prime[i]==true) p.pb(i);
}
}
	const int MAXN=1e7+5; ll spf[MAXN]; void sieve() { spf[1]=1; for (ll i=2;i<MAXN;i++){ spf[i]=i;
	} for (ll i=4;i<MAXN;i+=2){ spf[i]=2;
	} for (ll i=3;i*i<MAXN;i++) { if (spf[i]==i) { for (int j=i*i;j<MAXN;j+=i) { if (spf[j]==j){ spf[j]=i;} } }} }
	vector<ll> get_factors(ll x) {
	vector<ll> ret; while (x!=1) {
	ret.push_back(spf[x]); x=x/spf[x];
	} return ret; }

Kmp :

```
vector<int> lps(ust.length());

Int i = 1; l = 0;

while(i < ust.length()) { if(ust[i] == ust[l]) {
l++; Lps[i] = l;

I++;

} else { if(l > 0) { l = lps[l-1]; } else {

Lps[i] = 0;

I++; }}}
```

Aho-Corasick :

```
const int K = 20;
struct vertex {
vertex *next[K], *go[K], *link, *p;
int pch;
bool leaf;
int is_accepting = -1;
```

Manachers :

```
// ust = "@" + alternate string char and "#"
+ "$"; Int c = 0, r = 0, i = 1;

vector<int> lps(ust.length(), 0);

while(i < ust.length() - 1) { Int m = c -(i-c);
if(i < r)

Lps[i] = min(lps[m], r-i);

while(ust[i+lps[i]+1] == ust[i - lps[i] - 1]) {
Lps[i]++; }

if(lps[i] + i > r) { c = i; r = lps[i] + i; }}
```

Rolling hash

```
int q = 311;
struct Hasher { // use two of those, with
different mod (e.g. 1e9+7 and 1e9+9)
string s; int mod;
vector<int> power, pref;
Hasher(const string& s, int mod) : s(s),
mod(mod) {
```

```
};
vertex *create() {
vertex *root = new vertex();
root->link = root;
return root;
}
void add_string (vertex *v, const
vector<int>& s) {
for (int a: s) {
if (!v->next[a]) {
vertex *w = new vertex();
w->p = v;
w->pch = a;
v->next[a] = w; }
v = v->next[a]; }
v->leaf = 1; }
vertex* go(vertex* v, int c);
vertex* get_link(vertex *v) {
if (!v->link)
v->link = v->p->p ? go(get_link(v->p), v-
>pch) : v->p;
return v->link; }
vertex* go(vertex* v, int c) {
if (!v->go[c]) {
if (v->next[c])
v->go[c] = v->next[c];
else  v->go[c] = v->p ? go(get_link(v), c) :
v;  }
return v->go[c]; }
bool is_accepting(vertex *v) {
if (v->is_acceping == -1)
v->is_accepting = get_link(v) == v ? false :
(v->leaf || is_accepting(get_link(v)));
return v->is_accepting; }
```

Graph

Maximum bipartitite: O(ve / v3) int n, k;

```
vector<vector<int>> g;
vector<int> mt;
vector<bool> used;
bool try_kuhn(int v) {
    if (used[v])
        return false;
```

```
power.pb(1);
rep(i,1,s.size()) power.pb((ll)power.back() *
q % mod);
pref.pb(0);
rep(i,0,s.size()) pref.pb(((ll)pref.back() * q
% mod + s[i]) % mod);}
int hash(int l, int r) { // compute hash(s[l..r])
with r inclusive
return (pref[r+1] - (ll)power[r-l+1] * pref[l]
% mod + mod) % mod;} };
```

Graph

Max flow(ford):O(ve2) :   int n;

```
vector<vector<int>> capacity;

vector<vector<int>> adj;

int bfs(int s, int t, vector<int>& parent) {

    fill(parent.begin(), parent.end(), -1);

    parent[s] = -2;

    queue<pair<int, int>> q;

    q.push({s, INF});

    while (!q.empty()) {

        int cur = q.front().first;

        int flow = q.front().second;

        q.pop();

        for (int next : adj[cur]) {

            if (parent[next] == -1 &&
capacity[cur][next]) {

                parent[next] = cur;

                int new_flow = min(flow,
capacity[cur][next]);

                if (next == t)

                return new_flow;

                q.push({next, new_flow}); } } }
return 0; }
```

```
    used[v] = true;
    for (int to : g[v]) {
        if (mt[to] == -1 || try_kuhn(mt[to])) {
            mt[to] = v;
            return true; } }
    return false; }
int main() {
 mt.assign(k, -1);
    vector<bool> used1(n, false);
    for (int v = 0; v < n; ++v) {
        for (int to : g[v]) {
            if (mt[to] == -1) {
                mt[to] = v;
                used1[v] = true;
                break; } } }
    for (int v = 0; v < n; ++v) {
        if (used1[v]) continue;
        used.assign(n, false);
        try_kuhn(v); }
    for (int i = 0; i < k; ++i)
        if (mt[i] != -1)
            printf("%d %d\n", mt[i] + 1, i + 1); }
```

Min-cost max flow : O(2^n/2*n^2*logn)

```
struct Edge {  int from, to, capacity, cost;  };
vector<vector<int>> adj, cost, capacity;
const int INF = 1e9;
void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p) {
    d.assign(n, INF);
```

```
int maxflow(int s, int t) {
    int flow = 0;
    vector<int> parent(n);
    int new_flow;
    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev; } }
    return flow; }
```

---

max-flow(push-relabel):O(ve+v2sqrt(e))

```
const int inf = 1000000000; int n;
vector<vector<int>> capacity, flow;
vector<int> height, excess;
void push(int u, int v) {
    int d = min(excess[u], capacity[u][v] - flow[u][v]);
    flow[u][v] += d;
    flow[v][u] -= d;
    excess[u] -= d;
    excess[v] += d; }
void relabel(int u) {
    int d = inf;
    for (int i = 0; i < n; i++) {
        if (capacity[u][i] - flow[u][i] > 0)
            d = min(d, height[i]);
```

```
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inq[u] = false;
        for (int v : adj[u]) {
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (!inq[v]) {
                    inq[v] = true;
                    q.push(v); }}}}}
int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity; }
    int flow = 0;
    int cost = 0;
    vector<int> d, p;
```

```
    } if (d < inf) { height[u] = d + 1; }  }
vector<int> find_max_height_vertices(int s, int t) {
    vector<int> max_height;
    for (int i = 0; i < n; i++) {
        if (i != s && i != t && excess[i] > 0) {
            if (!max_height.empty() && height[i] > height[max_height[0]])
                max_height.clear();
            if (max_height.empty() || height[i] == height[max_height[0]])
                max_height.push_back(i);
        } } return max_height; }
int max_flow(int s, int t) {
    height.assign(n, 0);
    height[s] = n;
    flow.assign(n, vector<int>(n, 0));
    excess.assign(n, 0);
    excess[s] = inf;
    for (int i = 0; i < n; i++) {
        if (i != s)
            push(s, i); }
    vector<int> current;
    while (!(current = find_max_height_vertices(s, t)).empty()) {
        for (int i : current) {
            bool pushed = false;
            for (int j = 0; j < n && excess[i]; j++) {
                if (capacity[i][j] - flow[i][j] > 0 && height[i] == height[j] + 1) {
                    push(i, j);
```

```
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INF) break;
        int f = K - flow;
        int cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur]; }
 flow += f;
        cost += f * d[t];
        cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur]; } }
    if (flow < K)  return -1;
    else return cost; }
```

Lca: int n, l, timer ;

```
vector<vector<int>> adj;
vector<int> tin, tout;
vector<vector<int>> up;
void dfs(int v, int p) {
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i) up[v][i] =
up[up[v][i-1]][i-1];
    for (int u : adj[v]) { if (u != p) dfs(u, v); }
    tout[v] = ++timer; }
```

```
                    pushed = true; } }
            if (!pushed) {
                relabel(i);
                break; } } }
    int max_flow = 0;
    for (int i = 0; i < n; i++)
        max_flow += flow[i][t];
    return max_flow; }
```

2-sat:

```
int n;  vector<vector<int>> adj, adj_t;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;
void dfs1(int v) {
    used[v] = true;
    for (int u : adj[v]) { if (!used[u])  dfs1(u);
}
    order.push_back(v); }
void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : adj_t[v]) {
        if (comp[u] == -1)
            dfs2(u, cl); } }
bool solve_2SAT() {
    order.clear();
    used.assign(n, false);
    for (int i = 0; i < n; ++i) { if (!used[i])
dfs1(i); }
    comp.assign(n, -1);
```

```
bool is_ancestor(int u, int v){
    return tin[u] <= tin[v] && tout[u] >= tout[v]; }
int lca(int u, int v) {
    if (is_ancestor(u, v))  return u;
    if (is_ancestor(v, u)) return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    } return up[u][0]; }
void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root); }
```

```
Scc+condense:O(v +e)
vector<vector<int>> adj, adj_rev;
vector<bool> used;  vector<int> order, component;
void dfs1(int v) {
    used[v] = true;
    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);
    order.push_back(v);  }
void dfs2(int v) {
    used[v] = true;
    component.push_back(v);
```

```
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++); }
    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1]) return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }return true; }
void add_disjunction(int a, bool na, int b, bool nb) {
    a = 2*a ^ na;
    b = 2*b ^ nb;
    int neg_a = a ^ 1;
    int neg_b = b ^ 1;
    adj[neg_a].push_back(b);
    adj[neg_b].push_back(a);
    adj_t[b].push_back(neg_a);
    adj_t[a].push_back(neg_b); }
```

```
Strong orient:
vector<vector<pair<int, int>>> adj; // adjacency list - vertex and edge pairs
vector<pair<int, int>> edges;
vector<int> tin, low;
int bridge_cnt;
string orient;
vector<bool> edge_used;
void find_bridges(int v) {
    static int time = 0;
```

```
    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);   }
int main() {
    int n;
    for (;;) {
        int a, b; /*edge (a,b)*/
        adj[a].push_back(b);
        adj_rev[b].push_back(a); }
    used.assign(n, false);
    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);
    used.assign(n, false);
    reverse(order.begin(), order.end());
vector<int> roots(n, 0);
vector<int> root_nodes;
vector<vector<int>> adj_scc(n);
for (auto v : order)
if (!used[v]) {
dfs2 (v);
/*condense*/
int root = component.front();
for (auto u : component) roots[u] = root;
root_nodes.push_back(root);
component.clear();
}
for (int v = 0; v < n; v++) {
for (auto u : adj[v]) {
int root_v = roots[v], root_u = roots[u];
if (root_u != root_v)
```

```
    low[v] = tin[v] = time++;
    for (auto p : adj[v]) {
        if (edge_used[p.second]) continue;
        edge_used[p.second] = true;
        orient[p.second] = v ==
edges[p.second].first ? '>' : '<';
        int nv = p.first;
        if (tin[nv] == -1) { // if nv is not visited
yet
            find_bridges(nv);
            low[v] = min(low[v], low[nv]);
            if (low[nv] > tin[v]) {
                // a bridge between v and nv
                bridge_cnt++;
            }
        } else {
            low[v] = min(low[v], low[nv]);
        } } }
int main() {
    int n, m;
    scanf("%d %d", &n, &m);
    adj.resize(n);
    tin.resize(n, -1);
    low.resize(n, -1);
    orient.resize(m);
    edges.resize(m);
    edge_used.resize(m);
    for (int i = 0; i < m; i++) {
        int a, b;
        scanf("%d %d", &a, &b);
        a--; b--;
```

adj_scc[root_v].push_back(root_u); } }
return 0 ; }
	adj[a].push_back({b, i});
	adj[b].push_back({a, i});
	edges[i] = {a, b}; }
	int comp_cnt = 0;
	for (int v = 0; v < n; v++) {
	if (tin[v] == -1) {
	comp_cnt++;
	find_bridges(v); } }
	printf("%d\n%s\n", comp_cnt + bridge_cnt, orient.c_str()); return 0; }

Topo-sort : for(auto x: adj[v]) { if(!vis[x]) { vis[x]=1 ; topo(x); } } stk.push(x);

Bridges:O(n+m) void find_bridges(list<int> adj[]){ int timer=0; vector<bool> vis(n, 0); vector<int> tin(n, -1), low(n,-1) ; function<void(int, int)> dfs = [&](int v, int p) {vis[v]=1 ; tin[v]=low[v]=timer++ ; for(int to:adj[v]) { if(to==p) continue ; if(vis[to]) {low[v] = min(low[v], tin[to]) ; } else { dfs(to, v) ; low[v] = min(low[v], low[to]) ; if(low[to]>tin[v]) {/*add edge(v, to) to bridge */ } } } } ; for(i:0, n) { if(!vis[i]) { dfs(i, -1) ; } }

Articulations: O(v+e) void find_cutpoints(list<int> adj[n]) { int timer=0; vector<bool> vis(n, 0); vector<int> tin(n, -1), low(n, -1) ; function<void(int, int)> dfs = [&](int v, int p) { vis[v]=1 ; tin[v]=low[v]=timer++ ; int children=0; for(int to:adj[v]) { if (to == p) continue; if(vis[to]){ low[v] = min(low[v], tin[to]); } else { dfs(to, v); low[v] = min(low[v], low[to]); if (low[to] >= tin[v] && p!=-1) {/* add v to cutpoint */} ++children; } } if(p == -1 && children > 1) {/* add v to cutpoint */} }; for(int i:0, n) { if(!vis[i]) dfs(i, -1) ; } }

Floyd-warshal : for(k:0,V) { for(i:0,V) { for(j:0, V) { if(dist[i][j] > dist[i][k]+dist[k][j] && dist[k][j] !=INF && dist[i][k]!=INF) { dist[i][j] = dist[i][k] + dist[k][j];} } } }

---

4) Segment Tree

```
ll t[4*N]; ll
merge(ll a,ll b){
return a+b;

}
//build(1,0,n-1)--> for building,root node=1
void build(ll i,ll tl,ll tr) { if(tl==tr) {
t[i]=a[tl];

    } else{ ll tm=(tl+tr)/2;
    build(2*i,tl,tm);
    build(2*i+1,tm+1,tr);
    t[i]=merge(t[2*i],t[2*i+1]);

    }
}
//sum(1,0,n-1,l,r)-->to get sum from l to r,l &
r should be zero indexed

ll f(ll i,ll tl,ll tr,ll l,ll r) { if(tr<l ||
    tl>r) return 0; if(tl>=l &&
    tr<=r) return t[i]; else { ll
    tm=(tl+tr)/2; return
    merge(f(2*i,tl,tm,l,r),f(2*i+1,t
    m+1,tr,l,r)); }

}
//update(1,0,n-1,pos,new_val) void
update(ll i,ll tl,ll tr,ll pos,ll new_val) {
if(tl==tr) { t[i]=new_val;

    } else { ll
    tm=(tl+tr)/2;
    if(pos<=tm)

update(2*i,tl,tm,pos,new_val); else
        update(2*i+1,tm+1,tr,pos,new_val);
        t[i]=merge(t[2*i],t[2*i+1]);

    }
```

5) Lazy Propagation

```
ll t[4*N]; void build(ll a[], ll v, ll
tl, ll tr) { if (tl==tr) { t[v] = a[tl];

    } else { ll tm = (tl + tr) / 2;
    build(a, v*2, tl, tm); build(a,
    v*2+1, tm+1, tr); t[v] = 0;

    }
}

void update(ll v,ll tl,ll tr,ll l,ll r,ll add) {
    if (l>r) return; if (l==tl && r==tr){ t[v]
    += add;

    }else { ll tm = (tl + tr) / 2; update(v*2, tl,
        tm, l, min(r, tm), add); update(v*2+1,
        tm+1, tr, max(l, tm+1), r, add);

    }   }

ll get(ll v,ll tl,ll tr,ll pos) {
    if (tl == tr) return t[v];

    ll tm = (tl + tr) / 2; if (pos <= tm){
    return t[v] + get(v*2, tl, tm, pos);

    } else { return t[v] + get(v*2+1, tm+1, tr,
    pos); }

    }
```

HLD

```
vector<int> parent, depth, heavy, head, pos;

int cur_pos;

int dfs(int v, vector<vector<int>> const& adj)
{

    int size = 1;
```

PERSISTENT SEG TREE:

```
struct Vertex {
    Vertex *l, *r; int sum;
    Vertex(int val) : l(nullptr), r(nullptr), sum(val) {}
    Vertex(Vertex *l, Vertex *r) : l(l), r(r), sum(0) {
        if (l) sum += l->sum;
        if (r) sum += r->sum; } };
Vertex* build(int a[], int tl, int tr) {
    if (tl == tr) return new Vertex(a[tl]);
    int tm = (tl + tr) / 2;
    return new Vertex(build(a, tl, tm), build(a, tm+1, tr)); }
int get_sum(Vertex* v, int tl, int tr, int l, int r) {
    if (l > r) return 0;
    if (l == tl && tr == r) return v->sum;
    int tm = (tl + tr) / 2;
    return get_sum(v->l, tl, tm, l, min(r, tm)) + get_sum(v->r, tm+1, tr, max(l, tm+1), r); }
Vertex* update(Vertex* v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) return new Vertex(new_val);
    int tm = (tl + tr) / 2;
    if (pos <= tm)  return new Vertex(update(v->l, tl, tm, pos, new_val), v->r);
    else return new Vertex(v->l, update(v->r, tm+1, tr, pos, new_val)); }
```

Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
```

```
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        } }    return size; }
void decompose(int v, int h, vector<vector<int>> const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
            decompose(c, c, adj); } }
void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    cur_pos = 0;
    dfs(0, adj);
    decompose(0, 0, adj); }
int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
```

```
using namespace __gnu_pbds; typedef tree<ll, null_type, less<ll>, rb_tree_tag, tree_order_statistics_node_update> ordered_set; ordered_set o_set;

// Finding the second smallest element , in the set using * because , find_by_order returns an iterator cout<<*(o_set.find_by_order(1)) ;
```

Sparse table

```
int st[K + 1][MAXN];

std::copy(array.begin(), array.end(), st[0]);

for (int i = 1; i <= K; i++)

    for (int j = 0; j + (1 << i) <= N; j++)

        st[i][j] = f(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);

long long st[K + 1][MAXN];

std::copy(array.begin(), array.end(), st[0]);

for (int i = 1; i <= K; i++)

for (int j = 0; j + (1 << i) <= N; j++)

st[i][j] = st[i - 1][j] + st[i - 1][j + (1 << (i - 1))];

long long sum = 0;

for (int i = K; i >= 0; i--) {

if ((1 << i) <= R - L + 1) {

sum += st[i][L];

L += 1 << i; }}

int i = lg[R - L + 1];

int minimum = min(st[i][L], st[i][R - (1 << i) + 1]);
```

```
        int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
        res = max(res, cur_heavy_path_max);
    }
    if (depth[a] > depth[b])
        swap(a, b);
    int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
    res = max(res, last_heavy_path_max);
    return res; }
```

Derangements

```
int countDerrangements(int n) { if (n == 1 or n == 2) { return n - 1; } int a = 0, b = 1;

for (int i = 3; i <= n; ++i) { int cur = (i - 1) * (a + b); a = b; b = cur; } return b;

}
```

```
Knuth Optimisation int solve() {

int N; ... // read N and input int dp[N][N], opt[N][N];

auto C = [&](int i, int j) {

... // Implement cost function C.

};

for (int i = 0; i < N; i++) { opt[i][i] = i;

... // Initialize dp[i][i] according to the problem

}

for (int i = N-2; i >= 0; i--) { for (int j = i+1; j < N; j++) {

int mn = INT_MAX;

int cost = C(i, j); for (int k = opt[i][j-1]; k <= min(j-1, opt[i+1][j]); k++) {

if (mn >= dp[i][k] + dp[k+1][j] + cost) {

opt[i][j] = k; mn = dp[i][k] + dp[k+1][j] + cost;

} } dp[i][j] = mn;

} } cout << dp[0][N-1] << endl;

}
```

27) Centroid Decomposition

```
vector<int> tree[MAXN]; vector<int> centroidTree[MAXN]; bool centroidMarked[MAXN];

/* method to add edge between to nodes of the undirected tree */ void addEdge(int u, int v) {

tree[u].push_back(v); tree[v].push_back(u);

}

/* method to setup subtree sizes and nodes in current tree */ void DFS(int src, bool visited[],
int subtree_size[], int* n) {

/* mark node visited */ visited[src] = true;

/* increase count of nodes visited */

*n += 1;

/* initialize subtree size for current node*/ subtree_size[src] = 1; vector<int>::iterator it;

/* recur on non-visited and non-centroid neighbours */ for (it = tree[src].begin();
it!=tree[src].end(); it++)

if (!visited[*it] && !centroidMarked[*it]) { DFS(*it, visited, subtree_size, n);
subtree_size[src]+=subtree_size[*it];
```

```
} }

int getCentroid(int src, bool visited[], int subtree_size[], int n) { /* assume the current node to be centroid */ bool is_centroid = true; /* mark it as visited */ visited[src] = true;

/* track heaviest child of node, to use in case node is not centroid */

int heaviest_child = 0; vector<int>::iterator it;

/* iterate over all adjacent nodes which are children (not visited) and not marked as centroid to some subtree */

for (it = tree[src].begin(); it!=tree[src].end(); it++) if (!visited[*it] && !centroidMarked[*it]){

/* If any adjacent node has more than n/2 nodes,

* current node cannot be centroid */ if (subtree_size[*it]>n/2) is_centroid=false;

/* update heaviest child */ if (heaviest_child==0 || subtree_size[*it]>subtree_size[heaviest_child]) heaviest_child = *it; }

/* if current node is a centroid */

if (is_centroid && n-subtree_size[src]<=n/2) return src;

/* else recur on heaviest child */ return getCentroid(heaviest_child, visited, subtree_size, n); }

/* function to get the centroid of tree rooted at src.

* tree may be the original one or may belong to the forest */ int getCentroid(int src) { bool visited[MAXN]; int subtree_size[MAXN];

/* initialize auxiliary arrays */ memset(visited, false, sizeof visited); memset(subtree_size, 0, sizeof subtree_size);

/* variable to hold number of nodes in the current tree */ int n = 0;

/* DFS to set up subtree sizes and nodes in current tree */ DFS(src, visited, subtree_size, &n);

for (int i=1; i<MAXN; i++) visited[i] = false; int centroid = getCentroid(src, visited, subtree_size, n); centroidMarked[centroid]=true;

return centroid; }

/* function to generate centroid tree of tree rooted at src */ int decomposeTree(int root) {

//printf("decomposeTree(%d)\n", root);

/* get centroid for current tree */ int cend_tree = getCentroid(root); printf("%d ", cend_tree); vector<int>::iterator it;

/* for every node adjacent to the found centroid

* and not already marked as centroid */

for (it=tree[cend_tree].begin(); it!=tree[cend_tree].end(); it++) {

if (!centroidMarked[*it]) {
```

```
/* decompose subtree rooted at adjacent node */ int cend_subtree = decomposeTree(*it);

/* add edge between tree centroid and centroid of subtree */
centroidTree[cend_tree].push_back(cend_subtree);
centroidTree[cend_subtree].push_back(cend_tree); } }

/* return centroid of tree */ return cend_tree;}
```

Treap

```
struct Node {
int val, prio, size;
Node* child[2];
void apply() { /* apply lazy actions and push them down*/ }
void maintain() {
size = 1;
rep(i,0,2) size += child[i] ? child[i]->size : 0; } };
pair<Node*, Node*> split(Node* n, int val) { // returns (< val, >= val)
if (!n) return {0,0};
n->apply();
Node*& c = n->child[val > n->val];
auto sub = split(c, val);
if (val > n->val) { c = sub.fst; n->maintain(); return mk(n, sub.snd); }
else { c = sub.snd; n->maintain(); return mk(sub.fst, n); } }
Node* merge(Node* l, Node* r) {
if (!l || !r) return l ? l : r;
if (l->prio > r->prio) {
l->apply();
l->child[1] = merge(l->child[1], r);
l->maintain();
return l;} else {
r->apply();
r->child[0] = merge(l, r->child[0]);
r->maintain();
return r; }}
Node* insert(Node* n, int val) {
auto sub = split(n, val);
Node* x = new Node { val, rand(), 1 };
return merge(merge(sub.fst, x), sub.snd); }
Node* remove(Node* n, int val) { if (!n) return 0;
n->apply();
if (val == n->val)
return merge(n->child[0], n->child[1]);
Node*& c = n->child[val > n->val];
c = remove(c, val);
n->maintain();  return n;}
```