Contents

1 Code

1.1 Header

```
#include <bits/extc++.h>

using namespace std;

typedef long long ll;
typedef unsigned long long ull;
typedef pair<ll, ll> pll;
typedef vector<ll> vll;
typedef vector<string> vs;

constexpr int oo = 0x3f3f3f3f;
constexpr ll ooo = 0x3f3f3f3f3f3f3f3fLL;
constexpr double eps = 1e-7;
constexpr double PI = 2.0 * acos(0.0);

#define sz(c) ll((c).size())
#define all(c) (c).begin(), (c).end()

#if defined(ONLINE_JUDGE) || defined(DOMJUDGE)
#define debug if(1); else
#define debugprintf(...) do { } while (0)
#else
#define debug
#define debugprintf(...) do { printf(__VA_ARGS__); }
    while (0)
#endif
#define SAFEREAD(type, name) type name; cin >> name;

int main() {
    ios::sync_with_stdio(0), cin.tie(0);
}
```

HASH: 856900c842f3d3afa932448153421bd9

1.2 Hashfunktion

hash.sh

```
#!/bin/sh
sed -e 's=//.*$==' "$@" | tr -d [:space:] | perl -pe
↪  's=/\*.*?\*/==g' | md5sum | cut -d ' ' -f 1
```

Benutzen als `./hash.sh <dateiname>`.
Kommentare und Whitespace werden vor der Hash-Berechnung entfernt.

1.3 Numbers

1.3.1 Greatest Common Divisor

```
ll gcd(ll a, ll b) {
  if (a == 0)
    return b;
  return gcd(b % a, a);
}
```

HASH: 41808fdfa0833c0ebbef4b07fa97b220

```
// returns d = gcd(a, b), it is ax + by = d
ll extGcd(ll a, ll b, ll& x, ll& y) {
  if (a == 0) {
    x = 0;
    y = 1;
    return b;
  }
  ll x1, y1, rst;
  rst = extGcd(b % a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return rst;
}
```

HASH: 9d080fc550a6270b7c99860f573c6be5

1.3.2 Chinese remainder

Gegeben $A := \{a_1, ..., a_m\}$ und $N := \{n_1, ..., n_m\}$, wird ein x berechnet mit $x \equiv a_i \pmod{n_i}, \forall i \in [1, m]$.

```
ll chinRem() {
  ll N = 1;
  for (ll i = 0; i < n.size(); ++i)
    N *= n[i];
  ll sol = 0;
  for (ll i = 0; i < n.size(); ++i) {
    ll r, s, ni;
    ni = N / n[i];
    extGcd(n[i], ni, r, s);
    sol += (a[i] * ni * s + N) % N;
  }
  return sol;
}
```

HASH: ca19455d67ec9b5f076c6a633a003de7

1.3.3 Diophantine Equation

Solves $\sum_i v_i * x_i = r$ for integral x_i given v_i and i

```
bool dio(vll v, int r, vll& res, ll n) {
  vll g(sz(v));
  for (ll i = 0; i < n; ++i) {
    g[i] = v[i];
    for (ll j = i + 1; i < n; ++j) g[i] = gcd(g[i],
        v[j]);
  }
  ll m = 1;
  if (r < 0) r *= -1, m *= -1;
  if (abs(r) % g[0] != 0) return false;
  else {
     if (n == 1) res.push_back(m * r / v[0]);
     else {
        int nr = r;
        for (ll i = 0; i < n - 1; ++i) {
           ll x, y;
           int d = extGcd(v[i], g[i + 1], x, y);
           res.push_back(m * x * r / g[i]);
           nr -= x * r / g[i] * v[i];
           if (i == n - 2) {
              res.push_back(m * y * r / g[i]);
              nr -= y * r / g[i] * v[i + 1];
           }
           r = g[i + 1] * y * r / g[i];
           if (r < 0) r *= -1, m *= -1;
        }
     }
  }
}
```

HASH: d183c58fe26783665a7a2dc12f36fd71

1.3.4 Generating the Prime Table

x	p(x)	x	p(x)
10	4	100	25
1000	168	10000	1229
100000	9592	1000000	78498
10000000	664579	100000000	5761455

Sieve of Eratosthenes

```
int primes_count; // out: number of primes < n
int primes[78498]; // out: all Primes < n
bool isprime[1000000]; // output

void sieve(int n) {
  for (ll i = 2; i < n; ++i)
    isprime[i] = true;
  primes_count = 0;
  for (ll i = 2; i < n; ++i)
    if (isprime[i]) {
      primes[primes_count++] = i;
      for (ll j = 2 * i; j < n; j += i)
        isprime[j] = false;
    }
}
```

HASH: 8876bc6e0dfd190a3ccfd9cd653814fd

Euler Phi

```
// vector<ll> primes = ...;

ll phi(ll n)
{
  ll res = 1;
  ll tmp = n;
  for (ll p : primes)
  {
    ll num = 0;
    while (tmp % p == 0)
    {
      tmp /= p;
      num++;
    }
    if (num == 0)
      continue;
    res *= powabm(p, num-1, ooo) * (p-1);
    if (p*p > tmp)
      break;
  }
  if (tmp != 1)
    res *= tmp-1;
  return res;
}
```

HASH: 3bd2d8e2395c48f122900c4c2fe6cbe2

1.3.5 Repeated Squaring

```
// Returns a^b mod m for large a and b
ll powabm(ll a, ll b, ll m) {
  ll r = 1;
  while (b) {
    if (b % 2)
      r = (r * a) % m;
    a = (a * a) % m;
    b /= 2;
  }
  return r;
}
```

HASH: 185a608eefe753173661249db2052708

1.3.6 Josephus

Ein Kreis mit in n Objekten aus dem solange jedes k-te gestrichen wird. Welches bleibt übrig?

```
ll josephus(ll n, ll k) {
  ll ck = 0;
  for (ll i = 2; i < n + 1; ++i)
    ck = (ck + k % i) % i;
  return ck;
}
```

HASH: ae864cbd4c8c42649d43bb8722849620

1.4 Algebra

```
struct MATRIX {
  ll n, m;                    // n rows m columns
  vector<vector<ld> > a;  // contains values

  // set size of Matrix to x rows and y columns and
      fills it with v
  void resize(ll x, ll y, ld v = 0.0) {
    n = x;
    m = y;
    a.resize(n);
    for (ll i = 0; i < n; ++i) a[i].resize(m, v);
  }

  /* Row elimination based on the first n columns if
   * the first n columns is not invertible, kill
    yourself
   * otherwise, return the determinant of the first n
   * columns
   */
  ld Gauss() {
    ll i, j, k;
    ld det = 1.0, r;
    for (i = 0; i < n; i++) {
      for (j = i, k = -1; j < n; j++)
        if (fabs(a[j][i]) > eps) {
          k = j;
          j = n + 1;
        }
        if (k < 0) {
          n = 0;
          return 0.0;
        }
        if (k != i) {
          swap(a[i], a[k]);
          det = -det;
        }
        r = a[i][i];
        det *= r;
        for (j = i; j < m; j++) a[i][j] /= r;
        for (j = i + 1; j < n; j++) {
          r = a[j][i];
          for (k = i; k < m; k++) a[j][k] -= a[i][k] *
              r;
        }
    }
    for (i = n - 2; i >= 0; i--)
      for (j = i + 1; j < n; j++) {
        r = a[i][j];
        for (k = j; k < m; k++) a[i][k] -= r * a[j][k];
      }
      return det;
  }

  // assume n=m. returns 0 if not invertible
  ll inverse() {
    MATRIX T;
    T.resize(n, 2 * n);
    for (ll i = 0; i < n; ++i) for (ll j = 0; j < n;
        ++j) T.a[i][j] = a[i][j];
    for (ll i = 0; i < n; ++i) T.a[i][i + n] = 1.0;
    T.Gauss();
    if (T.n == 0) return 0;
    for (ll i = 0; i < n; ++i) for (ll j = 0; j < n;
        ++j) a[i][j] = T.a[i][j + n];
    return 1;
  }

  // assume v is of size m
  vector<ld> operator*(vector<ld> v) {
    vector<ld> rv(n, 0.0);
    for (ll i = 0; i < n; ++i) for (ll j = 0; j < m;
        ++j)  rv[i] += a[i][j] * v[j];
    return rv;
  }

  MATRIX operator*(MATRIX M1) {
    MATRIX R;
    R.resize(n, M1.m);
    for (ll i = 0; i < n; ++i) for (ll j = 0; j < M1.m;
        ++j) for (ll k = 0; k < m; ++k) R.a[i][j] +=
        a[i][k] * M1.a[k][j];
    return R;
  }

  // compute the determinant of M
  ld det() {
    MATRIX M1 = *this;
    ld r = M1.Gauss();
    if (M1.n == 0) return 0.0;
    return r;
  }

  // return the vector x such that Mx = v; x is empty if
      M is not invertible
  vector<ld> solve(vector<ld> v) {
    vector<ld> x;
    MATRIX M1 = *this;
    if (!M1.inverse()) return x;
    return M1 * v;
  }

  /* return the vector x such that Mx = v;
   * x is empty if M is not invertible;
   * assume n = m = 3, otherwise use own - written
    function*/
  vector<ld> solve3D(vector<ld> v) {
    vector<ld> x;
    ld p[9][9];
    for (ll i = 0; i < 9; ++i) for (ll j = i + 1; j < 9;
        ++j)
      p[i][j] = p[j][i] = a[i / 3][i % 3] * a[j / 3][j %
          3];
    ld deter = a[0][0] * (p[4][8] - p[5][7]) -
      a[0][1] * (p[3][8] - p[5][6]) +
      a[0][2] * (p[3][7] - p[4][6]);
    if (deter == 0) return x;
    x.push_back((v[0] * (p[4][8] - p[5][7]) + v[1] *
        (p[2][7] - p[1][8]) + v[2] * (p[1][5] -
        p[2][4])) / deter);
```

```
    x.push_back((v[0] * (p[5][6] - p[3][8]) + v[1] *
        (p[0][8] - p[2][6]) + v[2] * (p[2][3] -
        p[0][5])) / deter);
    x.push_back((v[0] * (p[3][7] - p[4][6]) + v[1] *
        (p[1][6] - p[0][7]) + v[2] * (p[0][4] -
        p[1][3])) / deter);
    return x;
  }
};
```

HASH: 6b88c7f6357a80919ea884e6003ca1ce

```
//Signum einer Permutation TESTED
int signum(vector<ld> p){
  int ret = 1;
  for (ll i = 0; i < p.size()-1; ++i)
      FOR(j,i+1,p.size()) if(p[i] > p[j]) ret *= -1;
  return ret;
}
```

HASH: 346478db7da3948eccd064d8c3a90d88

```
/* Alg. direkt aus Def. abgeleitet, TESTED
 * a ist nxn-Matrix
 * benötigt signum(vector<ld> p) */
ld determinantDef(vector<vector<ld> > a, int n){
  vector<ld> p; ld deter = 0, temp;
  for (ll i = 0; i < n; ++i) p.push_back(i);
  do{
    temp = signum(p);
    for (ll i = 0; i < n; ++i) temp *= a[i][p[i]];
    deter += temp;
  }while(next_permutation(p.begin(), p.end()));
  return deter;
}
```

HASH: d734e8edf1604c5844aeeaa08263044f

```
/* Nutzt Laplace'schen Entwicklungssatz;
 * bis 8x8 gute Ergebnisse für Hilbertmatrix
 * ab 10x10-Matrix nicht mehr einsenden! */
ld determinant(vector<vector<ld> > m, ll n){
  if (n == 1) return m[0][0];
  if (n == 2) return m[0][0] * m[1][1] - m[0][1] *
      m[1][0];
  //if(n==3) return längere Zeile, bei Bedarf kann sie
      eingegeben werden
  ld deter = 0, t = -1;
  vector<vector<ld> > a;
  a.resize(n - 1); for (ll i = 0; i < n - 1; ++i)
      a[i].resize(n - 1);
  for (ll i = 0; i < n; ++i) {
    t *= -1;
    if(m[0][i] != 0) {
      for (ll j = 1; j < n; ++j) for (ll k = 0; k < i;
          ++k) a[j - 1][k] = m[j][k];
      for (ll j = 1, j < n; ++j) for (ll k = i + 1; k <
          n; ++k) a[j - 1][k - 1] = m[j][k];
      deter += t * m[0][i] * determinant(a, n - 1);
    }
  }
  return deter;
}
```

HASH: d8cb5a5a3e931e18c8fb42f5cacd8715

```
/* Löst LGS mit Cramerscher Regel (TESTED)
 * n = m
 * ab 9x9 nicht mehr benutzen!
 * benutzt MATRIX-Struct und determinant(...)*/
vector<ld> solveND(MATRIX &M1, vector<ld> v) {
  MATRIX M = M1;
  ld deter = determinant(M.a, M.n);
  vector<ld> x,temp; temp.resize(M.n);
  if (deter == 0) return x;
  for (ll i = 0; i < M.n; ++i) {
    temp[i] = M.a[i][0];
    M.a[i][0] = v[i];
  }
  x.push_back(determinant(M.a, M.n) / deter);
  for (ll i = 1; i < M.n; ++i) {
    for (ll j = 0; j < M.n; ++j) {
      M.a[j][i-1] = temp[j];
      temp[j] = M.a[j][i];
      M.a[j][i] = v[j];
    }
    x.push_back(determinant(M.a, M.n) / deter);
  }
  return x;
}
```

HASH: 83e4b731208cb7196c81012bd7a18d98

1.5 Graphs

1.5.1 Dijkstra

```
vll adj[MAXN];
vll w[MAXN];
ll dist[MAXN];

void dijkstra(ll s, ll n) {
  priority_queue<pll> q;
  for (ll i = 0; i < n; ++i) dist[i] = oo;
  dist[s] = 0;
  pll _ini(0, s); q.push(_ini);
  while (sz(q)) {
    ll d = -q.top().first;
    ll x = q.top().second;
    q.pop();
    if (dist[x] != d) continue;
    for (ll i = 0; i < sz(adj[x]); ++i) {
      ll nd = d + w[x][i];
      ll nx = adj[x][i];
      if (nd >= dist[nx]) continue;
      pll next (-nd, nx);
      dist[nx] = nd;
      q.push(next);
    }
  }
}
```

HASH: f204e3e348af2e1a2f4e35009e6ea0c9

1.5.2 Floyd Warshall (n^3)

```
ll dist[100][100]; // in/out: weighted adj matrix
ll prev[100][100]; // output: predecessor of dest node
void floyd(ll n) {
  for (ll i = 0; i < n; ++i) for (ll j = 0; j < n; ++j)
      prev[i][j] = i;
  for (ll k = 0; k < n; ++k) for (ll i = 0; i < n; ++i)
      for (ll j = 0; j < n; ++j)
    if (dist[i][k] < ooo && dist[k][j] < ooo &&
        dist[i][k] + dist[k][j] < dist[i][j]) {
      dist[i][j] = max(dist[i][k] + dist[k][j], -ooo);
      prev[i][j] = (prev[k][j] != j ? prev[k][j] :
          prev[i][k]);
    }
  // Detect negative cycles
  for (ll k = 0; k < n; ++k) if (dist[k][k] < 0) for (ll
      i = 0; i < n; ++i) for (ll j = 0; j < n; ++j)
    if (dist[i][k] < ooo && dist[k][j] < ooo)
      dist[i][j] = -ooo;
}
```

HASH: f2721d0510eccd3fc79a658dd1ca7588

1.5.3 Bellman-Ford (nm)

```
vll adj[MAXN]; // input: adjacency node list
vll w[MAXN]; // input: adjacency weight list
ll d[MAXN]; // output: resulting distances
void bellmanford(ll n, ll src) {
  for (ll i = 0; i < n; ++i) d[i] = ooo;
  d[src] = 0;
  for (ll k = 0; k < n; ++k) for (ll i = 0; i < n; ++i)
      if (d[i] < ooo) for (ll j = 0; j < sz(adj[i]);
      ++j)
    d[adj[i][j]] = min(d[adj[i][j]], max(d[i] + w[i][j],
        -ooo));

  // negative cycle detection: not neccessary
  for (ll i = 0; i < n; ++i) if (d[i] < ooo) for (ll j =
      0; j < sz(adj[i]); ++j)
    if (d[adj[i][j]] > d[i] + w[i][j]) d[adj[i][j]] =
        -ooo;
}
```

HASH: 6e123e2e5166fbadb89937637c952b12

1.5.4 Minimum Spanning Tree / Union Find

```
struct edge { ll x,y; double d; };

bool operator <(const edge& e1, const edge& e2){ return
    e1.d < e2.d; }

edge e[MAXE]; // input: edge list
ll u[MAXE]; // output: edge used in MST (0/1)
ll pa[MAXN]; // temp: union find parent
ll rk[MAXN]; // temp: union find rank

ll ufind(ll i) { // returns component i belongs to
  if (pa[i] != i) pa[i] = ufind(pa[i]);
  return pa[i];
}

ll uunion(ll a, ll b) { // 1 if unified, 0 else
  a = ufind(a);
  b = ufind(b);
  if (a == b) return 0;
  if (rk[a] > rk[b])
    pa[b] = a;
  else
    pa[a] = b;
  if (rk[a] == rk[b]) rk[b]++;
  return 1;
}

ll kruskal(ll n, ll m) { // returns sum of weights
  sort(e, e + m);
  for (ll i = 0; i < n; ++i) { // init union find
    pa[i] = i;
    rk[i] = 0;
  }
  ll sum = 0;
  for (ll i = 0; i < m; ++i) {
    u[i] = uunion(e[i].x, e[i].y);
    if (u[i]) sum += e[i].d;
  }
  return sum;
}
```

HASH: adc1e28574f9bf06d08ad6a4dd17dd00

1.5.5 Minimum Average Cost Cycle

```
vll adj[MAXN];      // input: adj list
ll ct[MAXN][MAXN];  // input: cost mtx
ll A[MAXN][MAXN];   // temp: dists to t
ll p[MAXN][MAXN];   // temp: succ in path
ll ccnt, cc[MAXN];  // output: nodes of cycle

// returns min avg cycle cost, or NAN if no cycle
// WARNING: adds new node to graph, modify if needed!
double minAvgCycle(ll n) {
  adj[n].clear();  // add new node to graph
  for (ll i = 0; i < n; ++i) adj[i].push_back(n);
  for (ll i = 0; i < n; ++i) ct[i][n] = 0;
  for (ll i = 0; i < n; ++i) A[0][i] = oo;
  A[0][n] = 0;
  // Bellman-Ford search
  for (ll t = 1; t < n + 2; ++t) for (ll i = 0; i < n +
      1; ++i) {
    A[t][i] = oo;
    for (ll z = 0; z < sz(adj[i]); ++z) {
      ll j = adj[i][z];
      if (ct[i][j] < oo && A[t - 1][j] < oo)
        if (ct[i][j] + A[t - 1][j] < A[t][i]) {
          A[t][i] = ct[i][j] + A[t - 1][j];
          p[t][i] = j;
        }
    }
  }

  // calc min avg cycle
  double ans = 1e+15;
  ll st = -1;
  for (ll i = 0; i < n; ++i) if (A[n + 1][i] < oo) {
    double tmp = -(1e+15);
    for (ll t = 0; t < n+1; ++t) if (A[t][i] < oo)
      tmp = max(tmp, (double) (A[n + 1][i] - A[t][i]) /
          (n + 1 - t));
    if (tmp < ans) {
      ans = tmp;
      st = i;
    }
  }
  if (st == -1) return NAN;
  ccnt = 0;  // read cycle nodes
  ll wk = st;
  for (ll t = n + 2; t >= 0; --t) {
    cc[ccnt++] = wk;
    wk = p[t][wk];
  }
  static ll used[MAXN];  // find some cycle
  for (ll i = 0; i < n + 1; ++i) used[i] = -1;
  for (ll i = 0; i < ccnt; ++i)
    if (used[cc[i]] == -1)
      used[cc[i]] = i;
    else {
      for (ll j = used[cc[i]]; j < i; ++j)
        cc[j - used[cc[i]]] = cc[j];
      ccnt = i - used[cc[i]];
      break;
    }
  return ans;
}
```

HASH: 0a6496946458c72d2bb62fcfe7255772

1.5.6 Maximum Flow / Minimum Cut

```
typedef ll weight;
struct edge {
```

```
  ll to,   // adjacent node
     cap,  // capacity of this edge (double?)
     flow, // calculated flow on this edge (double?)
     oi;   // index number of opposite directed edge
};

vector<edge> e[MAXN]; // in/out: adj list
ll aug[MAXN]; // temp: max aug in path to given node
ll pa[MAXN]; // temp: predecessor in path

// adds edge to network
// note that you have to set both opposing edge
    capacities simultaneously
void addEdge(ll u, ll v, ll capUV, ll capVU) {
  edge uv, vu;
  uv.to = v;
  uv.cap = capUV;
  uv.flow = 0;
  uv.oi = e[v].size();

  vu.to = u;
  vu.cap = capVU;
  vu.flow = 0;
  vu.oi = e[u].size();

  e[u].push_back(uv);
  e[v].push_back(vu);
}

// find shortest augmenting path (Edmonds-Karp)
// time complexity: O(n+m) (adjacency list)
bool findAugPathEK(ll src, ll sink, ll n) {
  static ll qu[MAXN]; // pseudo-queue
  for (ll i = 0; i < n; ++i) pa[i] = -1;
  aug[src] = oo;

  ll start = 0, end = 0;
  qu[end++] = src;
  while (start != end) { // do simple BFS
    ll u = qu[start++];
    for (ll i = 0; i < sz(e[u]); ++i) {
      ll v = e[u][i].to;
      ll curaug = e[u][i].cap - e[u][i].flow;
      if (pa[v] == -1 && curaug > 0) {
        qu[end++] = v;
        aug[v] = min(aug[u], curaug);
        pa[v] = e[u][i].oi;
        if (v == sink) return true;
      }
    }
  }
  return false;
}

// calculates the maximum flow of the given network
    using augmenting paths (Ford-Fulkerson)
ll calcMaxFlow(ll src, ll sink, ll n) {
  // initialize empty flow
  ll ret = 0;
  for (ll i = 0; i < n; ++i) for (ll j = 0; j <
      sz(e[i]); ++j) e[i][j].flow = 0;
  while (findAugPathEK(src,sink,n)) {
    // inc flow on path (max aug is aug[sink])
    ll v = sink;
    while (v != src) {
      e[e[v][pa[v]].to][e[v][pa[v]].oi].flow +=
          aug[sink];
      e[v][pa[v]].flow -= aug[sink];
      v = e[v][pa[v]].to;
    }
    ret += aug[sink];
    if (ret >= oo) break;
  }
  return ret;
}
```

HASH: a3142302d9cd7e3188d9d0659a4ad291

1.5.7 Minimum Cost Maximum Flow

```
typedef ll weight;
struct edge {
  ll to,   // adjacent node
     cap,  // capacity of this edge (double?)
     flow, // calculated flow on this edge (double?)
     oi;   // index number of opposite directed edge
};

vector<edge> e[MAXN]; // in/out: adj list
ll aug[MAXN]; // temp: max aug in path to given node
ll pa[MAXN]; // temp: predecessor in path

// adds edge to network
// note that you have to set both opposing edge
    capacities simultaneously
void addEdge(ll u, ll v, ll capUV, ll capVU) {
  edge uv, vu;
  uv.to = v;
  uv.cap = capUV;
  uv.flow = 0;
  uv.oi = e[v].size();

  vu.to = u;
  vu.cap = capVU;
  vu.flow = 0;
  vu.oi = e[u].size();

  e[u].push_back(uv);
  e[v].push_back(vu);
}

// find shortest augmenting path (Edmonds-Karp)
// time complexity: O(n+m) (adjacency list)
bool findAugPathEK(ll src, ll sink, ll n) {
  static ll qu[MAXN]; // pseudo-queue
  for (ll i = 0; i < n; ++i) pa[i] = -1;
  aug[src] = oo;

  ll start = 0, end = 0;
  qu[end++] = src;
  while (start != end) { // do simple BFS
    ll u = qu[start++];
    for (ll i = 0; i < sz(e[u]); ++i) {
      ll v = e[u][i].to;
      ll curaug = e[u][i].cap - e[u][i].flow;
      if (pa[v] == -1 && curaug > 0) {
        qu[end++] = v;
        aug[v] = min(aug[u], curaug);
        pa[v] = e[u][i].oi;
        if (v == sink) return true;
      }
    }
  }
  return false;
}

weight cost[MAXN][MAXN]; // in:cost mtx (no neg cycles)
weight d[MAXN]; // temp: dist from src to i
```

```
// find aug path with min sum of costs (Bellman-Ford)
// time complexity: O(nm) (adjacency list)
bool findAugPathMinCost(ll src, ll sink, ll n) {
  for (ll i = 0; i < n; ++i) d[i] = oo;
  d[src] = 0;
  aug[src] = oo;

  for (ll k = 0; k < n; ++k) for (ll u = 0; u < n; ++u)
      if (d[u] < oo) for (ll i = 0; i < sz(e[u]); ++i) {
    ll v = e[u][i].to;
    ll curaug = e[u][i].flow < 0 ? -e[u][i].flow :
        e[u][i].cap - e[u][i].flow;
    ll curcost = e[u][i].flow<0 ? -cost[v][u] :
        cost[u][v];
    if (curaug > 0 && d[v] > d[u] + curcost) {
      d[v] = d[u] + curcost;
      aug[v] = min(aug[u], curaug);
      pa[v] = e[u][i].oi;
    }
  }
  return d[sink] < oo;
}

ll pi[MAXN]; // temp: // node potentials
// find aug path with min sum of costs (Dijkstra)
// uses node potentials to avoid negative edge weights
// time complexity: O(m log n) (adjacency list)
bool findAugPathDijkstra(ll src, ll sink, ll n) {
  priority_queue<pair<weight, ll> > h;

  for (ll i = 0; i < n; ++i) d[i] = oo;
  d[src] = 0;
  aug[src] = oo;

  h.push(pair<ll, ll>(-d[src], src));

  while (!h.empty()) {
    ll u = h.top().second; // the node
    weight dst = -h.top().first; // the distance
    h.pop();

    if (d[u] != dst) continue;

    for (ll i = 0; i < sz(e[u]); ++i) { // for each
        neighbor of v
      ll v = e[u][i].to;
      ll curaug = e[u][i].flow < 0 ? -e[u][i].flow :
        e[u][i].cap - e[u][i].flow;
      ll curcost = e[u][i].flow < 0 ? -cost[v][u] -
          pi[v] + pi[u] : cost[u][v] + pi[u] - pi[v];
      if (curaug > 0 && dst + curcost < d[v]) {
        d[v] = dst + curcost;
        aug[v] = min(aug[u], curaug);
        pa[v] = e[u][i].oi;
        h.push(pair<ll, ll>(-d[v], v));
      }
    }
  }
  return d[sink] < oo;
}
// calculates the maximum flow of the given network
    using augmenting paths (Ford-Fulkerson)
ll calcMaxFlow(ll src, ll sink, ll n) {
  // initialize llempty flow
  ll ret = 0;
  for (ll i = 0; i < n; ++i) FOR(ll j = 0; j < sz(e[i]);
      ++j) e[i][j].flow = 0;
  for (ll i = 0; i < n; ++i) pi[i] = 0; // zero node
      potentials
  findAugPathMinCost(src, sink, n);
  for (ll i = 0; i < n; ++i) if (d[i] < oo) pi[i] +=
      d[i];
  // increase flow as long as an aug path can be found
  while (findAugPathDijkstra(src, sink, n)) {
    for (ll i = 0; i < n; ++i) if (d[i] < oo) pi[i] +=
        d[i];
    // inc flow on path (max aug is aug[sink])
    ll v = sink;
    while (v != src) {
      e[e[v][pa[v]].to][e[v][pa[v]].oi].flow +=
          aug[sink];
      e[v][pa[v]].flow -= aug[sink];
      v = e[v][pa[v]].to;
    }
    ret += aug[sink];
    if (ret >= oo) break;
  }
  return ret;
}
```

HASH: 7ae4fd885e873b0721afccd9f1144a2b

1.5.8 Push-Relabel

```
struct edge {
  ll to, cap, flow, oi;
};

#define MAXN 1000000

vector<edge> adj[MAXN];
ll h[MAXN];
ll e[MAXN];
ll act[MAXN];

void addEdge(ll u, ll v, ll capUV, ll capVU) {
  edge uv, vu;
  uv.to = v;
  uv.cap = capUV;
  uv.flow = 0;
  uv.oi = adj[v].size();

  vu.to = u;
  vu.cap = capVU;
  vu.flow = 0;
  vu.oi = adj[u].size();

  adj[u].push_back(uv);
  adj[v].push_back(vu);
}

priority_queue<pll> active;

void push(ll u, ll nei, ll t) {
  ll delta = min(e[u], adj[u][nei].cap -
      adj[u][nei].flow);
  adj[u][nei].flow += delta;
  adj[adj[u][nei].to][adj[u][nei].oi].flow -= delta;
  e[u] -= delta;
  if (e[adj[u][nei].to] == 0 && adj[u][nei].to != t)
    active.push(make_pair(h[adj[u][nei].to],
        adj[u][nei].to));
  e[adj[u][nei].to] += delta;
}

ll relCount;
```

```
void relable(ll u) {
  relCount++;
  h[u] = oo;
  for (auto n : adj[u])
    if (n.flow < n.cap)
      h[u] = min(h[u], h[n.to]);
  h[u]++;
}

void discharge(ll u, ll t) {
  while (e[u]) {
    if (act[u] == sz(adj[u])) {
      relable(u);
      act[u] = 0;
    } else {
      if (adj[u][act[u]].cap - adj[u][act[u]].flow > 0
          && h[u] > h[adj[u][act[u]].to])
        push(u, act[u], t);
      else
        act[u]++;
    }
  }
}

void backBFS(ll s, ll t, ll n) {
  queue<pll> q;
  q.push(make_pair(t, 0));
  for (ll i = 0; i < n; ++i) h[i] = -1;
  h[s] = n;
  h[t] = 0;
  while (sz(q)) {
    pll no = q.front();
    q.pop();
    for (auto i : adj[no.first]) if (h[i.to] == -1)
      if (adj[i.to][i.oi].cap > adj[i.to][i.oi].flow) {
        h[i.to] = no.second + 1;
        q.push(make_pair(i.to, no.second + 1));
      }
  }
  // set unreachable nodes to height n+1
  ll nr = 0;
  for (ll i = 0; i < n; ++i) if (h[i] == -1) h[i] = n +
      1, nr++;
  // rebuild active queue
  while (sz(active)) active.pop();
  for (ll i = 0; i < n; ++i) if (i != s && i != t &&
      e[i])
    active.push(make_pair(h[i], i));
}

ll pushrelable(ll s, ll t, ll n) {
  for (ll i = 0; i < n; ++i) e[i] = act[i] = 0, h[i] =
      -1;
  for (ll i = 0; i < n; ++i) for (auto n : adj[i])
      n.flow = 0;
  h[s] = n;
  e[s] = oo;
  h[n] = -1;
  e[n] = 0; // just for convenience

  backBFS(s, t, n);

  while (sz(active)) active.pop();
  for (ll i = 0; i < sz(adj[s]); ++i) if (adj[s][i].cap)
      {
    push(s, i, t);
    ll to = adj[s][i].to;
    if (h[to] > 0) active.push(make_pair(h[to], to));
  }
  relCount = 0;
  while (sz(active)) {
    pll no = active.top();
    active.pop();
    if (!e[no.second]) continue;
    discharge(no.second, t);

    if (e[no.second])
      active.push(make_pair(h[no.second], no.second));
    if (relCount / 2 % n == n - 1) backBFS(s, t, n),
        relCount = 0;
  }

  ll flow = 0;
  for (auto n : adj[s]) flow += n.flow;
  return flow;
}
```

HASH: ef8f75cf271f27097990609c0b67e0f6

1.5.9 Bipartite Matching

```
vll adj[MAXN];
bool matched[MAXN];
bool matchedEdge[MAXN][MAXN];
bool visited[MAXN];

bool dfs(ll node, bool backEdge, bool recursive)
{
  if (visited[node])
    return false;
  visited[node] = true;

  if (!matched[node] && !recursive)
  {
    matched[node] = true;
    return true;
  }

  for (ll nb : adj[node])
  {
    if (matchedEdge[node][nb] == backEdge && dfs(nb,
        !backEdge, false))
    {
      matchedEdge[node][nb] = !matchedEdge[node][nb];
      matchedEdge[nb][node] = !matchedEdge[nb][node];
      matched[node] = true;
      return true;
    }
  }

  return false;
}

ll match(ll n)
{
  memset(matched, 0, sizeof(matched));
  memset(matchedEdge, 0, sizeof(matchedEdge));

  ll nMatched = 0;
  while (true) // Michael würde diese Zeile löschen!
  {
    bool foundMatch = false;
    for (ll i = 0; i < n; ++i)
    {
      if (matched[i])
        continue;

      memset(visited, 0, sizeof(visited));
```

```
      bool ret = dfs(i, false, true);
      if (ret)
      {
        ++nMatched;
        foundMatch = true;
      }
    }
    if (!foundMatch)
      break;
  }

  return nMatched;
}
```

HASH: 44f917c63533e83399b13cada7347d92

1.5.10 DFS (finding articulation nodes, biconnected components, bridges)

```
vll adj[MAXN]; // input: adj list
ll curpre; // temp: next preorder num
bool articulation[MAXN]; // output: articulation nodes
ll num[MAXN];
vector<ll> st;

template<class F>
ll biconnectedDfs(ll i, ll pa, F&f){
  ll me = num[i] = ++curpre, top = me, cc = 0;
  bool isan = false;
  for(auto j : adj[i])
    if(pa != j){
      if(num[j]){
        top = min(top,num[j]);
        if(num[j] < me){
          st.push_back(i);
          st.push_back(j);
        }
      } else {
        cc++;
        ll si = sz(st);
        ll up = biconnectedDfs(j, i, f);
        isan |= pa != -1 && up >= me;
        top = min(top, up);
        if(up == me){
          f(vector<ll>(st.begin()+si, st.end()));
          st.resize(si);
        } else if(up < me){
          st.push_back(i);
          st.push_back(j);
        } else { /* bridge */ }
      }
    }
  isan |= pa==-1 && cc>=2;
  articulation[i] = isan;
  return top;
}
template<class F>
void bicomps(ll n, F f){
  for(ll i=0; i<n; i++) num[i] = 0;
  for(ll i=0; i<n; i++)
    if(num[i]==0)
      biconnectedDfs(i,-1,f);
}
```

HASH: d4bc834026dfab24fc1a452ca912c5ff

1.5.11 Topological sorting

```
vll adj[MAXN]; // input: adjacency list
ll p, v[MAXN]; // temp: node color
ll fg; // output: cycle found (0/1)
ll od[MAXN]; // output: nodes in order

void dfs(ll a) {
  if (v[a] == 1) fg = 1;
  if (v[a]) return;
  v[a] = 1; // gray
  for (ll i = 0; i < sz(adj[a]); ++i) dfs(adj[a][i]);
  v[a] = 2; // black
  od[p] = a;
  p--;
}

void topsort(ll n) {
  for (ll i = 0;  i < n; ++i) v[i] = 0; //white
  fg = 0; p = n-1;
  for (ll i = 0; i < n; ++i) if (!v[i]) dfs(i);
}
```

HASH: a76828a43a58c0a4be93ae77eb75258b

1.5.12 Strongly connected components

```
vll adj[MAXN][2]; // input: graph and rev. graph
ll ccomp, comp[MAXN]; // output: component of each node
vll st; // temp: stack

void dfs(ll n, ll c, ll dir) {
  if (comp[n] != -1) return;
  comp[n] = c;
  for (ll i = 0; i < sz(adj[n][dir]); ++i)
    dfs(adj[n][dir][i], c, dir);
  st.push_back(n);
}

void kosaraju(ll n) {
  memset(comp, -1, sizeof(comp));
  st.clear();

  for (ll i = 0; i < n; ++i) dfs(i, 0, 0);

  reverse(all(st));
  memset(comp, -1, sizeof(comp));
  ccomp = 0;

  for (ll i = 0; i < n; ++i)
    if (comp[st[i]] == -1) dfs(st[i], ccomp++, 1);
}
```

HASH: b7f3260e2ac64c7893d2013a9bd071f2

1.5.13 Travelling Salesman Problem

```
ll r[MAXN][MAXN]; // input: edge weight matrix
ll dp[1 << MAXN][MAXN]; // temp: dp[S][s] is min length
    of path from s to 0 visiting all nodes in S

ll play(ll n, ll S, ll s) {
  if (S == (1 << s)) return r[s][0];
  ll& v = dp[S][s];
  if (v >= 0) return v;
  v = oo;
  for (ll i = 0; i < n; i++) if (i != s) if (S & (1 <<
      i))
  v = min(v, r[s][i] + play(n, S - (1 << s), i));
  return v;
}
ll tsp(ll n) {
  memset(dp, -1, sizeof(dp));
  return play(n, (1 << n) - 1, 0);
}
```

HASH: bc2d9d67dcae7bf828bd44fee3c3e484

1.6 Geometry - Integersafe

```
typedef double coord;
struct pt{
  coord x,y;
  pt():x(0),y(0){};
  pt(coord _x,coord _y):x(_x),y(_y){};
  pt operator+(const pt& p) { return pt(x+p.x,y+p.y);}
  pt operator-(const pt& p) { return pt(x-p.x,y-p.y);}
  bool operator==(const pt& p) { return abs(x-p.x)<eps
      && abs(y-p.y)<eps;}
  pt operator*(const coord f) { return pt(x*f,y*f); }
  pt operator/(const coord f) { return pt(x/f,y/f); }
  bool operator <(const pt& p) const {
    return x < p.x || (x == p.x && y < p.y);
  }
  coord cross(pt p) const { return x*p.y - y*p.x; }
  coord cross(pt a, pt b) const { return
      (a-*this).cross(b-*this); }
  coord operator*(const pt& p) { return x*p.x+y*p.y; }
};
coord len2(pt p) { return p*p; }
double len(pt p) { return sqrt(double(len2(p))); }
double phi(pt p) { return atan2((double)p.y,p.x); }
double cosSatz(double r2, double r, double r1) {
  return acos((-r2*r2 + r1*r1 + r*r) / (2*r1*r));
}
```

HASH: ba891df5c5c595321fb8760bbde4d0c0

1.6.1 Counter-Clockwise-Test

```
// ccw test. decides whether three points are arranged
    counterclockwise. 1=ccw, 0=straight, -1=cw
int ccw(pt p0, pt p1, pt p2) {
  coord d1 =(p1.x-p0.x)*(p2.y-p0.y);
  coord d2 =(p2.x-p0.x)*(p1.y-p0.y);
  return (d1-d2>eps)-(d2-d1>eps);
}
```

HASH: 1a11cbf248e6cad97cae2bfeb587e07e

1.6.2 Is Point on Segment

```
// 0 = no, 1= on-end-point, 2=strict
int isPointOnSegment(pt p, pt a0, pt a1) {
  if(ccw(a0,a1,p)) return 0;
  coord cx = (p.x-a0.x)*(p.x-a1.x);
  coord cy = (p.y-a0.y)*(p.y-a1.y);
  if(cx > eps || cy > eps) return 0;
  if(cx < -eps || cy < -eps) return 2;
  return 1;
}
```

HASH: a7fca27d0a3268fde859b885eef23e6b

1.6.3 Are Segments Intersecting

```
// line intersection test. decides whether two lines
    have a common point
// 0 = none, 1=strict, 2=on-end-point
int isSegmentIntersect(pt a0, pt a1, pt b0, pt b1) {
  int c1 = ccw(a0, a1, b0);
  int c2 = ccw(a0, a1, b1);
  int c3 = ccw(b0, b1, a0);
  int c4 = ccw(b0, b1, a1);
  if (c1*c2>0 || c3*c4>0) return 0;
  if (!c1 && !c2 && !c3 && !c4) {
    c1 = isPointOnSegment(a0,b0,b1);
    c2 = isPointOnSegment(a1,b0,b1);
    c3 = isPointOnSegment(b0,a0,a1);
    c4 = isPointOnSegment(b1,a0,a1);
    if (c1 && c2 && c3 && c4)
      return 1 + (a0.x != a1.x || a0.y != a1.y);
    if (c1 + c2 + c3 + c4 == 0)
      return 0;
    return 3 - max({c1, c2, c3, c4});
  }
  return 1 + (c1 == 0 || c2 == 0 || c3 == 0 || c4 == 0);
}
```

HASH: ebcba7b3d2004c53ab04f71a6ed6c647

1.6.4 Is Polygon Convex

```
// 0 = no, 1 = non-strict, 2 = strict
int isConvex(vector<pt>& poly) {
  int ret=2, c=0, n=sz(poly);
  for (ll i = 0; i < n; ++i) {
    int cc = ccw(poly[i],poly[(i+1)%n],poly[(i+2)%n]);
    if(!cc)ret=1;
    else if(!c) c=cc;
    else if(c!=cc) return 0;
  }
  return ret;
}
```

HASH: 60d86f862c5c07b1c218759c2b25b1c6

1.6.5 Area of Polygon

```
// double of area of a simple polygon, not necessarily
    convex
coord twoarea(vector<pt>& poly) {
  int n = sz(poly);
  coord ret = 0;
  FOR(i,0,n)
  ret += (poly[(i+1)%n].x-poly[i].x)*
  (poly[(i+1)%n].y+poly[i].y);
  return abs(ret);
}
```

HASH: 22918b7c9679347d15eba59da1eccb4f

1.6.6 Point in Polygon Test

```
// test whether point is inside polygon
// 0=outside, 1=edge, 2=inside
int isInside(pt p, vector<pt>& poly) {
  int numAbove = 0;
  int numIntersects = 0;
  for (ll i = 0; i < sz(poly); ++i) {
    pt p0 = poly[i];
    pt p1 = poly[(i+1)%sz(poly)];
    if(isPointOnSegment(p,p0,p1)) return 1;
    if(p0.y-p.y<=eps && p1.y-p.y<=eps) continue;
    if(p0.y-p.y>eps && p1.y-p.y>eps) continue;
    pt d=p-p0;
    pt d1=p1-p0;
    if(d1.y < 0) d1.y*=-1, d.y*=-1;
    if(d.y*d1.x > d.x*d1.y) {
      if((p0.y-p.y) * (p1.y-p.y) < 0) numIntersects++;
      else numAbove++;
    }
  }
  return (((numIntersects+(numAbove%2))%2)!=0?2:0);
}
```

HASH: 6073c7966a124f41495b36f5c8dc5ec5

1.6.7 Convex Hull

```
// strict: duplicate points and points on edges removed
vector<pt> convexhull(vector<pt> poly) {
  ll n = sz(poly), k = 0;
  vector<pt> h(2 * n);
  sort(all(poly));
  for (ll i = 0; i < n; ++i) {
    while (k > 1 && ccw(h[k-2], h[k-1], poly[i]) <= 0)
        k--;
    h[k++] = poly[i];
  }
  ll t = k;
  for (ll i = n - 2; i >= 0; --i) {
    while (k > t && ccw(h[k-2], h[k-1], poly[i]) <= 0)
        k--;
    h[k++] = poly[i];
  }
  h.resize(k > 1 ? k - 1 : k);
  return h;
}
```

HASH: 44e15c44b9404441cf26b861778adbbc

1.6.8 Delaunay Triangulation

```
/**
 * Author: Philippe Legault
 * Date: 2016
 * License: MIT
 * Source:
    https://github.com/Bathlamos/delaunay-triangulation/
 * Description: Fast Delaunay triangulation.
 * Each circumcircle contains none of the input points.
 * There must be no duplicate points.
 * If all points are on a line, no triangles will be
    returned.
 * Should work for doubles as well, though there may be
    precision issues in 'circ'.
 * Returns triangles in order \{t[0][0], t[0][1],
    t[0][2], t[1][0], \dots\}, all counter-clockwise.
 * Time: O(n \log n)
 * Status: fuzz-tested
 */

typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are <
    2e4)
pt arb(LLONG_MAX,LLONG_MAX); // not equal to any other
    point

struct Quad {
  bool mark; Q o, rot; pt p;
  pt F() { return r()->p; }
  Q r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
};

bool circ(pt p, pt a, pt b, pt c) { // is p in the
    circumcircle?
  lll p2 = len2(p), A = len2(a)-p2,
      B = len2(b)-p2, C = len2(c)-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A +
      p.cross(c,a)*B > 0;
}
Q makeEdge(pt orig, pt dest) {
  Q q[] = {new Quad{0,0,0,orig}, new Quad{0,0,0,arb},
           new Quad{0,0,0,dest}, new Quad{0,0,0,arb}};
  for(ll i=0; i<4; i++)
    q[i]->o = q[-i & 3], q[i]->rot = q[(i+1) & 3];
  return *q;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

pair<Q,Q> rec(const vector<pt>& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1],
        s.back());
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r()
        };
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec({all(s) - half});
  tie(B, rb) = rec({sz(s) - half + all(s)});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e))
    \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}

vector<pt> triangulate(vector<pt> pts) {
  sort(all(pts));  assert(unique(all(pts)) ==
      pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1;
    pts.push_back(c->p); \
  q.push_back(c->r()); c = c->next(); } while (c != e);
    }
  ADD; pts.clear();
```

```
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

HASH: e819bee412a843289dc639f3a72e7d6e

1.7 Geometry - Non Integersafe

1.7.1 Intersection of lines

```
// (oo,0)=same, (oo,_)=parallel, (x,y)=point
pt lineIntersect(pt a0, pt a1, pt b0, pt b1) {
  pt d13 = a0-b0;
  pt d43 = b1-b0;
  pt d21 = a1-a0;
  coord un = d43.x*d13.y - d43.y*d13.x;
  coord ud = d43.y*d21.x - d43.x*d21.y;
  if(abs(ud)<eps) return pt(oo,un);
  return pt(a0.x + un*d21.x/ud, a0.y + un*d21.y/ud);
}
```

HASH: c87ee5c9b69b39fe63bebd8353a2afa0

1.7.2 Polygon/Polygon Intersect

```
// assumes intersection is convex
vector<pt> intersect(vector<pt>& p1, vector<pt>& p2) {
  vector<pt> tmp;
  for (auto i : p1) if(isInside(i,p2)) tmp.push_back(i);
  for (auto i : p2) if(isInside(i,p1)) tmp.push_back(i);
  for (ll i = 0; i < sz(p1); ++i) for (ll j = 0; j <
      sz(p2); ++j) {
    if(isSegmentIntersect(p1[i], p1[(i+1) % sz(p1)],
        p2[j], p2[(j+1) % sz(p2)])) {
      pt in = lineIntersect(p1[i],p1[(i+1)%sz(p1)],
      p2[j],p2[(j+1)%sz(p2)]);
      if(abs(in.x-oo)>eps) tmp.push_back(in);
    }
  }
  return convexhull(tmp); // strict
}
```

HASH: 438c57b00139f2bdc1878434bdd80e4d

1.7.3 Polygon/Line Intersect

```
// returns poly ccw of a0a1
// assumes intersection is convex
vector<pt> intersect(vector<pt>& p1, pt a0, pt a1) {
  vector<pt> tmp;
  for (auto i : p1) if(ccw(a0,a1,i)>0) tmp.push_back(i);
  for (ll i = 0; i < sz(p1); ++i) {
    pt in = lineIntersect(p1[i],p1[(i+1)%sz(p1)],a0,a1);
    if(abs(in.x-oo)<eps) continue;
    if(isPointOnSegment(in,p1[i],p1[(i+1)%sz(p1)]))
    tmp.push_back(in);
  }
  return convexhull(tmp); // strict
}
```

HASH: 442a93028e3eb8afbf0f0d07ccda2fd4

1.7.4 Rotate point

```
// rotate ccw, phi in radians
pt rotate(pt p, double phi) {
  return pt(p.x*cos(phi)-p.y*sin(phi),
  p.x*sin(phi)+p.y*cos(phi));
}
```

HASH: b431884d279038e10d55dd9390339644

1.7.5 Center of Mass

```
pair<double,double> centerOfMass(vector<pt> poly) {
  ll n = sz(poly);
  coord sum=0,sumx=0,sumy = 0;
  for (ll i = 0; i < n; ++i) {
    coord tmp = (poly[(i+1)%n].y*poly[i].x)
    -(poly[(i+1)%n].x*poly[i].y);
    sum += tmp;
    sumx+=(poly[i].x+poly[(i+1)%n].x)*tmp;
    sumy+=(poly[i].y+poly[(i+1)%n].y)*tmp;
  }
  return pair<double,double>
  ((sumx/3.0)/sum,(sumy/3.0)/sum);
}
```

HASH: 903f62115978645c1f9851382a70d2a2

1.7.6 Closest Point aka Lot

```
// calc "Lot" of p on a0a1
pt closestpt(pt a0, pt a1, pt p) {
  pt d = a1-a0;
  return a0+(d*(d*(p-a0))/(d*d));
}
```

HASH: b13bb1a98aee0d3a889b963478cb675b

1.7.7 Tangentiale Berührungspunkte von P an einen Kreis

```
pt points[2]; // results
void boundaryPoints(pt P, pt M, double r){ // circle is
    given by center point M and radius r
  double dx,dy;
  dx = M.x; dy = M.y;
  P.x -= dx; P.y -= dy;
  double rq = r*r,ypq = P.y*P.y, xpq = P.x*P.x;
  if(P.y == 0){
    points[0].x = points[1].x = rq/P.x;
    points[0].y = r*sqrt(1-rq/(xpq));
    points[1].y = -points[0].y;
  }
  else{
    points[0].x = (P.y*sqrt(rq*ypq+rq*xpq-rq*rq)+rq*P.x)
      /(ypq+xpq);
    points[0].y = (rq - points[0].x*P.x)/P.y;
    points[1].x =
        -(P.y*sqrt(rq*ypq+rq*xpq-rq*rq)-rq*P.x)
      /(ypq+xpq);
    points[1].y = (rq - points[1].x*P.x)/P.y;
  }
  points[0].x += dx; points[1].x += dx;
  points[0].y += dy; points[1].y += dy;
}
```

HASH: 4a929eeba7932f297fab964380848cf1

1.7.8 Circle from three Points

center of a circle through p123

```
pt center(pt p1, pt p2, pt p3) {
  pt a1, a2, b1, b2;
  a1 = (p2 + p3) * 0.5;
  a2 = (p1 + p3) * 0.5;
  b1.x = a1.x - (p3.y - p2.y);
  b1.y = a1.y + (p3.x - p2.x);
  b2.x = a2.x - (p3.y - p1.y);
  b2.y = a2.y + (p3.x - p1.x);
  return lineIntersect(a1, b1, a2, b2);
}
```

HASH: b4124da681c2e58f8e6cea78c16922e5

1.7.9 Point on Bisection of Angle

r is point on the bisection line of $\angle p_1 p_2 p_3$

```
pt bcenter(pt p1, pt p2, pt p3) {
    double s1, s2, s3;
    s1 = len(p2-p3);
    s2 = len(p1-p3);
    s3 = len(p1-p2);
    double rt = s2 / (s2 + s3);
    pt a1, a2;
    a1 = p2 * rt + p3 * (1.0 - rt);
    rt = s1 / (s1 + s3);
    a2 = p1 * rt + p3 * (1.0 - rt);
    return lineIntersect(a1, p1, a2, p2);
}
```

HASH: 2f408495b499accbecb584a99bde4c9a

1.7.10 Circle/Line Intersection

1→only one intersec, 0 → normal, -1→no interdec

```
int lineAndCircle(pt& oo, double r, pt& p1, pt& p2, pt&
    r1, pt& r2) {
  pt m = closestpt(p1, p2, oo);
  pt v = p2 - p1;
  double l = len(v);
  v = v/l;
  double r0 = len(oo-m);
  if (r0 > r + eps) return -1;
  if (fabs(r0 - r) < eps) {
    r1 = r2 = m;
    return 1;
  }
  double dd = sqrt(r * r - r0 * r0);
  r1 = m - v * dd;
  r2 = m + v * dd;
  return 0;
}
```

HASH: 8dc0a5e9fb18942c134d09e6f8cbdb92

1.7.11 Circle/Circle Intersection

```
void rotate(pt p0, pt p1, double a, pt& r) {
  p1 = p1 - p0;
  r.x = cos(a) * p1.x - sin(a) * p1.y;
  r.y = sin(a) * p1.x + cos(a) * p1.y;
  r = r + p0;
}
// Intersection of two circles. Return 1 -> only one
// Intersection, 0 -> two intersections, -1 -> non
    intersections or oo.
// Returns the points in q1 and q2
ll CAndC(pt o1, double r1, pt o2, double r2, pt& q1, pt&
    q2) {
  double r = len(o1-o2);

  if (r1 < r2) {
    swap(o1, o2);
    swap(r1, r2);
  }
  if (r < eps) return -1;
  if (r > r1 + r2 + eps) return -1;
  if (r < r1 - r2 - eps) return -1;
  pt v = o2 - o1;
  double l = len(v);
  v = v/l;
  q1 = o1 + v * r1;
  if (fabs(r - r1 - r2) < eps || fabs(r + r2 - r1) <
      eps) {
    q2 = q1;
    return 1;
  }
  double a = cosSatz(r2, r, r1);
  q2 = q1;
  rotate(o1, q1, a, q1);
  rotate(o1, q2, -a, q2);
  return 0;
}
```

HASH: 5d93a0c6a3793282f4df053d6b331d63

1.8 Geometry - 3D

1.8.1 3D Point Structure

```
typedef double coord;

struct pt
{
  coord x, y, z;
  pt() : x(0), y(0), z(0) {}
  pt(coord x, coord y, coord z) : x(x), y(y), z(z) {}
  pt(coord phi, coord theta)
  {
    // φ ∈ [0; 2π)
    // θ ∈ [0; π) (π/2 = equator)
    x = r * sin(theta) * cos(phi);
    y = r * sin(theta) * sin(phi);
    z = r * cos(theta);
  }
  pt operator+(pt o) { return pt(x+o.x, y+o.y, z+o.z); }
  pt operator-(pt o) { return pt(x-o.x, y-o.y, z-o.z); }
  pt operator*(coord f) { return pt(x*f, y*f, z*f); }
  pt operator/(coord f) { return pt(x/f, y/f, z/f); }
  bool operator<(pt p) { return tie(x, y, z) < tie(p.x,
      p.y, p.z); }
  coord operator*(pt o) { return x*o.x+y*o.y+z*o.z; }
  pt operator^(pt o) { return pt(y*o.z-z*o.y,
      z*o.x-x*o.z, x*o.y-y*o.x); }
};

coord len(pt p) { return hypot(p.x, p.y, p.z); }
pt norm(pt p) { return p / len(p); }
coord theta(pt p) { return acos(p.z / hypot(p.x, p.y,
    p.z)); }
coord phi(pt p) { return atan2(p.y, p.x); }
coord greatCircleDistance(pt a, pt b) { return
    atan2(len(a^b), a*b); }
```

HASH: df02c49e1a56da8e8a814d35d556534e

1.8.2 Great Circle Stuff

```
/**
 * Returns 0 if great circles are identical, 1 if not.
 */
ll greatCircleIntersect(pt & out1, pt & out2, pt a0, pt
    a1, pt b0, pt b1)
{
  out1 = norm(a0^a1) ^ norm(b0^b1);
  if (out1 * out1 < eps)
    return 0;

  out1 = norm(out1);
  out2 = pt() - out1;
  return 1;
}

bool isOnGreatCircleSegment(pt a0, pt a1, pt b)
{
```

```
  return abs(greatCircleDistance(a0, a1) +
      greatCircleDistance(a0, b) -
      greatCircleDistance(b, a1)) < eps;
}

/**
 * Returns 0 if all points on the great circle have the
    same distance, 1 if p is on the great circle, 2
    otherwise.
 */
ll closestPt(pt & out, pt a, pt b, pt p)
{
  pt axb = norm(a^b);
  if (abs(axb * p) < eps)
  {
    out = len(a - p) < len(b - p) ? a : b;
    return 1;
  }
  pt tmp = axb^p;
  if (tmp * tmp < eps)
    return 0;
  out = norm(axb^norm(tmp));
  out = out * (out*p < 0 ? -1 : 1);
  return 2;
}
```

HASH: 288edc9bd205613d1c9a43c365932f3c

1.9 Strings

1.9.1 Substringsearch

Usage: call `kmpsetup();` to create pattern, and `kmpscan()` returns index of `pat` in `text`.

```
char text[1000000], pat[10000];
ll f[10000];

void kmpsetup() {
  ll i, k, len = strlen(pat);
  for (f[0] = -1, i = 1; i < len; i++) {
    k = f[i - 1];
    while (k >= 0)
      if (pat[k] == pat[i - 1])
        break;
      else
        k = f[k];
    f[i] = k + 1;
  }
}

ll kmpscan() {
  ll i, k, ret = -1, len = strlen(pat);
  for (i = k = 0; text[i];) {
    if (k == -1) {
      i++;
      k = 0;
    } else if (text[i] == pat[k]) {
      i++;
      k++;
      if (k >= len) {
        ret = i - len;
        break;
        // Alle Matches finden: break löschen;
        // i--; k--; k = f[k]
      }
    } else
      k = f[k];
  }
  return ret;
}
```

HASH: 4f75be6eca7ba581546232e3b44f7b32

1.9.2 String-Periode

```
// kmpsetup() is needed form above, put string into pat
int periode(){
  kmpsetup();
  int strl = strlen(pat);
  return strl - f[strl - 1] - 1;
}
```

HASH: d63e468e20de346e5936aed65510590d

1.9.3 Suffix Array

```
// sortiert alle Suffixe eines Strings lexikographisch
// Laufzeit O(nlogn) ≈ 100000 chars

#define MAX_N 100010
char T[MAX_N]; // the input string
ll n; // the length of input string
ll RA[MAX_N], tempRA[MAX_N]; // rank array
ll SA[MAX_N], tempSA[MAX_N]; // suffix array
ll c[MAX_N]; // for counting/radix sort

void countingSort(ll k) { // O(n)
  ll i, sum, maxi = max(300ll, n);
  memset(c, 0, sizeof(c));
  for(i = 0; i < n; i++)
  c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    ll t = c[i]; c[i] = sum; sum += t;
  }
  for (i = 0; i < n; i++)
  tempSA[c[SA[i]+k<n?RA[SA[i]+k]:0]++] = SA[i];
  for (i = 0; i < n; i++) SA[i] = tempSA[i];
}

void constructSA() {
  ll i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];
  for (i = 0; i < n; i++) SA[i] = i;
  for (k = 1; k < n; k <<= 1) {
    countingSort(k);
    countingSort(0);
    tempRA[SA[0]] = r = 0;
    for (i = 1; i < n; i++)
      tempRA[SA[i]] = (RA[SA[i]] == RA[SA[i-1]] &&
          RA[SA[i]+k] == RA[SA[i-1] + k]) ? r : ++r;
    for (i = 0; i < n; i++) RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;
  }
}
// call example
ll example() {
  n = (ll)strlen(gets(T));
  T[n++] = '$';
  constructSA();
  for (ll i = 0; i < n; i++)
  printf("%2lld %s\n", SA[i], T + SA[i]);
}
```

HASH: 16ba02188b4603b705d71509d9debd7c

1.9.4 Levenshtein/Edit-distance

Die Levenshtein-Distanz zwischen zwei Zeichenketten ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um die erste Zeichenkette in die zweite umzuwandeln.
If you only want to have the min edit-distance, omit the blue part.

```
#define NONE 0
#define REPLACE 1
#define DELETE 2
#define INSERT 3
#define MAXN 1000 // what ever you need
char w1[MAXN]; // Put first string in here
char w2[MAXN]; // Put second string in here
struct {
  ll d, op, pos;
} lf[MAXN + 1][MAXN + 1];

// Levenshtein-dance in O(n) returns min edit dance
// n and m are the lengths of the words
ll levenshtein(ll n, ll m) {
  lf[0][0].d = 0;
  lf[0][0].pos = 1;
  for (ll i = 1; i < n+1; ++i) {lf[i][0].d=i;
      lf[i][0].pos=1;}
  for (ll i = 1; i < m+1; ++i) {lf[0][i].d=i;
      lf[0][i].pos=i+1;}
  for (ll i = 1; i < n+1; ++i) {
    for (ll j = 1; j < m+1; ++j) {
      lf[i][j].d = min(lf[i-1][j-1].d+((w1[i-1] ==
          w2[j-1])?0:1),min(lf[i][j-1].d+1,
      lf[i-1][j].d+1));
      // BEGINN BLAU
      if (lf[i][j].d==lf[i-1][j].d+1){
        lf[i][j].op = DELETE;
        lf[i][j].pos = lf[i-1][j].pos;
        continue;
      }
      if (lf[i][j].d == lf[i][j-1].d + 1) {
        lf[i][j].op = INSERT;
        lf[i][j].pos = lf[i][j-1].pos + 1;
        continue;
      }
      if (lf[i][j].d == lf[i-1][j-1].d + 1) {
        lf[i][j].op = REPLACE;
        lf[i][j].pos = lf[i-1][j-1].pos + 1;
        continue;
      }
      lf[i][j].op = NONE;
      lf[i][j].pos = lf[i-1][j-1].pos + 1;
      // END BLAU
    }
  }
  return lf[n][m].d;
}

// Edit-Program, writes a program for the transform to
    the output
void edit_prog(ll n, ll m) { // let them be the length
    of the two words
  stack<pll> output;
  while (n != 0 && m != 0) {
    pll pos (n,m);
    if (lf[n][m].op != NONE) output.push(pos);
    ll x, y;
    x = n - ((lf[n][m].op != INSERT)?1:0);
    y = m - ((lf[n][m].op != DELETE)?1:0);
    n = x;
    m = y;
  }
  pll pos (n,m);
  if (lf[n][m].op != NONE)
  output.push(pos);
  if (n == 0)
  for (ll i = 0; i < lf[n][m].d; ++i) {
    cout << "I" << w2[i];
    if (i + 1 < 10) cout << "0";
    cout << i + 1;
  } else
  for (ll i = 0; i < lf[n][m].d; ++i)
    cout << "D" << w1[i] << "01";
  while (!output.empty()) {
    pll pos = output.top();
    n = pos.first; m = pos.second;
    ll outpos;
    switch (lf[n][m].op) {
    case INSERT:
      cout << "I" << w2[m - 1];
      outpos = lf[n][m-1].pos;
      if (outpos < 10) cout << "0";
      cout << outpos;
      break;
    case DELETE:
      cout << "D" << w1[m + (n - lf[n - 1][m].pos)];
      outpos = lf[n-1][m].pos;
      if (outpos < 10) cout << "0";
      cout << outpos;
      break;
    case REPLACE:
      cout << "C" << w2[m - 1];
      outpos = lf[n-1][m-1].pos;
      if (outpos < 10) cout << "0";
      cout << outpos;
      break;
    }
    output.pop();
  }
  cout << "E" << endl;
}
```

HASH: 45e112ddca36a3deaa6a1cd5b93bcaaf

1.10 Trees

1.10.1 Policy based data structures

```
using namespace __gnu_pbds;
typedef tree<ll, null_type, less<ll>, rb_tree_tag,
    tree_order_statistics_node_update> betterSet;
// supports order_of_key(k): #items strictly smaller
    than k, find_by_order(k): iterator to the kth
    element (0-based)
// use less_equal for multiset
```

HASH: 4c2ac7ccc5750a0301a06ffdb99434ac

1.10.2 Fenwick-Trees

```
// given (size) boxes you can add/remove elements to
    these boxes (add).
// with rank(r) you will know how many elements
// the boxes with index smaller than r contain
// all operations take log n
ll fwt_size, bi_tree[MAXN];
void clear(ll size){
  memset(bi_tree,0,sizeof(ll)*size);
  fwt_size = size;
}

void add(ll pos, ll val) {
  if (!pos) {
    bi_tree[0] += val;
    return;
  }
  while(pos < fwt_size) {
    bi_tree[pos] += val;
    pos += pos&(-pos);
```

```
  }
}

ll rank(ll pos) {
  ll res = bi_tree[0];
  while(pos) {
    res += bi_tree[pos];
    pos &= pos-1;
  }
  return res;
}
```

HASH: f47c0d882a7660f81d0137254071c414

1.10.3 Segment-Trees

```
ll arr[MAXN]; // Nur nötig für initialisierung
struct segtree{
    segtree *left, *right;
    ll from, to, mid, len;
    ll val, lazy;
};

segtree* build(ll from, ll to){
    segtree* t = new segtree();
    t->to = to;
    t->from = from;
    t->lazy = 0;
    t->mid = (from + to) / 2;
    t->len = to - from + 1;
    if(t->len > 1){
        t->left  = build(from, t->mid);
        t->right = build(t->mid+1, to);
        t->val = t->left->val + t->right->val;
    } else t->val = arr[to];
    return t;
}
void propagateLazy(segtree* t){
    t->val = t->val + (t->len) * t->lazy;
  if(t->len > 1){
        t->left->lazy = t->left->lazy + t->lazy;
        t->right->lazy = t->right->lazy + t->lazy;
    }
    t-> lazy = 0;
}
void updateRange(segtree * t, ll from, ll to, ll val){
    if(to < t->from || from > t->to) return;
    else if(from <= t->from && to >= t->to) t->lazy =
        t->lazy + val;
    else{
        updateRange(t->left, from, to, val);
        updateRange(t->right, from, to, val);
        t->val = t->left->val + t->left->lazy *
            t->left->len + t->right->val +
            t->right->lazy * t->right->len;
    }
}
void updatePoint(segtree * t, ll x, ll v){
    t->val = t->val + v;
    if(t->len == 1) return;
    if(x <= t->mid) updatePoint(t->left, x, v);
    else updatePoint(t->right, x, v);
}
ll getRange(segtree* t, ll from, ll to){
    if(to < t->from || from > t->to) return 0;
    propagateLazy(t); // Nur nötig, wenn updateRange
        verwendet wird
    if(from <= t->from && to >= t->to) return t->val;
    return getRange(t->left, from, to) +
        getRange(t->right, from, to);
}
```

HASH: a71ea2cbfad92671e0921755d70b0481

1.10.4 Splay-Trees

```
template <typename K = ll, typename V = ll, typename C =
    less<K>>
struct SplayTree {
  SplayTree *SplayTree::*L = &SplayTree::l,
      *SplayTree::*R = &SplayTree::r;
  const C comp = C();
  K k;
  V v; //to support multiple values per key: list<V> v;
  SplayTree *p, *l, *r;
  SplayTree(K key, V val) : k(key), v(val), p(0), l(0),
      r(0) {}
  ~SplayTree() { delete l, delete r; }
  void setC(SplayTree *t, SplayTree *SplayTree::*c) {
    this->*c = t;
          if (t) t->p = this;
  }
  SplayTree *rot(bool rRight) {
    SplayTree *res;
    if (rRight) res = l, setC(res->r, L),
        res->setC(this, R);
    else res = r, setC(res->l, R), res->setC(this, L);
    return res;
  }
  SplayTree *splay(K key) {
    if (k == key) return this;
    SplayTree *SplayTree::*c;
    bool rRight;
    if (comp(k, key)) c = &SplayTree::l, rRight = true;
    else c = &SplayTree::r, rRight = false;
    if (this->*c == 0) return this;
    auto res = this;
    if (comp((res->*c)->k, key) && (res->*c)->l)
        (res->*c)->setC((res->*c)->l->splay(key), L),
        res = rot(true);
    else if ((res->*c)->r != 0)
        (res->*c)->setC((res->*c)->r->splay(key), R),
        res->setC((res->*c)->rot(false), c);
    return !(res->*c) ? res : rot(rRight);
  }
  SplayTree *splayToRoot() {
    auto res = this;
    while (res->p) res = res->p->splay(res->k);
    return res;
  }
  SplayTree *insert(K key, V val) {
    auto res = this->splay(k);
    if (k == key) return res; //to support multiple
        values per key insert val into this->v before
        return
    auto n = new SplayTree(key, val);
    if (comp(k, key)) n->setC(res, R), n->setC(res->l,
        L), res->l = 0;
    else n->setC(res, L), n->setC(res->r, R), res->r =
        0;
    return n;
  }
  SplayTree *remove(K key) {
    auto res = splay(key);
    if (res->k != key) return res;
    if (!res->l) res = res->r;
    else {
      auto tmp = res->r;
      res = res->splay(key);
      res->setC(tmp, R);
```

```
    }
    return res;
  }
};
```

HASH: 64349a59d5081d427df14c914f2fd6b0

1.10.5 AVL-Trees

```
typedef ll T;
struct AVL{
  T val;
  ll ht, s;
  AVL *left, *right;
  AVL const * par;
  AVL(T v=0, AVL const*pa=0) :
      val(v),ht(0),s(1),left(0),right(0),par(pa){}
  ~AVL(){
    if(left) delete left;
    if(right) delete right;
  }
};
typedef AVL*AVL::*MF;
vector<MF> chlds{&AVL::left, &AVL::right};
ll size(AVL*t){ return t ? t->s : 0ll; }
ll height(AVL*t){ return t ? t->ht : -1ll; }
ll balanceFactor(AVL* t){ return t ? height(t->left) -
    height(t->right) : 0; }
void update(AVL*t){
  tie(t->ht,t->s) = pll(-1,1);
  for(auto child : chlds){
    auto c = t->*child;
    if(c){
      t->s += c->s;
      t->ht = max(t->ht, c->ht);
      c->par = t;
    }
  }
  t->ht++;
}
void rotate(AVL*& t, bool rotateRight){
  auto l = chlds[0], r = chlds[1];
  if(rotateRight) swap(l,r);
  AVL *tmp = t,
    *b = t->*r->*l;
  t = t->*r;
  t->*l = tmp;
  tmp->*r = b;
  update(tmp);
  update(t);
}
void balance(AVL*&t, AVL const * pa){
  update(t);
  ll b = balanceFactor(t);
  if(b > 1){
    if(balanceFactor(t->left) < 0) rotate(t->left,
        false);
    rotate(t, true);
  } else if(b < -1){
    if(balanceFactor(t->right) > 0) rotate(t->right,
        true);
    rotate(t, false);
  }
  t->par = pa;
}
AVL const * insert(AVL*& t, T val, AVL const * pa = 0){
  if(!t) return t = new AVL(val, pa);
  if(val == t->val) return t;
  auto res = insert(val < t->val ? t->left : t->right,
      val, t);
  balance(t,pa);
  return res;
}
AVL const * minValue(AVL const * t){
  if(!t) return 0;
  return t->left ? minValue(t->left) : t;
}
void erase(AVL*& t, AVL const * n, AVL const * pa=0){
  if(n==t){
    if(t->left && t->right){ // two children
      auto tmp = minValue(t->right);
      t->val = tmp->val;
      erase(t->right, tmp, t);
    } else t = t->left ? t->left : t->right;
  }else{
    if(n->val < t->val) erase(t->left, n, t);
    else         erase(t->right, n, t);
  }
  if(t) balance(t, pa);
}
AVL const * find(AVL* t, const T& v){
  if(!t || t->val == v) return t;
  return find(v < t->val ? t->left : t->right, v);
}
ll order(AVL const * t){
  ll res = size(t->left);
  for(AVL const * n = t->par; n; t=n, n=t->par)
    if(n->right == t)
      res += 1 + size(n->left);
  return res;
}
```

HASH: 0269a7ecbbe8b21b1e59aa6d305b269d

1.11 Search

1.11.1 Binary Search

```
// P(lo) muss falsch sein, P(hi) immer wahr.
// lo ist der größte Wert für den P nicht gilt
// hi ist der kleinste Wert für den P gilt
ll binarySearch() {
  ll lo, hi, mi;
  lo = ???;
  hi = ???;
  while (lo + 1 < hi) {
    mi = (lo + hi) / 2;
    if (P(mi)) hi = mi;
    else lo = mi;
  }
  return hi;
}
```

HASH: ce3d60b45dad5208f1b6d776d358e9ad

1.11.2 2-Pointer Search

Gegeben ein Array von Zahlen *vals* und eine Zahl M finde zwei Elemente des Arrays deren Summe M ist.
Ausgabe sind *lop* und *hop*.

```
void pointer2search(ll N, ll* vals) {
  ll M;
  sort(vals,vals+N);
  ll cup = N-1;
  ll clo = 0;
  ll hop = oo;
  ll lop = -1;
  ll cbst = 0;
  while(cup != clo){
    if (vals[cup] + vals[clo] <= M &&
```

```
      vals[cup] + vals[clo] >= cbst){
    hop = cup; lop = clo; cbst = vals[cup] +
        vals[clo];
    }
    if (vals[cup] + vals[clo] <= M) clo++;
    else cup--;
  }
}
```

HASH: 6d5033a17538c6c7cdd836494aef1e8d

1.12 Misc

1.12.1 Longest Common Subsequence (Hunt-Szymanski)

Nicht mit "longest common substring" verwechseln. In LC-Subsequence können Buchstaben ausgelassen werden, in LC-String nicht.

```
char w1[MAXN],w2[MAXN]; // in: strings
ll n1,n2; // in: string lengths
ll kk[MAXN]; // temp
vll buck[256]; // temp: sorting buckets
// time: O((r + n1 + n2) log n1) with r being the number
    of matching character pairs
ll lcs() {
  for (ll j = 0; j < 256; ++j) buck[j].clear();
  for (ll i = 0; i < n1+1; ++i) kk[i] = oo;
  kk[0]=-1;
  for (ll i = n2 - 1; i >= 0; --i)
      buck[w2[i]].push_back(i);
  for (ll i = 0; i < n1; ++i) for (auto j : buck[w1[i]])
    *(lower_bound(kk,kk+n1+1,j)) = j;
  for (ll i = n1; i >= 0; --i) if(kk[i]<oo) return i;
}
```

HASH: 87b797ccf2795ec78fbe60e8c57f9850

1.12.2 Longest Increasing Subsequence

```
vll LIS(vll A) {
  ll N = A.size(),j =- 1;
  vll pre(N,-1),erg;
  map<ll,ll> m;
  map<ll,ll>::iterator k,l;
  for (ll i = 0; i < N; ++i) if (m.insert(pll(A[i],
      i)).second) {
    l = k = m.find(A[i]);
    if (l == m.begin()) pre[i] = -1;
    else pre[i] = (--l)->second;
    if ((++k) != m.end()) m.erase(k);
  }
  for (j = (--(k = m.end()))->second; j != -1; j =
      pre[j])
  erg.push_back(A[j]);
  reverse(erg.begin(),erg.end());
  return erg;
}
```

HASH: 911d86a1c841a334b5bc1ef9ae2b716a

1.12.3 Maximum Sum 2D

br is beginning row,bc is beginning column, er is ending row, ec is ending column, all initialized with 0,S is maximum Sum, initialized with S=1<<31

```
void maxSum2D(ll N, ll a[100][100], ll &br, ll &er, ll
    &bc, ll &ec, ll &S){
  ll pr[100];
  ll s = 0, t = 0, k, l, j;
  for (ll z = 0; z < N; ++z){
    for (ll i = 0; i < N; ++i) pr[i] = 0;
    for (ll x = z; x < N; ++x){
      t = j = k = l= 0;
      s = 1<<31;
      for (ll i = 0; i < N; ++i){
        pr[i] = pr[i] + a[x][i];
        t = t + pr[i];
        if(t > s){
          s = t;
          k = i;
          l = j;
        }
        if(t < 0){
          t = 0;
          j = i + 1;
        }
      }
      if(s > S){
        S = s;
        er = x;
        ec = k;
        br = z;
        bc = l;
      }
    }
  }
}
```

HASH: 7dc3aaa6a5bcc3ac5b8a559b396ec597

1.12.4 Inversion Counting

array with inversions, size of the array with sizeof.

```
ll mac(ll array[], ll size){
  ll m;
  if(size <= 1) return 0;
  m = size/2;
  ll invCountA = 0, invCountB = 0, invCountC = 0;
  invCountA = mac(array, m);
  invCountB = mac(array + m, size - m);
  ll partA[m], partB[size-m];
  memcpy(partA,array, sizeof(array) * m);
  memcpy(partB, array + m, sizeof(array) * (size - m));
  ll i = 0, j = 0, k = 0;
  while(k < size){
    if(partA[i] < partB[j]){
      array[k] = partA[i];
      i++;
      invCountC += j;
    }
    else {
      array[k] = partB[j];
      j++;
      invCountC += 1;
    }
    k++;
    if(i >= m || j >= (size - m)) break;
  }
  invCountC -= j;
  while(i < m){
    array[k] = partA[i];
    k++;
    i++;
    invCountC += j;
  }
  while(j < (size - m)){
    array[k] = partB[j];
    k++;
    j++;
  }
```

```
  return (invCountA + invCountB + invCountC);
}
```

HASH: accb546ad74b03ec12a3e88168e24f45

1.12.5 2-SAT

```
ll varcnt;
ll as[MAXV];

ll neg(ll var){
    return  2 * varcnt - 1 - var;
}

void addImplic(ll lit1, ll lit2){
    adj[lit1][0].push_back(lit2);
    adj[neg(lit2)][0].push_back(neg(lit1));

    adj[lit2][1].push_back(lit1);
    adj[neg(lit1)][1].push_back(neg(lit2));
}

void addClause(ll lit1, ll lit2){
    adj[neg(lit1)][0].push_back(lit2);
    adj[neg(lit2)][0].push_back(lit1);

    adj[lit2][1].push_back(neg(lit1));
    adj[lit1][1].push_back(neg(lit2));
}

void assign(ll var){
    if(as[var] >= 0) return;
    as[var] = 1;
    as[neg(var)] = 0;
    for(int j : adj[var][0]) assign(j);
}

bool solve(){
   kosaraju(2*varcnt);
   for(int v=0; v < varcnt; v++)
       if(comp[v] == comp[neg(v)]) return false;
   memset(as, -1, sizeof(as));
   for(int i = 4 * varcnt - 1; i >= 2 * varcnt; i--)
       assign(st[i]);
   return true;
}
```

HASH: 7fefcb82ff1855cfabf7dce202fb8b51

1.12.6 128 Bit Integer

```
typedef __int128_t lll;
typedef __uint128_t ulll;
ostream & operator<< (ostream & os, ulll val) {
  if (val < 10) os << (int) val;
  else os << (ulll) (val / 10) << (int) (val % 10);
  return os;
}
ostream & operator<< (ostream & os, lll val) {
  if (val < 0) os << "-" << (ulll) (~val + 1);
  else os << (ulll) val;
  return os;
}
istream & operator>> (istream & is, lll & val) {
  string in;
  is >> in;
  val = 0;
  lll sign = 1;
  if (in[0] == '+' || in[0] == '-') {
    if (in[0] == '-') sign = -1;
    in.erase(0, 1);
  }
  for (size_t i = 0; i < in.length(); ++i)
  val = (val * 10) + (int) (in[i] - '0');
  val *= sign;
  return is;
}
istream & operator>> (istream & is, ulll & val) {
  return is >> ((lll &) val);
}
```

HASH: f75cc67be236423e43a584b8b6d450aa

1.12.7 Bit Permutation

```
ull nextBitperm(ull v)
{
  ull t = v | (v - 1l);
  return (t + 1l) | (((~t & -~t) - 1l) >>
      (__builtin_ctzll(v) + 1l));
}
```

HASH: 5a7d00eb00bd2304e0903f3d773d0990

1.12.8 FFT Stuff

```
typedef complex<double> cmplx;

void fft(vector<cmplx> & a, bool inv)
{
  ll n = a.size();
  if (n <= 1)
    return;
  vector<cmplx> even(n/2), odd(n/2);
  for (ll i = 0; 2*i < n; i++)
  {
    even[i] = a[i*2];
    odd[i] = a[i*2 + 1];
  }
  fft(even, inv);
  fft(odd, inv);
  cmplx x = 1, z = exp((cmplx)2i*M_PI/double(inv ? -n :
      n));
  for (ll k=0; 2*k<n; k++, x*=z)
  {
    a[k] = even[k] + x*odd[k];
    a[k + n/2] = even[k] - x*odd[k];
    if (inv)
    {
      a[k] /= 2.;
      a[k + n/2] /= 2.;
    }
  }
}

vector<ll> mul(const vector<ll>& a, const vector<ll>& b)
{
  vector<cmplx> fa(a.begin(), a.end());
  vector<cmplx> fb(b.begin(), b.end());
  ll n = 1 << (32ll - __builtin_clzll(a.size() +
      b.size() - 1));
  fa.resize(n);
  fb.resize(n);

  fft(fa, false);
  fft(fb, false);
  for (ll i = 0; i < n; i++)
    fa[i] *= fb[i];
  fft(fa, true);
  vector<ll> res(n);
  for (ll i = 0; i < n; i++)
    res[i] = real(fa[i]) + 0.5;
```

```
  return res;
}

vector<ll> sqr(const vector<ll> &a)
{
  vector<cmplx> f(a.begin(), a.end());
  ll n = 1 << (32 - __builtin_clz(2 * a.size() - 1));
  f.resize(n);
  fft(f, false);
  for (ll i = 0; i < n; i++)
    f[i] *= f[i];
  fft(f, true);
  vector<ll> res(n);
  for (ll i = 0; i < n; i++)
    res[i] = real(f[i]) + 0.5;
  return res;
}
```

HASH: e757a34636f7dd90fab8b9e54662e2fb

2 Der ultimative Matheteil

2.1 Kombinatorik

Aus n Elementen k Elemente auswählen
Mit zurücklegen, mit Reihenfolge: n^k
Mit zurücklegen, ohne Reihenfolge: $\binom{n+k-1}{k}$
Ohne zurücklegen, mit Reihenfolge: $n*(n-1)*...*(n-k+1)$
Ohne zurücklegen, ohne Reihenfolge: $\binom{n}{k}$
Anzahl der Rechtecke in einem $m \times n$ Rechteck: $\binom{n}{2} * \binom{m}{2}$
Anzahl der Gitterwege von (0,0) nach (n,m) (immer nach rechts oder nach oben laufen) in einem Rechteck: $\binom{n+m}{n}$
Anzahl, wenn man die Hauptdiagonale (im Quadrat) nicht überqueren darf: n-te Catalan-Zahl.

Catalan-Zahlen:
$C_m = \frac{1}{1+n}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$
Anzahl der

- Triangulationen eines regelmäßigen n-Ecks
- geordneten, vollen binären Bäume mit $n+1$ Blättern
- möglichen Klammerausdrücke mit n Klammerpaaren
- Möglichkeiten ein Produkt aus $n+1$ Faktoren zu klammern

2.2 Ein paar Rechenregeln

$$\sum_{k=1}^{n} k = \frac{n*(n+1)}{2} \tag{1}$$

$$\sum_{k=1}^{n} k^2 = \frac{n*(n+1)*(2n+1)}{6} \tag{2}$$

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} \tag{3}$$

$$\sum_{k=0}^{n} c^k = \frac{c^{n+1}-1}{c-1} \tag{4}$$

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} \tag{5}$$

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \tag{6}$$

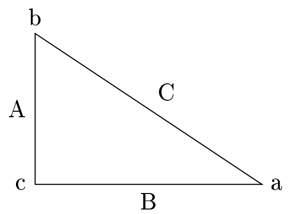
$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n} \tag{7}$$

$$\sum_{k=0}^{n} \binom{m+k}{k} = \binom{m+n+1}{n} \tag{8}$$

$$\binom{n}{2} = \binom{k}{2} + k*(n-k) + \binom{n-k}{k} \tag{9}$$

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n-k+i}{i} \tag{10}$$

2.3 Trigonometrie



$sin\ a = \frac{A}{C}$, $cos\ a = \frac{B}{C}$,
$tan\ a = \frac{sin\ a}{cos\ a} = \frac{A}{B}$, $cot\ a = \frac{1}{tan\ a}$
$cos\ x = cos(-x), sin\ x = -sin\ x$
$cos(x \pm y) = cos(x)cos(y) \mp sin(x)sin(y)$
$sin(x \pm y) = sin(x)cos(y) \pm sin(y)cos(x)$

Im allgemeinen Dreieck gelten folgende Regeln:
Sinussatz: $\frac{A}{sin(a)} = 2r$, (A ist hier die Seite, nicht der Flächeninhalt!) wobei r der Radius des Umkreises ist.
Cosinussatz: $A^2 = B^2 + C^2 - 2BC * cos(a)$
Mit diesen beiden Sätzen lassen sich fehlende Winkel und Seiten berechnen.
Flächeninhalt F:

$$F = \frac{1}{2} * B * C * sin(a)$$

2.4 Zahlentheorie

erweiterter euklidischer Algorithmus:
Berechnet s und t:
ggT(a,b) = s*a+t*b
$r_0 = a, r_1 = b, s_0 = 1, s_1 = 0,$
$t_0 = 0, t_1 = 1$
$q_k = \frac{r_{k-1}}{r_k}$, $r_{k+1} = r_{k-1} - q_k * r_k$,
$s_{k+1} = s_{k-1} - q_k * s_k$,
$t_k = (r_k - s_k * a) : b$. bis $r_{k+1} = 0$. Dann $r_k = ggT(a,b) = s_k * a + t_k * b$.
Kleinstes gemeinsames Vielfaches:

$kgV(n,m) = n \cdot m / ggT(n,m)$
Chinesischer Restsatz:
$m_1, ..., m_n$ teilerfremd, $a_1, ..., a_n$ ganze Zahlen. $\Rightarrow \exists\, x : x \equiv a_i(m_i)$
Dieses x finden:
$M = m_1 * ... * m_n, M_i = \frac{M}{m_i}$
Bestimme mit erw. eukl. Algo r_i, s_i:
$r_i * m_i + s_i * M_i = 1$

$$\Rightarrow x = \sum_{i=1}^{n} a_i * s_i * M_i$$

Kleiner Fermat:
$a^{\varphi(n)} \equiv 1 \mod n$ für a, n teilerfremd
Euler-Phi (Anzahl der kleineren teilerfremden Zahlen):
$x = \prod_{i=1}^{n} p_i^{e_i}$ Primfaktorzerlegung
$\Rightarrow \varphi(x) = \prod_{i=1}^{n} p_i^{e_i - 1}(p_i - 1)$
Lineare Kongruenzen:
Gleichung: ax + by = c, g = ggT(a,b). Die Gleichung hat keine Lösung wenn $g \nmid c$. Mit dem erweiterten euklidischen Algorithmus findet man eine Lösung.
Sucht man eine positive Lösung, kann man verwenden, dass alle Lösungen die Form $x_k = x_0 + \frac{b}{g}k, y_k = y_0 - \frac{a}{g}k$ haben, wobei x_0, y_0 eine Lösung ist. Daraus ergeben sich mit der Bedingung, dass x und y positiv sind zwei Ungleichungen, für die es Lösungen geben kann oder nicht.
Modulorechnen mod n:
Division von $\frac{a}{b}$ entspricht Multiplikation von a mit der Inversen von b mod n. **Vorsicht:** existiert nur dann, wenn n und b teilerfremd sind.
Berechnung der Inversen:
Sind b und n teilerfremd, so erhält man mit dem erweiterten euklidischen Algorithmus x und y, sodass: $bx + ny = 1$. Dann ist x die Inverse von b. Ist n eine Primzahl, ist die Berechnung einfacher. Dann gilt mit dem kleinen Satz von Fermat: $b^{n-1} = b^{-1}$.
Frobenius-Zahl:
Die Frobenius-Zahl von $a_1, ..., a_n$ ist die größte Zahl, die man nicht als Summe von $a_1, ..., a_n$ darstellen kann. Sie existiert nur, wenn $gcd(a_1, ..., a_n) = 1$, da sonst alle durch Summation entstehenden Zahlen durch diesen gcd teilbar sind. Für $n = 2$ ist die Frobeniuszahl $m = a_1 * a_2 - a_1 - a_2$. Für größere n ist keine direkte Formel bekannt. Es gilt aber: $m(a_1, ..., a_n) \leq min(a_i) * max(a_i) - min(a_i) - max(a_i)$. Mit Dijkstra kann man die Frobenius-Zahl für beliebige n berechnen:
1. Betrachte die Gleichung $m \equiv x_2 a_2 + ... + x_n a_n \ mod\ a_1$.
2. Konstruiere Digraph mit Knoten von 0 bis a_{n-1}. Dabei existiert ein Bogen von u nach v, wenn $u + a_i \equiv v \ mod\ a_1$. Der Bogen erhält das Gewicht a_i.
3. Suche mit Dijkstra alle kürzesten Wege vom Knoten 0 aus. Die Länge des längsten der kürzesten Wege sei D. Dann ist die Frobenius-Zahl $D - a_1$.
Sucht man nun die Darstellung einer Zahl t, so sucht man nach dem kürzesten Weg von 0 zu $t \ mod\ a_1$. Die Kantengewichte auf diesem Weg geben an, welche der a_i man wählen muss. Was dann noch fehlt ist $\equiv 0 \ mod\ a_1$ und lässt sich daher mit Vielfachen von a_1 auffüllen.

2.5 Geometrie

2.5.1 Vektoren

Vektor orthogonal zu $\begin{pmatrix} a \\ b \end{pmatrix}$: $\begin{pmatrix} -b \\ a \end{pmatrix}$

2.5.2 Geraden und Ebenen

Hesse-Normalform: Darstellung einer Gerade/Ebene durch den Abstand zum Ursprung und einen Normalenvektor
n Normalenvektor, $n_0 = \frac{n}{|n|}$,
a Abstand zum Ursprung
$g : n_0 * (x - a) = 0$
* ist hierbei Skalarprodukt
Das Einsetzen eines Punktes ergibt den Abstand dieses Punktes zur Gerade/Ebene
Schnitt Gerade/Ebene:
Ausmultiplizieren der Normalform bringt Ebenengleichung in Koordinatenform. Einsetzen der Geradengleichung in die Ebenengleichung.
$g : x = a + \lambda b$, $E : c_1x_1 + c_2x_2 + c_3x_3 - d = 0$
Fall 1: $c_1b_1 + c_2b_2 + c_3b_3 = 0$
Fall 1a): $c_1a_1 + c_2a_2 + c_3a_3 - d \neq 0 \ \Rightarrow$ kein Schnitt
Fall 1b): $c_1a_1 + c_2a_2 + c_3a_3 - d = 0 \ \Rightarrow$ Gerade liegt in Ebene
Fall 2: $c_1b_1 + c_2b_2 + c_3b_3 \neq 0 \Rightarrow \lambda = -\frac{c_1a_1 + c_2a_2 + c_3a_3 - d}{c_1b_1 + c_2b_2 + c_3b_3}$, Schnitt bei $p = a + \lambda b$
Schnitt Ebene/Ebene:
Eine Ebene in Koordinatenform, eine in Parameterform. Einsetzen.
$E_1 : x = a + \lambda b + \mu c$, $E_2 : d_1x_1 + d_2x_2 + d_3x_3 - e = 0$
Fall 1: $d_1b_1 + d_2b_2 + d_3b_3 = 0, d_1c_1 + d_2c_2 + d_3c_3 = 0$
Fall 1a): $d_1a_1 + d_2a_2 + d_3a_3 - e = 0 \Rightarrow$ Ebenen sind identisch
Fall 1b): $d_1a_1 + d_2a_2 + d_3a_3 - e \neq 0 \Rightarrow$ Ebenen sind parallel
Fall 2: $d_1b_1 + d_2b_2 + d_3b_3 = 0, d_1c_1 + d_2c_2 + d_3c_3 \neq 0 \Rightarrow$
Schnitt bei $p = a + \lambda b - \frac{d_1a_1 + d_2a_2 + d_3a_3 - e}{d_1c_1 + d_2c_2 + d_3c_3} c$
Fall 3: $d_1b_1 + d_2b_2 + d_3b_3 \neq 0, d_1c_1 + d_2c_2 + d_3c_3 = 0 \Rightarrow$
Schnitt bei $p = a - \frac{d_1x_1 + d_2x_2 + d_3x_3 - e}{d_1b_1 + d_2b_2 + d_3b_3} b + \mu c$
Fall 4: $d_1b_1 + d_2b_2 + d_3b_3 \neq 0, d_1c_1 + d_2c_2 + d_3c_3 \neq 0 \Rightarrow$
Schnitt bei $p = a + \lambda b - \frac{d_1a_1 + d_2a_2 + d_3a_3 - e + \lambda(d_1b_1 + d_2b_2 + d_3b_3)}{d_1c_1 + d_2c_2 + d_3c_3} c$

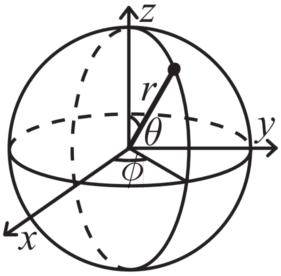
2.5.3 Polygone

Inkreise:
Der Inkreis existiert, wenn sich alle Winkelhalbierenden in einem Punkt schneiden. Dann gilt für Umfang u und Fläche A: $r = \frac{2A}{u}$
Pick's Theorem:
Für jedes Polygon, dessen Eckpunkte auf Integer-Koordinaten liegen gilt $A = I + \frac{R}{2} - 1$, wobei A der Flächeninhalt, I die Anzahl der Punkte mit Integer-Koordinaten echt innerhalb des Polygons und R die Zahl der Integer-Punkte auf dem Rand.
Transversalen im Dreieck:
- Die Mittelsenkrechten schneiden sich im Umkreismittelpunkt
- Die Höhen schneiden sich in einem Punkt
- Die Seitenhalbierenden schneiden sich im Schwerpunkt. Dieser teilt die Seitenhalbierenden im Verhältnis 2:1
- Die Winkelhalbierenden schneiden sich im Inkreismittelpunkt
- Satz von Ceva: Schneiden sich drei Ecktransversalen in einem Punkt, ist das Produkt der Teilverhältnisse gleich 1
- Satz von Menelaos: Schneidet eine Gerade ein Dreieck in zwei Seiten und der Verlängerung der dritten Seite, ist das Produkt der Teilverhältnisse gleich -1
- Südpolsatz: die Mittelsenkrechte und die Winkelhalbierende des gegenüberliegenden Winkels schneiden sich auf dem Umkreis
Sehnen, Sekanten, Tangenten:
- Sehnensatz: Schneiden sich zwei Sehnen AC und DB in S, so gilt $AS \cdot CS = BS \cdot DS$
- Sekantensatz: Schneiden sich zwei Sekanten AD und BC in P, so gilt $AP \cdot DP = BP \cdot CP$
- Tangentensatz: Schneiden sich zwei Tangenten mit T1 und T2 Kreispunkte in P, so gilt: $PT1 = PT2$
- Sekanten-Tangenten-Satz: Schneiden sich eine Sekante AB und eine Tangente mit Kreispunkt T in P, so gilt $AP \cdot BP = PT^2$
Diverses:
- Im Parallelogramm halbieren die Diagonalen einander
- Im Trapez ist der Abstand der Mittelpunkte der Diagonalen gleich der halben Differenz der Längen der beiden Grundlinien

2.5.4 Kreise und Kugeln

Kreise:
Kreissektor: Ausschnitt des Kreises vom Mittelpunkt zu zwei Punkten auf der Kreislinie. Flächeninhalt $F = \frac{b*r}{2}$, wobei $b =$

$\frac{\alpha * \pi * r}{180 \deg}$ die Bogenlänge des Sektors ist, mit α Mittelpunktswinkel und r Radius.
Kreissegment: begrenzt durch Kreisbogen und Kreissehne. Flächeninhalt $F = \frac{r^2}{2} * (\alpha - sin(\alpha)$, wobei r Radius und α der Mittelpunktswinkel.



Kugelzweieck:
Fläche, die zwischen zwei Großkreisen entsteht (der kleinere Teil).
Seitenlängen: der halbe Umfang des Großkreises.
Flächeninhalt: $A = \frac{\gamma}{90°} \cdot \pi r^2$, γ ist die Größe des Innenwinkels im Gradmaß
γ im Bogenmaß $\Rightarrow$ $A = \gamma \cdot 2r^2$
Kugeldreieck:
Teil der Kugeloberfläche, der von drei Großkreisbögen begrenzt wird.
Flächeninhalt: $A_D = (\alpha + \beta + \gamma - \pi) \cdot r^2$, α, β, γ Winkel des Dreiecks im Bogenmaß.
Trigonometrie im Kugeldreieck:
Es sind α, β, γ die Winkel des Dreiecks und a, b, c die Seiten. Es gilt:
Sinussatz: $sin(\alpha) : sin(\beta) : sin(\gamma) = sin(a) : sin(b) : sin(c)$
Cosinusseitensatz: $cos(a) = cos(b) * cos(c) + sin(b) * sin(c) * cos(\alpha)$
Cosinuswinkelsatz: $cos(\alpha) = -cos(\beta) * cos(\gamma) + sin(\beta) * sin(\gamma) * cos(a)$
Mit diesen Sätzen lassen sich bei 3 gegebenen Stücken des Kugeldreiecks die übrigen berechnen.

2.5.5 Volumina und Oberflächen

Körper	Volumen	Oberfläche	Kommentar
Zylinder	$\pi \cdot r^2 \cdot h$	$2\pi r \cdot (r + h)$	
Pyramide	$\frac{1}{3} A_G h$	$A_G + A_M$	A_G Grundfläche, A_M Mantelfläche
Kreiskegel	$\frac{1}{3} \cdot \pi \cdot r^2 \cdot h$	$r \cdot \pi \cdot (r + s)$	nur für senkrechte Kegel, Außerdem: $s^2 = r^2 + h^2$
Kugel	$\frac{4}{3} \cdot \pi \cdot r^3$	$4 \cdot \pi \cdot r^2$	
Kugelkappe		$2 \cdot r \cdot \pi \cdot h$	
Kugelsegment	$\frac{h^2 \cdot \pi}{3} \cdot (3r - h)$	$2 \cdot r \cdot \pi \cdot h + (h \cdot (2r - h))^2 \pi$	Kugelkappe, die mehr als die Hälfte der Kugel ausmachen kann
Kugelschicht	$\frac{1}{6} \pi \cdot h (3 \cdot a^2 + 3 \cdot b^2 + h^2)$	$\pi \cdot (2 \cdot r \cdot h + a^2 + b^2)$	a Radius des unteren Schnitkreises, b des oberen

2.6 Kruscht und Krempel

Teilbarkeitsregeln

Teilbar durch	Bedingung
2	Zahl endet auf 0,2,4,6,8
3	Quersumme teilbar durch 3
4	letzten 2 Ziffern der Zahl durch 4 teilbar
5	Zahl endet auf 0 oder 5
6	teilbar durch 2 und 3
7	keine bekannte einfache Regel
8	letzten 3 Ziffern der Zahl durch 8 teilbar
9	die Quersumme ist durch 9 teilbar
10	Zahl endet auf 0
11	alternierende Quersumme durch 11 teilbar (abwechselnd addieren und subtrahieren)

Anzahl Stellen a einer Zahl n in Basis b
Es gilt: $a = \lfloor \log_b(n) \rfloor + 1$. Mit einem Basiswechsel folgt: $a = \lfloor \frac{ln(n)}{ln(b)} \rfloor + 1$. Möchte man die k-te Potenz von n betrachten, so folgt: $a = \lfloor \frac{k * ln(n)}{ln(b)} \rfloor + 1$
Lemma von Burnside:
Es gilt: $|X/G| = \frac{1}{|G|} * \sum_{g \in G} |X^g|$
Beispiel Perlenkette mit 5 Perlen in 3 Farben:
Welche Gruppe liegt zugrunde? Hier Diedergruppe: Drehung um 0 bis 4 Perlen, Achsenspiegelung an jeder der 5 Perlen $\Rightarrow |G| = 10$
Wie viele Fixpunkte hat welches Element? Hier:
Drehung um 0 Perlen ändert nichts $\rightarrow 3^5$ Möglichkeiten die Perlen zu platzieren. Drehung um 1 bis 4 Perlen sorgt dafür, dass alle Perlen gleich sein müssen $\rightarrow$ 3 Möglichkeiten. Spiegelung an jeder der 5 Perlen liefert 2 Bahnen mit 2 Elementen und 1 Bahn mit einem $\rightarrow 3^3$ Möglichkeiten.
Somit ergibt sich: $|X/G| = \frac{1}{10} * (1 * 3^5 + 4 * 3 + 5 * 3^3) = 39$.

3 typische Fehler

- alle Angaben eingelesen und abgespeichert?
- korrektes Ausgabeformat?
- Debugausgaben vergessen?
- alle $>, <$ Zeichen richtig rum?
- Vorzeichen korrekt?
- korrekte Datentypen?
- clearen?
- einfach mal ableiten?
- Sonderfälle? (0, 1, groß)
- Overflow? (int, long long, int128?)
- **Problemstellung korrekt gelesen?**

4 GCC builtins

Bei allen Funktionen kann hinten `l` bzw. `ll` angehängt werden für `long` bzw. `long long`.

- `int __builtin_ffs (int x)`
  Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.

- `int __builtin_clz (unsigned int x)`
  Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.

- `int __builtin_ctz (unsigned int x)`
  Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.

- `int __builtin_clrsb (int x)`
  Returns the number of leading redundant sign bits in x, i.e. the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.

- `int __builtin_popcount (unsigned int x)`
  Returns the number of 1-bits in x.

- `int __builtin_parity (unsigned int x)`
  Returns the parity of x, i.e. the number of 1-bits in x modulo 2.

- `bool __builtin_add_overflow (t1 a, t2 b, t3 *res)`
  These built-in functions promote the first two operands into infinite precision signed type and perform addition on those promoted operands. The result is then cast to the type the third pointer argument points to and stored there. If the stored result is equal to the infinite precision result, the built-in functions return false, otherwise they return true. As the addition is performed in infinite signed precision, these built-in functions have fully defined behavior for all argument values.
  This built-in function allows arbitrary integral types for operands and the result type must be pointer to some integral type other than enumerated or boolean type.
  The compiler will attempt to use hardware instructions to implement these built-in functions where possible, like conditional jump on overflow after addition, conditional jump on carry etc.

- `bool __builtin_sub_overflow (t1 a, t2 b, t3 *res)`
  This built-in function is similar to the add overflow checking built-in function above, except it performs subtraction, subtracting the second argument from the first one, instead of addition.

- `bool __builtin_mul_overflow (t1 a, t2 b, t3 *res)`
  This built-in function aris similar to the add overflow checking built-in function above, except it performs multiplication, instead of addition.