



# **Team Contest Reference**

2014 ACM ICPC Northwestern European Regional Contest  
Linköping, November 29-30, 2014

Team hackKIT

# 1 Stringology

## 1.1 Z Algorithm

```

1  /* calculate the $z array for string $s of length $n in O(n) time.
2  * z[i] := the longest common prefix of s[0..n-1] and s[i..n-1].
3  * For pattern matching, make a string P$S and output positions with z[i]==|P|
4  * For pattern matching, there's no need to store (but to calculate) z[i] for i>|P|. */
5  void calc_Z(const char *s, int n, int *z) {
6      int l = 0, r = 0, p, q;
7      if(n > 0) z[0] = n;
8      for (int i = 1; i < n; ++i) {
9          if (i <= r && z[i - l] < r - i + 1) {
10             z[i] = z[i - l];
11         } else {
12             if (i > r) p = 0, q = i;
13             else p = r - i + 1, q = r + 1;
14             while (q < n && s[p] == s[q]) ++p, ++q;
15             z[i] = q - i, l = i, r = q - 1;
16         }
17     }
18 }

```

## 1.2 Rolling hash

```

1  int q = 311;
2  struct Hasher { // use two of those, with different mod (e.g. 1e9+7 and 1e9+9)
3      string s;
4      int mod;
5      vector<int> power, pref;
6      Hasher(const string& s, int mod) : s(s), mod(mod) {
7          power.pb(1);
8          rep(i, 1, s.size()) power.pb((ll)power.back() * q % mod);
9          pref.pb(0);
10         rep(i, 0, s.size()) pref.pb(((ll)pref.back() * q % mod + s[i]) % mod);
11     }
12     int hash(int l, int r) { // compute hash(s[l..r]) with r inclusive
13         return (pref[r+1] - (ll)power[r-l+1] * pref[l] % mod + mod) % mod;
14     }
15 };

```

## 1.3 Suffix Array - LCP Based

```

1  const int maxn = 200010, maxlg = 18; // maxlg = ceil(log_2(maxn))
2  struct SA {
3      pair<pii, int> L[maxn]; // O(n * log n) space
4      int P[maxlg+1][maxn], n, stp, cnt, sa[maxn];
5      SA(const string& s) : n(s.size()) { // O(n * log n)
6          rep(i, 0, n) P[0][i] = s[i];
7          sa[0] = 0; // in case n == 1
8          for (stp = 1, cnt = 1; cnt < n; stp++, cnt <= 1) {
9              rep(i, 0, n) L[i] = mk(mk(P[stp-1][i], i + cnt < n ? P[stp-1][i+cnt] : -1), i);
10             std::sort(L, L + n);
11             rep(i, 0, n)
12                 P[stp][L[i].snd] = i>0 && L[i].fst == L[i-1].fst ? P[stp][L[i-1].snd] : i;
13         }
14         rep(i, 0, n) sa[i] = L[i].snd;
15     }
16     int lcp(int x, int y) { // time log(n); x, y = indices into string, not SA
17         int k, ret = 0;
18         if (x == y) return n - x;
19         for (k = stp - 1; k >= 0 && x < n && y < n; k --)
20             if (P[k][x] == P[k][y])
21                 x += 1<<k, y += 1<<k, ret += 1<<k;
22         return ret;
23     }
24 };

```

## 1.4 Suffix automaton

```

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

```

```

6  void append(const string& str, int offset=0) { // O(|str|)
7      for (int i = 0; i < str.size(); ++i) {
8          int a = str[i];
9          State *cur = sink, *sufState;
10         sink = new State { sink->depth + 1, offset + i, {0}, 0 };
11         while (cur && !cur->go[a]) {
12             cur->go[a] = sink;
13             cur = cur->suffix;
14         }
15         if (!cur) sufState = root;
16         else {
17             State *q = cur->go[a];
18             if (q->depth == cur->depth + 1)
19                 sufState = q;
20             else {
21                 State *r = new State(*q);
22                 r->depth = cur->depth + 1;
23                 q->suffix = sufState = r;
24                 while (cur && cur->go[a] == q) {
25                     cur->go[a] = r;
26                     cur = cur->suffix;
27                 }
28             }
29         }
30         sink->suffix = sufState;
31     }
32 }
33 int walk(const string& str) { // O(|str|) returns LCS with automaton string
34     int tmp = 0;
35     State *cur = root;
36     int ans = 0;
37     for (int i = 0; i < str.size(); ++i) {
38         int a = str[i];
39         if (cur->go[a]) {
40             tmp++;
41             cur = cur->go[a];
42         } else {
43             while (cur && !cur->go[a])
44                 cur = cur->suffix;
45             if (!cur) {
46                 cur = root;
47                 tmp = 0;
48             } else {
49                 tmp = cur->depth + 1;
50                 cur = cur->go[a];
51             }
52         }
53         ans = max(ans, tmp); // i - tmp + 1 is start of match
54     }
55     return ans;
56 }
57 };

```

## 1.5 Aho-Corasick automaton

```

1  const int K = 20;
2  struct vertex {
3      vertex *next[K], *go[K], *link, *p;
4      int pch;
5      bool leaf;
6      int is_accepting = -1;
7  };
8
9  vertex *create() {
10     vertex *root = new vertex();
11     root->link = root;
12     return root;
13 }
14
15 void add_string (vertex *v, const vector<int>& s) {
16     for (int a: s) {
17         if (!v->next[a]) {
18             vertex *w = new vertex();
19             w->p = v;
20             w->pch = a;
21             v->next[a] = w;
22         }
23         v = v->next[a];
24     }
25     v->leaf = 1;

```

```

26 }
27
28 vertex* go(vertex* v, int c);
29
30 vertex* get_link(vertex *v) {
31     if (!v->link)
32         v->link = v->p->p ? go(get_link(v->p), v->pch) : v->p;
33     return v->link;
34 }
35
36 vertex* go(vertex* v, int c) {
37     if (!v->go[c]) {
38         if (v->next[c])
39             v->go[c] = v->next[c];
40         else
41             v->go[c] = v->p ? go(get_link(v), c) : v;
42     }
43     return v->go[c];
44 }
45
46 bool is_accepting(vertex *v) {
47     if (v->is_accepting == -1)
48         v->is_accepting = v->leaf || is_accepting(get_link(v));
49     return v->is_accepting;
50 }

```

## 2 Arithmetik und Algebra

### 2.1 Lineare Gleichungssysteme (LGS) und Determinanten

#### 2.1.1 Gauß-Algorithmus

```

1  class R {
2      BigInteger n, d;
3      R(BigInteger n_, BigInteger d_) {
4          n = n_; d = d_;
5          BigInteger g = n.gcd(d);
6          n.divide(g); d.divide(g);
7      }
8      R add(R x) {
9          return new R(n.multiply(x.d).add(d.multiply(x.n)), d.multiply(x.d));
10     }
11     R negate() { return new R(n.negate(), d); }
12     R subtract(R x) { return add(x.negate()); }
13     R multiply(R y) {
14         return new R(n.multiply(x.n), d.multiply(x.d));
15     }
16     R invert() { return new R(d, n); }
17     R divide(R y) { return multiply(y.invert()); }
18     boolean zero() { return d.equals(BigInteger.ZERO); }
19 }
20
21 int maxm = 13, maxn = 4;
22 R[][] M = new R[maxm][maxn]; // the LGS matrix
23 R[] B = new R[maxm];         // the right side
24
25 void gauss(int m, int n) { // reduces M to Gaussian normal form
26     int row = 0;
27     for (int col = 0; col < n; ++col) { // eliminate downwards
28         int pivot=row;
29         while(pivot<m&&M[pivot][col].zero())pivot++;
30         if (pivot == m || M[pivot][col].zero()) continue;
31         if (row!=pivot) {
32             for (int j = 0; j < n; ++j) {
33                 R tmp = M[row][j];
34                 M[row][j] = M[pivot][j];
35                 M[pivot][j] = tmp;
36             }
37             R tmp = B[row];
38             B[row] = B[pivot];
39             B[pivot] = tmp;
40         }
41         // for double, normalize pivot row here (divide it by pivot value)
42         for (int j = row+1; j < m; ++j) {
43             if (M[j][col].zero()) continue;
44             R a = M[row][col], b = M[j][col];
45             for(int k=0; k<n; ++k)
46                 M[j][k] = M[j][k].multiply(a).subtract(M[row][k].multiply(b));
47             B[j] = B[j].multiply(a).subtract(B[row].multiply(b));
48         }

```

```

49     row++;
50 }
51 for (int col = 0; col < n; ++col) { // eliminate upwards
52     for (row = m-1; row >= 0; --row) {
53         if (M[row][col].zero()) continue;
54         boolean valid=true;
55         for (int j = 0; j < col; ++j)
56             if (!M[row][j].zero()) { valid=false; break; }
57         if (!valid) continue;
58         for (int i = 0; i < row; ++i) {
59             R a = M[row][col], b = M[i][col];
60             for (int k = 0; k < i; ++k)
61                 M[i][k] = M[i][k].multiply(a).subtract(M[row][k].multiply(b));
62             B[i] = B[i].multiply(a).subtract(B[row].multiply(b));
63         }
64         break;
65     }
66 }
67 }

```

### 2.1.2 LR-Zerlegung, Determinanten

```

1  const int MAX = 42;
2  void lr(double a[MAX][MAX], int n) {
3      for (int i = 0; i < n; ++i) {
4          for (int k = 0; k < i; ++k) a[i][i] -= a[i][k] * a[k][i];
5          for (int j = i + 1; j < n; ++j) {
6              for (int k = 0; k < i; ++k) a[j][i] -= a[j][k] * a[k][i];
7              a[j][i] /= a[i][i];
8              for (int k = 0; k < i; ++k) a[i][j] -= a[i][k] * a[k][j];
9          }
10     }
11 }
12 double det(double a[MAX][MAX], int n) {
13     lr(a, n);
14     double d = 1;
15     for (int i = 0; i < n; ++i) d *= a[i][i];
16     return d;
17 }
18 void solve(double a[MAX][MAX], double *b, int n) {
19     for (int i = 1; i < n; ++i)
20         for (int j = 0; j < i; ++j) b[i] -= a[i][j] * b[j];
21     for (int i = n - 1; i >= 0; --i) {
22         for (int j = i + 1; j < n; ++j) b[i] -= a[i][j] * b[j];
23         b[i] /= a[i][i];
24     }
25 }

```

## 2.2 Numerical Integration (Adaptive Simpson's rule)

```

1  double f(double x) { return exp(-x*x); }
2  const double eps=1e-12;
3
4  double simps(double a, double b) { // for ~4x less f() calls, pass fa, fm, fb around
5      return (f(a) + 4*f((a+b)/2) + f(b))* (b-a)/6;
6  }
7  double integrate(double a, double b) {
8      double m = (a+b)/2;
9      double l = simps(a,m), r = simps(m,b), tot=simps(a,b);
10     if (fabs(l+r-tot) < eps) return tot;
11     return integrate(a,m) + integrate(m,b);
12 }

```

## 2.3 FFT

```

1  typedef double D; // or long double?
2  typedef complex<D> cplx; // use own implementation for 2x speedup
3  const D pi = acos(-1); // or -1.L for long double
4
5  // input should have size 2^k
6  vector<cplx> fft(const vector<cplx>& a, bool inv=0) {
7      int logn=1, n=a.size();
8      vector<cplx> A(n);
9      while((1<<logn)<n) logn++;
10     rep(i,0,n) {
11         int j=0; // precompute j = rev(i) if FFT is used more than once

```

```

12     rep(k,0,logn) j = (j<<1) | ((i>>k)&1);
13     A[j] = a[i]; }
14     for(int s=2; s<=n; s<=1) {
15         D ang = 2 * pi / s * (inv ? -1 : 1);
16         cplx ws(cos(ang), sin(ang));
17         for(int j=0; j<n; j+=s) {
18             cplx w=1;
19             rep(k,0,s/2) {
20                 cplx u = A[j+k], t = A[j+s/2+k];
21                 A[j+k] = u + w*t;
22                 A[j+s/2+k] = u - w*t;
23                 if(inv) A[j+k] /= 2, A[j+s/2+k] /= 2;
24                 w *= ws; } } }
25     return A;
26 }
27 vector<cplx> a = {0,0,0,0,1,2,3,4}, b = {0,0,0,0,2,3,0,1}; // polynomials
28 a = fft(a); b = fft(b);
29 rep(i,0,a.size()) a[i] *= b[i]; // convult spectrum
30 a = fft(a,1); // ifft, a = a * b

```

## 3 Zahlentheorie

### 3.1 Miscellaneous

```

1 ll multiply_mod(ll a, ll b, ll mod) {
2     if (b == 0) return 0;
3     if (b & 1) return ((ull)multiply_mod(a, b-1, mod) + a) % mod;
4     return multiply_mod((ull)a + a) % mod, b/2, mod);
5 }
6
7 ll powmod(ll a, ll n, ll mod) {
8     if (n == 0) return 1 % mod;
9     if (n & 1) return multiply_mod(powmod(a, n-1, mod), a, mod);
10    return powmod(multiply_mod(a, a, mod), n/2, mod);
11 }
12
13 // simple modinv, returns 0 if inverse doesn't exist
14 ll modinv(ll a, ll m) {
15     return a < 2 ? a : ((1 - m * ll1 * modinv(m % a, a)) / a % m + m) % m;
16 }
17 ll modinv_prime(ll a, ll p) { return powmod(a, p-2, p); }
18
19 ll extended_gcd(ll a, ll b, ll& lastx, ll& lasty) {
20     ll x, y, q, tmp;
21     x = 0; lastx = 1;
22     y = 1; lasty = 0;
23     while (b != 0) {
24         q = a / b;
25         tmp = b;
26         b = a % b;
27         a = tmp;
28         tmp = x; x = lastx - q*x; lastx = tmp;
29         tmp = y; y = lasty - q*y; lasty = tmp;
30     }
31     return a;
32 }
33
34 // solve the linear equation a x == b (mod n)
35 // returns the number of solutions up to congruence (can be 0)
36 // sol: the minimal positive solution
37 // dis: the distance between solutions
38 ll linear_mod(ll a, ll b, ll n, ll &sol, ll &dis) {
39     a = (a % n + n) % n, b = (b % n + n) % n;
40     ll d, x, y;
41     d = extended_gcd(a, n, x, y);
42     if (b % d)
43         return 0;
44     x = (x % n + n) % n;
45     x = b / d * x % n;
46     dis = n / d;
47     sol = x % dis;
48     return d;
49 }
50
51 bool rabin(ll n) {
52     // bases chosen to work for all n < 2^64, see https://oeis.org/A014233
53     set<int> p { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };
54     if (n <= 37) return p.count(n);
55     ll s = 0, t = n - 1;
56     while (~t & 1)

```

```

57     t >>= 1, ++s;
58     for (int x: p) {
59         ll pt = powmod(x, t, n);
60         if (pt == 1) continue;
61         bool ok = 0;
62         for (int j = 0; j < s && !ok; ++j) {
63             if (pt == n - 1) ok = 1;
64             pt = multiply_mod(pt, pt, n);
65         }
66         if (!ok) return 0;
67     }
68     return 1;
69 }
70
71 ll rho(ll n) {
72     if (~n & 1) return 2;
73     ll c = rand() % n, x = rand() % n, y = x, d = 1;
74     while (d == 1) {
75         x = (multiply_mod(x, x, n) + c) % n;
76         y = (multiply_mod(y, y, n) + c) % n;
77         y = (multiply_mod(y, x, n) + c) % n;
78         d = __gcd(abs(x - y), n);
79     }
80     return d == n ? rho(n) : d;
81 }
82
83 void factor(ll n, map<ll, int> &facts) {
84     if (n == 1) return;
85     if (rabin(n)) {
86         facts[n]++;
87         return;
88     }
89     ll f = rho(n);
90     factor(n/f, facts);
91     factor(f, facts);
92 }
93
94 // use inclusion-exclusion to get the number of integers <= n
95 // that are not divisible by any of the given primes.
96 // This essentially enumerates all the subsequences and adds or subtracts
97 // their product, depending on the current parity value.
98 ll count_coprime_rec(int primes[], int len, ll n, int i, ll prod, bool parity) {
99     if (i >= len || prod * primes[i] > n) return 0;
100     return (parity ? 1 : (-1)) * (n / (prod*primes[i]))
101         + count_coprime_rec(primes, len, n, i + 1, prod, parity)
102         + count_coprime_rec(primes, len, n, i + 1, prod * primes[i], !parity);
103 }
104 // use cnt(B) - cnt(A-1) to get matching integers in range [A..B]
105 ll count_coprime(int primes[], int len, ll n) {
106     if (n <= 1) return max(0LL, n);
107     return n - count_coprime_rec(primes, len, n, 0, 1, true);
108 }
109
110 // find x. a[i] x = b[i] (mod m[i]) 0 <= i < n. m[i] need not be coprime
111 bool crt(int n, ll *a, ll *b, ll *m, ll &sol, ll &mod) {
112     ll A = 1, B = 0, ta, tm, tsol, tdis;
113     for (int i = 0; i < n; ++i) {
114         if (!linear_mod(a[i], b[i], m[i], tsol, tdis)) return 0;
115         ta = tsol, tm = tdis;
116         if (!linear_mod(A, ta - B, tm, tsol, tdis)) return 0;
117         B = A * tsol + B;
118         A = A * tdis;
119     }
120     sol = B, mod = A;
121     return 1;
122 }
123
124 // get number of permutations {P_1, ..., P_n} of size n,
125 // where no number is at its original position (that is, P_i != i for all i)
126 // also called subfactorial !n
127 ll get_derangement_mod_m(ll n, ll m) {
128     vector<ll> res(m * 2);
129     ll d = 1 % m, p = 1;
130     res[0] = d;
131     for (int i = 1; i <= min(n, 2 * m - 1); ++i) {
132         p *= -1;
133         d = (1LL * i * d + p + m) % m;
134         res[i] = d;
135         if (i == n) return d;
136     }
137     // it turns out that !n mod m == !(n mod 2m) mod m
138     return res[n % (2 * m)];

```

```

139 }
140
141 // compute totient function for integers <= n
142 vector<int> compute_phi(int n) {
143     vector<int> phi(n + 1, 0);
144     for (int i = 1; i <= n; ++i) {
145         phi[i] += i;
146         for (int j = 2 * i; j <= n; j += i) {
147             phi[j] -= phi[i];
148         }
149     }
150     return phi;
151 }
152
153 // checks if g is primitive root mod p. Generate random g's to find primitive root.
154 bool is_primitive(ll g, ll p) {
155     map<ll, int> facs;
156     factor(p - 1, facs);
157     for (auto& f : facs)
158         if (1 == powmod(g, (p-1)/f.first, p))
159             return 0;
160     return 1;
161 }
162
163 ll dlog(ll g, ll b, ll p) { // find x such that g^x = b (mod p)
164     ll m = (ll)(ceil(sqrt(p-1))+0.5); // better use binary search here...
165     unordered_map<ll,ll> powers; // should compute this only once per g
166     rep(j,0,m) powers[powmod(g, j, p)] = j;
167     ll gm = powmod(g, -m + 2*(p-1), p);
168     rep(i,0,m) {
169         if (powers.count(b)) return i*m + powers[b];
170         b = b * gm % p;
171     }
172     assert(0); return -1;
173 }
174
175 // compute p(n,k), the number of possibilities to write n as a sum of
176 // k non-zero integers
177 ll count_partitions(int n, int k) {
178     if (n==k) return 1;
179     if (n<k || k==0) return 0;
180     vector<ll> p(n + 1);
181     for (int i = 1; i <= n; ++i) p[i] = 1;
182     for (int l = 2; l <= k; ++l)
183         for (int m = l+1; m <= n-l+1; ++m)
184             p[m] = p[m] + p[m-l];
185     return p[n-k+1];
186 }

```

## 3.2 Binomial Coefficient modulo M

```

1 // calculate (product_{i=1,i%p!=0}^n i) % p^e. cnt is the exponent of p in n!
2 // Time: p^e + log(p, n)
3 int get_part_of_fac_n_mod_pe(int n, int p, int mod, int *upto, int &cnt) {
4     if (n < p) { cnt = 0; return upto[n]; }
5     else {
6         int res = powmod(upto[mod], n / mod, mod);
7         res = (ll) res * upto[n % mod] % mod;
8         res = (ll) res * get_part_of_fac_n_mod_pe(n / p, p, mod, upto, cnt) % mod;
9         cnt += n / p;
10        return res;
11    }
12 }
13 //C(n,k) % p^e. Use Chinese Remainder Theorem to get C(n,k)%m
14 int get_n_choose_k_mod_pe(int n, int k, int p, int mod) {
15     static int upto[maxm + 1];
16     upto[0] = 1 % mod;
17     for (int i = 1; i <= mod; ++i)
18         upto[i] = i % p ? (ll) upto[i - 1] * i % mod : upto[i - 1];
19     int cnt1, cnt2, cnt3;
20     int a = get_part_of_fac_n_mod_pe(n, p, mod, upto, cnt1);
21     int b = get_part_of_fac_n_mod_pe(k, p, mod, upto, cnt2);
22     int c = get_part_of_fac_n_mod_pe(n - k, p, mod, upto, cnt3);
23     int res = (ll) a * modinv(b, mod) % mod * modinv(c, mod) % mod * powmod(p, cnt1 - cnt2 - cnt3, mod) % mod;
24     return res;
25 }
26 // Lucas's Theorem (p prime, m_i, n_i base p repr. of m, n): binom(m,n)==product(binom(m_i,n_i)) (mod p)

```



## 4 Graphen

### 4.1 Maximum Bipartite Matching

```

1 // run time:  $O(n \cdot \min(\text{ans}^2, |E|))$ , where  $n$  is the size of the left side
2 vector<int> madj[1001]; // adjacency list
3 int pairs[1001]; // for every node, stores the matching node on the other side or -1
4 bool vis[1001];
5 bool dfs(int i) {
6     if (vis[i]) return 0;
7     vis[i] = 1;
8     foreach(it, madj[i]) {
9         if (pairs[*it] < 0 || dfs(pairs[*it])) {
10             pairs[*it] = i, pairs[i] = *it;
11             return 1;
12         }
13     }
14     return 0;
15 }
16 int kuhn(int n) { // n = nodes on left side (numbered 0..n-1)
17     clr(pairs, -1); // to accelerate, just initialize with a greedy matching
18     int ans = 0;
19     rep(i, 0, n) {
20         clr(vis, 0);
21         ans += dfs(i);
22     }
23     return ans;
24 }

```

### 4.2 Maximaler Fluss (FF + Capacity Scaling)

```

1 // FF with cap scaling,  $O(m^2 \log C)$ 
2 const int MAXN = 190000, MAXC = 1<<29;
3 struct edge { int dest, capacity, rev; };
4 vector<edge> adj[MAXN];
5 int vis[MAXN], target, iter, cap;
6
7 void addedge(int x, int y, int c) {
8     adj[x].push_back(edge {y, c, (int)adj[y].size()});
9     adj[y].push_back(edge {x, 0, (int)adj[x].size() - 1});
10 }
11
12 bool dfs(int x) {
13     if (x == target) return 1;
14     if (vis[x] == iter) return 0;
15     vis[x] = iter;
16     for (edge& e: adj[x])
17         if (e.capacity >= cap && dfs(e.dest)) {
18             e.capacity -= cap;
19             adj[e.dest][e.rev].capacity += cap;
20             return 1;
21         }
22     return 0;
23 }
24
25 int maxflow(int S, int T) {
26     cap = MAXC, target = T;
27     int flow = 0;
28     while(cap) {
29         while(++iter, dfs(S))
30             flow += cap;
31         cap /= 2;
32     }
33     return flow;
34 }

```

### 4.3 Min-Cost-Max-Flow

```

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

```

```

10 } edges[maxm<<1];
11 int graph[maxn], queue[maxn], pre[maxn], con[maxn], n, m, source, target, top;
12 bool inq[maxn];
13 Capttype maxflow;
14 Valtype mincost, dis[maxn];
15 MinCostFlow() { memset(graph, -1, sizeof(graph)); top = 0; }
16 inline int inverse(int x) {return 1+((x>>1)<<2)-x; }
17 inline int addedge(int u, int v, Capttype c, Valtype w) { // add a directed edge
18     edges[top].value = w; edges[top].flow = c; edges[top].node = v;
19     edges[top].next = graph[u]; graph[u] = top++;
20     edges[top].value = -w; edges[top].flow = 0; edges[top].node = u;
21     edges[top].next = graph[v]; graph[v] = top++;
22     return top-2;
23 }
24 bool SPFA() { // Bellmanford, also works with negative edge weight.
25     int point, nod, now, head = 0, tail = 1;
26     memset(pre, -1, sizeof(pre));
27     memset(inq, 0, sizeof(inq));
28     memset(dis, 0x7f, sizeof(dis));
29     dis[source] = 0; queue[0] = source; pre[source] = source; inq[source] = true;
30     while (head!=tail) {
31         now = queue[head++]; point = graph[now]; inq[now] = false; head %= maxn;
32         while (point != -1) {
33             nod = edges[point].node;
34             if (edges[point].flow>0 && dis[nod]>dis[now]+edges[point].value) {
35                 dis[nod] = dis[now] + edges[point].value;
36                 pre[nod] = now;
37                 con[nod] = point;
38                 if (!inq[nod]) {
39                     inq[nod] = true;
40                     queue[tail++] = nod;
41                     tail %= maxn;
42                 }
43             }
44             point = edges[point].next;
45         }
46     }
47     return pre[target]!=-1; //&& dis[target]<=0; // for min-cost rather than max-flow
48 }
49 void extend()
50 {
51     Capttype w = flowlimit;
52     for (int u = target; pre[u]!=u; u = pre[u])
53         w = min(w, edges[con[u]].flow);
54     maxflow += w;
55     mincost += dis[target]*w;
56     for (int u = target; pre[u]!=u; u = pre[u]) {
57         edges[con[u]].flow-=w;
58         edges[inverse(con[u])].flow+=w;
59     }
60 }
61 void mincostflow() {
62     maxflow = 0; mincost = 0;
63     while (SPFA()) extend();
64 }

```

## 4.4 Value of Maximum Matching

```

1 const int N=200, MOD=1000000007, I=10;
2 int n, adj[N][N], a[N][N];
3 int rank() {
4     int r = 0;
5     rep(j, 0, n) {
6         int k = r;
7         while (k < n && !a[k][j]) ++k;
8         if (k == n) continue;
9         swap(a[r], a[k]);
10        int inv = powmod(a[r][j], MOD - 2);
11        rep(i, j, n)
12            a[r][i] = 1LL * a[r][i] * inv % MOD;
13        rep(u, r+1, n) rep(v, j, n)
14            a[u][v] = (a[u][v] - 1LL * a[r][v] * a[u][j] % MOD + MOD) % MOD;
15        ++r;
16    }
17    return r;
18 }
19 // failure probability = (n / MOD)^I
20 int max_matching() {
21     int ans = 0;
22     rep(_, 0, I) {

```

```

23     rep(i,0,n) rep(j,0,i)
24         if (adj[i][j]) {
25             a[i][j] = rand() % (MOD - 1) + 1;
26             a[j][i] = MOD - a[i][j];
27         }
28     ans = max(ans, rank()/2);
29 }
30 return ans;
31 }

```

## 4.5 SCC + 2-SAT

```

1  const int maxn = 10010; // 2-sat: maxn = 2*maxvars
2  vector<int> adj[maxn], radj[maxn];
3  bool vis[maxn];
4  int col, color[maxn];
5  vector<int> bycol[maxn];
6  vector<int> st;
7
8  void init() { rep(i,0,maxn) adj[i].clear(), radj[i].clear(); }
9  void dfs(int u, vector<int> adj[]) {
10     if (vis[u]) return;
11     vis[u] = 1;
12     foreach(it,adj[u]) dfs(*it, adj);
13     if (col) {
14         color[u] = col;
15         bycol[col].pb(u);
16     } else st.pb(u);
17 }
18 // this computes SCCs, outputs them in bycol, in topological order
19 void kosaraju(int n) { // n = number of nodes
20     st.clear();
21     clr(vis,0);
22     col=0;
23     rep(i,0,n) dfs(i,adj);
24     clr(vis,0);
25     clr(color,0);
26     while(!st.empty()) {
27         bycol[++col].clear();
28         int x = st.back(); st.pop_back();
29         if(color[x]) continue;
30         dfs(x, radj);
31     }
32 }
33 // 2-SAT
34 int assign[maxn]; // for 2-sat only
35 int var(int x) { return x<<1; }
36 bool solvable(int vars) {
37     kosaraju(2*vars);
38     rep(i,0,vars) if (color[var(i)] == color[1^var(i)]) return 0;
39     return 1;
40 }
41 void assign_vars() {
42     clr(assign,0);
43     rep(c,1,col+1) {
44         foreach(it,bycol[c]) {
45             int v = *it >> 1;
46             bool neg = *it&1;
47             if (assign[v]) continue;
48             assign[v] = neg?1:-1;
49         }
50     }
51 }
52 void add_impl(int v1, int v2) { adj[v1].push_back(v2); radj[v2].push_back(v1); }
53 void add_equiv(int v1, int v2) { add_impl(v1, v2); add_impl(v2, v1); }
54 void add_or(int v1, int v2) { add_impl(1^v1, v2); add_impl(1^v2, v1); }
55 void add_xor(int v1, int v2) { add_or(v1, v2); add_or(1^v1, 1^v2); }
56 void add_true(int v1) { add_impl(1^v1, v1); }
57 void add_and(int v1, int v2) { add_true(v1); add_true(v2); }
58
59 int parse(int i) {
60     if (i>0) return var(i-1);
61     else return 1^var(-i-1);
62 }
63 int main() {
64     int n, m; cin >> n >> m; // m = number of clauses to follow
65     while (m--) {
66         string op; int x, y; cin >> op >> x >> y;
67         x = parse(x);
68         y = parse(y);

```

```

69     if (op == "or") add_or(x, y);
70     if (op == "and") add_and(x, y);
71     if (op == "xor") add_xor(x, y);
72     if (op == "imp") add_impl(x, y);
73     if (op == "equiv") add_equiv(x, y);
74 }
75 if (!solvable(n)) {
76     cout << "Impossible" << endl; return 0;
77 }
78 assign_vars();
79 rep(i,0,n) cout << ((assign[i]>0)?(i+1):-i-1) << endl;
80 }

```

## 5 Geometrie

### 5.1 Verschiedenes

```

1  using D=long double;
2  using P=complex<D>;
3  using L=vector<P>;
4  using G=vector<P>;
5  const D eps=1e-12, inf=1e15, pi=acos(-1), e=exp(1.);
6
7  D sq(D x) { return x*x; }
8  D rem(D x, D y) { return fmod(fmod(x,y)+y,y); }
9  D rtod(D rad) { return rad*180/pi; }
10 D dtor(D deg) { return deg*pi/180; }
11 int sgn(D x) { return (x > eps) - (x < -eps); }
12 // when doing printf("%.Xf", x), fix '-0' output to '0'.
13 D fixzero(D x, int d) { return (x>0 || x<=-5/pow(10,d+1)) ? x:0; }
14
15 namespace std {
16     bool operator<(const P& a, const P& b) {
17         return mk(real(a), imag(a)) < mk(real(b), imag(b));
18     }
19 }
20
21 D cross(P a, P b) { return imag(conj(a) * b); }
22 D cross(P a, P b, P c) { return cross(b-a, c-a); }
23 D dot(P a, P b) { return real(conj(a) * b); }
24 P scale(P a, D len) { return a * (len/abs(a)); }
25 P rotate(P p, D ang) { return p * polar(D(1), ang); }
26 D angle(P a, P b) { return arg(b) - arg(a); }
27
28 int ccw(P a, P b, P c) {
29     b -= a; c -= a;
30     if (cross(b, c) > eps) return +1; // counter clockwise
31     if (cross(b, c) < -eps) return -1; // clockwise
32     if (dot(b, c) < 0) return +2; // c--a--b on line
33     if (norm(b) < norm(c)) return -2; // a--b--c on line
34     return 0;
35 }
36
37 G dummy;
38 L line(P a, P b) {
39     L res; res.pb(a); res.pb(b); return res;
40 }
41 P dir(const L& l) { return l[1]-l[0]; }
42
43 D project(P e, P x) { return dot(e,x) / norm(e); }
44 P pedal(const L& l, P p) { return l[1] + dir(l) * project(dir(l), p-l[1]); }
45 int intersectLL(const L& l, const L& m) {
46     if (abs(cross(l[1]-l[0], m[1]-m[0])) > eps) return 1; // non-parallel
47     if (abs(cross(l[1]-l[0], m[0]-l[0])) < eps) return -1; // same line
48     return 0;
49 }
50 bool intersectLS(const L& l, const L& s) {
51     return cross(dir(l), s[0]-l[0]) * // s[0] is left of l
52            cross(dir(l), s[1]-l[0]) < eps; // s[1] is right of l
53 }
54 bool intersectLP(const L& l, const P& p) {
55     return abs(cross(l[1]-p, l[0]-p)) < eps;
56 }
57 bool intersectSS(const L& s, const L& t) {
58     return sgn(ccw(s[0],s[1],t[0]) * ccw(s[0],s[1],t[1])) <= 0 &&
59            sgn(ccw(t[0],t[1],s[0]) * ccw(t[0],t[1],s[1])) <= 0;
60 }
61 bool intersectSP(const L& s, const P& p) {
62     return abs(s[0]-p)+abs(s[1]-p)-abs(s[1]-s[0]) < eps; // triangle inequality
63 }

```

```

64 P reflection(const L& l, P p) {
65     return p + P(2,0) * (pedal(l, p) - p);
66 }
67 D distanceLP(const L& l, P p) {
68     return abs(p - pedal(l, p));
69 }
70 D distanceLL(const L& l, const L& m) {
71     return intersectLL(l, m) ? 0 : distanceLP(l, m[0]);
72 }
73 D distanceLS(const L& l, const L& s) {
74     if (intersectLS(l, s)) return 0;
75     return min(distanceLP(l, s[0]), distanceLP(l, s[1]));
76 }
77 D distanceSP(const L& s, P p) {
78     P r = pedal(s, p);
79     if (intersectSP(s, r)) return abs(r - p);
80     return min(abs(s[0] - p), abs(s[1] - p));
81 }
82 D distanceSS(const L& s, const L& t) {
83     if (intersectSS(s, t)) return 0;
84     return min(min(distanceSP(s, t[0]), distanceSP(s, t[1])),
85               min(distanceSP(t, s[0]), distanceSP(t, s[1])));
86 }
87 P crosspoint(const L& l, const L& m) { // return intersection point
88     D A = cross(dir(l), dir(m));
89     D B = cross(dir(l), l[1] - m[0]);
90     return m[0] + B / A * dir(m);
91 }
92 L bisector(P a, P b) {
93     P A = (a+b)*P(0.5,0);
94     return line(A, A+(b-a)*P(0,1));
95 }
96
97 #define next(g,i) g[(i+1)%g.size()]
98 #define prev(g,i) g[(i+g.size()-1)%g.size()]
99 L edge(const G& g, int i) { return line(g[i], next(g,i)); }
100 D area(const G& g) {
101     D A = 0;
102     rep(i,0,g.size())
103         A += cross(g[i], next(g,i));
104     return abs(A/2);
105 }
106
107 // intersect with half-plane left of l[0] -> l[1]
108 G convex_cut(const G& g, const L& l) {
109     G Q;
110     rep(i,0,g.size()) {
111         P A = g[i], B = next(g,i);
112         if (ccw(l[0], l[1], A) != -1) Q.pb(A);
113         if (ccw(l[0], l[1], A)*ccw(l[0], l[1], B) < 0)
114             Q.pb(crosspoint(line(A, B), l));
115     }
116     return Q;
117 }
118 bool convex_contain(const G& g, P p) { // check if point is inside convex polygon
119     rep(i,0,g.size())
120         if (ccw(g[i], next(g, i), p) == -1) return 0;
121     return 1;
122 }
123 G convex_intersect(G a, G b) { // intersect two convex polygons
124     rep(i,0,b.size())
125         a = convex_cut(a, edge(b, i));
126     return a;
127 }
128 void triangulate(G g, vector<G>& res) { // triangulate a simple polygon
129     while (g.size() > 3) {
130         bool found = 0;
131         rep(i,0,g.size()) {
132             if (ccw(prev(g,i), g[i], next(g,i)) != +1) continue;
133             G tri;
134             tri.pb(prev(g,i));
135             tri.pb(g[i]);
136             tri.pb(next(g,i));
137             bool valid = 1;
138             rep(j,0,g.size()) {
139                 if ((j+1)%g.size() == i || j == i || j == (i+1)%g.size()) continue;
140                 if (convex_contain(tri, g[j])) {
141                     valid = 0;
142                     break;
143                 }
144             }
145             if (!valid) continue;

```

```

146         res.pb(tri);
147         g.erase(g.begin() + i);
148         found = 1; break;
149     }
150     assert(found);
151 }
152 res.pb(g);
153 }
154 void graham_step(G& a, G& st, int i, int bot) {
155     while (st.size()>bot && sgn(cross(*(st.end()-2), st.back(), a[i]))<=0)
156         st.pop_back();
157     st.pb(a[i]);
158 }
159 bool cmpY(P a, P b) { return mk(imag(a),real(a)) < mk(imag(b),real(b)); }
160 G graham_scan(const G& points) { // will return points in ccw order
161     // special case: all points coincide, algo might return point twice
162     G a = points; sort(all(a),cmpY);
163     int n = a.size();
164     if (n<=1) return a;
165     G st; st.pb(a[0]); st.pb(a[1]);
166     for (int i = 2; i < n; i++) graham_step(a,st,i,1);
167     int mid = st.size();
168     for (int i = n - 2; i >= 0; i--) graham_step(a,st,i,mid);
169     while (st.size() > 1 && !sgn(abs(st.back() - st.front()))) st.pop_back();
170     return st;
171 }
172 G gift_wrap(const G& points) { // will return points in clockwise order
173     // special case: duplicate points, not sure what happens then
174     int n = points.size();
175     if (n<=2) return points;
176     G res;
177     P nxt, p = *min_element(all(points), [](const P& a, const P& b){
178         return real(a) < real(b);
179     });
180     do {
181         res.pb(p);
182         nxt = points[0];
183         for (auto& q: points)
184             if (abs(p - q) > eps && (abs(p - nxt) < eps || ccw(p, nxt, q) == 1))
185                 nxt = q;
186         p = nxt;
187     } while (nxt != *begin(res));
188     return res;
189 }
190 G voronoi_cell(G g, const vector<P> &v, int s) {
191     rep(i,0,v.size())
192         if (i!=s)
193             g = convex_cut(g, bisector(v[s], v[i]));
194     return g;
195 }
196 const int ray_iters = 20;
197 bool simple_contain(const G& g, P p) { // check if point is inside simple polygon
198     int yes = 0;
199     rep(_,0,ray_iters) {
200         D angle = 2*pi * (D)rand() / RAND_MAX;
201         P dir = rotate(P(1,0), angle);
202         L s = line(p, p + dir);
203         int cnt = 0;
204         rep(i,0,g.size()) {
205             if (intersectSS(edge(g, i), s)) cnt++;
206         }
207         yes += cnt%2;
208     }
209     return yes > ray_iters/2;
210 }
211 bool intersectGG(const G& g1, const G& g2) {
212     if (convex_contain(g1, g2[0])) return 1;
213     if (convex_contain(g2, g1[0])) return 1;
214     rep(i,0,g1.size()) rep(j,0,g2.size()) {
215         if (intersectSS(edge(g1, i), edge(g2, j))) return 1;
216     }
217     return 0;
218 }
219 D distanceGP(const G& g, P p) {
220     if (convex_contain(g, p)) return 0;
221     D res = inf;
222     rep(i,0,g.size())
223         res = min(res, distanceSP(edge(g, i), p));
224     return res;
225 }
226 P centroid(const G& v) {
227     D S = 0;

```

```

228 P res;
229 rep(i,0,v.size()) {
230     D tmp = cross(v[i], next(v,i));
231     S += tmp;
232     res += (v[i] + next(v,i)) * tmp;
233 }
234 S /= 2;
235 res /= 6*S;
236 return res;
237 }
238
239 struct C {
240     P p; D r;
241     C(P p, D r) : p(p),r(r) {}
242     C() {}
243 };
244 // intersect circle with line through (c.p + v * dst/abs(v)) "orthogonal" to the circle
245 // dst can be negative
246 G intersectCL2(const C& c, D dst, P v) {
247     G res;
248     P mid = c.p + v * (dst/abs(v));
249     if (sgn(abs(dst)-c.r) == 0) { res.pb(mid); return res; }
250     D h = sqrt(sq(c.r) - sq(dst));
251     P hi = scale(v * P(0,1), h);
252     res.pb(mid + hi); res.pb(mid - hi);
253     return res;
254 }
255 G intersectCL(const C& c, const L& l) {
256     if (intersectLP(l, c.p)) {
257         P h = scale(dir(l), c.r);
258         G res; res.pb(c.p + h); res.pb(c.p - h); return res;
259     }
260     P v = pedal(l, c.p) - c.p;
261     return intersectCL2(c, abs(v), v);
262 }
263 G intersectCS(const C& c, const L& s) {
264     G res1 = intersectCL(c,s), res2;
265     for(auto it: res1) if (intersectSP(s, it)) res2.pb(it);
266     return res2;
267 }
268 int intersectCC(const C& a, const C& b, G& res=dummy) {
269     D sum = a.r + b.r, diff = abs(a.r - b.r), dst = abs(a.p - b.p);
270     if (dst > sum + eps || dst < diff - eps) return 0;
271     if (max(dst, diff) < eps) { // same circle
272         if (a.r < eps) { res.pb(a.p); return 1; } // degenerate
273         return -1; // infinitely many
274     }
275     D p = (sq(a.r) - sq(b.r) + sq(dst))/(2*dst);
276     P ab = b.p - a.p;
277     res = intersectCL2(a, p, ab);
278     return res.size();
279 }
280
281 using P3 = valarray<D>;
282 P3 p3(D x=0, D y=0, D z=0) {
283     P3 res(3);
284     res[0]=x;res[1]=y;res[2]=z;
285     return res;
286 }
287 ostream& operator<<(ostream& out, const P3& x) {
288     return out << "(" << x[0]<<","<<x[1]<<","<<x[2]<<")";
289 }
290 P3 cross(const P3& a, const P3& b) {
291     P3 res;
292     rep(i,0,3) res[i]=a[(i+1)%3]*b[(i+2)%3]-a[(i+2)%3]*b[(i+1)%3];
293     return res;
294 }
295 D dot(const P3& a, const P3& b) {
296     return a[0]*b[0]+a[1]*b[1]+a[2]*b[2];
297 }
298 D norm(const P3& x) { return dot(x,x); }
299 D abs(const P3& x) { return sqrt(norm(x)); }
300 D project(const P3& e, const P3& x) { return dot(e,x) / norm(e); }
301 P project_plane(const P3& v, P3 w, const P3& p) {
302     w -= project(v,w)*v;
303     return P(dot(p,v)/abs(v), dot(p,w)/abs(w));
304 }
305
306 template <typename T, int N> struct Matrix {
307     T data[N][N];
308     Matrix<T,N>(T d=0) { rep(i,0,N) rep(j,0,N) data[i][j] = i==j?d:0; }
309     Matrix<T,N> operator+(const Matrix<T,N>& other) const {

```

```

310     Matrix res; rep(i,0,N) rep(j,0,N) res[i][j] = data[i][j] + other[i][j]; return res;
311 }
312 Matrix<T,N> operator*(const Matrix<T,N>& other) const {
313     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;
314 }
315 Matrix<T,N> transpose() const {
316     Matrix res; rep(i,0,N) rep(j,0,N) res[i][j] = data[j][i]; return res;
317 }
318 array<T,N> operator*(const array<T,N>& v) const {
319     array<T,N> res;
320     rep(i,0,N) rep(j,0,N) res[i] += data[i][j] * v[j];
321     return res;
322 }
323 const T* operator[](int i) const { return data[i]; }
324 T* operator[](int i) { return data[i]; }
325 };
326 template <typename T, int N> ostream& operator<<(ostream& out, Matrix<T,N> mat) {
327     rep(i,0,N) { rep(j,0,N) out << mat[i][j] << " "; cout << endl; } return out;
328 } // creates a rotation matrix around axis x (must be normalized). Rotation is
329 // counter-clockwise if you look in the inverse direction of x onto the origin
330 template<typename M> void create_rot_matrix(M& m, double x[3], double a) {
331     rep(i,0,3) rep(j,0,3) {
332         m[i][j] = x[i]*x[j]*(1-cos(a));
333         if (i == j) m[i][j] += cos(a);
334         else m[i][j] += x[(6-i-j)%3] * ((i == (2+j) % 3) ? -1 : 1) * sin(a);
335     }
336 }

```

## 5.2 Graham's Scan + max. Abstand

```

1  /* Runtime: O(n*log(n)). Find 2 farthest points in a set of points.
2   * Use graham algorithm to get the convex hull.
3   * Note: In extreme situation, when all points coincide, the program won't work
4   * probably. A prejudice of this situation may consequently be needed */
5  const int mn = 100005;
6  const double pi = acos(-1.0), eps = 1e-5;
7  struct point { double x, y; } a[mn];
8  int n, cn, st[mn];
9  inline bool cmp(const point &a, const point &b) {
10     if (a.y != b.y) return a.y < b.y; return a.x < b.x;
11 }
12 inline int dblcmp(const double &d) {
13     if (abs(d) < eps) return 0; return d < 0 ? -1 : 1;
14 }
15 inline double cross(const point &a, const point &b, const point &c) {
16     return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
17 }
18 inline double dis(const point &a, const point &b) {
19     double dx = a.x - b.x, dy = a.y - b.y;
20     return sqrt(dx * dx + dy * dy);
21 } // get the convex hull
22 void graham_scan() {
23     sort(a, a + n, cmp);
24     cn = -1;
25     st[++cn] = 0;
26     st[++cn] = 1;
27     for (int i = 2; i < n; i++) {
28         while (cn>0 && dblcmp(cross(a[st[cn-1]],a[st[cn]],a[i]))<=0) cn--;
29         st[++cn] = i;
30     }
31     int newtop = cn;
32     for (int i = n - 2; i >= 0; i--) {
33         while (cn>newtop && dblcmp(cross(a[st[cn-1]],a[st[cn]],a[i]))<=0) cn--;
34         st[++cn] = i;
35     }
36 }
37 inline int next(int x) { return x + 1 == cn ? 0 : x + 1; }
38 inline double angle(const point &a,const point &b,const point &c,const point &d){
39     double x1 = b.x - a.x, y1 = b.y - a.y, x2 = d.x - c.x, y2 = d.y - c.y;
40     double tc = (x1 * x2 + y1 * y2) / dis(a, b) / dis(c, d);
41     return acos(abs(tc) > 1.0 ? (tc > 0 ? 1 : -1) * 1.0 : tc);
42 }
43 void maintain(int &p1, int &p2, double &nowh, double &nowd) {
44     nowd = dis(a[st[p1]], a[st[next(p1)]]);
45     nowh = cross(a[st[p1]], a[st[next(p1)]], a[st[p2]]) / nowd;
46     while (1) {
47         double h = cross(a[st[p1]], a[st[next(p1)]], a[st[next(p2)]] / nowd;
48         if (dblcmp(h - nowh) > 0) {
49             nowh = h;
50             p2 = next(p2);

```



```

51         } else break;
52     }
53 }
54 double find_max() {
55     double suma = 0, nowh = 0, nowd = 0, ans = 0;
56     int p1 = 0, p2 = 1;
57     maintain(p1, p2, nowh, nowd);
58     while (dblcmp(suma - pi) <= 0) {
59         double t1 = angle(a[st[p1]], a[st[next(p1)]], a[st[next(p1)]],
60             a[st[next(next(p1))]]);
61         double t2 = angle(a[st[next(p1)]], a[st[p1]], a[st[p2]], a[st[next(p2)]]);
62         if (dblcmp(t1 - t2) <= 0) {
63             p1 = next(p1); suma += t1;
64         } else {
65             p1 = next(p1); swap(p1, p2); suma += t2;
66         }
67         maintain(p1, p2, nowh, nowd);
68         double d = dis(a[st[p1]], a[st[p2]]);
69         if (d > ans) ans = d;
70     }
71     return ans;
72 }
73 int main() {
74     while (scanf("%d", &n) != EOF && n) {
75         for (int i = 0; i < n; i++)
76             scanf("%lf%lf", &a[i].x, &a[i].y);
77         if (n == 2)
78             printf("%.2lf\n", dis(a[0], a[1]));
79         else {
80             graham_scan();
81             double mx = find_max();
82             printf("%.2lf\n", mx);
83         }
84     }
85     return 0;
86 }

```

## 6 Datenstrukturen

### 6.1 STL order statistics tree

```

1 #include <bits/stdc++.h>
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 using namespace std; using namespace __gnu_pbds;
5 typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> Tree;
6 int main() {
7     Tree X;
8     for (int i = 1; i <= 16; i <= 1) X.insert(i); // { 1, 2, 4, 8, 16 };
9     cout << *X.find_by_order(3) << endl; // => 8
10    cout << X.order_of_key(10) << endl; // => 4 = successor of 10 = min i such that X[i] >= 10
11 }

```

### 6.2 Skew Heaps (meldable priority queue)

```

1 /* The simplest meldable priority queues: Skew Heap
2 Merging (destroying both trees), inserting, deleting min: O(logn) amortised;*/
3 struct node{
4     int key;
5     node *lc,*rc;
6     node(int k):key(k),lc(0),rc(0){}
7 }*root=0;
8 int size=0;
9 node* merge(node* x, node* y){
10     if(!x)return y;
11     if(!y)return x;
12     if(x->key > y->key)swap(x,y);
13     x->rc=merge(x->rc,y);
14     swap(x->lc,x->rc);
15     return x;
16 }
17 void insert(int x) { root=merge(root,new node(x)); size++;}
18 int delmin() {
19     if(!root)return -1;
20     int ret=root->key;
21     node *troot=merge(root->lc,root->rc);
22     delete root;
23     root=troot;

```

```

24     size--;
25     return ret;
26 }

```

## 6.3 Treap

```

1  struct Node {
2      int val, prio, size;
3      Node* child[2];
4      void apply() { // apply lazy actions and push them down
5          }
6      void maintain() {
7          size = 1;
8          rep(i,0,2) size += child[i] ? child[i]->size : 0;
9      }
10 };
11 pair<Node*, Node*> split(Node* n, int val) { // returns (< val, >= val)
12     if (!n) return {0,0};
13     n->apply();
14     Node&& c = n->child[val > n->val];
15     auto sub = split(c, val);
16     if (val > n->val) { c = sub.fst; n->maintain(); return mk(n, sub.snd); }
17     else             { c = sub.snd; n->maintain(); return mk(sub.fst, n); }
18 }
19 Node* merge(Node* l, Node* r) {
20     if (!l || !r) return l ? l : r;
21     if (l->prio > r->prio) {
22         l->apply();
23         l->child[1] = merge(l->child[1], r);
24         l->maintain();
25         return l;
26     } else {
27         r->apply();
28         r->child[0] = merge(l, r->child[0]);
29         r->maintain();
30         return r;
31     }
32 }
33 Node* insert(Node* n, int val) {
34     auto sub = split(n, val);
35     Node* x = new Node { val, rand(), 1 };
36     return merge(merge(sub.fst, x), sub.snd);
37 }
38 Node* remove(Node* n, int val) {
39     if (!n) return 0;
40     n->apply();
41     if (val == n->val)
42         return merge(n->child[0], n->child[1]);
43     Node&& c = n->child[val > n->val];
44     c = remove(c, val);
45     n->maintain();
46     return n;
47 }

```

## 6.4 Fenwick Tree

```

1  const int n = 10; // ALL INDICES START AT 1 WITH THIS CODE!!
2
3  // mode 1: update indices, read prefixes
4  void update_idx(int tree[], int i, int val) { // v[i] += val
5      for (; i <= n; i += i & -i) tree[i] += val;
6  }
7  int read_prefix(int tree[], int i) { // get sum v[1..i]
8      int sum = 0;
9      for (; i > 0; i -= i & -i) sum += tree[i];
10     return sum;
11 }
12 int kth(int k) { // find kth element in tree (1-based index)
13     int ans = 0;
14     for (int i = maxl; i >= 0; --i) // maxl = largest i s.t. (1<<i) <= n
15         if (ans + (1<<i) <= N && tree[ans + (1<<i)] < k) {
16             ans += 1<<i;
17             k -= tree[ans];
18         }
19     return ans+1;
20 }
21
22 // mode 2: update prefixes, read indices

```

```

23 void update_prefix(int tree[], int i, int val) { // v[l..i] += val
24     for (; i > 0; i -= i & -i) tree[i] += val;
25 }
26 int read_idx(int tree[], int i) { // get v[i]
27     int sum = 0;
28     for (; i <= n; i += i & -i) sum += tree[i];
29     return sum;
30 }
31
32 // mode 3: range-update range-query (using point-wise of linear functions)
33 const int maxn = 100100;
34 int n;
35 ll mul[maxn], add[maxn];
36
37 void update_idx(ll tree[], int x, ll val) {
38     for (int i = x; i <= n; i += i & -i) tree[i] += val;
39 }
40 void update_prefix(int x, ll val) { // v[x] += val
41     update_idx(mul, 1, val);
42     update_idx(mul, x + 1, -val);
43     update_idx(add, x + 1, x * val);
44 }
45 ll read_prefix(int x) { // get sum v[l..x]
46     ll a = 0, b = 0;
47     for (int i = x; i > 0; i -= i & -i) a += mul[i], b += add[i];
48     return a * x + b;
49 }
50 void update_range(int l, int r, ll val) { // v[l..r] += val
51     update_prefix(l - 1, -val);
52     update_prefix(r, val);
53 }
54 ll read_range(int l, int r) { // get sum v[l..r]
55     return read_prefix(r) - read_prefix(l - 1);
56 }

```

## 6.5 Simple tree aggregations

```

1 void maintain(int x, int exclude) {
2     g[x] = 1;
3     for (int y: adj[x]) {
4         if (y == exclude) continue;
5         g[x] += g[y];
6     }
7 }
8 // build initial data structures with fixed root
9 void dfs1(int x, int from) {
10     for (int y: adj[x]) if (y != from)
11         dfs1(y, x);
12     maintain(x, from);
13 }
14 // inspect data structures with x as root
15 void dfs2(int x, int from) {
16     for (int y: adj[x]) if (y != from) {
17         maintain(x, y);
18         maintain(y, -1);
19         dfs2(y, x);
20     }
21     maintain(x, from);
22 }

```

## 7 DP optimization

### 7.1 Convex hull (monotonic insert)

```

1 // convex hull, minimum
2 vector<ll> M, B;
3 int ptr;
4 bool bad(int a, int b, int c) {
5     // use deterministic computation with long long if sufficient
6     return (long double) (B[c] - B[a]) * (M[a] - M[b]) < (long double) (B[b] - B[a]) * (M[a] - M[c]);
7 }
8 // insert with non-increasing m
9 void insert(ll m, ll b) {
10     M.push_back(m);
11     B.push_back(b);
12     while (M.size() >= 3 && bad(M.size()-3, M.size()-2, M.size()-1)) {
13         M.erase(M.end()-2);
14         B.erase(B.end()-2);
15     }
16 }

```

```

15 }
16 }
17 ll get(int i, ll x) {
18     return M[i]*x + B[i];
19 }
20 // query with non-decreasing x
21 ll query(ll x) {
22     ptr=min((int)M.size()-1,ptr);
23     while (ptr<M.size()-1 && get(ptr+1,x)<get(ptr,x))
24         ptr++;
25     return get(ptr,x);
26 }

```

## 7.2 Dynamic convex hull

```

1 const ll is_query = -(1LL<<62);
2 struct Line {
3     ll m, b;
4     mutable function<const Line*> succ;
5     bool operator<(const Line& rhs) const {
6         if (rhs.b != is_query) return m < rhs.m;
7         const Line* s = succ();
8         if (!s) return 0;
9         ll x = rhs.m;
10        return b - s->b < (s->m - m) * x;
11    }
12 };
13 struct HullDynamic : public multiset<Line> { // will maintain upper hull for maximum
14     bool bad(iterator y) {
15         auto z = next(y);
16         if (y == begin()) {
17             if (z == end()) return 0;
18             return y->m == z->m && y->b <= z->b;
19         }
20         auto x = prev(y);
21         if (z == end()) return y->m == x->m && y->b <= x->b;
22         return (x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x->m);
23     }
24     void insert_line(ll m, ll b) {
25         auto y = insert({ m, b });
26         y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
27         if (bad(y)) { erase(y); return; }
28         while (next(y) != end() && bad(next(y))) erase(next(y));
29         while (y != begin() && bad(prev(y))) erase(prev(y));
30     }
31     ll eval(ll x) {
32         auto l = *lower_bound((Line) { x, is_query });
33         return l.m * x + l.b;
34     }
35 };

```

# 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.)	$m \equiv 1 \pmod{\frac{L}{p}},$
HT min steps A to C clockw.	$Q_n = 2R_{n-1} + 1$	$L(n) = lcm(1, \dots, n), p$ prime $\in [\frac{n}{2}, n]$	$m \equiv j + 1 - n \pmod{p}$
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 =  \det(C - 1r - 1c) $	Farey Seq	$mn' - m'n = -1$
#heaps of a tree (keys: 1..n)	$\frac{(n-1)!}{\prod_{i \neq root} size(i)}$	#ways $0 \rightarrow m$ in $n$ steps (never $< 0$ )	$\frac{m+1}{\frac{n+m}{2}+1} \left(\frac{n}{2}\right)$
#seq $\langle a_0, \dots, a_{mn} \rangle$ of 1's and $(1-m)$ 's with sum $+1 = \binom{mn+1}{n}$	$\frac{1}{mn+1} = \binom{mn}{n} \frac{1}{(m-1)n+1}$		$D_n = nD_{n-1} + (-1)^n$

## 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 $ x/y  < 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 $n$	$\sum_{k \leq m} \binom{r}{k} (r-k) = \frac{m+1}{2} \binom{r}{m+1}$ , 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 $m, n$	$\binom{\binom{k}{2}}{2} = 3 \binom{k+1}{4}$   $\sum_{i=0}^n \binom{n}{i}^2 = \binom{2n}{n}$	$\sum_k \binom{l}{m+k} \binom{s}{n+k} = \binom{l+s}{l-m+n}$ int $l \geq 0$ , int $m, n$
$\sum_k \binom{n}{2k} = 2^{n-\text{even}(n)}$	$lcm_{i=0}^n \binom{n}{i} = \frac{L(n+1)}{n+1}$	$S(n, 1) = S(n, n) = n \Rightarrow S(n, k) = \binom{n+1}{k} - \binom{n-1}{k-1}$
$\sum_{i=1}^n \binom{n}{i} F_i = F_{2n}$ , $F_n = n$ -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	$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1, \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0, \begin{bmatrix} n \\ k \end{bmatrix} = (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$	#perms of $n$ objs with exactly $k$ cycles
Stirling 2nd kind	$\left\{ \begin{smallmatrix} n \\ 1 \end{smallmatrix} \right\} = \left\{ \begin{smallmatrix} n \\ n \end{smallmatrix} \right\} = 1, \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = k \left\{ \begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right\}$	#ways to partition $n$ objs into $k$ nonempty sets
Euler	$\langle n \rangle = \langle n-1 \rangle = 1, \langle n \rangle = (k+1) \langle n-1 \rangle + (n-k) \langle n-1 \rangle$	#perms of $n$ objs with exactly $k$ ascents
Euler 2nd Order	$\langle\langle n \rangle\rangle = (k+1) \langle\langle n-1 \rangle\rangle + (2n-k-1) \langle\langle n-1 \rangle\rangle$	#perms of $1, 1, 2, 2, \dots, n, n$ with exactly $k$ ascents
Bell	$B_1 = 1, B_n = \sum_{k=0}^{n-1} B_k \binom{n-1}{k} = \sum_{k=0}^n \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$	#partitions of $1..n$ (Stirling 2nd, no limit on $k$ )

The Twelffold Way (Putting  $n$  balls into  $k$  boxes)

Balls Boxes	same same	distinct same	same distinct	distinct distinct	Remarks
-	$p_k(n)$	$\sum_{i=0}^k \left\{ \begin{smallmatrix} n \\ i \end{smallmatrix} \right\}$	$\binom{n+k-1}{k-1}$	$k^n$	$p_k(n)$ : #partitions of $n$ into $\leq k$ positive parts
size $\geq 1$	$p(n, k)$	$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$	$\binom{n-1}{k-1}$	$k! \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$	$p(n, k)$ : #partitions of $n$ into $k$ positive parts (NrPartitions)
size $\leq 1$	$[n \leq k]$	$[n \leq k]$	$\binom{k}{n}$	$n! \binom{n}{k}$	$[cond]$ : 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  $L = \frac{1}{|G|} \sum_{g_i \in G} m^{c(g_i)}$ .  $G$ : 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	$(n^{30} + 15n^{16} + 20n^{10} + 24n^6)/60$	<i>This row may be wrong.</i>

**Exponential families (unlabelled):**  $h(n)$  = number of possible hands of weight  $n$ ,  $h(n, k)$  = number of hands of weight  $n$  with  $k$  cards,  $d(n)$  = number of cards of weight  $n$ . Then  $k \cdot h(n, k) = \sum_{r, m \geq 1} h(n - rm, k - m) \cdot d(r)$  and  $n \cdot h(n) = \sum_{m \geq 1} h(n - m) \cdot \sum_{r|m} r \cdot d(r)$ .

## 8.2 Number Theory

## Classical Theorems

exp of $p$ in $n!$ is $\sum_{i \geq 1} \lfloor \frac{n}{p^i} \rfloor$	$p_n \sim n \log n; \quad \forall_{n > 1} \exists_{n < p < 2n} : p \text{ is prime}$	$\pi(x) \sim \frac{x}{\log x}; \quad \text{Norm}(\alpha\beta) = \text{Norm}(\alpha) \cdot \text{Norm}(\beta)$
$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$	$a \equiv b \pmod{x, y} \Rightarrow a \equiv b \pmod{\text{lcm}(x, y)}$	All prime factors of $2^{2^n} + 1$ have form $2^{n+2}k + 1$
$(2^a - 1, 2^b - 1) = 2^{(a, b)} - 1$	$ac \equiv bc \pmod{m} \Rightarrow a \equiv b \pmod{\frac{m}{\gcd(c, m)}}$	$n$ -plygn drawable $\Leftrightarrow n = 2^k \prod F_i, F_i \text{ fermatNum}$
$p_i$ is prime, $\prod_{p_i \leq n} p_i < 4^n$	$W \equiv d + [2.6m - 0.2] - 2C + Y + [\frac{Y}{4}] + [\frac{C}{4}] \pmod{7}$	$m = 11, 12, 1$ for Ja, Fe, Ma. J&F $\in$ lastyear
Fibonacci period (pisano): $\gcd(n, m) = 1 \Rightarrow \pi(n, m) = \text{lcm}(\pi(n), \pi(m)), \pi(p^e)   p^{e-1} \pi(p), \pi(5) = 20, \pi(p \neq 5)   p - 1 \text{ or } 2(p + 1)$		

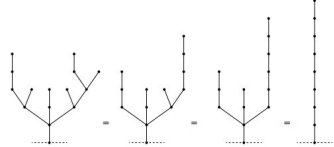
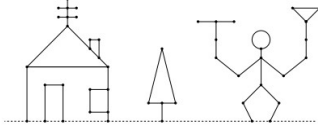
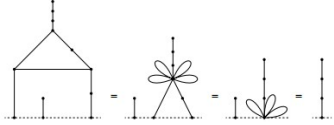
## Classical Theorems

$p$ prime $\Leftrightarrow (p-1)! \equiv -1 \pmod{p}$	$a \perp m \Rightarrow a^{\phi(m)} \equiv 1 \pmod{m}$	Min general idx $\lambda(n): \forall_a : a^{\lambda(n)} \equiv 1 \pmod{n}$
$\sum_{d n} \phi(d) = \sum_{d n} \phi(n/d) = n$	$\sum_{m \perp n, m < n} m = \frac{n\phi(n)}{2}$	$\sum_{i=1}^n \sigma_0(i) = 2 \sum_{i=1}^{\lfloor \sqrt{n} \rfloor} [n/i] - [\sqrt{n}]^2$
$(\sum_{d n} \sigma_0(d))^2 = \sum_{d n} \sigma_0(d)^3$	$\sum_{d n} n \sigma_1(d)/d = \sum_{d n} d \sigma_0(d)$	$[\sqrt{n}]$ Newton: $y = [\frac{x + [n/x]}{2}], x_0 = 2^{\lceil \frac{\log_2(n)+2}{2} \rceil}$
$\sigma_0(p_1^{e_1} \cdots p_s^{e_s}) = \prod_{i=1}^s (e_i + 1)$	$\sigma_1(p_1^{e_1} \cdots p_s^{e_s}) = \prod_{i=1}^s \frac{p_i^{e_i+1} - 1}{p_i - 1}$	$r_1 = 4, r_k \equiv r_{k-1}^2 - 2 \pmod{M_p}, M_p \text{ prime} \Leftrightarrow r_{p-1} \equiv 0 \pmod{M_p}$
$\mu(p_1 p_2 \cdots p_s) = (-1)^s, \text{ else } 0$	$\sum_{d n} \mu(d) = 1 \text{ if } n = 1, \text{ else } 0$	$F(n) = \sum_{d n} f(d) \Leftrightarrow f(n) = \sum_{d n} \mu(d) F(\frac{n}{d})$
$n = \sum_{d n} \mu(\frac{n}{d}) \sigma_1(d)$	$n = 2, 4, p^t, 2p^t \Leftrightarrow n \text{ has } p\text{-roots}$	$a \perp n, \text{ then } a^i \equiv a^j \pmod{n} \Leftrightarrow i \equiv j \pmod{\text{ord}_n(a)}$
$1 = \sum_{d n} \mu(\frac{n}{d}) \sigma_0(d)$	$r = \text{ord}_n(a), \text{ord}_n(a^u) = \frac{r}{\gcd(r, u)}$	$r$ p_root of $n$ , then $r^u$ is p_root of $n \Leftrightarrow u \perp \phi(n)$
$\text{ord}_n(a) = \text{ord}_n(a^{-1})$	$r$ p_root of $n \Leftrightarrow r^{-1}$ p_root of $n$	$n$ has p_roots $\Leftrightarrow n$ has $\phi(\phi(n))$ p_roots
$a^n \equiv a^{\phi(m) + n\% \phi(m)} \pmod{m}, n \text{ big}$	$\lambda(2^t) = 2^{t-2}, \lambda(p^t) = \phi(p^t) = (p-1)p^{t-1}, \lambda(2^{t_0} p_1^{t_1} \cdots p_m^{t_m}) = \text{lcm}(\lambda(2^{t_0}), \phi(p_1^{t_1}), \dots, \phi(p_m^{t_m}))$	
$(\frac{a}{p}) \equiv a^{(p-1)/2} \pmod{p}$	Legendre sym $(\frac{a}{p}) = 1$ if $a$ is quad residue $\%p$ ; $-1$ if $a$ is non-residue; $0$ if $a = 0$	
$a \equiv b \pmod{p} \Rightarrow (\frac{a}{p}) = (\frac{b}{p})$	$(\frac{a}{p})(\frac{b}{p}) = (\frac{ab}{p}); (\frac{a^2}{p}) = 1$	$a \perp p, s \text{ from } a, 2a, \dots, \frac{p-1}{2}a \pmod{p} \text{ are } > \frac{p}{2} \Rightarrow (\frac{a}{p}) = (-1)^s$
$(\frac{p}{q})(\frac{q}{p}) = (-1)^{\frac{p-1}{2} \frac{q-1}{2}}$	Gauss Integer $\pi = a + bi$ . $\text{Norm}(\pi) = p$ prime $\Rightarrow \pi$ and $\bar{\pi}$ prime, $p$ not prime	

## 8.3 Game Theory

## Classical Games (1 last one wins (normal); 2 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 $M = [pile_i], [x] := \{1, 2, \dots, [x]\}$ ).	$SG = \otimes_{i=1}^n pile_i$ . Strategy: 1 make the Nim-Sum 0 by decreasing a heap; 2 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 2 is the same as 1, opposite if all piles are 1's. Many games are essentially NIM.
NIM (powers)	$M = \{a^m   m \geq 0\}$	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 $SG_0 = 0, SG_n = \lfloor \log_2 n \rfloor + 1$	
NIM (divisors)	$M_1 = \text{divisors of } pile_i$ $M_2 = \text{proper divisors of } pile_i$	1 $SG_0 = 0, SG_n = SG_{2,n} + 1$ 2 $SG_1 = 0, SG_n = \text{number of 0's at the end of } n_{\text{binary}}$	
Subtraction Game	$M_1 = [k]$ $M_2 = S$ (finite) $M_3 = S \cup \{pile_i\}$	$SG_{1,n} = n \bmod (k+1)$ . 1 lose if $SG = 0$ ; 2 lose if $SG = 1$ . $SG_{3,n} = SG_{2,n} + 1$	For any finite $M$ , $SG$ of one pile is eventually periodic.
Moore's NIM <sub>k</sub>	One can take any number of objs from at most $k$ piles.	1 Write $pile_i$ in binary, sum up in base $k+1$ without carry. Losing if the result is 0.	2 If all piles are 1's, losing iff $n \equiv 1 \% (k+1)$ . Otherwise the result is the same as 1.
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 $n \equiv 1, 2 \pmod{4}$ $SG_n = n + 1$ , if $n \equiv 3 \pmod{4}$ $SG_n = n - 1$ , if $n \equiv 0 \pmod{4}$	
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	<b>Two</b> piles of objs. One can take any number of objs from either pile, or take the <i>same</i> number from <i>both</i> piles.	$n_k = \lfloor k\phi \rfloor = \lfloor m_k\phi \rfloor - m_k$ $m_k = \lfloor k\phi^2 \rfloor = \lfloor n_k\phi \rfloor = n_k + k$ $\phi := \frac{1+\sqrt{5}}{2}$ . $(n_k, m_k)$ 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 $\text{ones}(2n)$ odd; $SG_n = 2n + 1$ , else. $\text{ones}(x)$ : the number of 1's in $x_{\text{binary}}$	$SG_n$ 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 <i>consecutive</i> 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 non-branching stalk of length equal to their nim-sum.	
Hackenbush-graph		Vertices on any circuit can be <i>fused</i> 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  $M[v] = (\text{sum of lowerbounds of ingoing edges to } v) - (\text{sum of lowerbounds of outgoing edges from } v)$ . 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  $a-b \leq c$ . 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  $s \rightarrow A$  with capacities  $w(A)$  and edges  $B \rightarrow t$  with capacities  $w(B)$ . add edges of capacity  $\infty$  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  $\rightarrow$  max-weight independent set is complement of min-weight vertex cover.
- optimal proportion spanning tree:  $z = \sum (\text{benefit}[i] * x[i]) - l * \sum (\text{cost}[i] * x[i]) = \sum (d[i] * x[i])$ . 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:  $n - \text{max matching}$ . Min dominating set = max matching + isolated nodes. Max independent set =  $n - \text{max matching}$
- 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  $\infty$  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  $d_1 \geq \dots \geq d_n$  can be represented as the degree sequence of a finite simple graph on  $n$  vertices if and only if  $d_1 + \dots + d_n$  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