// Aho Corasick. --------------------------------------------------------

**struct** AhoCorasick {

**const** **int** N = 30030;

**int** fail[N];

**int** to[N][26];

**int** ending[N];

**int** sz;

    AhoCorasick() {

        sz = 1;

    }

**int** add(**const** string **&**s) {

**int** node = 1;

**for** (**int** i = 0; i < s.size(); ++i) {

**if** (!to[node][s[i] - 'a']) {

                to[node][s[i] - 'a'] = ++sz;

            }

            node = to[node][s[i] - 'a'];

        }

        ending[node] = **true**;

**return** node;

    }

**void** push() {

        queue<**int**> Q;

        Q.push(1);

        fail[1] = 1;

**while** (!Q.empty()) {

**int** u = Q.front(); Q.pop();

**for** (**int** i = 0; i < 26; ++i) {

**int** &v = to[u][i];

**if** (!v) {

                    v = u == 1 ? 1 : to[fail[u]][i];

                } **else** {

                    fail[v] = u == 1 ? 1 : to[fail[u]][i];

                    // G[fail[v]].push\_back(v);

                    ending[v] |= ending[fail[v]];

                    Q.push(v);

                }

            }

        }

    }

};

/\*

\* Blossom ---------------------------------------------------------------

\* Complexity: O(E\*sqrt(V))

\*/

**struct** Blossom {

**static** **const** **int** MAXV = 1**e**3 + 5;

**static** **const** **int** MAXE = 1**e**6 + 5;

**int** n, E, lst[MAXV], next[MAXE], adj[MAXE];

**int** nxt[MAXV], mat[MAXV], dad[MAXV], col[MAXV];

**int** que[MAXV], qh, qt;

**int** vis[MAXV], act[MAXV];

**int** tag, total;

**void** init(**int** n) {

**this**->n = n;

**for** (**int** i = 0; i <= n; i++) {

            lst[i] = nxt[i] = mat[i] = vis[i] = 0;

        }

        E = 1, tag = total = 0;

    }

**void** add(**int** u,**int** v) {

**if** (!mat[u] && !mat[v]) mat[u] = v, mat[v] = u, total++;

        E++, adj[E] = v, next[E] = lst[u], lst[u] = E;

        E++, adj[E] = u, next[E] = lst[v], lst[v] = E;

    }

**int** lca(**int** u, **int** v) {

        tag++;

**for**(; ; swap(u, v)) {

**if** (u) {

**if** (vis[u = dad[u]] == tag) {

**return** u;

                }

                vis[u] = tag;

                u = nxt[mat[u]];

            }

        }

    }

**void** blossom(**int** u, **int** v, **int** g) {

**while** (dad[u] != g) {

            nxt[u] = v;

**if** (col[mat[u]] == 2) {

                col[mat[u]] = 1;

                que[++qt] = mat[u];

            }

**if** (u == dad[u]) dad[u] = g;

**if** (mat[u] == dad[mat[u]]) dad[mat[u]] = g;

            v = mat[u];

            u = nxt[v];

        }

}

**int** augument(**int** s) {

**for** (**int** i = 1; i <= n; i++) {

            col[i] = 0;

            dad[i] = i;

        }

        qh = 0; que[qt = 1] = s; col[s] = 1;

**for** (**int** u, v, i; qh < qt; ) {

            act[u = que[++qh]] = 1;

**for** (i = lst[u];i ; i = next[i]) {

                v = adj[i];

**if** (col[v] == 0) {

                    nxt[v] = u;

                    col[v] = 2;

**if** (!mat[v]) {

**for** (; v; v = u) {

                            u = mat[nxt[v]];

                            mat[v] = nxt[v];

                            mat[nxt[v]] = v;

                        }

**return** 1;

                    }

                    col[mat[v]] = 1;

                    que[++qt] = mat[v];

                }

**else** **if** (dad[u] != dad[v] && col[v] == 1) {

**int** g = lca(u, v);

                    blossom(u, v, g);

                    blossom(v, u, g);

**for** (**int** j = 1; j <= n; j++) {

                        dad[j] = dad[dad[j]];

                    }

                }

            }

        }

**return** 0;

    }

**int** maxmat() {

**for** (**int** i = 1; i <= n; i++) {

**if** (!mat[i]) {

                total += augument(i);

            }

        }

**return** total;

    }

}

// Debug.----------------------------------------------------------------

**void** DEBUG() {

**std**::cerr << "Unsuccesfull Console Valuable;";

}

**template**<**class** T, **class** ...U> **void** DEBUG(T a, U ...b) {

**std**::cerr << a;

**if**(sizeof...(b)) {

**std**::cerr << " | ";

        DEBUG(b...);

    }

}

#define console(...) \

**std**::cerr << "Line " << \_\_LINE\_\_ << "| "<<  #\_\_VA\_ARGS\_\_ << ": ", DEBUG(\_\_VA\_ARGS\_\_), **std**::cerr << '\n'

**template**<**class** T1, **class** T2>

**std**::ostream**&** operator << (**std**::ostream**&** cout, **std**::pair<T1, T2> Q) {

    cout << "[" << Q.first << ", " << Q.second << "]";

**return** cout;

}

**template**<**class** Con, **class** = **decltype**(**std**::declval<Con>().begin())>

**typename** **std**::**enable\_if**<!**std**::**is\_same**<Con, **std**::string>::value, **std**::ostream&>::type

operator << (**std**::ostream**&** cout, Con Q) {

    cout << "[SIZE = " << Q.size() << "]: {";

**for**(**auto** It = Q.begin(); It != Q.end(); ++It)

        cout << (It == Q.begin() ? "" : ", ") << \*It;

    cout << "}";

**return** cout;

}

**template**<**class** Con, **class** T = **decltype**(**std**::declval<Con>().top())>

T Pick(Con u) { **return** u.top(); }

**template**<**class** T>

T Pick(**std**::queue<T> u) { **return** u.front(); }

**template**<**class** Con, **class** = **decltype**(Pick(**std**::declval<Con>()))>

**std**::ostream**&** operator << (**std**::ostream**&** cout, Con Q) {

    cout << "[SIZE = " << Q.size() << "]: {";

**if**(Q.size())

**for**(cout << Pick(Q), Q.pop(); Q.size(); Q.pop())

        cout << ", " << Pick(Q);

    cout << "}";

**return** cout;

}

// ExtendEuclide --------------------------------------------------------

**int** bezout(**int** a, **int** b) {

    // return x such that ax + by == gcd(a, b)

**int** xa = 1, xb = 0;

**while** (b) {

**int** q = a / b;

**int** r = a - q \* b, xr = xa - q \* xb;

        a = b; xa = xb;

        b = r; xb = xr;

    }

**return** xa;

}

pair<**int**, **int**> solve(**int** a, **int** b, **int** c) {

    // solve ax + by == c

**int** d = \_\_gcd(a, b);

**int** x = bezout(a, b);

**int** y = (d - a \* x) / b;

    c /= d;

**return** make\_pair(x \* c, y \* c);

}

// Fenwick Tree.--------------------------------------------------------

**template**<**class** Tp>

**struct** fenwick {

    fenwick() = **default**;

    // Construct BIT in range [1, n).

    fenwick(**int** n) {

        f.assign(n, 0);

    }

    // a[id] += val.

**void** update(**int** id, Tp val) {

**for**(; id < (**int**) f.size(); id += id&-id)

            f[id] += val;

    }

    // return a[1] + a[2] + ... + a[id].

    Tp query(**int** id) {

        Tp ret = 0;

**for**(; id > 0; id -= id&-id)

            ret += f[id];

**return** ret;

    }

    // return a[l] + a[l + 1] + ... + a[r].

    Tp query(**int** l, **int** r) {

**return** query(r) - query(l - 1);

    }

    // Minimum idx st: a[1] + a[2] + ... + a[idx] >= sum.

**int** lower\_bound(Tp sum) {

**int** n = f.size(), i = 0;

**for**(**int** k = 31 - \_\_builtin\_clz(n - 1); k >= 0; --k)

**if**(i + (1 << k) <= n && f[i + (1 << k)] < sum) sum -= f[i += (1 << k)];

**return** i + 1;

    }

    // Minimum idx st: a[1] + a[2] + ... + a[idx] > sum.

**int** upper\_bound(Tp sum) {

**int** n = f.size(), i = 0;

**for**(**int** k = 31 - \_\_builtin\_clz(n - 1); k >= 0; --k)

**if**(i + (1 << k) <= n && f[i + (1 << k)] <= sum) sum -= f[i += (1 << k)];

**return** i + 1;

    }

**private**:

    vector<**int**> f;

};

// LiChaoTree ----------------------------------------------------------

**const** ll inf = 1**LL** << 60;

**const** **int** N = 100100;

**struct** Line {

    ll a = 0, b = -inf;

    Line() = **default**;

    Line(ll a, ll b): a(a), b(b) {}

    ll eval(ll x) {

**return** a \* x + b;

    }

} IT[N << 2];

**void** update(**int** id, **int** l, **int** r, **int** s, **int** t, Line val) {

**if**(l > t || r < s) **return**;

**int** m = (l + r) >> 1;

**if**(s <= l && r <= t) {

**if**(IT[id].eval(l) >= val.eval(l) && IT[id].eval(r) >= val.eval(r)) **return**;

**if**(IT[id].eval(l) <= val.eval(l) && IT[id].eval(r) <= val.eval(r)) **return** **void**(IT[id] = val);

**if**(IT[id].eval(l) >= val.eval(l) && IT[id].eval(m) >= val.eval(m))

**return** update(id << 1 | 1, m + 1, r, s, t, val);

**if**(IT[id].eval(l) <= val.eval(l) && IT[id].eval(m) <= val.eval(m)) {

            update(id << 1 | 1, m + 1, r, s, t, IT[id]); IT[id] = val;

**return**;

        }

**if**(IT[id].eval(m + 1) >= val.eval(m + 1) && IT[id].eval(r) >= val.eval(r))

**return** update(id << 1, l, m, s, t, val);

**if**(IT[id].eval(m + 1) <= val.eval(m + 1) && IT[id].eval(r) <= val.eval(r)) {

            update(id << 1, l, m, s, t, IT[id]); IT[id] = val;

**return**;

        }

    }

    update(id << 1, l, m, s, t, val);

    update(id << 1 | 1, m + 1, r, s, t, val);

}

ll query(**int** id, **int** l, **int** r, **int** x) {

**if**(l > x || r < x) **return** -inf;

**if**(l == r) **return** IT[id].eval(x);

**int** m = (l + r) >> 1;

**return** max({IT[id].eval(x), query(id << 1, l, m, x), query(id << 1 | 1, m + 1, r, x)});

}

// Line Container -------------------------------------------------------

**inline** **namespace** **\_LineContainer** {

**typedef** **long** **long** ll;

**const** ll LINF = LLONG\_MAX;

**bool** \_Line\_Comp;

**struct** Line {

        // k is slope, m is intercept, p is intersection point

**mutable** ll k, m, p;

**bool** **operator**<(**const** Line **&**o) **const** { **return** \_Line\_Comp ? p < o.p : k < o.k; }

    };

**struct** LineContainer : multiset<Line> {

        ll div(ll a, ll b) { **return** a / b - ((a ^ b) < 0 && a % b); }

**bool** isect(iterator x, iterator y) {

**if** (y == end()) **return** (x->p = LINF, **false**);

**if** (x->k == y->k) x->p = (x->m > y->m ? LINF : -LINF);

**else** x->p = div(y->m - x->m, x->k - y->k);

**return** (x->p >= y->p);

        }

**void** add(ll k, ll m) {

**auto** z = insert({k, m, 0}), y = z++, x = y;

**while** (isect(y, z)) z = erase(z);

**if** (x != begin() && isect(--x, y)) isect(x, y = erase(y));

**while** ((y = x) != begin() && (--x)->p >= y->p) isect(x, erase(y));

        }

        ll query(ll x) {

            assert(!empty());

            \_Line\_Comp = 1;

**auto** l = \*lower\_bound({0, 0, x});

            \_Line\_Comp = 0;

**return** l.k \* x + l.m;

        }

    };

}  // namespace \_LineContainer

// Manhattan MST --------------------------------------------------------

**struct** pt {

**long** **long** x, y;

    pt() {}

    pt(**long** **long** \_x, **long** **long** \_y) : x(\_x), y(\_y) {}

};

**struct** edge {

**int** u, v; **long** **long** c;

    edge() {}

    edge(**int** \_u, **int** \_v, **long** **long** \_c) : u(\_u), v(\_v), c(\_c) {}

**bool** **operator**<(**const** edge **&**rhs) **const** { **return** c < rhs.c; }

};

**struct** disjoint\_set {

    vector<**int**> rt, rnk;

    disjoint\_set(**int** \_sz) : rt(\_sz + 1), rnk(\_sz + 1, 1) { iota(all(rt), 0); }

**int** find(**int** x) { **return** (x == rt[x] ? x : rt[x] = find(rt[x])); }

**bool** unite(**int** u, **int** v) {

        u = find(u), v = find(v); **if** (u == v) **return** **false**;

**if** (rnk[u] < rnk[v]) swap(u, v);

**return** rt[v] = u, rnk[u] += rnk[v], **true**;

    }

};

vector<edge> manhattan\_mst(vector<pt> arr) {

    vector<edge> res1;

    vector<**int**> idx(arr.size()); iota(all(idx), 0);

**for** (**int** dir = 0; dir < 4; dir++) {

**for** (**auto** &it : arr) {

**if** (dir & 1) swap(it.x, it.y);

**else** **if** (dir == 2) it.x = -it.x;

        }

        sort(all(idx), [&arr](**int** u, **int** v) { **return** arr[u].x - arr[v].x < arr[v].y - arr[u].y; });

        map<**long** **long**, **int**> mp;

**for** (**int** u : idx) {

**for** (**auto** it = mp.lower\_bound(-arr[u].y); it != mp.end(); it = mp.erase(it)) {

**int** v = it->second;

**if** (arr[u].x - arr[v].x < arr[u].y - arr[v].y) **break**;

                res1.emplace\_back(u, v, arr[u].x - arr[v].x + arr[u].y - arr[v].y);

            }

            mp.emplace(-arr[u].y, u);

        }

    }

    sort(all(res1));

    disjoint\_set conn(res1.size());

    vector<edge> res2;

**for** (**auto** e : res1) **if** (conn.unite(e.u, e.v)) res2.push\_back(e);

**return** res2;

}

// Rabin Miller.--------------------------------------------------------

**using** ull = **unsigned** **long** **long**;

ull mult(ull x, ull y, ull mod) {

**return** \_\_int128\_t(x) \* y % mod;

}

ull powMod(ull x, ull p, ull mod) {

    ull ret = 1;

**for**(; p; p >>= 1, x = mult(x, x, mod))

**if**(p & 1) ret = mult(ret, x, mod);

**return** ret;

}

**bool** checkMillerRabin(ull x, ull mod, ull s, **int** k) {

    x = powMod(x, s, mod);

**if** (x == 1) **return** **true**;

**while**(k--) {

**if** (x == mod - 1) **return** **true**;

        x = mult(x, x, mod);

**if** (x == 1) **return** **false**;

    }

**return** **false**;

}

**bool** is\_prime(ull x) {

**for**(ull z : {2, 3, 5, 7})

**if**(x % z == 0) **return** x == z;

**if** (x < 121) **return** x > 1;

    ull s = x - 1;

**int** k = 0;

**while** (s % 2 == 0) {

        s /= 2;

        k++;

    }

**if** (x < 1**LL** << 32) {

**for** (ull z : {2, 7, 61}) {

**if** (!checkMillerRabin(z, x, s, k)) **return** **false**;

        }

    } **else** {

**for** (ull z : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {

**if** (!checkMillerRabin(z, x, s, k)) **return** **false**;

        }

    }

**return** **true**;

}

// Strings -------------------------------------------------------------

vector<**int**> kmp(**const** string **&**s) {

**int** \_n = (**int**)s.size();

    vector<**int**> ret = vector<**int**>(\_n);

**for** (**int** i = 1, j = 0; i < \_n; i++) {

**while** (j && s[i] != s[j]) j = ret[j - 1];

**if** (s[i] == s[j]) j++;

        ret[i] = j;

    }

**return** ret;

}

vector<**int**> Z(**const** string **&**s) {

**int** \_n = (**int**)s.size();

    vector<**int**> ret = vector<**int**>(\_n);

**for** (**int** i = 1, l = 0, r = 0; i < \_n; i++) {

**if** (i <= r) ret[i] = min(ret[i - l], r - i + 1);

**while** (i + ret[i] < \_n && s[ret[i]] == s[i + ret[i]]) ret[i]++;

**if** (i + ret[i] - 1 > r) l = i, r = i + ret[i] - 1;

    }

**return** ret;

}

array<vector<**int**>, 2> manacher(**const** string **&**s) {

**int** n = (**int**)s.size();

    array<vector<**int**>, 2> p = {vector<**int**>(n), vector<**int**>(n)};

**for** (**int** z = 0; z < 2; z++) { // z = 0: even, z = 1: odd

**for** (**int** i = 0, l = 0, r = 0; i < n; i++) {

**if** (i < r) p[z][i] = min(r - i + !z, p[z][l + r - i + !z]);

**int** L = i - p[z][i], R = i + p[z][i] - !z;

**while** (L && R + 1 < n && s[L - 1] == s[R + 1]) p[z][i]++, L--, R++;

**if**(R > r) l = L, r = R;

        }

**if** (z) **for** (**int** i = 0; i < n; i++) p[z][i]++;

    }

**return** p;

}

// Treap ----------------------------------------------------------------

mt19937 rd(time(**NULL**));

**int** randint(**int** l, **int** r) {

**return** uniform\_int\_distribution<**int**>(l, r) (rd);

}

**struct** Tnode {

    Tnode \*l = **NULL**, \*r = **NULL**;

**char** key; **bool** rev = **false**;

**int** size = 1, prior = 0;

    Tnode(**char** key = **char**(), **int** prior = randint(-1 << 30, 1 << 30)): key(key), prior(prior) {}

};

**using** Pnode = Tnode\*;

**int** SIZE(Pnode u) {

**return** (u ? u->size : 0);

}

Pnode FIX(Pnode u) {

**if**(u) u->size = 1 + SIZE(u->l) + SIZE(u->r);

**return** u;

}

**void** Down(Pnode t) {

**if**(t->rev) {

        swap(t->l, t->r);

**if**(t->l) t->l->rev ^= 1;

**if**(t->r) t->r->rev ^= 1;

        t->rev ^= 1;

    }

}

Pnode merge(Pnode l, Pnode r) {

**if**(!l || !r) **return** (l ? l : r);

    Down(l); Down(r);

**if**(l->prior > r->prior) {

        l->r = merge(l->r, r);

**return** FIX(l);

    } **else** {

        r->l = merge(l, r->l);

**return** FIX(r);

    }

}

pair<Pnode, Pnode> split(Pnode t, **int** k) {

**if**(!t) **return** {**NULL**, **NULL**};

**else** Down(t);

    Pnode l = **NULL**, r = **NULL**;

**if**(k <= SIZE(t->l)) tie(l, t->l) = split(t->l, k), r = t;

**else**                tie(t->r, r) = split(t->r, k - 1 - SIZE(t->l)), l = t;

    FIX(t);

**return** {l, r};

}

tuple<Pnode, Pnode, Pnode> split(Pnode t, **int** u, **int** v) {

**if**(!t) **return** {**NULL**, **NULL**, **NULL**};

    Pnode l = **NULL**, m = **NULL**, r = **NULL**;

    tie(t, r) = split(t, v + 1);

    tie(l, m) = split(t, u);

**return** {l, m, r};

}

**void** DFS(Pnode t) {

**if**(!t) **return**;

**else** Down(t);

    DFS(t->l);

    cout << t->key;

    DFS(t->r);

}

Pnode root = **NULL**;

// Two-Sat --------------------------------------------------------------

**const** **int** N = 200200;

**int** n;

vector<**int**> G[N];

array<**int**, 3> E[N];

**int** low[N], ord[N], id[N], timer = 0, scc = 0;

stack<**int**> stk;

#define ID(u) (u \* 2 - 1)

**void** push(**int** u, **int** v) {

    G[u ^ 1].push\_back(v);

    G[v ^ 1].push\_back(u);

}

**void** dfs(**int** u) {

    low[u] = ord[u] = ++timer;

    stk.push(u);

**for**(**const** **int**& v : G[u]) {

**if**(ord[v]) low[u] = min(low[u], ord[v]);

**else** {

            dfs(v);

            low[u] = min(low[u], low[v]);

        }

    }

**if**(low[u] == ord[u]) {

**for**(++scc; ; ) {

**int** v = stk.top(); stk.pop();

            ord[v] = 1 << 30;

            id[v] = scc;

**if**(u == v) **break**;

        }

    }

}

vector<**int**> TwoSat() {

**for**(**int** i = 0; i <= ID(n); ++i)

**if**(!ord[i]) dfs(i);

**for**(**int** i = 1; i <= ID(n); i += 2) {

**if**(id[i] == id[i ^ 1]) **return** {};

    }

    vector<**int**> ret;

**for**(**int** i = 1; i <= ID(n); i += 2) {

**if**(id[i] < id[i ^ 1])

            ret.push\_back((i + 1) / 2);

    }

**return** ret;

}

// berlekampMassey ------------------------------------------------------

**template**<**typename** T> vector<T> berlekampMassey(**const** vector<T> **&**sequence) {

**int** n = (**int**)sequence.size(), len = 0, m = 1;

    vector<T> prevBest(n), coefficients(n);

    T prevDelta = prevBest[0] = coefficients[0] = 1;

**for** (**int** i = 0; i < n; i++, m++) {

        T delta = sequence[i];

**for** (**int** j = 1; j <= len; j++) delta += coefficients[j] \* sequence[i - j];

**if** ((**long** **long**)delta == 0) **continue**;

        vector<T> temp = coefficients;

        T coef = delta / prevDelta;

**for** (**int** j = m; j < n; j++) coefficients[j] -= coef \* prevBest[j - m];

**if** ((len << 1) <= i)

            len = i + 1 - len, prevBest = temp, prevDelta = delta, m = 0;

    }

    coefficients.resize(len + 1);

    coefficients.erase(coefficients.begin());

**for** (T &x : coefficients) x = -x;

**return** coefficients;

}

**template**<**typename** T> T calcKthTerm(

**const** vector<T> **&**coefficients, **const** vector<T> **&**sequence, **long** **long** k

) {

    assert(coefficients.size() <= sequence.size());

**int** n = (**int**)coefficients.size();

**auto** mul = [&](**const** vector<T> **&**a, **const** vector<T> **&**b) {

        vector<T> res(a.size() + b.size() - 1**u**);

**for** (**int** i = 0; i < (**int**)a.size(); i++)

**for** (**int** j = 0; j < (**int**)b.size(); j++)

                res[i + j] += a[i] \* b[j];

**for** (**int** i = (**int**)res.size() - 1; i >= n; i--)

**for** (**int** j = n - 1; j >= 0; j--)

                res[i - j - 1] += res[i] \* coefficients[j];

        res.resize(min((**int**)res.size(), n));

**return** res;

    };

    vector<T> a = (n == 1 ? vector<T>{coefficients[0]} : vector<T>{0, 1}), x{1};

**for** (; k; k >>= 1) {

**if** (k & 1) x = mul(x, a);

        a = mul(a, a);

    }

    x.resize(n);

    T res = 0;

**for** (**int** i = 0; i < n; i++) res += x[i] \* sequence[i];

**return** res;

}

// Mincost-Maxflow -----------------------------------------------------

**template**<**class** F, **class** C> **struct** MCMF {

**private**:

**template**<**class** T> **bool** ckmin(T **&**u, T v) { **return** u > v ? (u = v, 1) : 0; }

**struct** edge { **int** v; F flow, cap; C cost; };

**int** \_n;

    vector<C> p, dist;

    vector<**int**> prv;

    vector<vector<**int**>> adj;

    vector<edge> el;

**public**:

    MCMF(**int** n) : \_n(n), p(n), dist(n), prv(n), adj(n) {}

**void** add\_edge(**int** u, **int** v, F cap, C cost) {

        assert(cap > 0);

        adj[u].push\_back(sz(el)); el.push\_back({ v, 0, cap, cost });

        adj[v].push\_back(sz(el)); el.push\_back({ u, 0, 0, -cost });

    }

**bool** path(**int** s, **int** t) {

**const** C INF = **numeric\_limits**<C>::max();

        fill(all(dist), INF);

**using** T = pair<C, **int**>;

        priority\_queue<T, vector<T>, greater<T>> pq; pq.emplace(dist[s] = 0, s);

**while** (!pq.empty()) {

            T u = pq.top(); pq.pop();

**if** (u.first > dist[u.second]) **continue**;

**for** (**auto** i : adj[u.second]) {

**const** edge &e = el[i];

**if** (e.flow < e.cap &&

                ckmin(dist[e.v], u.first + e.cost + p[u.second] - p[e.v]))

                    prv[e.v] = i, pq.emplace(dist[e.v], e.v);

            }

        }

**return** dist[t] != INF;

    }

    pair<F, C> calc(**int** s, **int** t) {

        assert(s != t);

**for** (**int** i = 0; i < \_n; i++) **for** (**int** j = 0; j < sz(el); j++) {

**const** edge &e = el[j];

**if** (e.cap) ckmin(p[e.v], p[el[j ^ 1].v] + e.cost);

        }

        F totFlow = 0;

        C totCost = 0;

**while** (path(s, t)) {

**for** (**int** i = 0; i < \_n; i++) p[i] += dist[i];

            F df = **numeric\_limits**<F>::max();

**for** (**int** i = t; i != s; i = el[prv[i] ^ 1].v) {

**const** edge &e = el[prv[i]];

                ckmin(df, e.cap - e.flow);

            }

            totFlow += df, totCost += (p[t] - p[s]) \* df;

**for** (**int** x = t; x != s; x = el[prv[x] ^ 1].v)

                el[prv[x]].flow += df, el[prv[x] ^ 1].flow -= df;

        }

**return** { totFlow, totCost };

    }

};

// Maxflow --------------------------------------------------------------

**template**<**class** S> **struct** flow\_network {

**private**:

**int** \_n;

**template**<**class** T> **struct** edge { **int** to, rev; T f, c; };

    vector<vector<edge<S>>> g;

    vector<S> xs;

    vector<stack<**int**>> ovfl;

    vector<**int**> h, cnt\_h;

    vector<edge<S> \*> cur;

**public**:

    flow\_network() : \_n(0) {}

    flow\_network(**int** n) : \_n(n), g(n), xs(n),

        ovfl(n << 1), h(n), cnt\_h(n << 1), cur(n) {}

**void** add\_edge(**int** u, **int** v, S cap, S rcap = S(0)) {

        assert(0 <= u && u < \_n && 0 <= v && v < \_n);

        assert(S(0) <= cap);

**if** (u == v) **return**;

        g[u].push\_back({v, sz(g[v]), 0, cap});

        g[v].push\_back({u, sz(g[u]) - 1, 0, rcap});

    }

**void** push(edge<S> **&**e, S delta) {

        edge<S> &r = g[e.to][e.rev];

**if** (!xs[e.to] && delta) ovfl[h[e.to]].push(e.to);

        e.f += delta, e.c -= delta, xs[e.to] += delta;

        r.f -= delta, r.c += delta, xs[r.to] -= delta;

    }

    S calc(**int** s, **int** t) {

        assert(0 <= s && s < \_n && 0 <= t && t < \_n);

        h[s] = \_n, xs[t] = 1, cnt\_h[0] = \_n - 1;

**for** (**int** i = 0; i < \_n; i++) cur[i] = g[i].data();

**for** (**auto** &e : g[s]) push(e, e.c);

**for** (**int** cur\_h = 0;;) {

**while** (ovfl[cur\_h].empty()) **if** (!cur\_h--) **return** -xs[s];

**int** u = ovfl[cur\_h].top(); ovfl[cur\_h].pop();

**while** (xs[u] > 0) { // discharge u

**if** (cur[u] == g[u].data() + sz(g[u])) {

                    h[u] = (\_n << 1);

**for** (**auto** &e : g[u]) **if** (e.c && h[u] > h[e.to] + 1)

                        h[u] = h[e.to] + 1, cur[u] = &e;

**if** (++cnt\_h[h[u]], !--cnt\_h[cur\_h] && cur\_h)

**for** (**int** i = 0; i < \_n; i++)

**if** (cur\_h < h[i] && h[i] < \_n)

                                cnt\_h[h[i]]--, h[i] = \_n + 1;

                    cur\_h = h[u];

                } **else** **if** (cur[u]->c && h[u] == h[cur[u]->to] + 1)

                    push(\*cur[u], min(xs[u], cur[u]->c));

**else** ++cur[u];

            }

        }

    }

**bool** left\_of\_min\_cut(**int** u) { **return** h[u] >= \_n; }

};

// Segment tree ---------------------------------------------------------

**template**<**class** S, S (\*op)(S, S), S (\*e)()> **struct** segtree {

**private**:

**int** \_n, \_size;

    vector<S> d;

**void** pull(**int** k) { d[k] = op(d[k << 1], d[k << 1 | 1]); }

**public**:

    segtree() : segtree(0) {}

    segtree(**int** n) : segtree(vector<S>(n, e())) {}

    segtree(**const** vector<S> **&**v) : \_n(sz(v)) {

        \_size = (\_n == 1 ? 1 : 1 << (32 - \_\_builtin\_clz(\_n - 1)));

        d = vector<S>(\_size << 1, e());

**for** (**int** i = 0; i < \_n; i++) d[i + \_size] = v[i];

**for** (**int** i = \_size - 1; i >= 1; i--) pull(i);

    }

**void** replace(**int** p, S x) {

        assert(0 <= p && p < \_n);

        d[p + \_size] = x;

**for** (p = (p + \_size) >> 1; p > 0; p >>= 1) pull(p);

    }

    S query(**int** p) **const** {

        assert(0 <= p && p < \_n);

**return** d[p + \_size];

    }

    S query(**int** l, **int** r) **const** { // [l, r)

        assert(0 <= l && l <= r && r <= \_n);

        S sml = e(), smr = e();

**for** (l += \_size, r += \_size; l < r; l >>= 1, r >>= 1) {

**if** (l & 1) sml = op(sml, d[l++]);

**if** (r & 1) smr = op(d[--r], smr);

        }

**return** op(sml, smr);

    }

    S query\_all() **const** { **return** d[1]; }

**template**<**bool** (\*f)(S)> **int** max\_right(**int** l) **const** {

**return** max\_right(l, [](S x) { **return** f(x); });

    }

**template**<**class** F> **int** max\_right(**int** l, F f) **const** {

        assert(0 <= l && l <= \_n); assert(f(e()));

**if** (l == \_n) **return** \_n;

        l += \_size;

        S sm = e(), v;

**do** {

            l >>= \_\_builtin\_ctz(l);

**if** (!f(op(sm, d[l]))) {

**while** (l < \_size) {

                    l <<= 1, v = op(sm, d[l]);

**if** (f(v)) sm = v, l++;

                }

**return** l - \_size;

            }

            sm = op(sm, d[l++]);

        } **while** (l != (l & -l));

**return** \_n;

    }

**template**<**bool** (\*f)(S)> **int** min\_left(**int** r) **const** {

**return** min\_left(r, [](S x) { **return** f(x); });

    }

**template**<**class** F> **int** min\_left(**int** r, F f) **const** {

        assert(0 <= r && r <= \_n); assert(f(e()));

**if** (r == 0) **return** 0;

        r += \_size;

        S sm = e(), v;

**do** {

            r--; r >>= \_\_builtin\_ctz(~r); **if** (!r) r = 1;

**if** (!f(op(d[r], sm))) {

**while** (r < \_size) {

                    r = r << 1 | 1, v = op(d[r], sm);

**if** (f(v)) sm = v, r--;

                }

**return** r + 1 - \_size;

            }

            sm = op(d[r], sm);

        } **while** (r != (r & -r));

**return** 0;

    }

};

// Segment Tree Beats (Cây phân đoạn hai max) ---------------------------

**template**<**class** T = int, **T** starto = -1>

**struct** stb {

**private**:

**struct** Node {

        T max1 = T(), max2 = starto, cmax = 1, sune = T(), lz = starto;

        Node(T val = T()): max1(val), sune(val) {}

**void** setMin(T val) { // Minimize a[i] with val, i: [l, r]

            assert(T(max2) < T(val));

**if**(T(max1) <= T(val)) **return**;

            sune -= (max1 - val) \* cmax;

            lz = max1 = val;

        }

        Node operator+ (**const** Node**&** P) {

            Node res;

            res.max1 = max(max1, P.max1), res.max2 = max(max2, P.max2);

            res.cmax = 0, res.lz = starto, res.sune = sune + P.sune;

**if**(max1 != res.max1) res.max2 = max(res.max2, max1);

**else** res.cmax += cmax;

**if**(P.max1 != res.max1) res.max2 = max(res.max2, P.max1);

**else** res.cmax += P.cmax;

**return** res;

        }

    };

    Node it[8 \* N];

**void** down(**int** id) {

**if**(it[id].lz == starto) **return**;

        it[id << 1].setMin(it[id].lz);

        it[id << 1 | 1].setMin(it[id].lz);

        it[id].lz = starto;

    }

**public**:

**void** Init(**int** id, **int** left, **int** right, T \*a) {

**if**(left > right) **return**;

**if**(left == right) **return** it[id] = Node(a[left]), **void**();

**int** mid = (left + right) >> 1;

        Init(id << 1, left, mid, a);

        Init(id << 1 | 1, mid + 1, right, a);

        it[id] = it[id << 1] + it[id << 1 | 1];

    }

    // Minimize a[i] with c, i: [l, r]

**void** UpdateMin(**int** id, **int** left, **int** right, **int** sta, **int** endo, T c) {

**if**(left > endo or right < sta or it[id].max1 <= c) **return**;

**if**(sta <= left and right <= endo and it[id].max2 < c) **return** it[id].setMin(c), **void**();

        down(id);

**int** mid = (left + right) >> 1;

        UpdateMin(id << 1, left, mid, sta, endo, c);

        UpdateMin(id << 1 | 1, mid + 1, right, sta, endo, c);

        it[id] = it[id << 1] + it[id << 1 | 1];

    }

    // Return sum of a[i], i: [l, r]

    T GetSum(**int** id, **int** left, **int** right, **int** sta, **int** endo) {

**if**(left > endo or right < sta) **return** 0;

**if**(sta <= left and right <= endo) **return** it[id].sune;

        down(id);

**int** mid = (left + right) >> 1;

**return** GetSum(id << 1, left, mid, sta, endo) + GetSum(id << 1 | 1, mid + 1, right, sta, endo);

    }

    T GetMax(**int** id, **int** left, **int** right, **int** sta, **int** endo) {

**if**(left > endo or right < sta) **return** -1;

**if**(sta <= left and right <= endo) **return** it[id].max1;

        down(id);

**int** mid = (left + right) >> 1;

**return** max(GetMax(id << 1, left, mid, sta, endo), GetMax(id << 1 | 1, mid + 1, right, sta, endo));

    }

};

/\*\*

 \* Operator +, -, \*, /, ==, !=

 \* Modulus

 \* mod: số modulus

 \* Mul(c, d) : Nhân 2 số với số modulus lớn

 \* Pow(c, exp) : Lũy thừa bậc exp của c

 \* BuildMOD(m) : Xây dựng hàm giai thừa từ 1 tới m

 \* nCk(n, k) : Tổ hợp chập k của n

 \* Ps: n <= m với BuildMOD(m) đã được từ trước.

\*/

// ModInt --------------------------------------------------------------

**namespace** **md**

{

**using** T = **int**;

    T mod = 1**e**9 + 7;

**struct** Modulus

    {

        ll x = T();

**inline** **void** fix() { **while**(x >= mod) x -= mod; }

        Modulus(ll v = ll()) { x = v; fix(); }

**friend** Modulus Mul(Modulus c, Modulus d); **friend** Modulus Pow(Modulus c, ll exp); **friend** Modulus Inv(Modulus c);

        Modulus operator + (**const** Modulus**&** c) { **return** Modulus(x + c.x); }

        Modulus operator - (**const** Modulus**&** c) { **return** Modulus(x - c.x + mod); }

        Modulus operator \* (**const** Modulus**&** c) { **return** Mul(\***this**, c.x); }

        Modulus operator / (**const** Modulus**&** c) { **return** Mul(\***this**, Inv(c)); }

**bool** operator == (**const** Modulus**&** c) { **return** x == c.x; }

**bool** operator != (**const** Modulus**&** c) { **return** x != c.x; }

**friend** istream**&** operator >> (istream**&** cin, Modulus **&**c) {

            cin >> c.x; c.x = c.x % mod + mod;

**return** c.fix(), cin;

        }

**friend** ostream**&** operator << (ostream**&** cout, **const** Modulus **&**c) {

**return** cout << c.x;

        }

    };

**int** m = -1;

    vector<Modulus> Fact, Invert;

    // [T = int] O(1) | [T == ll] O(log(exp))

    Modulus Mul(Modulus c, Modulus d) {

        ll exp = d.x;

**if**(sizeof(T) == 4**UL**) **return** Modulus(c.x \* exp % mod);

        Modulus res = 0;

**for**(; exp > 0; exp >>= 1, c = c + c)

**if**(exp & 1) res = res + c;

**return** res;

    }

    // [T = int] O(log(exp)) | [T = ll] O(log(exp) \* log(mod))

    Modulus Pow(Modulus c, ll exp) {

        Modulus res = 1;

**for**(; exp > 0; exp >>= 1, c = c \* c)

**if**(exp & 1) res = res \* c;

**return** res;

    }

    // [c.x <= m] <O(m), O(1)> | <O(1), O(log(min(mod, c.x)))

    Modulus Inv(Modulus c) {

**if**(c.x <= m)

**return** Mul(Invert[c.x], Fact[c.x - 1]);

        ll a = c.x, b = mod, ax = 1, bx = 0;

**while**(b > 0) {

            ll q = a / b, r = a % b;

            a = b, b = r;

            r = ax - bx \* q;

            ax = bx, bx = r;

        }

**if**(ax < 0) ax += mod;

**return** Modulus(ax);

    }

    // O(m)

**void** BuildMOD(**int** M) {

        m = M;

        Fact = Invert = vector<Modulus>(m + 1, 0);

        Fact[0] = Invert[0] = 1;

**for**(**int** i = 1; i <= m; ++i)

            Fact[i] = Fact[i - 1] \* i;

        Invert[m] = Pow(Fact[m], mod - 2);

**for**(**int** i = m - 1; i >= 1; --i)

            Invert[i] = Invert[i + 1] \* ll(i + 1);

    }

    // <O(m), O(1))

    Modulus nCk(ll n, ll k) {

        Modulus res = 1;

**while**(n != 0 or k != 0) {

            ll rn = n % mod, rk = k % mod;

            n /= mod, k /= mod;

**if**(rn < rk) **return** 0;

            res = res \* Fact[rn] / (Fact[rk] \* Fact[rn - rk]);

        }

**return** res;

    }

} // namespace md

**using** **namespace** **md**;

// Bit Presum {{{ -----------------------------------------------------

**class** Bit\_Presum {

**public**:

**static** **constexpr** **uint32\_t** omega = CHAR\_BIT \* sizeof(**uint64\_t**);

**static** **constexpr** **uint32\_t** lg\_omega = \_\_lg(omega);

**static\_assert**(omega == 64**u**);

    Bit\_Presum(vector<**uint64\_t**> mask\_)

            : n(mask\_.size()), mask(move(mask\_)), presum(n+1) {

        build();

    }

    Bit\_Presum(**uint32\_t** bits, **bool** init\_val = 0)

            : n((bits>>lg\_omega) + 1),

              mask(n, init\_val ? ~**uint64\_t**{0} : **uint64\_t**{0}),

              presum(n+1) {

**if** (init\_val) mask.back()<<=((n<<lg\_omega) - bits);

        build();

    }

    // popcount l <= i < r

**uint32\_t** query(**uint32\_t** l, **uint32\_t** r) **const** {

**if** (\_\_builtin\_expect(r < l, **false**)) **return** 0;

**return** query(r) - query(l);

    }

    // popcount 0 <= i < x

**uint32\_t** query(**uint32\_t** x) **const** {

**uint32\_t** high = x>>lg\_omega, low = x & ((**uint64\_t**{1}<<lg\_omega) - 1);

**uint32\_t** ret = presum\_query(high);

        ret += \_\_builtin\_popcountll(mask[high]& ((**uint64\_t**{1} << low)-1));

**return** ret;

    }

**void** update\_pre\_build(**uint32\_t** x, **bool** val) {

**uint32\_t** high = x>>lg\_omega, low = x & ((1**u**<<lg\_omega) - 1);

        mask[high] = (mask[high] & ~(**uint64\_t**{1} << low)) | (**uint64\_t**{val}<<low);

    }

**void** do\_build() {

        build();

    }

**friend** ostream**&** operator<<(ostream**&**o, Bit\_Presum **const&**b) {

**for** (**auto** **const**& e : b.mask) {

            stringstream ss;

            ss << bitset<omega>(e);

**auto** s = ss.str();

            reverse(s.begin(), s.end());

            o << s << "|";

        }

        o << " : ";

**for** (**auto** **const**&e:b.presum) o << e << " ";

        o << "\n";

**return** o;

    }

**private**:

**void** presum\_build() {

**for** (**uint32\_t** x = 1; x <= n; ++x) {

            presum[x] += presum[x-1];

        }

    }

    // sum 0 <= i < x

**uint32\_t** presum\_query(**uint32\_t** x) **const** {

**return** presum[x];

    }

**void** build() {

**for** (**uint32\_t** x = 0; x < n; ++x) {

            presum[x+1] = \_\_builtin\_popcountll(mask[x]);

        }

        presum\_build();

    }

**const** **uint32\_t** n;

    vector<**uint64\_t**> mask;

    vector<**uint32\_t**> presum;

};

// }}}

**template**<**typename** T, **typename** Bit\_Ds = Bit\_Presum>

**class** WaveletMatrix {

**public**:

**static\_assert**(**is\_integral**<T>::value);

**static** **constexpr** **uint32\_t** height = CHAR\_BIT \* sizeof(T);

    WaveletMatrix(vector<T> v): n(v.size()), data(height, n) {

        build(move(v));

    }

    // count l <= i < r  s.t.  A <= val[i] < B

**uint32\_t** range\_count(**int** l, **int** r, T A, T B) **const** {

        assert(0 <= l && r <= n);

**return** count\_lower(l, r, B) - count\_lower(l, r, A);

    }

    // count l <= i < r  s.t.  A <= val[i]

**uint32\_t** range\_count\_up(**int** l, **int** r, T A) **const** {

        assert(0 <= l && r <= n);

**if** (\_\_builtin\_expect(l>r, **false**)) **return** **uint32\_t**{0};

**return** (r-l) - count\_lower(l, r, A);

    }

    // k from 0

    // range: [l, r-1]

    T k\_th(**int** l, **int** r, **int** k) **const** {

        assert(0 <= k && k < n);

**return** get\_kth(l, r, k);

    }

    // internal functions {{{

**private**:

**void** build(vector<T> v) {

        m\_index.resize(height);

        T **const** a = **numeric\_limits**<T>::min();

**for** (**int** h = height-1; h>=0;--h) {

            T **const** b = a + (T{1}<<(max(0, h-1))) - !h + (T{1}<<(max(0, h-1)));

**for** (**int** i=0;i<n;++i) {

                data[h].update\_pre\_build(i, v[i]<b);

            }

            data[h].do\_build();

**const** **int** m = stable\_partition(v.begin(), v.end(), [&b](T **const&**x) {**return** x < b;}) - v.begin();

**for** (**int** i=m;i<n;++i) {

                v[i] = v[i] - (T{1}<<(max(0, h-1))) + !h - (T{1}<<(max(0, h-1)));

            }

            m\_index[h] = m;

        }

    }

    /// count l <= i < r  s.t.  val[i] < B

**uint32\_t** count\_lower(**int** l, **int** r, T **const&**B) **const** {

        assert(0 <= l && r <= n);

**if** (\_\_builtin\_expect(r<l, **false**)) **return** 0;

**uint32\_t** ret = 0;

**int** h = height;

        T a = **numeric\_limits**<T>::min();

**while**(h > 0) {

            --h;

**bool** go\_left = B < a + (T{1}<<(max(0, h-1))) - !h + (T{1}<<(max(0, h-1)));

**const** **int** low\_l = data[h].query(l), low\_r = data[h].query(r);

**if** (go\_left) {

                l = low\_l;

                r = low\_r;

            } **else** {

                a = a + (T{1}<<(max(0, h-1))) - !h + (T{1}<<(max(0, h-1)));

                ret+= low\_r-low\_l;

                l = m\_index[h] + l-low\_l;

                r = m\_index[h] + r-low\_r;

            }

        }

**return** ret;

    }

    T get\_kth(**int** l, **int** r, **int** k) **const** {

        assert(0 <= l && r <= n);

        assert(0 <= k && k < r-l);

**int** h = height;

        T a = **numeric\_limits**<T>::min();

**while** (h > 0) {

            --h;

**const** **int** low\_l = data[h].query(l), low\_r = data[h].query(r), low\_lr = low\_r-low\_l;

**bool** go\_left = k < low\_lr;

**if** (go\_left) {

                l = low\_l;

                r = low\_r;

            } **else** {

                a+= T{1}<<h;

                k-= low\_lr;

                l = m\_index[h] + l-low\_l;

                r = m\_index[h] + r-low\_r;

            }

        }

**return** a;

    }

**const** **int** n;

    vector<**int**> m\_index;

    vector<Bit\_Ds> data;

    // }}}

};

// }}}

/\*

\* Clique -------------------------------------------------------------

 \*  Recursive algorithm for maximum clique

 \*  Takes ~1s in the worst case for n ~ 53 (random with p=0.95)

 \*  Faster on sparse graph (p <= 0.85)

 \*/

**template** <**size\_t** max\_n>

**class** Clique

{

**using** bits = bitset<max\_n>;

    bits MASK, ZERO, ans;

**const** bits \*e;

**int** N;

    // int64\_t calls;

**void** bk\_init()

    {

        ans = ZERO;

        MASK = ZERO;

        MASK.flip();

        // calls = 0;

    }

**void** bk(bits use, bits can\_start, bits can\_other)

    {

        // ++calls;

**if** (can\_start.none() && can\_other.none())

        {

**if** (use.count() > ans.count())

                ans = use;

**return**;

        }

        bits r = can\_start;

**bool** fi = 1;

**for** (**int** i = 0; i < N; ++i)

        {

**if** (r[i])

            {

**if** (fi)

                {

                    fi = 0;

                    r &= e[i] ^ MASK;

                }

                use[i] = 1;

                bk(use, can\_start & e[i], can\_other & e[i]);

                use[i] = 0;

                can\_start[i] = 0;

                can\_other[i] = 1;

            }

        }

    }

**static** Clique **&**get()

    {

**static** Clique c;

**return** c;

    }

**public**:

**static** bits find\_clique(bits **const** \*g, **const** **int** **&**n)

    {

        Clique &c = get();

        c.e = g;

        c.N = n;

        c.bk\_init();

        bits me;

        c.bk(me, c.MASK, c.ZERO);

        // cerr << "Calls: " << c.calls << "\n";

        // c.calls = 0;

**return** c.ans;

    }

**static** bits find\_clique(vector<bits> **const** **&**g)

    {

**return** find\_clique(g.data(), g.size());

    }

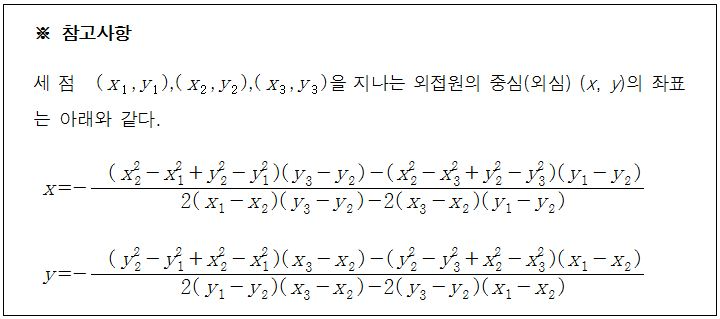
**static** bits find\_clique(array<bits, max\_n> **const** **&**g, **const** **int** **&**n)

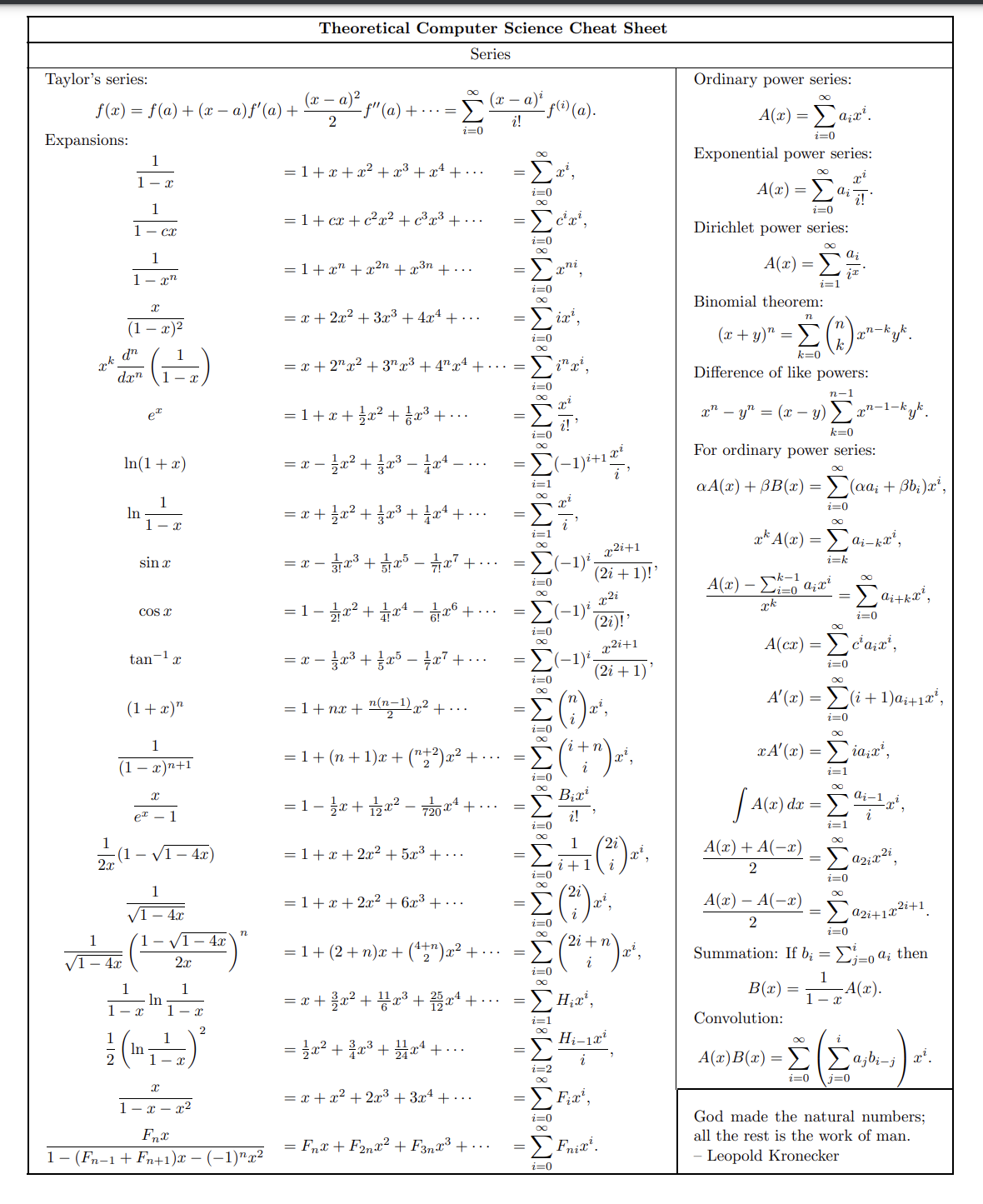
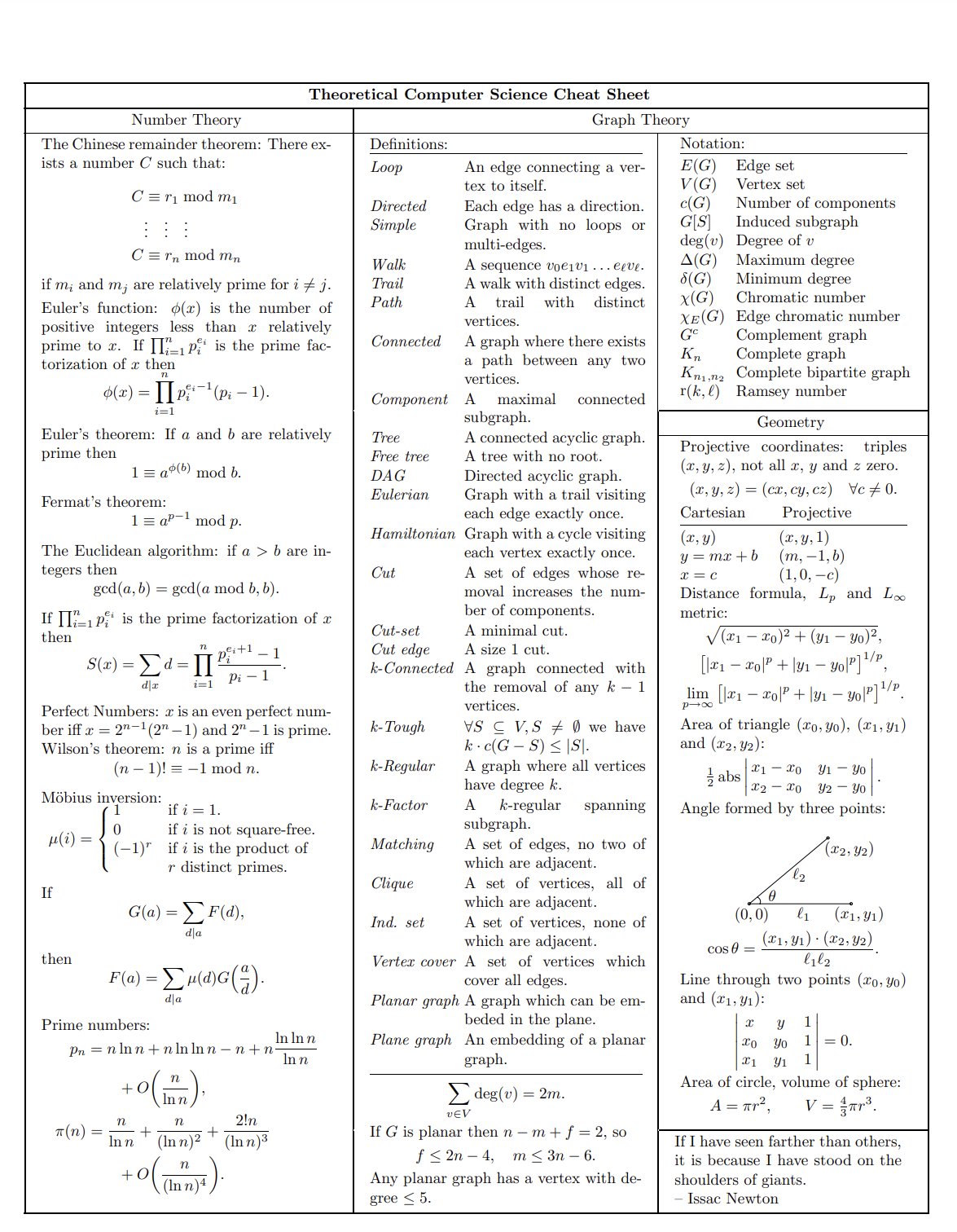
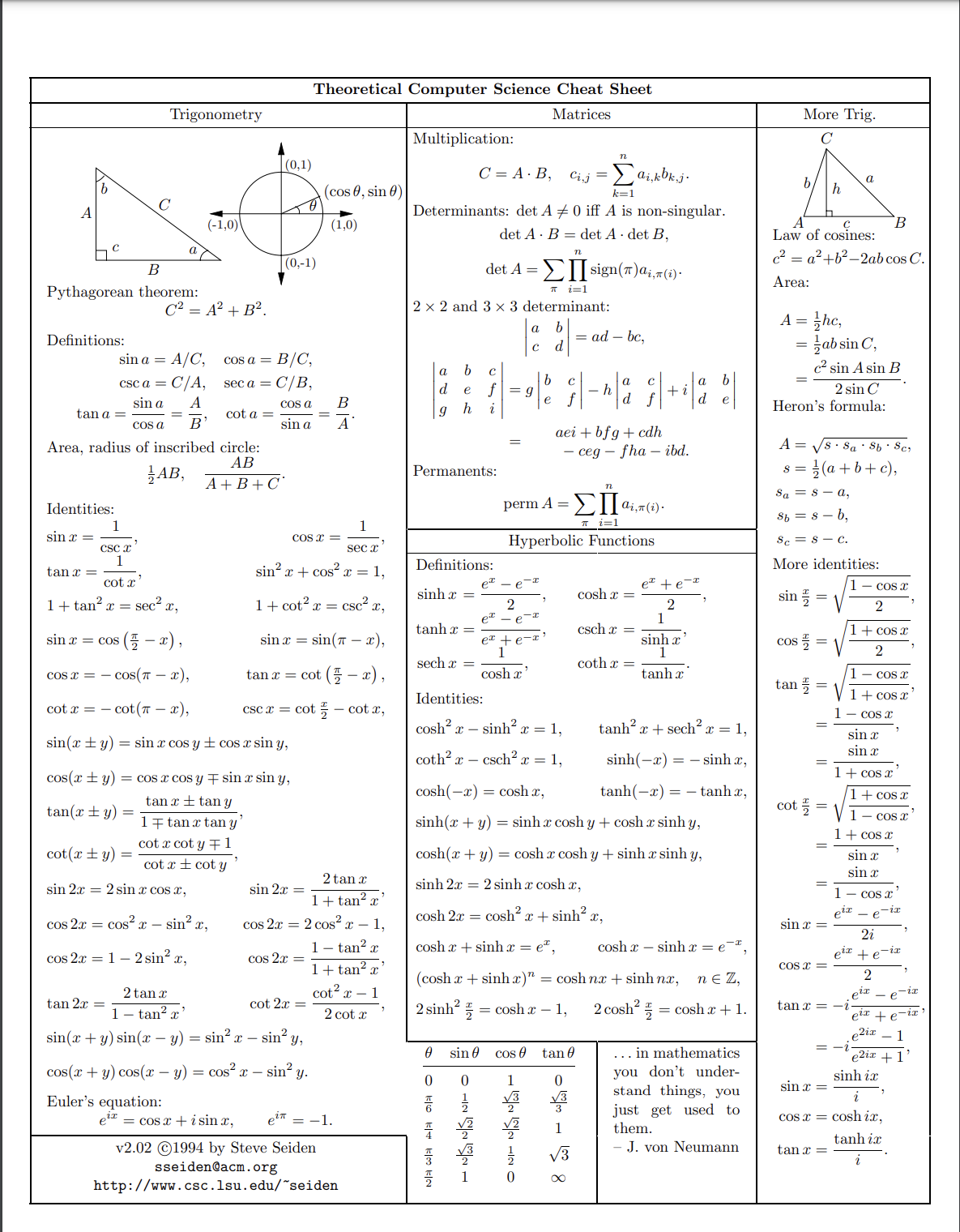
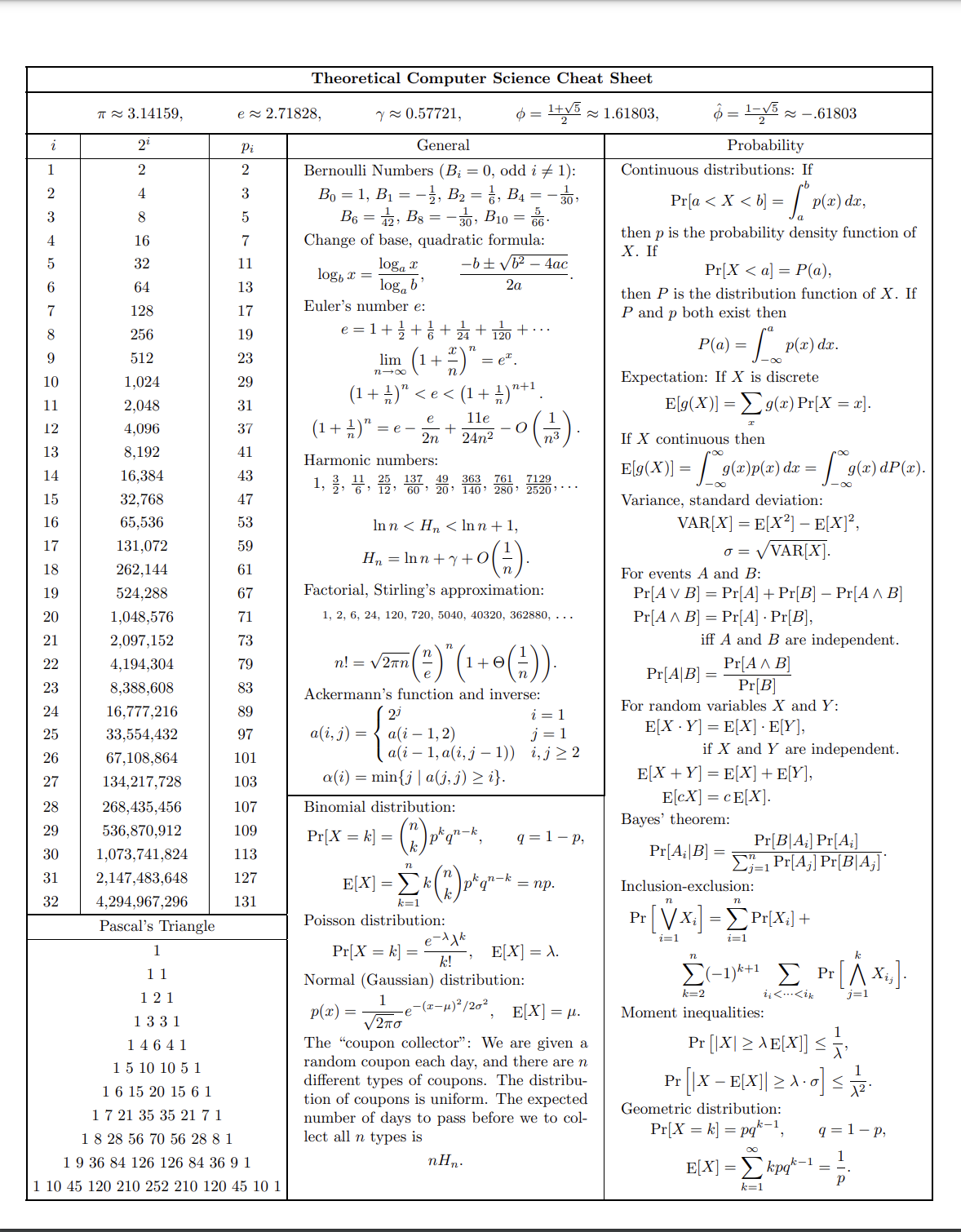
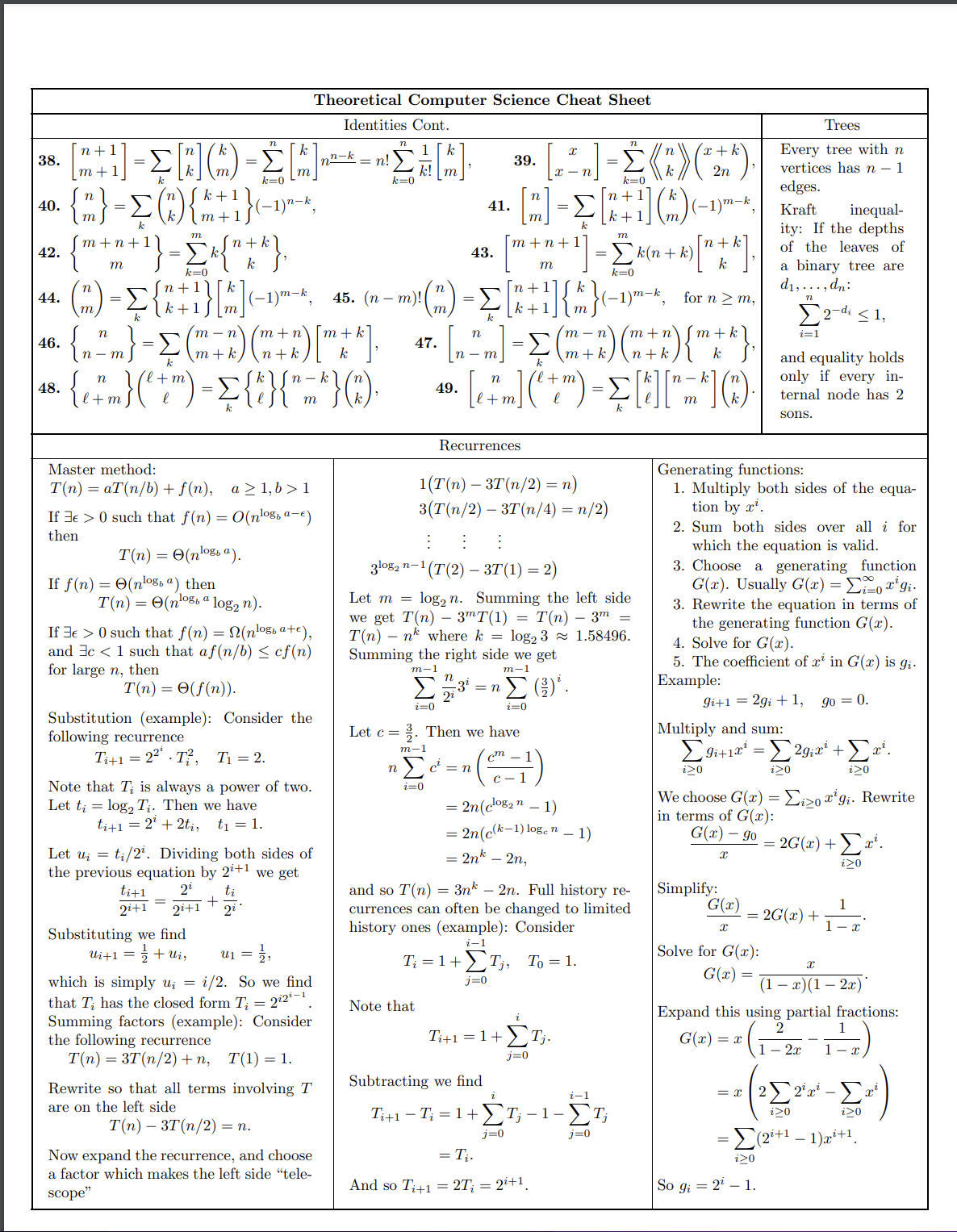
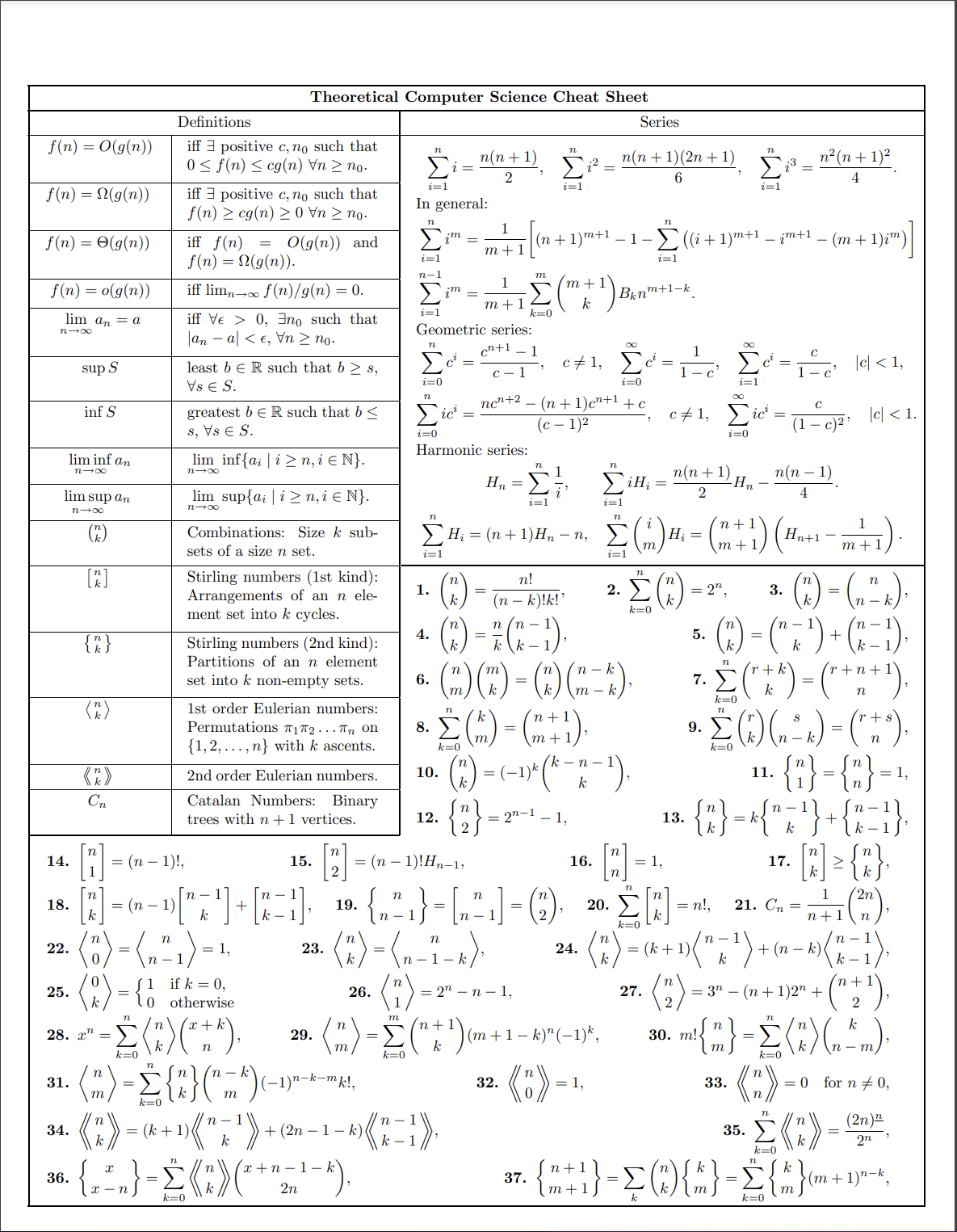
    {

**return** find\_clique(g.data(), n);

    }

};





***> Mục Lục:***

Page 1: Aho, Blossom.

Page 2: debug file.

Page 3: Extend Euclide, Fenwick Tree.  
Page 4: Lichao Tree, Line Container.

Page 5: Mahattan MST, Rabin Miller.

Page 6: Strings, Treap.

Page 7: Two-Sat.

Page 8: berlekampMassey.

Page 9: MinCost-MaxFlow, MaxFlow.

Page 10: Segment Tree.

Page 11: Segment Tree beats.

Page 12: Mod int.

Page 13: WaveletMatrix.

Page 15: Clique.

Page 16~19: Math.

Page 20: Mục Lục.