Karlsruhe Institute of Technology

Team Contest Reference

2014 ACM ICPC Northwestern European Regional Contest

Linköping, November 29-30, 2014

Team hacKIT

1 Stringology

1.1 Z Algorithm

```
/* calculate the $z array for string $s of length $n in O(n) time.
 * z[i] := the longest common prefix of s[0..n-1] and s[i..n-1].
 * For pattern matching, make a string P$S and output positions with z[i]==|P|
 * For pattern matching, there's no need to store (but to calculate) z[i] for i>|P|. */
void calc_Z(const char *s, int n, int *z) {
    int l = 0, r = 0, p, q;
    if(n > 0) z[0] = n;
    for (int i = 1; i < n; ++i) {
        if (i <= r && z[i - l] < r - i + 1) {
            z[i] = z[i - l];
        } else {
            if (i > r) p = 0, q = i;
            else p = r - i + 1, q = r + 1;
            while (q < n && s[p] == s[q]) ++p, ++q;
            z[i] = q - i, l = i, r = q - 1;
        }
    }
}
```

1.2 Rolling hash

```
int q = 311;
struct Hasher { // use two of those, with different mod (e.g. 1e9+7 and 1e9+9)
  string s;
  int mod;
  vector<int> power, pref;
  Hasher(const string& s, int mod) : s(s), mod(mod) {
    power.pb(1);
    rep(i,1,s.size()) power.pb((ll)power.back() * q % mod);
    pref.pb(0);
    rep(i,0,s.size()) pref.pb(((ll)pref.back() * q % mod + s[i]) % mod);
  }
  int hash(int l, int r) { // compute hash(s[l..r]) with r inclusive
    return (pref[r+1] - (ll)power[r-l+1] * pref[l] % mod + mod) % mod;
  }
};
```

1.3 Suffix Array - LCP Based

```
const int maxn = 200010, maxlg = 18; // maxlg = ceil(log_2(maxn))
struct SA {
  pair<pii, int> L[maxn]; // O(n * log n) space
  int P[maxlg+1][maxn], n, stp, cnt, sa[maxn];
  SA(const string& s) : n(s.size()) { // O(n * log n)
    rep(i,0,n) P[0][i] = s[i];
    sa[0] = 0; // in case n == 1
    for (stp = 1, cnt = 1; cnt < n; stp++, cnt <<= 1) {
      rep(i,0,n) L[i] = mk(mk(P[stp-1][i], i + cnt < n ? P[stp-1][i+cnt] : -1), i);
      std::sort(L, L + n);
      rep(i,0,n)
        P[stp][L[i].snd] = i>0 && L[i].fst == L[i-1].fst ?  P[stp][L[i-1].snd] : i;
    }
    rep(i,0,n) sa[i] = L[i].snd;
  }
  int lcp(int x, int y) { // time log(n); x, y = indices into string, not SA
    int k, ret = 0;
    if (x == y) return n - x;
    for (k = stp - 1; k >= 0 && x < n && y < n; k --)
      if (P[k][x] == P[k][y])
        x += 1<<k, y += 1<<k, ret += 1<<k;
    return ret;
  }
};
```

1.4 Suffix automaton

```
struct SuffixAutomaton { // can be used for LCS and others
    struct State {
        int depth, id;
        State *go[128], *suffix;
    } *root = new State {0}, *sink = root;
```

```
    void append(const string& str, int offset=0) { // O(|str|)
        for (int i = 0; i < str.size(); ++i) {
            int a = str[i];
            State *cur = sink, *sufState;
            sink = new State { sink->depth + 1, offset + i, {0}, 0 };
            while (cur && !cur->go[a]) {
                cur->go[a] = sink;
                cur = cur->suffix;
            }
            if (!cur) sufState = root;
            else {
                State *q = cur->go[a];
                if (q->depth == cur->depth + 1)
                    sufState = q;
                else {
                    State *r = new State(*q);
                    r->depth = cur->depth + 1;
                    q->suffix = sufState = r;
                    while (cur && cur->go[a] == q) {
                        cur->go[a] = r;
                        cur = cur->suffix;
                    }
                }
            }
            sink->suffix = sufState;
        }
    }
    int walk(const string& str) { // O(|str|) returns LCS with automaton string
        int tmp = 0;
        State *cur = root;
        int ans = 0;
        for (int i = 0; i < str.size(); ++i) {
            int a = str[i];
            if (cur->go[a]) {
                tmp++;
                cur = cur->go[a];
            } else {
                while (cur && !cur->go[a])
                    cur = cur->suffix;
                if (!cur) {
                    cur = root;
                    tmp = 0;
                } else {
                    tmp = cur->depth + 1;
                    cur = cur->go[a];
                }
            }
            ans = max(ans, tmp);  // i - tmp + 1 is start of match
        }
        return ans;
    }
};
```

1.5 Aho-Corasick automaton

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

vertex *create() {
  vertex *root = new vertex();
  root->link = root;
  return root;
}

void add_string (vertex *v, const vector<int>& s) {
  for (int a: s) {
    if (!v->next[a]) {
      vertex *w = new vertex();
      w->p = v;
      w->pch = a;
      v->next[a] = w;
    }
    v = v->next[a];
  }
  v->leaf = 1;
```

```
}

vertex* go(vertex* v, int c);

vertex* get_link(vertex *v) {
  if (!v->link)
    v->link = v->p->p ? go(get_link(v->p), v->pch) : v->p;
  return v->link;
}

vertex* go(vertex* v, int c) {
  if (!v->go[c]) {
    if (v->next[c])
      v->go[c] = v->next[c];
    else
      v->go[c] = v->p ? go(get_link(v), c) : v;
  }
  return v->go[c];
}

bool is_accepting(vertex *v) {
  if (v->is_acceping == -1)
    v->is_accepting = v->leaf || is_accepting(get_link(v));
  return v->is_accepting;
}
```

2 Arithmetik und Algebra

2.1 Lineare Gleichungssysteme (LGS) und Determinanten

2.1.1 Gauß-Algorithmus

```
class R {
    BigInteger n, d;
    R(BigInteger n_, BigInteger d_) {
        n = n_; d = d_;
        BigInteger g = n.gcd(d);
        n.divide(g); d.divide(g);
    }
    R add(R x) {
        return new R(n.multiply(x.d).add(d.multiply(x.n)), d.multiply(x.d));
    }
    R negate() { return new R(n.negate(), d); }
    R subtract(R x) { return add(x.negate()); }
    R multiply(R y) {
        return new R(n.multiply(x.n), d.multiply(x.d));
    }
    R invert() { return new R(d, n); }
    R divide(R y) { return multiply(y.invert()); }
    boolean zero() { return d.equals(BigInteger.ZERO); }
}

int maxm = 13, maxn = 4;
R[][] M = new R[maxm][maxn]; // the LGS matrix
R[] B = new R[maxm];          // the right side

void gauss(int m, int n) { // reduces M to Gaussian normal form
    int row = 0;
    for (int col = 0; col < n; ++col) { // eliminate downwards
        int pivot=row;
        while(pivot<m&&M[pivot][col].zero())pivot++;
        if (pivot == m || M[pivot][col].zero()) continue;
        if (row!=pivot) {
            for (int j = 0; j < n; ++j) {
                R tmp = M[row][j];
                M[row][j] = M[pivot][j];
                M[pivot][j] = tmp;
            }
            R tmp = B[row];
            B[row] = B[pivot];
            B[pivot] = tmp;
        }
        // for double, normalize pivot row here (divide it by pivot value)
        for (int j = row+1; j < m; ++j) {
            if (M[j][col].zero()) continue;
            R a = M[row][col], b = M[j][col];
            for(int k=0; k<n; ++k)
                M[j][k] = M[j][k].multiply(a).subtract(M[row][k].multiply(b));
            B[j] = B[j].multiply(a).subtract(B[row].multiply(b));
        }
```

```
        row++;
    }
    for (int col = 0; col < n; ++col) { // eliminate upwards
        for (row = m-1; row >= 0; --row) {
            if (M[row][col].zero()) continue;
            boolean valid=true;
            for (int j = 0; j < col; ++j)
                if (!M[row][j].zero()) { valid=false; break; }
            if (!valid) continue;
            for (int i = 0; i < row; ++i) {
                R a = M[row][col], b = M[i][col];
                for (int k =0; k<n; ++k)
                    M[i][k] = M[i][k].multiply(a).subtract(M[row][k].multiply(b));
                B[i] = B[i].multiply(a).subtract(B[row].multiply(b));
            }
            break;
        }
    }
}
```

2.1.2 LR-Zerlegung, Determinanten

```
const int MAX = 42;
void lr(double a[MAX][MAX], int n) {
    for (int i = 0; i < n; ++i) {
        for (int k = 0; k < i; ++k) a[i][i] -= a[i][k] * a[k][i];
        for (int j = i + 1; j < n; ++j) {
            for (int k = 0; k < i; ++k) a[j][i] -= a[j][k] * a[k][i];
            a[j][i] /= a[i][i];
            for (int k = 0; k < i; ++k) a[i][j] -= a[i][k] * a[k][j];
        }
    }
}
double det(double a[MAX][MAX], int n) {
    lr(a, n);
    double d = 1;
    for (int i = 0; i < n; ++i) d *= a[i][i];
    return d;
}
void solve(double a[MAX][MAX], double *b, int n) {
    for (int i = 1; i < n; ++i)
        for (int j = 0; j < i; ++j) b[i] -= a[i][j] * b[j];
    for (int i = n - 1; i >= 0; --i) {
        for (int j = i + 1; j < n; ++j) b[i] -= a[i][j] * b[j];
        b[i] /= a[i][i];
    }
}
```

2.2 Numerical Integration (Adaptive Simpson's rule)

```
double f(double x) { return exp(-x*x); }
const double eps=1e-12;

double simps(double a, double b) { // for ~4x less f() calls, pass fa,fm,fb around
  return (f(a) + 4*f((a+b)/2) + f(b))*(b-a)/6;
}
double integrate(double a, double b) {
  double m = (a+b)/2;
  double l = simps(a,m),r = simps(m,b),tot=simps(a,b);
  if (fabs(l+r-tot) < eps) return tot;
  return integrate(a,m) + integrate(m,b);
}
```

2.3 FFT

```
typedef double D; // or long double?
typedef complex<D> cplx; // use own implementation for 2x speedup
const D pi = acos(-1); // or -1.L for long double

// input should have size 2^k
vector<cplx> fft(const vector<cplx>& a, bool inv=0) {
    int logn=1, n=a.size();
    vector<cplx> A(n);
    while((1<<logn)<n) logn++;
    rep(i,0,n) {
        int j=0; // precompute j = rev(i) if FFT is used more than once
```

```
        rep(k,0,logn) j = (j<<1) | ((i>>k)&1);
        A[j] = a[i]; }
    for(int s=2; s<=n; s<<=1) {
        D ang = 2 * pi / s * (inv ? -1 : 1);
        cplx ws(cos(ang), sin(ang));
        for(int j=0; j<n; j+=s) {
            cplx w=1;
            rep(k,0,s/2) {
                cplx u = A[j+k], t = A[j+s/2+k];
                A[j+k] = u + w*t;
                A[j+s/2+k] = u - w*t;
                if(inv) A[j+k] /= 2, A[j+s/2+k] /= 2;
                w *= ws; } } }
    return A;
}
vector<cplx> a = {0,0,0,0,1,2,3,4}, b = {0,0,0,0,2,3,0,1}; // polynomials
a = fft(a); b = fft(b);
rep(i,0,a.size()) a[i] *= b[i]; // convult spectrum
a = fft(a,1); // ifft, a = a * b
```

3 Zahlentheorie

3.1 Miscellaneous

```
ll multiply_mod(ll a, ll b, ll mod) {
  if (b == 0) return 0;
  if (b & 1) return ((ull)multiply_mod(a, b-1, mod) + a) % mod;
  return multiply_mod(((ull)a + a) % mod, b/2, mod);
}

ll powmod(ll a, ll n, ll mod) {
  if (n == 0) return 1 % mod;
  if (n & 1) return multiply_mod(powmod(a, n-1, mod), a, mod);
  return powmod(multiply_mod(a, a, mod), n/2, mod);
}

// simple modinv, returns 0 if inverse doesn't exist
ll modinv(ll a, ll m) {
  return a < 2 ? a : ((1 - m * 1ll * modinv(m % a, a)) / a % m + m) % m;
}
ll modinv_prime(ll a, ll p) { return powmod(a, p-2, p); }

ll extended_gcd(ll a, ll b, ll& lastx, ll& lasty) {
  ll x, y, q, tmp;
  x = 0; lastx = 1;
  y = 1; lasty = 0;
  while (b != 0) {
    q = a / b;
    tmp = b;
    b = a % b;
    a = tmp;
    tmp = x; x = lastx - q*x; lastx = tmp;
    tmp = y; y = lasty - q*y; lasty = tmp;
  }
  return a;
}

// solve the linear equation a x == b (mod n)
// returns the number of solutions up to congruence (can be 0)
//    sol: the minimal positive solution
//    dis: the distance between solutions
ll linear_mod(ll a, ll b, ll n, ll &sol, ll &dis) {
  a = (a % n + n) % n, b = (b % n + n) % n;
  ll d, x, y;
  d = extended_gcd(a, n, x, y);
  if (b % d)
    return 0;
  x = (x % n + n) % n;
  x = b / d * x % n;
  dis = n / d;
  sol = x % dis;
  return d;
}

bool rabin(ll n) {
  // bases chosen to work for all n < 2^64, see https://oeis.org/A014233
  set<int> p { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };
  if (n <= 37) return p.count(n);
  ll s = 0, t = n - 1;
  while (~t & 1)
```

```
    t >>= 1, ++s;
  for (int x: p) {
    ll pt = powmod(x, t, n);
    if (pt == 1) continue;
    bool ok = 0;
    for (int j = 0; j < s && !ok; ++j) {
      if (pt == n - 1) ok = 1;
      pt = multiply_mod(pt, pt, n);
    }
    if (!ok) return 0;
  }
  return 1;
}

ll rho(ll n) {
  if (~n & 1) return 2;
  ll c = rand() % n, x = rand() % n, y = x, d = 1;
  while (d == 1) {
    x = (multiply_mod(x, x, n) + c) % n;
    y = (multiply_mod(y, y, n) + c) % n;
    y = (multiply_mod(y, y, n) + c) % n;
    d = __gcd(abs(x - y), n);
  }
  return d == n ? rho(n) : d;
}

void factor(ll n, map<ll, int> &facts) {
  if (n == 1) return;
  if (rabin(n)) {
    facts[n]++;
    return;
  }
  ll f = rho(n);
  factor(n/f, facts);
  factor(f, facts);
}

// use inclusion-exclusion to get the number of integers <= n
// that are not divisable by any of the given primes.
// This essentially enumerates all the subsequences and adds or subtracts
// their product, depending on the current parity value.
ll count_coprime_rec(int primes[], int len, ll n, int i, ll prod, bool parity) {
  if (i >= len || prod * primes[i] > n) return 0;
  return (parity ? 1 : (-1)) * (n / (prod*primes[i]))
        + count_coprime_rec(primes, len, n, i + 1, prod, parity)
        + count_coprime_rec(primes, len, n, i + 1, prod * primes[i], !parity);
}
// use cnt(B) - cnt(A-1) to get matching integers in range [A..B]
ll count_coprime(int primes[], int len, ll n) {
  if (n <= 1) return max(0LL, n);
  return n - count_coprime_rec(primes, len, n, 0, 1, true);
}

// find x.  a[i] x = b[i] (mod m[i])  0 <= i < n. m[i] need not be coprime
bool crt(int n, ll *a, ll *b, ll *m, ll &sol, ll &mod) {
  ll A = 1, B = 0, ta, tm, tsol, tdis;
  for (int i = 0; i < n; ++i) {
    if (!linear_mod(a[i], b[i], m[i], tsol, tdis)) return 0;
    ta = tsol, tm = tdis;
    if (!linear_mod(A, ta - B, tm, tsol, tdis)) return 0;
    B = A * tsol + B;
    A = A * tdis;
  }
  sol = B, mod = A;
  return 1;
}

// get number of permutations {P_1, ..., P_n} of size n,
// where no number is at its original position (that is, P_i != i for all i)
// also called subfactorial !n
ll get_derangement_mod_m(ll n, ll m) {
  vector<ll> res(m * 2);
  ll d = 1 % m, p = 1;
  res[0] = d;
  for (int i = 1; i <= min(n, 2 * m - 1); ++i) {
    p *= -1;
    d = (1LL * i * d + p + m) % m;
    res[i] = d;
    if (i == n) return d;
  }
  // it turns out that !n mod m == !(n mod 2m) mod m
  return res[n % (2 * m)];
```

```
}

// compute totient function for integers <= n
vector<int> compute_phi(int n) {
  vector<int> phi(n + 1, 0);
  for (int i = 1; i <= n; ++i) {
    phi[i] += i;
    for (int j = 2 * i; j <= n; j += i) {
      phi[j] -= phi[i];
    }
  }
  return phi;
}

// checks if g is primitive root mod p. Generate random g's to find primitive root.
bool is_primitive(ll g, ll p) {
  map<ll, int> facs;
  factor(p - 1, facs);
  for (auto& f : facs)
    if (1 == powmod(g, (p-1)/f.first, p))
      return 0;
  return 1;
}

ll dlog(ll g, ll b, ll p) { // find x such that g^x = b (mod p)
  ll m = (ll)(ceil(sqrt(p-1))+0.5); // better use binary search here...
  unordered_map<ll,ll> powers; // should compute this only once per g
  rep(j,0,m) powers[powmod(g, j, p)] = j;
  ll gm = powmod(g, -m + 2*(p-1), p);
  rep(i,0,m) {
    if (powers.count(b)) return i*m + powers[b];
    b = b * gm % p;
  }
  assert(0); return -1;
}

// compute p(n,k), the number of possibilities to write n as a sum of
// k non-zero integers
ll count_partitions(int n, int k) {
  if (n==k) return 1;
  if (n<k || k==0) return 0;
  vector<ll> p(n + 1);
  for (int i = 1; i <= n; ++i) p[i] = 1;
  for (int l = 2; l <= k; ++l)
    for (int m = l+1; m <= n-l+1; ++m)
      p[m] = p[m] + p[m-l];
  return p[n-k+1];
}
```

3.2 Binomial Coefficient modulo M

```
// calculate (product_{i=1,i%p!=0}^n i) % p^e. cnt is the exponent of p in n!
// Time: p^e + log(p, n)
int get_part_of_fac_n_mod_pe(int n, int p, int mod, int *upto, int &cnt) {
    if (n < p) { cnt = 0; return upto[n];}
    else {
        int res = powmod(upto[mod], n / mod, mod);
        res = (ll) res * upto[n % mod] % mod;
        res = (ll) res * get_part_of_fac_n_mod_pe(n / p, p, mod, upto, cnt) % mod;
        cnt += n / p;
        return res;
    }
}
//C(n,k) % p^e.  Use Chinese Remainder Theorem to get C(n,k)%m
int get_n_choose_k_mod_pe(int n, int k, int p, int mod) {
    static int upto[maxm + 1];
    upto[0] = 1 % mod;
    for (int i = 1; i <= mod; ++i)
        upto[i] = i % p ? (ll) upto[i - 1] * i % mod : upto[i - 1];
    int cnt1, cnt2, cnt3;
    int a = get_part_of_fac_n_mod_pe(n, p, mod, upto, cnt1);
    int b = get_part_of_fac_n_mod_pe(k, p, mod, upto, cnt2);
    int c = get_part_of_fac_n_mod_pe(n - k, p, mod, upto, cnt3);
    int res = (ll) a * modinv(b, mod) % mod * modinv(c, mod) % mod * powmod(p, cnt1 - cnt2 - cnt3, mod) % mod;
    return res;
}
// Lucas's Theorem (p prime, m_i,n_i base p repr. of m, n): binom(m,n)==procduct(binom(m_i,n_i)) (mod p)
```

4 Graphen

4.1 Maximum Bipartite Matching

```
// run time: O(n * min(ans^2, |E|)), where n is the size of the left side
vector<int> madj[1001]; // adjacency list
int pairs[1001]; // for every node, stores the matching node on the other side or -1
bool vis[1001];
bool dfs(int i) {
    if (vis[i]) return 0;
    vis[i] = 1;
    foreach(it,madj[i]) {
        if (pairs[*it] < 0 || dfs(pairs[*it])) {
            pairs[*it] = i, pairs[i] = *it;
            return 1;
        }
    }
    return 0;
}
int kuhn(int n) { // n = nodes on left side (numbered 0..n-1)
    clr(pairs,-1); // to accelerate, just initialize with a greedy matching
    int ans = 0;
    rep(i,0,n) {
        clr(vis,0);
        ans += dfs(i);
    }
    return ans;
}
```

4.2 Maximaler Fluss (FF + Capacity Scaling)

```
// FF with cap scaling, O(m^2 log C)
const int MAXN = 190000, MAXC = 1<<29;
struct edge { int dest, capacity, rev; };
vector<edge> adj[MAXN];
int vis[MAXN], target, iter, cap;

void addedge(int x, int y, int c) {
  adj[x].push_back(edge {y, c, (int)adj[y].size()});
  adj[y].push_back(edge {x, 0, (int)adj[x].size() - 1});
}

bool dfs(int x) {
  if (x == target) return 1;
  if (vis[x] == iter) return 0;
  vis[x] = iter;
  for (edge& e: adj[x])
    if (e.capacity >= cap && dfs(e.dest)) {
      e.capacity -= cap;
      adj[e.dest][e.rev].capacity += cap;
      return 1;
    }
  return 0;
}

int maxflow(int S, int T) {
  cap = MAXC, target = T;
  int flow = 0;
  while(cap) {
    while(++iter, dfs(S))
      flow += cap;
    cap /= 2;
  }
  return flow;
}
```

4.3 Min-Cost-Max-Flow

```
typedef long long Captype;       // set capacity type (long long or int)
// for Valtype double, replace clr(dis,0x7f) and use epsilon for distance comparison
typedef long long Valtype;       // set type of edge weight (long long or int)
static const Captype flowlimit = 1LL<<60;     // should be bigger than maxflow
struct MinCostFlow {             //XXX Usage: class should be created by new.
static const int maxn = 450;                 // number of nodes, should be bigger than n
static const int maxm = 5000;               // number of edges
struct edge {
    int node,next; Captype flow; Valtype value;
```

```
}   edges[maxm<<1];
int graph[maxn],queue[maxn],pre[maxn],con[maxn],n,m,source,target,top;
bool inq[maxn];
Captype maxflow;
Valtype mincost,dis[maxn];
MinCostFlow() { memset(graph,-1,sizeof(graph)); top = 0; }
inline int inverse(int x) {return 1+((x>>1)<<2)-x; }
inline int addedge(int u,int v,Captype c, Valtype w) { // add a directed edge
    edges[top].value = w; edges[top].flow = c; edges[top].node = v;
    edges[top].next = graph[u]; graph[u] = top++;
    edges[top].value = -w; edges[top].flow = 0; edges[top].node = u;
    edges[top].next = graph[v]; graph[v] = top++;
    return top-2;
}
bool SPFA() { // Bellmanford, also works with negative edge weight.
    int point, nod, now, head = 0, tail = 1;
    memset(pre,-1,sizeof(pre));
    memset(inq,0,sizeof(inq));
    memset(dis,0x7f,sizeof(dis));
    dis[source] = 0; queue[0] = source; pre[source] = source; inq[source] = true;
    while (head!=tail) {
        now = queue[head++]; point = graph[now]; inq[now] = false; head %= maxn;
        while (point != -1) {
            nod = edges[point].node;
            if (edges[point].flow>0 && dis[nod]>dis[now]+edges[point].value) {
                dis[nod] = dis[now] + edges[point].value;
                pre[nod] = now;
                con[nod] = point;
                if (!inq[nod]) {
                    inq[nod] = true;
                    queue[tail++] = nod;
                    tail %= maxn;
                }
            }
            point = edges[point].next;
        }
    }
    return pre[target]!=-1; //&& dis[target]<=0; // for min-cost rather than max-flow
}
void extend()
{
    Captype w = flowlimit;
    for (int u = target; pre[u]!=u; u = pre[u])
        w = min(w, edges[con[u]].flow);
    maxflow += w;
    mincost += dis[target]*w;
    for (int u = target; pre[u]!=u; u = pre[u]) {
        edges[con[u]].flow-=w;
        edges[inverse(con[u])].flow+=w;
    }
}
void mincostflow() {
    maxflow = 0; mincost = 0;
    while (SPFA()) extend();
}};
```

4.4 Value of Maximum Matching

```
const int N=200, MOD=1000000007, I=10;
int n, adj[N][N], a[N][N];
int rank() {
    int r = 0;
    rep(j,0,n) {
        int k = r;
        while (k < n && !a[k][j]) ++k;
        if (k == n) continue;
        swap(a[r], a[k]);
        int inv = powmod(a[r][j], MOD - 2);
        rep(i,j,n)
            a[r][i] = 1LL * a[r][i] * inv % MOD;
        rep(u,r+1,n) rep(v,j,n)
            a[u][v] = (a[u][v] - 1LL * a[r][v] * a[u][j] % MOD + MOD) % MOD;
        ++r;
    }
    return r;
}
// failure probability = (n / MOD)^I
int max_matching() {
    int ans = 0;
    rep(_,0,I) {
```

```
        rep(i,0,n) rep(j,0,i)
            if (adj[i][j]) {
                a[i][j] = rand() % (MOD - 1) + 1;
                a[j][i] = MOD - a[i][j];
            }
        ans = max(ans, rank()/2);
    }
    return ans;
}
```

4.5 SCC + 2-SAT

```
const int maxn = 10010; // 2-sat: maxn = 2*maxvars
vector<int> adj[maxn], radj[maxn];
bool vis[maxn];
int col, color[maxn];
vector<int> bycol[maxn];
vector<int> st;

void init() { rep(i,0,maxn) adj[i].clear(), radj[i].clear(); }
void dfs(int u, vector<int> adj[]) {
  if (vis[u]) return;
  vis[u] = 1;
  foreach(it,adj[u]) dfs(*it, adj);
  if (col) {
    color[u] = col;
    bycol[col].pb(u);
  } else st.pb(u);
}
// this computes SCCs, outputs them in bycol, in topological order
void kosaraju(int n) { // n = number of nodes
  st.clear();
  clr(vis,0);
  col=0;
  rep(i,0,n) dfs(i,adj);
  clr(vis,0);
  clr(color,0);
  while(!st.empty()) {
    bycol[++col].clear();
    int x = st.back(); st.pop_back();
    if(color[x]) continue;
    dfs(x, radj);
  }
}
// 2-SAT
int assign[maxn]; // for 2-sat only
int var(int x) { return x<<1; }
bool solvable(int vars) {
  kosaraju(2*vars);
  rep(i,0,vars) if (color[var(i)] == color[1^var(i)]) return 0;
  return 1;
}
void assign_vars() {
  clr(assign,0);
  rep(c,1,col+1) {
    foreach(it,bycol[c]) {
      int v = *it >> 1;
      bool neg = *it&1;
      if (assign[v]) continue;
      assign[v] = neg?1:-1;
    }
  }
}
void add_impl(int v1, int v2) { adj[v1].push_back(v2); radj[v2].push_back(v1); }
void add_equiv(int v1, int v2) { add_impl(v1, v2); add_impl(v2, v1); }
void add_or(int v1, int v2) { add_impl(1^v1, v2); add_impl(1^v2, v1); }
void add_xor(int v1, int v2) { add_or(v1, v2); add_or(1^v1, 1^v2); }
void add_true(int v1) { add_impl(1^v1, v1); }
void add_and(int v1, int v2) { add_true(v1); add_true(v2); }

int parse(int i) {
  if (i>0) return var(i-1);
  else return 1^var(-i-1);
}
int main() {
  int n, m; cin >> n >> m; // m = number of clauses to follow
  while (m--) {
    string op; int x, y; cin >> op >> x >> y;
    x = parse(x);
    y = parse(y);
```

```
    if (op == "or") add_or(x, y);
    if (op == "and") add_and(x, y);
    if (op == "xor") add_xor(x, y);
    if (op == "imp") add_impl(x, y);
    if (op == "equiv") add_equiv(x, y);
  }
  if (!solvable(n)) {
    cout << "Impossible" << endl; return 0;
  }
  assign_vars();
  rep(i,0,n) cout << ((assign[i]>0)?(i+1):-i-1) << endl;
}
```

5 Geometrie

5.1 Verschiedenes

```
using D=long double;
using P=complex<D>;
using L=vector<P>;
using G=vector<P>;
const D eps=1e-12, inf=1e15, pi=acos(-1), e=exp(1.);

D sq(D x) { return x*x; }
D rem(D x, D y) { return fmod(fmod(x,y)+y,y); }
D rtod(D rad) { return rad*180/pi; }
D dtor(D deg) { return deg*pi/180; }
int sgn(D x) { return (x > eps) - (x < -eps); }
// when doing printf("%.Xf", x), fix '-0' output to '0'.
D fixzero(D x, int d) { return (x>0 || x<=-5/pow(10,d+1)) ? x:0; }

namespace std {
  bool operator<(const P& a, const P& b) {
    return mk(real(a), imag(a)) < mk(real(b), imag(b));
  }
}

D cross(P a, P b)    { return imag(conj(a) * b); }
D cross(P a, P b, P c) { return cross(b-a, c-a); }
D dot(P a, P b)      { return real(conj(a) * b); }
P scale(P a, D len)  { return a * (len/abs(a)); }
P rotate(P p, D ang) { return p * polar(D(1), ang); }
D angle(P a, P b)    { return arg(b) - arg(a); }

int ccw(P a, P b, P c) {
  b -= a; c -= a;
  if (cross(b, c) > eps)  return +1;  // counter clockwise
  if (cross(b, c) < -eps) return -1;  // clockwise
  if (dot(b, c) < 0)      return +2;  // c--a--b on line
  if (norm(b) < norm(c))  return -2;  // a--b--c on line
  return 0;
}

G dummy;
L line(P a, P b) {
  L res; res.pb(a); res.pb(b); return res;
}
P dir(const L& l) { return l[1]-l[0]; }

D project(P e, P x) { return dot(e,x) / norm(e); }
P pedal(const L& l, P p) { return l[1] + dir(l) * project(dir(l), p-l[1]); }
int intersectLL(const L &l, const L &m) {
  if (abs(cross(l[1]-l[0], m[1]-m[0])) > eps) return 1;  // non-parallel
  if (abs(cross(l[1]-l[0], m[0]-l[0])) < eps) return -1; // same line
  return 0;
}
bool intersectLS(const L& l, const L& s) {
  return cross(dir(l), s[0]-l[0])*       // s[0] is left of l
         cross(dir(l), s[1]-l[0]) < eps; // s[1] is right of l
}
bool intersectLP(const L& l, const P& p) {
  return abs(cross(l[1]-p, l[0]-p)) < eps;
}
bool intersectSS(const L& s, const L& t) {
  return sgn(ccw(s[0],s[1],t[0]) * ccw(s[0],s[1],t[1])) <= 0 &&
         sgn(ccw(t[0],t[1],s[0]) * ccw(t[0],t[1],s[1])) <= 0;
}
bool intersectSP(const L& s, const P& p) {
  return abs(s[0]-p)+abs(s[1]-p)-abs(s[1]-s[0]) < eps; // triangle inequality
}
```

```
P reflection(const L& l, P p) {
  return p + P(2,0) * (pedal(l, p) - p);
}
D distanceLP(const L& l, P p) {
  return abs(p - pedal(l, p));
}
D distanceLL(const L& l, const L& m) {
  return intersectLL(l, m) ? 0 : distanceLP(l, m[0]);
}
D distanceLS(const L& l, const L& s) {
  if (intersectLS(l, s)) return 0;
  return min(distanceLP(l, s[0]), distanceLP(l, s[1]));
}
D distanceSP(const L& s, P p) {
  P r = pedal(s, p);
  if (intersectSP(s, r)) return abs(r - p);
  return min(abs(s[0] - p), abs(s[1] - p));
}
D distanceSS(const L& s, const L& t) {
  if (intersectSS(s, t)) return 0;
  return min(min(distanceSP(s, t[0]), distanceSP(s, t[1])),
             min(distanceSP(t, s[0]), distanceSP(t, s[1])));
}
P crosspoint(const L& l, const L& m) { // return intersection point
  D A = cross(dir(l), dir(m));
  D B = cross(dir(l), l[1] - m[0]);
  return m[0] + B / A * dir(m);
}
L bisector(P a, P b) {
  P A = (a+b)*P(0.5,0);
  return line(A, A+(b-a)*P(0,1));
}

#define next(g,i) g[(i+1)%g.size()]
#define prev(g,i) g[(i+g.size()-1)%g.size()]
L edge(const G& g, int i) { return line(g[i], next(g,i)); }
D area(const G& g) {
  D A = 0;
  rep(i,0,g.size())
    A += cross(g[i], next(g,i));
  return abs(A/2);
}

// intersect with half-plane left of l[0] -> l[1]
G convex_cut(const G& g, const L& l) {
  G Q;
  rep(i,0,g.size()) {
    P A = g[i], B = next(g,i);
    if (ccw(l[0], l[1], A) != -1) Q.pb(A);
    if (ccw(l[0], l[1], A)*ccw(l[0], l[1], B) < 0)
      Q.pb(crosspoint(line(A, B), l));
  }
  return Q;
}
bool convex_contain(const G& g, P p) { // check if point is inside convex polygon
  rep(i,0,g.size())
    if (ccw(g[i], next(g, i), p) == -1) return 0;
  return 1;
}
G convex_intersect(G a, G b) { // intersect two convex polygons
  rep(i,0,b.size())
    a = convex_cut(a, edge(b, i));
  return a;
}
void triangulate(G g, vector<G>& res) { // triangulate a simple polygon
  while (g.size() > 3) {
    bool found = 0;
    rep(i,0,g.size()) {
      if (ccw(prev(g,i), g[i], next(g,i)) != +1) continue;
      G tri;
      tri.pb(prev(g,i));
      tri.pb(g[i]);
      tri.pb(next(g,i));
      bool valid = 1;
      rep(j,0,g.size()) {
        if ((j+1)%g.size() == i || j == i || j == (i+1)%g.size()) continue;
        if (convex_contain(tri, g[j])) {
          valid = 0;
          break;
        }
      }
      if (!valid) continue;
```

```
      res.pb(tri);
      g.erase(g.begin() + i);
      found = 1; break;
    }
    assert(found);
  }
  res.pb(g);
}
void graham_step(G& a, G& st, int i, int bot) {
  while (st.size()>bot && sgn(cross(*(st.end()-2), st.back(), a[i]))<=0)
    st.pop_back();
  st.pb(a[i]);
}
bool cmpY(P a, P b) { return mk(imag(a),real(a)) < mk(imag(b),real(b)); }
G graham_scan(const G& points) { // will return points in ccw order
  // special case: all points coincide, algo might return point twice
  G a = points; sort(all(a),cmpY);
  int n = a.size();
  if (n<=1) return a;
  G st; st.pb(a[0]); st.pb(a[1]);
  for (int i = 2; i < n; i++) graham_step(a,st,i,1);
  int mid = st.size();
  for (int i = n - 2; i >= 0; i--) graham_step(a,st,i,mid);
  while (st.size() > 1 && !sgn(abs(st.back() - st.front()))) st.pop_back();
  return st;
}
G gift_wrap(const G& points) { // will return points in clockwise order
  // special case: duplicate points, not sure what happens then
  int n = points.size();
  if (n<=2) return points;
  G res;
  P nxt, p = *min_element(all(points), [](const P& a, const P& b){
    return real(a) < real(b);
  });
  do {
    res.pb(p);
    nxt = points[0];
    for (auto& q: points)
      if (abs(p - q) > eps && (abs(p - nxt) < eps || ccw(p, nxt, q) == 1))
        nxt = q;
    p = nxt;
  } while (nxt != *begin(res));
  return res;
}
G voronoi_cell(G g, const vector<P> &v, int s) {
  rep(i,0,v.size())
    if (i!=s)
      g = convex_cut(g, bisector(v[s], v[i]));
  return g;
}
const int ray_iters = 20;
bool simple_contain(const G& g, P p) { // check if point is inside simple polygon
  int yes = 0;
  rep(_,0,ray_iters) {
    D angle = 2*pi * (D)rand() / RAND_MAX;
    P dir = rotate(P(inf,inf), angle);
    L s = line(p, p + dir);
    int cnt = 0;
    rep(i,0,g.size()) {
      if (intersectSS(edge(g, i), s)) cnt++;
    }
    yes += cnt%2;
  }
  return yes > ray_iters/2;
}
bool intersectGG(const G& g1, const G& g2) {
  if (convex_contain(g1, g2[0])) return 1;
  if (convex_contain(g2, g1[0])) return 1;
  rep(i,0,g1.size()) rep(j,0,g2.size()) {
    if (intersectSS(edge(g1, i), edge(g2, j))) return 1;
  }
  return 0;
}
D distanceGP(const G& g, P p) {
  if (convex_contain(g, p)) return 0;
  D res = inf;
  rep(i,0,g.size())
    res = min(res, distanceSP(edge(g, i), p));
  return res;
}
P centroid(const G& v) {
  D S = 0;
```

```cpp
  P res;
  rep(i,0,v.size()) {
    D tmp = cross(v[i], next(v,i));
    S += tmp;
    res += (v[i] + next(v,i)) * tmp;
  }
  S /= 2;
  res /= 6*S;
  return res;
}

struct C {
  P p; D r;
  C(P p, D r) : p(p),r(r) {}
  C(){}
};
// intersect circle with line through (c.p + v * dst/abs(v)) "orthogonal" to the circle
// dst can be negative
G intersectCL2(const C& c, D dst, P v) {
  G res;
  P mid = c.p + v * (dst/abs(v));
  if (sgn(abs(dst)-c.r) == 0) { res.pb(mid); return res; }
  D h = sqrt(sq(c.r) - sq(dst));
  P hi = scale(v * P(0,1), h);
  res.pb(mid + hi); res.pb(mid - hi);
  return res;
}
G intersectCL(const C& c, const L& l) {
  if (intersectLP(l, c.p)) {
    P h = scale(dir(l), c.r);
    G res; res.pb(c.p + h); res.pb(c.p - h); return res;
  }
  P v = pedal(l, c.p) - c.p;
  return intersectCL2(c, abs(v), v);
}
G intersectCS(const C& c, const L& s) {
  G res1 = intersectCL(c,s), res2;
  for(auto it: res1) if (intersectSP(s, it)) res2.pb(it);
  return res2;
}
int intersectCC(const C& a, const C& b, G& res=dummy) {
  D sum = a.r + b.r, diff = abs(a.r - b.r), dst = abs(a.p - b.p);
  if (dst > sum + eps || dst < diff - eps) return 0;
  if (max(dst, diff) < eps) { // same circle
    if (a.r < eps) { res.pb(a.p); return 1; } // degenerate
    return -1; // infinitely many
  }
  D p = (sq(a.r) - sq(b.r) + sq(dst))/(2*dst);
  P ab = b.p - a.p;
  res = intersectCL2(a, p, ab);
  return res.size();
}

using P3 = valarray<D>;
P3 p3(D x=0, D y=0, D z=0) {
  P3 res(3);
  res[0]=x;res[1]=y;res[2]=z;
  return res;
}
ostream& operator<<(ostream& out, const P3& x) {
  return out << "(" << x[0]<<","<<x[1]<<","<<x[2]<<")";
}
P3 cross(const P3& a, const P3& b) {
  P3 res;
  rep(i,0,3) res[i]=a[(i+1)%3]*b[(i+2)%3]-a[(i+2)%3]*b[(i+1)%3];
  return res;
}
D dot(const P3& a, const P3& b) {
  return a[0]*b[0]+a[1]*b[1]+a[2]*b[2];
}
D norm(const P3& x) { return dot(x,x); }
D abs(const P3& x) { return sqrt(norm(x)); }
D project(const P3& e, const P3& x) { return dot(e,x) / norm(e); }
P project_plane(const P3& v, P3 w, const P3& p) {
  w -= project(v,w)*v;
  return P(dot(p,v)/abs(v), dot(p,w)/abs(w));
}

template <typename T, int N> struct Matrix {
  T data[N][N];
  Matrix<T,N>(T d=0) { rep(i,0,N) rep(j,0,N) data[i][j] = i==j?d:0; }
  Matrix<T,N> operator+(const Matrix<T,N>& other) const {
```

```
    Matrix res; rep(i,0,N) rep(j,0,N) res[i][j] = data[i][j] + other[i][j]; return res;
  }
  Matrix<T,N> operator*(const Matrix<T,N>& other) const {
    Matrix res; rep(i,0,N) rep(k,0,N) rep(j,0,N) res[i][j] += data[i][k] * other[k][j]; return res;
  }
  Matrix<T,N> transpose() const {
    Matrix res; rep(i,0,N) rep(j,0,N) res[i][j] = data[j][i]; return res;
  }
  array<T,N> operator*(const array<T,N>& v) const {
    array<T,N> res;
    rep(i,0,N) rep(j,0,N) res[i] += data[i][j] * v[j];
    return res;
  }
  const T* operator[](int i) const { return data[i]; }
  T* operator[](int i) { return data[i]; }
};
template <typename T, int N> ostream& operator<<(ostream& out, Matrix<T,N> mat) {
  rep(i,0,N) { rep(j,0,N) out << mat[i][j] << " "; cout << endl; } return out;
} // creates a rotation matrix around axis x (must be normalized). Rotation is
// counter-clockwise if you look in the inverse direction of x onto the origin
template<typename M> void create_rot_matrix(M& m, double x[3], double a) {
  rep(i,0,3) rep(j,0,3) {
    m[i][j] = x[i]*x[j]*(1-cos(a));
    if (i == j) m[i][j] += cos(a);
    else m[i][j] += x[(6-i-j)%3] * ((i == (2+j) % 3) ? -1 : 1) * sin(a);
  }
}
```

5.2 Graham's Scan + max. Abstand

```
/* Runtime: O(n*log(n)). Find 2 farthest points in a set of points.
 * Use graham algorithm to get the convex hull.
 * Note: In extreme situation, when all points coincide, the program won't work
 * probably. A prejudge of this situation may consequently be needed */
const int mn = 100005;
const double pi = acos(-1.0), eps = 1e-5;
struct point { double x, y; } a[mn];
int n, cn, st[mn];
inline bool cmp(const point &a, const point &b) {
    if (a.y != b.y) return a.y < b.y; return a.x < b.x;
}
inline int dblcmp(const double &d) {
    if (abs(d) < eps) return 0; return d < 0 ? -1 : 1;
}
inline double cross(const point &a, const point &b, const point &c) {
    return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
}
inline double dis(const point &a, const point &b) {
    double dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
} // get the convex hull
void graham_scan() {
    sort(a, a + n, cmp);
    cn = -1;
    st[++cn] = 0;
    st[++cn] = 1;
    for (int i = 2; i < n; i++) {
        while (cn>0 && dblcmp(cross(a[st[cn-1]],a[st[cn]],a[i]))<=0) cn--;
        st[++cn] = i;
    }
    int newtop = cn;
    for (int i = n - 2; i >= 0; i--) {
        while (cn>newtop && dblcmp(cross(a[st[cn-1]],a[st[cn]],a[i]))<=0) cn--;
        st[++cn] = i;
    }
}
inline int next(int x) { return x + 1 == cn ? 0 : x + 1; }
inline double angle(const point &a,const point &b,const point &c,const point &d){
    double x1 = b.x - a.x, y1 = b.y - a.y, x2 = d.x - c.x, y2 = d.y - c.y;
    double tc = (x1 * x2 + y1 * y2) / dis(a, b) / dis(c, d);
    return acos(abs(tc) > 1.0 ? (tc > 0 ? 1 : -1) * 1.0 : tc);
}
void maintain(int &p1, int &p2, double &nowh, double &nowd) {
    nowd = dis(a[st[p1]], a[st[next(p1)]]);
    nowh = cross(a[st[p1]], a[st[next(p1)]], a[st[p2]]) / nowd;
    while (1) {
        double h = cross(a[st[p1]], a[st[next(p1)]], a[st[next(p2)]]) / nowd;
        if (dblcmp(h - nowh) > 0) {
            nowh = h;
            p2 = next(p2);
```

```
        } else break;
    }
}
double find_max() {
    double suma = 0, nowh = 0, nowd = 0, ans = 0;
    int p1 = 0, p2 = 1;
    maintain(p1, p2, nowh, nowd);
    while (dblcmp(suma - pi) <= 0) {
        double t1 = angle(a[st[p1]], a[st[next(p1)]], a[st[next(p1)]],
                a[st[next(next(p1))]]);
        double t2 = angle(a[st[next(p1)]], a[st[p1]], a[st[p2]],a[st[next(p2)]]);
        if (dblcmp(t1 - t2) <= 0) {
            p1 = next(p1); suma += t1;
        } else {
            p1 = next(p1); swap(p1, p2); suma += t2;
        }
        maintain(p1, p2, nowh, nowd);
        double d = dis(a[st[p1]], a[st[p2]]);
        if (d > ans) ans = d;
    }
    return ans;
}
int main() {
    while (scanf("%d", &n) != EOF && n) {
        for (int i = 0; i < n; i++)
            scanf("%lf%lf", &a[i].x, &a[i].y);
        if (n == 2)
            printf("%.2lf\n", dis(a[0], a[1]));
        else {
            graham_scan();
            double mx = find_max();
            printf("%.2lf\n", mx);
        }
    }
    return 0;
}
```

6 Datenstrukturen

6.1 STL order statistics tree

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std; using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> Tree;
int main() {
    Tree X;
    for (int i = 1; i <= 16; i <<= 1) X.insert(i); // { 1, 2, 4, 8, 16 };
    cout << *X.find_by_order(3) << endl; // => 8
    cout << X.order_of_key(10) << endl; // => 4 = successor of 10 = min i such that X[i] >= 10
}
```

6.2 Skew Heaps (meldable priority queue)

```
/* The simplest meldable priority queues: Skew Heap
Merging (distroying both trees), inserting, deleting min: O(logn) amortised;*/
struct node{
    int key;
    node *lc,*rc;
    node(int k):key(k),lc(0),rc(0){}
}*root=0;
int size=0;
node* merge(node* x, node* y){
    if(!x)return y;
    if(!y)return x;
    if(x->key > y->key)swap(x,y);
    x->rc=merge(x->rc,y);
    swap(x->lc,x->rc);
    return x;
}
void insert(int x) { root=merge(root,new node(x)); size++;}
int delmin() {
    if(!root)return -1;
    int ret=root->key;
    node *troot=merge(root->lc,root->rc);
    delete root;
    root=troot;
```

```
    size--;
    return ret;
}
```

6.3 Treap

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

6.4 Fenwick Tree

```
const int n = 10; // ALL INDICES START AT 1 WITH THIS CODE!!

// mode 1: update indices, read prefixes
void update_idx(int tree[], int i, int val) { // v[i] += val
  for (; i <= n; i += i & -i) tree[i] += val;
}
int read_prefix(int tree[], int i) { // get sum v[1..i]
  int sum = 0;
  for (; i > 0; i -= i & -i) sum += tree[i];
  return sum;
}
int kth(int k) { // find kth element in tree (1-based index)
  int ans = 0;
  for (int i = maxl; i >= 0; --i) // maxl = largest i s.t. (1<<i) <= n
    if (ans + (1<<i) <= N && tree[ans + (1<<i)] < k) {
      ans += 1<<i;
      k -= tree[ans];
    }
  return ans+1;
}

// mode 2: update prefixes, read indices
```

```
void update_prefix(int tree[], int i, int val) { // v[1..i] += val
  for (; i > 0; i -= i & -i) tree[i] += val;
}
int read_idx(int tree[], int i) { // get v[i]
  int sum = 0;
  for (; i <= n; i += i & -i) sum += tree[i];
  return sum;
}

// mode 3: range-update range-query (using point-wise of linear functions)
const int maxn = 100100;
int n;
ll mul[maxn], add[maxn];

void update_idx(ll tree[], int x, ll val) {
  for (int i = x; i <= n; i += i & -i) tree[i] += val;
}
void update_prefix(int x, ll val) { // v[x] += val
  update_idx(mul, 1, val);
  update_idx(mul, x + 1, -val);
  update_idx(add, x + 1, x * val);
}
ll read_prefix(int x) { // get sum v[1..x]
  ll a = 0, b = 0;
  for (int i = x; i > 0; i -= i & -i) a += mul[i], b += add[i];
  return a * x + b;
}
void update_range(int l, int r, ll val) { // v[l..r] += val
  update_prefix(l - 1, -val);
  update_prefix(r, val);
}
ll read_range(int l, int r) { // get sum v[l..r]
  return read_prefix(r) - read_prefix(l - 1);
}
```

6.5 Simple tree aggregations

```
void maintain(int x, int exclude) {
  g[x] = 1;
  for (int y: adj[x]) {
    if (y == exclude) continue;
    g[x] += g[y];
  }
}
// build initial data structures with fixed root
void dfs1(int x, int from) {
  for (int y: adj[x]) if (y != from)
    dfs1(y, x);
  maintain(x, from);
}
// inspect data structures with x as root
void dfs2(int x, int from) {
  for (int y: adj[x]) if (y != from) {
    maintain(x, y);
    maintain(y, -1);
    dfs2(y, x);
  }
  maintain(x, from);
}
```

7 DP optimization

7.1 Convex hull (monotonic insert)

```
// convex hull, minimum
vector<ll> M, B;
int ptr;
bool bad(int a,int b,int c) {
  // use deterministic comuputation with long long if sufficient
  return (long double)(B[c]-B[a])*(M[a]-M[b])<(long double)(B[b]-B[a])*(M[a]-M[c]);
}
// insert with non-increasing m
void insert(ll m, ll b) {
  M.push_back(m);
  B.push_back(b);
  while (M.size() >= 3 && bad(M.size()-3, M.size()-2, M.size()-1)) {
    M.erase(M.end()-2);
    B.erase(B.end()-2);
```

```
  }
}
ll get(int i, ll x) {
  return M[i]*x + B[i];
}
// query with non-decreasing x
ll query(ll x) {
  ptr=min((int)M.size()-1,ptr);
  while (ptr<M.size()-1 && get(ptr+1,x)<get(ptr,x))
    ptr++;
  return get(ptr,x);
}
```

7.2 Dynamic convex hull

```
const ll is_query = -(1LL<<62);
struct Line {
    ll m, b;
    mutable function<const Line*()> succ;
    bool operator<(const Line& rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
        const Line* s = succ();
        if (!s) return 0;
        ll x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};
struct HullDynamic : public multiset<Line> { // will maintain upper hull for maximum
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;
        return (x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x->m);
    }
    void insert_line(ll m, ll b) {
        auto y = insert({ m, b });
        y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
        if (bad(y)) { erase(y); return; }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll eval(ll x) {
        auto l = *lower_bound((Line) { x, is_query });
        return l.m * x + l.b;
    }
};
```

8 Formelsammlung

8.1 Combinatorics

Classical Problems			
HanoiTower(HT) min steps	$T_n = 2^n - 1$	Regions by n lines	$L_n = n(n+1)/2 + 1$
Regions by n Zig lines	$Z_n = 2n^2 - n + 1$	Joseph Problem (every m-th)	$F_1 = 0$, $F_i = (F_{i-1} + m)\%i$
Joseph Problem (every 2nd)	rotate n 1-bit to left	HanoiTower (no direct A to C)	$T_n = 3^n - 1$
Bounded regions by n lines	$(n^2 - 3n + 2)/2$	Joseph given pos j,find m.(↓con.)	$m \equiv 1$ (mod $\frac{L}{p}$),
HT min steps A to C clockw.	$Q_n = 2R_{n-1} + 1$	$L(n) = lcm(1, ..., n)$, p prime $\in [\frac{n}{2}, n]$	$m \equiv j + 1 - n$ (mod p)
HT min steps C to A clockw.	$R_n = 2R_{n-1} + Q_{n-1} + 2$	$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$	$\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$
Egyptian Fraction	$\frac{m}{n} = \frac{1}{\lceil n/m \rceil} + (\frac{m}{n} - \frac{1}{\lceil n/m \rceil})$	Farey Seq given m/n, m'/n'	$m'' = \lfloor (n+N)/n' \rfloor m' - m$
Farey Seq given m/n, m''/n''	$m'/n' = \frac{m+m''}{n+n''}$	$m/n = 0/1$, $m'/n' = 1/N$	$n'' = \lfloor (n+N)/n' \rfloor n' - n$
#labeled rooted trees	n^{n-1}	#labeled unrooted trees	n^{n-2}
#SpanningTree of G (no SL)	$C(G) = D(G) - A(G)(\downarrow)$	Stirling's Formula	$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \frac{1}{12n})$
D : DegMat; A : AdjMat	$Ans = \lvert \det(C - 1r - 1c) \rvert$	Farey Seq	$mn' - m'n = -1$
#heaps of a tree (keys: $1..n$)	$\frac{(n-1)!}{\prod_{i \neq root} \text{size}(i)}$	#ways $0 \to m$ in n steps (never < 0)	$\frac{m+1}{\frac{n+m}{2}+1}\binom{n}{\frac{n+m}{2}}$
$\#seq\langle a_0, ..., a_{mn} \rangle$ of 1's and $(1-m)$'s with sum $+1 = \binom{mn+1}{n}\frac{1}{mn+1} = \binom{mn}{n}\frac{1}{(m-1)n+1}$			$\mathrm{D}_n = n\mathrm{D}_{n-1} + (-1)^n$

Binomial Coefficients

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$, int $n \geq k \geq 0$	$\binom{n}{k} = \binom{n}{n-k}$, int $n \geq 0$, int k	$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}$, int $k \neq 0$
$\binom{r}{k} = (-1)^k \binom{k-r-1}{k}$, int k	$\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k}$, int m, k	$(x+y)^r = \sum_k \binom{r}{k} x^k y^{r-k}$, int $r \geq 0$ or $\lvert x/y \rvert < 1$
$\binom{r}{k} = \binom{r-1}{k} + \binom{r-1}{k-1}$, int k	$\sum_{k \leq n} \binom{r+k}{k} = \binom{r+n+1}{n}$, int n	$\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}$, int $m, n \geq 0$
$\binom{r+s}{n} = \sum_k \binom{r}{k}\binom{s}{n-k}$, int n	$\sum_{k \leq m} \binom{r}{k}(\frac{r}{2} - k) = \frac{m+1}{2}\binom{r}{m+1}$, int m	$\sum_{k \leq m} \binom{r}{k}(-1)^k = (-1)^m \binom{r-1}{m}$, int m
$\sum_k \binom{r}{m+k}\binom{s}{n-k} = \binom{r+s}{m+n}$, int m, n	$\binom{\binom{k}{2}}{2} = 3\binom{k+1}{4}$ $\quad\vert\quad$ $\sum_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n}$	$\sum_k \binom{l}{m+k}\binom{s}{n+k} = \binom{l+s}{l-m+n}$ int $l \geq 0$, int m, n
$\sum_k \binom{n}{2k} = 2^{n-even(n)}$	$lcm_{i=0}^{n} \binom{n}{i} = \frac{L(n+1)}{n+1}$	$S(n,1) = S(n,n) = n \Rightarrow S(n,k) = \binom{n+1}{k} - \binom{n-1}{k-1}$
$\sum_{i=1}^{n} \binom{n}{i} F_i = F_{2n}$, $F_n = n$-th Fib	$\sum_i \binom{n-i}{i} = F_{n+1}$	

Famous Numbers

Catalan	$C_0 = 1$, $C_n = \frac{1}{n+1}\binom{2n}{n} = \sum_{i=0}^{n-1} C_i C_{n-i-1} = \frac{4n-2}{n+1} C_{n-1}$	
Stirling 1st kind	$\left[{0 \atop 0}\right] = 1$, $\left[{n \atop 0}\right] = \left[{0 \atop n}\right] = 0$, $\left[{n \atop k}\right] = (n-1)\left[{n-1 \atop k}\right] + \left[{n-1 \atop k-1}\right]$	#perms of n objs with exactly k cycles
Stirling 2nd kind	$\left\{{n \atop 1}\right\} = \left\{{n \atop n}\right\} = 1$, $\left\{{n \atop k}\right\} = k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\}$	#ways to partition n objs into k nonempty sets
Euler	$\left\langle{n \atop 0}\right\rangle = \left\langle{n \atop n-1}\right\rangle = 1$, $\left\langle{n \atop k}\right\rangle = (k+1)\left\langle{n-1 \atop k}\right\rangle + (n-k)\left\langle{n-1 \atop k-1}\right\rangle$	#perms of n objs with exactly k ascents
Euler 2nd Order	$\left\langle\!\left\langle{n \atop k}\right\rangle\!\right\rangle = (k+1)\left\langle\!\left\langle{n-1 \atop k}\right\rangle\!\right\rangle + (2n-k-1)\left\langle\!\left\langle{n-1 \atop k-1}\right\rangle\!\right\rangle$	#perms of $1, 1, 2, 2, ..., n, n$ with exactly k ascents
Bell	$B_1 = 1$, $B_n = \sum_{k=0}^{n-1} B_k \binom{n-1}{k} = \sum_{k=0}^{n} \left\{{n \atop k}\right\}$	#partitions of $1..n$ (Stirling 2nd, no limit on k)

The Twelvefold Way (Putting n balls into k boxes)

Balls Boxes	same same	distinct same	same distinct	distinct distinct	Remarks
-	$\mathrm{p}_k(n)$	$\sum_{i=0}^{k} \left\{{n \atop i}\right\}$	$\binom{n+k-1}{k-1}$	k^n	$\mathrm{p}_k(n)$: #partitions of n into $\leq k$ positive parts
size ≥ 1	$\mathrm{p}(n,k)$	$\left\{{n \atop k}\right\}$	$\binom{n-1}{k-1}$	$k!\left\{{n \atop k}\right\}$	$\mathrm{p}(n,k)$: #partitions of n into k positive parts (`NrPartitions`)
size ≤ 1	$[n \leq k]$	$[n \leq k]$	$\binom{k}{n}$	$n!\binom{k}{n}$	$[cond]$: 1 if $cond = true$, else 0

Classical Formulae

Ballot.Always $\#A > k\#B$	$Pr = \frac{a-kb}{a+b}$	Ballot.Always $\#B - \#A \leq k$	$Pr = 1 - \frac{a!b!}{(a+k+1)!(b-k-1)!}$
Ballot.Always $\#A \geq k\#B$	$Pr = \frac{a+1-kb}{a+1}$	Ballot.Always $\#A \geq \#B + k$	$Num = \frac{a-k+1-b}{a-k+1}\binom{a+b-k}{b}$
$E(X+Y) = EX + EY$	$E(\alpha X) = \alpha EX$	X,Y indep. $\Leftrightarrow E(XY) = (EX)(EY)$	

Burnside's Lemma: $L = \frac{1}{|G|}\sum_{k=1}^{n} |Z_k| = \frac{1}{|G|}\sum_{a_i \in G} C_1(a_i)$. Z_k : the set of permutations in G under which k stays stable; $C_1(a_i)$: the number of cycles of order 1 in a_i. **Pólya's Theorem:** The number of colorings of n objects with m colors $L = \frac{1}{|\overline{G}|}\sum_{g_i \in \overline{G}} m^{c(g_i)}$. $\overline{G}$: the group over n objects; $c(g_i)$:the number of cycles in g_i.

Regular Polyhedron Coloring **with at most n colors (up to isomorph)**

Description	*Formula*	Remarks	
vertices of octahedron or **faces of cube**	$(n^6 + 3n^4 + 12n^3 + 8n^2)/24$		(V, F, E)
vertices of cube or **faces of octahedron**	$(n^8 + 17n^4 + 6n^2)/24$	tetrahedron:	$(4, 4, 6)$
edges of cube or **edges of octahedron**	$(n^{12} + 6n^7 + 3n^6 + 8n^4 + 6n^3)/24$	cube:	$(8, 6, 12)$
vertices or **faces of tetrahedron**	$(n^4 + 11n^2)/12$	octahedron:	$(6, 8, 12)$
edges of tetrahedron	$(n^6 + 3n^4 + 8n^2)/12$	dodecahedron:	$(20, 12, 30)$
vertices of icosahedron or **faces of dodecahedron**	$(n^{12} + 15n^6 + 44n^4)/60$	icosahedron	$(12, 20, 30)$
vertices of dodecahedron or **faces of icosahedron**	$(n^{20} + 15n^{10} + 20n^8 + 24n^4)/60$		
edges of dodecahedron or **edges of icosahedron**	$(n^{30} + 15n^{16} + 20n^{10} + 24n^6)/60$	*This row may be wrong.*	

8.2 Number Theory

Classical Theorems

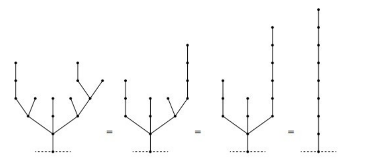
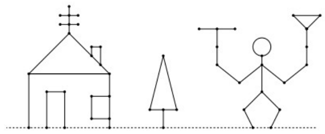
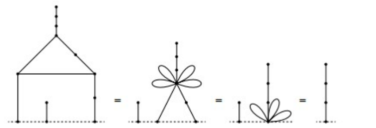
exp of p in $n!$ is $\sum_{i \geq 1}[\frac{n}{p^i}]$	$p_n \sim n \log n$; $\quad \forall_{n>1} \exists_{n<p<2n} : p$ is prime	$\pi(x) \sim \frac{x}{\log x}$; $\quad$ Norm($\alpha\beta$) = Norm(α)·Norm(β)
lcm(a,b)=$\frac{ab}{\mathrm{gcd}(a,b)}$	$a \equiv b$ (mod x, y) $\Rightarrow a \equiv b$ (mod lcm(x,y))	All prime factors of $2^{2^n} + 1$ have form $2^{n+2}k + 1$
$(2^a - 1, 2^b - 1) = 2^{(a,b)} - 1$	$ac \equiv bc$ (mod m) $\Rightarrow a \equiv b$(mod $\frac{m}{\mathrm{gcd}(c,m)}$)	n-plygn drawable $\Leftrightarrow n = 2^k \prod F_i$, F_i fermatNum
p_i is prime, $\prod_{p_i \leq n} p_i < 4^n$	$W \equiv d + [2.6m - 0.2] - 2C + Y + \left[\frac{Y}{4}\right] + \left[\frac{C}{4}\right]$ $(\%7)$.	$m = 11, 12, 1$ for Ja,Fe,Ma. J&F$\in$lastyear

Fibonacci period (pisano): gcd$(n,m) = 1 \implies \pi(n,m) =$ lcm$(\pi(n), \pi(m))$, $\pi(p^e) \mid p^{e-1}\pi(p)$, $\pi(5) = 20$, $\pi(p \neq 5) \mid p - 1$ or $2(p+1)$

Classical Theorems		
p prime $\Leftrightarrow (p-1)! \equiv -1(\%p)$	$a \perp m \Rightarrow a^{\phi(m)} = 1(\%m)$	Min general idx $\lambda(n)$: $\forall_a : a^{\lambda(n)} \equiv 1(\%n)$
$\sum_{d\mid n} \phi(d) = \sum_{d\mid n} \phi(n/d) = n$	$\sum_{m\perp n, m<n} m = \frac{n\phi(n)}{2}$	$\sum_{i=1}^{n} \sigma_0(i) = 2\sum_{i=1}^{[\sqrt{n}]}[n/j] - [\sqrt{n}]^2$
$(\sum_{d\mid n} \sigma_0(d))^2 = \sum_{d\mid n} \sigma_0(d)^3$	$\sum_{d\mid n} n\sigma_1(d)/d = \sum_{d\mid n} d\sigma_0(d)$	$[\sqrt{n}]$ Newton: $y = [\frac{x+[n/x]}{2}]$, $x_0 = 2^{[\frac{\log_2(n)+2}{2}]}$
$\sigma_0(p_1^{e_1}\cdots p_s^{e_s}) = \prod_{i=1}^{s}(e_i+1)$	$\sigma_1(p_1^{e_1}\cdots p_s^{e_s}) = \prod_{i=1}^{s} \frac{p_i^{e_i+1}-1}{p_i-1}$	$r_1 = 4$, $r_k \equiv r_{k-1}^2 - 2(\%M_p)$, M_p prime $\Leftrightarrow r_{p-1} \equiv 0(\%M_p)$
$\mu(p_1p_2\cdots p_s) = (-1)^s$, else 0	$\sum_{d\mid n} \mu(d) = 1$ if $n=1$, else 0	$F(n) = \sum_{d\mid n} f(d) \Leftrightarrow f(n) = \sum_{d\mid n} \mu(d)F(\frac{n}{d})$
$n = \sum_{d\mid n} \mu(\frac{n}{d})\sigma_1(d)$	$n = 2, 4, p^t, 2p^t \Leftrightarrow n$ has p_roots	$a \perp n$, then $a^i \equiv a^j(\%n) \Leftrightarrow i \equiv j(\%\mathrm{ord}_n(a))$
$1 = \sum_{d\mid n} \mu(\frac{n}{d})\sigma_0(d)$	$r = \mathrm{ord}_n(a)$, $\mathrm{ord}_n(a^u) = \frac{r}{\gcd(r,u)}$	r p_root of n, then r^u is p_root of $n \Leftrightarrow u \perp \phi(n)$
$\mathrm{ord}_n(a) = \mathrm{ord}_n(a^{-1})$	r p_root of $n \Leftrightarrow r^{-1}$ p_root of n	n has p_roots $\Leftrightarrow n$ has $\phi(\phi(n))$ p_roots
$a^n \equiv a^{\phi(m)+n\%\phi(m)}(\%m)$,$n$big	$\lambda(2^t) = 2^{t-2}$, $\lambda(p^t) = \phi(p^t) = (p-1)p^{t-1}$, $\lambda(2^{t_0}p_1^{t_1}\cdots p_m^{t_m}) = lcm(\lambda(2^{t0}), \phi(p_1^{t_1}), \cdots, \phi(p_m^{t_m}))$	
$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2}(\%p)$	Legendre sym $\left(\frac{a}{p}\right) = 1$ if a is quad residue $\%p$; -1 if a is non-residue; 0 if $a = 0$	
$a \equiv b(\%p) \Rightarrow \left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$	$\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$; $\left(\frac{a^2}{p}\right) = 1$	$a \perp p$, s from $a, 2a, ..., \frac{p-1}{2}a(\%p)$ are $> \frac{p}{2} \Rightarrow \left(\frac{a}{p}\right) = (-1)^s$
$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}$	Gauss Integer $\pi = a + bi$. Norm$(\pi) = p$ prime $\Rightarrow \pi$ and $\overline{\pi}$ prime, p not prime	

8.3 Game Theory

Classical Games (❶ last one wins (normal); ❷ last one loses (misère))			
Name	Description	Criteria / Opt.strategy	Remarks
NIM	n piles of objs. One can take any number of objs from any pile (i.e. set of possible moves for the i-th pile is $M = [pile_i]$, $[x] := \{1, 2, ..., \lfloor x \rfloor\}$).	$SG = \otimes_{i=1}^{n} pile_i$. Strategy: ❶ make the Nim-Sum 0 by *decreasing* a heap; ❷ the same, except when the normal move would only leave heaps of size 1. In that case, leave an odd number of 1's.	The result of ❷ is the same as ❶, opposite if all piles are 1's. Many games are essentially NIM.
NIM (powers)	$M = \{a^m \mid m \geq 0\}$	If a odd: $SG_n = n\%2$	If a even: $SG_n = 2$, if $n \equiv a\%(a+1)$; $SG_n = n\%(a+1)\%2$, else.
NIM (half)	$M_{①} = [\frac{pile_i}{2}]$ $M_{②} = [\lceil \frac{pile_i}{2} \rceil, pile_i]$	①$SG_{2n} = n$, $SG_{2n+1} = SG_n$ ②$SG_0 = 0$, $SG_n = [\log_2 n] + 1$	
NIM (divisors)	$M_{①} =$ divisors of $pile_i$ $M_{②} =$ proper divisors of $pile_i$	①$SG_0 = 0$, $SG_n = SG_{②,n} + 1$ ②$SG_1 = 0$, $SG_n =$ number of 0's at the end of n_{binary}	
Subtraction Game	$M_{①} = [k]$ $M_{②} = S$ (finite) $M_{③} = S \cup \{pile_i\}$	$SG_{①,n} = n \mod (k+1)$. ❶lose if $SG = 0$; ❷lose if $SG = 1$. $SG_{③,n} = SG_{②,n} + 1$	For any finite M, SG of one pile is eventually periodic.
Moore's NIM$_k$	One can take any number of objs from at most k piles.	❶Write $pile_i$ in binary, sum up in base $k+1$ without carry. Losing if the result is 0.	❷ If all piles are 1's, losing iff $n \equiv 1\%(k+1)$. Otherwise the result is the same as ❶.
Staircase NIM	n piles in a line. One can take any number of objs from $pile_i$, $i > 0$ to $pile_{i-1}$	Losing if the NIM formed by the odd-indexed piles is losing(i.e. $\otimes_{i=0}^{(n-1)/2} pile_{2i+1} = 0$)	
Lasker's NIM	Two possible moves: 1.take any number of objs; 2.split a pile into two (no obj removed)	$SG_n = n$, if $n \equiv 1, 2(\%4)$ $SG_n = n+1$, if $n \equiv 3(\%4)$ $SG_n = n-1$, if $n \equiv 0(\%4)$	
Kayles	Two possible moves: 1.take 1 or 2 objs; 2.split a pile into two (after removing objs)	SG_n for small n can be computed recursively. SG_n for $n \in [72, 83]$: 4 1 2 8 1 4 7 2 1 8 2 7	SG_n becomes periodic from the 72-th item with period length 12.
Dawson's Chess	n boxes in a line. One can occupy a box if its neighbours are not occupied.	SG_n for $n \in [1, 18]$: 1 1 2 0 3 1 1 0 3 3 2 2 4 0 5 2 2 3	Period = 34 from the 52-th item.
Wythoff's Game	**Two** piles of objs. One can take any number of objs from either pile, or take the *same* number from *both* piles.	$n_k = \lfloor k\phi \rfloor = \lfloor m_k\phi \rfloor - m_k$ $m_k = \lfloor k\phi^2 \rfloor = \lceil n_k\phi \rceil = n_k + k$ $\phi := \frac{1+\sqrt{5}}{2}$. (n_k, m_k) is the k-th losing position.	n_k and m_k form a pair of complementary Beatty Sequences (since $\frac{1}{\phi} + \frac{1}{\phi^2} = 1$). Every $x > 0$ appears either in n_k or in m_k.
Mock Turtles	n coins in a line. One can turn over 1, 2 or 3 coins, with the rightmost from head to tail.	$SG_n = 2n$, if $\mathrm{ones}(2n)$ odd; $SG_n = 2n+1$, else. $\mathrm{ones}(x)$: the number of 1's in x_{binary}	SG_n for $n \in [0, 10]$ (leftmost position is 0): 1 2 4 7 8 11 13 14 16 19 21

Ruler	n coins in a line. One can turn over any *consecutive* coins, with the rightmost from head to tail.	SG_n = the largest power of 2 dividing n. This is implemented as $n\&-n$(lowbit)	SG_n for $n \in [1, 10]$: 1 2 1 4 1 2 1 8 1 2
Hackenbush-tree	Given a forest of rooted trees, one can take an edge and remove the part which becomes unrooted.	At every branch, one can replace the branches by a nonbranching stalk of length equal to their nim-sum.	
Hackenbush-graph		Vertices on any circuit can be *fused* without changing SG of the graph. Fusion: two neighbouring vertices into one, and bend the edge into a loop.	

- Johnson's Reweighting Algorithm: add a new source S, it can reach all other nodes with 0 cost. Use bellmanford to calculate the shortest path d[i] from S to all other nodes i. Exit when negative cycle is found. Otherwise the weights of all edges (u,v) in the original graph are changed to d[u]+w[u,v]-d[v]. Now all weights are nonnegative, so dijkstra algorithm can be used.
- feasible flow in a network with both upper and lower capacity constraints, no source or sink : capacity are changed to upperbound-lowerbound. Add a new source and a sink. let M[v] = (sum of lowerbounds of ingoing edges to v) - (sum of lowserbounds of outgoing edges from v). For all v, if M[v]>0 then add edge (S,v) with capacity M, otherwise add (v,T) with capacity -M. If all outgoing edges from S are full, then a feasible flow exists, it is the flow plus the original lowerbounds.
- feasible flow in a network with both upper and lower capacity constraints, with source s and sink t: add edge (t,s) with capacity infinity. Binary search for the lower bound, check whether a feasible exists for a network WITHOUT source or sink (B).
- system of difference constraints: change all the conditions to the form a-b<=c. For every such condition add an edge (b,a) with weight c. Add a source which can reach all the nodes with 0 cost. Find shortest paths with bellman ford from s. d[v] is a solution.
- min-weight vertex cover in a bipartite graph: partition into A and B. add edges $s \to A$ with capacities $w(A)$ and edges $B \to t$ with capacities $w(B)$. add edges of capacity ∞ from A to B where there are edges in the graph. answer is maxflow. the vertex cover is the set of nodes that are adjacent to cut edges, or alternatively, the left-side nodes NOT reachable from the source and the right-side edges reachable from the source (in the residual network).
- general graph: complement of a vertex cover is an independent set $\to$ max-weight independent set is complement of min-weight vertex cover.
- optimal proportion spanning tree: z=sigma(benifit[i] * x[i]) - l * sigma(cost[i] * x[i]) = sigma(d[i] * x[i]). binary search for l, find the MST so that z = 0, then l is the best proportion.
- optimal proportion cycle: same as above, change the "find MST"to "check if there're positive cycles"
- Bipartite Graph: Min Cover (fewest nodes cover all edges) = max matching. Min path covering for DAG: n - maxmatching. Min dominating set = max matching + isolated nodes. Max independent set = n - max matching
- Bipartite matching with weights on the left-hand nodes, minimizing the matched weight sum: sort left-hand nodes ascending by weight, then just use the normal bipartite matching algorithm (Kuhn's)
- Closure problem: Find a subset $V' \subset V$ such that V' is closed (every successor of a node in V' is also in V') and such that $\sum_{v \in V'} w(v)$ is maximal under all such subsets V'. We use min-cut: for every node v, if $w(v) > 0$, add an edge (S, v) with capacity $w(v)$, otherwise add edge (v, T) with capacity $-w(v)$. Add edges (v, w) with capacity ∞ for all edges (v, w) in the original graph. The source partition of the min-cut is the optimal V'.
- Erdős–Gallai theorem: A sequence of non-negative integers $d_1 \geq \cdots \geq d_n$ can be represented as the degree sequence of a finite simple graph on n vertices if and only if $d_1 + \cdots + d_n$ is even and $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k) \; \forall \, 1 \leq k \leq n$
- In a connected undirected graph, a random walk (uniform choice of next node) with any start node will hit all nodes in expected time $2m \cdot (n-1)$. We can also walk on the projection of some more complex graph into fewer dimensions. E.g. 2-SAT: Let T be a valid truth assignment. Start with any assignment $T*$. Let n be the number of variables in which T and $T*$ coincide. If we fix a broken clause by picking any of its variables at random and adding it to $T*$, we increase n with probability of at least $\frac{1}{2}$ (and decrease it otherwise). The graph we walk on is the integer number line, and we are expected to hit T after $2n^2$ iterations. If the distribution is non-uniform against your favor, it does not work at all (even if the probability to go in the "right"direction is only slightly less than $\frac{1}{2}$)

- Generally useful solution ideas (always consider!): divide and conquer, binary search, precomputation, output-sensitive algorithms, meet-in-the-middle, use different algos for different situations