



# Team Code Reference

## Curiously Recurring

ACM-ICPC World Finals  
May 19, 2016

<b>1</b>	<b>Templates</b>	<b>1</b>	4.3	Minimum Cut Inference . .	13
1.1	Vimrc . . . . .	1	4.4	Min cost flow . . . . .	14
1.2	C++ Template . . . . .	1	4.5	Min edge capacities . . . . .	14
1.3	Java Template . . . . .	1	4.6	Min vertex capacities . . . . .	14
<b>2</b>	<b>Data Structures</b>	<b>2</b>	<b>5</b>	<b>Combinatorics &amp; Probability</b>	<b>14</b>
2.1	Union Find . . . . .	2	5.1	Stable Marriage Problem . .	14
2.2	Max Queue . . . . .	2	5.2	KP procedure . . . . .	15
2.3	Fenwick Tree . . . . .	2	5.3	2-SAT . . . . .	15
2.4	2D Fenwick Tree . . . . .	2	<b>6</b>	<b>Geometry</b>	<b>15</b>
2.5	Sparse Table . . . . .	3	6.1	Essentials . . . . .	15
2.6	Segment Tree . . . . .	3	6.2	Convex Hull . . . . .	16
2.7	Lazy Dynamic Segment Tree	3	6.3	Upper envelope . . . . .	17
2.8	Implicit Cartesian Tree . . .	3	6.4	Formulae . . . . .	17
2.9	KD tree . . . . .	4	<b>7</b>	<b>Mathematics</b>	<b>17</b>
2.10	AVL Tree . . . . .	5	7.1	Primes . . . . .	17
2.11	Treap . . . . .	6	7.2	Euler Phi . . . . .	18
2.12	Prefix Trie . . . . .	7	7.3	Number theoretic algorithms	18
2.13	Suffix Array . . . . .	7	7.4	Lucas' theorem . . . . .	18
2.14	Suffix Tree . . . . .	7	7.5	Finite Field . . . . .	19
2.15	Suffix Automaton . . . . .	8	7.6	Complex Numbers . . . . .	19
2.16	Built-in datastructures . . .	8	7.7	Fast Fourier Transform . . .	19
<b>3</b>	<b>Basic Graph algorithms</b>	<b>8</b>	7.8	Matrix equation solver . . .	19
3.1	Edge Classification . . . . .	8	7.9	Matrix Exponentiation . . .	20
3.2	Topological sort . . . . .	9	7.10	Simplex algorithm . . . . .	20
3.3	Tarjan: SCCs . . . . .	9	7.11	Game theory . . . . .	21
3.4	Biconnected components . .	9	<b>8</b>	<b>Strings</b>	<b>21</b>
3.5	Kruskal's algorithm . . . . .	10	8.1	Knuth Morris Pratt . . . . .	21
3.6	Prim's algorithm . . . . .	10	8.2	Z-algorithm . . . . .	21
3.7	Dijkstra's algorithm . . . . .	10	8.3	Aho-Corasick . . . . .	21
3.8	Bellman-Ford . . . . .	10	8.4	Manacher's Algorithm . . .	22
3.9	Floyd-Warshall algorithm .	11	<b>9</b>	<b>Miscellaneous</b>	<b>22</b>
3.10	Johnson's reweighting algo-	11	9.1	LIS . . . . .	22
	rithm . . . . .	11	9.2	Randomisation . . . . .	23
3.11	Hierholzer's algorithm . . .	11	9.3	All Nearest Smaller Values .	23
3.12	Bron-Kerbosch . . . . .	11	<b>10</b>	<b>Helpers</b>	<b>23</b>
3.13	Theorems in Graph Theory	11	10.1	Golden Section Search . . .	23
3.14	Centroid Decomposition . .	12	10.2	Binary Search . . . . .	23
3.15	Heavy-Light decomposition	12	10.3	Bitmasking . . . . .	23
3.16	HLD with Segtree . . . . .	12	10.4	Fast IO . . . . .	24
<b>4</b>	<b>Flow and Matching</b>	<b>13</b>	10.5	Detecting overflow . . . . .	24
4.1	Flow Graph . . . . .	13	<b>11</b>	<b>Strategies</b>	<b>25</b>
4.2	Dinic . . . . .	13			

## 1 Templates

### 1.1 Vimrc

```
1 syntax on noet wrap lbr nu is cin ai
2 ts=4 sts=4 sw=4 mouse=nvc cb=unnamed bs=indent,eol,start cino=:0,l1,g0,(0
```

### 1.2 C++ Template

```
1 // #define _GLIBCXX_DEBUG
2 #include <bits/stdc++.h>
3 // iostream string sstream vector list set map queue stack bitset
4 // tuple cstdio numeric iterator algorithm cmath chrono cassert
5 using namespace std; // :s/ /\r/g :s/\w*/#include <\0>/g
6 #define REP(i,n) for(auto i = decltype(n)(0); i<(n); i++)
7 #define all(x) x.begin(), x.end()
8 using ll = long long; using ld = long double; using vi = vector<ll>;
9 const bool LOG = false; void Log() { if(LOG) cerr << "\n"; }
10 template<class T, class... S> void Log(T t, S... s){
11     if(LOG) cerr << t << "\t", Log(s...); }
12 int main(){ ios::sync_with_stdio(false); cin.tie(nullptr); return 0; }
```

### 1.3 Java Template

```
1 import java.io.OutputStream;
2 import java.io.InputStream;
3 import java.io.PrintWriter;
4 import java.util.StringTokenizer;
5 import java.io.BufferedReader;
6 import java.io.InputStreamReader;
7 import java.io.IOException;
8 import java.io.IOException;
9
10 import java.util.Arrays;
11 import java.math.BigInteger;
12
13 public class Main { // Check what this should be called
14     public static void main(String[] args) {
15         InputReader in = new InputReader(System.in);
16         PrintWriter out = new PrintWriter(System.out);
17         Solver s = new Solver();
18         s.solve(in, out);
19         out.close();
20     }
21
22     static class Solver {
23         public void solve(InputReader in, PrintWriter out) {
24             // solve
25         }
26     }
27
28     static class InputReader {
29         public BufferedReader reader;
30         public StringTokenizer tokenizer;
31         public InputReader(InputStream st) {
32             reader = new BufferedReader(new InputStreamReader(st), 32768);
```

```

33     tokenizer = null;
34 }
35 public String next() {
36     while (tokenizer == null || !tokenizer.hasMoreTokens()) {
37         try {
38             String s = reader.readLine();
39             if (s == null) {
40                 tokenizer = null; break; }
41             if (s.isEmpty()) continue;
42             tokenizer = new StringTokenizer(s);
43         } catch (IOException e) {
44             throw new RuntimeException(e);
45         }
46     }
47     return (tokenizer != null && tokenizer.hasMoreTokens()
48         ? tokenizer.nextToken() : null);
49 }
50 public int nextInt() {
51     String s = next();
52     if (s != null) return Integer.parseInt(s);
53     else return -1; // handle appropriately
54 }
55 }
56 }

```

## 2 Data Structures

### 2.1 Union Find

```

1 struct UnionFind {
2     vi par, rank, size; int c;
3     UnionFind(int n) : par(n), rank(n,0), size(n,1), c(n) {
4         for (int i = 0; i < n; ++i) par[i] = i;
5     }
6
7     int find(int i) { return (par[i] == i ? i : (par[i] = find(par[i]))); }
8     bool same(int i, int j) { return find(i) == find(j); }
9     int get_size(int i) { return size[find(i)]; }
10    int count() { return c; }
11
12    void merge(int i, int j) {
13        if ((i = find(i)) == (j = find(j))) return;
14        c--;
15        if (rank[i] > rank[j]) swap(i, j);
16        par[i] = j; size[j] += size[i];
17        if (rank[i] == rank[j]) rank[j]++;
18    }
19 };

```

### 2.2 Max Queue

deque runs in amortized constant time. Can be modified to query minimum, gcd/lcm, set union/intersection (use bitmasks), etc.

```

1 template <class T>
2 class MaxQueue {
3 public:

```

```

4     stack< pair<T, T> > inbox, outbox;
5     void enqueue(T val) {
6         T m = val;
7         if (!inbox.empty()) m = max(m, inbox.top().second);
8         inbox.push(pair<T, T>(val, m));
9     }
10    bool dequeue(T* d = nullptr) {
11        if (outbox.empty() && !inbox.empty()) {
12            pair<T, T> t = inbox.top(); inbox.pop();
13            outbox.push(pair<T, T>(t.first, t.first));
14            while (!inbox.empty()) {
15                t = inbox.top(); inbox.pop();
16                T m = max(t.first, outbox.top().second);
17                outbox.push(pair<T, T>(t.first, m));
18            }
19        }
20        if (outbox.empty()) return false;
21        else {
22            if (d != nullptr) *d = outbox.top().first;
23            outbox.pop();
24            return true;
25        }
26    }
27    bool empty() { return outbox.empty() && inbox.empty(); }
28    size_t size() { return outbox.size() + inbox.size(); }
29    T get_max() {
30        if (outbox.empty()) return inbox.top().second;
31        if (inbox.empty()) return outbox.top().second;
32        return max(outbox.top().second, inbox.top().second);
33    }
34 };

```

### 2.3 Fenwick Tree

The tree is 1-based! Use indices 1.. $n$ .

```

1 template <class T>
2 struct FenwickTree { // use 1 based indices!!!
3     int n; vector<T> tree;
4     FenwickTree(int n) : n(n) { tree.assign(n + 1, 0); }
5     T query(int l, int r) { return query(r) - query(l - 1); }
6     T query(int r) {
7         T s = 0;
8         for(; r > 0; r -= (r & (-r))) s += tree[r];
9         return s;
10    }
11    void update(int i, T v) {
12        for(; i <= n; i += (i & (-i))) tree[i] += v;
13    }
14 };

```

### 2.4 2D Fenwick Tree

Can easily be extended to any dimension.

```

1 template <class T>
2 struct FenwickTree2D {

```

```

3 vector< vector<T> > tree;
4 int n;
5 FenwickTree2D(int n) : n(n) { tree.assign(n + 1, vector<T>(n + 1, 0)); }
6 T query(int x1, int y1, int x2, int y2) {
7     return query(x2,y2)+query(x1-1,y1-1)-query(x2,y1-1)-query(x1-1,y2);
8 }
9 T query(int x, int y) {
10     T s = 0;
11     for (int i = x; i > 0; i -= (i & (-i)))
12         for (int j = y; j > 0; j -= (j & (-j)))
13             s += tree[i][j];
14     return s;
15 }
16 void update(int x, int y, T v) {
17     for (int i = x; i <= n; i += (i & (-i)))
18         for (int j = y; j <= n; j += (j & (-j)))
19             tree[i][j] += v;
20 }
21 };

```

## 2.5 Sparse Table

For  $O(1)$  range minimum query with  $O(n \lg n)$  precalculation.

```

1 using T = double; using vt = vector<T>; using vvt = vector<vt>;
2 struct SparseTable{
3     vvt d;
4     SparseTable(vt &a) : d(vvt{a}) {
5         int N = a.size();
6         for(auto s = 1; 2*s <= N; s *= 2){
7             d.push_back(vt(N - 2*s + 1));
8             auto &n = d.back(); auto &l = d[d.size()-2];
9             for(int i = 0; i + 2*s <= N; ++i) n[i] = min(l[i], l[i+s]);
10        }
11    }
12    int rmq(int l, int r){ // 0 <= l <= r < a.size()
13        int p = 8*sizeof(int) - 1 - __builtin_clz(r+1-l);
14        return min(d[p][l], d[p][r+1-(1<<p)]);
15    }
16 };

```

## 2.6 Segment Tree

The range should be of the form  $2^p$ .

```

1 template <class T, T(*op)(T, T), T ident>
2 struct SegmentTree {
3     int n; vector<T> tree;
4     SegmentTree(vector<T> &init) : n(init.size()), tree(2 * n, ident) {
5         copy(init.begin(), init.end(), tree.begin() + n);
6         for (int j = n - 1; j > 0; --j)
7             tree[j] = op(tree[2*j], tree[2*j+1]);
8     }
9     void update(int i, T val) {
10         for (tree[i+n] = val, i = (i+n)/2; i > 0; i /= 2)
11             tree[i] = op(tree[2*i], tree[2*i+1]);
12     }

```

```

13 T query(int l, int r) {
14     T lhs = T(ident), rhs = T(ident);
15     for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
16         if (l&1) lhs = op(lhs, tree[l++]);
17         if (!(r&1)) rhs = op(tree[r--], rhs);
18     }
19     return op(l == r ? op(lhs, tree[l]) : lhs, rhs);
20 }
21 };

```

## 2.7 Lazy Dynamic Segment Tree

```

1 using T=ll; using U=ll; // exclusive right bounds
2 T t_id; U u_id;
3 T op(T a, T b){ return a+b; }
4 void join(U &a, U b){ a+=b; }
5 void apply(T &t, U u, int x){ t+=x*u; }
6 T part(T t, int r, int p){ return t/r*p; }
7 struct DynamicSegmentTree {
8     struct Node { int l, r, lc, rc; T t; U u;
9         Node(int l, int r):l(l),r(r),lc(-1),rc(-1),t(t_id),u(u_id){}
10    };
11    vector<Node> tree;
12    DynamicSegmentTree(int N) { tree.push_back({0,N}); }
13    void push(Node &n, U u){ apply(n.t, u, n.r-n.l); join(n.u,u); }
14    void push(Node &n){push(tree[n.lc],n.u);push(tree[n.rc],n.u);n.u=u_id;}
15    T query(int l, int r, int i = 0) { auto &n = tree[i];
16        if(r <= n.l || n.r <= l) return t_id;
17        if(l <= n.l && n.r <= r) return n.t;
18        if(n.lc < 0) return part(n.t, n.r-n.l, min(n.r,r)-max(n.l,l));
19        return push(n, op(query(l,r,n.lc),query(l,r,n.rc)));
20    }
21    void update(int l, int r, U u, int i = 0) { auto &n = tree[i];
22        if(r <= n.l || n.r <= l) return;
23        if(l <= n.l && n.r <= r) return push(n,u);
24        if(n.lc < 0) { int m = (n.l + n.r) / 2;
25            n.lc = tree.size(); n.rc = n.lc+1;
26            tree.push_back({tree[i].l, m}); tree.push_back({m, tree[i].r});
27        }
28        push(tree[i]); update(l,r,u,tree[i].lc); update(l,r,u,tree[i].rc);
29        tree[i].t = op(tree[tree[i].lc].t, tree[tree[i].rc].t);
30    }
31 };

```

## 2.8 Implicit Cartesian Tree

The indices are zero-based. Also, don't forget to initialise the empty tree to NULL. (Pretty much) all operations take  $O(\log n)$  time.

```

1 struct Node {
2     ll val, mx;
3     int size, priority;
4     bool rev = false;
5     Node *l = NULL, *r = NULL;
6     Node(ll _val) : val(_val), mx(_val), size(1) { priority = rand(); }
7 };

```

```

8 int size(Node *p) { return p == NULL ? 0 : p->size; }
9 ll getmax(Node *p) { return p == NULL ? -LLINF : p->mx; }
10 void update(Node *p) {
11     if (p == NULL) return;
12     p->size = 1 + size(p->l) + size(p->r);
13     p->mx = max(p->val, max(getmax(p->l), getmax(p->r)));
14 }
15 void propagate(Node *p) {
16     if (p == NULL || !p->rev) return;
17     swap(p->l, p->r);
18     if (p->l != NULL) p->l->rev ^= true;
19     if (p->r != NULL) p->r->rev ^= true;
20     p->rev = false;
21 }
22 void merge(Node *t, Node *l, Node *r) {
23     propagate(l); propagate(r);
24     if (l == NULL) { t = r; }
25     else if (r == NULL) { t = l; }
26     else if (l->priority > r->priority) {
27         merge(l->r, l->r, r); t = l; }
28     else { merge(r->l, l, r->l); t = r; }
29     update(t);
30 }
31 void split(Node *t, Node *&l, Node *&r, int at) {
32     propagate(t);
33     if (t == NULL) { l = r = NULL; return; }
34     int id = size(t->l) + 1;
35     if (id > at) { split(t->l, l, t->l, at); r = t; }
36     else { split(t->r, t->r, r, at - id); l = t; }
37     update(t);
38 }
39 void insert(Node *t, ll val, int pos) {
40     propagate(t);
41     Node *n = new Node(val), *l, *r;
42     split(t, l, r, pos);
43     merge(t, l, n);
44     merge(t, t, r);
45 }
46 void erase(Node *t, int pos, bool del = true) {
47     propagate(t);
48     Node *L, *rm;
49     split(t, t, L, pos);
50     split(L, rm, L, 1);
51     merge(t, t, L);
52     if (del && rm != NULL) delete rm;
53 }
54 void reverse(Node *t, int l, int r) {
55     propagate(t);
56     Node *L, *R;
57     split(t, t, L, l);
58     split(L, L, R, r - l + 1);
59     if (L != NULL) L->rev = true;
60     merge(t, t, L);
61     merge(t, t, R);
62 }
63 ll at(Node *t, int pos) {
64     propagate(t);
65     int id = size(t->l);

```

```

66     if (pos == id) return t->val;
67     else if (ps > id) return at(t->r, pos - id - 1);
68     else return at(t->l, pos);
69 }
70 ll range_maximum(Node *t, int l, int r) {
71     propagate(t);
72     Node *L, *R;
73     split(t, t, L, l);
74     split(L, L, R, r - l + 1);
75     ll ret = getmax(L);
76     merge(t, t, L);
77     merge(t, t, R);
78     return ret;
79 }
80 void cleanup(Node *p) {
81     if (p == NULL) return;
82     cleanup(p->l); cleanup(p->r);
83     delete p;
84 }

```

## 2.9 KD tree

```

1 struct P{ ll x,y; };
2 struct Box{
3     ll xl, xh, yl, yh;
4     Box(ll xl=-LLINF, ll xh=-LLINF, ll yl=LLINF, ll yh=LLINF) :
5         xl(xl), xh(xh), yl(yl), yh(yh) {}
6     bool contains(const P &p) const {
7         return xl <= p.x && p.x <= xh && yl <= p.y && p.y <= yh;
8     }
9     bool contains(const Box &b) const {
10         return xl <= b.xl && b.xh <= xh && yl <= b.yl && b.yh <= yh;
11     }
12     bool disjunct(const Box &b) const {
13         return xh < b.xl || b.xh < xl || yh < b.yl || b.yh < yl;
14     }
15 };
16 struct Node {
17     ll i, cl, cr; bool hz;
18     Node(ll i, bool h) : i{i}, cl{-1}, cr{-1}, hz{h} {};
19 };
20 struct KDTree {
21     vector<P> &ps; vector<Node> tree;
22     KDTree(vector<P> &ps) : ps{ps} {
23         vi x(ps.size()); iota(x.begin(), x.end(), 0); vi y(x);
24         sort(x.begin(), x.end(), [&](ll l, ll r){ return compx(l,r); });
25         sort(y.begin(), y.end(), [&](ll l, ll r){ return compy(l,r); });
26         tree.reserve(ps.size());
27         build(x, y, true);
28     }
29     bool compx(ll l,ll r){return tie(ps[l].x,ps[l].y,l)<tie(ps[r].x,ps[r].y,r);}
30     bool compy(ll l,ll r){return tie(ps[l].y,ps[l].x,l)<tie(ps[r].y,ps[r].x,r);}
31     int build(vi &x, vi &y, bool h){
32         if(x.size()==0) return -1;
33         ll m = x.size()/2, n = tree.size();
34         vi xl, xh, yl, yh;
35         if(h){ // horizontal

```

```

36     ll s = x[m]; tree.push_back({s, h});
37     xh.assign(x.begin()+m+1, x.end()), xl = move(x);
38     xl.resize(m);
39     for(const auto &p : y)
40         if(p==s) continue;
41         else if(compx(p,s)) yl.push_back(p);
42         else yh.push_back(p);
43 } else { // vertical
44     ll s = y[m]; tree.push_back({s, h});
45     yh.assign(y.begin()+m+1, y.end()), yl = move(y);
46     yl.resize(m);
47     for(const auto &p : x)
48         if(p==s) continue;
49         else if(compy(p,s)) xl.push_back(p);
50         else xh.push_back(p);
51 }
52 tree[n].cl = build(xl,yl,!h); tree[n].cr = build(xh,yh,!h);
53 return n;
54 }
55 vi ans; // returns a list of indices in ps
56 vi query(const Box &q){ ans.clear(); query(q, Box(), 0); return ans; }
57 void query(const Box &q, const Box &b, ll n){
58     auto &node = tree[n]; auto &p = ps[node.i];
59     if(q.contains(b)){ allq(n); return; }
60     if(q.disjunct(b)) return;
61     if(q.contains(p)) ans.push_back(node.i);
62     Box b1=b, b2=b;
63     if(node.hz) b1.xh = b2.xl = p.x;
64     else b1.yh = b2.yl = p.y;
65     query(q,b1,node.cl); query(q,b2,node.cr);
66 }
67 void allq(ll n){ if(n==-1) return;
68     ans.push_back(tree[n].i); allq(tree[n].cl); allq(tree[n].cr);
69 }
70 };

```

## 2.10 AVL Tree

Can be augmented to support in  $O(\log n)$  time: range queries/updates (similar to a segment tree), insert at position  $n$ /query for position  $n$ , order statistics, etc.

```

1  template <class T>
2  struct AVL_Tree {
3      struct AVL_Node {
4          T val;
5          AVL_Node *p, *l, *r;
6          int size, height;
7          AVL_Node(T &_val, AVL_Node *_p = NULL)
8              : val(_val), p(_p), l(NULL), r(NULL), size(1), height(0) { }
9      };
10     AVL_Node *root;
11     AVL_Tree() : root(NULL) { }
12
13     // Querying
14     AVL_Node *find(T &key) { // O(lg n)
15         AVL_Node *c = root;
16         while (c != NULL && c->val != key) {
17             if (c->val < key) c = c->r;

```

```

18             else c = c->l;
19         }
20         return c;
21     }
22     // maximum and predecessor can be written in a similar manner
23     AVL_Node *minimum(AVL_Node *n) { // O(lg n)
24         if (n != NULL) while (n->l != NULL) n = n->l; return n;
25     }
26     AVL_Node *minimum() { return minimum(root); } // O(lg n)
27     AVL_Node *successor(AVL_Node *n) { // O(lg n)
28         if (n->r != NULL) return minimum(n->r);
29         AVL_Node *p = n->p;
30         while (p != NULL && n == p->r) { n = p; p = n->p; }
31         return p;
32     }
33
34     // Modification
35     AVL_Node *insert(T &nval) { // O(lg n)
36         AVL_Node *p = NULL, *c = root;
37         while (c != NULL) {
38             p = c;
39             c = (c->val < nval ? c->r : c->l);
40         }
41         AVL_Node *r = new AVL_Node(nval, p);
42         (p == NULL ? root : (
43             nval < p->val ? p->l : p->r)) = r;
44         _fixup(r);
45         return r;
46     }
47     void remove(AVL_Node *n, bool del = true) { // O(lg n)
48         if (n == NULL) return;
49         if (n->l != NULL && n->r != NULL) {
50             AVL_Node *y = successor(n), *z = y->par;
51             if (z != n) {
52                 _transplant(y, y->r);
53                 y->r = n->r;
54                 y->r->p = y;
55             }
56             _transplant(n, y);
57             y->l = n->l;
58             y->l->p = y;
59             _fixup(z->r == NULL ? z : z->r);
60             if (del) delete n;
61             return;
62         } else if (n->l != NULL) {
63             _pchild(n) = n->l;
64             n->l->p = n->p;
65         } else if (n->r != NULL) {
66             _pchild(n) = n->r;
67             n->r->p = n->p;
68         } else _pchild(n) = NULL;
69         _fixup(n->p);
70         if (del) delete n;
71     }
72     void cleanup() { _cleanup(root); }
73
74     // Helpers
75     void _transplant(AVL_Node *u, AVL_Node *v) {

```

```

76     _pchild(u) = v;
77     if (v != NULL) v->p = u->p;
78 }
79 AVL_Node *&_pchild(AVL_Node *n) {
80     return (n == NULL ? root : (n->p == NULL ? root :
81         (n->p->l == n ? n->p->l : n->p->r)));
82 }
83 void _augmentation(AVL_Node *n) {
84     if (n == NULL) return;
85     n->height = 1 + max(_get_height(n->l), _get_height(n->r));
86     n->size = 1 + _get_size(n->l) + _get_size(n->r);
87 }
88 int _get_height(AVL_Node *n) { return (n == NULL ? 0 : n->height); }
89 int _get_size(AVL_Node *n) { return (n == NULL ? 0 : n->size); }
90 bool _balanced(AVL_Node *n) {
91     return (abs(_get_height(n->l) - _get_height(n->r)) <= 1);
92 }
93 bool _leans_left(AVL_Node *n) {
94     return _get_height(n->l) > _get_height(n->r);
95 }
96 bool _leans_right(AVL_Node *n) {
97     return _get_height(n->r) > _get_height(n->l);
98 }
99 #define ROTATE(L, R) \
100     AVL_Node *o = n->R; \
101     n->R = o->L; \
102     if (o->L != NULL) o->L->p = n; \
103     o->p = n->p; \
104     _pchild(n) = o; \
105     o->L = n; \
106     n->p = o; \
107     _augmentation(n); \
108     _augmentation(o);
109 void _left_rotate(AVL_Node *n) { ROTATE(l, r); }
110 void _right_rotate(AVL_Node *n) { ROTATE(r, l); }
111 void _fixup(AVL_Node *n) {
112     while (n != NULL) {
113         _augmentation(n);
114         if (!_balanced(n)) {
115             if (_leans_left(n)&&_leans_right(n->l)) _left_rotate(n->l);
116             else if (_leans_right(n)&& _leans_left(n->r))
117                 _right_rotate(n->r);
118             if (_leans_left(n)) _right_rotate(n);
119             if (_leans_right(n)) _left_rotate(n);
120         }
121         n = n->p;
122     }
123 }
124 void _cleanup(AVL_Node *n) {
125     if (n->l != NULL) _cleanup(n->l);
126     if (n->r != NULL) _cleanup(n->r);
127 }
128 };

```

## 2.11 Treap

Can be used like the built-in `set`, except that it also supports order statistics, can be merged/split in  $O(\log n)$  time, can support range queries, and more.

```

1 struct Node {
2     ll val;
3     int size, priority;
4     Node *l = NULL, *r = NULL;
5     Node(ll _v) : val(_v), size(1) { priority = rand(); }
6 };
7
8 int size(Node *p) { return p == NULL ? 0 : p->size; }
9 void update(Node *p) {
10     if (p == NULL) return;
11     p->size = 1 + size(p->l) + size(p->r);
12 }
13 void merge(Node *&t, Node *l, Node *r) {
14     if (l == NULL) { t = r; }
15     else if (r == NULL) { t = l; }
16     else if (l->priority > r->priority) {
17         merge(l->r, l->r, r); t = l;
18     } else {
19         merge(r->l, l, r->l); t = r;
20     } update(t);
21 }
22 void split(Node *t, Node *&l, Node *&r, ll val) {
23     if (t == NULL) { l = r = NULL; return; }
24     if (t->val >= val) { // val goes with the right set
25         split(t->l, l, t->l, val); r = t;
26     } else {
27         split(t->r, t->r, r, val); l = t;
28     } update(t);
29 }
30 bool insert(Node *&t, ll val) {
31     // returns false if the element already existed
32     Node *n = new Node(val), *l, *r;
33     split(t, l, t, val);
34     split(t, t, r, val + 1);
35     bool empty = (t == NULL);
36     merge(t, l, n);
37     merge(t, t, r);
38     return empty;
39 }
40 void erase(Node *&t, ll val, bool del = true) {
41     // returns false if the element did not exist
42     Node *l, *rm;
43     split(t, l, t, val);
44     split(t, rm, t, val + 1);
45     bool exists = (t != NULL);
46     merge(t, l, t);
47     if (del && rm != NULL) delete rm;
48     return exists;
49 }
50 void cleanup(Node *p) {
51     if (p == NULL) return;
52     cleanup(p->l); cleanup(p->r);
53     delete p;

```

54 }

## 2.12 Prefix Trie

```

1  const int ALPHABET_SIZE = 26;
2  inline int mp(char c) { return c - 'a'; }
3
4  struct Node {
5      Node* ch[ALPHABET_SIZE];
6      bool isleaf = false;
7      Node() {
8          for(int i = 0; i < ALPHABET_SIZE; ++i) ch[i] = nullptr;
9      }
10
11     void insert(string &s, int i = 0) {
12         if (i == s.length()) isleaf = true;
13         else {
14             int v = mp(s[i]);
15             if (ch[v] == nullptr)
16                 ch[v] = new Node();
17             ch[v]->insert(s, i + 1);
18         }
19     }
20
21     bool contains(string &s, int i = 0) {
22         if (i == s.length()) return isleaf;
23         else {
24             int v = mp(s[i]);
25             if (ch[v] == nullptr) return false;
26             else return ch[v]->contains(s, i + 1);
27         }
28     }
29
30     void cleanup() {
31         for (int i = 0; i < ALPHABET_SIZE; ++i)
32             if (ch[i] != nullptr) {
33                 ch[i]->cleanup();
34                 delete ch[i];
35             }
36     }
37 };

```

## 2.13 Suffix Array

Note: dont forget to invert the returned array. **Complexity:**  $O(n \log n)$

```

1  string s;
2  int n;
3  vvi P;
4  SuffixArray(string &s) : s(_s), n(_s.length()) { construct(); }
5  void construct() {
6      P.push_back(vi(n, 0));
7      compress();
8      vi occ(n + 1, 0), s1(n, 0), s2(n, 0);
9      for (int k = 1, cnt = 1; cnt / 2 < n; ++k, cnt *= 2) {
10         P.push_back(vi(n, 0));
11         fill(occ.begin(), occ.end(), 0);

```

```

12         for (int i = 0; i < n; ++i)
13             occ[i+cnt<n ? P[k-1][i+cnt]+1 : 0]++;
14         partial_sum(occ.begin(), occ.end(), occ.begin());
15         for (int i = n - 1; i >= 0; --i)
16             s1[--occ[i+cnt<n ? P[k-1][i+cnt]+1 : 0]] = i;
17         fill(occ.begin(), occ.end(), 0);
18         for (int i = 0; i < n; ++i)
19             occ[P[k-1][s1[i]]]++;
20         partial_sum(occ.begin(), occ.end(), occ.begin());
21         for (int i = n - 1; i >= 0; --i)
22             s2[--occ[P[k-1][s1[i]]]] = s1[i];
23         for (int i = 1; i < n; ++i) {
24             P[k][s2[i]] = same(s2[i], s2[i - 1], k, cnt)
25                 ? P[k][s2[i - 1]] : i;
26         }
27     }
28 }
29 bool same(int i, int j, int k, int l) {
30     return P[k - 1][i] == P[k - 1][j]
31         && (i + 1 < n ? P[k - 1][i + 1] : -1)
32         == (j + 1 < n ? P[k - 1][j + 1] : -1);
33 }
34 void compress() {
35     vi cnt(256, 0);
36     for (int i = 0; i < n; ++i) cnt[s[i]]++;
37     for (int i = 0, mp = 0; i < 256; ++i)
38         if (cnt[i] > 0) cnt[i] = mp++;
39     for (int i = 0; i < n; ++i) P[0][i] = cnt[s[i]];
40 }
41 vi &get_array() { return P.back(); }
42 int lcp(int x, int y) {
43     int ret = 0;
44     if (x == y) return n - x;
45     for (int k = P.size() - 1; k >= 0 && x < n && y < n; --k)
46         if (P[k][x] == P[k][y]) {
47             x += 1 << k;
48             y += 1 << k;
49             ret += 1 << k;
50         }
51     return ret;
52 }
53 };

```

## 2.14 Suffix Tree

**Complexity:**  $O(n)$

```

1  using T = char;
2  using M = map<T, int>;           // or array<T, ALPHABET_SIZE>
3  using V = string;               // could be vector<T> as well
4  using It = V::const_iterator;
5  struct Node{
6      It b, e; M edges; int link;   // end is exclusive
7      Node(It b, It e) : b(b), e(e), link(-1) {}
8      int size() const { return e-b; }
9  };
10 struct SuffixTree{
11     const V &s; vector<Node> t;

```



```

12 int root,n,len,remainder,llink; It edge;
13 SuffixTree(const V &s) : s(s) { build(); }
14 int add_node(It b, It e){ return t.push_back({b,e}), t.size()-1; }
15 int add_node(It b){ return add_node(b,s.end()); }
16 void link(int node){ if(llink) t[llink].link = node; llink = node; }
17 void build(){
18     len = remainder = 0; edge = s.begin();
19     n = root = add_node(s.begin(), s.begin());
20     for(auto i = s.begin(); i != s.end(); ++i){
21         ++remainder; llink = 0;
22         while(remainder){
23             if(len == 0) edge = i;
24             if(t[n].edges[*edge] == 0){ // add new leaf
25                 t[n].edges[*edge] = add_node(i); link(n);
26             } else {
27                 auto x = t[n].edges[*edge]; // neXt node [with edge]
28                 if(len >= t[x].size()){ // walk to next node
29                     len -= t[x].size(); edge += t[x].size(); n = x;
30                     continue;
31                 }
32                 if(*(t[x].b + len) == *i){ // walk along edge
33                     ++len; link(n); break;
34                 } // split edge
35                 auto split = add_node(t[x].b, t[x].b+len);
36                 t[n].edges[*edge] = split;
37                 t[x].b += len;
38                 t[split].edges[*i] = add_node(i);
39                 t[split].edges[*t[x].b] = x;
40                 link(split);
41             }
42             --remainder;
43             if(n == root && len > 0)
44                 --len, edge = i - remainder + 1;
45             else n = t[n].link > 0 ? t[n].link : root;
46         }
47     }
48 }
49 };

```

## 2.15 Suffix Automaton

Complexity:  $O(n)$

```

1 using T = char; using M = map<T,int>; using V = string;
2 struct Node { // s: start, len: length, link: suffix link, e: edges
3     int s, len, link; M e; bool term; // term: terminal node?
4     Node(int s, int len, int link=-1):s(s), len(len), link(link), term(0) {}
5 };
6 struct SuffixAutomaton{
7     const V &s; vector<Node> t; int l; // string; tree; last added state
8     SuffixAutomaton(const V &s) : s(s) { build(); }
9     void build(){
10         l = t.size(); t.push_back({0,-1}); // root node
11         for(auto c : s){
12             int p=l, x=t.size(); t.push_back({0,t[l].len + 1}); // new node
13             while(p>0 && t[p].e[c] == 0) t[p].e[c] = x, p = t[p].link;
14             if(p<0) t[x].link = 0; // at root
15             else {

```

```

16         int q = t[p].e[c]; // the c-child of q
17         if(t[q].len == t[p].len + 1) t[x].link = q;
18         else { // cloning of q
19             int cl = t.size(); t.push_back(t[q]);
20             t[cl].len = t[p].len + 1;
21             t[cl].s = t[q].s + t[q].len - t[p].len - 1;
22             t[x].link = t[q].link = cl;
23             while(p >= 0 && t[p].e.count(c) > 0 && t[p].e[c] == q)
24                 t[p].e[c] = cl, p = t[p].link; // relink suffix
25             }
26         }
27         l = x; // update last
28     }
29     while(l>=0) t[l].term = true, l = t[l].link;
30 }
31 };

```

## 2.16 Built-in datastructures

```

1 // Minimum Heap
2 #include <queue>
3 template<class T>
4 using min_queue = priority_queue<T, vector<T>, greater<T>>;
5
6 // Order Statistics Tree
7 #include <ext/pb_ds/assoc_container.hpp>
8 #include <ext/pb_ds/tree_policy.hpp>
9 using namespace __gnu_pbds;
10 template<class TIn, class TOut>
11 using order_tree = tree<
12     TIn, TOut, less<TIn>, // key, value types. TOut can be null_type
13     rb_tree_tag, tree_order_statistics_node_update>;
14 // find_by_order(int r) (0-based)
15 // order_of_key(TIn v)
16 // use key pair<Tin,int> {value, counter} for multiset/multimap

```

## 3 Basic Graph algorithms

### 3.1 Edge Classification

Complexity:  $O(V + E)$

```

1 struct Edge_Classification {
2     vector<vi> &edges; int V; vi color, parent;
3     Edge_Classification(vector<vi> &edges) :
4         edges(edges), V(edges.size()),
5         color(V,-1), parent(V, -1) {}
6
7     void visit(int u) {
8         color[u] = 1; // in progress
9         for (int v : edges[u]) {
10             if (color[v] == -1) { // u -> v is a tree edge
11                 parent[v] = u;
12                 visit(v);
13             } else if (color[v] == 1) {
14                 if (v == parent[u]) {} // u -> v is a bidirectional edge
15                 else {} // u -> v is a back edge (thus contained in a cycle)

```



```

16         } else if (color[v] == 2) {} // u -> v is a forward/cross edge
17     }
18     color[u] = 2;          // done
19 }
20 void run(){
21     for (int u = 0; u < V; ++u) if(color[u] < 0) visit(u);
22 }
23 };

```

### 3.2 Topological sort

Complexity:  $O(V + E)$

```

1 struct Toposort {
2     vector<vi> &edges;
3     int V, s_ix; // sorted-index
4     vi sorted, visited;
5
6     Toposort(vector<vi> &edges) :
7         edges(edges), V(edges.size()), s_ix(V),
8         sorted(V,-1), visited(V,false) {}
9
10    void visit(int u) {
11        visited[u] = true;
12        for (int v : edges[u])
13            if (!visited[v]) visit(v);
14        sorted[--s_ix] = u;
15    }
16    void topo_sort() {
17        for (int i = 0; i < V; ++i) if (!visited[i]) visit(i);
18    }
19 };

```

### 3.3 Tarjan: SCCs

Complexity:  $O(V + E)$

```

1 struct Tarjan {
2     vvi &edges;
3     int V, counter = 0, C = 0;
4     vi n, l;
5     vb vs;
6     stack<int> st;
7
8     Tarjan(vvi &e) : edges(e), V(e.size()),
9         n(V, -1), l(V, -1), vs(V, false) {}
10
11    void visit(int u, vi &com) {
12        l[u] = n[u] = counter++;
13        st.push(u); vs[u] = true;
14        for (auto &v : edges[u]) {
15            if (n[v] == -1) visit(v, com);
16            if (vs[v]) l[u] = min(l[u], l[v]);
17        }
18        if (l[u] == n[u]) {
19            while (true) {
20                int v = st.top(); st.pop(); vs[v] = false;

```

```

21        com[v] = C;          //<== ACT HERE
22        if (u == v) break;
23    }
24    C++;
25 }
26 }
27
28 int find_sccs(vi &com) { // component indices will be stored in 'com'
29     com.assign(V, -1);
30     C = 0;
31     for (int u = 0; u < V; ++u)
32         if (n[u] == -1) visit(u, com);
33     return C;
34 }
35
36 // scc is a map of the original vertices of the graph
37 // to the vertices of the SCC graph, scc_graph is its
38 // adjacency list.
39 // Scc indices and edges are stored in 'scc' and 'scc_graph'.
40 void scc_collapse(vi &scc, vvi &scc_graph) {
41     find_sccs(scc);
42     scc_graph.assign(C, vi());
43     set<ii> rec; // recorded edges
44     for (int u = 0; u < V; ++u) {
45         assert(scc[u] != -1);
46         for (int v : edges[u]) {
47             if (scc[v] == scc[u] ||
48                 rec.find({scc[u], scc[v]}) != rec.end()) continue;
49             scc_graph[scc[u]].push_back(scc[v]);
50             rec.insert({scc[u], scc[v]});
51         }
52     }
53 }
54 };

```

### 3.4 Biconnected components

Complexity:  $O(V + E)$

```

1 struct BCC{ // find AVs and bridges in an undirected graph
2     vvi &edges;
3     int V, counter = 0, root, rcs; // root and # children of root
4     vi n,l; // nodes,low
5     stack<int> s;
6     BCC(vvi &e) : edges(e), V(e.size()), n(V,-1), l(V,-1) {}
7     void visit(int u, int p) { // also pass the parent
8         l[u] = n[u] = counter++; s.push(u);
9         for(auto &v : edges[u]){
10             if (n[v] == -1) {
11                 if (u == root) rcs++; visit(v,u);
12                 if (l[v] >= n[u]) {} // u is an articulation point
13                 if (l[v] > n[u]) { // u<->v is a bridge
14                     while(true){
15                         int w = s.top(); s.pop(); // <= ACT HERE
16                         if(w==v) break;
17                     }
18                 }
19                 l[u] = min(l[u], l[v]);

```

```

20     } else if (v != p) l[u] = min(l[u], n[v]);
21 }
22 }
23 void run() {
24     for (int u = 0; u < V; ++u) if (n[u] == -1) {
25         root = u; rcs = 0; visit(u, -1);
26         if (rcs > 1) {} // u is articulation point
27     }
28 }
29 };

```

### 3.5 Kruskal's algorithm

**Complexity:**  $O(E \log V)$  **Dependencies:** Union Find

```

1 #include "../datastructures/unionfind.cpp"
2 // Edges are given as (weight, (u, v)) triples.
3 struct E {int u, v, weight;};
4 bool operator<(const E &l, const E &r){return l.weight < r.weight;}
5 int kruskal(vector<E> &edges, int V) {
6     sort(edges.begin(), edges.end());
7     int cost = 0, count = 0;
8     UnionFind uf(V);
9     for (auto &e : edges) {
10         if (!uf.same(e.u, e.v)) {
11             // (w, (u, v)) is part of the MST
12             cost += e.weight;
13             uf.union_set(e.u, e.v);
14             if ((++count) == V - 1) break;
15         }
16     }
17     return cost;
18 }

```

### 3.6 Prim's algorithm

**Complexity:**  $O(E \log V)$

```

1 struct AdjEdge { int v; ll weight; }; // adjacency list edge
2 struct Edge { int u, v; }; // edge u->v for output
3 struct PQ { ll weight; Edge e; }; // PQ element
4 bool operator>(const PQ &l, const PQ &r) { return l.weight > r.weight; }
5 ll prim(vector<vector<AdjEdge>> &adj, vector<Edge> &tree) {
6     ll tc = 0; vb intree(adj.size(), false);
7     priority_queue<PQ, vector<PQ>, greater<PQ>> pq;
8     intree[0] = true;
9     for (auto &e : adj[0]) pq.push({e.weight, {0, e.v}});
10    while (!pq.empty()) {
11        auto &top = pq.top();
12        ll c = top.weight; auto e = top.e; pq.pop();
13        if (intree[e.v]) continue;
14        intree[e.v] = true; tc += c; tree.push_back(e);
15        for (auto &e2 : adj[e.v])
16            if (!intree[e2.v]) pq.push({e2.weight, {e.v, e2.v}});
17    }
18    return tc;
19 }

```

### 3.7 Dijkstra's algorithm

**Complexity:**  $O((V + E) \log V)$

```

1 struct Edge{ int v; ll weight; }; // input edges
2 struct PQ{ ll d; int v; }; // distance and target
3 bool operator>(const PQ &l, const PQ &r){ return l.d > r.d; }
4 ll dijkstra(vector<vector<Edge>> &edges, int s, int t) {
5     vector<ll> dist(edges.size(), LLINF);
6     priority_queue<PQ, vector<PQ>, greater<PQ>> pq;
7     dist[s] = 0; pq.push({0, s});
8     while (!pq.empty()) {
9         auto d = pq.top().d; auto u = pq.top().v; pq.pop();
10        if (u==t) break; // target reached
11        if (d == dist[u])
12            for(auto &e : edges[u]) if (dist[e.v] > d + e.weight)
13                pq.push({dist[e.v] = d + e.weight, e.v});
14    }
15    return dist[t];
16 }

```

### 3.8 Bellman-Ford

An improved (but slower) version of Bellmann-Ford that can indicate for each vertex separately whether it is reachable, and if so, whether there is a lowerbound on the length of the shortest path. **Complexity:**  $O(VE)$

```

1 void bellmann_ford_extended(vvii &e, int source, vi &dist, vb &cyc) {
2     dist.assign(e.size(), INF);
3     cyc.assign(e.size(), false); // true when u is in a <0 cycle
4     dist[source] = 0;
5     for (int iter = 0; iter < e.size() - 1; ++iter){
6         bool relax = false;
7         for (int u = 0; u < e.size(); ++u)
8             if (dist[u] == INF) continue;
9             else for (auto &e : e[u])
10                if (dist[u]+e.second < dist[e.first])
11                    dist[e.first] = dist[u]+e.second, relax = true;
12        if(!relax) break;
13    }
14    bool ch = true;
15    while (ch) { // keep going untill no more changes
16        ch = false; // set dist to -INF when in cycle
17        for (int u = 0; u < e.size(); ++u)
18            if (dist[u] == INF) continue;
19            else for (auto &e : e[u])
20                if (dist[e.first] > dist[u] + e.second
21                    && !cyc[e.first]) {
22                    dist[e.first] = -INF;
23                    ch = true; //return true for cycle detection only
24                    cyc[e.first] = true;
25                }
26    }
27 }

```

### 3.9 Floyd-Warshall algorithm

Transitive closure:  $R[a,c] = R[a,c] \mid (R[a,b] \ \& \ R[b,c])$ , transitive reduction:  $R[a,c] = R[a,c] \ \& \ \neg(R[a,b] \ \& \ R[b,c])$ . **Complexity:**  $O(V^3)$

```

1 // adj should be a V*V array s.t. adj[i][j] contains the weight of
2 // the edge from i to j, INF if it does not exist.
3 // set adj[i][i] to 0; and always do adj[i][j] = min(adj[i][j], w)
4 int adj[100][100];
5 void floyd_warshall(int V) {
6     for (int b = 0; b < V; ++b)
7         for (int a = 0; a < V; ++a)
8             for (int c = 0; c < V; ++c)
9                 if (adj[a][b] != INF && adj[b][c] != INF)
10                    adj[a][c] = min(adj[a][c], adj[a][b] + adj[b][c]);
11 }
12 void setnegcycle(int V){           // set all -Infinity distances
13     REP(a,V) REP(b,V) REP(c,V)     //tested on Kattis
14         if (adj[a][c] != INF && adj[c][b] != INF && adj[c][c]<0){
15             adj[a][b] = - INF;
16             break;
17         }
18 }
```

### 3.10 Johnson's reweighting algorithm

Apply Bellman-Ford to the graph with  $d[u] = 0$  (as if an extra vertex with zero weight edges were added), then reweight edges to  $w_{uv} + h_u - h_v$ , then use Dijkstra. **Complexity:**  $O(VE \log V)$

### 3.11 Hierholzer's algorithm

Verify existence of the circuit/trail in advance (see Theorems in Graph Theory for more information). When looking for a trail, be sure to specify the starting vertex. **Complexity:**  $O(V + E)$

```

1 struct edge {
2     int v;
3     list<edge>::iterator rev;
4     edge(int _v) : v(_v) {};
5 };
6
7 void add_edge(vector< list<edge> > &adj, int u, int v) {
8     adj[u].push_front(edge(v));
9     adj[v].push_front(edge(u));
10    adj[u].begin()->rev = adj[v].begin();
11    adj[v].begin()->rev = adj[u].begin();
12 }
13
14 void remove_edge(vector< list<edge> > &adj, int s, list<edge>::iterator e) {
15     adj[e->v].erase(e->rev);
16     adj[s].erase(e);
17 }
18
19 eulerian_circuit(vector< list<edge> > &adj, vi &c, int start = 0) {
20     stack<int> st;
21     st.push(start);
```

```

22
23     while(!st.empty()) {
24         int u = st.top().first;
25         if (adj[u].empty()) {
26             c.push_back(u);
27             st.pop();
28         } else {
29             st.push(adj[u].front().v);
30             remove_edge(adj, u, adj[u].begin());
31         }
32     }
33 }
```

### 3.12 Bron-Kerbosch

Count the number of maximal cliques in a graph with up to a few hundred nodes. **Complexity:**  $O(3^{n/3})$

```

1 constexpr size_t M = 128; using S = bitset<M>;
2 // count maximal cliques. Call with R=0, X=0, P[u]=1 forall u
3 int BronKerbosch(const vector<S> &edges, S &R, S &&P, S &&X){
4     if(P.count() == 0 && X.count() == 0) return 1;
5     auto PX = P | X; int p=-1; // the last true bit is the pivot
6     for(int i = M-1; i>=0; i--) if(PX[i]){ p = i; break; }
7     auto mask = P & (~edges[p]); int count = 0;
8     for (size_t u = 0; u < edges.size(); ++u) {
9         if(!mask[u]) continue;
10        R[u]=true;
11        count += BronKerbosch(edges,R,P & edges[u],X & edges[u]);
12        if(count > 1000) return count;
13        R[u]=false; X[u]=true; P[u]=false;
14    }
15    return count;
16 }
```

### 3.13 Theorems in Graph Theory

**Dilworth's theorem** : The minimum number of disjoint chains into which  $S$  can be decomposed equals the length of a longest antichain of  $S$ .

Compute by defining a bipartite graph with a source  $u_x$  and sink  $v_x$  for each vertex  $x$ , and adding an edge  $(u_x, v_y)$  if  $x \leq y, x \neq y$ . Let  $m$  denote the size of the maximum matching, then the number of disjoint chains is  $|S| - m$  (the collection of unmatched endpoints).

**Mirsky's theorem** : The minimum number of disjoint antichains into which  $S$  can be decomposed equals the length of a longest chain of  $S$ .

Compute by defining  $L_v$  to be the length of the longest chain ending at  $v$ . Sort  $S$  topologically and use bottom-up DP to compute  $L_u$  for all  $u \in S$ .

**Kirchhoff's theorem** : Define a  $V \times V$  matrix  $M$  as:  $M_{ij} = \deg(i)$  if  $i = j$ ,  $M_{ij} = -1$  if  $\{i, j\} \in E$ ,  $M_{ij} = 0$  otherwise. Then the number of distinct spanning trees equals any minor of  $M$ .

**Acyclicity** : A directed graph is acyclic if and only if a depth-first search yields no back edges.

**Euler Circuits and Trails** : In an *undirected graph*, an *Eulerian Circuit* exists if and only if all vertices have even degree, and all vertices of nonzero degree belong to a single connected component. In an *undirected graph*, an *Eulerian Trail* exists if and only if at most two vertices have odd degree, and all of its vertices of nonzero degree belong to a single connected component. In a *directed graph*, an *Eulerian Circuit* exists if and only if every vertex has equal indegree and outdegree, and all vertices of nonzero degree belong to a single strongly connected component. In a *directed graph*, an *Eulerian Trail* exists if and only if at most one vertex has  $\text{outdegree} - \text{indegree} = 1$ , at most one vertex has  $\text{indegree} - \text{outdegree} = 1$ , every other vertex has equal indegree and outdegree, and all vertices of nonzero degree belong to a single strongly connected component *in the underlying undirected graph*.

### 3.14 Centroid Decomposition

In case it is necessary to work with the subtrees directly, consider timestamping each node during the decomposition **Complexity:**  $O(n \log n)$

```

1 struct CentroidDecomposition {
2     vvi &e;          // The original tree
3     vb tocheck;      // Used during decomposition
4     vi size, p;
5     int root;        // The decomposition
6     vvi cd;
7     CentroidDecomposition(vvi &tree) : e(tree) {
8         int V = e.size();          // create initializer list?
9         tocheck.assign(V, true);
10        cd.assign(V, vi());
11        p.assign(V, -1);
12        size.assign(V, 0);
13
14        dfs(0);
15        root = decompose(0, V);
16    }
17
18    void dfs(int u) {
19        for (int v : e[u]) {
20            if (v == p[u]) continue;
21            p[v] = u;
22            dfs(v);
23            size[u] += 1 + size[v];
24        }
25    }
26
27    int decompose(int _u, int V) {
28        // Find centroid
29        int u = _u;
30        while (true) {
31            int nu = -1;
32            for (int v : e[u]) {
33                if (!tocheck[v] || v == p[u])
34                    continue;
35                if (1 + size[v] > V / 2) nu = v;

```

```

36            }
37            if (V - 1 - size[u] > V / 2 && p[u] != -1
38                && tocheck[p[u]]) nu = p[u];
39            if (nu != -1) u = nu; else break;
40        }
41        // Fix the sizes of the parents of the centroid
42        for (int v = p[u]; v != -1 && tocheck[v]; v = p[v])
43            size[v] -= 1 + size[u];
44        // Find centroid children
45        tocheck[u] = false;
46        for (int v : e[u]) {
47            if (!tocheck[v]) continue;
48            int V2 = 1 + size[v];
49            if (v == p[u]) V2 = V - 1 - size[u];
50            cd[u].push_back(decompose(v, V2));
51        }
52        return u;
53    }
54 };

```

### 3.15 Heavy-Light decomposition

**Complexity:**  $O(n)$

```

1 struct HLD {
2     int V; vvi &graph; // graph can be graph or childs only
3     vi p, r, d, h; // parents, path-root; heavy child, depth
4     HLD(vvi &graph, int root = 0) : V(graph.size()), graph(graph),
5     p(V, -1), r(V, -1), d(V, 0), h(V, -1) { dfs(root);
6         for (int i=0; i<V; ++i) if (p[i]==-1 || h[p[i]]!=i)
7             for (int j=i; j!=-1; j=h[j]) r[j] = i;
8     }
9     int dfs(int u){
10        ii best={-1,-1}; int s=1, ss; // best, size (of subtree)
11        for(auto &v : graph[u]) if(v!=p[u])
12            d[v]=d[u]+1, p[v]=u, s += ss=dfs(v), best = max(best,{ss,v});
13        h[u] = best.second; return s;
14    }
15    int lca(int u, int v){
16        for(; r[u]!=r[v]; v=p[r[v]]) if(d[r[u]] > d[r[v]]) swap(u,v);
17        return d[u] < d[v] ? u : v;
18    }
19 };

```

### 3.16 HLD with Segtree

**Complexity:**  $O(n \lg^2 n)$

```

1 #include "../datastructures/segmenttree.cpp"
2 template <class T, T(*op)(T, T), T ident>
3 struct HLD { //graph may contain childs only
4     int V; vvi &graph; SegmentTree<T,op,ident> st;
5     vi p, r, d, h, t; // parents, path-root, depth heavy, tree index
6     HLD(vvi &graph, vector<T> &init, int root = 0) :
7         V(graph.size()), graph(graph), st({}),
8         p(V, -1), r(V, -1), d(V, 0), h(V, -1), t(V, -1){
9         dfs(root); int k=0; vector<T> v(V);

```

```

10     for(int i=0; i<V; ++i) if (p[i]==-1 || h[p[i]]!=i)
11         for (int j=i; j!=-1; j=h[j]) r[j] = i, v[k]=init[j], t[j]=k++;
12     st={v};
13 }
14 int dfs(int u){
15     ii best={-1,-1}; int s=1, ss;    // best, size (of subtree)
16     for(auto &v : graph[u]) if(v!=p[u])
17         d[v]=d[u]+1, p[v]=u, s += ss=dfs(v), best = max(best,{ss,v});
18     h[u] = best.second; return s;
19 }
20 int lca(int u, int v){
21     for(; r[u]!=r[v]; v=p[r[v]]) if(d[r[u]] > d[r[v]]) swap(u,v);
22     return d[u] < d[v] ? u : v;
23 }
24 void update(int u, ll v){ st.update(t[u],v); }
25 T query(int u, int v){
26     T a = ident;
27     for(; r[u]!=r[v]; v=p[r[v]]){
28         if(d[r[u]] > d[r[v]]) swap(u,v);
29         a = op(a,st.query(t[r[v]], t[v]));
30     }
31     if(d[u] > d[v]) swap(u,v);
32     return op(a,st.query(t[u],t[v])); // t[u]+1 if data is on edges
33 }
34 };

```

## 4 Flow and Matching

### 4.1 Flow Graph

Structure used by the following flow algorithms.

```

1 using F = ll; using W = ll; // types for flow and weight/cost
2 struct S{
3     const int v;           // neighbour
4     const int r;           // index of the reverse edge
5     F f;                   // current flow
6     const F cap;           // capacity
7     const W cost;          // unit cost
8     S(int v, int ri, F c, W cost = 0) :
9         v(v), r(ri), f(0), cap(c), cost(cost) {}
10 };
11 struct FlowGraph : vector<vector<S>> {
12     FlowGraph(size_t n) : vector<vector<S>>(n) {}
13     void add_edge(int u, int v, F c, W cost = 0){ auto &t = *this;
14         t[u].emplace_back(v, t[v].size(), c, cost);
15         t[v].emplace_back(u, t[u].size()-1, 0, -cost);
16     }
17 };

```

### 4.2 Dinic

**Complexity:**  $O(V^2E)$  **Dependencies:** Flow Graph

```

1 #include "flowgraph.cpp"
2 struct Dinic{
3     FlowGraph &edges; int V,s,t;
4     vi l; vector<vector<S>::iterator> its; // levels and iterators

```

```

5     Dinic(FlowGraph &edges, int s, int t) :
6         edges(edges), V(edges.size()), s(s), t(t), l(V,-1), its(V) {}
7     ll augment(int u, F c) { // we reuse the same iterators
8         if (u == t) return c;
9         for(auto &i = its[u]; i != edges[u].end(); i++){
10             auto &e = *i;
11             if (e.cap > e.f && l[u] < l[e.v]) {
12                 auto d = augment(e.v, min(c, e.cap - e.f));
13                 if (d > 0) { e.f += d; edges[e.v][e.r].f -= d; return d; }
14             }
15             return 0;
16         }
17     ll run() {
18         ll flow = 0, f;
19         while(true) {
20             fill(l.begin(), l.end(), -1); l[s]=0; // recalculate the layers
21             queue<int> q; q.push(s);
22             while(!q.empty()){
23                 auto u = q.front(); q.pop();
24                 for(auto &e : edges[u]) if(e.cap > e.f && l[e.v]<0)
25                     l[e.v] = l[u]+1, q.push(e.v);
26             }
27             if (l[t] < 0) return flow;
28             for (int u = 0; u < V; ++u) its[u] = edges[u].begin();
29             while ((f = augment(s, INF)) > 0) flow += f;
30         }
31     };

```

### 4.3 Minimum Cut Inference

The maximum flow equals the minimum cut. Only use this if the specific edges are needed. Run a flow algorithm in advance. **Complexity:**  $O(V+E)$  **Dependencies:** Flow Network

```

1 void imc_dfs(FlowGraph &fg, int u, vb &cut) {
2     cut[u] = true;
3     for (auto &e : fg[u]) {
4         if (e.cap > e.f && !cut[e.v])
5             imc_dfs(fg, e.v, cut);
6     }
7 }
8 ll infer_minimum_cut(FlowGraph &fg, int s, vb &cut) {
9     cut.assign(fg.size(), false);
10    imc_dfs(fg, s, cut);
11    ll cut_value = 0LL;
12    for (size_t u = 0; u < fg.size(); ++u) {
13        if (!cut[u]) continue;
14        for (auto &e : fg[u]) {
15            if (cut[e.v]) continue;
16            cut_value += e.cap;
17            // The edge e from u to e.v is
18            // in the minimum cut.
19        }
20    }
21    return cut_value;
22 }

```

## 4.4 Min cost flow

### Dependencies: Flow Graph

```

1 #include "flowgraph.cpp"
2 using F = ll; using W = ll; W WINF = LLINF; F FINF = LLINF;
3 struct Q{ int u; F c; W w;}; // target, maxflow and total cost/weight
4 bool operator>(const Q &l, const Q &r){return l.w > r.w;}
5 struct Edmonds_Karp_Dijkstra{
6     FlowGraph &g; int V,s,t; vector<W> pot;
7     Edmonds_Karp_Dijkstra(FlowGraph &g, int s, int t) :
8         g(g), V(g.size()), s(s), t(t), pot(V) {}
9     pair<F,W> run() { // return pair<f, cost>
10         F maxflow = 0; W cost = 0; // Bellmann-Ford for potentials
11         fill(pot.begin(),pot.end(),WINF); pot[s]=0;
12         for (int i = 0; i < V - 1; ++i) {
13             bool relax = false;
14             for (int u = 0; u < V; ++u) if(pot[u] != WINF) for(auto &e : g[u])
15                 if(e.cap>e.f)
16                     if(pot[u] + e.cost < pot[e.v])
17                         pot[e.v] = pot[u] + e.cost, relax=true;
18             if(!relax) break;
19         }
20         for (int u = 0; u < V; ++u) if(pot[u] == WINF) pot[u] = 0;
21         while(true){
22             priority_queue<Q,vector<Q>,greater<Q>> q;
23             vector<vector<S>::iterator> p(V,g.front().end());
24             vector<W> dist(V, WINF); F f, tf = -1;
25             q.push({s, FINF, 0}); dist[s]=0;
26             while(!q.empty()){
27                 int u = q.top().u; W w = q.top().w;
28                 f = q.top().c; q.pop();
29                 if(w!=dist[u]) continue; if(u==t && tf < 0) tf = f;
30                 for(auto it = g[u].begin(); it!=g[u].end(); it++){
31                     auto &e = *it;
32                     W d = w + e.cost + pot[u] - pot[e.v];
33                     if(e.cap>e.f && d < dist[e.v]){
34                         q.push({e.v, min(f, e.cap-e.f),dist[e.v] = d});
35                         p[e.v]=it;
36                     }
37                 }
38                 auto it = p[t];
39                 if(it == g.front().end()) return {maxflow,cost};
40                 maxflow += f = tf;
41                 while(it != g.front().end()){
42                     auto &r = g[it->v][it->r];
43                     cost += f * it->cost; it->f+=f;
44                     r.f -= f; it = p[r.v];
45                 }
46                 for (int u = 0; u < V; ++u) if(dist[u]!=WINF) pot[u] += dist[u];
47             }
48         }
49     };

```

## 4.5 Min edge capacities

Make a supersource  $S$  and supersink  $T$ . When there are a lowerbound  $l(u,v)$  and upperbound  $c(u,v)$ , add edge with capacity  $c - l$ . Furthermore, add  $(t,s)$  with capacity

$\infty$ .

$$M(u) = \sum_v l(v,u) - \sum_v l(u,v)$$

If  $M(u) > 0$ , add  $(S,u)$  with capacity  $M(u)$ . Otherwise add  $(u,T)$  with capacity  $-M(u)$ . Run Dinic to find a max flow. This is a feasible flow in the original graph if all edges from  $S$  are saturated. Run Dinic again in the residual graph of the original problem to find the maximal feasible flow.

## 4.6 Min vertex capacities

$x(u)$  is the amount of flow that is extracted at  $u$ , or inserted when  $x(u) < 0$ . If  $\sum_u s(u) > 0$ , add edge  $(t,\tilde{t})$  with capacity  $\infty$ , and set  $x(\tilde{t}) = -\sum_u x(u)$ . Otherwise add  $(\tilde{s},s)$  and set  $x(\tilde{s}) = -\sum_u x(u)$ .  $\tilde{s}$  or  $\tilde{t}$  is the new source/sink. Now, add  $S$  and  $T$ ,  $(t,s)$  with capacity  $\infty$ . If  $x(u) > 0$ , add  $(S,u)$  with capacity  $x(u)$ . Otherwise add  $(u,T)$  with capacity  $x(u)$ . Use Dinic to find a max flow. If all edges from  $S$  are saturated, this is a feasible flow. Run Dinic again in the residual graph to find the maximal feasible flow.

# 5 Combinatorics & Probability

## 5.1 Stable Marriage Problem

If  $m = w$ , the algorithm finds a complete, optimal matching. `mpref[i][j]` gives the id of the  $j$ 'th preference of the  $i$ 'th man. `wpref[i][j]` gives the preference the  $j$ 'th woman assigns to the  $i$ 'th man. Both `mpref` and `wpref` should be zero-based permutations. **Complexity:**  $O(mw)$

```

1 void stable_marriage(vvi &mpref, vvi &wpref, vi &mmatch) {
2     size_t M = mpreff.size(), W = wpref.size();
3     vi wmatch(W, -1);
4     mmatch.assign(M, -1);
5     vector<size_t> mnext(M, 0);
6     stack<size_t> st;
7     for (size_t m = 0; m < M; ++m) st.push(m);
8
9     while (!st.empty()) {
10         size_t m = st.top(); st.pop();
11         if (mmatch[m] != -1) continue;
12         if (mnext[m] >= W) continue;
13
14         size_t w = mpreff[m][mnext[m]++];
15         if (wmatch[w] == -1) {
16             mmatch[m] = w;
17             wmatch[w] = m;
18         } else {
19             size_t mp = size_t(wmatch[w]);
20             if (wpref[w][m] < wpref[w][mp]) {
21                 mmatch[m] = w;
22                 wmatch[w] = m;
23                 mmatch[mp] = -1;
24                 st.push(mp);
25             } else st.push(m);
26         }
27     }
28 }

```



## 5.2 KP procedure

Solves a two variable single constraint integer linear programming problem. It can be extended to an arbitrary number of constraints by inductively decomposing the constrained region into its binding constraints (hence the  $L$  and  $U$ ), and solving for each region. **Complexity:**  $O(d^2 \log(d) \log(\log(d)))$

```

1 ll solve_single(ll c, ll a, ll b, ll L, ll U) {
2     if (c <= 0) return max(0LL, L);
3     else return min(U, b / a);
4 }
5 ll cdiv(ll a, ll b) { return ceil(a / ll(b)); }
6
7 pair<ll, ll> KP(ll c1, ll c2, ll a1, ll a2, ll b, ll L, ll U) {
8     // Trivial solutions
9     if (b < 0) return {-LLINF, -LLINF};
10    if (c1 <= 0) return {L, solve_single(c2, a2, b - a1 * L, 0, LLINF)};
11    if (c2 <= 0) return {solve_single(c1, a1, b, L, U), 0};
12    if (a1 == 0) return {U, solve_single(c2, a2, b, 0, LLINF)};
13    if (a2 == 0) return {0, LLINF};
14    if (L == U) return {L, solve_single(c2, a2, b - a1 * L, 0, LLINF)};
15    if (b == 0) return {0, 0};
16    // Bound U if possible and recursively solve
17    if (U != LLINF) U = min(U, b / a1);
18    if (L != 0 || U != LLINF) {
19        pair<ll, ll>
20        kp = KP(c1, c2, a1, a2, b - cdiv(b - a1 * U, a2) * a2 - a1 * L, 0, LLINF),
21        s1 = {U, (b - a1 * U) / a2},
22        s2 = {L + kp.first, cdiv(b - a1 * U, a2) + kp.second};
23        return (c1 * s1.first + c2 * s1.second > c1 * s2.first + c2 * s2.second ? s1 : s2);
24    } else if (a1 < a2) {
25        pair<ll, ll> s = KP(c2, c1, a2, a1, b, 0, LLINF);
26        return pair<ll, ll>(s.second, s.first);
27    } else {
28        ll k = a1 / a2, p = a1 - k * a2;
29        pair<ll, ll> kp = KP(c1 - c2 * k, c2, p, a2, b - k * (b / a1) * a2, 0, b / a1);
30        return {kp.first, kp.second - k * kp.first + k * (b / a1)};
31    }
32 }

```

## 5.3 2-SAT

**Complexity:**  $O(|\text{variables}| + |\text{implications}|)$  **Dependencies:** Tarjan's

```

1 #include "../graphs/tarjan.cpp"
2 struct TwoSAT {
3     int n;
4     vvi imp; // implication graph
5     Tarjan tj;
6
7     TwoSAT(int _n) : n(_n), imp(2 * _n, vi()), tj(imp) {}
8
9     // Only copy the needed functions:
10    void add_implies(int c1, bool v1, int c2, bool v2) {
11        int u = 2 * c1 + (v1 ? 1 : 0),
12        v = 2 * c2 + (v2 ? 1 : 0);
13        imp[u].push_back(v); // u => v
14        imp[v^1].push_back(u^1); // -v => -u

```

```

15    }
16    void add_equivalence(int c1, bool v1, int c2, bool v2) {
17        add_implies(c1, v1, c2, v2);
18        add_implies(c2, v2, c1, v1);
19    }
20    void add_or(int c1, bool v1, int c2, bool v2) {
21        add_implies(c1, !v1, c2, v2);
22    }
23    void add_and(int c1, bool v1, int c2, bool v2) {
24        add_true(c1, v1); add_true(c2, v2);
25    }
26    void add_xor(int c1, bool v1, int c2, bool v2) {
27        add_or(c1, v1, c2, v2);
28        add_or(c1, !v1, c2, !v2);
29    }
30    void add_true(int c1, bool v1) {
31        add_implies(c1, !v1, c1, v1);
32    }
33
34    // on true: a contains an assignment.
35    // on false: no assignment exists.
36    bool solve(vb &a) {
37        vi com;
38        tj.find_sccs(com);
39        for (int i = 0; i < n; ++i)
40            if (com[2 * i] == com[2 * i + 1])
41                return false;
42
43        vvi bycom(com.size());
44        for (int i = 0; i < 2 * n; ++i)
45            bycom[com[i]].push_back(i);
46
47        a.assign(n, false);
48        vb vis(n, false);
49        for (auto &&component : bycom) {
50            for (int u : component) {
51                if (vis[u / 2]) continue;
52                vis[u / 2] = true;
53                a[u / 2] = (u % 2 == 1);
54            }
55        }
56        return true;
57    }
58 }

```

## 6 Geometry

### 6.1 Essentials

```

1 using C = ld; // could be long long or long double
2 constexpr C EPS = 1e-10; // change to 0 for C=ll
3 struct P { // may also be used as a 2D vector
4     C x, y;
5     P(C x = 0, C y = 0) : x(x), y(y) {}
6     P operator+ (const P &p) const { return {x + p.x, y + p.y}; }
7     P operator- (const P &p) const { return {x - p.x, y - p.y}; }
8     P operator* (C c) const { return {x * c, y * c}; }

```



```

9   P operator/ (C c) const { return {x / c, y / c}; }
10  bool operator==(const P &r) const { return y == r.y && x == r.x; }
11  C lensq() const { return x*x + y*y; }
12  C len() const { return sqrt(lensq()); }
13 };
14 C sq(C x){ return x*x; }
15 C dot(P p1, P p2){ return p1.x*p2.x + p1.y*p2.y; }
16 C dist(P p1, P p2) { return (p1-p2).len(); }
17 C det(P p1, P p2) { return p1.x * p2.y - p1.y * p2.x; }
18 C det(P p1, P p2, P o) { return det(p1-o, p2-o); }
19 C det(vector<P> ps) {
20     C sum = 0; P prev = ps.back();
21     for(auto &p : ps) sum+=det(p,prev), prev=p;
22     return sum;
23 }
24 C area(P p1, P p2, P p3) { return abs(det(p1, p2, p3))/C(2); }
25 C area(vector<P> poly) { return abs(det(poly))/C(2); }
26 int sign(C c){ return (c > C(0)) - (c < C(0)); }
27 int ccw(P p1, P p2, P p3) { return sign(det(p1, p2, p3)); }
28 // bool: non-parallel (P is valid), p = a*l1+(1-a)*l2 = b*r1 + (1-b)*r2
29 pair<bool,P> intersect(P l1, P l2, P r1, P r2, ld &a, ld &b, bool &intern){
30     P dl = l2-l1, dr = r2-r1; ld d = det(dl,dr);
31     if(abs(d)<=EPS) return {false,{0,0}}; // parallel
32     C x = det(l1,l2)*(r1.x-r2.x) - det(r1,r2)*(l1.x-l2.x);
33     C y = det(l1,l2)*(r1.y-r2.y) - det(r1,r2)*(l1.y-l2.y);
34     P p = {x/d, y/d}; a = det(r1-l1,dr)/d; b = det(r1-l1,dl)/d;
35     intern = 0<= a && a <= 1 && 0 <= b && b <= 1;
36     return {true,p};
37 }
38 P project(P p1, P p2, P p){ // Project p on the line p1-p2
39     return p1 + (p2-p1)/(p2-p1).len() * dot(p-p1,p2-p1); }
40 P reflection(P p1, P p2, P p){ return project(p1,p2,p)*2-p; }
41 struct L { // also a 3D point
42     C a, b, c; // ax + by + cz = 0
43     L(C a = 0, C b = 0, C c = 0) : a(a), b(b), c(c) {}
44     L(P p1, P p2) : a(p2.y-p1.y), b(p1.x-p2.x), c(p2.x*p1.y - p2.y*p1.x) {}
45     void to_points(P &p1, P &p2){
46         if(abs(a)<=EPS) p1 = {0, -c/b}, p2 = {1, -(c+a)/b};
47         else p1 = {-c/a, 0}, p2 = {-(c+b)/a, 1};
48     }
49 };
50 L cross(L p1, L p2){
51     return {p1.b*p2.c-p1.c*p2.b, p1.c*p2.a-p1.a*p2.c, p1.a*p2.b-p1.b*p2.a};
52 }
53 pair<bool,P> intersect(L l1, L l2) {
54     L p = cross(l1,l2);
55     return {p.c!=0, {p.a/p.c, p.b/p.c}};
56 }
57
58 struct Circle{ P p; C r; };
59 vector<P> intersect(const Circle& cc, const L& l){
60     const double &x = cc.p.x, &y = cc.p.y, &r = cc.r, &a=l.a,&b=l.b,&c=l.c;
61     double n = a*a + b*b, t1 = c + a*x + b*y, D = n*r*r - t1*t1;
62     if(D<0) return {};
63     double xmid = b*b*x - a*(c + b*y), ymid = a*a*y - b*(c + a*x);
64     if(D==0) return {P{xmid/n, ymid/(n)}};
65     double sd = sqrt(D);
66     return {P{(xmid - b*sd)/n,(ymid + a*sd)/n},

```

```

67         P{(xmid + b*sd)/n,(ymid - a*sd)/n}};
68 }
69 vector<P> intersect(const Circle& c1, const Circle& c2){
70     C x = c1.p.x-c2.p.x, y = c1.p.y-c2.p.y;
71     const C &r1 = c1.r, &r2 = c2.r;
72     C n = x*x+y*y, D = -(n - (r1+r2)*(r1+r2))*(n - (r1-r2)*(r1-r2));
73     if(D<0) return {};
74     C xmid = x*(-r1*r1+r2*r2+n), ymid = y*(-r1*r1+r2*r2+n);
75     if(D==0) return {P{c2.p.x + xmid/(2.*n),c2.p.y + ymid/(2.*n)}};
76     double sd = sqrt(D);
77     return {P{c2.p.x + (xmid - y*sd)/(2.*n),c2.p.y + (ymid + x*sd)/(2.*n)},
78         P{c2.p.x + (xmid + y*sd)/(2.*n),c2.p.y + (ymid - x*sd)/(2.*n)}};
79 }

```

## 6.2 Convex Hull

Complexity:  $O(n \log n)$  Dependencies: Geometry Essentials

```

1 struct point { ll x, y; };
2 bool operator==(const point &l, const point &r) {
3     return l.x == r.x && l.y == r.y; }
4
5 ll dsq(const point &p1, const point &p2) {
6     return (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y); }
7 ll det(ll x1, ll y1, ll x2, ll y2) {
8     return x1 * y2 - x2 * y1; }
9 ll det(const point &p1, const point &p2, const point &d) {
10    return det(p1.x - d.x, p1.y - d.y, p2.x - d.x, p2.y - d.y); }
11 bool comp_lexo(const point &l, const point &r) {
12    return l.y != r.y ? l.y < r.y : l.x < r.x; }
13 bool comp_angl(const point &l, const point &r, const point &c) {
14    ll d = det(l, r, c);
15    if (d != 0) return d > 0;
16    else return dsq(c, l) < dsq(c, r);
17 }
18
19 struct ConvexHull {
20     vector<point> &p;
21     vector<int> h; // incides of the hull in p, ccw
22     ConvexHull(vector<point> &p) : p(_p) { compute_hull(); }
23     void compute_hull() {
24         int pivot = 0, n = p.size();
25         vector<int> ps(n + 1, 0);
26         for (int i = 1; i < n; ++i) {
27             ps[i] = i;
28             if (comp_lexo(p[i], p[pivot])) pivot = i;
29         }
30         ps[0] = ps[n] = pivot; ps[pivot] = 0;
31         sort(ps.begin()+1, ps.end()-1, [this, &pivot](int l, int r) {
32             return comp_angl(p[l], p[r], p[pivot]); });
33
34         h.push_back(ps[0]);
35         size_t i = 1; ll d;
36         while (i < ps.size()) {
37             if (p[ps[i]] == p[h.back()]) { i++; continue; }
38             if (h.size() < 2 || ((d = det(p[h.end()[-2]],
39                 p[h.back()], p[ps[i]])) > 0)) { // >= for col.
40                 h.push_back(ps[i]);

```

```

41         i++; continue;
42     }
43     if (p[h.end()[-2]] == p[ps[i]]) { i++; continue; }
44     h.pop_back();
45     if (d == 0) h.push_back(ps[i]);
46 }
47 if (h.size() > 1 && h.back() == pivot) h.pop_back();
48 }
49 };
50
51 // Note: if h.size() is small (<5), consider brute forcing to avoid
52 // the usual nasty computational-geometry-edge-cases.
53 void rotating_calipers(vector<point> &p, vector<int> &h) {
54     int n = h.size(), i = 0, j = 1, a = 1, b = 2;
55     while (i < n) {
56         if (det(p[h[j]].x - p[h[i]].x, p[h[j]].y - p[h[i]].y,
57             p[h[b]].x - p[h[a]].x, p[h[b]].y - p[h[a]].y) >= 0) {
58             a = (a + 1) % n;
59             b = (b + 1) % n;
60         } else {
61             ++i; // NOT %n!!
62             j = (j + 1) % n;
63         }
64         // Make computations on the pairs: h[i%n], h[a] and h[j], h[a]
65     }
66 }

```

### 6.3 Upper envelope

To find the envelope of lines  $a_i + b_i x$ , find the convex hull of points  $(b_i, a_i)$ . Add  $(0, -\infty)$  for upper envelope, and  $(0, +\infty)$  for lower envelope.

### 6.4 Formulae

$$[ABC] = rs = \frac{1}{2}ab\sin\gamma = \frac{abc}{4R} = \sqrt{s(s-a)(s-b)(s-c)} = \frac{1}{2} |(B-A, C-A)^T|$$

$$s = \frac{a+b+c}{2} \quad 2R = \frac{a}{\sin\alpha}$$

$$\text{cosine rule:} \quad c^2 = a^2 + b^2 - 2ab\cos\gamma$$

$$\text{Euler:} \quad 1 + CC = V - E + F$$

$$\text{Pick:} \quad \text{Area} = \text{interior points} + \frac{\text{boundary points}}{2} - 1$$

Given a non-self-intersecting closed polygon on  $n$  vertices, given as  $(x_i, y_i)$ , its centroid  $(C_x, C_y)$  is given as:

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \quad C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) = \text{polygon area}$$

## 7 Mathematics

### 7.1 Primes

$$10^3 + \{-9, -3, 9, 13\}, \quad 10^6 + \{-17, 3, 33\}, \quad 10^9 + \{7, 9, 21, 33, 87\}$$

```

1 #include "numbertheory.cpp"
2 ll SIZE; vector<bool> bs; vector<ll> primes, mf; // mf[i]==i when prime
3
4 void sieve(ll size = 1e6) { // call at start in main!
5     SIZE = size; bs.assign(SIZE+1, 1);
6     bs[0] = bs[1] = 0;
7     for (ll i = 2; i <= SIZE; i++) if (bs[i]) {
8         for (ll j = i * i; j <= SIZE; j += i) bs[j] = 0;
9         primes.push_back(i);
10    }
11 }
12 bool is_prime(ll n) { // for N <= SIZE^2
13     if (n <= SIZE) return bs[n];
14     for(const auto &prime : primes)
15         if (n % prime == 0) return false;
16     return true;
17 }
18 struct Factor{ll p; ll exp;}; using FS = vector<Factor>;
19 FS factor(ll n) { FS fs;
20     for(const auto &p: primes){ ll exp=0;
21         if(n==1 || p*p > n) break;
22         while(n % p == 0) n/=p, exp++;
23         if(exp>0) fs.push_back({p,exp});
24     }
25     if (n != 1) fs.push_back({n,1});
26     return fs;
27 }
28
29 void sieve2(ll size=1e6) { // call at start in main!
30     SIZE = size; mf.assign(SIZE+1, -1);
31     mf[0] = mf[1] = 1;
32     for (ll i = 2; i <= SIZE; i++) if (mf[i] < 0) {
33         mf[i] = i;
34         for (ll j = i * i; j <= SIZE; j += i)
35             if(mf[j] < 0) mf[j] = i;
36         primes.push_back(i);
37     }
38 }
39 bool is_prime2(ll n) { assert(n<=SIZE); return mf[n]==n; }
40 FS factor2(ll n){ FS fs;
41     for(; n>1; n/=mf[n])
42         if(!fs.empty() && fs.back().p== mf[n]) fs.back().exp++;
43         else fs.push_back({mf[n],1});
44     return fs;
45 }
46
47 vector<ll> divisors(const FS &fs){ vector<ll> ds{1};
48     ll s=1; for(auto &f:fs) s*=f.exp+1; ds.reserve(s);
49     for(auto f : fs) for(auto d : ds) for(ll i=0; i<f.exp; ++i)
50         ds.push_back(d*f.p);
51     return ds;

```

```

52 }
53 ll num_div( const FS &fs) { ll d = 1;
54     for(auto &f : fs) d *= f.exp+1; return d; }
55 ll sum_div( const FS &fs) { ll s = 1;
56     for(auto &f : fs) s *= (pow(f.p,f.exp+1)-1)/(f.p-1); return s; }
57 ll phi(ll n, const FS &fs) { ll p = n;
58     for(auto &f : fs) p -= p/f.p; return p; }
59 ll ord(ll n, ll m, const FS &fs){ ll o = phi(m,fs); // n^ord(n,m)=1 mod m
60     for(auto f : factor(o)) while(f.exp-- && powmod(n,o/f.p,m)==1) o/=f.p;
61     return o; }

```

## 7.2 Euler Phi

Complexity:  $O(n \log \log n)$

```

1 void calculate_phi(int n, vector<ll> &phi) {
2     phi.resize(n);
3     iota(phi.begin(), phi.end(), 0); // numeric
4     for (ll i=2; i<=n; ++i) if (phi[i] == i)
5         for (ll j=i; j<=n; j+=i) phi[j] -= phi[j]/i;
6 }

```

## 7.3 Number theoretic algorithms

```

1 ll gcd(ll a, ll b) { while (b) { a %= b; swap(a, b); } return a; }
2 ll lcm(ll a, ll b) { return (a / gcd(a, b)) * b; }
3 ll mod(ll a, ll b) { return ((a % b) + b) % b; }
4
5 // Finds x, y s.t. ax + by = d = gcd(a, b).
6 void extended_euclid(ll a, ll b, ll &x, ll &y, ll &d) {
7     ll xx = y = 0;
8     ll yy = x = 1;
9     while (b) {
10         ll q = a / b;
11         ll t = b; b = a % b; a = t;
12         t = xx; xx = x - q * xx; x = t;
13         t = yy; yy = y - q * yy; y = t;
14     }
15     d = a;
16 }
17
18 // solves ab = 1 (mod n), -1 on failure
19 ll mod_inverse(ll a, ll n) {
20     ll x, y, d;
21     extended_euclid(a, n, x, y, d);
22     return (d > 1 ? -1 : mod(x, n));
23 }
24
25 // (a*b)%m
26 ll mulmod(ll a, ll b, ll m){
27     ll x = 0, y=a%m;
28     while(b>0){
29         if(b&1)
30             x = (x+y)%m;
31         y = (2*y)%m;
32         b/=2;
33     }

```

```

34     return x % m;
35 }
36 ll mulmod2(ll a, ll b, ll m){ return __int128(a)*b%m; }
37
38 ll pow(ll b, ll e) { // b^e in logarithmic time
39     ll p = e<2 ? 1 : pow(b*b,e/2);
40     return e&1 ? p*b : p;
41 }
42
43 // Finds b^e % m in O(lg n) time, ensure that b < m to avoid overflow!
44 ll powmod(ll b, ll e, ll m) {
45     ll p = e<2 ? 1 : powmod((b*b)%m,e/2,m);
46     return e&1 ? p*b%m : p;
47 }
48
49 // Solve ax + by = c, returns false on failure.
50 bool linear_diophantine(ll a, ll b, ll c, ll &x, ll &y) {
51     ll d = gcd(a, b);
52     if (c % d) {
53         return false;
54     } else {
55         x = c / d * mod_inverse(a / d, b / d);
56         y = (c - a * x) / b;
57         return true;
58     }
59 }
60
61 ll binom(ll n, ll k){
62     ll ans = 1;
63     for(ll i = 1; i <= min(k,n-k); ++i) ans = ans*(n+1-i)/i;
64     return ans;
65 }
66
67 // Solves x = a1 mod m1, x = a2 mod m2, x is unique modulo lcm(m1, m2).
68 // Returns {0, -1} on failure, {x, lcm(m1, m2)} otherwise.
69 pair<ll, ll> crt(ll a1, ll m1, ll a2, ll m2) {
70     ll s, t, d;
71     extended_euclid(m1, m2, s, t, d);
72     if (a1 % d != a2 % d) return {0, -1};
73     return {mod(s * a2 * m1 + t * a1 * m2, m1 * m2) / d, m1 / d * m2};
74 }
75
76 // Solves x = ai mod mi. x is unique modulo lcm mi.
77 // Returns {0, -1} on failure, {x, lcm mi} otherwise.
78 pair<ll, ll> crt(vector<ll> &a, vector<ll> &m) {
79     pair<ll, ll> res = {a[0], m[0]};
80     for (ull i = 1; i < a.size(); ++i) {
81         res = crt(res.first, res.second, mod(a[i], m[i]), m[i]);
82         if (res.second == -1) break;
83     }
84     return res;
85 }

```

## 7.4 Lucas' theorem

```

1 #include "<primes.cpp>"
2 ll lucas(ll n, ll k, ll p){ // calculate (n \choose k) % p

```

```

3   ll ans = 1;
4   while(n){
5       ll np = n%p, kp = k%p;
6       if(kp > np) return 0;
7       ans *= binom(np,kp);
8       n /= p; k /= p;
9   }
10  return ans;
11 }

```

## 7.5 Finite Field

```

1 #include "../numbertheory.cpp"
2 template<ll p,ll w> // prime, primitive root
3 struct Field { using T = Field; ll x; Field(ll x=0) : x{x} {}
4     T operator+(T r) const { return {(x+r.x)%p}; }
5     T operator-(T r) const { return {(x-r.x+p)%p}; }
6     T operator*(T r) const { return {(x*r.x)%p}; }
7     T inv(){ return {mod_inverse(x,p)}; }
8     static T root(ll k) { assert( (p-1)%k==0 ); // (p-1)%k == 0?
9         auto r = powmod(w,(p-1)/abs(k),p); // k-th root of unity
10        return k>0 ? T{r} : T{r}.inv();
11    };
12 };
13 using F1 = Field<1004535809,3 >;
14 using F2 = Field<1107296257,10>; // 1<<30 + 1<<25 + 1
15 using F3 = Field<2281701377,3 >; // 1<<31 + 1<<27 + 1

```

## 7.6 Complex Numbers

Faster-than-built-in complex numbers

```

1 constexpr ld pi = 3.1415926535897932384626433;
2 struct Complex { using T = Complex; ld u,v;
3     Complex(ld u=0, ld v=0) : u{u}, v{v} {}
4     T operator+(T r) const { return {u+r.u, v+r.v}; }
5     T operator-(T r) const { return {u-r.u, v-r.v}; }
6     T operator*(T r) const { return {u*r.u - v*r.v, u*r.v + v*r.u}; }
7     T operator/(T r) {
8         auto norm = r.u*r.u+r.v*r.v;
9         return {(u*r.u + v*r.v)/norm, (v*r.u - u*r.v)/norm};
10    };
11    T inv(){ return T{1,0}/ *this; }
12    static T root(ll k){ return {cos(2*pi/k), sin(2*pi/k)}; }
13 };

```

## 7.7 Fast Fourier Transform

Calculates the discrete convolution of two vectors. Note that the method accepts and outputs complex numbers, and the input is changed in place. **Complexity:**  $O(n \log n)$

**Dependencies:** Bitmasking, Complex Numbers

```

1 #include "../helpers/bitmasking.cpp"
2 #include "../complex.cpp"
3 #include "../field.cpp"
4 using T = Complex; // using T=F1,F2,F3

```

```

5 void fft(vector<T> &A, int p, bool inv = false) {
6     int N = 1<<p;
7     for(int i = 0, r = 0; i < N; ++i, r = brinc(r, p))
8         if (i < r) swap(A[i], A[r]);
9     for (int m = 2; m <= N; m <= 1) {
10        T w, w_m = T::root(inv ? -m : m);
11        for (int k = 0; k < N; k += m) {
12            w = T{1};
13            for (int j = 0; j < m/2; ++j) {
14                T t = w * A[k + j + m/2];
15                A[k + j + m/2] = A[k + j] - t;
16                A[k + j] = A[k + j] + t;
17                w = w * w_m;
18            }
19        }
20    }
21    if(inv){ T inverse = T(N).inv(); for(auto &x : A) x = x*inverse; }
22 }
23 // convolution leaves A and B in frequency domain state
24 // C may be equal to A or B for in-place convolution
25 void convolution(vector<T> &A, vector<T> &B, vector<T> &C){
26     int s = A.size() + B.size() - 1;
27     int q = 32 - __builtin_clz(s-1), N=1<<q; // fails if s=1
28     A.resize(N,{0}); B.resize(N,{0}); C.resize(N,{0});
29     fft(A, q, false); fft(B, q, false);
30     for (int i = 0; i < N; ++i) C[i] = A[i] * B[i];
31     fft(C, q, true); C.resize(s);
32 }
33 void convolution(vector<vector<T>> &ps, vector<T> &C){
34     int s=1; for(auto &p : ps) s+=p.size()-1;
35     int q = 32 - __builtin_clz(s-1), N=1<<q; // fails if s=1
36     C.assign(N,{0});
37     for(auto &p : ps){ p.resize(N,{0}); fft(p, q, false);
38         for(int i = 0; i < N; ++i) C[i] = C[i] * p[i];
39     }
40     fft(C, q, true); C.resize(s);
41 }
42 void square_inplace(vector<T> &A) {
43     int s = 2*A.size()-1, q = 32 - __builtin_clz(s-1), N=1<<q;
44     A.resize(N,{0}); fft(A, q, false);
45     for(auto &x : A) x = x*x;
46     fft(A, q, true); A.resize(s);
47 }

```

## 7.8 Matrix equation solver

Solve  $MX = A$  for  $X$ , and write the square matrix  $M$  in reduced row echelon form, where each row starts with a 1, and this 1 is the only nonzero value in its column.

```

1 using T = double;
2 constexpr T EPS = 1e-8;
3 template<int R, int C>
4 using M = array<array<T,C>,R>; // matrix
5 template<int R, int C>
6 T ReducedRowEchelonForm(M<R,C> &m, int rows) { // return the determinant
7     int r = 0; T det = 1; // MODIFIES the input
8     for(int c = 0; c < rows && r < rows; c++) {
9         int p = r;

```

```

10     for(int i=r+1; i<rows; i++) if(abs(m[i][c]) > abs(m[p][c])) p=i;
11     if(abs(m[p][c]) < EPS){ det = 0; continue; }
12     swap(m[p], m[r]); det *= ( (p-r)%2 ? -1 : 1 );
13     T s = 1.0 / m[r][c], t; det *= m[r][c];
14     REP(j,C) m[r][j] *= s; // make leading term in row 1
15     REP(i,rows) if (i!=r){ t = m[i][c]; REP(j,C) m[i][j] -= t*m[r][j]; }
16     ++r;
17 }
18 return det;
19 }
20 bool error, inconst; // error => multiple or inconsistent
21 template<int R,int C> // Mx = a; M:R*R, v:R*C => x:R*C
22 M<R,C> solve(const M<R,R> &m, const M<R,C> &a, int rows){
23     M<R,R+C> q;
24     REP(r,rows){
25         REP(c,rows) q[r][c] = m[r][c];
26         REP(c,C) q[r][R+c] = a[r][c];
27     }
28     ReducedRowEchelonForm<R,R+C>(q,rows);
29     M<R,C> sol; error = false, inconst = false;
30     REP(c,C) for(auto j = rows-1; j >= 0; --j){
31         T t=0; bool allzero=true;
32         for(auto k = j+1; k < rows; ++k)
33             t += q[j][k]*sol[k][c], allzero &= abs(q[j][k]) < EPS;
34         if(abs(q[j][j]) < EPS)
35             error = true, inconst |= allzero && abs(q[j][R+c]) > EPS;
36         else sol[j][c] = (q[j][R+c] - t) / q[j][j];
37     }
38     return sol;
39 }

```

## 7.9 Matrix Exponentiation

Matrix exponentiation in logarithmic time.

```

1 #define ITERATE_MATRIX(w) for (int r = 0; r < (w); ++r) \
2                             for (int c = 0; c < (w); ++c)
3 template <class T, int N>
4 struct M {
5     array<array<T,N>,N> m;
6     M() { ITERATE_MATRIX(N) m[r][c] = 0; }
7     static M id() {
8         M I; for (int i = 0; i < N; ++i) I.m[i][i] = 1; return I;
9     }
10    M operator*(const M &rhs) const {
11        M out;
12        ITERATE_MATRIX(N) for (int i = 0; i < N; ++i)
13            out.m[r][c] += m[r][i] * rhs.m[i][c];
14        return out;
15    }
16    M raise(ll n) const {
17        if(n == 0) return id();
18        if(n == 1) return *this;
19        auto r = (*this**this).raise(n / 2);
20        return (n%2 ? *this*r : r);
21    }
22 };

```

## 7.10 Simplex algorithm

Maximize  $c^T x$  subject to  $Ax \leq b$  and  $x \geq 0$ .  $A[m \times n]$ ,  $b[m]$ ,  $c[n]$ ,  $x[n]$ . Solution in  $x$ .

```

1 using T = long double; using vd = vector<T>; using vvd = vector<vd>;
2 const T EPS = 1e-9;
3 struct LPSolver {
4     int m, n; vi B, N; vvd D;
5     LPSolver(const vvd &A, const vd &b, const vd &c) :
6         m(b.size()), n(c.size()), B(m), N(n+1), D(m+2, vd(n+2)) {
7         REP(i,m) REP(j,n) D[i][j] = A[i][j];
8         REP(i,m) B[i] = n+i, D[i][n] = -1, D[i][n+1] = b[i];
9         REP(j,n) N[j] = j, D[m][j] = -c[j];
10        N[n] = -1; D[m+1][n] = 1;
11    }
12    void Pivot(int r, int s) {
13        REP(i,m+2) if (i != r) REP(j,n+2) if (j != s)
14            D[i][j] -= D[r][j] * D[i][s] / D[r][s];
15        REP(j,n+2) if (j != s) D[r][j] /= D[r][s];
16        REP(i,m+2) if (i != r) D[i][s] /= -D[r][s];
17        D[r][s] = 1.0 / D[r][s];
18        swap(B[r], N[s]);
19    }
20    bool Simplex(int phase) {
21        int x = phase == 1 ? m+1 : m;
22        while (true) {
23            int s = -1;
24            REP(j,n+1){
25                if (phase == 2 && N[j] == -1) continue;
26                if (s == -1 || D[x][j] < D[x][s] ||
27                    (D[x][j] == D[x][s] && N[j] < N[s])) s = j;
28            }
29            if (D[x][s] >= -EPS) return true;
30            int r = -1;
31            REP(i,m){
32                if (D[i][s] <= 0) continue;
33                if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
34                    (D[i][n+1]/D[i][s] == D[r][n+1]/D[r][s] && B[i] < B[r]))
35                    r = i;
36            }
37            if (r == -1) return false;
38            Pivot(r, s);
39        }
40    }
41    T Solve(vd &x) {
42        int r = 0;
43        for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
44        if (D[r][n+1] <= -EPS) {
45            Pivot(r, n);
46            if (!Simplex(1) || D[m+1][n+1] < -EPS) return -INF;
47            REP(i,m) if (B[i] == -1) {
48                int s = -1;
49                REP(j,n+1)
50                    if (s == -1 || D[i][j] < D[i][s] ||
51                        (D[i][j] == D[i][s] && N[j] < N[s])) s = j;
52                Pivot(i, s);
53            }
54        }
55    }

```

```

55     if (!Simplex(2)) return INF;
56     x = vd(n);
57     REP(i,m) if (B[i] < n) x[B[i]] = D[i][n+1];
58     return D[m][n+1];
59 }
60 };

```

## 7.11 Game theory

A game can be reduced to Nim if it is a finite impartial game, then for any state  $x$ ,  $g(x) = \inf(\mathbb{N}_0 - \{g(y) : y \in F(x)\})$ . Nim and its variants include:

**Nim** Let  $X = \bigoplus_{i=1}^n x_i$ , then  $(x_i)_{i=1}^n$  is a winning position iff  $X \neq 0$ . Find a move by picking  $k$  such that  $x_k > x_k \oplus X$ .

**Misère Nim** Regular Nim, except that the last player to move *loses*. Play regular Nim until there is only one pile of size larger than 1, reduce it to 0 or 1 such that there is an odd number of piles.

**Staricase Nim** Stones are moved down a staircase and only removed from the last pile.  $(x_i)_{i=1}^n$  is an  $L$ -position if  $(x_{2i-1})_{i=1}^{n/2}$  is (i.e. only look at odd-numbered piles).

**Moore's Nim<sub>k</sub>** The player may remove from at most  $k$  piles (Nim = Nim<sub>1</sub>). Expand the piles in base 2, do a carry-less addition in base  $k+1$  (i.e. the number of ones in each column should be divisible by  $k+1$ ).

**Dim<sup>+</sup>** The number of removed stones must be a divisor of the pile size. The Sprague-Grundy function is  $k+1$  where  $2^k$  is the largest power of 2 dividing the pile size.

**Aliquot game** Same as above, except the divisor should be proper (hence 1 is also a terminal state, but watch out for size 0 piles). Now the Sprague-Grundy function is just  $k$ .

**Nim (at most half)** Write  $n+1 = 2^m y$  with  $m$  maximal, then the Sprague-Grundy function of  $n$  is  $(y-1)/2$ .

**Lasker's Nim** Players may alternatively split a pile into two new non-empty piles.  $g(4k+1) = 4k+1$ ,  $g(4k+2) = 4k+2$ ,  $g(4k+3) = 4k+4$ ,  $g(4k+4) = 4k+3$  ( $k \geq 0$ ).

**Hackenbush on trees** A tree with stalks  $(x_i)_{i=1}^n$  may be replaced with a single stalk with length  $\bigoplus_{i=1}^n x_i$ .

A useful identity:  $\bigoplus_{x=0}^{a-1} x = \{0, a-1, 1, a\}[a\%4]$ .

## 8 Strings

### 8.1 Knuth Morris Pratt

**Complexity:**  $O(n+m)$

```

1 void compute_prefix_function(string &w, vi &pi) {
2     pi.assign(w.length(), 0);
3     int k = pi[0] = -1;
4

```

```

5     for (int i = 1; i < w.length(); ++i) {
6         while (k >= 0 && w[k+1] != w[i])
7             k = pi[k];
8         if (w[k+1] == w[i]) k++;
9         pi[i] = k;
10    }
11 }
12
13 void knuth_morris_pratt(string &s, string &w) {
14     int q = -1; vi pi;
15     compute_prefix_function(w, pi);
16     for (int i = 0; i < s.length(); ++i) {
17         while (q >= 0 && w[q+1] != s[i]) q = pi[q];
18         if (w[q+1] == s[i]) q++;
19         if (q+1 == w.length()) {
20             // Match at position (i - w.length() + 1)
21             q = pi[q];
22         }
23     }
24 }

```

### 8.2 Z-algorithm

To match pattern  $P$  on string  $S$ : pick  $\Phi$  s.t.  $\Phi \notin P$ , find  $Z$  of  $P\Phi S$ . **Complexity:**  $O(n)$

```

1 void Z_algorithm(string &s, vector<int> &Z) {
2     Z.assign(s.length(), -1);
3     int L = 0, R = 0, n = s.length();
4     for (int i = 1; i < n; ++i) {
5         if (i > R) {
6             L = R = i;
7             while (R < n && s[R-L] == s[R]) R++;
8             Z[i] = R - L; R--;
9         } else if (Z[i-L] >= R - i + 1) {
10            L = i;
11            while (R < n && s[R-L] == s[R]) R++;
12            Z[i] = R - L; R--;
13        } else Z[i] = Z[i-L];
14    }
15    Z[0] = n;
16 }

```

### 8.3 Aho-Corasick

Constructs a Finite State Automaton that can match  $k$  patterns of total length  $m$  on a string of size  $n$ . **Complexity:**  $O(n+m+k)$

```

1 template <int ALPHABET_SIZE, int (*mp)(char)>
2 struct AC_FSM {
3     struct Node {
4         int child[ALPHABET_SIZE], failure = 0, match_par = -1;
5         vi match;
6         Node() { for (int i = 0; i < ALPHABET_SIZE; ++i) child[i] = -1; }
7     };
8     vector<Node> a;
9     vector<string> &words;
10    AC_FSM(vector<string> &words) : words(words) {

```



```

11     a.push_back(Node());
12     construct_automaton();
13 }
14 void construct_automaton() {
15     for (int w = 0, n = 0; w < words.size(); ++w, n = 0) {
16         for (int i = 0; i < words[w].size(); ++i) {
17             if (a[n].child[mp(words[w][i])] == -1) {
18                 a[n].child[mp(words[w][i])] = a.size();
19                 a.push_back(Node());
20             }
21             n = a[n].child[mp(words[w][i])];
22         }
23         a[n].match.push_back(w);
24     }
25
26     queue<int> q;
27     for (int k = 0; k < ALPHABET_SIZE; ++k) {
28         if (a[0].child[k] == -1) a[0].child[k] = 0;
29         else if (a[0].child[k] > 0) {
30             a[a[0].child[k]].failure = 0;
31             q.push(a[0].child[k]);
32         }
33     }
34     while (!q.empty()) {
35         int r = q.front(); q.pop();
36         for (int k = 0, arck; k < ALPHABET_SIZE; ++k) {
37             if ((arck = a[r].child[k]) != -1) {
38                 q.push(arck);
39                 int v = a[r].failure;
40                 while (a[v].child[k] == -1) v = a[v].failure;
41                 a[arck].failure = a[v].child[k];
42                 a[arck].match_par = a[v].child[k];
43                 while (a[arck].match_par != -1 && a[a[arck].match_par].
44                     match.empty())
45                     a[arck].match_par = a[a[arck].match_par].match_par;
46             }
47         }
48     }
49
50     void aho_corasick(string &sentence, vvi &matches){
51         matches.assign(words.size(), vi());
52         int state = 0, ss = 0;
53         for (int i = 0; i < sentence.length(); ++i, ss = state) {
54             while (a[ss].child[mp(sentence[i])] == -1)
55                 ss = a[ss].failure;
56             state = a[ss].child[mp(sentence[i])]
57                 = a[ss].child[mp(sentence[i])];
58             for (ss = state; ss != -1; ss = a[ss].match_par)
59                 for (int w : a[ss].match)
60                     matches[w].push_back(i + 1 - words[w].length());
61         }
62     }
63 };

```

## 8.4 Manacher's Algorithm

Finds the largest palindrome centered at each position. **Complexity:**  $O(|S|)$

```

1 void manacher(string &s, vector<int> &pal) {
2     int n = s.length(), i = 1, l, r;
3     pal.assign(2 * n + 1, 0);
4     while (i < 2 * n + 1) {
5         if ((i&1) && pal[i] == 0) pal[i] = 1;
6         l = i / 2 - pal[i] / 2; r = (i-1) / 2 + pal[i] / 2;
7
8         while (l - 1 >= 0 && r + 1 < n && s[l - 1] == s[r + 1])
9             --l, ++r, pal[i] += 2;
10
11         for (l = i - 1, r = i + 1; l >= 0 && r < 2 * n + 1; --l, ++r) {
12             if (l <= i - pal[i]) break;
13             if (l / 2 - pal[l] / 2 > i / 2 - pal[i] / 2)
14                 pal[r] = pal[l];
15             else if (l >= 0)
16                 pal[r] = min(pal[l], i + pal[i] - r);
17             break;
18         }
19         i = r;
20     }
21 }

```

## 9 Miscellaneous

### 9.1 LIS

Finds the longest strictly increasing subsequence. To find the longest non-decreasing subsequence, insert pairs  $(a_i, i)$ . Note that the elements should be totally ordered. To find the LIS of a sequence of elements from a partially ordered set (e.g. coordinates in the plane), replace `lis[]` with a set of equivalent elements, at a cost of another  $O(\log n)$  factor. **Complexity:**  $O(n \log n)$

```

1 // Length only
2 template<class T>
3 int longest_increasing_subsequence(vector<T> &a) {
4     set<T> st;
5     typename set<T>::iterator it;
6     for (int i = 0; i < a.size(); ++i) {
7         it = st.lower_bound(a[i]);
8         if (it != st.end()) st.erase(it);
9         st.insert(a[i]);
10    }
11    return st.size();
12 }
13
14 // Entire sequence (indices)
15 template<class T>
16 int longest_increasing_subsequence(vector<T> &a, vector<int> &seq) {
17     vector<int> lis(a.size(), 0), pre(a.size(), -1);
18     int L = 0;
19     for (int i = 0; i < a.size(); ++i) {
20         int l = 1, r = L;
21         while (l <= r) {
22             int m = (l + r + 1) / 2;
23             if (a[lis[m - 1]] < a[i])
24                 l = m + 1;

```



```

25         else
26             r = m - 1;
27     }
28
29     pre[i] = (l > 1 ? lis[l - 2] : -1);
30     lis[l - 1] = i;
31     if (l > L) L = l;
32 }
33
34 seq.assign(L, -1);
35 int j = lis[L - 1];
36 for (int i = L - 1; i >= 0; --i) {
37     seq[i] = j;
38     j = pre[j];
39 }
40 return L;
41 }

```

## 9.2 Randomisation

Might be useful for NP-Complete/Backtracking problems

```

1 #include <chrono>
2 using namespace chrono;
3 auto beg = high_resolution_clock::now();
4 while(high_resolution_clock::now() - beg < milliseconds(TIMELIMIT - 250)){

```

## 9.3 All Nearest Smaller Values

Complexity:  $O(n)$

```

1 void all_nearest_smaller_values(vi &a, vi &b) {
2     b.assign(a.size(), -1);
3     for (int i = 1; i < b.size(); ++i) {
4         b[i] = i - 1;
5         while (b[i] >= 0 && a[i] < a[b[i]])
6             b[i] = b[b[i]];
7     }
8 }

```

# 10 Helpers

## 10.1 Golden Section Search

For a discrete search: use binary search on the difference of successive elements, see the section on Binary Search. **Complexity:**  $O(\log 1/\epsilon)$

```

1 #define RES_PHI (2 - ((1.0 + sqrt(5)) / 2.0))
2 #define EPSILON 1e-7
3
4 double gss(double (*f)(double), double leftbound, double rightbound) {
5     double lb = leftbound, rb = rightbound, mlb = lb + RES_PHI * (rb - lb),
6         mrb = rb + RES_PHI * (lb - rb);
7     double lbv = f(lb), rbv = f(rb), mlbv = f(mlb), mrbv = f(mrb);
8
9     while (rb - lb >= EPSILON) { // || abs(rbv - lbv) >= EPSILON) {
10         if (mlbv < mrbv) { // > to maximize

```

```

10         rb = mrb; rbv = mrbv;
11         mrb = mlb; mrbv = mlbv;
12         mlb = lb + RES_PHI * (rb - lb);
13         mlbv = f(mlb);
14     } else {
15         lb = mlb; lbv = mlbv;
16         mlb = mrb; mlbv = mrbv;
17         mrb = rb + RES_PHI * (lb - rb);
18         mrbv = f(mrb);
19     }
20 }
21 return mlb; // any bound should do
22 }

```

## 10.2 Binary Search

Complexity:  $O(\log n), O(\log 1/\epsilon)$

```

1 # define EPSILON 1e -7
2
3 // Finds the first i s.t. arr[i]>=val, assuming that arr[l] <= val <= arr[h]
4 int integer_binary_search(int l, int h, vector<double> &arr, double val) {
5     while (l < h) {
6         int m = l + (h - l) / 2;
7         if (arr[m] >= val) h = m;
8         else l = m + 1;
9     }
10    return l;
11 }
12
13 // Given a monotonically increasing function f, approximately solves f(x)=c,
14 // assuming that f(l) <= c <= f(h)
15 double binary_search(double l, double h, double (*f)(double), double c) {
16     while (true) {
17         double m = (l + h) / 2, v = f(m);
18         if (abs(v - c) < EPSILON) return m;
19         if (v < c) l = m;
20         else h = m;
21     }
22 }
23
24 // Modifying binary search to do an integer ternary search:
25 int integer_ternary_search(int l, int h, vector<double> &arr) {
26     while (l < h) {
27         int m = l + (h - l) / 2;
28         if (arr[m + 1] - arr[m] >= 0) h = m;
29         else l = m + 1;
30     }
31    return l;
32 }

```

## 10.3 Bitmasking

```

1 #ifdef _MSC_VER
2 #define popcount(x) __popcnt(x)
3 #else
4 #define popcount(x) __builtin_popcount(x)

```

```

5 #endif
6 template<typename F>      // All subsets of {0..N-1}
7 void iterate_subset(ll N, F f){for(ll mask=0; mask < 1ll<<N; ++mask) f(mask)
    ;}
8 template<typename F>      // All subsets of size k of {0..N-1}
9 void iterate_k_subset(ll N, ll k, F f){
10     ll mask = (1ll << k) - 1;
11     while (!(mask & 1ll<<N)) { f(mask);
12         ll t = mask | (mask-1);
13         mask = (t+1) | (((~t & ~t) - 1) >> (__builtin_ctzll(mask)+1));
14     }
15 }
16 template<typename F>      // All subsets of set
17 void iterate_mask_subset(ll set, F f){ ll mask = set;
18     do f(mask), mask = (mask-1) & set;
19     while (mask != set);
20 }
21 ll next_power_of_2(ll x) { // used in FFT
22     x = (x - 1) | ((x - 1) >> 1);
23     x |= x >> 2; x |= x >> 4; x |= x >> 8; x |= x >> 16;
24     return x + 1;
25 }
26 ll brinc(ll x, ll k) {
27     ll i = k - 1, s = 1 << i;
28     x ^= s;
29     if ((x & s) != s) {
30         --i; s >>= 1;
31         while (i >= 0 && ((x & s) == s))
32             x = x &~ s, --i, s >>= 1;
33         if (i >= 0) x |= s;
34     }
35     return x;
36 }

```

## 10.4 Fast IO

```

1 int r() {
2     int sign = 1, n = 0;
3     char c;
4     while ((c = getchar_unlocked()) != '\n')
5         switch (c) {
6             case '-': sign = -1; break;
7             case '_': case '\n': return n * sign;
8             default: n *= 10; n += c - '0'; break;
9         }
10 }
11
12 void scan(ll &x){ // doesn't handle negative numbers
13     char c;
14     while((x=getchar_unlocked())<'0');
15     for(x='0'; '0'<=(c=getchar_unlocked()); x=10*x+c-'0');
16 }
17 void print(ll x){
18     char buf[20], *p=buf;
19     if(!x) putchar_unlocked('0');
20     else{
21         while(x) *p++='0'+x%10, x/=10;

```

```

22         do putchar_unlocked(*--p); while(p!=buf);
23     }
24 }

```

## 10.5 Detecting overflow

These are GNU builtins, detect both over- and underflow. Returns a boolean upon failure, otherwise the result is present in `ref`. Follow the template:

```
__builtin_[u|s][add|mul|sub](ll)?_overflow(in, out, &ref)
```

## 11 Strategies

Take a break after 2 hours.

### Techniques

- Bruteforce: meet-in-the-middle, backtracking, memoization
- DP (write full draft, include ALL loop bounds), easy direction
- Precomputation
- Divide and Conquer
- Binary search
- $\lg(n)$  datastructures
- Mathematical insight
- Randomisation
- Look at it backwards
- Common subproblems? Memoization
- Compute modulo primes and use CRT

### WA

- Beware of typos
- Test sample input; make custom testcases
- Read carefully
- Check bounds (use long long or long double)
- EDGE CASES:  $n \in \{-1, 0, 1, 2\}$ . Empty list/graph?
- Off by one error (in indices or loop bounds)
- Not enough precision
- Assertions
- Missing modulo operators
- Cases that need a (completely) different approach

### TLE

- Infinite loop
- Use scanf or fastIO instead of cin
- Wrong algorithm (is it theoretically fast enough)
- Micro optimizations (but probably the approach just isn't right)

### RTE

- Typos
- Off by one error (in array index of loop bound)
- return 0 at end of program