



# Team Code Reference

## Curiously Recurring

Benelux Algorithm Programming Contest (BAPC)  
October 24, 2015

<b>1</b>	<b>Templates</b>	<b>1</b>	4.3	Minimum Cut Inference . .	13
1.1	Vimrc . . . . .	1	4.4	Min cost flow . . . . .	13
1.2	C++ Template . . . . .	1	4.5	Min edge capacities . . . . .	13
1.3	Java Template . . . . .	1	4.6	Min vertex capacities . . . . .	13
<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 . . . . .	14
2.3	Fenwick Tree . . . . .	2	5.3	2-SAT . . . . .	14
2.4	2D Fenwick Tree . . . . .	3	<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 . . . . .	16
2.8	Implicit Cartesian Tree . . .	4	6.4	Formulae . . . . .	16
2.9	AVL Tree . . . . .	5	<b>7</b>	<b>Mathematics</b>	<b>16</b>
2.10	Treap . . . . .	6	7.1	Primes . . . . .	16
2.11	Prefix Trie . . . . .	6	7.2	Euler Phi . . . . .	17
2.12	Suffix Array . . . . .	7	7.3	Number theoretic algorithms	18
2.13	Suffix Tree . . . . .	7	7.4	Lucas' theorem . . . . .	18
2.14	Suffix Automaton . . . . .	8	7.5	Complex Numbers . . . . .	18
2.15	Built-in datastructures . . .	8	7.6	Fast Fourier Transform . . .	19
<b>3</b>	<b>Basic Graph algorithms</b>	<b>8</b>	7.7	Matrix equation solver . . .	19
3.1	Edge Classification . . . . .	8	7.8	Matrix Exponentiation . . .	19
3.2	Topological sort . . . . .	8	7.9	Simplex algorithm . . . . .	20
3.3	Tarjan: SCCs . . . . .	8	7.10	Game theory . . . . .	20
3.4	Biconnected components . .	9	<b>8</b>	<b>Strings</b>	<b>21</b>
3.5	Kruskal's algorithm . . . . .	9	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	Bellmann-Ford . . . . .	10	8.4	Manacher's Algorithm . . .	22
3.9	Floyd-Warshall algorithm .	10	<b>9</b>	<b>Miscellaneous</b>	<b>22</b>
3.10	Hierholzer's algorithm . . .	10	9.1	LIS . . . . .	22
3.11	Bron-Kerbosch . . . . .	11	9.2	Randomisation . . . . .	22
3.12	Theorems in Graph Theory	11	<b>10</b>	<b>Helpers</b>	<b>22</b>
3.13	Centroid Decomposition . .	11	10.1	Golden Section Search . . .	22
3.14	Heavy-Light decomposition	12	10.2	Binary Search . . . . .	23
<b>4</b>	<b>Flow and Matching</b>	<b>12</b>	10.3	Bitmasking . . . . .	23
4.1	Flow Graph . . . . .	12	10.4	Fast IO . . . . .	23
4.2	Dinic . . . . .	12	<b>11</b>	<b>Strategies</b>	<b>24</b>

## 1 Templates

### 1.1 Vimrc

---

```

1 syntax on                      colorscheme default                hi Comment ctermfg=cyan
2 tabstop=4 softtabstop=4 shiftwidth=4 laststatus=2 encoding=utf-8 mouse=nvc
3 clipboard=unnamed backspace=indent,eol,start cinoptions=:0,l1,g0,(0
4 noexpandtab wrap linebreak nu incsearch cindent autoindent hidden

```

---

### 1.2 C++ Template

---

```

1 iostream string sstream vector list set map unordered_map queue stack bitset
2 tuple cstdio numeric iterator algorithm cmath chrono cassert unordered_set
3 using namespace std;          // :s/ /\r/g :s/\w*/#include <\0>/g
4 #define REP(i,n)               for(auto i = decltype(n)(0); i<(n); i++)
5 #define F(v)                   begin(v), end(v)
6 constexpr bool LOG =
7 #ifdef _LOG                     // -D_LOG compiler option
8 true;
9 #define _GLIBCXX_DEBUG         // for bounds checking etc
10 #else
11 false;
12 #endif
13 using ll = long long; using ii = pair<int,int>; using vi = vector<int>;
14 using vb = vector<bool>; using vvi = vector<vi>;
15 constexpr int INF = 1e9+1; // < 1e9 - -1e9
16 constexpr ll LLINF = 1e18+1;
17 void Log() { if(LOG) cerr << "\n"; }
18 template<class T, class... S> void Log(T t, S... s){
19     if(LOG) cerr << t << "\t", Log(s...);
20 }
21 int main(){
22     ios::sync_with_stdio(false); cin.tie(nullptr);
23     return 0;
24 }

```

---

### 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.InputStream;
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 class UnionFind {
2 private:
3     vi par, rank, size; int c;
4 public:
5     UnionFind(int n) : par(n), rank(n,0), size(n,1), c(n) {
6         for (int i = 0; i < n; ++i) par[i] = i;
7     }
8
9     int find(int i) { return (par[i] == i ? i : (par[i] = find(par[i]))); }
10    bool same(int i, int j) { return find(i) == find(j); }
11    int get_size(int i) { return size[find(i)]; }
12    int count() { return c; }
13
14    void union_set(int i, int j) {

```

```

15         if ((i = find(i)) == (j = find(j))) return;
16         c--;
17         if (rank[i] > rank[j]) swap(i, j);
18         par[i] = j; size[j] += size[i];
19         if (rank[i] == rank[j]) rank[j]++;
20     }
21 };

```

### 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 class FenwickTree {
3 private:
4     vector<T> tree;
5     int n;

```

```

6 public:
7     FenwickTree(int n) : n(n) { tree.assign(n + 1, 0); }
8     T query(int l, int r) { return query(r) - query(l - 1); }
9     T query(int r) {
10         T s = 0;
11         for(; r > 0; r -= (r & (-r))) s += tree[r];
12         return s;
13     }
14     void update(int i, T v) {
15         for(; i <= n; i += (i & (-i))) tree[i] += v;
16     }
17 };

```

## 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     struct Node {
4         T val;
5         int l, r;
6         Node(T _val, int _l, int _r) : val(_val), l(_l), r(_r) { };
7     };
8     int n;
9     vector<Node> tree;
10    SegmentTree(int p, vector<T> &init) : n(1 << p) { // Needs 2^p leaves
11        tree.assign(2 * n, Node(ident, 0, n - 1));
12        for (int j = 1; j < n; ++j) {
13            int m = (tree[j].l + tree[j].r) / 2;
14            tree[2*j].l = tree[j].l;
15            tree[2*j].r = m;
16            tree[2*j+1].l = m + 1;
17            tree[2*j+1].r = tree[j].r;
18        }
19        for (int j = 2 * n - 1; j > 0; --j) {
20            if (j >= n) tree[j].val = init[j - n];
21            else tree[j].val = op(tree[2*j].val, tree[2*j+1].val);
22        }
23    }
24    void update(int i, T val) {
25        for (tree[i+n].val = val, i = (i+n)/2; i > 0; i /= 2)
26            tree[i].val = op(tree[2*i].val, tree[2*i+1].val);
27    }
28    T query(int l, int r) {
29        T lhs = T(ident), rhs = T(ident);
30        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
31            if (l&1) lhs = op(lhs, tree[l++].val);
32            if (!(r&1)) rhs = op(tree[r--].val, rhs);
33        }
34        return op(l == r ? op(lhs, tree[l].val) : lhs, rhs);
35    }
36 };

```

## 2.7 Lazy Dynamic Segment Tree

```

1 using T=int; using U=int;
2 T t_id; U u_id;
3 T merge(T a, T b){ return a+b; }
4 void join(U &a, U b){ a=a+b; }
5 struct Node {
6     int l, r, lc, rc; T t; U u;
7     Node(int l, int r, T t=t_id,lc(-1),rc(-1),t(t),u(u_id){}
8 };
9 T apply(const Node &n, int l=-1){ return merge(n.t,(l<0?n.r-n.l+1:1)*n.u); }
10 pair<T,T> split(T t, ll a, ll b){ return {t/(a+b)*a, t/(a+b)*b}; }
11 struct DynamicSegmentTree {

```

```

12 vector<Node> tree;
13 DynamicSegmentTree(int N) { tree.push_back({0,N-1}); }
14 T query(int l, int r, int i = 0) { // 0 <= l <= r < n
15     auto &n = tree[i];
16     if(l > n.r || r < n.l) return t_id; // <- disjunct, vv internal
17     if(l <= n.l && n.r <= r) return apply(n);
18     if(n.lc < 0) return apply(n, min(n.r,r) - max(n.l,l) + 1);
19     n.t = apply(n);
20     join(tree[n.lc].u, n.u); // push the update
21     join(tree[n.rc].u, n.u); n.u = u_id; // and reset the update
22     return merge(query(l, r, n.lc), query(l, r, n.rc));
23 }
24 void update(int l, int r, U u, int i = 0) {
25     auto &n = tree[i];
26     if(l > n.r || r < n.l) return;
27     if(l <= n.l && n.r <= r){ join(n.u,u); return; }
28     if(n.lc < 0 || n.rc < 0) {
29         int m = (n.l + n.r) / 2;
30         n.lc = tree.size(); n.rc = tree.size()+1;
31         auto sp = split(n.t,m-n.l+1,n.r-m);
32         tree.push_back({tree[i].l, m, sp.first});
33         tree.push_back({m+1,tree[i].r, sp.second});
34     } // DON'T use 'n' anymore, because tree may reallocate
35     update(l,r,u,tree[i].lc); update(l,r,u,tree[i].rc);
36     tree[i].t = merge(apply(tree[tree[i].lc]),apply(tree[tree[i].rc]));
37 }
38 };

```

## 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, *r;
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 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 };

```

## 2.10 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, *r;
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 }

```

## 2.11 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 };

```

## 2.12 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 + l < n ? P[k - 1][i + l] : -1)
32         == (j + l < n ? P[k - 1][j + l] : -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 };

```

## 2.13 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             }
43             --remainder;
44             if(n == root && len > 0)
45                 --len, edge = i - remainder + 1;
46             else n = t[n].link > 0 ? t[n].link : root;
47         }
48     }
49 };

```



## 2.14 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 p
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 };

```

---

## 2.15 Built-in datastructures

---

```

1 // Minimum Heap
2 #include <queue>
3 using min_queue = priority_queue<T, vector<T>, greater<T>>;
4
5 // Order Statistics Tree
6 #include <ext/pb_ds/assoc_container.hpp>
7 #include <ext/pb_ds/tree_policy.hpp>
8 using namespace __gnu_pbds;
9 using order_tree =
10 typedef tree<
11     TIn, TOut, less<TIn>, // key, value types. TOut can be null_type
12     rb_tree_tag, tree_order_statistics_node_update>;
13 // find_by_order(int r) (0-based)
14 // order_of_key(TIn v)
15 // 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(0),
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         REP(i,V) 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){ // biconnected component
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         REP(u,V) 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, weight; }; // input edges
2 struct PQ{ int d, v; }; // distance and target
3 bool operator>(const PQ &l, const PQ &r){ return l.d > r.d; }
4 int dijkstra(vector<vector<Edge>> &edges, int s, int t) {
5     vi dist(edges.size(), INF);
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, 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 Bellmann-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:  $R[a,c] = R[a,c] \ \& \ !(R[a,b] \ \& \ R[b, c])$ . **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 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) {};
```

```

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 }

```

### 3.11 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     REP(u, edges.size()){
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 }

```

### 3.12 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*.

### 3.13 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 };

```

### 3.14 Heavy-Light decomposition

Complexity:  $O(n)$

```

1 struct HLD {
2     int V,T; vi &p; vvi &childs;    // Size; dfs-time; input parent/childs
3     vi pr, size, heavy;           // path-root; size of subtrees; heavy child
4     vi t_in, t_out;               // dfs in and out times
5     HLD(vvi &childs, vi &p, int root = 0) :
6         V(p.size()), T(0), p(p), childs(childs), pr(V,-1),
7         size(V,-1), heavy(V,-1), t_in(V,-1), t_out(V,-1) {
8         dfs(root); set_pr(root,0);
9     }
10    int dfs(int u){
11        size[u] = 1; t_in[u] = T++;
12        int m = -1, mi = -1, s;    // max, max index, size of subtree
13        for(auto &v : childs[u]){
14            size[u] += s = dfs(v);
15            if(s > m) m=s, mi = v;
16        }
17        heavy[u] = mi; t_out[u] = T++; return size[u];
18    }

```

```

19 void set_pr(int u, int r){        // node, path root
20     pr[u] = r;
21     for(auto &v : childs[u]) set_pr(v, heavy[u] == v ? r : v);
22 }
23 bool is_parent(int p, int u){    // test whether p is a parent of u
24     return t_in[p] <= t_in[u] && t_out[p] >= t_out[u];
25 }
26 int lca(int u, int v){
27     while(!is_parent(pr[v],u)) v = p[pr[v]];
28     while(!is_parent(pr[u],v)) u = p[pr[u]];
29     return is_parent(u,v) ? u : v;
30 }
31 };

```

## 4 Flow and Matching

### 4.1 Flow Graph

Structure used by the following flow algorithms.

```

1 struct S{
2     int v;           // neighbour
3     const int r;     // index of the reverse edge
4     ll f;            // current flow
5     const ll cap;    // capacity
6     const ll cost;   // unit cost
7     S(int v, int reverse_index, ll capacity, ll cost = 0) :
8         v(v), r(reverse_index), f(0), cap(capacity), cost(cost) {}
9 };
10 struct FlowGraph : vector<vector<S>> {
11     FlowGraph(size_t n) : vector<vector<S>>(n) {}
12     void add_edge(int u, int v, ll capacity, ll cost = 0){
13         operator[](u).emplace_back(v, operator[](v).size(), capacity, cost);
14         operator[](v).emplace_back(u, operator[](u).size()-1, 0, -cost);
15     }
16 };

```

### 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, ll 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(F(1),-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         REP(u,V) 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 struct Q{ int u; ll c,w;}; // target, maxflow and total weight (cost)
3 bool operator>(const Q &l, const Q &r){return l.w > r.w;}
4 struct Edmonds_Karp_Dijkstra{
5     FlowGraph &g; int V,s,t; vector<ll> pot;
6     Edmonds_Karp_Dijkstra(FlowGraph &g, int s, int t) :
7         g(g), V(g.size()), s(s), t(t), pot(V) {}
8     pair<ll,ll> run() { // return pair<f, cost>
9         ll maxflow = 0, cost = 0;
10        fill(F(pot), LLINF); pot[s]=0; // Bellman-Ford for potentials

```

```

11        REP(i,V-1) {
12            bool relax = false;
13            REP(u,V) if(pot[u] != LLINF) for(auto &e : g[u])
14                if(e.cap>e.f)
15                    if(pot[u] + e.cost < pot[e.v])
16                        pot[e.v] = pot[u] + e.cost, relax=true;
17            if(!relax) break;
18        }
19        REP(u,V) if(pot[u] == LLINF) pot[u] = 0;
20        while(true){
21            priority_queue<Q,vector<Q>,greater<Q>> q;
22            vector<vector<S>::iterator> p(V,g.front().end());
23            vector<ll> dist(V, LLINF); ll f, tf = -1;
24            q.push({s, LLINF, 0}); dist[s]=0;
25            while(!q.empty()){
26                auto u = q.top().u; ll w = q.top().w;
27                f = q.top().c; q.pop();
28                if(w!=dist[u]) continue; if(u==t && tf < 0) tf = f;
29                for(auto it = g[u].begin(); it!=g[u].end(); it++){
30                    const auto &e = *it;
31                    ll d = w + e.cost + pot[u] - pot[e.v];
32                    if(e.cap>e.f && d < dist[e.v]){
33                        q.push({e.v, min(f, e.cap-e.f),dist[e.v] = d});
34                        p[e.v]=it;
35                    }
36                }
37                auto it = p[t];
38                if(it == g.front().end()) return {maxflow,cost};
39                maxflow += f = tf;
40                while(it != g.front().end()){
41                    auto &r = g[it->v][it->r];
42                    cost += f * it->cost; it->f+=f;
43                    r.f -= f; it = p[r.v];
44                }
45                REP(u,V) if(dist[u]!=LLINF) pot[u] += dist[u];
46            }
47        };

```

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

### 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 constexpr long double EPS = 1e-10;
2 using C = double; // could be long long or long double
3 struct P { // may also be used as a 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 dot(const P &p) const { return x * p.x + y * p.y; }
12    C lensq() const { return x*x + y*y; }
13    C len() const { return sqrt(lensq()); }
14 };
15
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> pts) {
20     C sum = 0;
21     REP(i,pts.size()) sum += det(pts[i], pts[(i+1)%pts.size()]);
22     return sum;
23 }
24
25 double area(P p1, P p2, P p3) { return abs(det(p1, p2, p3)) / 2.0; }

```

```

26 double area(vector<P> polygon) { return abs(det(polygon)) / 2.0; }
27
28 // 1 when p1-p2-p3 is a left turn (when viewed from p1) [use EPS if needed]
29 int ccw(P p1, P p2, P p3) { C d = det(p1, p2, p3); return (d>0) - (d<0); }
30 struct S {
31     P p1, p2;
32     enum Type { Segment, Ray, Line } type;
33     S(P p1 = 0, P p2 = 0, Type type = Line) : p1(p1), p2(p2), type(type) {}
34     bool internal(P p) const {
35         if(det(p1,p2,p) > EPS) return false; // not on a line
36         switch(type){
37             case Segment: return dist(p1, p) + dist(p, p2) - dist(p1,p2) <= EPS;
38             case Ray: return dist(p,p2) - abs(dist(p1,p) - dist(p1,p2)) <= EPS;
39             default: return true;
40         }
41     }
42 };
43 struct L{
44     C a,b,c; // ax + by + c = 0
45     L(C a = 0, C b = 0, C c = 0) : a(a), b(b), c(c) {}
46     L(S s) : a(s.p2.y-s.p1.y), b(s.p1.x-s.p2.x),
47             c(s.p2.x*s.p1.y - s.p2.y*s.p1.x) {}
48     operator S(){
49         S s; s.type = S::Line;
50         if(abs(a)<EPS) s.p1 = {0, -c/b}, s.p2 = {1, -c/b};
51         else s.p1 = {-c/a, 0}, s.p2 = {-(c+b)/a, 1};
52         return s;
53     }
54 };
55 struct Circle{ P p; C r; };
56 P project(S s, P p) {
57     double l = (p-s.p1).dot(s.p2-s.p1)/double((s.p2-s.p1).dot(s.p2-s.p1));
58     switch(s.type){
59         case S::Segment: l = min(1.0, l);
60         case S::Ray: l = max(0.0, l);
61         default:;
62     }
63     return s.p1 + (s.p2 - s.p1) * l;
64 }
65 pair<bool,P> intersect(const L &l1, const L &l2) {
66     double x = l1.b*l2.c-l1.c*l2.b, y = l1.c*l2.a-l1.a*l2.c,
67            z = l1.a*l2.b-l1.b*l2.a;
68     return {z!=0, {x/z, y/z}};
69 }
70 vector<P> intersect(const Circle& cc, const L& l){
71     const double &x = cc.p.x, &y = cc.p.y, &r = cc.r, &a=l.a,&b=l.b,&c=l.c;
72     double n = a*a + b*b, t1 = c + a*x + b*y, D = n*r*r - t1*t1;
73     if(D<0) return {};
74     double xmid = b*b*x - a*(c + b*y), ymid = a*a*y - b*(c + a*x);
75     if(D==0) return {P{xmid/n, ymid/(n)}};
76     double sd = sqrt(D);
77     return {P{(xmid - b*sd)/n,(ymid + a*sd)/n},
78           P{(xmid + b*sd)/n,(ymid - a*sd)/n}};
79 }
80 vector<P> intersect(const Circle& c1, const Circle& c2){
81     C x = c1.p.x-c2.p.x, y = c1.p.y-c2.p.y;
82     const C &r1 = c1.r, &r2 = c2.r;
83     C n = x*x+y*y, D = -(n - (r1+r2)*(r1+r2))*(n - (r1-r2)*(r1-r2));

```



```

84 if(D<0) return {};
85 C xmid = x*(-r1*r1+r2*r2+n), ymid = y*(-r1*r1+r2*r2+n);
86 if(D==0) return {P{c2.p.x + xmid/(2.*n),c2.p.y + ymid/(2.*n)}};
87 double sd = sqrt(D);
88 return {P{c2.p.x + (xmid - y*sd)/(2.*n),c2.p.y + (ymid + x*sd)/(2.*n)},
89         P{c2.p.x + (xmid + y*sd)/(2.*n),c2.p.y + (ymid - x*sd)/(2.*n)}};
90 }

```

## 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:
65         // h[i%n], h[a]
66         // h[j], h[b]
67     }
68 }

```

## 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

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

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

cosine rule:

$$c^2 = a^2 + b^2 - 2ab \cos \gamma$$

Euler:

$$1 + CC = V - E + F$$

Pick:

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

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 constexpr ll SIZE = 1e6+10;
3 bitset<SIZE + 1> bs;
4 vector<ll> primes;
5
6 void sieve() { // call at start in main!
7     bs.set();
8     bs[0] = bs[1] = 0;
9     for (ll i = 2; i <= SIZE; i++) if (bs[i]) {
10         for (ll j = i * i; j <= SIZE; j += i) bs[j] = 0;
11         primes.push_back(i);
12     }
13 }
14
15 bool is_prime(ll n) { // for N <= SIZE^2
16     if (n <= SIZE) return bs[n];
17     for(const auto &prime : primes)
18         if (n % prime == 0) return false;
19     return true;
20 }
21
22 struct Factor{ll prime; ll exp;};
23 vector<Factor> factor(ll n) {
24     vector<Factor> factors;
25     for(const auto &prime : primes){
26         if(n==1 || prime*prime > n) break;
27         ll exp=0;
28         while(n % prime == 0)
29             n/=prime, exp++;
30         if(exp>0)
31             factors.push_back({prime,exp});
32     }
33     if (n != 1) factors.push_back({n,1});
34     return factors;
35 }
36
37 vector<ll> mf(SIZE + 1, -1); // mf[i]==i when prime
38 void sieve2() { // call at start in main!
39     mf[0] = mf[1] = 1;
40     for (ll i = 2; i <= SIZE; i++) if (mf[i] < 0) {
41         mf[i] = i;
42         for (ll j = i * i; j <= SIZE; j += i)
43             if(mf[j] < 0) mf[j] = i;
44         primes.push_back(i);
45     }
46 }
47
48 vector<Factor> factor2(ll n){
49     vector<Factor> factors;
50     while(n>1){
51         if(factors.back().prime == mf[n]) factors.back().exp++;
52         else factors.push_back({mf[n],1});
53         n/=mf[n];
54     }
55     return factors;
56 }
57
58 ll numDiv(ll n) {

```

```

59     ll divisors = 1;
60     for(auto &&p : factor(n))
61         divisors *= p.exp + 1;
62     return divisors;
63 }
64
65 ll bin_pow(ll b, ll e){
66     ll p = e==0 ? 1 : pow(b*b,e>>1);
67     return p * p * (e&1 ? b : 1);
68 }
69
70 ll sumDiv(ll n) {
71     ll sum = 1;
72     for(const auto &p : factor(n))
73         sum *= (pow(p.prime, p.exp+1) - 1) / (p.prime - 1);
74     return sum;
75 }
76
77 ll EulerPhi(ll n) {
78     ll ans = n;
79     for(const auto &p : factor(n))
80         ans -= ans / p.prime;
81     return ans;
82 }
83
84 vector<ll> test_primes = {2,3,5,7,11,13,17,19,23}; // sufficient to 3.8e18
85 bool miller_rabin(const ll n){ // true when prime
86     if(n<2) return false;
87     if(n%2==0) return n==2;
88     ll s = 0, d = n-1; // n-1 = 2^s * d
89     while(d&1) s++, d/=2;
90     for(auto a : test_primes){
91         if(a > n-2) break;
92         ll x = powmod(a,d,n); // needs powmod with mulmod!
93         if(x == 1 || x == n-1) continue;
94         REP(i,s-1){
95             x = mulmod(x,x,n);
96             if(x==1) return false;
97             if(x==n-1) goto next_it;
98         }
99         return false;
100     next_it::
101     }
102     return true;
103 }

```

## 7.2 Euler Phi

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

```

1 void calculate_phi(int N, vector<int> &phi) {
2     phi.assign(N + 1, 0);
3     iota(phi.begin(), phi.end(), 0); // numeric
4     for (int i = 2; i <= N; ++i) if (phi[i] == i)
5         for (int 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
37 // Finds a^n % m in O(lg n) time, ensure that a < m to avoid overflow!
38 ll powmod(ll a, ll n, ll m) {
39     if (n == 0) return 1;
40     if (n == 1) return a;
41     ll aa = (a*a)%m; // use mulmod when b > 1e9
42     if (n % 2 == 0) return powmod(aa, n / 2, m);
43     return (a * powmod(aa, (n - 1) / 2, m)) % m;
44 }
45
46 // Solve ax + by = c, returns false on failure.
47 bool linear_diophantine(ll a, ll b, ll c, ll &x, ll &y) {
48     ll d = gcd(a, b);
49     if (c % d) {
50         return false;
51     } else {
52         x = c / d * mod_inverse(a / d, b / d);
53         y = (c - a * x) / b;
54         return true;
55     }
56 }

```

```

57
58 // Chinese remainder theorem: finds z s.t. z % xi = ai. z is
59 // unique modulo M = lcm(xi). Returns (z, M), m = -1 on failure.
60 ii crm(ll x1, ll a1, ll x2, ll a2) {
61     ll s, t, d;
62     extended_euclid(x1, x2, s, t, d);
63     if (a1 % d != a2 % d) return ii(0, -1);
64     return ii(mod(s * a2 * x1 + t * a1 * x2, x1 * x2) / d, x1 * x2 / d);
65 }
66 ii crm(vi &x, vi &a){ // ii = pair<long,long>!
67     ii ret = ii(a[0], x[0]);
68     for (size_t i = 1; i < x.size(); ++i) {
69         ret = crm(ret.second, ret.first, x[i], a[i]);
70         if (ret.second == -1) break;
71     }
72     return ret;
73 }
74
75 ll binom(ll n, ll k){
76     ll ans = 1;
77     for(ll i = 1; i <= min(k,n-k); i++) ans *= (n-k+i), ans/=i;
78     return ans;
79 }

```

## 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 Complex Numbers

Faster-than-built-in complex numbers

```

1 struct Complex {
2     long double u,v;
3     Complex operator+(Complex r) const { return {u+r.u, v+r.v}; }
4     Complex operator-(Complex r) const { return {u-r.u, v-r.v}; }
5     Complex operator*(Complex r) const {
6         return {u * r.u - v * r.v, u * r.v + v * r.u};
7     }
8     Complex operator/(Complex r) {
9         auto norm = r.u*r.u+r.v*r.v;
10        return {(u * r.u + v * r.v) / norm, (v * r.u - u * r.v) / norm};
11    }
12    static
13    Complex exp(complex<ld> c){ c = std::exp(c); return {c.real(), c.imag()}; }
14 };

```

## 7.6 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 #define MY_PI 3.14159265358979323846
2 #include "../helpers/bitmasking.cpp"
3 #include <complex>
4 #include "complex.cpp"
5
6 // A.size() = N = 2^p
7 void fft(vector<Complex> &A, int N, int p, bool inv = false) {
8     for(int i = 0, r = 0; i < N; ++i, r = brinc(r, p))
9         if (i < r) swap(A[i], A[r]);
10    for (int m = 2; m <= N; m <= 1) {
11        Complex w, w_m = Complex::exp(complex<ld>(0, 2*MY_PI/m*(inv?-1:1)));
12        for (int k = 0; k < N; k += m) {
13            w = {1, 0};
14            for (int j = 0; j < m / 2; ++j) {
15                Complex t = w * A[k + j + m / 2];
16                A[k + j + m / 2] = A[k + j] - t;
17                A[k + j] = A[k + j] + t;
18                w = w * w_m;
19            }
20        }
21    }
22    if (inv) for (int i = 0; i < N; ++i) {
23        A[i].u /= N; A[i].v /= N;
24    }
25 }
26
27 void convolution(vector<Complex> &A, vector<Complex> &B, vector<Complex> &C){
28     // Pad with zeroes
29     int N = 2 * max(next_power_of_2(A.size()), next_power_of_2(B.size()));
30     A.reserve(N); B.reserve(N); C.reserve(N);
31     for (int i = A.size(); i < N; ++i) A.push_back({0, 0});
32     for (int i = B.size(); i < N; ++i) B.push_back({0, 0});
33     int p = int(log2(N) + 0.5);
34     // Transform A and B
35     fft(A, N, p, false);
36     fft(B, N, p, false);
37     // Calculate the convolution in C
38     for (int i = 0; i < N; ++i) C.push_back(A[i] * B[i]);
39     fft(C, N, p, true);
40 }
41
42 void square_inplace(vector<Complex> &A) {
43     int N = 2 * next_power_of_2(A.size());
44     A.reserve(N);
45     for (int i = A.size(); i < N; ++i) A.push_back({0, 0});
46     int p = int(log2(N) + 0.5);
47     fft(A, N, p, false);
48     for (int i = 0; i < N; ++i) A[i] = A[i] * A[i];
49     fft(A, N, p, true);
50 }

```

## 7.7 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 }

```

## 7.8 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 static M raise(const M &m, int n) {
17     if(n == 0) return id();
18     if(n == 1) return m;
19     auto r = (m*m).raise(n / 2);
20     return (n%2 ? m*r : r);
21 }
22 };

```

## 7.9 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     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 };

```

## 7.10 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;
5         vector<int> match;
6         Node() {
7             for (int i = 0; i < ALPHABET_SIZE; ++i) child[i] = -1;
8         }
9     };
10    vector<Node> a;
11    AC_FSM() { a.push_back(Node()); }
12    void construct_automaton(vector<string> &words) {
13        for (int w = 0; w < words.size(); ++w, n = 0) {
14            for (int i = 0; i < words[w].size(); ++i) {
15                if (a[n].child[mp(words[w][i])] == -1) {
16                    a[n].child[mp(words[w][i])] = a.size();
17                    a.push_back(Node());
18                }
19                n = a[n].child[mp(words[w][i])];
20            }
21            a[n].match.push_back(w);
22        }
23
24        queue<int> q;
25        for (int k = 0; k < ALPHABET_SIZE; ++k) {
26            if (a[0].child[k] == -1) a[0].child[k] = 0;
27            else if (a[0].child[k] > 0) {
28                a[a[0].child[k]].failure = 0;
29                q.push(a[0].child[k]);
30            }
31        }
32        while (!q.empty()) {
33            int r = q.front(); q.pop();
34            for (int k = 0; k < ALPHABET_SIZE; ++k) {
35                if (a[r].child[k] != -1) {
36                    q.push(a[r].child[k]);
37                    int v = a[r].failure;
38                    while (a[v].child[k] == -1) v = a[v].failure;
39                    a[a[r].child[k]].failure = a[v].child[k];
40                    for (int w : a[a[v].child[k]].match)
41                        a[a[r].child[k]].match.push_back(w);
42                }
43            }
44        }
45    }
46
47    void aho_corasick(string &sentence, vector<string> &words, vi &matches) {
48        matches.assign(words.size(), vector<int>());
49        int state = 0, ss = 0;
50        for (int i = 0; i < sentence.length(); ++i, ss = state) {
51            while (a[ss].child[mp(sentence[i])] == -1)
52                ss = a[ss].failure;
53            state = a[ss].child[mp(sentence[i])]
```

```

54         = a[ss].child[mp(sentence[i])];
55     for (int w : a[state].match)
56         matches[w].push_back(i - words[w].length() + 1);
57     }
58 }
59 };

```

## 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        }
20        i = r;
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)){}

```

## 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
11             rb = mrb; rbv = mrbv;
12             mrb = mlb; mrbv = mlbv;
13             mlb = lb + RES_PHI * (rb - lb);
14             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
7 bool bit_set(int mask, int pos) {
8     return ((mask & (1 << pos)) != 0);

```

```

9 }
10
11 // Iterate over all subsets of a set of size N
12 for (int mask = 0; mask < (1 << N); ++mask) {
13     // Decode mask here
14 }
15
16 // Iterate over all k-subsets of a set of size N
17 int mask = (1 << k) - 1;
18 while (!(mask & 1 << N)) {
19     // Decode mask here
20     int lo = mask & ~(mask - 1);
21     int lz = (mask + lo) & ~mask;
22     mask |= lz;
23     mask &= ~(lz - 1);
24     mask |= (lz / lo / 2) - 1;
25 }
26
27 // Iterate over all subsets of a subset
28 int mask = givenMask;
29 do {
30     // Decode mask here
31     mask = (mask - 1) & givenMask;
32 } while (mask != givenMask);
33
34 // The two functions below are used in the FFT:
35 inline int next_power_of_2(int x) {
36     x = (x - 1) | ((x - 1) >> 1);
37     x |= x >> 2; x |= x >> 4;
38     x |= x >> 8; x |= x >> 16;
39     return x + 1;
40 }
41
42 inline int brinc(int x, int k) {
43     int I = k - 1, s = 1 << I;
44     x ^= s;
45     if ((x & s) != s) {
46         I--; s >>= 1;
47         while (I >= 0 && ((x & s) == s)) {
48             x = x &~ s;
49             I--;
50             s >>= 1;
51         }
52         if (I >= 0) x |= s;
53     }
54    return x;
55 }

```

## 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 '\u': 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 }

```

## 11 Strategies

### 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