UNIWERSYTET
MIKOŁAJA KOPERNIKA
W TORUNIU

Nicolaus Copernicus University

# NCU1

Paweł Czarkowski, Mikołaj Juda, Łukasz Skabowski

AMPPZ 2025 / CERC 2025 / EUC 2026 / World Finals 2026

2025-03-11

# Headers (1)

## headers includes: <bits/stdc++.h> Główny nagłówek

```
using namespace std;
using LL=long long;
#define FOR(i,l,r)for(int i=(l);i<=(r);++i)
#define REP(i,n)FOR(i,0,(n)-1)
#define ssize(x)int(x.size())
#ifdef DEBUG
auto&operator<<(auto&o,pair<auto,auto>p){return o<<"("
  <<p.first<<", "<<p.second<<")";}
auto operator<<(auto&o,auto x)->decltype(x.end(),o){o
  <<"{";int i=0;for(auto e:x)o<<","+!i++<<e;return o<<
  "}";}
#define debug(X...)cerr<<"["#X"]: ",[](auto...$){((
  cerr<<$<<"; "),...)<<endl;}(X)
#else
#define debug(...){}
#endif
int main() {
  cin.tie(0)->sync_with_stdio(0);
}
```

## gen.cpp Dodatek do generatorki

```
mt19937 rng(random_device{}());
int rd(int l, int r) {
  return uniform_int_distribution<int>(l, r)(rng);
}
```

## freopen.cpp Kod do IO z/do plików

```
#define PATH "fillme"
  assert(strcmp(PATH, "fillme") != 0);
#ifndef LOCAL
  freopen(PATH ".in", "r", stdin);
  freopen(PATH ".out", "w", stdout);
#endif
```

# Wzorki (2)

## 2.1 Równości

$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$, Wierzchołek paraboli $= (-\frac{b}{2a}, -\frac{\Delta}{4a})$, $ax + by = e \wedge cx + dy = f \implies x = \frac{ed-bf}{ad-bc} \wedge y = \frac{af-ec}{ad-bc}$.

## 2.2 Pitagoras

Trójki $(a, b, c)$, takie że $a^2 + b^2 = c^2$: Jest $a = k \cdot (m^2 - n^2),\ b = k \cdot (2mn),\ c = k \cdot (m^2 + n^2)$, gdzie $m > n > 0, k > 0, m \perp n$, oraz albo $m$ albo $n$ jest parzyste.

## 2.3 Generowanie względnie pierwszych par

Dwa drzewa, zaczynając od $(2, 1)$ (parzysta-nieparzysta) oraz $(3, 1)$ (nieparzysta-nieparzysta), rozgałęzienia są do $(2m - n, m)$, $(2m + n, m)$ oraz $(m + 2n, n)$.

## 2.4 Liczby pierwsze

$p = 962592769$ to liczba na NTT, czyli $2^{21} \mid p - 1$. Do hashowania: 970592641 (31-bit), 31443539979727 (45-bit), 3006703054056749 (52-bit). Jest 78498 pierwszych $\leq$ 1 000 000. Generatorów jest $\phi(\phi(p^a))$, czyli dla $p > 2$ zawsze istnieje.

## 2.5 Liczby antypierwsze

| $lim$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
|---|---|---|---|---|---|---|---|
| $n$ | 60 | 840 | 7560 | 83160 | 720720 | 8648640 | 73513440 |
| $d(n)$ | 12 | 32 | 64 | 128 | 240 | 448 | 768 |

| $lim$ | $10^9$ | $10^{12}$ | $10^{15}$ |
|---|---|---|---|
| $n$ | 735134400 | 963761198400 | 866421317361600 |
| $d(n)$ | 1344 | 6720 | 26880 |

| $lim$ | $10^{18}$ |
|---|---|
| $n$ | 897612484786617600 |
| $d(n)$ | 103680 |

## 2.6 Dzielniki

$\sum_{d|n} d = O(n \log \log n)$

## 2.7 Lemat Burnside'a

Liczba takich samych obiektów z dokładnością do symetrii wynosi $\frac{1}{|G|} \sum_{g \in G} |X^g|$, gdzie $G$ to zbiór symetrii (ruchów) oraz $X^g$ to punkty (obiekty) stałe symetrii $g$.

## 2.8 Silnia

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $n!$ | 1 | 2 | 6 | 24 | 120 | 720 | 5040 | 40320 | 362880 | 3628800 |

| $n$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|
| $n!$ | 4.0e7 | 4.8e8 | 6.2e9 | 8.7e10 | 1.3e12 | 2.1e13 | 3.6e14 |

| $n$ | 20 | 25 | 30 | 40 | 50 | 100 | 150 | 171 |
|---|---|---|---|---|---|---|---|---|
| $n!$ | 2e18 | 2e25 | 3e32 | 8e47 | 3e64 | 9e157 | 6e262 | >DBL_MAX |

## 2.9 Symbol Newtona

$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n^{\underline{k}}}{k!}$,
$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n-1}{k-1} + \binom{n-2}{k-1} + \cdots + \binom{k-1}{k-1}$,
$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$, $\sum_{i=0}^{k} \binom{n+i}{i} = \binom{n+k+1}{k}$,
$(-1)^i \binom{x}{i} = \binom{i-1-x}{i}$, $\sum_{i=0}^{k} \binom{n}{i}\binom{m}{k-i} = \binom{n+m}{k}$,
$\binom{n}{k}\binom{k}{i} = \binom{n}{i}\binom{n-i}{k-i}$.

## 2.10 Wzorki na pewne ciągi

### 2.10.1 Nieporządek

Liczba takich permutacji, że $p_i \neq i$ (żadna liczba nie wraca na tą samą pozycję): $D(n) = (n-1)(D(n-1) + D(n-2)) = nD(n-1) + (-1)^n = \left\lfloor \frac{n!}{e} \right\rceil$

### 2.10.2 Liczba podziałów

Liczba sposobów zapisania $n$ jako sumę posortowanych liczb dodatnich: $p(0) = 1,\ p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k-1)/2)$, szacujemy $p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$.

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 50 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $p(n)$ | 1 | 1 | 2 | 3 | 5 | 7 | 11 | 15 | 22 | 30 | 627 | ∼2e5 | ∼2e8 |

### 2.10.3 Liczby Eulera pierwszego rzędu

Liczba permutacji $\pi \in S_n$ gdzie $k$ elementów jest większych niż poprzedni: $k$ razy $\pi(j) > \pi(j+1)$, $k+1$ razy $\pi(j) \geq j$, $k$ razy $\pi(j) > j$. Zachodzi
$E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)$,
$E(n,0) = E(n,n-1) = 1$,
$E(n,k) = \sum_{j=0}^{k} (-1)^j \binom{n+1}{j} (k+1-j)^n$.

### 2.10.4 Stirling pierwszego rzędu

Liczba permutacji długości $n$ mające $k$ cykli:
$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k)$, $c(0,0) = 1$, $\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\dots(x+n-1)$. Małe wartości: $c(8,k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$, $c(n,2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots$.

### 2.10.5 Stirling drugiego rzędu

Liczba podziałów zbioru rozmiaru $n$ na $k$ bloków:
$S(n,k) = S(n-1,k-1) + kS(n-1,k)$,
$S(n,1) = S(n,n) = 1$,
$S(n,k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n$.

### 2.10.6 Liczby Catalana

$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$,
$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2} C_n,\ C_{n+1} = \sum C_i C_{n-i}, C_n =$
$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots$.
Równoważne: ścieżki na planszy $n \times n$, nawiasowania po $n$ (), liczba drzew binarnych z $n+1$ liściami (0 lub 2 syny), skierowanych drzew z $n+1$ wierzchołkami, triangulacje $n+2$-kąta, permutacji $[n]$ bez 3-wyrazowego rosnącego podciągu?

### 2.10.7 Formuła Cayley'a

Liczba różnych drzew (z dokładnością do numerowania wierzchołków) wynosi $n^{n-2}$. Liczba sposobów by zespójnić $k$ spójnych o rozmiarach $s_1, s_2, \dots, s_k$ wynosi $s_1 \cdot s_2 \cdot \dots \cdot s_k \cdot n^{k-2}$.

### 2.10.8 Twierdzenie Kirchhoffa

Liczba różnych drzew rozpinających spójnego nieskierowanego grafu $G$ bez pętelek (mogą być multikrawędzie) o $n$ wierzchołkach jest równa det $A_{n-1}$, gdzie $A = D - M$, $D$ to macierz diagonalna mająca na przekątnej stopnie wierzchołków w grafie $G$, $M$ to macierz incydencji grafu $G$, a $A_{n-1}$ to macierz $A$ z usuniętymi ostatnim wierszem oraz ostatnią kolumną.

## 2.11 Funkcje tworzące

$\frac{1}{(1-x)^k} = \sum_{n \geq 0} \binom{k-1+n}{k-1} x^n$, $\exp(x) = \sum_{n \geq 0} \frac{x^n}{n!}$,
$\log(1-x) = -\sum_{n \geq 1} \frac{x^n}{n}$.

## 2.12 Funkcje multiplikatywne

$\epsilon(n) = [n = 1], id_k(n) = n^k, id = id_1, \mathbb{1} = id_0$,
$\sigma_k(n) = \sum_{d|n} d^k, \sigma = \sigma_1, \tau = \sigma_0, \mu(p^k) = [k = 0] - [k = 1]$,
$\varphi(p^k) = p^k - p^{k-1}, (f * g)(n) = \sum_{d|n} f(d)\, g\left(\frac{n}{d}\right)$,
$f * g = g * f, f * (g * h) = (f * g) * h$,
$f * (g + h) = f * g + f * h$, jak dwie z trzech funkcji $f * g = h$ są multiplikatywne, to trzecia też, $f * \mathbb{1} = g \Leftrightarrow g * \mu = f, f * \epsilon = f$,
$\mu * \mathbb{1} = \epsilon, [n = 1] = \sum_{d|n} \mu(d) = \sum_{d=1}^{n} \mu(d)[d|n], \varphi * \mathbb{1} = id$,
$id_k * \mathbb{1} = \sigma_k, id * \mathbb{1} = \sigma, \mathbb{1} * \mathbb{1} = \tau, s_f(n) = \sum_{i=1}^{n} f(i)$,
$s_f(n) = \frac{s_{f*g}(n) - \sum_{d=2}^{n} s_f\left(\lfloor \frac{n}{d} \rfloor\right) g(d)}{g(1)}$.

## 2.13 Fibonacci

$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$, $F_{n-1}F_{n+1} - F_n^2 = (-1)^n$,
$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$, $F_n | F_{nk}$,
$NWD(F_m, F_n) = F_{NWD(m,n)}$

## 2.14 Woodbury matrix identity

Dla $A \equiv n \times n, C \equiv k \times k, U \equiv n \times k, V \equiv k \times n$ jest $(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$, przy czym często $C = Id$. Używane gdy $A^{-1}$ jest już policzone i chcemy policzyć odwrotność lekko zmienionego $A$ poprzez $C^{-1}$ i $VA^{-1}U$. Często występuje w kombinacji z tożsamością $\frac{1}{1-A} = \sum_{i=0}^{\infty} A^i$.

## 2.15 Zasada włączeń i wyłączeń

$X$ - uniwersum, $A_1, \dots, A_n$ - podzbiory $X$ zwane własnościami $S_j = \sum_{1 \leq i_1 \leq \dots \leq i_j \leq n} |A_{i_1} \cap \dots \cap A_{i_j}|$ W szczególności $S_0 = |X|$. Niech $D(k)$ oznacza liczbę elementów $X$ mających dokładnie $k$ własności. $D(k) = \sum_{j \geq k} \binom{j}{k} (-1)^{j-k} S_j$ W szczególności $D(0) = \sum_{j \geq 0} (-1)^j S_j$

## 2.16 Karp's minimum mean-weight cycle algorithm

$G = (V, E)$ - directed graph with weight function $w : E \to \mathbb{R}$ $n = |V|$ Assume that every vertex is reachable from $s \in V$. $\delta_k(s, v)$ shortest $k$-path from $s$ to $v$ (simple dp) Minimum mean-weight cycle is

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s,v) - \delta_k(s,v)}{n-k}$$

# Matma (3)

## berlekamp-massey includes: simple-modulo $\mathcal{O}(n^2 \log k)$,

BerlekampMassey<mod> bm(x) zgaduje rekurencję ciągu $x$, bm.get(k) zwraca $k$-ty wyraz ciągu $x$ (index 0)

```
struct BerlekampMassey {
  int n;
  vector<int> x, C;
  BerlekampMassey(const vector<int> &_x) : x(_x) {
    auto B = C = {1};
    int b = 1, m = 0;
    REP(i, ssize(x)) {
      m++; int d = x[i];
      FOR(j, 1, ssize(C) - 1)
        d = add(d, mul(C[j], x[i - j]));
      if(d == 0) continue;
      auto _B = C;
      C.resize(max(ssize(C), m + ssize(B)));
      int coef = mul(d, inv(b));
      FOR(j, m, m + ssize(B) - 1)
        C[j] = sub(C[j], mul(coef, B[j - m]));
      if(ssize(_B) < m + ssize(B)) { B = _B; b = d; m
        = 0; }
    }
    C.erase(C.begin());
    for(int &t : C) t = sub(0, t);
    n = ssize(C);
  }
  vector<int> combine(vector<int> a, vector<int> b) {
    vector<int> ret(n * 2 + 1);
    REP(i, n + 1) REP(j, n + 1)
      ret[i + j] = add(ret[i + j], mul(a[i], b[j]));
    for(int i = 2 * n; i > n; i--) REP(j, n)
      ret[i - j - 1] = add(ret[i - j - 1], mul(ret[i],
        C[j]));
    return ret;
  }
  int get(LL k) {
    if (!n) return 0;
    vector<int> r(n + 1), pw(n + 1);
    r[0] = pw[1] = 1;
    for(k++; k; k /= 2) {
      if(k % 2) r = combine(r, pw);
      pw = combine(pw, pw);
    }
    int ret = 0;
```

```
    REP(i, n) ret = add(ret, mul(r[i + 1], x[i]));
    return ret;
  }
};
```

## bignum

Podstawa wynosi $1e9$. Mnożenie, dzielenie, nwd oraz modulo jest kwadratowe, wersje operatorX(Num, int) liniowe. Podstawę można zmieniać (ma zachodzić base == 10^digits_per_elem).

```
struct Num {
  static constexpr int digits_per_elem = 9, base = int
    (1e9);
  int sign = 0;
  vector<int> x;
  Num& shorten() {
    while(ssize(x) and x.back() == 0)
      x.pop_back();
    for(int a : x)
      assert(0 <= a and a < base);
    if(x.empty())
      sign = 0;
    return *this;
  }
  Num(string s) {
    sign = ssize(s) and s[0] == '-' ? s.erase(s.begin
      ()), -1 : 1;
    for(int i = ssize(s); i > 0; i -= digits_per_elem)
      if(i < digits_per_elem)
        x.emplace_back(stoi(s.substr(0, i)));
      else
        x.emplace_back(stoi(s.substr(i -
          digits_per_elem, digits_per_elem)));
    shorten();
  }
  Num() {}
  Num(LL s) : Num(to_string(s)) {}
};
string to_string(const Num& n) {
  stringstream s;
  s << (n.sign == -1 ? "-" : "") << (ssize(n.x) ? n.x.
    back() : 0);
  for(int i = ssize(n.x) - 2; i >= 0; --i)
    s << setfill('0') << setw(n.digits_per_elem) << n.
      x[i];
  return s.str();
}
ostream& operator<<(ostream &o, const Num& n) {
  return o << to_string(n).c_str();
}
auto operator<=>(const Num& a, const Num& b) {
  if(a.sign != b.sign or ssize(a.x) != ssize(b.x))
    return ssize(a.x) * a.sign <=> ssize(b.x) * b.sign
      ;
  for(int i = ssize(a.x) - 1; i >= 0; --i)
    if(a.x[i] != b.x[i])
      return a.x[i] * a.sign <=> b.x[i] * b.sign;
  return strong_ordering::equal;
}
bool operator==(const Num& a, const Num& b) {
  return a.x == b.x and a.sign == b.sign;
}
Num abs(Num n) { n.sign &= 1; return n; }
Num operator+(Num a, Num b) {
  int mode = a.sign * b.sign >= 0 ? a.sign |= b.sign,
    1 : abs(b) > abs(a) ? swap(a, b), -1 : -1, carry =
     0;
  for(int i = 0; i < max(ssize((mode == 1 ? a : b).x),
     ssize(b.x)) or carry; ++i) {
    if(mode == 1 and i == ssize(a.x))
      a.x.emplace_back(0);
    a.x[i] += mode * (carry + (i < ssize(b.x) ? b.x[i]
       : 0));
    carry = a.x[i] >= a.base or a.x[i] < 0;
    a.x[i] -= mode * carry * a.base;
  }
  return a.shorten();
}
Num operator-(Num a) { a.sign *= -1; return a; }
Num operator-(Num a, Num b) { return a + -b; }
Num operator*(Num a, int b) {
  assert(abs(b) < a.base);
  int carry = 0;
  for(int i = 0; i < ssize(a.x) or carry; ++i) {
    if(i == ssize(a.x))
      a.x.emplace_back(0);
    LL cur = a.x[i] * LL(abs(b)) + carry;
    a.x[i] = int(cur % a.base);
    carry = int(cur / a.base);
  }
  if(b < 0)
    a.sign *= -1;
  return a.shorten();
}
Num operator*(const Num& a, const Num& b) {
  Num c;
  c.x.resize(ssize(a.x) + ssize(b.x));
  REP(i, ssize(a.x))
    for(int j = 0, carry = 0; j < ssize(b.x) or carry;
       ++j) {
      LL cur = c.x[i + j] + a.x[i] * LL(j < ssize(b.x)
         ? b.x[j] : 0) + carry;
      c.x[i + j] = int(cur % a.base);
      carry = int(cur / a.base);
    }
  c.sign = a.sign * b.sign;
  return c.shorten();
}
Num operator/(Num a, int b) {
  assert(b != 0 and abs(b) < a.base);
  int carry = 0;
  for(int i = ssize(a.x) - 1; i >= 0; --i) {
    LL cur = a.x[i] + carry * LL(a.base);
    a.x[i] = int(cur / abs(b));
    carry = int(cur % abs(b));
  }
  if(b < 0)
    a.sign *= -1;
  return a.shorten();
}
// zwraca a * pow(a.base, b)
Num shift(Num a, int b) {
  vector v(b, 0);
  a.x.insert(a.x.begin(), v.begin(), v.end());
  return a.shorten();
}
Num operator/(Num a, Num b) {
  assert(ssize(b.x));
  int s = a.sign * b.sign;
  Num c;
  a = abs(a);
  b = abs(b);
  for(int i = ssize(a.x) - ssize(b.x); i >= 0; --i) {
    if (a < shift(b, i)) continue;
    int l = 0, r = a.base - 1;
    while (l < r) {
      int m = (l + r + 1) / 2;
      if (shift(b * m, i) <= a)
        l = m;
      else
        r = m - 1;
    }
    c = c + shift(l, i);
    a = a - shift(b * l, i);
  }
  c.sign = s;
  return c.shorten();
}
template<typename T>
Num operator%(const Num& a, const T& b) { return a -
  ((a / b) * b); }
Num nwd(const Num& a, const Num& b) { return b == Num
  () ? a : nwd(b, a % b); }
```

## binsearch-stern-brocot

$\mathcal{O}(\log max\_val)$, szuka największego a/b, że is_ok(a/b) oraz 0 <= a,b <= max_value. Zakłada, że is_ok(0) == true.

```
using Frac = pair<LL, LL>;
Frac my_max(Frac l, Frac r) {
  return l.first * __int128_t(r.second) > r.first *
    __int128_t(l.second) ? l : r;
}
Frac binsearch(LL max_value, function<bool (Frac)>
  is_ok) {
  assert(is_ok(pair(0, 1)) == true);
  Frac left = {0, 1}, right = {1, 0}, best_found =
    left;
  int current_dir = 0;
  while(max(left.first, left.second) <= max_value) {
    best_found = my_max(best_found, left);
    auto get_frac = [&](LL mul) {
      LL mull = current_dir ? 1 : mul;
      LL mulr = current_dir ? mul : 1;
      return pair(left.first * mull + right.first *
        mulr, left.second * mull + right.second * mulr
        );
    };
    auto is_good_mul = [&](LL mul) {
      Frac mid = get_frac(mul);
      return is_ok(mid) == current_dir and max(mid.
        first, mid.second) <= max_value;
    };
    LL power = 1;
    for(; is_good_mul(power); power *= 2) {}
    LL bl = power / 2 + 1, br = power;
    while(bl != br) {
      LL bm = (bl + br) / 2;
      if(not is_good_mul(bm))
        br = bm;
      else
        bl = bm + 1;
    }
    tie(left, right) = pair(get_frac(bl - 1), get_frac
      (bl));
    if(current_dir == 0)
      swap(left, right);
    current_dir ^= 1;
  }
  return best_found;
}
```

## crt

includes: extended-gcd $\mathcal{O}(\log n)$, crt(a, m, b, n) zwraca takie $x$, że $x$ mod $m = a$ oraz $x$ mod $n = b$, $m$ oraz $n$ nie muszą być wzlędnie pierwsze, ale może nie być wtedy rozwiązania (assert wywali, ale można zmienić na return -1).

```
LL crt(LL a, LL m, LL b, LL n) {
  if(n > m) swap(a, b), swap(m, n);
  auto [d, x, y] = extended_gcd(m, n);
  assert((a - b) % d == 0);
  LL ret = (b - a) % n * x % n / d * m + a;
  return ret < 0 ? ret + m * n / d : ret;
}
```

## determinant

includes: matrix-header $\mathcal{O}(n^3)$, wyznacznik macierzy (modulo lub double)

```
T determinant(vector<vector<T>>& a) {
  int n = ssize(a);
  T res = 1;
  REP(i, n) {
    int b = i;
    FOR(j, i + 1, n - 1)
      if(abs(a[j][i]) > abs(a[b][i]))
        b = j;
    if(i != b)
      swap(a[i], a[b]), res = sub(0, res);
    res = mul(res, a[i][i]);
    if (equal(res, 0))
      return 0;
    FOR(j, i + 1, n - 1) {
      T v = divide(a[j][i], a[i][i]);
      if (not equal(v, 0))
        FOR(k, i + 1, n - 1)
          a[j][k] = sub(a[j][k], mul(v, a[i][k]));
    }
  }
  return res;
}
```

## discrete-log

includes: simple-modulo $\mathcal{O}\left(\sqrt{m}\log n\right)$ czasowo, $\mathcal{O}\left(\sqrt{n}\right)$ pamięciowo, dla liczby pierwszej $mod$ oraz $a, b \nmid mod$ znajdzie $e$ takie że $a^e \equiv b \pmod{mod}$. Jak zwróci $-1$ to nie istnieje.

```
int discrete_log(int a, int b) {
  int n = int(sqrt(mod)) + 1;
  int an = 1;
  REP(i, n)
    an = mul(an, a);
  unordered_map<int, int> vals;
  int cur = b;
  FOR(q, 0, n) {
    vals[cur] = q;
    cur = mul(cur, a);
  }
  cur = 1;
  FOR(p, 1, n) {
    cur = mul(cur, an);
    if(vals.count(cur)) {
      int ans = n * p - vals[cur];
      return ans;
    }
  }
  return -1;
}
```

## discrete-root

includes: primitive-root, discrete-log Dla pierwszego $mod$ oraz $a \perp mod, k$ znajduje $b$ takie, że $b^k = a$ (pierwiastek k-tego stopnia z $a$). Jak zwróci -1 to nie istnieje.

```
int discrete_root(int a, int k) {
  int g = primitive_root();
  int y = discrete_log(powi(g, k), a);
  if(y == -1)
    return -1;
  return powi(g, y);
}
```

## extended-gcd

$\mathcal{O}(\log(\min(a, b)))$, dla danego $(a, b)$ znajduje takie $(gcd(a, b), x, y)$, że $ax + by = gcd(a, b)$. auto [gcd, x, y] = extended_gcd(a, b);

```
tuple<LL, LL, LL> extended_gcd(LL a, LL b) {
  if(a == 0)
    return {b, 0, 1};
  auto [gcd, x, y] = extended_gcd(b % a, a);
  return {gcd, y - x * (b / a), x};
}
```

## fft-mod

includes: fft $\mathcal{O}\left(n \log n\right)$, conv_mod(a, b) zwraca iloczyn wielomianów modulo, ma większą dokładność niż zwykłe fft.

```
vector<int> conv_mod(vector<int> a, vector<int> b, int
   M) {
  if(a.empty() or b.empty()) return {};
  vector<int> res(ssize(a) + ssize(b) - 1);
  const int CUTOFF = 125;
  if (min(ssize(a), ssize(b)) <= CUTOFF) {
    if (ssize(a) > ssize(b))
      swap(a, b);
    REP (i, ssize(a))
      REP (j, ssize(b))
        res[i + j] = int((res[i + j] + LL(a[i]) * b[j
          ]) % M);
    return res;
  }
  int B = 32 - __builtin_clz(ssize(res)), n = 1 << B;
```

```
  int cut = int(sqrt(M));
  vector<Complex> L(n), R(n), outl(n), outs(n);
  REP(i, ssize(a)) L[i] = Complex((int) a[i] / cut, (
    int) a[i] % cut);
  REP(i, ssize(b)) R[i] = Complex((int) b[i] / cut, (
    int) b[i] % cut);
  fft(L), fft(R);
  REP(i, n) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) /
       1i;
  }
  fft(outl), fft(outs);
  REP(i, ssize(res)) {
    LL av = LL(real(outl[i]) + 0.5), cv = LL(imag(outs
      [i]) + 0.5);
    LL bv = LL(imag(outl[i]) + 0.5) + LL(real(outs[i])
       + 0.5);
    res[i] = int(((av % M * cut + bv) % M * cut + cv)
      % M);
  }
  return res;
}
```

## fft $\mathcal{O}(n \log n)$, conv(a, b) to iloczyn wielomianów.

```
using Complex = complex<double>;
void fft(vector<Complex> &a) {
  int n = ssize(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<Complex> rt(2, 1);
  for(static int k = 2; k < n; k *= 2) {
    R.resize(n), rt.resize(n);
    auto x = polar(1.0L, acosl(-1) / k);
    FOR(i, k, 2 * k - 1)
      rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
  }
  vector<int> rev(n);
  REP(i, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  REP(i, n) if(i < rev[i]) swap(a[i], a[rev[i]]);
  for(int k = 1; k < n; k *= 2) {
    for(int i = 0; i < n; i += 2 * k) REP(j, k) {
      Complex z = rt[j + k] * a[i + j + k]; // mozna
         zoptowac rozpisujac
      a[i + j + k] = a[i + j] - z;
      a[i + j] += z;
    }
  }
}
vector<double> conv(vector<double> &a, vector<double>
  &b) {
  if(a.empty() || b.empty()) return {};
  vector<double> res(ssize(a) + ssize(b) - 1);
  int L = 32 - __builtin_clz(ssize(res)), n = (1 << L)
    ;
  vector<Complex> in(n), out(n);
  copy(a.begin(), a.end(), in.begin());
  REP(i, ssize(b)) in[i].imag(b[i]);
  fft(in);
  for(auto &x : in) x *= x;
  REP(i, n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  REP(i, ssize(res)) res[i] = imag(out[i]) / (4 * n);
  return res;
}
```

## floor-sum $\mathcal{O}(\log a)$, liczy $\sum_{i=0}^{n-1} \left\lfloor \frac{a \cdot i + b}{c} \right\rfloor$. Działa dla $0 \le a, b < c$ oraz $1 \le c, n \le 10^9$. Dla innych $n, a, b, c$ trzeba uważać lub użyć __int128.

```
LL floor_sum(LL n, LL a, LL b, LL c) {
  LL ans = 0;
  if (a >= c) {
    ans += (n - 1) * n * (a / c) / 2;
    a %= c;
  }
  if (b >= c) {
    ans += n * (b / c);
    b %= c;
  }
  LL d = (a * (n - 1) + b) / c;
  if (d == 0) return ans;
  ans += d * (n - 1) - floor_sum(d, c, c - b - 1, a);
  return ans;
}
```

## fwht $\mathcal{O}(n \log n)$, $n$ musi być potęgą dwójki, fwht_or(a)[i] = suma(j będące podmaską i) a[j], ifwht_or(fwht_or(a)) == a, convolution_or(a, b)[i] = suma(j | k == i) a[j] * b[k], fwht_and(a)[i] = suma(j będące nadmaską i) a[j], ifwht_and(fwht_and(a)) == a, convolution_and(a, b)[i] = suma(j & k == i) a[j] * b[k], fwht_xor(a)[i] = suma(j oraz i mają parzyście wspólnie zapalonych bitów) a[j] - suma(j oraz i mają nieparzyście) a[j], ifwht_xor(fwht_xor(a)) == a, convolution_xor(a, b)[i] = suma(j k□ == i) a[j] * b[k].

```
vector<int> fwht_or(vector<int> a) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0);
  for(int s = 1; 2 * s <= n; s *= 2)
    for(int l = 0; l < n; l += 2 * s)
      for(int i = l; i < l + s; ++i)
        a[i + s] += a[i];
  return a;
}
vector<int> ifwht_or(vector<int> a) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0);
  for(int s = n / 2; s >= 1; s /= 2)
    for(int l = 0; l < n; l += 2 * s)
      for(int i = l; i < l + s; ++i)
        a[i + s] -= a[i];
  return a;
}
vector<int> convolution_or(vector<int> a, vector<int>
  b) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0 and ssize(b) == n);
  a = fwht_or(a);
  b = fwht_or(b);
  REP(i, n)
    a[i] *= b[i];
  return ifwht_or(a);
}
vector<int> fwht_and(vector<int> a) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0);
  for(int s = 1; 2 * s <= n; s *= 2)
    for(int l = 0; l < n; l += 2 * s)
      for(int i = l; i < l + s; ++i)
        a[i] += a[i + s];
  return a;
}
vector<int> ifwht_and(vector<int> a) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0);
  for(int s = n / 2; s >= 1; s /= 2)
    for(int l = 0; l < n; l += 2 * s)
      for(int i = l; i < l + s; ++i)
        a[i] -= a[i + s];
  return a;
}
vector<int> convolution_and(vector<int> a, vector<int>
   b) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0 and ssize(b) == n);
  a = fwht_and(a);
  b = fwht_and(b);
  REP(i, n)
    a[i] *= b[i];
  return ifwht_and(a);
}
vector<int> fwht_xor(vector<int> a) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0);
  for(int s = 1; 2 * s <= n; s *= 2)
    for(int l = 0; l < n; l += 2 * s)
      for(int i = l; i < l + s; ++i) {
        int t = a[i + s];
        a[i + s] = a[i] - t;
        a[i] += t;
      }
  return a;
}
vector<int> ifwht_xor(vector<int> a) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0);
  for(int s = n / 2; s >= 1; s /= 2)
    for(int l = 0; l < n; l += 2 * s)
      for(int i = l; i < l + s; ++i) {
        int t = a[i + s];
        a[i + s] = (a[i] - t) / 2;
        a[i] = (a[i] + t) / 2;
      }
  return a;
}
vector<int> convolution_xor(vector<int> a, vector<int>
   b) {
  int n = ssize(a);
  assert((n & (n - 1)) == 0 and ssize(b) == n);
  a = fwht_xor(a);
  b = fwht_xor(b);
  REP(i, n)
    a[i] *= b[i];
  return ifwht_xor(a);
}
```

## gauss includes: matrix-header $\mathcal{O}(nm(n+m))$, Wrzucam $n$ vectorów {wsp_x0, wsp_x1, ..., wsp_xm - 1, suma}, gauss wtedy zwraca liczbę rozwiązań (0, 1 albo 2 (tzn. nieskończoność)) oraz jedno poprawne rozwiązanie (o ile istnieje). Przykład gauss({2, -1, 1, 7}, {1, 1, 1, 1}, {0, 1, -1, 6.5}) zwraca (1, {6.75, 0.375, -6.125}).

```
pair<int, vector<T>> gauss(vector<vector<T>> a) {
  int n = ssize(a); // liczba wierszy
  int m = ssize(a[0]) - 1; // liczba zmiennych
  vector<int> where(m, -1); // w ktorym wierszu jest
    zdefiniowana i-ta zmienna
  for(int col = 0, row = 0; col < m and row < n; ++col
    ) {
    int sel = row;
    for(int y = row; y < n; ++y)
      if(abs(a[y][col]) > abs(a[sel][col]))
        sel = y;
    if(equal(a[sel][col], 0))
      continue;
    for(int x = col; x <= m; ++x)
      swap(a[sel][x], a[row][x]);
    // teraz sel jest nieaktualne
    where[col] = row;
    for(int y = 0; y < n; ++y)
      if(y != row) {
        T wspolczynnik = divide(a[y][col], a[row][col
          ]);
        for(int x = col; x <= m; ++x)
          a[y][x] = sub(a[y][x], mul(wspolczynnik, a[
            row][x]));
      }
    ++row;
  }
  vector<T> answer(m);
  for(int col = 0; col < m; ++col)
    if(where[col] != -1)
      answer[col] = divide(a[where[col]][m], a[where[
        col]][col]);
  for(int row = 0; row < n; ++row) {
    T got = 0;
    for(int col = 0; col < m; ++col)
      got = add(got, mul(answer[col], a[row][col]));
    if(not equal(got, a[row][m]))
      return {0, answer};
  }
  for(int col = 0; col < m; ++col)
    if(where[col] == -1)
      return {2, answer};
  return {1, answer};
}
```

## integral $\mathcal{O}(idk)$, zwraca całkę f na [l, r].

```
using D = long double;
D simpson(function<D (D)> f, D l, D r) {
  return (f(l) + 4 * f((l + r) / 2) + f(r)) * (r - l)
    / 6;
}
D integrate(function<D (D)> f, D l, D r, D s, D eps) {
  D m = (l + r) / 2;
  D sl = simpson(f, l, m), sr = simpson(f, m, r), s2 =
     sl + sr;
  if(abs(s2 - s) < 15 * eps or r - l < 1e-10)
    return s2 + (s2 - s) / 15;
  return integrate(f, l, m, sl, eps / 2)
    + integrate(f, m, r, sr, eps / 2);
}
D integrate(function<D (D)> f, D l, D r) {
  return integrate(f, l, r, simpson(f, l, r), 1e-8);
}
```

## lagrange-consecutive includes: simple-modulo $\mathcal{O}(n)$, przyjmuje wartości wielomianu w punktach $0, 1, \ldots, n-1$ i wylicza jego wartość w $x$. lagrange_consecutive({2, 3, 4}, 3) == 5

```
int lagrange_consecutive(vector<int> y, int x) {
  int n = ssize(y), fac = 1, pref = 1, suff = 1, ret =
     0;
  FOR(i, 1, n) fac = mul(fac, i);
  fac = inv(fac);
  REP(i, n) {
    fac = mul(fac, n - i);
    y[i] = mul(y[i], mul(pref, fac));
    y[n - 1 - i] = mul(y[n - 1 - i], mul(suff, mul(i %
       2 ? mod - 1 : 1, fac)));
    pref = mul(pref, sub(x, i));
    suff = mul(suff, sub(x, n - 1 - i));
  }
  REP(i, n) ret = add(ret, y[i]);
  return ret;
}
```

## matrix-header Funkcje pomocnicze do algorytmów macierzowych.

```
#if 1
#ifdef CHANGABLE_MOD
int mod = 998'244'353;
#else
constexpr int mod = 998'244'353;
#endif
bool equal(int a, int b) {
  return a == b;
}
int mul(int a, int b) {
  return int(a * LL(b) % mod);
}
int add(int a, int b) {
  a += b;
  return a >= mod ? a - mod : a;
}
int powi(int a, int b) {
  for(int ret = 1;; b /= 2) {
    if(b == 0)
      return ret;
    if(b & 1)
      ret = mul(ret, a);
    a = mul(a, a);
  }
```

```
}
int inv(int x) {
  return powi(x, mod - 2);
}
int divide(int a, int b) {
  return mul(a, inv(b));
}
int sub(int a, int b) {
  return add(a, mod - b);
}
using T = int;
#else
constexpr double eps = 1e-9;
bool equal(double a, double b) {
  return abs(a - b) < eps;
}
#define OP(name, op) double name(double a, double b) {
    return a op b; }
OP(mul, *)
OP(add, +)
OP(divide, /)
OP(sub, -)
using T = double;
#endif
```

## matrix-inverse

includes: `matrix-header` $\mathcal{O}\left(n^3\right)$, odwrotność macierzy (modulo lub double). Zwraca rząd macierzy. Dla odwracalnych macierzy (rząd = n) w $a$ znajdzie się jej odwrotność.

```
int inverse(vector<vector<T>>& a) {
  int n = ssize(a);
  vector<int> col(n);
  vector h(n, vector<T>(n));
  REP(i, n)
    h[i][i] = 1, col[i] = i;
  REP(i, n) {
    int r = i, c = i;
    FOR(j, i, n - 1) FOR(k, i, n - 1)
      if(abs(a[j][k]) > abs(a[r][c]))
        r = j, c = k;
    if (equal(a[r][c], 0))
      return i;
    a[i].swap(a[r]);
    h[i].swap(h[r]);
    REP(j, n)
      swap(a[j][i], a[j][c]), swap(h[j][i], h[j][c]);
    swap(col[i], col[c]);
    T v = a[i][i];
    FOR(j, i + 1, n - 1) {
      T f = divide(a[j][i], v);
      a[j][i] = 0;
      FOR(k, i + 1, n - 1)
        a[j][k] = sub(a[j][k], mul(f, a[i][k]));
      REP(k, n)
        h[j][k] = sub(h[j][k], mul(f, h[i][k]));
    }
    FOR(j, i + 1, n - 1)
      a[i][j] = divide(a[i][j], v);
    REP(j, n)
      h[i][j] = divide(h[i][j], v);
    a[i][i] = 1;
  }
  for(int i = n - 1; i > 0; --i) REP(j, i) {
    T v = a[j][i];
    REP(k, n)
      h[j][k] = sub(h[j][k], mul(v, h[i][k]));
  }
  REP(i, n)
    REP(j, n)
      a[col[i]][col[j]] = h[i][j];
  return n;
}
```

## miller-rabin

$\mathcal{O}\left(\log^2 n\right)$ test pierwszości Millera-Rabina, działa dla long longów.

```
LL llmul(LL a, LL b, LL m) {
  return LL(__int128_t(a) * b % m);
}
LL llpowi(LL a, LL n, LL m) {
  for (LL ret = 1;; n /= 2) {
    if (n == 0)
      return ret;
    if (n % 2)
      ret = llmul(ret, a, m);
    a = llmul(a, a, m);
  }
}
bool miller_rabin(LL n) {
  if(n < 2) return false;
  int r = 0;
  LL d = n - 1;
  while(d % 2 == 0)
    d /= 2, r++;
  for(int a : {2, 325, 9375, 28178, 450775, 9780504,
    1795265022}) {
    if (a % n == 0) continue;
    LL x = llpowi(a, d, n);
    if(x == 1 || x == n - 1)
      continue;
    bool composite = true;
    REP(i, r - 1) {
      x = llmul(x, x, n);
      if(x == n - 1) {
        composite = false;
        break;
      }
    }
    if(composite) return false;
  }
  return true;
}
```

## multiplicative

includes: `sieve` $\mathcal{O}\left(n\right)$, `mobius(n)` oblicza funkcję Möbiusa na [0..n], `totient(n)` oblicza funkcję Eulera na [0..n], wartości w $0$ niezdefiniowane.

```
vector<int> mobius(int n) {
  sieve(n);
  vector<int> ans(n + 1, 0);
  if (n) ans[1] = 1;
  FOR(i, 2, n) {
    int p = prime_div[i];
    if (i / p % p) ans[i] = -ans[i / p];
  }
  return ans;
}
vector<int> totient(int n) {
  sieve(n);
  vector<int> ans(n + 1, 1);
  FOR(i, 2, n) {
    int p = prime_div[i];
    ans[i] = ans[i / p] * (p - bool(i / p % p));
  }
  return ans;
}
```

## ntt

includes: `simple-modulo` $\mathcal{O}\left(n \log n\right)$ mnożenie wielomianów mod $998244353$.

```
using vi = vector<int>;
constexpr int root = 3;
void ntt(vi& a, int n, bool inverse = false) {
  assert((n & (n - 1)) == 0);
  a.resize(n);
  vi b(n);
  for(int w = n / 2; w; w /= 2, swap(a, b)) {
    int r = powi(root, (mod - 1) / n * w), m = 1;
    for(int i = 0; i < n; i += w * 2, m = mul(m, r))
      REP(j, w) {
      int u = a[i + j], v = mul(a[i + j + w], m);
      b[i / 2 + j] = add(u, v);
      b[i / 2 + j + n / 2] = sub(u, v);
    }
  }
  if(inverse) {
    reverse(a.begin() + 1, a.end());
    int invn = inv(n);
    for(int& e : a) e = mul(e, invn);
  }
}
vi conv(vi a, vi b) {
  if(a.empty() or b.empty()) return {};
  int l = ssize(a) + ssize(b) - 1, sz = 1 << __lg(2 *
    l - 1);
  ntt(a, sz), ntt(b, sz);
  REP(i, sz) a[i] = mul(a[i], b[i]);
  ntt(a, sz, true), a.resize(l);
  return a;
}
```

## pell

$\mathcal{O}\left(\log n\right)$, `pell(n)` oblicza rozwiązanie fundamentalne $x^2 - ny^2 = 1$, zwraca $(0, 0)$ jeżeli nie istnieje ($n$ jest kwadratem lub wynik przekracza LL), `all_pell(n, limit)` wyznacza wszystkie rozwiązania $x^2 - ny^2 = 1$ z $x \leq$ `limit`, w razie potrzeby można przepisać na pythona lub użyć bignumów.

```
pair<LL, LL> pell(LL n) {
  LL s = LL(sqrtl(n));
  if (s * s == n) return {0, 0};
  LL m = 0, d = 1, a = s;
  __int128 num1 = 1, num2 = a, den1 = 0, den2 = 1;
  while (num2 * num2 - n * den2 * den2 != 1) {
    m = d * a - m;
    d = (n - m * m) / d;
    a = (s + m) / d;
    if (num2 > (1ll << 62) / a) return {0, 0};
    tie(num1, num2) = pair(num2, a * num2 + num1);
    tie(den1, den2) = pair(den2, a * den2 + den1);
  }
  return {num2, den2};
}
vector<pair<LL, LL>> all_pell(LL n, LL limit) {
  auto [x0, y0] = pell(n);
  if (!x0) return {};
  vector<pair<LL, LL>> ret;
  __int128 x = x0, y = y0;
  while (x <= limit) {
    ret.emplace_back(x, y);
    if (y0 * y > (1ll << 62) / n) break;
    tie(x, y) = pair(x0 * x + n * y0 * y, x0 * y + y0
      * x);
  }
  return ret;
}
```

## pi

$\mathcal{O}\left(n^{\wedge}\frac{3}{4}\right)$, liczba liczb pierwszych na przedziale $[1, n]$. `Pi pi(n); pi.query(d); // musi zachodzic d | n`

```
struct Pi {
  vector<LL> w, dp;
  int id(LL v) {
    if (v <= w.back() / v)
      return int(v - 1);
    return ssize(w) - int(w.back() / v);
  }
  Pi(LL n) {
    for (LL i = 1; i * i <= n; ++i) {
      w.push_back(i);
      if (n / i != i)
        w.emplace_back(n / i);
    }
    sort(w.begin(), w.end());
    for (LL i : w)
      dp.emplace_back(i - 1);
    for (LL i = 1; (i + 1) * (i + 1) <= n; ++i) {
      if (dp[i] == dp[i - 1])
        continue;
      for (int j = ssize(w) - 1; w[j] >= (i + 1) * (i
        + 1); --j)
        dp[j] -= dp[id(w[j] / (i + 1))] - dp[i - 1];
    }
  }
  LL query(LL v) {
    assert(w.back() % v == 0);
    return dp[id(v)];
  }
};
```

## polynomial

includes: `ntt` Operacje na wielomianach mod $998244353$, deriv, integr $\mathcal{O}\left(n\right)$, powi_deg $\mathcal{O}\left(n \cdot deg\right)$, sqrt, inv, log, exp, powi, div $\mathcal{O}\left(n \log n\right)$, powi_slow, eval, inter $\mathcal{O}\left(n \log^2 n\right)$ Ogólnie to przepisujemy co chcemy. Funkcje oznaczone jako KONIECZNE są wymagane. Funkcje oznaczone WYMAGA ABC wymagają wcześniejszego przepisania ABC. `deriv(a)` zwraca $a'$, `integr(a)` zwraca $\int a$, `powi(_deg_slow)(a, k, n)` zwraca $a^k (\bmod x^n)$, `sqrt(a, n)` zwraca $a^{\wedge}\frac{1}{2} (\bmod x^n)$, `inv(a, n)` zwraca $a^{-1} (\bmod x^n)$, `log(a, n)` zwraca $ln(a) (\bmod x^n)$, `exp(a, n)` zwraca $exp(a) (\bmod x^n)$, `div(a, b)` zwraca $(q, r)$ takie, że $a = qb + r$, `eval(a, x)` zwraca $y$ taki, że $a(x_i) = y_i$, `inter(x, y)` zwraca $a$ taki, że $a(x_i) = y_i$.

```
vi mod_xn(const vi& a, int n) { // KONIECZNE
  return vi(a.begin(), a.begin() + min(n, ssize(a)));
}
void sub(vi& a, const vi& b) { // KONIECZNE
  a.resize(max(ssize(a), ssize(b)));
  REP(i, ssize(b)) a[i] = sub(a[i], b[i]);
}
vi deriv(vi a) {
  REP(i, ssize(a)) a[i] = mul(a[i], i);
  if(ssize(a)) a.erase(a.begin());
  return a;
}
vi integr(vi a) {
  int n = ssize(a);
  a.insert(a.begin(), 0);
  static vi f{1};
  FOR(i, ssize(f), n) f.emplace_back(mul(f[i - 1], i))
    ;
  int r = inv(f[n]);
  for(int i = n; i > 0; --i)
    a[i] = mul(a[i], mul(r, f[i - 1])), r = mul(r, i);
  return a;
}
vi powi_deg(const vi& a, int k, int n) {
  assert(ssize(a) and a[0] != 0);
  vi v(n), f(n, 1);
  v[0] = powi(a[0], k);
  REP(i, n - 1) f[i + 1] = mul(f[i], n - i);
  int r = inv(mul(f[n - 1], a[0]));
  FOR(i, 1, n - 1) {
    FOR(j, 1, min(ssize(a) - 1, i)) {
      v[i] = add(v[i], mul(a[j], mul(v[i - j], sub(mul
        (k, j), i - j))));
    }
    v[i] = mul(v[i], mul(r, f[n - i]));
    r = mul(r, i);
  }
  return v;
}
vi powi_slow(const vi &a, int k, int n) {
  vi v{1}, b = mod_xn(a, n);
  int x = 1; while(x < n) x *= 2;
  while(k) {
    ntt(b, 2 * x);
    if(k & 1) {
      ntt(v, 2 * x);
      REP(i, 2 * x) v[i] = mul(v[i], b[i]);
      ntt(v, 2 * x, true);
      v.resize(x);
    }
    REP(i, 2 * x) b[i] = mul(b[i], b[i]);
    ntt(b, 2 * x, true);
    b.resize(x);
    k /= 2;
  }
  return mod_xn(v, n);
```

```
}
vi sqrt(const vi& a, int n) {
  auto at = [&](int i) { if(i < ssize(a)) return a[i];
      else return 0; };
  assert(ssize(a) and a[0] == 1);
  const int inv2 = inv(2);
  vi v{1}, f{1}, g{1};
  for(int x = 1; x < n; x *= 2) {
    vi z = v;
    ntt(z, x);
    vi b = g;
    REP(i, x) b[i] = mul(b[i], z[i]);
    ntt(b, x, true);
    REP(i, x / 2) b[i] = 0;
    ntt(b, x);
    REP(i, x) b[i] = mul(b[i], g[i]);
    ntt(b, x, true);
    REP(i, x / 2) f.emplace_back(sub(0, b[i + x / 2]))
      ;
    REP(i, x) z[i] = mul(z[i], z[i]);
    ntt(z, x, true);
    vi c(2 * x);
    REP(i, x) c[i + x] = sub(add(at(i), at(i + x)), z[
      i]);
    ntt(c, 2 * x);
    g = f;
    ntt(g, 2 * x);
    REP(i, 2 * x) c[i] = mul(c[i], g[i]);
    ntt(c, 2 * x, true);
    REP(i, x) v.emplace_back(mul(c[i + x], inv2));
  }
  return mod_xn(v, n);
}
vi inv(const vi& a, int n) {
  assert(ssize(a) and a[0] != 0);
  vi v{inv(a[0])};
  for(int x = 1; x < n; x *= 2) {
    vi f = mod_xn(a, 2 * x), g = v;
    ntt(g, 2 * x);
    REP(k, 2) {
      ntt(f, 2 * x);
      REP(i, 2 * x) f[i] = mul(f[i], g[i]);
      ntt(f, 2 * x, true);
      REP(i, x) f[i] = 0;
    }
    sub(v, f);
  }
  return mod_xn(v, n);
}
vi log(const vi& a, int n) { // WYMAGA deriv, integr,
  inv
  assert(ssize(a) and a[0] == 1);
  return integr(mod_xn(conv(deriv(mod_xn(a, n)), inv(a
    , n)), n - 1));
}
vi exp(const vi& a, int n) { // WYMAGA deriv, integr
  assert(a.empty() or a[0] == 0);
  vi v{1}, f{1}, g, h{0}, s;
  for(int x = 1; x < n; x *= 2) {
    g = v;
    REP(k, 2) {
      ntt(g, (2 - k) * x);
      if(!k) s = g;
      REP(i, x) g[i] = mul(g[(2 - k) * i], h[i]);
      ntt(g, x, true);
      REP(i, x / 2) g[i] = 0;
    }
    sub(f, g);
    vi b = deriv(mod_xn(a, x));
    ntt(b, x);
    REP(i, x) b[i] = mul(s[2 * i], b[i]);
    ntt(b, x, true);
    vi c = deriv(v);
    sub(c, b);
    rotate(c.begin(), c.end() - 1, c.end());
    ntt(c, 2 * x);
    h = f;
    ntt(h, 2 * x);
    REP(i, 2 * x) c[i] = mul(c[i], h[i]);
    ntt(c, 2 * x, true);
    c.resize(x);
    vi t(x - 1);
    c.insert(c.begin(), t.begin(), t.end());
    vi d = mod_xn(a, 2 * x);
    sub(d, integr(c));
    d.erase(d.begin(), d.begin() + x);
    ntt(d, 2 * x);
    REP(i, 2 * x) d[i] = mul(d[i], s[i]);
    ntt(d, 2 * x, true);
    REP(i, x) v.emplace_back(d[i]);
  }
  return mod_xn(v, n);
}
vi powi(const vi& a, int k, int n) { // WYMAGA log,
  exp
  vi v = mod_xn(a, n);
  int cnt = 0;
  while(cnt < ssize(v) and !v[cnt])
    ++cnt;
  if(LL(cnt) * k >= n)
    return {};
  v.erase(v.begin(), v.begin() + cnt);
  if(v.empty())
    return k ? vi{} : vi{1};
  int powi0 = powi(v[0], k);
  int inv0 = inv(v[0]);
  for(int& e : v) e = mul(e, inv0);
  v = log(v, n - cnt * k);
  for(int& e : v) e = mul(e, k);
  v = exp(v, n - cnt * k);
  for(int& e : v) e = mul(e, powi0);
  vi t(cnt * k, 0);
  v.insert(v.begin(), t.begin(), t.end());
  return v;
}
pair<vi, vi> div_slow(vi a, const vi& b) {
  vi x;
  while(ssize(a) >= ssize(b)) {
    x.emplace_back(mul(a.back(), inv(b.back())));
    if(x.back() != 0)
      REP(i, ssize(b))
        a.end()[-i - 1] = sub(a.end()[-i - 1], mul(x.
          back(), b.end()[-i - 1]));
    a.pop_back();
  }
  reverse(x.begin(), x.end());
  return {x, a};
}
pair<vi, vi> div(vi a, const vi& b) { // WYMAGA inv,
  div_slow
  const int d = ssize(a) - ssize(b) + 1;
  if (d <= 0)
    return {{}, a};
  if (min(d, ssize(b)) < 250)
    return div_slow(a, b);
  vi x = mod_xn(conv(mod_xn({a.rbegin(), a.rend()}, d)
    , inv({b.rbegin(), b.rend()}, d)), d);
  reverse(x.begin(), x.end());
  sub(a, conv(x, b));
  return {x, mod_xn(a, ssize(b))};
}
vi build(vector<vi> &tree, int v, auto l, auto r) {
  if (r - l == 1) {
    return tree[v] = vi{sub(0, *l), 1};
  } else {
    auto M = l + (r - l) / 2;
    return tree[v] = conv(build(tree, 2 * v, l, M),
      build(tree, 2 * v + 1, M, r));
  }
}
int eval_single(const vi& a, int x) {
  int y = 0;
  for (int i = ssize(a) - 1; i >= 0; --i) {
    y = mul(y, x);
    y = add(y, a[i]);
  }
  return y;
}
vi eval_helper(const vi& a, vector<vi>& tree, int v,
  auto l, auto r) {
  if (r - l == 1) {
    return {eval_single(a, *l)};
  } else {
    auto m = l + (r - l) / 2;
    vi A = eval_helper(div(a, tree[2 * v]).second,
      tree, 2 * v, l, m);
    vi B = eval_helper(div(a, tree[2 * v + 1]).second,
      tree, 2 * v + 1, m, r);
    A.insert(A.end(), B.begin(), B.end());
    return A;
  }
}
vi eval(const vi& a, const vi& x) { // WYMAGA div,
  eval_single, build, eval_helper
  if (x.empty())
    return {};
  vector<vi> tree(4 * ssize(x));
  build(tree, 1, begin(x), end(x));
  return eval_helper(a, tree, 1, begin(x), end(x));
}
vi inter_helper(const vi& a, vector<vi>& tree, int v,
  auto l, auto r, auto ly, auto ry) {
  if (r - l == 1) {
    return {mul(*ly, inv(a[0]))};
  }
  else {
    auto m = l + (r - l) / 2;
    auto my = ly + (ry - ly) / 2;
    vi A = inter_helper(div(a, tree[2 * v]).second,
      tree, 2 * v, l, m, ly, my);
    vi B = inter_helper(div(a, tree[2 * v + 1]).second
      , tree, 2 * v + 1, m, r, my, ry);
    vi L = conv(A, tree[2 * v + 1]);
    vi R = conv(B, tree[2 * v]);
    REP(i, ssize(R))
      L[i] = add(L[i], R[i]);
    return L;
  }
}
vi inter(const vi& x, const vi& y) { // WYMAGA deriv,
  div, build, inter_helper
  assert(ssize(x) == ssize(y));
  if (x.empty())
    return {};
  vector<vi> tree(4 * ssize(x));
  return inter_helper(deriv(build(tree, 1, begin(x),
    end(x))), tree, 1, begin(x), end(x), begin(y), end
    (y));
}
```

## **power-sum** includes: `lagrange-consecutive`

power_monomial_sum $\mathcal{O}\left(k \log k\right)$, power_binomial_sum $\mathcal{O}\left(k\right)$. power_monomial_sum(a, k, n) liczy $\sum_{i=0}^{n-1} a^i \cdot i^k$, power_binomial_sum(a, k, n) liczy $\sum_{i=0}^{n-1} a^i \cdot \binom{i}{k}$. Działa dla $0 \leq n$ oraz $a \neq 1$.

```
int power_monomial_sum(int a, int k, int n) {
  if (n == 0) return 0;
  int p = 1, b = 1, c = 0, d = a, inva = inv(a);
  vector<int> v(k + 1, k == 0);
  FOR(i, 1, k) v[i] = add(v[i - 1], mul(p = mul(p, a),
    powi(i, k)));
  BinomCoeff bc(k + 1);
  REP(i, k + 1) {
    c = add(c, mul(bc(k + 1, i), mul(v[k - i], b)));
    b = mul(b, sub(0, a));
  }
  c = mul(c, inv(powi(sub(1, a), k + 1)));
  REP(i, k + 1) v[i] = mul(sub(v[i], c), d = mul(d,
    inva));
  return add(c, mul(lagrange_consecutive(v, n - 1),
    powi(a, n - 1)));
}
int power_binomial_sum(int a, int k, int n) {
  int p = powi(a, n), inva1 = inv(sub(a, 1)), binom =
    1, ans = 0;
  BinomCoeff bc(k + 1);
  REP(i, k + 1) {
    ans = sub(mul(p, binom), mul(ans, a));
    if(!i) ans = sub(ans, 1);
    ans = mul(ans, inva1);
    binom = mul(binom, mul(n - i, mul(bc.rev[i + 1],
      bc.fac[i])));
  }
  return ans;
}
```

## **primitive-root** includes: `simple-modulo`, `rho-pollard`

$\mathcal{O}\left(\log^2(mod)\right)$, dla pierwszego $mod$ znajduje generator modulo $mod$ (z być może sporą stałą).

```
int primitive_root() {
  if(mod == 2)
    return 1;
  int q = mod - 1;
  vector<LL> v = factor(q);
  vector<int> fact;
  REP(i, ssize(v))
    if(!i or v[i] != v[i - 1])
      fact.emplace_back(v[i]);
  while(true) {
    int g = rd(2, q);
    auto is_good = [&] {
      for(auto &f : fact)
        if(powi(g, q / f) == 1)
          return false;
      return true;
    };
    if(is_good())
      return g;
  }
}
```

## **pythagorean-triples** Wyznacza wszystkie trójki

$(a, b, c)$ takie, że $a^2 + b^2 = c^2, gcd(a, b, c) = 1$ oraz $c \leq$ limit. Zwraca tylko jedną z $(a, b, c)$ oraz $(b, a, c)$.

```
vector<tuple<int, int, int>> pythagorean_triples(int
  limit) {
  vector<tuple<int, int, int>> ret;
  function<void(int, int, int)> gen = [&](int a, int b
    , int c) {
    if (c > limit)
      return;
    ret.emplace_back(a, b, c);
    REP(i, 3) {
      gen(a + 2 * b + 2 * c, 2 * a + b + 2 * c, 2 * a
        + 2 * b + 3 * c);
      a = -a;
      if (i) b = -b;
    }
  };
  gen(3, 4, 5);
  return ret;
}
```

## **rho-pollard** includes: `miller-rab` $\mathcal{O}\left(n\hat{}\frac{1}{4}\right)$, `factor(n)`

zwraca vector dzielników pierwszych $n$, niekoniecznie posortowany, `get_pairs(n)` zwraca posortowany vector par (dzielnik pierwszych, krotność) dla liczby $n$, `all_factors(n)` zwraca vector wszystkich dzielników $n$, niekoniecznie posortowany, `factor(12) = {2, 2, 3}`, `factor(545423) = {53, 41, 251};`, `get_pairs(12) = {(2, 2), (3, 1)}`, `all_factors(12) = {1, 3, 2, 6, 4, 12}`.

```
LL rho_pollard(LL n) {
  if(n % 2 == 0) return 2;
  for(LL i = 1;; i++) {
    auto f = [&](LL x) { return (llmul(x, x, n) + i) %
```

```
      n; };
    LL x = 2, y = f(x), p;
    while((p = __gcd(n - x + y, n)) == 1)
      x = f(x), y = f(f(y));
    if(p != n) return p;
  }
}
vector<LL> factor(LL n) {
  if(n == 1) return {};
  if(miller_rabin(n)) return {n};
  LL x = rho_pollard(n);
  auto l = factor(x), r = factor(n / x);
  l.insert(l.end(), r.begin(), r.end());
  return l;
}
vector<pair<LL, int>> get_pairs(LL n) {
  auto v = factor(n);
  sort(v.begin(), v.end());
  vector<pair<LL, int>> ret;
  REP(i, ssize(v)) {
    int x = i + 1;
    while (x < ssize(v) and v[x] == v[i])
      ++x;
    ret.emplace_back(v[i], x - i);
    i = x - 1;
  }
  return ret;
}
vector<LL> all_factors(LL n) {
  auto v = get_pairs(n);
  vector<LL> ret;
  function<void(LL,int)> gen = [&](LL val, int p) {
    if (p == ssize(v)) {
      ret.emplace_back(val);
      return;
    }
    auto [x, cnt] = v[p];
    gen(val, p + 1);
    REP(i, cnt) {
      val *= x;
      gen(val, p + 1);
    }
  };
  gen(1, 0);
  return ret;
}
```

### same-div

$\mathcal{O}(\sqrt{n})$, wyznacza przedziały o takiej samej wartości $\lfloor n/x \rfloor$ lub $\lceil n/x \rceil$. same_floor(8) = {(1, 1), (2, 2), (3, 4), (5, 8)}, same_ceil(8) = {(8, 8), (4, 7), (3, 3), (2, 2), (1, 1)}, na konteście raczej checemy przepisać tylko pętlę i od razu wykonywać obliczenia na parze (l, r) zamiast grupować wszyskie przedziały w vectorze. Dla $n$ będącego intem można zmienić wszystkie LL na int, w celu zbicia stałej.

```
vector<pair<LL, LL>> same_floor(LL n) {
  vector<pair<LL, LL>> v;
  for (LL l = 1, r; l <= n; l = r + 1) {
    r = n / (n / l);
    v.emplace_back(l, r);
  }
  return v;
}
vector<pair<LL, LL>> same_ceil(LL n) {
  vector<pair<LL, LL>> v;
  for (LL r = n, l; r >= 1; r = l - 1) {
    l = (n + r - 1) / r;
    l = (n + l - 1) / l;
    v.emplace_back(l, r);
  }
  return v;
}
```

### sieve

$\mathcal{O}(n)$, sieve(n) przetwarza liczby do $n$ włącznie, comp[i] oznacza czy $i$ jest złożone, primes zawiera wszystkie liczby pierwsze <= n, prime_div[i] zawiera najmniejszy dzielnik pierwszy $i$, na CF dla $n = 1e8$ działa w 1.2s.

```
vector<bool> comp;
vector<int> primes, prime_div;
void sieve(int n) {
  primes.clear();
  comp.resize(n + 1);
  prime_div.resize(n + 1);
  FOR(i, 2, n) {
    if (!comp[i]) primes.emplace_back(i), prime_div[i]
      = i;
    for (int p : primes) {
      int x = i * p;
      if (x > n) break;
      comp[x] = true;
      prime_div[x] = p;
      if (i % p == 0) break;
    }
  }
}
```

### simple-modulo

podstawowe operacje na modulo, pamiętać o constexpr.

```
#ifdef CHANGABLE_MOD
int mod = 998'244'353;
#else
constexpr int mod = 998'244'353;
#endif
int add(int a, int b) {
  a += b;
  return a >= mod ? a - mod : a;
}
int sub(int a, int b) {
  return add(a, mod - b);
}
int mul(int a, int b) {
  return int(a * LL(b) % mod);
}
int powi(int a, int b) {
  for(int ret = 1;; b /= 2) {
    if(b == 0)
      return ret;
    if(b & 1)
      ret = mul(ret, a);
    a = mul(a, a);
  }
}
int inv(int x) {
  return powi(x, mod - 2);
}
struct BinomCoeff {
  vector<int> fac, rev;
  BinomCoeff(int n) {
    fac = rev = vector(n + 1, 1);
    FOR(i, 1, n) fac[i] = mul(fac[i - 1], i);
    rev[n] = inv(fac[n]);
    for(int i = n; i > 0; --i)
      rev[i - 1] = mul(rev[i], i);
  }
  int operator()(int n, int k) {
    return mul(fac[n], mul(rev[n - k], rev[k]));
  }
};
```

### simplex

$\mathcal{O}(szybko)$, Simplex(n, m) tworzy lpsolver z $n$ zmiennymi oraz $m$ ograniczeniami, rozwiązuje max $cx$ przy $Ax \leq b$.

```
#define FIND(n, expr) [&] { REP(i, n) if(expr) return
  i; return -1; }()
struct Simplex {
  using T = double;
  const T eps = 1e-9, inf = 1/.0;
  int n, m;
  vector<int> N, B;
  vector<vector<T>> A;
  vector<T> b, c;
  T res = 0;
  Simplex(int vars, int eqs)
    : n(vars), m(eqs), N(n), B(m), A(m, vector<T>(n)),
      b(m), c(n) {
    REP(i, n) N[i] = i;
    REP(i, m) B[i] = n + i;
  }
  void pivot(int eq, int var) {
    T coef = 1 / A[eq][var], k;
    REP(i, n)
      if(abs(A[eq][i]) > eps) A[eq][i] *= coef;
    A[eq][var] *= coef, b[eq] *= coef;
    REP(r, m) if(r != eq && abs(A[r][var]) > eps) {
      k = -A[r][var], A[r][var] = 0;
      REP(i, n) A[r][i] += k * A[eq][i];
      b[r] += k * b[eq];
    }
    k = c[var], c[var] = 0;
    REP(i, n) c[i] -= k * A[eq][i];
    res += k * b[eq];
    swap(B[eq], N[var]);
  }
  bool solve() {
    int eq, var;
    while(true) {
      if((eq = FIND(m, b[i] < -eps)) == -1) break;
      if((var = FIND(n, A[eq][i] < -eps)) == -1) {
        res = -inf; // no solution
        return false;
      }
      pivot(eq, var);
    }
    while(true) {
      if((var = FIND(n, c[i] > eps)) == -1) break;
      eq = -1;
      REP(i, m) if(A[i][var] > eps
        && (eq == -1 || b[i] / A[i][var] < b[eq] / A[
          eq][var]))
        eq = i;
      if(eq == -1) {
        res = inf; // unbound
        return false;
      }
      pivot(eq, var);
    }
    return true;
  }
  vector<T> get_vars() {
    vector<T> vars(n);
    REP(i, m)
      if(B[i] < n) vars[B[i]] = b[i];
    return vars;
  }
};
```

### tonelli-shanks

$\mathcal{O}(\log^2(p))$, dla pierwszego $p$ oraz $0 \leq a \leq p-1$ znajduje takie $x$, że $x^2 \equiv a \pmod{p}$ lub $-1$ jeżeli takie $x$ nie istnieje, można przepisać by działało dla LL

```
int mul(int a, int b, int p) {
  return int(a * LL(b) % p);
}
int powi(int a, int b, int p) {
  for (int ret = 1;; b /= 2) {
    if (!b) return ret;
    if (b & 1) ret = mul(ret, a, p);
    a = mul(a, a, p);
  }
}
int tonelli_shanks(int a, int p) {
  if (a == 0) return 0;
  if (p == 2) return 1;
  if (powi(a, p / 2, p) != 1) return -1;
  int q = p - 1, s = 0, z = 2;
  while (q % 2 == 0) q /= 2, ++s;
  while (powi(z, p / 2, p) == 1) ++z;
  int c = powi(z, q, p), t = powi(a, q, p);
  int r = powi(a, q / 2 + 1, p);
  while (t != 1) {
    int i = 0, x = t;
    while (x != 1) x = mul(x, x, p), ++i;
    c = powi(c, 1 << (s - i - 1), p); // 1ll dla LL
    r = mul(r, c, p), c = mul(c, c, p);
    t = mul(t, c, p), s = i;
  }
  return r;
}
```

### xor-base

$\mathcal{O}(nB + B^2)$ dla $B = bits$, dla $S$ wyznacza minimalny zbiór $B$ taki, że każdy element $S$ można zapisać jako xor jakiegoś podzbioru $B$.

```
int highest_bit(int ai) {
  return ai == 0 ? 0 : __lg(ai) + 1;
}
constexpr int bits = 30;
vector<int> xor_base(vector<int> elems) {
  vector<vector<int>> at_bit(bits + 1);
  for(int ai : elems)
    at_bit[highest_bit(ai)].emplace_back(ai);
  for(int b = bits; b >= 1; --b)
    while(ssize(at_bit[b]) > 1) {
      int ai = at_bit[b].back();
      at_bit[b].pop_back();
      ai ^= at_bit[b].back();
      at_bit[highest_bit(ai)].emplace_back(ai);
    }
  at_bit.erase(at_bit.begin());
  REP(b0, bits - 1)
    for(int a0 : at_bit[b0])
      FOR(b1, b0 + 1, bits - 1)
        for(int &a1 : at_bit[b1])
          if((a1 >> b0) & 1)
            a1 ^= a0;
  vector<int> ret;
  for(auto &v : at_bit) {
    assert(ssize(v) <= 1);
    for(int ai : v)
      ret.emplace_back(ai);
  }
  return ret;
}
```

# Struktury danych (4)

### associative-queue

Kolejka wspierająca dowolną operację łączną, $\mathcal{O}(1)$ zamortyzowany. Konstruktor przyjmuje dwuargumentową funkcję oraz jej element neutralny. Dla minów jest AssocQueue<int> q([](int a, int b){ return min(a, b); }, numeric_limits<int>::max());

```
template<typename T>
struct AssocQueue {
  using fn = function<T(T, T)>;
  fn f;
  vector<pair<T, T>> s1, s2; // {x, f(pref)}
  AssocQueue(fn _f, T e = T()) : f(_f), s1({{e, e}}),
    s2({{e, e}}) {}
  void mv() {
    if (ssize(s2) == 1)
      while (ssize(s1) > 1) {
        s2.emplace_back(s1.back().first, f(s1.back().
          first, s2.back().second));
        s1.pop_back();
      }
  }
  void emplace(T x) {
    s1.emplace_back(x, f(s1.back().second, x));
  }
  void pop() {
    mv();
    s2.pop_back();
  }
  T calc() {
```

```
    return f(s2.back().second, s1.back().second);
  }
  T front() {
    mv();
    return s2.back().first;
  }
  int size() {
    return ssize(s1) + ssize(s2) - 2;
  }
  void clear() {
    s1.resize(1);
    s2.resize(1);
  }
};
```

## fenwick-tree-2d

includes: fenwick-tree $\mathcal{O}(\log^2 n)$, pamięć $\mathcal{O}(n \log n)$, 2D offline, wywołujemy preprocess(x, y) na pozycjach, które chcemy updateować, później init(). update(x, y, val) dodaje val do $[x, y]$, query(x, y) zwraca sumę na prostokącie $(0, 0) - (x, y)$.

```
struct Fenwick2d {
  vector<vector<int>> ys;
  vector<Fenwick> ft;
  Fenwick2d(int limx) : ys(limx) {}
  void preprocess(int x, int y) {
    for(; x < ssize(ys); x |= x + 1)
      ys[x].push_back(y);
  }
  void init() {
    for(auto &v : ys) {
      sort(v.begin(), v.end());
      ft.emplace_back(ssize(v));
    }
  }
  int ind(int x, int y) {
    auto it = lower_bound(ys[x].begin(), ys[x].end(),
      y);
    return int(distance(ys[x].begin(), it));
  }
  void update(int x, int y, LL val) {
    for(; x < ssize(ys); x |= x + 1)
      ft[x].update(ind(x, y), val);
  }
  LL query(int x, int y) {
    LL sum = 0;
    for(x++; x > 0; x &= x - 1)
      sum += ft[x - 1].query(ind(x - 1, y + 1) - 1);
    return sum;
  }
};
```

## fenwick-tree

$\mathcal{O}(\log n)$, indeksowane od 0, update(pos, val) dodaje val do elementu pos, query(pos) zwraca sumę $[0, pos]$.

```
struct Fenwick {
  vector<LL> s;
  Fenwick(int n) : s(n) {}
  void update(int pos, LL val) {
    for(; pos < ssize(s); pos |= pos + 1)
      s[pos] += val;
  }
  LL query(int pos) {
    LL ret = 0;
    for(pos++; pos > 0; pos &= pos - 1)
      ret += s[pos - 1];
    return ret;
  }
  LL query(int l, int r) {
    return query(r) - query(l - 1);
  }
};
```

## find-union

$\mathcal{O}(\alpha(n))$, mniejszy do wiekszego.

```
struct FindUnion {
  vector<int> rep;
  int size(int x) { return -rep[find(x)]; }
  int find(int x) {
    return rep[x] < 0 ? x : rep[x] = find(rep[x]);
  }
  bool same_set(int a, int b) { return find(a) == find
    (b); }
  bool join(int a, int b) {
    a = find(a), b = find(b);
    if(a == b)
      return false;
    if(-rep[a] < -rep[b])
      swap(a, b);
    rep[a] += rep[b];
    rep[b] = a;
    return true;
  }
  FindUnion(int n) : rep(n, -1) {}
};
```

## hash-map

includes: <ext/pb_ds/assoc_container.hpp> $\mathcal{O}(1)$, trzeba przed includem dać undef _GLIBCXX_DEBUG.

```
using namespace __gnu_pbds;
struct chash {
  const uint64_t C = LL(2e18 * acosl(-1)) + 69;
  const int RANDOM = mt19937(0)();
  size_t operator()(uint64_t x) const {
    return __builtin_bswap64((x^RANDOM) * C);
  }
};
template<class L, class R>
using hash_map = gp_hash_table<L, R, chash>;
```

## lazy-segment-tree

Drzewo przedział-przedział, w miarę abstrakcyjne. Wystarczy zmienić Node i funkcje na nim.

```
struct Node {
  LL sum = 0, lazy = 0;
  int sz = 1;
};
void push_to_sons(Node &n, Node &l, Node &r) {
  auto push_to_son = [&](Node &c) {
    c.sum += n.lazy * c.sz;
    c.lazy += n.lazy;
  };
  push_to_son(l);
  push_to_son(r);
  n.lazy = 0;
}
Node merge(Node l, Node r) {
  return Node{
    .sum = l.sum + r.sum,
    .lazy = 0,
    .sz = l.sz + r.sz
  };
}
void add_to_base(Node &n, int val) {
  n.sum += n.sz * LL(val);
  n.lazy += val;
}
struct Tree {
  vector<Node> tree;
  int sz = 1;
  Tree(int n) {
    while(sz < n)
      sz *= 2;
    tree.resize(sz * 2);
    for(int v = sz - 1; v >= 1; v--)
      tree[v] = merge(tree[2 * v], tree[2 * v + 1]);
  }
  void push(int v) {
    push_to_sons(tree[v], tree[2 * v], tree[2 * v +
      1]);
  }
  Node get(int l, int r, int v = 1) {
    if(l == 0 and r == tree[v].sz - 1)
      return tree[v];
    push(v);
    int m = tree[v].sz / 2;
    if(r < m)
      return get(l, r, 2 * v);
    else if(m <= l)
      return get(l - m, r - m, 2 * v + 1);
    else
      return merge(get(l, m - 1, 2 * v), get(0, r - m,
        2 * v + 1));
  }
  void update(int l, int r, int val, int v = 1) {
    if(l == 0 && r == tree[v].sz - 1) {
      add_to_base(tree[v], val);
      return;
    }
    push(v);
    int m = tree[v].sz / 2;
    if(r < m)
      update(l, r, val, 2 * v);
    else if(m <= l)
      update(l - m, r - m, val, 2 * v + 1);
    else {
      update(l, m - 1, val, 2 * v);
      update(0, r - m, val, 2 * v + 1);
    }
    tree[v] = merge(tree[2 * v], tree[2 * v + 1]);
  }
};
```

## lichao-tree

Dla funkcji, których pary przecinają się co najwyżej raz, oblicza minimum w punkcie $x$. Podany kod jest dla funkcji liniowych.

```
constexpr LL inf = LL(1e18);
struct Function {
  int a;
  LL b;
  LL operator()(int x) {
    return x * LL(a) + b;
  }
  Function(int p = 0, LL q = inf) : a(p), b(q) {}
};
ostream& operator<<(ostream &os, Function f) {
  return os << pair(f.a, f.b);
}
struct LiChaoTree {
  int size = 1;
  vector<Function> tree;
  LiChaoTree(int n) {
    while(size < n)
      size *= 2;
    tree.resize(size << 1);
  }
  LL get_min(int x) {
    int v = x + size;
    LL ans = inf;
    while(v) {
      ans = min(ans, tree[v](x));
      v >>= 1;
    }
    return ans;
  }
  void add_func(Function new_func, int v, int l, int r
    ) {
    int m = (l + r) / 2;
    bool domin_l = tree[v](l) > new_func(l),
      domin_m = tree[v](m) > new_func(m);
    if(domin_m)
      swap(tree[v], new_func);
    if(l == r)
      return;
    else if(domin_l == domin_m)
      add_func(new_func, v << 1 | 1, m + 1, r);
    else
      add_func(new_func, v << 1, l, m);
  }
  void add_func(Function new_func) {
    add_func(new_func, 1, 0, size - 1);
  }
};
```

## line-container

$\mathcal{O}(\log n)$ set dla funkcji liniowych, add(a, b) dodaje funkcję $y = ax + b$ query(x) zwraca największe $y$ w punkcie $x$.

```
struct Line {
  mutable LL a, b, p;
  LL eval(LL x) const { return a * x + b; }
  bool operator<(const Line & o) const { return a < o.
    a; }
  bool operator<(LL x) const { return p < x; }
};
struct LineContainer : multiset<Line, less<>> {
  // jak double to inf = 1 / .0, div(a, b) = a / b
  const LL inf = LLONG_MAX;
  LL div(LL a, LL b) { return a / b - ((a ^ b) < 0 &&
    a % b); }
  bool intersect(iterator x, iterator y) {
    if(y == end()) { x->p = inf; return false; }
    if(x->a == y->a) x->p = x->b > y->b ? inf : -inf;
    else x->p = div(y->b - x->b, x->a - y->a);
    return x->p >= y->p;
  }
  void add(LL a, LL b) {
    auto z = insert({a, b, 0}), y = z++, x = y;
    while(intersect(y, z)) z = erase(z);
    if(x != begin() && intersect(--x, y))
      intersect(x, erase(y));
    while((y = x) != begin() && (--x)->p >= y->p)
      intersect(x, erase(y));
  }
  LL query(LL x) {
    assert(!empty());
    return lower_bound(x)->eval(x);
  }
};
```

## link-cut

$\mathcal{O}(q \log n)$ Link-Cut Tree z wyznaczaniem odległości między wierzchołkami, lca w zakorzenionym drzewie, dodawaniem na ścieżce, dodawaniem na poddrzewie, zwracaniem sumy na ścieżce, zwracaniem sumy na poddrzewie. Przepisać co się chce (logika lazy jest tylko w AdditionalInfo, można np. zostawić puste funkcje). Wywołać konstruktor, potem set_value na wierzchołkach (aby się ustawiło, że nie-nil to nie-nil) i potem jazda.

```
struct AdditionalInfo {
  using T = LL;
  static constexpr T neutral = 0; // Remember that
    there is a nil vertex!
  T node_value = neutral, splay_value = neutral;//,
    splay_value_reversed = neutral;
  T whole_subtree_value = neutral, virtual_value =
    neutral;
  T splay_lazy = neutral; // lazy propagation on paths
  T splay_size = 0; // 0 because of nil
  T whole_subtree_lazy = neutral, whole_subtree_cancel
    = neutral; // lazy propagation on subtrees
  T whole_subtree_size = 0, virtual_size = 0; // 0
    because of nil
  void set_value(T x) {
    node_value = splay_value = whole_subtree_value = x
      ;
    splay_size = 1;
    whole_subtree_size = 1;
  }
  void update_from_sons(AdditionalInfo &l,
    AdditionalInfo &r) {
    splay_value = l.splay_value + node_value + r.
      splay_value;
    splay_size = l.splay_size + 1 + r.splay_size;
    whole_subtree_value = l.whole_subtree_value +
      node_value + virtual_value + r.
      whole_subtree_value;
```

```
    whole_subtree_size = l.whole_subtree_size + 1 +
      virtual_size + r.whole_subtree_size;
  }
  void change_virtual(AdditionalInfo &virtual_son, int
      delta) {
    assert(delta == -1 or delta == 1);
    virtual_value += delta * virtual_son.
      whole_subtree_value;
    whole_subtree_value += delta * virtual_son.
      whole_subtree_value;
    virtual_size += delta * virtual_son.
      whole_subtree_size;
    whole_subtree_size += delta * virtual_son.
      whole_subtree_size;
  }
  void push_lazy(AdditionalInfo &l, AdditionalInfo &r,
      bool) {
    l.add_lazy_in_path(splay_lazy);
    r.add_lazy_in_path(splay_lazy);
    splay_lazy = 0;
  }
  void cancel_subtree_lazy_from_parent(AdditionalInfo
    &parent) {
    whole_subtree_cancel = parent.whole_subtree_lazy;
  }
  void pull_lazy_from_parent(AdditionalInfo &parent) {
    if(splay_size == 0) // nil
      return;
    add_lazy_in_subtree(parent.whole_subtree_lazy -
      whole_subtree_cancel);
    cancel_subtree_lazy_from_parent(parent);
  }
  T get_path_sum() {
    return splay_value;
  }
  T get_subtree_sum() {
    return whole_subtree_value;
  }
  void add_lazy_in_path(T x) {
    splay_lazy += x;
    node_value += x;
    splay_value += x * splay_size;
    whole_subtree_value += x * splay_size;
  }
  void add_lazy_in_subtree(T x) {
    whole_subtree_lazy += x;
    node_value += x;
    splay_value += x * splay_size;
    whole_subtree_value += x * whole_subtree_size;
    virtual_value += x * virtual_size;
  }
};
struct Splay {
  struct Node {
    array<int, 2> child;
    int parent;
    int subsize_splay = 1;
    bool lazy_flip = false;
    AdditionalInfo info;
  };
  vector<Node> t;
  const int nil;
  Splay(int n)
  : t(n + 1), nil(n) {
    t[nil].subsize_splay = 0;
    for(Node &v : t)
      v.child[0] = v.child[1] = v.parent = nil;
  }
  void apply_lazy_and_push(int v) {
    auto &[l, r] = t[v].child;
    if(t[v].lazy_flip) {
      for(int c : {l, r})
        t[c].lazy_flip ^= 1;
      swap(l, r);
    }
    t[v].info.push_lazy(t[l].info, t[r].info, t[v].
      lazy_flip);
    for(int c : {l, r})
      if(c != nil)
        t[c].info.pull_lazy_from_parent(t[v].info);
    t[v].lazy_flip = false;
  }
  void update_from_sons(int v) {
    // assumes that v's info is pushed
    auto [l, r] = t[v].child;
    t[v].subsize_splay = t[l].subsize_splay + 1 + t[r
      ].subsize_splay;
    for(int c : {l, r})
      apply_lazy_and_push(c);
    t[v].info.update_from_sons(t[l].info, t[r].info);
  }
  // After that, v is pushed and updated
  void splay(int v) {
    apply_lazy_and_push(v);
    auto set_child = [&](int x, int c, int d) {
      if(x != nil and d != -1)
        t[x].child[d] = c;
      if(c != nil) {
        t[c].parent = x;
        t[c].info.cancel_subtree_lazy_from_parent(t[x
          ].info);
      }
    };
    auto get_dir = [&](int x) -> int {
      int p = t[x].parent;
      if(p == nil or (x != t[p].child[0] and x != t[p
        ].child[1]))
        return -1;
      return t[p].child[1] == x;
    };
    auto rotate = [&](int x, int d) {
      int p = t[x].parent, c = t[x].child[d];
      assert(c != nil);
      set_child(p, c, get_dir(x));
      set_child(x, t[c].child[!d], d);
      set_child(c, x, !d);
      update_from_sons(x);
      update_from_sons(c);
    };
    while(get_dir(v) != -1) {
      int p = t[v].parent, pp = t[p].parent;
      array path_up = {v, p, pp, t[pp].parent};
      for(int i = ssize(path_up) - 1; i >= 0; --i) {
        if(i < ssize(path_up) - 1)
          t[path_up[i]].info.pull_lazy_from_parent(t[
            path_up[i + 1]].info);
        apply_lazy_and_push(path_up[i]);
      }
      int dp = get_dir(v), dpp = get_dir(p);
      if(dpp == -1)
        rotate(p, dp);
      else if(dp == dpp) {
        rotate(pp, dpp);
        rotate(p, dp);
      }
      else {
        rotate(p, dp);
        rotate(pp, dpp);
      }
    }
  }
};
struct LinkCut : Splay {
  LinkCut(int n) : Splay(n) {}
  // Cuts the path from x downward, creates path to
    root, splays x.
  int access(int x) {
    int v = x, cv = nil;
    for(; v != nil; cv = v, v = t[v].parent) {
      splay(v);
      int &right = t[v].child[1];
      t[v].info.change_virtual(t[right].info, +1);
      right = cv;
      t[right].info.pull_lazy_from_parent(t[v].info);
      t[v].info.change_virtual(t[right].info, -1);
      update_from_sons(v);
    }
    splay(x);
    return cv;
  }
  // Changes the root to v.
  // Warning: Linking, cutting, getting the distance,
    etc, changes the root.
  void reroot(int v) {
    access(v);
    t[v].lazy_flip ^= 1;
    apply_lazy_and_push(v);
  }
  // Returns the root of tree containing v.
  int get_leader(int v) {
    access(v);
    while(apply_lazy_and_push(v), t[v].child[0] != nil
      )
      v = t[v].child[0];
    splay(v);
    return v;
  }
  bool is_in_same_tree(int v, int u) {
    return get_leader(v) == get_leader(u);
  }
  // Assumes that v and u aren't in same tree and v !=
    u.
  // Adds edge (v, u) to the forest.
  void link(int v, int u) {
    reroot(v);
    access(u);
    t[u].info.change_virtual(t[v].info, +1);
    assert(t[v].parent == nil);
    t[v].parent = u;
    t[v].info.cancel_subtree_lazy_from_parent(t[u].
      info);
  }
  // Assumes that v and u are in same tree and v != u.
  // Cuts edge going from v to the subtree where is u
  // (in particular, if there is an edge (v, u), it
    deletes it).
  // Returns the cut parent.
  int cut(int v, int u) {
    reroot(u);
    access(v);
    int c = t[v].child[0];
    assert(t[c].parent == v);
    t[v].child[0] = nil;
    t[c].parent = nil;
    t[c].info.cancel_subtree_lazy_from_parent(t[nil].
      info);
    update_from_sons(v);
    while(apply_lazy_and_push(c), t[c].child[1] != nil
      )
      c = t[c].child[1];
    splay(c);
    return c;
  }
  // Assumes that v and u are in same tree.
  // Returns their LCA after a reroot operation.
  int lca(int root, int v, int u) {
    reroot(root);
    if(v == u)
      return v;
    access(v);
    return access(u);
  }
  // Assumes that v and u are in same tree.
  // Returns their distance (in number of edges).
  int dist(int v, int u) {
    reroot(v);
    access(u);
    return t[t[u].child[0]].subsize_splay;
  }
  // Assumes that v and u are in same tree.
  // Returns the sum of values on the path from v to u
    .
  auto get_path_sum(int v, int u) {
    reroot(v);
    access(u);
    return t[u].info.get_path_sum();
  }
  // Assumes that v and u are in same tree.
  // Returns the sum of values on the subtree of v in
    which u isn't present.
  auto get_subtree_sum(int v, int u) {
    u = cut(v, u);
    auto ret = t[v].info.get_subtree_sum();
    link(v, u);
    return ret;
  }
  // Applies function f on vertex v (useful for a
    single add/set operation)
  void apply_on_vertex(int v, function<void (
    AdditionalInfo&)> f) {
    access(v);
    f(t[v].info);
  }
  // Assumes that v and u are in same tree.
  // Adds val to each vertex in path from v to u.
  void add_on_path(int v, int u, int val) {
    reroot(v);
    access(u);
    t[u].info.add_lazy_in_path(val);
  }
  // Assumes that v and u are in same tree.
  // Adds val to each vertex in subtree of v that
    doesn't have u.
  void add_on_subtree(int v, int u, int val) {
    u = cut(v, u);
    t[v].info.add_lazy_in_subtree(val);
    link(v, u);
  }
};
```

## majorized-set

$\mathcal{O}(\log n)$, w s jest zmajoryzowany set, `insert(p)` wrzuca parę p do setu, majoryzuje go (zamortyzowany czas) i zwraca, czy podany element został dodany.

```
template<typename A, typename B>
struct MajorizedSet {
  set<pair<A, B>> s;
  bool insert(pair<A, B> p) {
    auto x = s.lower_bound(p);
    if (x != s.end() && x->second >= p.second)
      return false;
    while (x != s.begin() && (--x)->second <= p.second
      )
      x = s.erase(x);
    s.emplace(p);
    return true;
  }
};
```

## ordered-set

includes: `<ext/pb_ds/assoc_container.hpp>`, `<ext/pb_ds/tree_policy.hpp>` `insert(x)` dodaje element $x$ (nie ma emplace), `find_by_order(i)` zwraca iterator do $i$-tego elementu, `order_of_key(x)` zwraca ile jest mniejszych elementów ($x$ nie musi być w secie). Jeśli chcemy multiseta, to używamy par $(val, id)$.

```
using namespace __gnu_pbds;
template<class T> using ordered_set = tree<
  T,
  null_type,
  less<T>,
  rb_tree_tag,
  tree_order_statistics_node_update
>;
```

## persistent-treap

$\mathcal{O}(\log n)$ Implict Persistent Treap, wszystko indexowane od 0, `insert(i, val)` insertuję na pozycję $i$, kopiowanie struktury działa w $\mathcal{O}(1)$, robimy sobie `vector<Treap>` żeby obsługiwać trwałość UPD. uwaga potencjalnie się kwadraci, spytać

Bartka kiedy

```
mt19937 rng_i(0);
struct Treap {
  struct Node {
    int val, prio, sub = 1;
    Node *l = nullptr, *r = nullptr;
    Node(int _val) : val(_val), prio(int(rng_i())) {}
    ~Node() { delete l; delete r; }
  };
  using pNode = Node*;
  pNode root = nullptr;
  int get_sub(pNode n) { return n ? n->sub : 0; }
  void update(pNode n) {
    if(!n) return;
    n->sub = get_sub(n->l) + get_sub(n->r) + 1;
  }
  void split(pNode t, int i, pNode &l, pNode &r) {
    if(!t) l = r = nullptr;
    else {
      t = new Node(*t);
      if(i <= get_sub(t->l))
        split(t->l, i, l, t->l), r = t;
      else
        split(t->r, i - get_sub(t->l) - 1, t->r, r), l
            = t;
    }
    update(t);
  }
  void merge(pNode &t, pNode l, pNode r) {
    if(!l || !r) t = (l ? l : r);
    else if(l->prio > r->prio) {
      l = new Node(*l);
      merge(l->r, l->r, r), t = l;
    }
    else {
      r = new Node(*r);
      merge(r->l, l, r->l), t = r;
    }
    update(t);
  }
  void insert(pNode &t, int i, pNode it) {
    if(!t) t = it;
    else if(it->prio > t->prio)
      split(t, i, it->l, it->r), t = it;
    else {
      t = new Node(*t);
      if(i <= get_sub(t->l))
        insert(t->l, i, it);
      else
        insert(t->r, i - get_sub(t->l) - 1, it);
    }
    update(t);
  }
  void insert(int i, int val) {
    insert(root, i, new Node(val));
  }
  void erase(pNode &t, int i) {
    if(get_sub(t->l) == i)
      merge(t, t->l, t->r);
    else {
      t = new Node(*t);
      if(i <= get_sub(t->l))
        erase(t->l, i);
      else
        erase(t->r, i - get_sub(t->l) - 1);
    }
    update(t);
  }
  void erase(int i) {
    assert(i < get_sub(root));
    erase(root, i);
  }
};
```

**range-add** includes: fenwick-tree $\mathcal{O}(\log n)$ drzewo przedział-punkt $(+,+)$, wszystko indexowane od 0, `update(l, r, val)` dodaje val na przedziale $[l, r]$, `query(pos)` zwraca wartość elementu $pos$.

```
struct RangeAdd {
  Fenwick f;
  RangeAdd(int n) : f(n) {}
  void update(int l, int r, LL val) {
    f.update(l, val);
    f.update(r + 1, -val);
  }
  LL query(int pos) {
    return f.query(pos);
  }
};
```

**rmq** $\mathcal{O}(n \log n)$ czasowo i pamięciowo, Range Minimum Query z użyciem sparse table, zapytanie jest w $\mathcal{O}(1)$.

```
struct RMQ {
  vector<vector<int>> st;
  RMQ(const vector<int> &a) {
    int n = ssize(a), lg = 0;
    while((1 << lg) < n) lg++;
    st.resize(lg + 1, a);
    FOR(i, 1, lg) REP(j, n) {
      st[i][j] = st[i - 1][j];
      int q = j + (1 << (i - 1));
      if(q < n) st[i][j] = min(st[i][j], st[i - 1][q])
          ;
    }
  }
  int query(int l, int r) {
    int q = __lg(r - l + 1), x = r - (1 << q) + 1;
    return min(st[q][l], st[q][x]);
  }
};
```

**segment-tree** Drzewa punkt-przedział. Pierwsze ustawia w punkcie i podaje max na przedziale. Drugie maxuje elementy na przedziale i podaje wartość w punkcie.

```
struct Tree_Get_Max {
  using T = int;
  T f(T a, T b) { return max(a, b); }
  const T zero = 0;
  vector<T> tree;
  int sz = 1;
  Tree_Get_Max(int n) {
    while(sz < n)
      sz *= 2;
    tree.resize(sz * 2, zero);
  }
  void update(int pos, T val) {
    tree[pos += sz] = val;
    while(pos /= 2)
      tree[pos] = f(tree[pos * 2], tree[pos * 2 + 1]);
  }
  T get(int l, int r) {
    l += sz, r += sz;
    if(l == r)
      return tree[l];
    T ret_l = tree[l], ret_r = tree[r];
    while(l + 1 < r) {
      if(l % 2 == 0)
        ret_l = f(ret_l, tree[l + 1]);
      if(r % 2 == 1)
        ret_r = f(tree[r - 1], ret_r);
      l /= 2, r /= 2;
    }
    return f(ret_l, ret_r);
  }
};
struct Tree_Update_Max_On_Interval {
  using T = int;
  vector<T> tree;
  int sz = 1;
  Tree_Update_Max_On_Interval(int n) {
    while(sz < n)
      sz *= 2;
    tree.resize(sz * 2);
  }
  T get(int pos) {
    T ret = tree[pos += sz];
    while(pos /= 2)
      ret = max(ret, tree[pos]);
    return ret;
  }
  void update(int l, int r, T val) {
    l += sz, r += sz;
    tree[l] = max(tree[l], val);
    if(l == r)
      return;
    tree[r] = max(tree[r], val);
    while(l + 1 < r) {
      if(l % 2 == 0)
        tree[l + 1] = max(tree[l + 1], val);
      if(r % 2 == 1)
        tree[r - 1] = max(tree[r - 1], val);
      l /= 2, r /= 2;
    }
  }
};
```

**treap** $\mathcal{O}(\log n)$ Implict Treap, wszystko indexowane od 0, do Node dopisujemy jakie chcemy mieć trzymać dodatkowo dane. Jeśli chcemy robić lazy, to wykonania push należy wstawić tam gdzie oznaczono komentarzem.

```
namespace Treap {
  mt19937 rng_key(0);
  struct Node {
    int prio, cnt = 1;
    Node *l = nullptr, *r = nullptr;
    Node() : prio(int(rng_key())) {}
    ~Node() { delete l; delete r; }
  };
  using pNode = Node*;
  int get_cnt(pNode t) { return t ? t->cnt : 0; }
  void update(pNode t) {
    if (!t) return;
    // push(t);
    t->cnt = get_cnt(t->l) + get_cnt(t->r) + 1;
  }
  void split(pNode t, int i, pNode &l, pNode &r) {
    if (!t) {
      l = r = nullptr;
      return;
    }
    // push(t);
    if (i <= get_cnt(t->l))
      split(t->l, i, l, t->l), r = t;
    else
      split(t->r, i - get_cnt(t->l) - 1, t->r, r), l =
          t;
    update(t);
  }
  void merge(pNode &t, pNode l, pNode r) {
    if (!l or !r) t = l ?: r;
    else if (l->prio > r->prio) {
      // push(l);
      merge(l->r, l->r, r), t = l;
    }
    else {
      // push(r);
      merge(r->l, l, r->l), t = r;
    }
    update(t);
  }
  void apply_on_interval(pNode &root, int l, int r,
     function<void (pNode)> f) {
    pNode left, mid, right;
    split(root, r + 1, mid, right);
    split(mid, l, left, mid);
    assert(l <= r and mid);
    f(mid);
    merge(mid, left, mid);
    merge(root, mid, right);
  }
}
```

# Grafy (5)

**2sat** $\mathcal{O}(n+m)$, Zwraca poprawne przyporządkowanie zmiennym logicznym dla problemu 2-SAT, albo mówi, że takie nie istnieje. Konstruktor przyjmuje liczbę zmiennych, $\sim$ oznacza negację zmiennej. Po wywołaniu `solve()`, `values[0..n-1]` zawiera wartości rozwiązania.

```
struct TwoSat {
  int n;
  vector<vector<int>> gr;
  vector<int> values;
  TwoSat(int _n = 0) : n(_n), gr(2 * n) {}
  void either(int f, int j) {
    f = max(2 * f, -1 - 2 * f);
    j = max(2 * j, -1 - 2 * j);
    gr[f].emplace_back(j ^ 1);
    gr[j].emplace_back(f ^ 1);
  }
  void set_value(int x) { either(x, x); }
  void implication(int f, int j) { either(~f, j); }
  int add_var() {
    gr.emplace_back();
    gr.emplace_back();
    return n++;
  }
  void at_most_one(vector<int>& li) {
    if(ssize(li) <= 1) return;
    int cur = ~li[0];
    FOR(i, 2, ssize(li) - 1) {
      int next = add_var();
      either(cur, ~li[i]);
      either(cur, next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }
  vector<int> val, comp, z;
  int t = 0;
  int dfs(int i) {
    int low = val[i] = ++t, x;
    z.emplace_back(i);
    for(auto &e : gr[i]) if(!comp[e])
      low = min(low, val[e] ?: dfs(e));
    if(low == val[i]) do {
      x = z.back(); z.pop_back();
      comp[x] = low;
      if (values[x >> 1] == -1)
        values[x >> 1] = x & 1;
    } while (x != i);
    return val[i] = low;
  }
  bool solve() {
    values.assign(n, -1);
    val.assign(2 * n, 0);
    comp = val;
    REP(i, 2 * n) if(!comp[i]) dfs(i);
    REP(i, n) if(comp[2 * i] == comp[2 * i + 1])
      return 0;
    return 1;
  }
};
```

**biconnected** $\mathcal{O}(n+m)$, dwuspójne składowe, mosty oraz punkty artykulacji. po skonstruowaniu, `bicon` = zbiór list id krawędzi, `bridges` = lista id krawędzi będącymi mostami, `arti_points` = lista wierzchołków będącymi punktami artykulacji. Tablice są nieposortowane. Wspiera multikrawędzie i wiele spójnych, ale nie pętle.

```
struct Low {
```

```
  vector<vector<int>> graph;
  vector<int> low, pre;
  vector<pair<int, int>> edges;
  vector<vector<int>> bicon;
  vector<int> bicon_stack, arti_points, bridges;
  int gtime = 0;
  void dfs(int v, int p) {
    low[v] = pre[v] = gtime++;
    bool considered_parent = false;
    int son_count = 0;
    bool is_arti = false;
    for(int e : graph[v]) {
      int u = edges[e].first ^ edges[e].second ^ v;
      if(u == p and not considered_parent)
        considered_parent = true;
      else if(pre[u] == -1) {
        bicon_stack.emplace_back(e);
        dfs(u, v);
        low[v] = min(low[v], low[u]);
        if(low[u] >= pre[v]) {
          bicon.emplace_back();
          do {
            bicon.back().emplace_back(bicon_stack.back
              ());
            bicon_stack.pop_back();
          } while(bicon.back().back() != e);
        }
        ++son_count;
        if(p != -1 and low[u] >= pre[v])
          is_arti = true;
        if(low[u] > pre[v])
          bridges.emplace_back(e);
      }
      else if(pre[v] > pre[u]) {
        low[v] = min(low[v], pre[u]);
        bicon_stack.emplace_back(e);
      }
    }
    if(p == -1 and son_count > 1)
      is_arti = true;
    if(is_arti)
      arti_points.emplace_back(v);
  }
  Low(int n, vector<pair<int, int>> _edges) : graph(n)
    , low(n), pre(n, -1), edges(_edges) {
    REP(i, ssize(edges)) {
      auto [v, u] = edges[i];
#ifdef LOCAL
      assert(v != u);
#endif
      graph[v].emplace_back(i);
      graph[u].emplace_back(i);
    }
    REP(v, n)
      if(pre[v] == -1)
        dfs(v, -1);
  }
};
```

## cactus-cycles

$\mathcal{O}(n)$, wyznaczanie cykli w grafie. Zakłada że jest nieskierowany graf bez pętelek i multikrawędzi, każda krawędź leży na co najwyżej jednym cyklu prostym (silniejsze założenie, niż o wierzchołkach). cactus_cycles(graph) zwraca taką listę cykli, że istnieje krawędź między $i$-tym, a $(i+1)$modssize(cycle)-tym wierzchołkiem.

```
vector<vector<int>> cactus_cycles(vector<vector<int>>
  graph) {
  vector<int> state(ssize(graph), 0), stack;
  vector<vector<int>> ret;
  function<void (int, int)> dfs = [&](int v, int p) {
    if(state[v] == 2) {
      ret.emplace_back(stack.rbegin(), find(stack.
        rbegin(), stack.rend(), v) + 1);
      return;
    }
    stack.emplace_back(v);
    state[v] = 2;
    for(int u : graph[v])
      if(u != p and state[u] != 1)
        dfs(u, v);
    state[v] = 1;
    stack.pop_back();
  };
  REP(i, ssize(graph))
    if(!state[i])
      dfs(i, -1);
  return ret;
}
```

## centro-decomp

$\mathcal{O}(n \log n)$, template do Centroid Decomposition Nie używamy podsz, odwi, ani odwi_cnt Konstruktor przyjmuje liczbę wierzchołków i drzewo. Jeśli chcemy mieć rozbudowane krawędzie, to zmienić tam gdzie zaznaczone. Mamy tablicę odwiedzonych z refreshem w $\mathcal{O}(1)$ (używać bez skrępowania). visit(v) odznacza v jako odwiedzony. is_vis(v) zwraca, czy v jest odwiedzony. refresh(v) zamienia niezablokowane wierzchołki na nieodwiedzone. W decomp mamy standardowe wykonanie CD na poziomie spójnej. Tablica par mówi kto jest naszym ojcem w drzewie CD. root to korzeń drzewa CD.

```
struct CentroDecomp {
  const vector<vector<int>> &graph; // tu
  vector<int> par, podsz, odwi;
  int odwi_cnt = 1;
  const int INF = int(1e9);
  int root;
  void refresh() { ++odwi_cnt; }
  void visit(int v) { odwi[v] = max(odwi[v], odwi_cnt)
    ; }
  bool is_vis(int v) { return odwi[v] >= odwi_cnt; }
  void dfs_podsz(int v) {
    visit(v);
    podsz[v] = 1;
    for (int u : graph[v]) // tu
      if (!is_vis(u)) {
        dfs_podsz(u);
        podsz[v] += podsz[u];
      }
  }
  int centro(int v) {
    refresh();
    dfs_podsz(v);
    int sz = podsz[v] / 2;
    refresh();
    while (true) {
      visit(v);
      for (int u : graph[v]) // tu
        if (!is_vis(u) && podsz[u] > sz) {
          v = u;
          break;
        }
      if (is_vis(v))
        return v;
    }
  }
  void decomp(int v) {
    refresh();
    // Tu kod. Centroid to v, ktory jest juz
      dozywotnie odwiedzony.
    // Koniec kodu.
    refresh();
    for(int u : graph[v]) // tu
      if (!is_vis(u)) {
        u = centro(u);
        par[u] = v;
        odwi[u] = INF;
        // Opcjonalnie tutaj przekazujemy info synowi
          w drzewie CD.
        decomp(u);
      }
  }
  CentroDecomp(int n, vector<vector<int>> &grph) // tu
      : graph(grph), par(n, -1), podsz(n), odwi(n) {
    root = centro(0);
    odwi[root] = INF;
    decomp(root);
  }
};
```

## coloring

$\mathcal{O}(nm)$, wyznacza kolorowanie grafu planaranego. coloring(graph) zwraca 5-kolorowanie grafu coloring(graph, 4) zwraca 4-kolorowanie grafu, jeżeli w każdym momencie procesu usuwania wierzchołka o najmniejszym stopniu jego stopień jest nie większy niż 4

```
vector<int> coloring(const vector<vector<int>>& graph,
    const int limit = 5) {
  const int n = ssize(graph);
  if (!n) return {};
  function<vector<int>(vector<bool>)> solve = [&](
    const vector<bool>& active) {
    if (not *max_element(active.begin(), active.end())
      )
      return vector (n, -1);
    pair<int, int> best = {n, -1};
    REP(i, n) {
      if (not active[i])
        continue;
      int cnt = 0;
      for (int e : graph[i])
        cnt += active[e];
      best = min(best, {cnt, i});
    }
    const int id = best.second;
    auto cp = active;
    cp[id] = false;
    auto col = solve(cp);
    vector<bool> used(limit);
    for (int e : graph[id])
      if (active[e])
        used[col[e]] = true;
    REP(i, limit)
      if (not used[i]) {
        col[id] = i;
        return col;
      }
    for (int e0 : graph[id]) {
      for (int e1 : graph[id]) {
        if (e0 >= e1)
          continue;
        vector<bool> vis(n);
        function<void(int, int, int)> dfs = [&](int v,
          int c0, int c1) {
          vis[v] = true;
          for (int e : graph[v])
            if (not vis[e] and (col[e] == c0 or col[e]
              == c1))
              dfs(e, c0, c1);
        };
        const int c0 = col[e0], c1 = col[e1];
        dfs(e0, c0, c1);
        if (vis[e1])
          continue;
        REP(i, n)
          if (vis[i])
            col[i] = col[i] == c0 ? c1 : c0;
        col[id] = c0;
        return col;
      }
    }
    assert(false);
  };
  return solve(vector (n, true));
}
```

## de-brujin

includes: eulerian-path $\mathcal{O}(k^n)$, ciag/cykl de Brujina słów długości $n$ nad alfabetem $\{0, 1, \ldots, k-1\}$. Jeżeli is_path to zwraca ciąg, wpp. zwraca cykl.

```
vector<int> de_brujin(int k, int n, bool is_path) {
  if (n == 1) {
    vector<int> v(k);
    iota(v.begin(), v.end(), 0);
    return v;
  }
  if (k == 1)
    return vector (n, 0);
  int N = 1;
  REP(i, n - 1)
    N *= k;
  vector<pair<int, int>> edges;
  REP(i, N)
    REP(j, k)
      edges.emplace_back(i, i * k % N + j);
  vector<int> path = get<2>(eulerian_path(N, edges,
    true));
  path.pop_back();
  for(auto& e : path)
    e = e % k;
  if (is_path)
    REP(i, n - 1)
      path.emplace_back(path[i]);
  return path;
}
```

## directed-mst

$\mathcal{O}(m \log n)$, dla korzenia i listy krawędzi skierowanych ważonych zwraca najtańszy podzbiór $n-1$ krawędzi taki, że z korzenia istnieje ścieżka do każdego innego wierzchołka, lub $-1$ gdy nie ma. Zwraca (koszt, ojciec każdego wierzchołka w zwróconym drzewie).

```
struct RollbackUF {
  vector<int> e; vector<pair<int, int>> st;
  RollbackUF(int n) : e(n, -1) {}
  int size(int x) { return -e[find(x)]; }
  int find(int x) { return e[x] < 0 ? x : find(e[x]);
    }
  int time() { return ssize(st); }
  void rollback(int t) {
    for(int i = time(); i --> t;)
      e[st[i].first] = st[i].second;
    st.resize(t);
  }
  bool join(int a, int b) {
    a = find(a), b = find(b);
    if(a == b) return false;
    if(e[a] > e[b]) swap(a, b);
    st.push_back({a, e[a]});
    st.push_back({b, e[b]});
    e[a] += e[b]; e[b] = a;
    return true;
  }
};
struct Edge { int a, b; LL w; };
struct Node {
  Edge key;
  Node *l = 0, *r = 0;
  LL delta = 0;
  void prop() {
    key.w += delta;
    if(l) l->delta += delta;
    if(r) r->delta += delta;
    delta = 0;
  }
};
Node* merge(Node *a, Node *b) {
  if(!a || !b) return a ?: b;
  a->prop(), b->prop();
  if(a->key.w > b->key.w) swap(a, b);
  swap(a->l, (a->r = merge(b, a->r)));
  return a;
}
pair<LL, vector<int>> directed_mst(int n, int r,
  vector<Edge> &g) {
  RollbackUF uf(n);
  vector<Node*> heap(n);
  vector<Node> pool(ssize(g));
```

```
  REP(i, ssize(g)) {
    Edge e = g[i];
    heap[e.b] = merge(heap[e.b], &(pool[i] = Node{e}))
      ;
  }
  LL res = 0;
  vector<int> seen(n, -1), path(n), par(n);
  seen[r] = r;
  vector<Edge> Q(n), in(n, {-1, -1, 0}), comp;
  deque<tuple<int, int, vector<Edge>>> cycs;
  REP(s, n) {
    int u = s, qi = 0, w;
    while(seen[u] < 0) {
      Node *&hu = heap[u];
      if(!hu) return {-1, {}};
      hu->prop();
      Edge e = hu->key;
      hu->delta -= e.w; hu->prop(); hu = merge(hu->l,
        hu->r);
      Q[qi] = e, path[qi++] = u, seen[u] = s;
      res += e.w, u = uf.find(e.a);
      if(seen[u] == s) {
        Node *c = 0;
        int end = qi, time = uf.time();
        do c = merge(c, heap[w = path[--qi]]);
        while(uf.join(u, w));
        u = uf.find(u), heap[u] = c, seen[u] = -1;
        cycs.push_front({u, time, {&Q[qi], &Q[end]}});
      }
    }
    REP(i,qi) in[uf.find(Q[i].b)] = Q[i];
  }
  for(auto [u, t, c] : cycs) { // restore sol (
    optional)
    uf.rollback(t);
    Edge inu = in[u];
    for(auto e : c) in[uf.find(e.b)] = e;
    in[uf.find(inu.b)] = inu;
  }
  REP(i, n) par[i] = in[i].a;
  return {res, par};
}
```

## dominator-tree $\mathcal{O}(m\ \alpha(n))$

dla spójnego DAGu o jednym korzeniu root wyznacza listę synów w dominator tree (które jest drzewem, gdzie ojciec wierzchołka v to najbliższy wierzchołek, którego usunięcie powoduje, że już nie ma ścieżki od korzenia do v).
dominator_tree({{1,2},{3},{4},{4},{5}},0) == {{1,4,2},{3},{},{},{5},{}}

```
vector<vector<int>> dominator_tree(vector<vector<int>>
    dag, int root) {
  int n = ssize(dag);
  vector<vector<int>> t(n), rg(n), bucket(n);
  vector<int> id(n, -1), sdom = id, par = id, idom =
    id, dsu = id, label = id, rev = id;
  function<int (int, int)> find = [&](int v, int x) {
    if(v == dsu[v]) return x ? -1 : v;
    int u = find(dsu[v], x + 1);
    if(u < 0) return v;
    if(sdom[label[dsu[v]]] < sdom[label[v]]) label[v]
      = label[dsu[v]];
    dsu[v] = u;
    return x ? u : label[v];
  };
  int gtime = 0;
  function<void (int)> dfs = [&](int u) {
    rev[gtime] = u;
    label[gtime] = sdom[gtime] = dsu[gtime] = id[u] =
      gtime;
    gtime++;
    for(int w : dag[u]) {
      if(id[w] == -1) dfs(w), par[id[w]] = id[u];
      rg[id[w]].emplace_back(id[u]);
    }
  };
  dfs(root);
  for(int i = n - 1; i >= 0; i--) {
    for(int u : rg[i]) sdom[i] = min(sdom[i], sdom[
      find(u, 0)]);
    if(i > 0) bucket[sdom[i]].push_back(i);
    for(int w : bucket[i]) {
      int v = find(w, 0);
      idom[w] = (sdom[v] == sdom[w] ? sdom[w] : v);
    }
    if(i > 0) dsu[i] = par[i];
  }
  FOR(i, 1, n - 1) {
    if(idom[i] != sdom[i]) idom[i] = idom[idom[i]];
    t[rev[idom[i]]].emplace_back(rev[i]);
  }
  return t;
}
```

## dynamic-connectivity $\mathcal{O}(q \log^2 n)$

offline, zaczyna z pustym grafem, dla danego zapytania stwierdza czy wierzchołki sa w jednej spójnej. Multikrawędzie oraz pętelki działają.

```
enum Event_type { Add, Remove, Query };
vector<bool> dynamic_connectivity(int n, vector<tuple<
  int, int, Event_type>> events) {
  vector<pair<int, int>> queries;
  for(auto &[v, u, t] : events) {
    if(v > u)
      swap(v, u);
    if(t == Query)
      queries.emplace_back(v, u);
  }
  int leaves = 1;
  while(leaves < ssize(queries))
    leaves *= 2;
  vector<vector<pair<int, int>>> edges_to_add(2 *
    leaves);
  map<pair<int, int>, deque<int>> edge_longevity;
  int query_i = 0;
  auto add = [&](int l, int r, pair<int, int> e) {
    if(l > r)
      return;
    debug(l, r, e);
    l += leaves;
    r += leaves;
    while(l <= r) {
      if(l % 2 == 1)
        edges_to_add[l++].emplace_back(e);
      if(r % 2 == 0)
        edges_to_add[r--].emplace_back(e);
      l /= 2;
      r /= 2;
    }
  };
  for(const auto &[v, u, t] : events) {
    auto &que = edge_longevity[pair(v, u)];
    if(t == Add)
      que.emplace_back(query_i);
    else if(t == Remove) {
      if(que.empty())
        continue;
      if(ssize(que) == 1)
        add(que.back(), query_i - 1, pair(v, u));
      que.pop_back();
    }
    else
      ++query_i;
  }
  for(const auto &[e, que] : edge_longevity)
    if(not que.empty())
      add(que.front(), query_i - 1, e);
  vector<bool> ret(ssize(queries));
  vector<int> lead(n), leadsz(n, 1);
  iota(lead.begin(), lead.end(), 0);
  function<int (int)> find = [&](int i) {
    return i == lead[i] ? i : find(lead[i]);
  };
  function<void (int)> dfs = [&](int v) {
    vector<tuple<int, int, int, int>> rollback;
    for(auto [e0, e1] : edges_to_add[v]) {
      e0 = find(e0);
      e1 = find(e1);
      if(e0 == e1)
        continue;
      if(leadsz[e0] > leadsz[e1])
        swap(e0, e1);
      rollback.emplace_back(e0, lead[e0], e1, leadsz[
        e1]);
      leadsz[e1] += leadsz[e0];
      lead[e0] = e1;
    }
    if(v >= leaves) {
      int i = v - leaves;
      assert(i < leaves);
      if(i < ssize(queries))
        ret[i] = find(queries[i].first) == find(
          queries[i].second);
    }
    else {
      dfs(2 * v);
      dfs(2 * v + 1);
    }
    reverse(rollback.begin(), rollback.end());
    for(auto [i, val, j, sz] : rollback) {
      lead[i] = val;
      leadsz[j] = sz;
    }
  };
  dfs(1);
  return ret;
}
```

## eulerian-path $\mathcal{O}(n+m)$

ścieżka eulera. Zwraca tupla (exists, ids, vertices). W exists jest informacja czy jest ścieżka/cykl eulera, ids zawiera id kolejnych krawędzi, vertices zawiera listę wierzchołków na tej ścieżce. Dla cyklu, vertices[0] == vertices[m].

```
tuple<bool, vector<int>, vector<int>> eulerian_path(
  int n, const vector<pair<int, int>> &edges, bool
  directed) {
  vector<int> in(n);
  vector<vector<int>> adj(n);
  int start = 0;
  REP(i, ssize(edges)) {
    auto [a, b] = edges[i];
    start = a;
    ++in[b];
    adj[a].emplace_back(i);
    if (not directed)
      adj[b].emplace_back(i);
  }
  int cnt_in = 0, cnt_out = 0;
  REP(i, n) {
    if (directed) {
      if (abs(ssize(adj[i]) - in[i]) > 1)
        return {};
      if (in[i] < ssize(adj[i]))
        start = i, ++cnt_in;
      else
        cnt_out += in[i] > ssize(adj[i]);
    }
    else if (ssize(adj[i]) % 2)
      start = i, ++cnt_in;
  }
  vector<int> ids, vertices;
  vector<bool> used(ssize(edges));
  function<void (int)> dfs = [&](int v) {
    while (ssize(adj[v])) {
      int id = adj[v].back(), u = v ^ edges[id].first
        ^ edges[id].second;
      adj[v].pop_back();
      if (used[id]) continue;
      used[id] = true;
      dfs(u);
      ids.emplace_back(id);
    }
  };
  dfs(start);
  if (cnt_in + cnt_out > 2 or not all_of(used.begin(),
     used.end(), identity{}))
    return {};
  reverse(ids.begin(), ids.end());
  if (ssize(ids))
    vertices = {start};
  for (int id : ids)
    vertices.emplace_back(vertices.back() ^ edges[id].
      first ^ edges[id].second);
  return {true, ids, vertices};
}
```

## hld $\mathcal{O}(q \log n)$

Heavy-Light Decomposition. get_vertex(v) zwraca pozycję odpowiadającą wierzchołkowi. get_path(v, u) zwraca przedziały do obsługiwania drzewem przedziałowym. get_path(v, u) jeśli robisz operacje na wierzchołkach. get_path(v, u, false) jeśli na krawędziach (nie zawiera lca). get_subtree(v) zwraca przedział preorderów odpowiadający podrzewu v.

```
struct HLD {
  vector<vector<int>> &adj;
  vector<int> sz, pre, pos, nxt, par;
  int t = 0;
  void init(int v, int p = -1) {
    par[v] = p;
    sz[v] = 1;
    if(ssize(adj[v]) > 1 && adj[v][0] == p)
      swap(adj[v][0], adj[v][1]);
    for(int &u : adj[v]) if(u != par[v]) {
      init(u, v);
      sz[v] += sz[u];
      if(sz[u] > sz[adj[v][0]])
        swap(u, adj[v][0]);
    }
  }
  void set_paths(int v) {
    pre[v] = t++;
    for(int &u : adj[v]) if(u != par[v]) {
      nxt[u] = (u == adj[v][0] ? nxt[v] : u);
      set_paths(u);
    }
    pos[v] = t;
  }
  HLD(int n, vector<vector<int>> &_adj)
    : adj(_adj), sz(n), pre(n), pos(n), nxt(n), par(n)
      {
    init(0), set_paths(0);
  }
  int lca(int v, int u) {
    while(nxt[v] != nxt[u]) {
      if(pre[v] < pre[u])
        swap(v, u);
      v = par[nxt[v]];
    }
    return (pre[v] < pre[u] ? v : u);
  }
  vector<pair<int, int>> path_up(int v, int u) {
    vector<pair<int, int>> ret;
    while(nxt[v] != nxt[u]) {
      ret.emplace_back(pre[nxt[v]], pre[v]);
      v = par[nxt[v]];
    }
    if(pre[u] != pre[v]) ret.emplace_back(pre[u] + 1,
      pre[v]);
    return ret;
  }
  int get_vertex(int v) { return pre[v]; }
  vector<pair<int, int>> get_path(int v, int u, bool
    add_lca = true) {
    int w = lca(v, u);
    auto ret = path_up(v, w);
    auto path_u = path_up(u, w);
    if(add_lca) ret.emplace_back(pre[w], pre[w]);
```

```
    ret.insert(ret.end(), path_u.begin(), path_u.end()
      );
    return ret;
  }
  pair<int, int> get_subtree(int v) { return {pre[v],
    pos[v] - 1}; }
};
```

## hld-online-bottom-up

includes: hld $\mathcal{O}\left(q \log^2 n\right)$, rozwala zadania, gdzie wynik to dp bottom-up na drzewie i zmienia się wartość wierzchołka/krawędzi. To zakłada, że da się tak uogólnić tego bottom-up'a, że da się trzymać fragmenty drzewa z "dwoma dziurami" i doczepiać jak LEGO dwa takie fragmenty do siebie.

```
// Information about a single vertex (e.g. color).
// A component contains answers for vertices, not
  edges.
using Value_v = int;
// Probably you want: some information about the up
  vertex, the down vertex,
// answer for whole component, answer containing up,
  answer containing down,
// answer containing both up and down.
struct DpTwoEnds;
// Merge two disjoint-vertex paths. Assume that there
  is an edge
// between "up" vertex of d and "down" vertex od u.
DpTwoEnds merge(DpTwoEnds u, DpTwoEnds d);
// DpOneEnd Contains information about a component
  after forgetting the "down" vertex.
// Probably you want: answer for whole component,
  informations about top vertices.
// It needs a default constructor.
struct DpOneEnd;
// Merge two parallel components. They are vertex-
  disjoint. They do not contain the
// parent (it will be included in the next function).
DpOneEnd merge(DpOneEnd a, DpOneEnd b);
// Assuming that DpOneEnd contain all components of
  the light sons of the parent,
// merge those components once with the parent. It has
  to support passing the
// default/neutral value of DpOneEnd -- it means that
  the vertex doesn't have light sons.
DpTwoEnds merge(DpOneEnd sons, Value_v value_parent);
// From a path that remembers "up" and "down" vertices
  , forget the "down" one.
DpOneEnd two_to_one(DpTwoEnds two);
template<class T> struct Tree {
  int leaves = 1;
  vector<T> tree;
  Tree(int n = 0) {
    while(leaves < n)
      leaves *= 2;
    tree.resize(2 * leaves);
  }
  void set(int i, T t) {
    tree[i += leaves] = t;
    while(i /= 2)
      tree[i] = merge(tree[2 * i], tree[2 * i + 1]);
  }
  T get() { return tree[1]; }
};
struct DpDynamicBottomUp {
  int n;
  HLD hld;
  vector<Tree<DpOneEnd>> tree_sons;
  vector<Tree<DpTwoEnds>> tree_path;
  vector<Value_v> current_values;
  vector<int> which_on_path, which_light_son;
  DpDynamicBottomUp(vector<vector<int>> graph, vector<
    Value_v> initial_values)
    : n(ssize(graph)), hld(n, graph), tree_sons(n),
      tree_path(n), current_values(initial_values),
      which_on_path(n, -1), which_light_son(n, -1) {
    function<void (int, int*)> dfs = [&](int v, int *
      on_heavy_cnt) {
      int light_sons_cnt = 0, tmp = 0;
      which_on_path[v] = (*(on_heavy_cnt =
        on_heavy_cnt ?: &tmp))++;
      for(int u : hld.adj[v])
        if(u != hld.par[v])
          dfs(u, hld.nxt[u] == u ? which_light_son[u]
            = light_sons_cnt++, nullptr : on_heavy_cnt
            );
      tree_sons[v] = Tree<DpOneEnd>(light_sons_cnt);
      tree_path[v] = Tree<DpTwoEnds>(tmp);
    };
    dfs(0, 0);
    REP(v, n)
      set(v, initial_values[v]);
  }
  void set(int v, int value_vertex) {
    current_values[v] = value_vertex;
    while(true) {
      tree_path[hld.nxt[v]].set(which_on_path[v],
        merge(tree_sons[v].get(), current_values[v]));
      v = hld.nxt[v];
      if(hld.par[v] == -1)
        break;
      tree_sons[hld.par[v]].set(which_light_son[v],
        two_to_one(tree_path[hld.nxt[v]].get()));
      v = hld.par[v];
    }
  }
  DpTwoEnds get() { return tree_path[0].get(); }
};
```

## jump-ptr

$\mathcal{O}\left((n+q)\log n\right)$, `jump_up(v, k)` zwraca wierzchołek o $k$ krawędzi wyżej niż $v$ lub $-1$. `OperationJumpPtr` może otrzymać wynik na ścieżce. Wynik na ścieżce do góry wymaga łączności, wynik dowolnej ścieżki jest poprawny, gdy jest odwrotność wyniku lub przemienna.

```
struct SimpleJumpPtr {
  int bits;
  vector<vector<int>> graph, jmp;
  vector<int> par, dep;
  void par_dfs(int v) {
    for(int u : graph[v])
      if(u != par[v]) {
        par[u] = v;
        dep[u] = dep[v] + 1;
        par_dfs(u);
      }
  }
  SimpleJumpPtr(vector<vector<int>> g = {}, int root =
    0) : graph(g) {
    int n = ssize(graph);
    bits = __lg(max(1, n)) + 1;
    dep.resize(n);
    par.resize(n, -1);
    if(n > 0)
      par_dfs(root);
    jmp.resize(bits, vector<int>(n, -1));
    jmp[0] = par;
    FOR(b, 1, bits - 1)
      REP(v, n)
        if(jmp[b - 1][v] != -1)
          jmp[b][v] = jmp[b - 1][jmp[b - 1][v]];
    debug(graph, jmp);
  }
  int jump_up(int v, int h) {
    for(int b = 0; (1 << b) <= h; ++b)
      if((h >> b) & 1)
        v = jmp[b][v];
    return v;
  }
  int lca(int v, int u) {
    if(dep[v] < dep[u])
      swap(v, u);
    v = jump_up(v, dep[v] - dep[u]);
    if(v == u)
      return v;
    for(int b = bits - 1; b >= 0; b--) {
      if(jmp[b][v] != jmp[b][u]) {
        v = jmp[b][v];
        u = jmp[b][u];
      }
    }
    return par[v];
  }
};
using PathAns = LL;
PathAns merge(PathAns down, PathAns up) {
  return down + up;
}
struct OperationJumpPtr {
  SimpleJumpPtr ptr;
  vector<vector<PathAns>> ans_jmp;
  OperationJumpPtr(vector<vector<pair<int, int>>> g,
    int root = 0) {
    debug(g, root);
    int n = ssize(g);
    vector<vector<int>> unweighted_g(n);
    REP(v, n)
      for(auto [u, w] : g[v]) {
        (void) w;
        unweighted_g[v].emplace_back(u);
      }
    ptr = SimpleJumpPtr(unweighted_g, root);
    ans_jmp.resize(ptr.bits, vector<PathAns>(n));
    REP(v, n)
      for(auto [u, w] : g[v])
        if(u == ptr.par[v])
          ans_jmp[0][v] = PathAns(w);
    FOR(b, 1, ptr.bits - 1)
      REP(v, n)
        if(ptr.jmp[b - 1][v] != -1 and ptr.jmp[b - 1][
          ptr.jmp[b - 1][v]] != -1)
          ans_jmp[b][v] = merge(ans_jmp[b - 1][v],
            ans_jmp[b - 1][ptr.jmp[b - 1][v]]);
  }
  PathAns path_ans_up(int v, int h) {
    PathAns ret = PathAns();
    for(int b = ptr.bits - 1; b >= 0; b--)
      if((h >> b) & 1) {
        ret = merge(ret, ans_jmp[b][v]);
        v = ptr.jmp[b][v];
      }
    return ret;
  }
  PathAns path_ans(int v, int u) { // discards order
    of edges on path
    int l = ptr.lca(v, u);
    return merge(
      path_ans_up(v, ptr.dep[v] - ptr.dep[l]),
      path_ans_up(u, ptr.dep[u] - ptr.dep[l])
    );
  }
};
```

## max-clique

$\mathcal{O}\left(idk\right)$, działa 1s dla n=155 na najgorszych przypadkach (losowe grafy p=.90). Działa szybciej dla grafów rzadkich. Zwraca listę wierzchołków w jakiejś max klice. Pętelki niedozwolone.

```
constexpr int max_n = 500;
vector<int> get_max_clique(vector<bitset<max_n>> e) {
  double limit = 0.025, pk = 0;
  vector<pair<int, int>> V;
  vector<vector<int>> C(ssize(e) + 1);
  vector<int> qmax, q, S(ssize(C)), old(S);
  REP(i, ssize(e)) V.emplace_back(0, i);
  auto init = [&](vector<pair<int, int>>& r) {
    for (auto& v : r) for (auto j : r) v.first += e[v.
      second][j.second];
    sort(r.rbegin(), r.rend());
    int mxD = r[0].first;
    REP(i, ssize(r)) r[i].first = min(i, mxD) + 1;
  };
  function<void (vector<pair<int, int>>&, int)> expand
    = [&](vector<pair<int, int>>& R, int lev) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (ssize(R)) {
      if (ssize(q) + R.back().first <= ssize(qmax))
        return;
      q.emplace_back(R.back().second);
      vector<pair<int, int>> T;
      for(auto [_, v] : R) if (e[R.back().second][v])
        T.emplace_back(0, v);
      if (ssize(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1, mnk = max(ssize(qmax) -
          ssize(q) + 1, 1);
        C[1] = C[2] = {};
        for (auto [_, v] : T) {
          int k = 1;
          while (any_of(C[k].begin(), C[k].end(), [&](
            int i) { return e[v][i]; })) k++;
          if (k > mxk) C[(mxk = k) + 1] = {};
          if (k < mnk) T[j++].second = v;
          C[k].emplace_back(v);
        }
        if (j > 0) T[j - 1].first = 0;
        FOR(k, mnk, mxk) for (int i : C[k]) T[j++] = {
          k, i};
        expand(T, lev + 1);
      } else if (ssize(q) > ssize(qmax)) qmax = q;
      q.pop_back(), R.pop_back();
    }
  };
  init(V), expand(V, 1); return qmax;
}
```

## negative-cycle

$\mathcal{O}\left(nm\right)$ stwierdzanie istnienia i wyznaczanie ujemnego cyklu. cycle spełnia `cycle[i]->cycle[(i+1)%ssize(cycle)]`. Żeby wyznaczyć krawędzie na cyklu, wystarczy wybierać najtańszą krawędź między wierzchołkami.

```
template<class I>
pair<bool, vector<int>> negative_cycle(vector<vector<
  pair<int, I>>> graph) {
  int n = ssize(graph);
  vector<I> dist(n);
  vector<int> from(n, -1);
  int v_on_cycle = -1;
  REP(iter, n) {
    v_on_cycle = -1;
    REP(v, n)
      for(auto [u, w] : graph[v])
        if(dist[u] > dist[v] + w) {
          dist[u] = dist[v] + w;
          from[u] = v;
          v_on_cycle = u;
        }
  }
  if(v_on_cycle == -1)
    return {false, {}};
  REP(iter, n)
    v_on_cycle = from[v_on_cycle];
  vector<int> cycle = {v_on_cycle};
  for(int v = from[v_on_cycle]; v != v_on_cycle; v =
    from[v])
    cycle.emplace_back(v);
  reverse(cycle.begin(), cycle.end());
  return {true, cycle};
}
```

## planar-graph-faces

$\mathcal{O}\left(m \log m\right)$, zakłada, że każdy punkt ma podane współrzędne, punkty są parami różne oraz krawędzie są nieprzecinającymi się odcinkami. Zwraca wszystkie ściany (wewnętrzne posortowane clockwise, zewnętrzne cc). WAŻNE czasem trzeba złączyć wszystkie ściany zewnętrzne (których może być kilka, gdy jest wiele spójnych) w jedną ścianę. Zewnętrzne ściany mogą wyglądać jak kaktusy, a wewnętrzne zawsze są niezdegenerowanym wielokątem.

```
struct Edge {
  int e, from, to;
  // face is on the right of "from -> to"
};
ostream& operator<<(ostream &o, Edge e) {
  return o << vector{e.e, e.from, e.to};
}
struct Face {
  bool is_outside;
  vector<Edge> sorted_edges;
  // edges are sorted clockwise for inside and cc for outside faces
};
ostream& operator<<(ostream &o, Face f) {
  return o << pair(f.is_outside, f.sorted_edges);
}
vector<Face> split_planar_to_faces(vector<pair<int, int>> coord, vector<pair<int, int>> edges) {
  int n = ssize(coord);
  int E = ssize(edges);
  vector<vector<int>> graph(n);
  REP(e, E) {
    auto [v, u] = edges[e];
    graph[v].emplace_back(e);
    graph[u].emplace_back(e);
  }
  vector<int> lead(2 * E);
  iota(lead.begin(), lead.end(), 0);
  function<int (int)> find = [&](int v) {
    return lead[v] == v ? v : lead[v] = find(lead[v]);
  };
  auto side_of_edge = [&](int e, int v, bool outward) {
    return 2 * e + ((v != min(edges[e].first, edges[e].second)) ^ outward);
  };
  REP(v, n) {
    vector<pair<pair<int, int>, int>> sorted;
    for(int e : graph[v]) {
      auto p = coord[edges[e].first ^ edges[e].second ^ v];
      auto center = coord[v];
      sorted.emplace_back(pair(p.first - center.first, p.second - center.second), e);
    }
    sort(sorted.begin(), sorted.end(), [&](pair<pair<int, int>, int> l0, pair<pair<int, int>, int> r0) {
      auto l = l0.first;
      auto r = r0.first;
      bool half_l = l > pair(0, 0);
      bool half_r = r > pair(0, 0);
      if(half_l != half_r)
        return half_l;
      return l.first * LL(r.second) - l.second * LL(r.first) > 0;
    });
    REP(i, ssize(sorted)) {
      int e0 = sorted[i].second;
      int e1 = sorted[(i + 1) % ssize(sorted)].second;
      int side_e0 = side_of_edge(e0, v, true);
      int side_e1 = side_of_edge(e1, v, false);
      lead[find(side_e0)] = find(side_e1);
    }
  }
  vector<vector<int>> comps(2 * E);
  REP(i, 2 * E)
    comps[find(i)].emplace_back(i);
  vector<Face> polygons;
  vector<vector<pair<int, int>>> outgoing_for_face(n);
  REP(leader, 2 * E)
    if(ssize(comps[leader])) {
      for(int id : comps[leader]) {
        int v = edges[id / 2].first;
        int u = edges[id / 2].second;
        if(v > u)
          swap(v, u);
        if(id % 2 == 1)
          swap(v, u);
        outgoing_for_face[v].emplace_back(u, id / 2);
      }
      vector<Edge> sorted_edges;
      function<void (int)> dfs = [&](int v) {
        while(ssize(outgoing_for_face[v])) {
          auto [u, e] = outgoing_for_face[v].back();
          outgoing_for_face[v].pop_back();
          dfs(u);
          sorted_edges.emplace_back(e, v, u);
        }
      };
      dfs(edges[comps[leader].front() / 2].first);
      reverse(sorted_edges.begin(), sorted_edges.end());
      LL area = 0;
      for(auto edge : sorted_edges) {
        auto l = coord[edge.from];
        auto r = coord[edge.to];
        area += l.first * LL(r.second) - l.second * LL(r.first);
      }
      polygons.emplace_back(area >= 0, sorted_edges);
    }
  // Remember that there can be multiple outside faces.
  return polygons;
}
```

## planarity-check

$\mathcal{O}\left(szybko\right)$ ale istnieją przykłady $\mathcal{O}\left(n^2\right)$, przyjmuje graf nieskierowany bez pętelek i multikrawędzi.

```
bool is_planar(vector<vector<int>> graph) {
  int n = ssize(graph), m = 0;
  REP(v, n)
    m += ssize(graph[v]);
  m /= 2;
  if(n <= 3) return true;
  if(m > 3 * n - 6) return false;
  vector<vector<int>> up(n), dn(n);
  vector<int> low(n, -1), pre(n);
  REP(start, n)
    if(low[start] == -1) {
      vector<pair<int, int>> e_up;
      int tm = 0;
      function<void (int, int)> dfs_low = [&](int v, int p) {
        low[v] = pre[v] = tm++;
        for(int u : graph[v])
          if(u != p and low[u] == -1) {
            dn[v].emplace_back(u);
            dfs_low(u, v);
            low[v] = min(low[v], low[u]);
          }
          else if(u != p and pre[u] < pre[v]) {
            up[v].emplace_back(ssize(e_up));
            e_up.emplace_back(v, u);
            low[v] = min(low[v], pre[u]);
          }
      };
      dfs_low(start, -1);
      vector<pair<int, bool>> dsu(ssize(e_up));
      REP(v, ssize(dsu)) dsu[v].first = v;
      function<pair<int, bool> (int)> find = [&](int v) {
        if(dsu[v].first == v)
          return pair(v, false);
        auto [u, ub] = find(dsu[v].first);
        return dsu[v] = pair(u, ub ^ dsu[v].second);
      };
      auto onion = [&](int x, int y, bool flip) {
        auto [v, vb] = find(x);
        auto [u, ub] = find(y);
        if(v == u)
          return not (vb ^ ub ^ flip);
        dsu[v] = {u, vb ^ ub ^ flip};
        return true;
      };
      auto interlace = [&](const vector<int> &ids, int lo) {
        vector<int> ans;
        for(int e : ids)
          if(pre[e_up[e].second] > lo)
            ans.emplace_back(e);
        return ans;
      };
      auto add_fu = [&](const vector<int> &a, const vector<int> &b) {
        FOR(k, 1, ssize(a) - 1)
          if(not onion(a[k - 1], a[k], 0))
            return false;
        FOR(k, 1, ssize(b) - 1)
          if(not onion(b[k - 1], b[k], 0))
            return false;
        return a.empty() or b.empty() or onion(a[0], b[0], 1);
      };
      function<bool (int, int)> dfs_planar = [&](int v, int p) {
        for(int u : dn[v])
          if(not dfs_planar(u, v))
            return false;
        REP(i, ssize(dn[v])) {
          FOR(j, i + 1, ssize(dn[v]) - 1)
            if(not add_fu(interlace(up[dn[v][i]], low[dn[v][j]]),
                  interlace(up[dn[v][j]], low[dn[v][i]])))
              return false;
          for(int j : up[v]) {
            if(e_up[j].first != v)
              continue;
            if(not add_fu(interlace(up[dn[v][i]], pre[e_up[j].second]),
                  interlace({j}, low[dn[v][i]])))
              return false;
          }
        }
        for(int u : dn[v]) {
          for(int idx : up[u])
            if(pre[e_up[idx].second] < pre[p])
              up[v].emplace_back(idx);
          exchange(up[u], {});
        }
        return true;
      };
      if(not dfs_planar(start, -1))
        return false;
    }
  return true;
}
```

## SCC

konstruktor $\mathcal{O}\left(n\right)$, get_compressed $\mathcal{O}\left(n \log n\right)$. `group[v]` to numer silnie spójnej wierzchołka $v$, `order` to toposort, w którym krawędzie idą w lewo (z lewej są liście), `get_compressed()` zwraca graf silnie spójnych, `get_compressed(false)` nie usuwa multikrawędzi.

```
struct SCC {
  int n;
  vector<vector<int>> &graph;
  int group_cnt = 0;
  vector<int> group;
  vector<vector<int>> rev_graph;
  vector<int> order;
  void order_dfs(int v) {
    group[v] = 1;
    for(int u : rev_graph[v])
      if(group[u] == 0)
        order_dfs(u);
    order.emplace_back(v);
  }
  void group_dfs(int v, int color) {
    group[v] = color;
    for(int u : graph[v])
      if(group[u] == -1)
        group_dfs(u, color);
  }
  SCC(vector<vector<int>> &_graph) : graph(_graph) {
    n = ssize(graph);
    rev_graph.resize(n);
    REP(v, n)
      for(int u : graph[v])
        rev_graph[u].emplace_back(v);
    group.resize(n);
    REP(v, n)
      if(group[v] == 0)
        order_dfs(v);
    reverse(order.begin(), order.end());
    debug(order);
    group.assign(n, -1);
    for(int v : order)
      if(group[v] == -1)
        group_dfs(v, group_cnt++);
  }
  vector<vector<int>> get_compressed(bool delete_same = true) {
    vector<vector<int>> ans(group_cnt);
    REP(v, n)
      for(int u : graph[v])
        if(group[v] != group[u])
          ans[group[v]].emplace_back(group[u]);
    if(not delete_same)
      return ans;
    REP(v, group_cnt) {
      sort(ans[v].begin(), ans[v].end());
      ans[v].erase(unique(ans[v].begin(), ans[v].end()), ans[v].end());
    }
    return ans;
  }
};
```

## toposort

$\mathcal{O}\left(n\right)$, `get_toposort_order(g)` zwraca listę wierzchołków takich, że krawędzie są od wierzchołków wcześniejszych w liście do późniejszych. `get_new_vertex_id_from_order(order)` zwraca odwrotność tej permutacji, tzn. dla każdego wierzchołka trzyma jego nowy numer, aby po przenumerowaniu grafu istniały krawędzie tylko do wierzchołków o większych numerach. `permute(elems, new_id)` zwraca przepermutowaną tablicę elems według nowych numerów wierzchołków (przydatne jak się trzyma informacje o wierzchołkach, a chce się zrobić przenumerowanie topologiczne). `renumerate_vertices(...)` zwraca nowy graf, w którym wierzchołki są przenumerowane. Nowy graf: `renumerate_vertices(graph, get_new_vertex_id_from_order(get_toposort_order(graph)))`.

```
vector<int> get_toposort_order(vector<vector<int>> graph) {
  int n = ssize(graph);
  vector<int> indeg(n);
  REP(v, n)
    for(int u : graph[v])
      ++indeg[u];
  vector<int> que;
  REP(v, n)
    if(indeg[v] == 0)
      que.emplace_back(v);
  vector<int> ret;
  while(not que.empty()) {
    int v = que.back();
    que.pop_back();
    ret.emplace_back(v);
    for(int u : graph[v])
      if(--indeg[u] == 0)
        que.emplace_back(u);
  }
  return ret;
```

```
}
vector<int> get_new_vertex_id_from_order(vector<int> order) {
  vector<int> ret(ssize(order), -1);
  REP(v, ssize(order))
    ret[order[v]] = v;
  return ret;
}
template<class T>
vector<T> permute(vector<T> elems, vector<int> new_id) {
  vector<T> ret(ssize(elems));
  REP(v, ssize(elems))
    ret[new_id[v]] = elems[v];
  return ret;
}
vector<vector<int>> renumerate_vertices(vector<vector<int>> graph, vector<int> new_id) {
  int n = ssize(graph);
  vector<vector<int>> ret(n);
  REP(v, n)
    for(int u : graph[v])
      ret[new_id[v]].emplace_back(new_id[u]);
  REP(v, n)
    for(int u : ret[v])
      assert(v < u);
  return ret;
}
```

### triangles

$\mathcal{O}\left(m\sqrt{m}\right)$, liczenie możliwych kształtów podzbiorów trzy- i czterokrawędziowych. Suma zmiennych *3 daje liczbę spójnych 3-elementowych podzbiorów krawędzi, analogicznie suma zmiennych *4.

```
struct Triangles {
  int triangles3 = 0;
  LL stars3 = 0, paths3 = 0;
  LL ps4 = 0, rectangles4 = 0, paths4 = 0;
  __int128_t ys4 = 0, stars4 = 0;
  Triangles(vector<vector<int>> &graph) {
    int n = ssize(graph);
    vector<pair<int, int>> sorted_deg(n);
    REP(i, n)
      sorted_deg[i] = {ssize(graph[i]), i};
    sort(sorted_deg.begin(), sorted_deg.end());
    vector<int> id(n);
    REP(i, n)
      id[sorted_deg[i].second] = i;
    vector<int> cnt(n);
    REP(v, n) {
      for(int u : graph[v]) if(id[v] > id[u])
        cnt[u] = 1;
      for(int u : graph[v]) if(id[v] > id[u]) for(int w : graph[u]) if(id[w] > id[u] and cnt[w]) {
        ++triangles3;
        for(int x : {v, u, w})
          ps4 += ssize(graph[x]) - 2;
      }
      for(int u : graph[v]) if(id[v] > id[u])
        cnt[u] = 0;
      for(int u : graph[v]) if(id[v] > id[u]) for(int w : graph[u]) if(id[v] > id[w])
        rectangles4 += cnt[w]++;
      for(int u : graph[v]) if(id[v] > id[u]) for(int w : graph[u])
        cnt[w] = 0;
    }
    paths3 = -3 * triangles3;
    REP(v, n) for(int u : graph[v]) if(v < u)
      paths3 += (ssize(graph[v]) - 1) * LL(ssize(graph[u]) - 1);
    ys4 = -2 * ps4;
    auto choose2 = [&](int x) { return x * LL(x - 1) / 2; };
    REP(v, n) for(int u : graph[v])
      ys4 += (ssize(graph[v]) - 1) * choose2(ssize(graph[u]) - 1);
    paths4 = -(4 * rectangles4 + 2 * ps4 + 3 * triangles3);
    REP(v, n) {
      int x = 0;
      for(int u : graph[v]) {
        x += ssize(graph[u]) - 1;
        paths4 -= choose2(ssize(graph[u]) - 1);
      }
      paths4 += choose2(x);
    }
    REP(v, n) {
      int s = ssize(graph[v]);
      stars3 += s * LL(s - 1) * LL(s - 2);
      stars4 += s * LL(s - 1) * LL(s - 2) * __int128_t(s - 3);
    }
    stars3 /= 6;
    stars4 /= 24;
  }
};
```

# Flowy i matchingi (6)

### blossom

Jeden rabin powie $\mathcal{O}\left(nm\right)$, drugi rabin powie, że to nawet nie jest $\mathcal{O}\left(n^3\right)$. W grafie nie może być pętelek. Funkcja zwraca match'a, tzn `match[v] == -1` albo z kim jest sparowany $v$. Rozmiar matchingu to $\frac{1}{2}\sum_v$ `int(match[v] != -1)`.

```
vector<int> blossom(vector<vector<int>> graph) {
  int n = ssize(graph), timer = -1;
  REP(v, n)
    for(int u : graph[v])
      assert(v != u);
  vector<int> match(n, -1), label(n), parent(n), orig(n), aux(n, -1), q;
  auto lca = [&](int x, int y) {
    for(++timer; ; swap(x, y)) {
      if(x == -1)
        continue;
      if(aux[x] == timer)
        return x;
      aux[x] = timer;
      x = (match[x] == -1 ? -1 : orig[parent[match[x]]]);
    }
  };
  auto blossom = [&](int v, int w, int a) {
    while(orig[v] != a) {
      parent[v] = w;
      w = match[v];
      if(label[w] == 1) {
        label[w] = 0;
        q.emplace_back(w);
      }
      orig[v] = orig[w] = a;
      v = parent[w];
    }
  };
  auto augment = [&](int v) {
    while(v != -1) {
      int pv = parent[v], nv = match[pv];
      match[v] = pv;
      match[pv] = v;
      v = nv;
    }
  };
  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    label[root] = 0;
    q = {root};
    REP(i, ssize(q)) {
      int v = q[i];
      for(int x : graph[v])
        if(label[x] == -1) {
          label[x] = 1;
          parent[x] = v;
          if(match[x] == -1) {
            augment(x);
            return 1;
          }
          label[match[x]] = 0;
          q.emplace_back(match[x]);
        }
        else if(label[x] == 0 and orig[v] != orig[x]) {
          int a = lca(orig[v], orig[x]);
          blossom(x, v, a);
          blossom(v, x, a);
        }
    }
    return 0;
  };
  REP(i, n)
    if(match[i] == -1)
      bfs(i);
  return match;
}
```

### dinic

$\mathcal{O}\left(V^2E\right)$ Dinic bez skalowania. funkcja `get_flowing()` zwraca dla każdej oryginalnej krawędzi ile przez nią leci.

```
struct Dinic {
  using T = int;
  struct Edge {
    int v, u;
    T flow, cap;
  };
  int n;
  vector<vector<int>> graph;
  vector<Edge> edges;
  Dinic(int N) : n(N), graph(n) {}
  void add_edge(int v, int u, T cap) {
    debug(v, u, cap);
    int e = ssize(edges);
    graph[v].emplace_back(e);
    graph[u].emplace_back(e + 1);
    edges.emplace_back(v, u, 0, cap);
    edges.emplace_back(u, v, 0, 0);
  }
  vector<int> dist;
  bool bfs(int source, int sink) {
    dist.assign(n, 0);
    dist[source] = 1;
    deque<int> que = {source};
    while(ssize(que) and dist[sink] == 0) {
      int v = que.front();
      que.pop_front();
      for(int e : graph[v])
        if(edges[e].flow != edges[e].cap and dist[edges[e].u] == 0) {
          dist[edges[e].u] = dist[v] + 1;
          que.emplace_back(edges[e].u);
        }
    }
    return dist[sink] != 0;
  }
  vector<int> ended_at;
  T dfs(int v, int sink, T flow = numeric_limits<T>::max()) {
    if(flow == 0 or v == sink)
      return flow;
    for(; ended_at[v] != ssize(graph[v]); ++ended_at[v]) {
      Edge &e = edges[graph[v][ended_at[v]]];
      if(dist[v] + 1 == dist[e.u])
        if(T pushed = dfs(e.u, sink, min(flow, e.cap - e.flow))) {
          e.flow += pushed;
          edges[graph[v][ended_at[v]] ^ 1].flow -= pushed;
          return pushed;
        }
    }
    return 0;
  }
  T operator()(int source, int sink) {
    T answer = 0;
    while(bfs(source, sink)) {
      ended_at.assign(n, 0);
      while(T pushed = dfs(source, sink))
        answer += pushed;
    }
    return answer;
  }
  map<pair<int, int>, T> get_flowing() {
    map<pair<int, int>, T> ret;
    REP(v, n)
      for(int i : graph[v]) {
        if(i % 2) // considering only original edges
          continue;
        Edge &e = edges[i];
        ret[pair(v, e.u)] += e.flow;
      }
    return ret;
  }
};
```

### gomory-hu

includes: dinic $\mathcal{O}\left(n^2 + n \cdot dinic(n, m)\right)$, zwraca min cięcie między każdą parą wierzchołków w nieskierowanym ważonym grafie o nieujemnych wagach. `gomory_hu(n, edges)[s][t] == min cut (s, t)`

```
pair<Dinic::T, vector<bool>> get_min_cut(Dinic &dinic, int s, int t) {
  for(Dinic::Edge &e : dinic.edges)
    e.flow = 0;
  Dinic::T flow = dinic(s, t);
  vector<bool> cut(dinic.n);
  REP(v, dinic.n)
    cut[v] = bool(dinic.dist[v]);
  return {flow, cut};
}
vector<vector<Dinic::T>> get_gomory_hu(int n, vector<tuple<int, int, Dinic::T>> edges) {
  Dinic dinic(n);
  for(auto [v, u, cap] : edges) {
    dinic.add_edge(v, u, cap);
    dinic.add_edge(u, v, cap);
  }
  using T = Dinic::T;
  vector<vector<pair<int, T>>> tree(n);
  vector<int> par(n, 0);
  FOR(v, 1, n - 1) {
    auto [flow, cut] = get_min_cut(dinic, v, par[v]);
    FOR(u, v + 1, n - 1)
      if(cut[u] == cut[v] and par[u] == par[v])
        par[u] = v;
    tree[v].emplace_back(par[v], flow);
    tree[par[v]].emplace_back(v, flow);
  }
  T inf = numeric_limits<T>::max();
  vector ret(n, vector(n, inf));
  REP(source, n) {
    function<void (int, int, T)> dfs = [&](int v, int p, T mn) {
      ret[source][v] = mn;
      for(auto [u, flow] : tree[v])
        if(u != p)
          dfs(u, v, min(mn, flow));
    };
    dfs(source, -1, inf);
  }
  return ret;
}
```

### hopcroft-karp

$\mathcal{O}\left(m\sqrt{n}\right)$ Hopcroft-Karp do liczenia matchingu. Przydaje się głównie w aproksymacji, ponieważ po $k$ iteracjach gwarantuje matching o rozmiarze przynajmniej $k/(k+1)\cdot$ best matching. Wierzchołki grafu muszą być podzielone na warstwy

$[0, n0)$ oraz $[n0, n0 + n1)$. Zwraca rozmiar matchingu oraz przypisanie (lub -1, gdy nie jest zmatchowane).

```
pair<int, vector<int>> hopcroft_karp(vector<vector<int
  >> graph, int n0, int n1) {
  assert(n0 + n1 == ssize(graph));
  REP(v, n0 + n1)
    for(int u : graph[v])
      assert((v < n0) != (u < n0));
  vector<int> matched_with(n0 + n1, -1), dist(n0 + 1);
  constexpr int inf = int(1e9);
  vector<int> manual_que(n0 + 1);
  auto bfs = [&] {
    int head = 0, tail = -1;
    fill(dist.begin(), dist.end(), inf);
    REP(v, n0)
      if(matched_with[v] == -1) {
        dist[1 + v] = 0;
        manual_que[++tail] = v;
      }
    while(head <= tail) {
      int v = manual_que[head++];
      if(dist[1 + v] < dist[0])
        for(int u : graph[v])
          if(dist[1 + matched_with[u]] == inf) {
            dist[1 + matched_with[u]] = dist[1 + v] +
              1;
            manual_que[++tail] = matched_with[u];
          }
    }
    return dist[0] != inf;
  };
  function<bool (int)> dfs = [&](int v) {
    if(v == -1)
      return true;
    for(auto u : graph[v])
      if(dist[1 + matched_with[u]] == dist[1 + v] + 1)
        {
        if(dfs(matched_with[u])) {
          matched_with[v] = u;
          matched_with[u] = v;
          return true;
        }
      }
    dist[1 + v] = inf;
    return false;
  };
  int answer = 0;
  for(int iter = 0; bfs(); ++iter)
    REP(v, n0)
      if(matched_with[v] == -1 and dfs(v))
        ++answer;
  return {answer, matched_with};
}
```

## hungarian

$\mathcal{O}(n_0^2 \cdot n_1)$, dla macierzy wag (mogą być ujemne) między dwoma warstami o rozmiarach $n0$ oraz $n1$ ($n0 <= n1$) wyznacza minimalną sumę wag skojarzenia pełnego. Zwraca sumę wag oraz matching.

```
pair<LL, vector<int>> hungarian(vector<vector<int>> a)
  {
  if(a.empty())
    return {0, {}};
  int n0 = ssize(a) + 1, n1 = ssize(a[0]) + 1;
    assert(n0 <= n1);
  vector<int> p(n1), ans(n0 - 1);
  vector<LL> u(n0), v(n1);
  FOR(i, 1, n0 - 1) {
    p[0] = i;
    int j0 = 0;
    vector<LL> dist(n1, numeric_limits<LL>::max());
    vector<int> pre(n1, -1);
    vector<bool> done(n1 + 1);
    do {
      done[j0] = true;
      int i0 = p[j0], j1 = -1;
      LL delta = numeric_limits<LL>::max();
      FOR(j, 1, n1 - 1)
        if(!done[j]) {
          auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
          if(cur < dist[j])
            dist[j] = cur, pre[j] = j0;
          if(dist[j] < delta)
            delta = dist[j], j1 = j;
        }
      REP(j, n1) {
        if(done[j])
          u[p[j]] += delta, v[j] -= delta;
        else
          dist[j] -= delta;
      }
      j0 = j1;
    } while(p[j0]);
    while(j0) {
      int j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  FOR(j, 1, n1 - 1)
    if(p[j])
      ans[p[j] - 1] = j - 1;
  return {-v[0], ans};
}
```

## konig-theorem

includes: `matching`

$\mathcal{O}(n + matching(n, m))$ wyznaczanie w grafie dwudzielnym kolejno minimalnego pokrycia krawędziowego (PK), maksymalnego zbioru niezależnych wierzchołków (NW), minimalnego pokrycia wierzchołkowego (PW) korzystając z maksymalnego zbioru niezależnych krawędzi (NK) (tak zwany matching). Z tw. Koniga zachodzi |NK|=n-|PK|=n-|NW|=|PW|.

```
vector<pair<int, int>> get_min_edge_cover(vector<
  vector<int>> graph) {
  vector<int> match = Matching(graph)().second;
  vector<pair<int, int>> ret;
  REP(v, ssize(match))
    if(match[v] != -1 and v < match[v])
      ret.emplace_back(v, match[v]);
    else if(match[v] == -1 and not graph[v].empty())
      ret.emplace_back(v, graph[v].front());
  return ret;
}
array<vector<int>, 2> get_coloring(vector<vector<int>>
  graph) {
  int n = ssize(graph);
  vector<int> match = Matching(graph)().second;
  vector<int> color(n, -1);
  function<void (int)> dfs = [&](int v) {
    color[v] = 0;
    for(int u : graph[v])
      if(color[u] == -1) {
        color[u] = true;
        dfs(match[u]);
      }
  };
  REP(v, n)
    if(match[v] == -1)
      dfs(v);
  REP(v, n)
    if(color[v] == -1)
      dfs(v);
  array<vector<int>, 2> groups;
  REP(v, n)
    groups[color[v]].emplace_back(v);
  return groups;
}
vector<int> get_max_independent_set(vector<vector<int
  >> graph) {
  return get_coloring(graph)[0];
}
vector<int> get_min_vertex_cover(vector<vector<int>>
  graph) {
  return get_coloring(graph)[1];
}
```

## matching

Średnio około $\mathcal{O}(n \log n)$, najgorzej $\mathcal{O}(n^2)$. Wierzchołki grafu nie muszą być ładnie podzielone na dwia przedziały, musi być po prostu dwudzielny. Na przykład `auto [match_size, match] = Matching(graph)();`.

```
struct Matching {
  vector<vector<int>> &adj;
  vector<int> mat, vis;
  int t = 0, ans = 0;
  bool mat_dfs(int v) {
    vis[v] = t;
    for(int u : adj[v])
      if(mat[u] == -1) {
        mat[u] = v;
        mat[v] = u;
        return true;
      }
    for(int u : adj[v])
      if(vis[mat[u]] != t && mat_dfs(mat[u])) {
        mat[u] = v;
        mat[v] = u;
        return true;
      }
    return false;
  }
  Matching(vector<vector<int>> &_adj) : adj(_adj) {
    mat = vis = vector<int>(ssize(adj), -1);
  }
  pair<int, vector<int>> operator()() {
    int d = -1;
    while(d != 0) {
      d = 0, ++t;
      REP(v, ssize(adj))
        if(mat[v] == -1)
          d += mat_dfs(v);
      ans += d;
    }
    return {ans, mat};
  }
};
```

## mcmf-dijkstra

$\mathcal{O}(VE + |flow|E \log V)$, Min-cost max-flow. Można przepisać funkcję get_flowing() z Dinic'a. Kiedy wie się coś więcej o początkowym grafie np. że jest DAG-iem lub że ma tylko nieujemne wagi krawędzi, można napisać własne calc_init_dist by usunąć $VE$ ze złożoności. Jeżeli $E = \mathcal{O}(V^2)$, to może być lepiej napisać samemu kwadratową dijkstrę.

```
struct MCMF {
  struct Edge {
    int v, u, flow, cap;
    LL cost;
    friend ostream& operator<<(ostream &os, Edge &e) {
      return os << vector<LL>{e.v, e.u, e.flow, e.cap,
        e.cost};
    }
  };
  int n;
  const LL inf_LL = 1e18;
  const int inf_int = 1e9;
  vector<vector<int>> graph;
  vector<Edge> edges;
  vector<LL> init_dist;
  MCMF(int N) : n(N), graph(n), init_dist(n) {}
  void add_edge(int v, int u, int cap, LL cost) {
    int e = ssize(edges);
    graph[v].emplace_back(e);
    graph[u].emplace_back(e + 1);
    edges.emplace_back(v, u, 0, cap, cost);
    edges.emplace_back(u, v, 0, 0, -cost);
  }
  void calc_init_dist(int source) {
    fill(init_dist.begin(), init_dist.end(), inf_LL);
    vector<bool> inside(n);
    inside[source] = true;
    deque<int> que = {source};
    init_dist[source] = 0;
    while (ssize(que)) {
      int v = que.front();
      que.pop_front();
      inside[v] = false;
      for (int i : graph[v]) {
        Edge &e = edges[i];
        if (e.flow < e.cap and init_dist[v] + e.cost <
          init_dist[e.u]) {
          init_dist[e.u] = init_dist[v] + e.cost;
          if (not inside[e.u]) {
            inside[e.u] = true;
            que.emplace_back(e.u);
          }
        }
      }
    }
  }
  pair<int, LL> augment(int source, int sink) {
    vector<bool> vis(n);
    vector<int> from(n, -1);
    vector<LL> dist(n, inf_LL);
    priority_queue<pair<LL, int>, vector<pair<LL, int
      >>, greater<>> que;
    que.emplace(0, source);
    dist[source] = 0;
    while(ssize(que)) {
      auto [d, v] = que.top();
      que.pop();
      if (vis[v]) continue;
      vis[v] = true;
      for (int i : graph[v]) {
        Edge &e = edges[i];
        LL new_dist = d + e.cost + init_dist[v];
        if (not vis[e.u] and e.flow != e.cap and
          new_dist < dist[e.u]) {
          dist[e.u] = new_dist;
          from[e.u] = i;
          que.emplace(new_dist - init_dist[e.u], e.u);
        }
      }
    }
    if (not vis[sink])
      return {0, 0};
    int flow = inf_int, e = from[sink];
    while(e != -1) {
      flow = min(flow, edges[e].cap - edges[e].flow);
      e = from[edges[e].v];
    }
    e = from[sink];
    while(e != -1) {
      edges[e].flow += flow;
      edges[e ^ 1].flow -= flow;
      e = from[edges[e].v];
    }
    init_dist.swap(dist);
    return {flow, flow * init_dist[sink]};
  }
  pair<int, LL> operator()(int source, int sink) {
    calc_init_dist(source);
    int flow = 0;
    LL cost = 0;
    pair<int, LL> got;
    do {
      got = augment(source, sink);
      flow += got.first;
      cost += got.second;
    } while(got.first);
    return {flow, cost};
  }
};
```

## mcmf-spfa

$\mathcal{O}(idk)$, Min-cost max-flow z SPFA. Można przepisać funkcję get_flowing() z Dinic'a.

```
struct MCMF {
```

```cpp
  struct Edge {
    int v, u, flow, cap;
    LL cost;
    friend ostream& operator<<(ostream &os, Edge &e) {
      return os << vector<LL>{e.v, e.u, e.flow, e.cap,
          e.cost};
    }
  };
  int n;
  const LL inf_LL = 1e18;
  const int inf_int = 1e9;
  vector<vector<int>> graph;
  vector<Edge> edges;
  MCMF(int N) : n(N), graph(n) {}
  void add_edge(int v, int u, int cap, LL cost) {
    int e = ssize(edges);
    graph[v].emplace_back(e);
    graph[u].emplace_back(e + 1);
    edges.emplace_back(v, u, 0, cap, cost);
    edges.emplace_back(u, v, 0, 0, -cost);
  }
  pair<int, LL> augment(int source, int sink) {
    vector<LL> dist(n, inf_LL);
    vector<int> from(n, -1);
    dist[source] = 0;
    deque<int> que = {source};
    vector<bool> inside(n);
    inside[source] = true;
    while(ssize(que)) {
      int v = que.front();
      inside[v] = false;
      que.pop_front();
      for(int i : graph[v]) {
        Edge &e = edges[i];
        if(e.flow != e.cap and dist[e.u] > dist[v] + e
            .cost) {
          dist[e.u] = dist[v] + e.cost;
          from[e.u] = i;
          if(not inside[e.u]) {
            inside[e.u] = true;
            que.emplace_back(e.u);
          }
        }
      }
    }
    if(from[sink] == -1)
      return {0, 0};
    int flow = inf_int, e = from[sink];
    while(e != -1) {
      flow = min(flow, edges[e].cap - edges[e].flow);
      e = from[edges[e].v];
    }
    e = from[sink];
    while(e != -1) {
      edges[e].flow += flow;
      edges[e ^ 1].flow -= flow;
      e = from[edges[e].v];
    }
    return {flow, flow * dist[sink]};
  }
  pair<int, LL> operator()(int source, int sink) {
    int flow = 0;
    LL cost = 0;
    pair<int, LL> got;
    do {
      got = augment(source, sink);
      flow += got.first;
      cost += got.second;
    } while(got.first);
    return {flow, cost};
  }
};
```

## weighted-blossom

$\mathcal{O}(N^3)$ (but fast in practice) Taken from: https://judge.yosupo.jp/submission/218005 pdfcompile, weighted_matching::init(n), weighted_matching::add_edge(a, b, c) vector<pii> temp, weighted_matching::solve(temp).first

---

```cpp
#define pii pair<int, int>
namespace weighted_matching{
const int INF = (int)1e9 + 7;
const int MAXN = 1050; //double of possible N
struct E{
  int x, y, w;
};
int n, m;
E G[MAXN][MAXN];
int lab[MAXN], match[MAXN], slack[MAXN], st[MAXN], pa[
  MAXN], flo_from[MAXN][MAXN], S[MAXN], vis[MAXN];
vector<int> flo[MAXN];
queue<int> Q;
void init(int _n) {
  n = _n;
  for(int x = 1; x <= n; ++x)
    for(int y = 1; y <= n; ++y)
      G[x][y] = E{x, y, 0};
}
void add_edge(int x, int y, int w) {
  G[x][y].w = G[y][x].w = w;
}
int e_delta(E e) {
  return lab[e.x] + lab[e.y] - G[e.x][e.y].w * 2;
}
void update_slack(int u, int x) {
  if(!slack[x] || e_delta(G[u][x]) < e_delta(G[slack[x
    ]][x]))
    slack[x] = u;
}
void set_slack(int x) {
  slack[x] = 0;
  for(int u = 1; u <= n; ++u)
    if(G[u][x].w > 0 && st[u] != x && S[st[u]] == 0)
      update_slack(u, x);
}
void q_push(int x) {
  if(x <= n) Q.push(x);
  else for(int i = 0; i < (int)flo[x].size(); ++i)
    q_push(flo[x][i]);
}
void set_st(int x, int b) {
  st[x] = b;
  if(x > n) for(int i = 0; i < (int)flo[x].size(); ++i
    )
    set_st(flo[x][i], b);
}
int get_pr(int b, int xr) {
  int pr = find(flo[b].begin(), flo[b].end(), xr) -
    flo[b].begin();
  if(pr & 1) {
    reverse(flo[b].begin() + 1, flo[b].end());
    return (int)flo[b].size() - pr;
  }
  else return pr;
}
void set_match(int x, int y) {
  match[x] = G[x][y].y;
  if(x <= n) return;
  E e = G[x][y];
  int xr = flo_from[x][e.x], pr = get_pr(x, xr);
  for(int i = 0; i < pr; ++i) set_match(flo[x][i], flo
    [x][i^1]);
  set_match(xr, y);
  rotate(flo[x].begin(), flo[x].begin() + pr, flo[x].
    end());
}
void augment(int x, int y) {
  while(1) {
    int ny = st[match[x]];
    set_match(x, y);
    if(!ny) return;
    set_match(ny, st[pa[ny]]);
    x = st[pa[ny]], y = ny;
  }
}
int get_lca(int x, int y) {
  static int t = 0;
  for(++t; x || y; swap(x, y)) {
    if(x == 0) continue;
    if(vis[x] == t) return x;
    vis[x] = t;
    x = st[match[x]];
    if(x) x = st[pa[x]];
  }
  return 0;
}
void add_blossom(int x, int l, int y) {
  int b = n + 1;
  while(b <= m && st[b]) ++b;
  if(b > m) ++m;
  lab[b] = 0, S[b] = 0;
  match[b] = match[l];
  flo[b].clear();
  flo[b].push_back(l);
  for(int u = x, v; u != l; u = st[pa[v]])
    flo[b].push_back(u), flo[b].push_back(v = st[match
      [u]]), q_push(v);
  reverse(flo[b].begin() + 1, flo[b].end());
  for(int u = y, v; u != l; u = st[pa[v]])
    flo[b].push_back(u), flo[b].push_back(v = st[match
      [u]]), q_push(v);
  set_st(b, b);
  for(int i = 1; i <= m; ++i) G[b][i].w = G[i][b].w =
    0;
  for(int i = 1; i <= n; ++i) flo_from[b][i] = 0;
  for(int i = 0; i < (int)flo[b].size(); ++i) {
    int us = flo[b][i];
    for(int u = 1; u <= m; ++u)
      if(G[b][u].w == 0 || e_delta(G[us][u]) < e_delta
        (G[b][u]))
        G[b][u] = G[us][u], G[u][b] = G[u][us];
    for(int u = 1; u <= n; ++u)
      if(flo_from[us][u])
        flo_from[b][u] = us;
  }
  set_slack(b);
}
void expand_blossom(int b) {
  for(int i = 0; i < (int)flo[b].size(); ++i)
    set_st(flo[b][i], flo[b][i]);
  int xr = flo_from[b][G[b][pa[b]].x], pr = get_pr(b,
    xr);
  for(int i = 0; i < pr; i += 2) {
    int xs = flo[b][i], xns = flo[b][i + 1];
    pa[xs] = G[xns][xs].x;
    S[xs] = 1, S[xns] = 0;
    slack[xs] = 0, set_slack(xns);
    q_push(xns);
  }
  S[xr] = 1, pa[xr] = pa[b];
  for(int i = pr + 1; i < (int)flo[b].size(); ++i) {
    int xs = flo[b][i];
    S[xs] = -1, set_slack(xs);
  }
  st[b] = 0;
}
bool on_found_edge(E e) {
  int x = st[e.x], y = st[e.y];
  if(S[y] == -1) {
    pa[y] = e.x, S[y] = 1;
    int ny = st[match[y]];
    slack[y] = slack[ny] = 0;
    S[ny] = 0, q_push(ny);
  }
  else if(S[y] == 0) {
    int l = get_lca(x, y);
    if(!l) return augment(x, y), augment(y, x), true;
    else add_blossom(x, l, y);
  }
  return false;
}
bool matching() {
  fill(S + 1, S + m + 1, -1);
  fill(slack + 1, slack + m + 1, 0);
  Q = queue<int>();
  for(int x = 1; x <= m; ++x)
    if(st[x] == x && !match[x]) pa[x] = 0, S[x] = 0,
      q_push(x);
  if(Q.empty()) return false;
  while(1) {
    while(Q.size()) {
      int x = Q.front(); Q.pop();
      if(S[st[x]] == 1) continue;
      for(int y = 1; y <= n; ++y) {
        if(G[x][y].w > 0 && st[x] != st[y]) {
          if(e_delta(G[x][y]) == 0) {
            if(on_found_edge(G[x][y])) return true;
          }
          else update_slack(x, st[y]);
        }
      }
    }
    int d = INF;
    for(int b = n + 1; b <= m; ++b)
      if(st[b] == b && S[b] == 1) d = min(d, lab[b] /
        2);
    for(int x = 1; x <= m; ++x)
      if(st[x] == x && slack[x]) {
        if(S[x] == -1) d = min(d, e_delta(G[slack[x]][
          x]));
        else if(S[x] == 0) d = min(d, e_delta(G[slack[
          x]][x]) / 2);
      }
    for(int x = 1; x <= n; ++x) {
      if(S[st[x]] == 0) {
        if(lab[x] <= d) return 0;
        lab[x] -= d;
      }
      else if(S[st[x]] == 1) lab[x] += d;
    }
    for(int b = n + 1; b <= m; ++b)
      if(st[b] == b) {
        if(S[st[b]] == 0) lab[b] += d * 2;
        else if(S[st[b]] == 1) lab[b] -= d * 2;
      }
    Q = queue<int>();
    for(int x = 1; x <= m; ++x)
      if(st[x] == x && slack[x] && st[slack[x]] != x
        && e_delta(G[slack[x]][x]) == 0)
        if(on_found_edge(G[slack[x]][x])) return true;
    for(int b = n + 1; b <= m; ++b)
      if(st[b] == b && S[b] == 1 && lab[b] == 0)
        expand_blossom(b);
  }
  return false;
}
pair<ll, int> solve(vector<pii> &ans) {
  fill(match + 1, match + n + 1, 0);
  m = n;
  int cnt = 0; LL sum = 0;
  for(int u = 0; u <= n; ++u) st[u] = u, flo[u].clear
    ();
  int mx = 0;
  for(int x = 1; x <= n; ++x)
    for(int y = 1; y <= n; ++y){
      flo_from[x][y] = (x == y ? x : 0);
      mx = max(mx, G[x][y].w);
    }
  for(int x = 1; x <= n; ++x) lab[x] = mx;
  while(matching()) ++cnt;
  for(int x = 1; x <= n; ++x)
    if(match[x] && match[x] < x) {
      sum += G[x][match[x]].w;
      ans.push_back({x, G[x][match[x]].y});
    }
  return {sum, cnt};
}
}
```

# Geometria (7)

## 3-points-circle Środek okręgu przez 3 punkty

```
x = 0.5*((x2*x2*y3 + y2*y2*y3 - x1*x1*y3 + x1*x1*y2 -
  y1*y1*y3 + y1*y1*y2 + y1*x3*x3 + y1*y3*y3 - y1*x2*x2
   - y1*y2*y2 - y2*x3*x3 - y2*y3*y3) / (y1*x3 - y1*x2
   - y2*x3 - y3*x1 + y3*x2 + y2*x1));
y = 0.5*((-x1*x3*x3 - x1*y3*y3 + x1*x2*x2 + x1*y2*y2 +
  x2*x3*x3 + x2*y3*y3 - x2*x2*x3 - y2*y2*x3 + x1*x1*
  x3 - x1*x1*x2 + y1*y1*x3 - y1*y1*x2) / (y1*x3 - y1*
  x2 - y2*x3 - y3*x1 + y3*x2 + y2*x1));
```

## advanced-complex includes: point Większość nie działa dla intów.

```
constexpr D pi = acosl(-1);
// nachylenie k-> y = kx + m
D slope(P a, P b) { return tan(arg(b - a)); }
// rzut p na ab
P project(P p, P a, P b) {
  return a + (b - a) * dot(p - a, b - a) / norm(a - b)
    ;
}
// odbicie p wzgledem ab
P reflect(P p, P a, P b) {
  return a + conj((p - a) / (b - a)) * (b - a);
}
// obrot a wzgledem p o theta radianow
P rotate(P a, P p, D theta) {
  return (a - p) * polar(1.0L, theta) + p;
}
// kat ABC, w radianach z przedzialu [0..pi]
D angle(P a, P b, P c) {
  return abs(remainder(arg(a - b) - arg(c - b), 2.0 *
    pi));
}
// szybkie przeciecie prostych, nie dziala dla
   rownoleglych
P intersection(P a, P b, P p, P q) {
  D c1 = cross(p - a, b - a), c2 = cross(q - a, b - a)
    ;
  return (c1 * q - c2 * p) / (c1 - c2);
}
// check czy sa rownolegle
bool is_parallel(P a, P b, P p, P q) {
  P c = (a - b) / (p - q); return equal(c, conj(c));
}
// check czy sa prostopadle
bool is_perpendicular(P a, P b, P p, P q) {
  P c = (a - b) / (p - q); return equal(c, -conj(c));
}
// zwraca takie q, ze (p, q) jest rownolegle do (a, b)
P parallel(P a, P b, P p) {
  return p + a - b;
}
// zwraca takie q, ze (p, q) jest prostopadle do (a, b
   )
P perpendicular(P a, P b, P p) {
  return reflect(p, a, b);
}
// przeciecie srodkowych trojkata
P centro(P a, P b, P c) {
  return (a + b + c) / 3.0L;
}
```

## angle-sort includes: point $\mathcal{O}(n \log n)$, zwraca wektory P posortowane kątowo zgodnie z ruchem wskazówek zegara od najbliższego kątowo do wektora (0, 1) włącznie. Aby posortować po argumencie (kącie) swapujemy x, y, używamy angle-sort i ponownie swapujemy x, y. Zakłada że nie ma punktu (0, 0) na wejściu.

```
vector<P> angle_sort(vector<P> t) {
  for(P p : t) assert(not equal(p, P(0, 0)));
  auto it = partition(t.begin(), t.end(), [](P a){
    return P(0, 0) < a; });
  auto cmp = [&](P a, P b) {
    return sign(cross(a, b)) == -1;
  };
  sort(t.begin(), it, cmp);
  sort(it, t.end(), cmp);
  return t;
}
```

## angle180-intervals includes: angle-sort $\mathcal{O}(n)$, ZAKŁADA że punkty są posortowane kątowo. Zwraca $n$ par $[i, r]$, gdzie r jest maksymalnym cyklicznie indeksem, że wszystkie punkty w tym cyklicznym przedziale są ściśle „po prawej" stronie wektora $(0, 0) - in[i]$, albo są na tej półprostej.

```
vector<pair<int, int>> angle180_intervals(vector<P> in
  ) {
  // in must be sorted by angle
  int n = ssize(in);
  vector<int> nxt(n);
  iota(nxt.begin(), nxt.end(), 1);
  int r = nxt[n - 1] = 0;
  vector<pair<int, int>> ret(n);
  REP(l, n) {
    if(nxt[r] == l) r = nxt[r];
    auto good = [&](int i) {
      auto c = cross(in[l], in[i]);
      if(not equal(c, 0)) return c < 0;
      if((P(0, 0) < in[l]) != (P(0, 0) < in[i]))
        return false;
      return l < i;
    };
    while(nxt[r] != l and good(nxt[r]))
      r = nxt[r];
    ret[l] = {l, r};
  }
  return ret;
}
```

## area includes: point Pole wielokąta, niekoniecznie wypukłego. W vectorze muszą być wierzchołki zgodnie z kierunkiem ruchu zegara. Jeśli $D$ jest intem to może się psuć / 2. area(a, b, c) zwraca pole trójkąta o takich długościach boku.

```
D area(vector<P> pts) {
  int n = ssize(pts);
  D ans = 0;
  REP(i, n) ans += cross(pts[i], pts[(i + 1) % n]);
  return fabsl(ans / 2);
}
D area(D a, D b, D c) {
  D p = (a + b + c) / 2;
  return sqrtl(p * (p - a) * (p - b) * (p - c));
}
```

## circle-intersection includes: point Przecięcia okręgu oraz prostej $ax + by + c = 0$ oraz przecięcia okręgu oraz okręgu. Gdy ssize(circle_circle(...)) == 3 to jest nieskończenie wiele rozwiązań.

```
vector<P> circle_line(D r, D a, D b, D c) {
  D len_ab = a * a + b * b,
    x0 = -a * c / len_ab,
    y0 = -b * c / len_ab,
    d = r * r - c * c / len_ab,
    mult = sqrt(d / len_ab);
  if(sign(d) < 0)
    return {};
  else if(sign(d) == 0)
    return {{x0, y0}};
  return {
    {x0 + b * mult, y0 - a * mult},
    {x0 - b * mult, y0 + a * mult}
  };
}
vector<P> circle_line(D x, D y, D r, D a, D b, D c) {
  return circle_line(r, a, b, c + (a * x + b * y));
}
vector<P> circle_circle(D x1, D y1, D r1, D x2, D y2,
  D r2) {
  x2 -= x1;
  y2 -= y1;
  // now x1 = y1 = 0;
  if(sign(x2) == 0 and sign(y2) == 0) {
    if(equal(r1, r2))
      return {{0, 0}, {0, 0}, {0, 0}}; // inf points
    else
      return {};
  }
  auto vec = circle_line(r1, -2 * x2, -2 * y2,
      x2 * x2 + y2 * y2 + r1 * r1 - r2 * r2);
  for(P &p : vec)
    p += P(x1, y1);
  return vec;
}
```

## circle-tangents includes: point $\mathcal{O}(1)$, dla dwóch okręgów zwraca dwie styczne (wewnętrzne lub zewnętrzne, zależnie od wartości inner). Zwraca 1 + sign(dist(p0, p1) - (inside ? r0 + r1 : abs(r0 - r1))) rozwiązań, albo 0 gdy $p1 = p2$. Działa gdy jakiś promień jest 0 – przydatne do policzenia stycznej punktu do okręgu.

```
vector<pair<P, P>> circle_tangents(P p1, D r1, P p2, D
   r2, bool inner) {
  if(inner) r2 *= -1;
  P d = p2 - p1;
  D dr = r1 - r2, d2 = dot(d, d), h2 = d2 - dr * dr;
  if(equal(d2, 0) or sign(h2) < 0)
    return {};
  vector<pair<P, P>> ret;
  for(D sign : {-1, 1}) {
    P v = (d * dr + P(-d.y(), d.x()) * sqrt(max(D(0),
       h2)) * sign) / d2;
    ret.emplace_back(p1 + v * r1, p2 + v * r2);
  }
  ret.resize(1 + (sign(h2) > 0));
  return ret;
}
```

## closest-pair includes: point $\mathcal{O}(n \log n)$, zakłada ssize(in) > 1.

```
pair<P, P> closest_pair(vector<P> in) {
  set<P> s;
  sort(in.begin(), in.end(), [](P a, P b) { return a.y
    () < b.y(); });
  pair<D, pair<P, P>> ret(1e18, {P(), P()});
  int j = 0;
  for (P p : in) {
    P d(1 + sqrt(ret.first), 0);
    while (in[j].y() <= p.y() - d.x()) s.erase(in[j
      ++]);
    auto lo = s.lower_bound(p - d), hi = s.upper_bound
      (p + d);
    for (; lo != hi; ++lo)
      ret = min(ret, {pow(dist(*lo, p), 2), {*lo, p}})
        ;
    s.insert(p);
  }
  return ret.second;
}
```

## convex-gen includes: point, angle-sort, headers/gen Generatorka wielokątów wypukłych. Zwraca wielokąt z co najmniej $n \cdot$ PROC punktami w zakresie $[-$range, range$]$. Jeśli $n$ $(n > 2)$ jest około range^ $\frac{2}{3}$, to powinno chodzić $\mathcal{O}(n \log n)$. Dla większych $n$ może nie dać rady. Ostatni punkt jest zawsze w $(0, 0)$ - można dodać przesunięcie o wektor dla pełnej losowości.

```
vector<int> num_split(int value, int n) {
  vector<int> v(n, value);
  REP(i, n - 1)
    v[i] = rd(0, value);
  sort(v.begin(), v.end());
  adjacent_difference(v.begin(), v.end(), v.begin());
  return v;
}
vector<int> capped_zero_split(int cap, int n) {
  int m = rd(1, n - 1);
  auto lf = num_split(cap, m);
  auto rg = num_split(cap, n - m);
  for (int i : rg)
    lf.emplace_back(-i);
  return lf;
}
vector<P> gen_convex_polygon(int n, int range, bool
   strictly_convex = false) {
  assert(n > 2);
  vector<P> t;
  const double PROC = 0.9;
  do {
    t.clear();
    auto dx = capped_zero_split(range, n);
    auto dy = capped_zero_split(range, n);
    shuffle(dx.begin(), dx.end(), rng);
    REP (i, n)
      if (dx[i] || dy[i])
        t.emplace_back(dx[i], dy[i]);
    t = angle_sort(t);
    if (strictly_convex) {
      vector<P> nt(1, t[0]);
      FOR (i, 1, ssize(t) - 1) {
        if (!sign(cross(t[i], nt.back())))
          nt.back() += t[i];
        else
          nt.emplace_back(t[i]);
      }
      while (!nt.empty() && !sign(cross(nt.back(), nt
        [0]))) {
        nt[0] += nt.back();
        nt.pop_back();
      }
      t = nt;
    }
  } while (ssize(t) < n * PROC);
  partial_sum(t.begin(), t.end(), t.begin());
  return t;
}
```

## convex-hull-online $\mathcal{O}(\log n)$ na każdą operację dodania, Wyznacza górną otoczkę wypukłą online.

```
using P = pair<int, int>;
LL operator*(P l, P r) {
  return l.first * LL(r.second) - l.second * LL(r.
    first);
}
P operator-(P l, P r) {
  return {l.first - r.first, l.second - r.second};
}
int sign(LL x) {
  return x > 0 ? 1 : x < 0 ? -1 : 0;
}
int dir(P a, P b, P c) {
  return sign((b - a) * (c - b));
}
struct UpperConvexHull {
  set<P> hull;
  void add_point(P p) {
    if(hull.empty()) {
      hull = {p};
      return;
    }
    auto it = hull.lower_bound(p);
    if(*hull.begin() < p and p < *prev(hull.end())) {
      assert(it != hull.end() and it != hull.begin());
      if(dir(*prev(it), p, *it) >= 0)
        return;
    }
    it = hull.emplace(p).first;
    auto have_to_rm = [&](auto iter) {
      if(iter == hull.end() or next(iter) == hull.end
        () or iter == hull.begin())
        return false;
      return dir(*prev(iter), *iter, *next(iter)) >=
        0;
```

```
  };
  while(have_to_rm(next(it)))
    it = prev(hull.erase(next(it)));
  while(it != hull.begin() and have_to_rm(prev(it)))
    it = hull.erase(prev(it));
 }
};
```

## convex-hull

includes: point $\mathcal{O}(n \log n)$, top_bot_hull zwraca osobno górę i dół, hull zwraca punkty na otoczce clockwise gdzie pierwszy jest najbardziej lewym.

```
array<vector<P>, 2> top_bot_hull(vector<P> in) {
  sort(in.begin(), in.end());
  array<vector<P>, 2> ret;
  REP(d, 2) {
    for(auto p : in) {
      while(ssize(ret[d]) > 1 and dir(ret[d].end()
        [-2], ret[d].back(), p) >= 0)
        ret[d].pop_back();
      ret[d].emplace_back(p);
    }
    reverse(in.begin(), in.end());
  }
  return ret;
}
vector<P> hull(vector<P> in) {
  if(ssize(in) <= 1) return in;
  auto ret = top_bot_hull(in);
  REP(d, 2) ret[d].pop_back();
  ret[0].insert(ret[0].end(), ret[1].begin(), ret[1].
    end());
  return ret[0];
}
```

## delaunay-triangulation

$\mathcal{O}(n \log n)$, zwraca zbiór trójkątów sumujący się do otoczki wypukłej, gdzie każdy trójkąt nie zawiera żadnego innego punktu wewnątrz okręgu opisanego (czyli maksymalizuje minimalny kąt trójkątów). Zakłada brak identycznych punktów. W przypadku współliniowości wszystkich punktów zwraca pusty vector. Zwraca vector rozmiaru 3X, gdzie wartości 3i, 3i+1, 3i+2 tworzą counter-clockwise trójkąt. Wśród sąsiadów zawsze jest najbliższy wierzchołek. Euclidean min. spanning tree to podzbiór krawędzi.

```
using PI = pair<int, int>;
typedef struct Quad* Q;
PI distinct(INT_MAX, INT_MAX);
LL dist2(PI p) {
  return p.first * LL(p.first)
    + p.second * LL(p.second);
}
LL operator*(PI a, PI b) {
  return a.first * LL(b.second)
    - a.second * LL(b.first);
}
PI operator-(PI a, PI b) {
  return {a.first - b.first,
    a.second - b.second};
}
LL cross(PI a, PI b, PI c) { return (a - b) * (b - c);
  }
struct Quad {
  Q rot, o = nullptr;
  PI p = distinct;
  bool mark = false;
  Quad(Q _rot) : rot(_rot) {}
  PI& F() { return r()->p; }
  Q& r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
} *H; // it's safe to use in multitests
vector<Q> to_dealloc;
bool is_p_inside_circle(PI p, PI a, PI b, PI c) {
  __int128_t p2 = dist2(p), A = dist2(a)-p2,
      B = dist2(b)-p2, C = dist2(c)-p2;
  return cross(p,a,b) * C + cross(p,b,c) * A + cross(p
    ,c,a) * B > 0;
}
Q makeEdge(PI orig, PI dest) {
  Q r = H;
  if (!r) {
    r = new Quad(new Quad(new Quad(new Quad(0))));
    Q del = r;
    REP(i, 4) {
      to_dealloc.emplace_back(del);
      del = del->rot;
    }
  }
  H = r->o; r->r()->r() = r;
  REP(i, 4) {
    r = r->rot, r->p = distinct;
    r->o = i & 1 ? r : r->r();
  }
  r->p = orig; r->F() = dest;
  return r;
}
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<PI>& s) {
  if (ssize(s) <= 3) {
    Q a = makeEdge(s[0], s[1]);
    Q b = makeEdge(s[1], s.back());
    if (ssize(s) == 2) return {a, a->r()};
    splice(a->r(), b);
    auto side = cross(s[0], s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a,
      side < 0 ? c : b->r()};
  }
  auto valid = [&](Q e, Q base) {
    return cross(e->F(), base->F(), base->p) > 0;
  };
  int half = ssize(s) / 2;
  auto [ra, A] = rec({s.begin(), s.end() - half});
  auto [B, rb] = rec({ssize(s) - half + s.begin(), s.
    end()});
  while ((cross(B->p, A->F(), A->p) < 0
        and (A = A->next()))
        or (cross(A->p, B->F(), B->p) > 0
        and (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;
  auto del = [&](Q init, function<Q (Q)> dir) {
    Q e = dir(init);
    if (valid(e, base))
      while (is_p_inside_circle(dir(e)->F(), base->F()
        , base->p, e->F())) {
        Q t = dir(e);
        splice(e, e->prev());
        splice(e->r(), e->r()->prev());
        e->o = H; H = e; e = t;
      }
    return e;
  };
  while(true) {
    Q LC = del(base->r(), [&](Q q) { return q->o; });
    Q RC = del(base, [&](Q q) { return q->prev(); });
    if (!valid(LC, base) and !valid(RC, base)) break;
    if (!valid(LC, base) or (valid(RC, base)
        and is_p_inside_circle(RC->F(), RC->p, LC->F
          (), LC->p)))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return {ra, rb};
}
vector<PI> triangulate(vector<PI> in) {
  sort(in.begin(), in.end());
  assert(unique(in.begin(), in.end()) == in.end());
  if (ssize(in) < 2) return {};
  Q e = rec(in).first;
  vector<Q> q = {e};
  int qi = 0;
  while (cross(e->o->F(), e->F(), e->p) < 0)
    e = e->o;
  auto add = [&] {
    Q c = e;
    do {
      c->mark = 1;
      in.emplace_back(c->p);
      q.emplace_back(c->r());
      c = c->next();
    } while (c != e);
  };
  add(); in.clear();
  while (qi < ssize(q))
    if (!(e = q[qi++])->mark) add();
  for (Q x : to_dealloc) delete x;
  to_dealloc.clear();
  return in;
}
```

## furthest-pair

includes: convex-hull $\mathcal{O}(n)$ po puszczeniu otoczki, zakłada n >= 2.

```
pair<P, P> furthest_pair(vector<P> in) {
  in = hull(in);
  int n = ssize(in), j = 1;
  pair<D, pair<P, P>> ret;
  REP(i, j)
    for(;; j = (j + 1) % n) {
      ret = max(ret, {dist(in[i], in[j]), {in[i], in[j
        ]}});
      if (sign(cross(in[(j + 1) % n] - in[j], in[i +
        1] - in[i])) <= 0)
        break;
    }
  return ret.second;
}
```

## geo3d

Geo3d od Warsaw Eagles.

```
using LD = long double;
const LD kEps = 1e-9;
const LD kPi = acosl(-1);
LD Sq(LD x) { return x * x; }
struct Point {
  LD x, y;
  Point() {}
  Point(LD a, LD b) : x(a), y(b) {}
  Point(const Point& a) : Point(a.x, a.y) {}
  void operator=(const Point &a) { x = a.x; y = a.y; }
  Point operator+(const Point &a) const { Point p(x +
    a.x, y + a.y); return p; }
  Point operator-(const Point &a) const { Point p(x -
    a.x, y - a.y); return p; }
  Point operator*(LD a) const { Point p(x * a, y * a);
    return p; }
  Point operator/(LD a) const { assert(abs(a) > kEps);
    Point p(x / a, y / a); return p; }
  Point &operator+=(const Point &a) { x += a.x; y += a
    .y; return *this; }
  Point &operator-=(const Point &a) { x -= a.x; y -= a
    .y; return *this; }
  LD CrossProd(const Point &a) const { return x * a.y
    - y * a.x; }
  LD CrossProd(Point a, Point b) const { a -= *this; b
    -= *this; return a.CrossProd(b); }
};
struct Line {
  Point p[2];
  Line(Point a, Point b) { p[0] = a; p[1] = b; }
  Point &operator[](int a) { return p[a]; }
};
struct P3 {
  LD x, y, z;
  P3 operator+(P3 a) { P3 p{x + a.x, y + a.y, z + a.z
    }; return p; }
  P3 operator-(P3 a) { P3 p{x - a.x, y - a.y, z - a.z
    }; return p; }
  P3 operator*(LD a) { P3 p{x * a, y * a, z * a};
    return p; }
  P3 operator/(LD a) { assert(a > kEps); P3 p{x / a, y
     / a, z / a}; return p; }
  P3 &operator+=(P3 a) { x += a.x; y += a.y; z += a.z;
     return *this; }
  P3 &operator-=(P3 a) { x -= a.x; y -= a.y; z -= a.z;
     return *this; }
  P3 &operator*=(LD a) { x *= a; y *= a; z *= a;
    return *this; }
  P3 &operator/=(LD a) { assert(a > kEps); x /= a; y
    /= a; z /= a; return *this; }
  LD &operator[](int a) {
    if (a == 0) return x;
    if (a == 1) return y;
    return z;
  }
  bool IsZero() { return abs(x) < kEps && abs(y) <
    kEps && abs(z) < kEps; }
  LD DotProd(P3 a) { return x * a.x + y * a.y + z * a.
    z; }
  LD Norm() { return sqrt(x * x + y * y + z * z); }
  LD SqNorm() { return x * x + y * y + z * z; }
  void NormalizeSelf() { *this /= Norm(); }
  P3 Normalize() {
    P3 res(*this); res.NormalizeSelf();
    return res;
  }
  LD Dis(P3 a) { return (*this - a).Norm(); }
  pair<LD, LD> SphericalAngles() {
    return {atan2(z, sqrt(x * x + y * y)), atan2(y, x)
      };
  }
  LD Area(P3 p) { return Norm() * p.Norm() * sin(Angle
    (p)) / 2; }
  LD Angle(P3 p) {
    LD a = Norm();
    LD b = p.Norm();
    LD c = Dis(p);
    return acos((a * a + b * b - c * c) / (2 * a * b))
      ;
  }
  LD Angle(P3 p, P3 q) { return p.Angle(q); }
  P3 CrossProd(P3 p) {
    P3 q(*this);
    return {q[1] * p[2] - q[2] * p[1], q[2] * p[0] - q
      [0] * p[2],
          q[0] * p[1] - q[1] * p[0]};
  }
  bool LexCmp(P3 &a, const P3 &b) {
    if (abs(a.x - b.x) > kEps) return a.x < b.x;
    if (abs(a.y - b.y) > kEps) return a.y < b.y;
    return a.z < b.z;
  }
};
struct Line3 {
  P3 p[2];
  P3 &operator[](int a) { return p[a]; }
  friend ostream &operator<<(ostream &out, Line3 m);
};
struct Plane {
  P3 p[3];
  P3 &operator[](int a) { return p[a]; }
  P3 GetNormal() {
    P3 cross = (p[1] - p[0]).CrossProd(p[2] - p[0]);
    return cross.Normalize();
  }
```

```
  void GetPlaneEq(LD &A, LD &B, LD &C, LD &D) {
    P3 normal = GetNormal();
    A = normal[0];
    B = normal[1];
    C = normal[2];
    D = normal.DotProd(p[0]);
    assert(abs(D - normal.DotProd(p[1])) < kEps);
    assert(abs(D - normal.DotProd(p[2])) < kEps);
  }
  vector<P3> GetOrthonormalBase() {
    P3 normal = GetNormal();
    P3 cand = {-normal.y, normal.x, 0};
    if (abs(cand.x) < kEps && abs(cand.y) < kEps) {
      cand = {0, -normal.z, normal.y};
    }
    cand.NormalizeSelf();
    P3 third = Plane{P3{0, 0, 0}, normal, cand}.
      GetNormal();
    assert(abs(normal.DotProd(cand)) < kEps &&
           abs(normal.DotProd(third)) < kEps &&
           abs(cand.DotProd(third)) < kEps);
    return {normal, cand, third};
  }
};
struct Circle3 {
  Plane pl; P3 o; LD r;
};
struct Sphere {
  P3 o;
  LD r;
};
// angle PQR
LD Angle(P3 P, P3 Q, P3 R) { return (P - Q).Angle(R -
  Q); }
P3 ProjPtToLine3(P3 p, Line3 l) { // ok
  P3 diff = l[1] - l[0];
  diff.NormalizeSelf();
  return l[0] + diff * (p - l[0]).DotProd(diff);
}
LD DisPtLine3(P3 p, Line3 l) { // ok
  //  LD area = Area(p, l[0], l[1]); LD dis1 = 2 *
    area / l[0].Dis(l[1]);
  LD dis2 = p.Dis(ProjPtToLine3(p, l)); // assert(abs(
    dis1 - dis2) < kEps);
  return dis2;
}
LD DisPtPlane(P3 p, Plane pl) {
  P3 normal = pl.GetNormal();
  return abs(normal.DotProd(p - pl[0]));
}
P3 ProjPtToPlane(P3 p, Plane pl) {
  P3 normal = pl.GetNormal();
  return p - normal * normal.DotProd(p - pl[0]);
}
bool PtBelongToLine3(P3 p, Line3 l) { return
  DisPtLine3(p, l) < kEps; }
bool Lines3Equal(Line3 p, Line3 l) {
  return PtBelongToLine3(p[0], l) && PtBelongToLine3(p
    [1], l);
}
bool PtBelongToPlane(P3 p, Plane pl) { return
  DisPtPlane(p, pl) < kEps; }
Point PlanePtTo2D(Plane pl, P3 p) { // ok
  assert(PtBelongToPlane(p, pl));
  vector<P3> base = pl.GetOrthonormalBase();
  P3 control{0, 0, 0};
  REP(tr, 3) { control += base[tr] * p.DotProd(base[tr
    ]); }
  assert(PtBelongToPlane(pl[0] + base[1], pl));
  assert(PtBelongToPlane(pl[0] + base[2], pl));
  assert((p - control).IsZero());
  return {p.DotProd(base[1]), p.DotProd(base[2])};
}
Line PlaneLineTo2D(Plane pl, Line3 l) {
  return {PlanePtTo2D(pl, l[0]), PlanePtTo2D(pl, l[1])
    };
}
P3 PlanePtTo3D(Plane pl, Point p) { // ok
  vector<P3> base = pl.GetOrthonormalBase();
  return base[0] * base[0].DotProd(pl[0]) + base[1] *
    p.x + base[2] * p.y;
}
Line3 PlaneLineTo3D(Plane pl, Line l) {
  return {PlanePtTo3D(pl, l[0]), PlanePtTo3D(pl, l[1])
    };
}
Line3 ProjLineToPlane(Line3 l, Plane pl) { // ok
  return {ProjPtToPlane(l[0], pl), ProjPtToPlane(l[1],
     pl)};
}
bool Line3BelongToPlane(Line3 l, Plane pl) {
  return PtBelongToPlane(l[0], pl) && PtBelongToPlane(
    l[1], pl);
}
LD Det(P3 a, P3 b, P3 d) { // ok
  P3 pts[3] = {a, b, d};
  LD res = 0;
  for (int sign : {-1, 1}) {
    REP(st_col, 3) {
      int c = st_col;
      LD prod = 1;
      REP(r, 3) {
        prod *= pts[r][c];
        c = (c + sign + 3) % 3;
      }
      res += sign * prod;
    }
  }
  return res;
}
LD Area(P3 p, P3 q, P3 r) {
  q -= p; r -= p;
  return q.Area(r);
}
vector<Point> InterLineLine(Line &a, Line &b) { //
  working fine
  Point vec_a = a[1] - a[0];
  Point vec_b1 = b[1] - a[0];
  Point vec_b0 = b[0] - a[0];
  LD tr_area = vec_b1.CrossProd(vec_b0);
  LD quad_area = vec_b1.CrossProd(vec_a) + vec_a.
    CrossProd(vec_b0);
  if (abs(quad_area) < kEps) { // parallel or
    coinciding
    if (abs(b[0].CrossProd(b[1], a[0])) < kEps) {
      return {a[0], a[1]};
    } else return {};
  }
  return {a[0] + vec_a * (tr_area / quad_area)};
}
vector<P3> InterLineLine(Line3 k, Line3 l) {
  if (Lines3Equal(k, l)) return {k[0], k[1]};
  if (PtBelongToLine3(l[0], k)) return {l[0]};
  Plane pl{l[0], k[0], k[1]};
  if (!PtBelongToPlane(l[1], pl)) return {};
  Line k2 = PlaneLineTo2D(pl, k);
  Line l2 = PlaneLineTo2D(pl, l);
  vector<Point> inter = InterLineLine(k2, l2);
  vector<P3> res;
  for (auto P : inter) res.push_back(PlanePtTo3D(pl, P
    ));
  return res;
}
LD DisLineLine(Line3 l, Line3 k) { // ok
  Plane together{l[0], l[1], l[0] + k[1] - k[0]}; //
    parallel FIXME
  Line3 proj = ProjLineToPlane(k, together);
  P3 inter = (InterLineLine(l, proj))[0];
  P3 on_k_inter = k[0] + inter - proj[0];
  return inter.Dis(on_k_inter);
}
Plane ParallelPlane(Plane pl, P3 A) { // plane
  parallel to pl going through A
  P3 diff = A - ProjPtToPlane(A, pl);
  return {pl[0] + diff, pl[1] + diff, pl[2] + diff};
}
// image of B in rotation wrt line passing through
  origin s.t. A1->A2
// implemented in more general case with similarity
  instead of rotation
P3 RotateAccordingly(P3 A1, P3 A2, P3 B1) { // ok
  Plane pl{A1, A2, {0, 0, 0}};
  Point A12 = PlanePtTo2D(pl, A1);
  Point A22 = PlanePtTo2D(pl, A2);
  complex<LD> rat = complex<LD>(A22.x, A22.y) /
    complex<LD>(A12.x, A12.y);
  Plane plb = ParallelPlane(pl, B1);
  Point B2 = PlanePtTo2D(plb, B1);
  complex<LD> Brot = rat * complex<LD>(B2.x, B2.y);
  return PlanePtTo3D(plb, {Brot.real(), Brot.imag()});
}
vector<Circle3> InterSpherePlane(Sphere s, Plane pl) {
    // ok
  P3 proj = ProjPtToPlane(s.o, pl);
  LD dis = s.o.Dis(proj);
  if (dis > s.r + kEps) return {};
  if (dis > s.r - kEps) return {{pl, proj, 0}}; // is
    it best choice?
  return {{pl, proj, sqrt(s.r * s.r - dis * dis)}};
}
bool PtBelongToSphere(Sphere s, P3 p) { return abs(s.r
   - s.o.Dis(p)) < kEps; }
struct PointS { // just for conversion purposes,
  probably toEucl suffices
  LD lat, lon;
  P3 toEucl() { return P3{cos(lat) * cos(lon), cos(lat
    ) * sin(lon), sin(lat)}; }
  PointS(P3 p) {
    p.NormalizeSelf();
    lat = asin(p.z);
    lon = acos(p.y / cos(lat));
  }
};
LD DistS(P3 a, P3 b) { return atan2l(b.CrossProd(a).
  Norm(), a.DotProd(b)); }
struct CircleS {
  P3 o; // center of circle on sphere
  LD r; // arc len
  LD area() const { return 2 * kPi * (1 - cos(r)); }
};
CircleS From3(P3 a, P3 b, P3 c) { // any three
  different points
  int tmp = 1;
  if ((a - b).Norm() > (c - b).Norm()) {
    swap(a, c); tmp = -tmp;
  }
  if ((b - c).Norm() > (a - c).Norm()) {
    swap(a, b); tmp = -tmp;
  }
  P3 v = (c - b).CrossProd(b - a);
  v = v * (tmp / v.Norm());
  return CircleS{v, DistS(a, v)};
}
CircleS From2(P3 a, P3 b) { // neither the same nor
  the opposite
  P3 mid = (a + b) / 2;
  mid = mid / mid.Norm();
  return From3(a, mid, b);
}
LD SphAngle(P3 A, P3 B, P3 C) { // angle at A, no two
  points opposite
  LD a = B.DotProd(C);
  LD b = C.DotProd(A);
  LD c = A.DotProd(B);
  return acos((b - a * c) / sqrt((1 - Sq(a)) * (1 - Sq
    (c))));
}
LD TriangleArea(P3 A, P3 B, P3 C) { // no two poins
  opposite
  LD a = SphAngle(C, A, B);
  LD b = SphAngle(A, B, C);
  LD c = SphAngle(B, C, A);
  return a + b + c - kPi;
}
```

## halfplane-intersection

includes: point

$\mathcal{O}(n \log n)$ wyznaczanie punktów na brzegu/otoczce przecięcia podanych półpłaszczyzn. Halfplane(a, b) tworzy półpłaszczyznę wzdłuż prostej $a \to b$ z obszarem po lewej stronie wektora $a \to b$. Jeżeli zostało zwróconych mniej, niż trzy punkty, to pole przecięcia jest puste. Na przykład halfplane_intersection({Halfplane(P(2, 1), P(4, 2)), Halfplane(P(6, 3), P(2, 4)), Halfplane(P(-4, 7), P(4, 2))}) == {(4, 2), (6, 3), (0, 4.5)}. Pole przecięcia jest zawsze ograniczone, ponieważ w kodzie są dodawane cztery półpłaszczyzny o współrzędnych w +/-inf, ale nie należy na tym polegać przez eps oraz błędy precyzji (najlepiej jest zmniejszyć inf tyle, ile się da).

```
struct Halfplane {
  P p, pq;
  D angle;
  Halfplane() {}
  Halfplane(P a, P b) : p(a), pq(b - a) {
    angle = atan2l(pq.imag(), pq.real());
  }
};
ostream& operator<<(ostream&o, Halfplane h) {
  return o << '(' << h.p << ", " << h.pq << ", " << h.
    angle << ')';
}
bool is_outside(Halfplane hi, P p) {
  return sign(cross(hi.pq, p - hi.p)) == -1;
}
P inter(Halfplane s, Halfplane t) {
  D alpha = cross(t.p - s.p, t.pq) / cross(s.pq, t.pq)
    ;
  return s.p + s.pq * alpha;
}
vector<P> halfplane_intersection(vector<Halfplane> h)
  {
  for(int i = 0; i < 4; ++i) {
    constexpr D inf = 1e9;
    array box = {P(-inf, -inf), P(inf, -inf), P(inf,
      inf), P(-inf, inf)};
    h.emplace_back(box[i], box[(i + 1) % 4]);
  }
  sort(h.begin(), h.end(), [&](Halfplane l, Halfplane
    r) {
    return l.angle < r.angle;
  });
  deque<Halfplane> dq;
  for(auto &hi : h) {
    while(ssize(dq) >= 2 and is_outside(hi, inter(dq.
      end()[-1], dq.end()[-2])))
      dq.pop_back();
    while(ssize(dq) >= 2 and is_outside(hi, inter(dq
      [0], dq[1])))
      dq.pop_front();
    if(ssize(dq) and sign(cross(hi.pq, dq.back().pq))
      == 0) {
      if(sign(dot(hi.pq, dq.back().pq)) < 0)
        return {};
      if(is_outside(hi, dq.back().p))
        dq.pop_back();
      else
        continue;
    }
    dq.emplace_back(hi);
  }
  while(ssize(dq) >= 3 and is_outside(dq[0], inter(dq.
    end()[-1], dq.end()[-2])))
    dq.pop_back();
  while(ssize(dq) >= 3 and is_outside(dq.end()[-1],
    inter(dq[0], dq[1])))
    dq.pop_front();
  vector<P> ret;
  REP(i, ssize(dq))
    ret.emplace_back(inter(dq[i], dq[(i + 1) % ssize(
      dq)]));
```

```
	ret.erase(unique(ret.begin(), ret.end(), [&](P l, P r) { return equal(l, r); }), ret.end());
	if(ssize(ret) >= 2 and equal(ret.front(), ret.back()))
		ret.pop_back();
	for(Halfplane hi : h)
		if(ssize(ret) <= 2 and is_outside(hi, ret[0]))
			return {};
	return ret;
}
```

## intersect-lines

includes: `point` $\mathcal{O}(1)$ ale intersect_segments ma sporą stałą (ale działa na wszystkich edge-case'ach). Jeżeli intersect_segments zwróci dwa punkty to wszystkie inf rozwiązań są pomiędzy.

```
P intersect_lines(P a, P b, P c, P d) {
	D c1 = cross(c - a, b - a), c2 = cross(d - a, b - a);
	// c1 == c2 => równolegle
	return (c1 * d - c2 * c) / (c1 - c2);
}
bool on_segment(P a, P b, P p) {
	return equal(cross(a - p, b - p), 0) and sign(dot(a - p, b - p)) <= 0;
}
bool is_intersection_segment(P a, P b, P c, P d) {
	auto aux = [&](D q, D w, D e, D r) {
		return sign(max(q, w) - min(e, r)) >= 0;
	};
	return  aux(c.x(), d.x(), a.x(), b.x()) and aux(a.x(), b.x(), c.x(), d.x())
		and aux(c.y(), d.y(), a.y(), b.y()) and aux(a.y(), b.y(), c.y(), d.y())
		and dir(a, d, c) * dir(b, d, c) != 1
		and dir(d, b, a) * dir(c, b, a) != 1;
}
vector<P> intersect_segments(P a, P b, P c, P d) {
	D acd = cross(c - a, d - c), bcd = cross(c - b, d - c),
	     cab = cross(a - c, b - a), dab = cross(a - d, b - a);
	if(sign(acd) * sign(bcd) < 0 and sign(cab) * sign(dab) < 0)
		return {(a * bcd - b * acd) / (bcd - acd)};
	set<P> s;
	if(on_segment(c, d, a)) s.emplace(a);
	if(on_segment(c, d, b)) s.emplace(b);
	if(on_segment(a, b, c)) s.emplace(c);
	if(on_segment(a, b, d)) s.emplace(d);
	return {s.begin(), s.end()};
}
```

## is-in-hull

includes: `intersect-lines` $\mathcal{O}(\log n)$, zwraca czy punkt jest wewnątrz otoczki h. Zakłada że punkty są clockwise oraz nie ma trzech współliniowych (działa na convex-hull).

```
bool is_in_hull(vector<P> h, P p, bool can_on_edge) {
	if(ssize(h) < 3) return can_on_edge and on_segment(h[0], h.back(), p);
	int l = 1, r = ssize(h) - 1;
	if(dir(h[0], h[l], p) >= can_on_edge or dir(h[0], h[r], p) <= -can_on_edge)
		return false;
	while(r - l > 1) {
		int m = (l + r) / 2;
		(dir(h[0], h[m], p) < 0 ? l : r) = m;
	}
	return dir(h[l], h[r], p) < can_on_edge;
}
```

## line

includes: `point` Konwersja różnych postaci prostej.

```
struct Line {
	D A, B, C;
	// postac ogolna Ax + By + C = 0
	Line(D a, D b, D c) : A(a), B(b), C(c) {}
	tuple<D, D, D> get_tuple() { return {A, B, C}; }
	// postac kierunkowa ax + b = y
	Line(D a, D b) : A(a), B(-1), C(b) {}
	pair<D, D> get_dir() { return {- A / B, - C / B}; }
	// prosta pq
	Line(P p, P q) {
		assert(not equal(p, q));
		if(not equal(p.x(), q.x())) {
			A = (q.y() - p.y()) / (p.x() - q.x());
			B = 1, C = -(A * p.x() + B * p.y());
		}
		else A = 1, B = 0, C = -p.x();
	}
	pair<P, P> get_pts() {
		if(!equal(B, 0)) return { P(0, - C / B), P(1, - (A + C) / B) };
		return { P(- C / A, 0), P(- C / A, 1) };
	}
	D directed_dist(P p) {
		return (A * p.x() + B * p.y() + C) / sqrt(A * A + B * B);
	}
	D dist(P p) {
		return abs(directed_dist(p));
	}
};
```

## point

Wrapper na `std::complex`, definy trzeba dać nad bitsami, wtedy istnieje `p.x()` oraz `p.y()`. abs długość, arg kąt $(-\pi, \pi]$ gdzie $(0, 1)$ daje $\frac{\pi}{2}$, `polar(len, angle)` tworzy $P$. Istnieją atan2, asin, sinh.

```
// Before include bits:
// #define real x
// #define imag y
using D = long double;
using P = complex<D>;
constexpr D eps = 1e-9;
bool equal(D a, D b) { return abs(a - b) < eps; }
bool equal(P a, P b) { return equal(a.x(), b.x()) and equal(a.y(), b.y()); }
int sign(D a) { return equal(a, 0) ? 0 : a > 0 ? 1 : -1; }
namespace std { bool operator<(P a, P b) { return sign(a.x() - b.x()) == 0 ? sign(a.y() - b.y()) < 0 : a.x() < b.x(); } }
// cross({1, 0}, {0, 1}) = 1
D cross(P a, P b) { return a.x() * b.y() - a.y() * b.x(); }
D dot(P a, P b) { return a.x() * b.x() + a.y() * b.y(); }
D dist(P a, P b) { return abs(a - b); }
int dir(P a, P b, P c) { return sign(cross(b - a, c - b)); }
```

## polygon-gen

includes: `point`, `intersect-lines`, `headers/gen` Generatorka wielokątów niekoniecznie-wypukłych. Zwraca wielokąt o $n$ punktach w zakresie $[-r, r]$, który nie zawiera jakiejkolwiek trójki współliniowych punktów. Ciągnie do $\sim 80$. Dla $n < 3$ zwraca zdegenerowane.

```
vector<P> gen_polygon(int n, int r) {
	vector<P> t;
	while (ssize(t) < n) {
		P p(rd(-r, r), rd(-r, r));
		if ([&]() {
			REP (i, ssize(t))
				REP (j, i)
					if (dir(t[i], t[j], p) == 0)
						return false;
			return find(t.begin(), t.end(), p) == t.end();
		}())
			t.emplace_back(p);
	}
	bool go = true;
	while (go) {
		go = false;
		REP (i, n)
			REP (j, i - 1)
				if ((i + 1) % n != j && ssize(intersect_segments(t[i], t[(i + 1) % n], t[j], t[j + 1]))) {
					swap(t[(i + rd(0, 1)) % n], t[(j + rd(0, 1)) % n]);
					go = true;
				}
	}
	return t;
}
```

## polygon-print

includes: `point` Należy przekierować stdout do pliku i otworzyć go np. w przeglądarce. `m` zwiększa obrazek, `d` zmniejsza rozmiar napisów/wierzchołków.

```
void polygon_print(vector<P> v, int r = 10) {
	int m = 350 / r, d = 50;
	auto ori = v;
	for (auto &p : v)
		p = P((p.x() + r * 1.1) * m, (p.y() + r * 1.1) * m);
	r = int(r * m * 2.5);
	printf("<svg height='%d' width='%d'><rect width='100%%' height='100%%' fill='white' />", r, r);
	int n = ssize(v);
	REP (i, n) {
		printf("<line x1='%Lf' y1='%Lf' x2='%Lf' y2='%Lf' style='stroke:black' />", v[i].x(), v[i].y(), v[(i + 1) % n].x(), v[(i + 1) % n].y());
		printf("<circle cx='%Lf' cy='%Lf' r='%f' fill='red' />", v[i].x(), v[i].y(), r / d / 10.0);
		printf("<text x='%Lf' y='%Lf' font-size='%d' fill='violet'>%d (%.1Lf, %.1Lf)</text>", v[i].x() + 5, v[i].y() - 5, r / d, i + 1, ori[i].x(), ori[i].y());
	}
	printf("</svg>\n");
}
```

## voronoi-diagram

includes: `delaunay-triangulatio`, `convex-hull` $\mathcal{O}(n \log n)$, dla każdego punktu zwraca odpowiadającą mu ścianę będącą otoczką wypukłą. Suma otoczek w całości zawiera kwadrat (-mx, mx) – (mx, mx), ale może zawierać więcej. Współrzędne ścian mogą być kilka rzędów wielkości większe niż te na wejściu. Max abs wartości współrzędnych to 3e8.

```
using Frac = pair<__int128_t, __int128_t>;
D to_d(Frac f) { return D(f.first) / D(f.second); }
Frac create_frac(__int128_t a, __int128_t b) {
	assert(b != 0);
	if(b < 0) a *= -1, b *= -1;
	__int128_t d = __gcd(a, b);
	return {a / d, b / d};
}
using P128 = pair<Frac, Frac>;
LL sq(int x) { return x * LL(x); }
__int128_t dist128(PI p) { return sq(p.first) + sq(p.second); }
pair<Frac, Frac> calc_mid(PI a, PI b, PI c) {
	__int128_t ux = dist128(a) * (b.second - c.second)
		+ dist128(b) * (c.second - a.second)
		+ dist128(c) * (a.second - b.second),
		uy = dist128(a) * (c.first - b.first)
		+ dist128(b) * (a.first - c.first)
		+ dist128(c) * (b.first - a.first),
		d = 2 * (a.first * LL(b.second - c.second)
		+ b.first * LL(c.second - a.second)
		+ c.first * LL(a.second - b.second));
	return {create_frac(ux, d), create_frac(uy, d)};
}
vector<vector<P>> voronoi_faces(vector<PI> in, const int max_xy = int(3e8)) {
	int n = ssize(in);
	map<PI, int> id_of_in;
	REP(i, n)
		id_of_in[in[i]] = i;
	for(int sx : {-1, 1})
		for(int sy : {-1, 1}) {
			int mx = 3 * max_xy + 100;
			in.emplace_back(mx * sx, mx * sy);
		}
	vector<PI> triangles = triangulate(in);
	debug(triangles);
	assert(not triangles.empty());
	int tn = ssize(triangles) / 3;
	vector<P128> mids(tn);
	map<pair<PI, PI>, vector<P128>> on_sides;
	REP(i, tn) {
		array<PI, 3> ps = {triangles[3 * i], triangles[3 * i + 1], triangles[3 * i + 2]};
		mids[i] = calc_mid(ps[0], ps[1], ps[2]);
		REP(j, 3) {
			PI a = ps[j], b = ps[(j + 1) % 3];
			on_sides[pair(min(a, b), max(a, b))].emplace_back(mids[i]);
		}
	}
	vector<vector<P128>> faces128(n);
	for(auto [edge, sides] : on_sides)
		if(ssize(sides) == 2)
			for(PI e : {edge.first, edge.second})
				if(id_of_in.find(e) != id_of_in.end())
					for(auto m : sides)
						faces128[id_of_in[e]].emplace_back(m);
	vector<vector<P>> faces(n);
	REP(i, ssize(faces128)) {
		auto &f = faces128[i];
		sort(f.begin(), f.end());
		f.erase(unique(f.begin(), f.end()), f.end());
		for(auto [x, y] : f)
			faces[i].emplace_back(to_d(x), to_d(y));
		faces[i] = hull(faces[i]);
	}
	return faces;
}
```

# Tekstówki (8)

## aho-corasick

$\mathcal{O}(|s|\alpha)$, Konstruktor tworzy sam korzeń w `node[0]`, `add(s)` dodaje słowo, `convert()` zamienia nieodwracalnie trie w automat Aho-Corasick, `link(x)` zwraca suffix link, `go(x, c)` zwraca następnik x przez literę c, najpierw dodajemy słowa, potem robimy convert(), a na koniec używamy $go$ i $link$.

```
constexpr int alpha = 26;
struct AhoCorasick {
	struct Node {
		array<int, alpha> next, go;
		int p, pch, link = -1;
		bool is_word_end = false;
		Node(int _p = -1, int ch = -1) : p(_p), pch(ch) {
			fill(next.begin(), next.end(), -1);
			fill(go.begin(), go.end(), -1);
		}
	};
	vector<Node> node;
	bool converted = false;
	AhoCorasick() : node(1) {}
	void add(const vector<int> &s) {
		assert(!converted);
		int v = 0;
		for (int c : s) {
			if (node[v].next[c] == -1) {
				node[v].next[c] = ssize(node);
				node.emplace_back(v, c);
			}
			v = node[v].next[c];
		}
		node[v].is_word_end = true;
	}
```

```
  int link(int v) {
    assert(converted);
    return node[v].link;
  }
  int go(int v, int c) {
    assert(converted);
    return node[v].go[c];
  }
  void convert() {
    assert(!converted);
    converted = true;
    deque<int> que = {0};
    while (not que.empty()) {
      int v = que.front();
      que.pop_front();
      if (v == 0 or node[v].p == 0)
        node[v].link = 0;
      else
        node[v].link = go(link(node[v].p), node[v].pch);
      REP (c, alpha) {
        if (node[v].next[c] != -1) {
          node[v].go[c] = node[v].next[c];
          que.emplace_back(node[v].next[c]);
        }
        else
          node[v].go[c] = v == 0 ? 0 : go(link(v), c);
      }
    }
  }
};
```

## eertree

$\mathcal{O}(n\alpha)$ konstrukcja, $\mathcal{O}(n)$ DP oraz odzyskanie. Eertree ma korzeń „pusty" w $0$ oraz „ujemny" w $1$. Z wierzchołka wychodzi krawędź z literą, gdy jego słowo można otoczyć z obu stron tą literą. Funkcja add_letter zwraca wierzchołek odpowiadający za największy palindromiczny suffix aktualnego słowa. Suffix link prowadzi do najdłuższego palindromicznego suffixu słowa wierzchołka. Linki tworzą drzewo z $1$ jako korzeń (który ma syna $0$). Żeby policzyć liczbę wystąpień wierzchołka, po każdym dodaniu litery „wystarczy" dodać $+1$ każdemu na ścieżce od last do korzenia po linkach. palindromic_split_dp zwraca na każdym prefixie (min podział palindromiczny, indeks do odzyskania min podziału, liczbę podziałów). Gdy only_even_lens to może nie istnieć odpowiedź, wtedy .mn == n + 1, .cnt == 0. construct_min_palindromic_split zwraca palindromiczne przedziały pokrywające słowo.

```
constexpr int alpha = 26;
struct Eertree {
  vector<array<int, alpha>> edge;
  array<int, alpha> empty;
  vector<int> str = {-1}, link = {1, 0}, len = {0, -1};
  int last = 0;
  Eertree() {
    empty.fill(0);
    edge.resize(2, empty);
  }
  int find(int v) {
    while(str.end()[-1] != str.end()[-len[v] - 2])
      v = link[v];
    return v;
  }
  int add_letter(int c) {
    str.emplace_back(c);
    last = find(last);
    if(edge[last][c] == 0) {
      edge.emplace_back(empty);
      len.emplace_back(len[last] + 2);
      link.emplace_back(edge[find(link[last])][c]);
      edge[last][c] = ssize(edge) - 1;
    }
    return last = edge[last][c];
  }
};
int add(int a, int b) { return a + b; } // ćDopisa modulo żjeeli trzeba.
struct Dp { int mn, mn_i, cnt; };
Dp operator+(Dp l, Dp r) {
  return {min(l.mn, r.mn), l.mn < r.mn ? l.mn_i : r.mn_i, add(l.cnt, r.cnt)};
}
vector<Dp> palindromic_split_dp(vector<int> str, bool only_even_lens = false) {
  int n = ssize(str);
  Eertree t;
  vector<int> big_link(2), diff(2);
  vector<Dp> series_ans(2), ans(n, {n + 1, -1, 0});
  REP(i, n) {
    int last = t.add_letter(str[i]);
    if(last >= ssize(big_link)) {
      diff.emplace_back(t.len.back() - t.len[t.link.back()]);
      big_link.emplace_back(diff.back() == diff[t.link.back()] ? big_link[t.link.back()] : t.link.back());
      series_ans.emplace_back();
    }
    for(int v = last; t.len[v] > 0; v = big_link[v]) {
      int j = i - t.len[big_link[v]] - diff[v];
      series_ans[v] = j == -1 ? Dp{0, j, 1} : Dp{ans[j].mn, j, ans[j].cnt};
      if(diff[v] == diff[t.link[v]])
        series_ans[v] = series_ans[v] + series_ans[t.link[v]];
      if(i % 2 == 1 or not only_even_lens)
        ans[i] = ans[i] + Dp{series_ans[v].mn + 1, series_ans[v].mn_i, series_ans[v].cnt};
    }
  }
  return ans;
}
vector<pair<int, int>> construct_min_palindromic_split(vector<Dp> ans) {
  if(ans.back().mn == ssize(ans) + 1)
    return {};
  vector<pair<int, int>> split = {{0, ssize(ans) - 1}};
  while(ans[split.back().second].mn_i != -1)
    split.emplace_back(0, ans[split.back().second].mn_i);
  reverse(split.begin(), split.end());
  REP(i, ssize(split) - 1)
    split[i + 1].first = split[i].second + 1;
  return split;
}
```

## hashing

Hashowanie z małą stałą. Można zmienić bazę (jeśli serio trzeba). openssl prime -generate -bits 60 generuje losową liczbę pierwszą o $60$ bitach ($\leq 1.15 \cdot 10^{18}$).

```
struct Hashing {
  vector<LL> ha, pw;
  static constexpr LL mod = (1ll << 61) - 1;
  LL reduce(LL x) { return x >= mod ? x - mod : x; }
  LL mul(LL a, LL b) {
    const auto c = __int128(a) * b;
    return reduce(LL(c & mod) + LL(c >> 61));
  }
  Hashing(const vector<int> &str, const int base = 37) {
    int len = ssize(str);
    ha.resize(len + 1);
    pw.resize(len + 1, 1);
    REP(i, len) {
      ha[i + 1] = reduce(mul(ha[i], base) + str[i] + 1);
      pw[i + 1] = mul(pw[i], base);
    }
  }
  LL operator()(int l, int r) {
    return reduce(ha[r + 1] - mul(ha[l], pw[r - l + 1]) + mod);
  }
};
```

## kmp

$\mathcal{O}(n)$, zachodzi [0, pi[i]) = (i - pi[i], i].
get_kmp({0,1,0,0,1,0,1,0,0,1}) == {0,0,1,1,2,3,2,3,4,5},
get_borders({0,1,0,0,1,0,1,0,0,1}) == {2,5,10}.

```
vector<int> get_kmp(vector<int> str) {
  int len = ssize(str);
  vector<int> ret(len);
  for(int i = 1; i < len; i++) {
    int pos = ret[i - 1];
    while(pos and str[i] != str[pos])
      pos = ret[pos - 1];
    ret[i] = pos + (str[i] == str[pos]);
  }
  return ret;
}
vector<int> get_borders(vector<int> str) {
  vector<int> kmp = get_kmp(str), ret;
  int len = ssize(str);
  while(len) {
    ret.emplace_back(len);
    len = kmp[len - 1];
  }
  return vector<int>(ret.rbegin(), ret.rend());
}
```

## lyndon-min-cyclic-rot

$\mathcal{O}(n)$, wyznaczanie faktoryzacji Lyndona oraz (przy jej pomocy) minimalnego suffixu oraz minimalnego przesunięcia cyklicznego. Ta faktoryzacja to unikalny podział słowa $s$ na $w_1 w_2 \ldots w_k$, że $w_1 \geq w_2 \geq \ldots \geq w_k$ oraz $w_i$ jest ściśle mniejsze od każdego jego suffixu. duval("abacaba") == {{0, 3}, {4, 5}, {6, 6}}, min_suffix("abacab") == "ab", min_cyclic_shift("abacaba") == "aabacab".

```
vector<pair<int, int>> duval(vector<int> s) {
  int n = ssize(s), i = 0;
  vector<pair<int, int>> ret;
  while(i < n) {
    int j = i + 1, k = i;
    while(j < n and s[k] <= s[j]) {
      k = (s[k] < s[j] ? i : k + 1);
      ++j;
    }
    while(i <= k) {
      ret.emplace_back(i, i + j - k - 1);
      i += j - k;
    }
  }
  return ret;
}
vector<int> min_suffix(vector<int> s) {
  return {s.begin() + duval(s).back().first, s.end()};
}
vector<int> min_cyclic_shift(vector<int> s) {
  int n = ssize(s);
  REP(i, n)
    s.emplace_back(s[i]);
  for(auto [l, r] : duval(s))
    if(n <= r) {
      return {s.begin() + l, s.begin() + l + n};
    }
  assert(false);
}
```

## manacher

$\mathcal{O}(n)$, radius[p][i] = $rad$ = największy promień palindromu parzystości $p$ o środku $i$. $L = i - rad + !p, \ R = i + rad$ to palindrom. Dla [abaababaab] daje [003000020], [0100141000].

```
array<vector<int>, 2> manacher(vector<int> &in) {
  int n = ssize(in);
  array<vector<int>, 2> radius = {{vector<int>(n - 1), vector<int>(n)}};
  REP(parity, 2) {
    int z = parity ^ 1, L = 0, R = 0;
    REP(i, n - z) {
      int &rad = radius[parity][i];
      if(i <= R - z)
        rad = min(R - i, radius[parity][L + (R - i - z)]);
      int l = i - rad + z, r = i + rad;
      while(0 <= l - 1 && r + 1 < n && in[l - 1] == in[r + 1])
        ++rad, ++r, --l;
      if(r > R)
        L = l, R = r;
    }
  }
  return radius;
}
```

## pref

$\mathcal{O}(n)$, zwraca tablicę prefixo prefixową $[0, pref[i]) = [i, i + pref[i])$.

```
vector<int> pref(vector<int> str) {
  int n = ssize(str);
  vector<int> ret(n);
  ret[0] = n;
  int i = 1, m = 0;
  while(i < n) {
    while(m + i < n and str[m + i] == str[m])
      m++;
    ret[i++] = m;
    m = max(0, m - 1);
    for(int j = 1; ret[j] < m; m--)
      ret[i++] = ret[j++];
  }
  return ret;
}
```

## squares

includes: pref $\mathcal{O}(n \log n)$, zwraca wszystkie skompresowane trójki $(start\_l, start\_r, len)$ oznaczające, że podsłowa zaczynające się w $[start\_l, start\_r]$ o długości $len$ są kwadratami, jest ich $\mathcal{O}(n \log n)$.

```
vector<tuple<int, int, int>> squares(const vector<int> &s) {
  vector<tuple<int, int, int>> ans;
  vector pos(ssize(s) + 2, -1);
  FOR(mid, 1, ssize(s) - 1) {
    int part = mid & ~(mid - 1), off = mid - part;
    int end = min(mid + part, ssize(s));
    vector a(s.begin() + off, s.begin() + off + part),
      b(s.begin() + mid, s.begin() + end),
      ra(a.rbegin(), a.rend());
    REP(j, 2) {
      auto z1 = pref(ra), bha = b;
      bha.emplace_back(-1);
      for(int x : a) bha.emplace_back(x);
      auto z2 = pref(bha);
      for(auto *v : {&z1, &z2}) {
        v[0][0] = ssize(v[0]);
        v->emplace_back(0);
      }
      REP(c, ssize(a)) {
       int l = ssize(a) - c, x = c - min(l - 1, z1[l]),
         y = c - max(l - z2[ssize(b) + c + 1], j),
         sb = (j ? end - y - l * 2 : off + x),
         se = (j ? end - x - l * 2 + 1 : off + y + 1),
         &p = pos[l];
       if (x > y) continue;
       if (p != -1 && get<1>(ans[p]) + 1 == sb)
         get<1>(ans[p]) = se - 1;
       else
         p = ssize(ans), ans.emplace_back(sb, se - 1, l);
      }
      a = vector(b.rbegin(), b.rend());
      b.swap(ra);
    }
  }
  return ans;
```

```
}
```

## suffix-array-interval includes: suffix-array-short

$\mathcal{O}(t \log n)$, wyznaczanie przedziałów podsłowa w tablicy suffixowej. Zwraca przedział $[l, r]$, gdzie dla każdego $i$ w $[l, r]$, $t$ jest podsłowem $sa.sa[i]$ lub $[-1, -1]$ jeżeli nie ma takiego $i$.

```
pair<int, int> get_substring_sa_range(const vector<int> &s, const vector<int> &sa, const vector<int> &t) {
  auto get_lcp = [&](int i) -> int {
    REP(k, ssize(t))
      if(i + k >= ssize(s) or s[i + k] != t[k])
        return k;
    return ssize(t);
  };
  auto get_side = [&](bool search_left) {
    int l = 0, r = ssize(sa) - 1;
    while(l < r) {
      int m = (l + r + not search_left) / 2, lcp = get_lcp(sa[m]);
      if(lcp == ssize(t))
        (search_left ? r : l) = m;
      else if(sa[m] + lcp >= ssize(s) or s[sa[m] + lcp] < t[lcp])
        l = m + 1;
      else
        r = m - 1;
    }
    return l;
  };
  int l = get_side(true);
  if(get_lcp(sa[l]) != ssize(t))
    return {-1, -1};
  return {l, get_side(false)};
}
```

## suffix-array-long

$\mathcal{O}(n + alpha)$, $sa$ zawiera posortowane suffixy, zawiera pusty suffix, $lcp[i]$ to $lcp$ suffixu $sa[i]$ i $sa[i+1]$, Dla s = aabaaab, sa={7,3,4,0,5,1,6,2},lcp={0,2,3,1,2,0,1}

```
void induced_sort(const vector<int> &vec, int alpha, vector<int> &sa,
    const vector<bool> &sl, const vector<int> &lms_idx) {
  vector<int> l(alpha), r(alpha);
  for (int c : vec) {
    if (c + 1 < alpha)
      ++l[c + 1];
    ++r[c];
  }
  partial_sum(l.begin(), l.end(), l.begin());
  partial_sum(r.begin(), r.end(), r.begin());
  fill(sa.begin(), sa.end(), -1);
  for (int i = ssize(lms_idx) - 1; i >= 0; --i)
    sa[--r[vec[lms_idx[i]]]] = lms_idx[i];
  for (int i : sa)
    if (i >= 1 and sl[i - 1])
      sa[l[vec[i - 1]]++] = i - 1;
  fill(r.begin(), r.end(), 0);
  for (int c : vec)
    ++r[c];
  partial_sum(r.begin(), r.end(), r.begin());
  for (int k = ssize(sa) - 1, i = sa[k]; k >= 1; --k, i = sa[k])
    if (i >= 1 and not sl[i - 1])
      sa[--r[vec[i - 1]]] = i - 1;
}
vector<int> sa_is(const vector<int> &vec, int alpha) {
  const int n = ssize(vec);
  vector<int> sa(n), lms_idx;
  vector<bool> sl(n);
  for (int i = n - 2; i >= 0; --i) {
    sl[i] = vec[i] > vec[i + 1] or (vec[i] == vec[i + 1] and sl[i + 1]);
    if (sl[i] and not sl[i + 1])
      lms_idx.emplace_back(i + 1);
  }
  reverse(lms_idx.begin(), lms_idx.end());
  induced_sort(vec, alpha, sa, sl, lms_idx);
  vector<int> new_lms_idx(ssize(lms_idx)), lms_vec(ssize(lms_idx));
  for (int i = 0, k = 0; i < n; ++i)
    if (not sl[sa[i]] and sa[i] >= 1 and sl[sa[i] - 1])
      new_lms_idx[k++] = sa[i];
  int cur = sa[n - 1] = 0;
  REP (k, ssize(new_lms_idx) - 1) {
    int i = new_lms_idx[k], j = new_lms_idx[k + 1];
    if (vec[i] != vec[j]) {
      sa[j] = ++cur;
      continue;
    }
    bool flag = false;
    for (int a = i + 1, b = j + 1;; ++a, ++b) {
      if (vec[a] != vec[b]) {
        flag = true;
        break;
      }
      if ((not sl[a] and sl[a - 1]) or (not sl[b] and sl[b - 1])) {
        flag = not (not sl[a] and sl[a - 1] and not sl[b] and sl[b - 1]);
        break;
      }
    }
    sa[j] = (flag ? ++cur : cur);
  }
  REP (i, ssize(lms_idx))
    lms_vec[i] = sa[lms_idx[i]];
  if (cur + 1 < ssize(lms_idx)) {
    vector<int> lms_sa = sa_is(lms_vec, cur + 1);
    REP (i, ssize(lms_idx))
      new_lms_idx[i] = lms_idx[lms_sa[i]];
  }
  induced_sort(vec, alpha, sa, sl, new_lms_idx);
  return sa;
}
vector<int> suffix_array(const vector<int> &s, int alpha) {
  vector<int> vec(ssize(s) + 1);
  REP(i, ssize(s))
    vec[i] = s[i] + 1;
  vector<int> ret = sa_is(vec, alpha + 2);
  return ret;
}
vector<int> get_lcp(const vector<int> &s, const vector<int> &sa) {
  int n = ssize(s), k = 0;
  vector<int> lcp(n), rank(n);
  REP (i, n)
    rank[sa[i + 1]] = i;
  for (int i = 0; i < n; i++, k ? k-- : 0) {
    if (rank[i] == n - 1) {
      k = 0;
      continue;
    }
    int j = sa[rank[i] + 2];
    while (i + k < n and j + k < n and s[i + k] == s[j + k])
      k++;
    lcp[rank[i]] = k;
  }
  lcp.pop_back();
  lcp.insert(lcp.begin(), 0);
  return lcp;
}
```

## suffix-array-short

$\mathcal{O}(n \log n)$, zawiera posortowane suffixy, zawiera pusty suffix, $lcp[i]$ to $lcp$ suffixu $sa[i-1]$ i $sa[i]$, Dla s = aabaaab, sa={7,3,4,0,5,1,6,2},lcp={0,0,2,3,1,2,0,1}

```
pair<vector<int>, vector<int>> suffix_array(vector<int> s, int alpha = 26) {
  ++alpha;
  for(int &c : s) ++c;
  s.emplace_back(0);
  int n = ssize(s), k = 0, a, b;
  vector<int> x(s.begin(), s.end());
  vector<int> y(n), ws(max(n, alpha)), rank(n);
  vector<int> sa = y, lcp = y;
  iota(sa.begin(), sa.end(), 0);
  for(int j = 0, p = 0; p < n; j = max(1, j * 2), alpha = p) {
    p = j;
    iota(y.begin(), y.end(), n - j);
    REP(i, n) if(sa[i] >= j)
      y[p++] = sa[i] - j;
    fill(ws.begin(), ws.end(), 0);
    REP(i, n) ws[x[i]]++;
    FOR(i, 1, alpha - 1) ws[i] += ws[i - 1];
    for(int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
    swap(x, y);
    p = 1, x[sa[0]] = 0;
    FOR(i, 1, n - 1) a = sa[i - 1], b = sa[i], x[b] =
      (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
  }
  FOR(i, 1, n - 1) rank[sa[i]] = i;
  for(int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
    for(k && k--, j = sa[rank[i] - 1];
      s[i + k] == s[j + k]; k++);
  lcp.erase(lcp.begin());
  return {sa, lcp};
}
```

## suffix-automaton

$\mathcal{O}(n\alpha)$ (szybsze, ale więcej pamięci) albo $\mathcal{O}(n \log \alpha)$ (mapa), buduje suffix automaton. Wystąpienia wzorca, liczba różnych podsłów, sumaryczna długość wszystkich podsłów, leksykograficznie $k$-te podsłowo, najmniejsze przesunięcie cykliczne, liczba wystąpień podsłowa, pierwsze wystąpienie, najkrótsze niewystępujące podsłowo, longest common substring wielu słów.

```
struct SuffixAutomaton {
  static constexpr int sigma = 26;
  using Node = array<int, sigma>; // map<int, int>
  Node new_node;
  vector<Node> edges;
  vector<int> link = {-1}, length = {0};
  int last = 0;
  SuffixAutomaton() {
    new_node.fill(-1);   // -1 - stan nieistniejacy
    edges = {new_node};  // dodajemy stan startowy, ktory reprezentuje puste slowo
  }
  void add_letter(int c) {
    edges.emplace_back(new_node);
    length.emplace_back(length[last] + 1);
    link.emplace_back(0);
    int r = ssize(edges) - 1, p = last;
    while(p != -1 && edges[p][c] == -1) {
      edges[p][c] = r;
      p = link[p];
    }
    if(p != -1) {
      int q = edges[p][c];
      if(length[p] + 1 == length[q])
        link[r] = q;
      else {
        edges.emplace_back(edges[q]);
        length.emplace_back(length[p] + 1);
        link.emplace_back(link[q]);
        int q_prim = ssize(edges) - 1;
        link[q] = link[r] = q_prim;
        while(p != -1 && edges[p][c] == q) {
          edges[p][c] = q_prim;
          p = link[p];
        }
      }
    }
    last = r;
  }
  bool is_inside(vector<int> &s) {
    int q = 0;
    for(int c : s) {
      if(edges[q][c] == -1)
        return false;
      q = edges[q][c];
    }
    return true;
  }
};
```

## suffix-tree

$\mathcal{O}(n \log n)$ lub $\mathcal{O}(n\alpha)$, Dla słowa abaab# (hash jest aby to zawsze liście były stanami kończącymi) stworzy sons[0]={(#,10),(a,4),(b,8)}, sons[4]={(a,5),(b,6)}, sons[6]={(#,7),(a,2)}, sons[8]={(#,9),(a,3)}, reszta sons'ów pusta, slink[6]=8 i reszta slink'ów $0$ (gdzie slink jest zdefiniowany dla nie-liści jako wierzchołek zawierający ten suffix bez ostatniej literki), up_edge_range[2]=up_edge_range[3]=(2,5), up_edge_range[5]=(3,5) i reszta jednoliterowa. Wierzchołek 1 oraz suffix wierzchołków jest roboczy. Zachodzi up_edge_range[0]=(-1,-1), parent[0]=0, slink[0]=1.

```
struct SuffixTree {
  const int n;
  const vector<int> &_in;
  vector<map<int, int>> sons;
  vector<pair<int, int>> up_edge_range;
  vector<int> parent, slink;
  int tv = 0, tp = 0, ts = 2, la = 0;
  void ukkadd(int c) {
    auto &lr = up_edge_range;
suff:
    if (lr[tv].second < tp) {
      if (sons[tv].find(c) == sons[tv].end()) {
        sons[tv][c] = ts; lr[ts].first = la; parent[ts++] = tv;
        tv = slink[tv]; tp = lr[tv].second + 1; goto suff;
      }
      tv = sons[tv][c]; tp = lr[tv].first;
    }
    if (tp == -1 || c == _in[tp])
      tp++;
    else {
      lr[ts + 1].first = la; parent[ts + 1] = ts;
      lr[ts].first = lr[tv].first; lr[ts].second = tp - 1;
      parent[ts] = parent[tv]; sons[ts][c] = ts + 1; sons[ts][_in[tp]] = tv;
      lr[tv].first = tp; parent[tv] = ts;
      sons[parent[ts]][_in[lr[ts].first]] = ts; ts += 2;
      tv = slink[parent[ts - 2]]; tp = lr[ts - 2].first;
      while (tp <= lr[ts - 2].second) {
        tv = sons[tv][_in[tp]]; tp += lr[tv].second - lr[tv].first + 1;
      }
      if (tp == lr[ts - 2].second + 1)
        slink[ts - 2] = tv;
      else
        slink[ts - 2] = ts;
      tp = lr[tv].second - (tp - lr[ts-2].second) + 2;
      goto suff;
    }
  }
  // Remember to append string with a hash.
  SuffixTree(const vector<int> &in, int alpha)
    : n(ssize(in)), _in(in), sons(2 * n + 1),
    up_edge_range(2 * n + 1, pair(0, n - 1)), parent(2 * n + 1), slink(2 * n + 1) {
    up_edge_range[0] = up_edge_range[1] = {-1, -1};
    slink[0] = 1;
    // When changing map to vector, fill sons exactly here with -1 and replace if in ukkadd with sons[
```

```
      tv][c] == -1.
    REP(ch, alpha)
      sons[1][ch] = 0;
    for(; la < n; ++la)
      ukkadd(in[la]);
  }
};
```

## wildcard-matching

includes: math/ntt $\mathcal{O}(n \log n)$, zwraca tablicę wystąpień wzorca. Alfabet od $0$. Znaki zapytania to $-1$. Mogą być zarówno w tekście jak i we wzrocu. Dla alfabetów większych niż $15$ lepiej użyć bezpieczniejszej wersji.

```
vector<bool> wildcard_matching(vi text, vi pattern) {
  for (int& e : text) ++e;
  for (int& e : pattern) ++e;
  reverse(pattern.begin(), pattern.end());
  int n = ssize(text), m = ssize(pattern);
  int sz = 1 << __lg(2 * n - 1);
  vi a(sz), b(sz), c(sz);
  auto h = [&](auto f, auto g) {
    fill(a.begin(), a.end(), 0);
    fill(b.begin(), b.end(), 0);
    REP(i, n) a[i] = f(text[i]);
    REP(i, m) b[i] = g(pattern[i]);
    ntt(a, sz), ntt(b, sz);
    REP(i, sz) a[i] = mul(a[i], b[i]);
    ntt(a, sz, true);
    REP(i, sz) c[i] = add(c[i], a[i]);
  };
  h([](int x){return powi(x,3);},identity());
  h([](int x){return sub(0, mul(2, mul(x, x)));}, [](
    int x){return mul(x, x);});
  h(identity(),[](int x){return powi(x,3);});
  vector<bool> ret(n - m + 1);
  FOR(i, m, n) ret[i - m] = !c[i - 1];
  return ret;
}
vector<bool> safer_wildcard_matching(vi text, vi
  pattern, int alpha = 26) {
  static mt19937 rng(0); // Can be changed.
  int n = ssize(text), m = ssize(pattern);
  vector ret(n - m + 1, true);
  vi v(alpha), a(n, -1), b(m, -1);
  REP(iters, 2) { // The more the better.
    REP(i, alpha) v[i] = int(rng() % (mod - 1));
    REP(i, n) if (text[i] != -1) a[i] = v[text[i]];
    REP(i, m) if (pattern[i] != -1) b[i] = v[pattern[i
      ]];
    auto h = wildcard_matching(a, b);
    REP(i, n - m + 1) ret[i] = min(ret[i], h[i]);
  }
  return ret;
}
```

# Optymalizacje (9)

## cantor

Szybki hash 2 intów.

```
int cantor(int a, int b) {
  return (a + b + 1) * (a + b) / 2 + b;
}
```

## divide-and-conquer-dp

$\mathcal{O}(nm \log m)$, dla funkcji $cost(k, j)$ wylicza $dp(i, j) = min_{0 \leq k \leq j}\ dp(i-1, k-1) + cost(k, j)$. Działa tylko wtedy, gdy $opt(i, j-1) \leq opt(i, j)$, a jest to zawsze spełnione, gdy $cost(b, c) \leq cost(a, d)$ oraz $cost(a, c) + cost(b, d) \leq cost(a, d) + cost(b, c)$ dla $a \leq b \leq c \leq d$.

```
vector<LL> divide_and_conquer_optimization(int n, int
  m, function<LL(int,int)> cost) {
  vector<LL> dp_before(m);
  auto dp_cur = dp_before;
  REP(i, m)
    dp_before[i] = cost(0, i);
  function<void(int,int,int,int)> compute = [&](int l,
    int r, int optl, int optr) {
    if (l > r)
      return;
    int mid = (l + r) / 2, opt;
    pair<LL, int> best = {numeric_limits<LL>::max(),
      -1};
    FOR(k, optl, min(mid, optr))
      best = min(best, {(k ? dp_before[k - 1] : 0) +
        cost(k, mid), k});
    tie(dp_cur[mid], opt) = best;
    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
  };
  REP(i, n) {
    compute(0, m - 1, 0, m - 1);
    swap(dp_before, dp_cur);
  }
  return dp_before;
}
```

## dp-1d1d

$\mathcal{O}(n \log n)$, $n > 0$ długość paska, cost(i, j) koszt odcinka $[i, j]$ Dla $a \leq b \leq c \leq d$ cost ma spełniać $cost(a, c) + cost(b, d) \leq cost(a, d) + cost(b, c)$. Dzieli pasek $[0, n)$ na odcinki $[0, cuts[0]], \ldots, (cuts[i-1], cuts[i]]$, gdzie cuts.back() == n - 1, aby sumaryczny koszt wszystkich odcinków był minimalny. cuts to prawe końce tych odcinków. Zwraca (opt_cost, cuts). Aby maksymalizować koszt zamienić nierówności tam, gdzie wskazane. Aby uzyskać $\mathcal{O}(n)$, należy przepisać overtake w oparciu o dodatkowe założenia, aby chodził w $\mathcal{O}(1)$.

```
pair<LL, vector<int>> dp_1d1d(int n, function<LL (int,
  int)> cost) {
  vector<pair<LL, int>> dp(n);
  vector<int> lf(n + 2), rg(n + 2), dead(n);
  vector<vector<int>> events(n + 1);
  int beg = n, end = n + 1;
  rg[beg] = end; lf[end] = beg;
  auto score = [&](int i, int j) {
    return dp[j].first + cost(j + 1, i);
  };
  auto overtake = [&](int a, int b, int mn) {
    int bp = mn - 1, bk = n;
    while (bk - bp > 1) {
      int bs = (bp + bk) / 2;
      if (score(bs, a) <= score(bs, b)) // tu >=
        bk = bs;
      else
        bp = bs;
    }
    return bk;
  };
  auto add = [&](int i, int mn) {
    if (lf[i] == beg)
      return;
    events[overtake(i, lf[i], mn)].emplace_back(i);
  };
  REP (i, n) {
    dp[i] = {cost(0, i), -1};
    REP (j, ssize(events[i])) {
      int x = events[i][j];
      if (dead[x])
        continue;
      dead[lf[x]] = 1; lf[x] = lf[lf[x]];
      rg[lf[x]] = x; add(x, i);
    }
    if (rg[beg] != end)
      dp[i] = min(dp[i], {score(i, rg[beg]), rg[beg]})
        ; // tu max
    lf[i] = lf[end]; rg[i] = end;
    rg[lf[i]] = i; lf[rg[i]] = i;
    add(i, i + 1);
  }
  vector<int> cuts;
  for (int p = n - 1; p != -1; p = dp[p].second)
    cuts.emplace_back(p);
  reverse(cuts.begin(), cuts.end());
  return pair(dp[n - 1].first, cuts);
}
```

## fio

FIO do wpychania kolanem. Nie należy wtedy używać cin/cout

```
#ifdef ONLINE_JUDGE
// write this when judge is on Windows
inline int getchar_unlocked() { return _getchar_nolock
  (); }
inline void putchar_unlocked(char c) { _putchar_nolock
  (c); }
#endif
int fastin() {
  int n = 0, c = getchar_unlocked();
  while(isspace(c))
    c = getchar_unlocked();
  while(isdigit(c)) {
    n = 10 * n + (c - '0');
    c = getchar_unlocked();
  }
  return n;
}
int fastin_negative() {
  int n = 0, negative = false, c = getchar_unlocked();
  while(isspace(c))
    c = getchar_unlocked();
  if(c == '-') {
    negative = true;
    c = getchar_unlocked();
  }
  while(isdigit(c)) {
    n = 10 * n + (c - '0');
    c = getchar_unlocked();
  }
  return negative ? -n : n;
}
double fastin_double() {
  double x = 0, t = 1;
  int negative = false, c = getchar_unlocked();
  while(isspace(c))
    c = getchar_unlocked();
  if (c == '-') {
    negative = true;
    c = getchar_unlocked();
  }
  while (isdigit(c)) {
    x = x * 10 + (c - '0');
    c = getchar_unlocked();
  }
  if (c == '.') {
    c = getchar_unlocked();
    while (isdigit(c)) {
      t /= 10;
      x = x + t * (c - '0');
      c = getchar_unlocked();
    }
  }
  return negative ? -x : x;
}
void fastout(int x) {
  if(x == 0) {
    putchar_unlocked('0');
    putchar_unlocked(' ');
    return;
  }
  if(x < 0) {
    putchar_unlocked('-');
    x *= -1;
  }
  static char t[10];
  int i = 0;
  while(x) {
    t[i++] = char('0' + (x % 10));
    x /= 10;
  }
  while(--i >= 0)
    putchar_unlocked(t[i]);
  putchar_unlocked(' ');
}
void nl() { putchar_unlocked('\n'); }
```

## knuth

$\mathcal{O}(n^2)$, dla tablicy $cost(i, j)$ wylicza $dp(i, j) = min_{i \leq k < j}\ dp(i, k) + dp(k+1, j) + cost(i, j)$. Działa tylko wtedy, gdy $opt(i, j-1) \leq opt(i, j) \leq opt(i+1, j)$, a jest to zawsze spełnione, gdy $cost(b, c) \leq cost(a, d)$ oraz $cost(a, c) + cost(b, d) \leq cost(a, d) + cost(b, c)$ dla $a \leq b \leq c \leq d$.

```
LL knuth_optimization(vector<vector<LL>> cost) {
  int n = ssize(cost);
  vector dp(n, vector<LL>(n, numeric_limits<LL>::max()
    ));
  vector opt(n, vector<int>(n));
  REP(i, n) {
    opt[i][i] = i;
    dp[i][i] = cost[i][i];
  }
  for(int i = n - 2; i >= 0; --i)
    FOR(j, i + 1, n - 1)
      FOR(k, opt[i][j - 1], min(j - 1, opt[i + 1][j]))
        if(dp[i][j] >= dp[i][k] + dp[k + 1][j] + cost[
          i][j]) {
          opt[i][j] = k;
          dp[i][j] = dp[i][k] + dp[k + 1][j] + cost[i
            ][j];
        }
  return dp[0][n - 1];
}
```

## linear-knapsack

$\mathcal{O}(n \cdot \max(w_i))$ zamiast typowego $\mathcal{O}(n \cdot \sum(w_i))$, pamięć $\mathcal{O}(n + \max(w_i))$, plecak zwracający największą otrzymywalną sumę ciężarów <= bound.

```
LL knapsack(vector<int> w, LL bound) {
  erase_if(w, [=](int x){ return x > bound; });
  {
    LL sum = accumulate(w.begin(), w.end(), 0LL);
    if(sum <= bound)
      return sum;
  }
  LL w_init = 0;
  int b;
  for(b = 0; w_init + w[b] <= bound; ++b)
    w_init += w[b];
  int W = *max_element(w.begin(), w.end());
  vector<int> prev_s(2 * W, -1);
  auto get = [&](vector<int> &v, LL i) -> int& {
    return v[i - (bound  - W + 1)];
  };
  for(LL mu = bound + 1; mu <= bound + W; ++mu)
    get(prev_s, mu) = 0;
  get(prev_s, w_init) = b;
  FOR(t, b, ssize(w) - 1) {
    vector curr_s = prev_s;
    for(LL mu = bound - W + 1; mu <= bound; ++mu)
      get(curr_s, mu + w[t]) = max(get(curr_s, mu + w[
        t]), get(prev_s, mu));
    for(LL mu = bound + w[t]; mu >= bound + 1; --mu)
      for(int j = get(curr_s, mu) - 1; j >= get(prev_s
        , mu); --j)
        get(curr_s, mu - w[j]) = max(get(curr_s, mu -
          w[j]), j);
    swap(prev_s, curr_s);
  }
  for(LL mu = bound; mu >= 0; --mu)
    if(get(prev_s, mu) != -1)
      return mu;
  assert(false);
}
```

## matroid-intersection

$\mathcal{O}(r^2 \cdot (init + n \cdot add))$, where r is max independent set. Find largest subset S of [n] such that S is independent in both matroid A and B, given by their oracles, see example implementations below. Returns vector V such that V[i] = 1 iff i-th element is included in found set; Zabrane z https://github.com/KacperTopolski/kactl/tree/main Zmienne w matroidach ustawiamy ręcznie aby "zainicjalizować" tylko jeśli mają komentarz co znaczą. W przeciwnym wypadku intersectMatroids zrobi robotę wołając init.

```
template<class T, class U>
vector<bool> intersectMatroids(T& A, U& B, int n) {
  vector<bool> ans(n);
  bool ok = 1;
// NOTE: for weighted matroid intersection find
// shortest augmenting paths first by weight change,
// then by length using Bellman-Ford,
  // Speedup trick (only for unweighted):
  A.init(ans); B.init(ans);
  REP(i, n)
    if (A.canAdd(i) && B.canAdd(i))
      ans[i] = 1, A.init(ans), B.init(ans);
  //End of speedup
  while (ok) {
    vector<vector<int>> G(n);
    vector<bool> good(n);
    queue<int> que;
    vector<int> prev(n, -1);
    A.init(ans); B.init(ans); ok = 0;
    REP(i, n) if (!ans[i]) {
      if (A.canAdd(i)) que.emplace(i), prev[i]=-2;
      good[i] = B.canAdd(i);
    }
    REP(i, n) if (ans[i]) {
      ans[i] = 0;
      A.init(ans); B.init(ans);
      REP(j, n) if (i != j && !ans[j]) {
        if (A.canAdd(j)) G[i].emplace_back(j); //-cost
          [j]
        if (B.canAdd(j)) G[j].emplace_back(i); // cost
          [i]
      }
      ans[i] = 1;
    }
    while (!que.empty()) {
      int i = que.front();
      que.pop();
      if (good[i]) { // best found (unweighted =
        shortest path)
        ans[i] = 1;
        while (prev[i] >= 0) { // alternate matching
          ans[i = prev[i]] = 0;
          ans[i = prev[i]] = 1;
        }
        ok = 1; break;
      }
      for(auto j: G[i]) if (prev[j] == -1)
        que.emplace(j), prev[j] = i;
    }
  }
  return ans;
}
// Matroid where each element has color
// and set is independent iff for each color c
// #{elements of color c} <= maxAllowed[c].
struct LimOracle {
  vector<int> color; // color[i] = color of i-th
    element
  vector<int> maxAllowed; // Limits for colors
  vector<int> tmp;
  // Init oracle for independent set S; O(n)
  void init(vector<bool>& S) {
    tmp = maxAllowed;
    REP(i, ssize(S)) tmp[color[i]] -= S[i];
  }
  // Check if S+{k} is independent; time: O(1)
  bool canAdd(int k) { return tmp[color[k]] > 0;}
};
// Graphic matroid - each element is edge,
// set is independent iff subgraph is acyclic.
struct GraphOracle {
  vector<pair<int, int>> elems; // Ground set: graph
    edges
  int n; // Number of vertices, indexed [0;n-1]
  vector<int> par;
  int find(int i) {
    return par[i] == -1 ? i : par[i] = find(par[i]);
  }
  // Init oracle for independent set S; ~O(n)
  void init(vector<bool>& S) {
    par.assign(n, -1);
    REP(i, ssize(S)) if (S[i])
      par[find(elems[i].first)] = find(elems[i].second
        );
  }
  // Check if S+{k} is independent; time: ~O(1)
  bool canAdd(int k) {
    return find(elems[k].first) != find(elems[k].
      second);
  }
};
// Co-graphic matroid - each element is edge,
// set is independent iff after removing edges
// from graph number of connected components
// doesn't change.
struct CographOracle {
  vector<pair<int, int>> elems; // Ground set: graph
    edges
  int n; // Number of vertices, indexed [0;n-1]
  vector<vector<int>> G;
  vector<int> pre, low;
  int cnt;
  int dfs(int v, int p) {
    pre[v] = low[v] = ++cnt;
    for(auto e: G[v]) if (e != p)
      low[v] = min(low[v], pre[e] ?: dfs(e,v));
    return low[v];
  }
  // Init oracle for independent set S; O(n)
  void init(vector<bool>& S) {
    G.assign(n, {});
    pre.assign(n, 0);
    low.resize(n);
    cnt = 0;
    REP(i,ssize(S)) if (!S[i]) {
      pair<int, int> e = elems[i];
      G[e.first].emplace_back(e.second);
      G[e.second].emplace_back(e.first);
    }
    REP(v, n) if (!pre[v]) dfs(v, -1);
  }
  // Check if S+{k} is independent; time: O(1)
  bool canAdd(int k) {
    pair<int, int> e = elems[k];
    return max(pre[e.first], pre[e.second]) != max(low
      [e.first], low[e.second]);
  }
};
// Matroid equivalent to linear space with XOR
struct XorOracle {
  vector<LL> elems; // Ground set: numbers
  vector<LL> base;
  // Init for independent set S; O(n+r^2)
  void init(vector<bool>& S) {
    base.assign(63, 0);
    REP(i, ssize(S)) if (S[i]) {
      LL e = elems[i];
      REP(j, ssize(base)) if ((e >> j) & 1) {
        if (!base[j]) {
          base[j] = e;
          break;
        }
        e ^= base[j];
      }
    }
  }
  // Check if S+{k} is independent; time: O(r)
  bool canAdd(int k) {
    LL e = elems[k];
    REP(i, ssize(base)) if ((e >> i) & 1) {
      if (!base[i]) return 1;
      e ^= base[i];
    }
    return 0;
  }
};
```

## pragmy

Pragmy do wypychania kolanem

```
#pragma GCC optimize("Ofast")
#pragma GCC target("avx,avx2")
```

## random

Szybsze rand.

```
uint32_t xorshf96() {
  static uint32_t x = 123456789, y = 362436069, z =
    521288629;
  uint32_t t;
  x ^= x << 16;
  x ^= x >> 5;
  x ^= x << 1;
  t = x;
  x = y;
  y = z;
  z = t ^ x ^ y;
  return z;
}
```

## sos-dp

$\mathcal{O}(n2^n)$, dla tablicy $A[i]$ oblicza tablicę $F[mask] = \sum_{i \subseteq mask} A[i]$, czyli sumę po podmaskach. Może też liczyć sumę po nadmaskach. `sos_dp(2, {4, 3, 7, 2})` zwraca `{4, 7, 11, 16}`, `sos_dp(2, {4, 3, 7, 2}, true)` zwraca `{16, 5, 9, 2}`.

```
vector<LL> sos_dp(int n, vector<LL> A, bool nad =
  false) {
  int N = (1 << n);
  if (nad) REP(i, N / 2) swap(A[i], A[(N - 1) ^ i]);
  auto F = A;
  REP(i, n)
    REP(mask, N)
      if ((mask >> i) & 1)
        F[mask] += F[mask ^ (1 << i)];
  if (nad) REP(i, N / 2) swap(F[i], F[(N - 1) ^ i]);
  return F;
}
```