Contest (1)

sol.cpp

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

#ifdef LOCAL
auto& operator<<(auto&, pair<auto, auto>);
auto operator<<(auto& o, auto x) -> decltype(x.end(), o) {
  o << '{';
  for (int i = 0; auto y : x) o << ", " + !i++ * 2 << y;
  return o << '}';
}
auto& operator<<(auto& o, pair<auto, auto> x) {
  return o << '(' << x.first << ", " << x.second << ')';
}
void __print(auto... x) { ((cerr << ' ' << x), ...) << endl; }
#define debug(x...) cerr << "[" #x "]:", __print(x)
#else
#define debug(...) 2137
#endif

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
}
```

.vimrc

```
set nu et ts=2 sw=2
filetype indent on
syntax on
colorscheme habamax
hi MatchParen ctermfg=66 ctermbg=234 cterm=underline
nnoremap ; :
nnoremap : ;
inoremap {<cr> {<cr>}<esc>O <bs>
```

Makefile

```
CXXFLAGS=-std=c++20 -Wall -Wextra -Wshadow

sol: sol.cpp
  g++ $(CXXFLAGS) -fsanitize=address,undefined -g -DLOCAL \
      sol.cpp -o sol

fast: sol.cpp
  g++ $(CXXFLAGS) -O2 sol.cpp -o fast
```

test.sh

```
#!/bin/bash

for((i=1;i>0;i++)) do
  echo "$i"
  echo "$i" | ./gen > int
  diff -w <(./sol < int) <(./slow < int) || break
done
```

Wzory (2)

2.1 Kombinatoryka

2.1.1 Lemat Burnside'a

Niech G oznacza grupę symetrii reprezentacji, S zbiór obiektów, a $I(\pi)$ ilość punktów stałych π. Wtedy

$$|S| = \frac{1}{|G|}\sum_{\pi \in G} I(\pi).$$

2.2 Grafy

2.2.1 Twierdzenie Königa

W grafie dwudzielnym zachodzi

- $\mathrm{nk} = \mathrm{pw}$
- $\mathrm{nk} + \mathrm{pk} = n$
- $\mathrm{pw} + \mathrm{nw} = n$

oraz

- pw to zbiór wierzchołków na brzegu min-cut
- nw to dopełnienie pw
- pk to nk z dodanymi pojedynczymi krawędziami każdego nieskojarzonego wierzchołka

2.2.2 Twierdzenie Erdősa-Gallaia

Ciąg stopni $d_1 \geq \ldots \geq d_n$ opisuje prosty graf wtw gdy $\sum d_i$ jest parzysta oraz dla każdego $1 \leq k \leq n$ zachodzi

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k).$$

2.2.3 Twierdzenie Gale'a-Rysera

Ciągi stopni $a_1 \geq \ldots \geq a_n$ oraz $b_1, \ldots, b_n$ opisują prosty graf dwudzielny wtw gdy $\sum a_i = \sum b_i$ oraz dla każdego $1 \leq k \leq n$ zachodzi

$$\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{n} \min(b_i, k).$$

2.2.4 Przepływy z wymaganiami

Szukamy przepływu $\leq F$ takiego, że $f_i \geq d_i$ dla każdej krawędzi. Tworzymy nowe źródło s' i ujście t'. Następnie dodajemy krawędzie

- (u_i, t', d_i), (s', v_i, d_i), $(u_i, v_i, c_i - d_i)$ zamiast (u_i, v_i, c_i, d_i)
- (t, s, F)

Przepływ spełnia wymagania jeżeli maksymalnie wypełnia wszystkie krawędzie s'.

2.3 Analiza

2.3.1 Mnożniki Lagrange'a

Jeżeli optymalizujemy $f(x_1, \ldots, x_n)$ przy ograniczeniach typu $g_k(x_1, \ldots, x_n) = 0$ to $x_1, \ldots, x_n$ jest ekstremum lokalnym tylko jeżeli gradient $\nabla f(x_1, \ldots, x_n)$ jest kombinacją liniową gradientów $\nabla g_k(x_1, \ldots, x_n)$.

Struktury danych (3)

WaveletTree.h

Opis: st – początek, ed – koniec, sst – posortowany początek. Niszczy wartości w przedziale [st, ed).

Czas: $\mathcal{O}((n+q)\log n)$

```
struct node {
  int lo, hi;
  vector<int> s;
  node *l = 0, *r = 0;
  node(auto st, auto ed, auto sst) {
    int n = ed - st;
    lo = sst[0];
    hi = sst[n - 1] + 1;
    if (lo + 1 < hi) {
      int mid = sst[n / 2];
      if (mid == sst[0]) mid = *upper_bound(sst, sst + n, mid);
      s.reserve(n + 1);
      s.push_back(0);
      for (auto it = st; it != ed; it++) {
        s.push_back(s.back() + (*it < mid));
      }
      auto k = stable_partition(st, ed, [&](int x) {
        return x < mid;
      });
      auto sm = lower_bound(sst, sst + n, mid);
      if (k != st) l = new node(st, k, sst);
      if (k != ed) r = new node(k, ed, sm);
    }
  }
```

```
  int kth(int a, int b, int k) {
    if (lo + 1 == hi) return lo;
    int x = s[a], y = s[b];
    return k < y - x ? l->kth(x, y, k)
                    : r->kth(a - x, b - y, k - (y - x));
  }
  int count(int a, int b, int k) {
    if (lo >= k) return 0;
    if (hi <= k) return b - a;
    int x = s[a], y = s[b];
    return (l ? l->count(x, y, k) : 0) +
           (r ? r->count(a - x, b - y, k) : 0);
  }
  int freq(int a, int b, int k) {
    if (k < lo || hi <= k) return 0;
    if (lo + 1 == hi) return b - a;
    int x = s[a], y = s[b];
    return (l ? l->freq(x, y, k) : 0) +
           (r ? r->freq(a - x, b - y, k) : 0);
  }
};
```

OrderedSet.h

Opis: `s.find_by_order(k)` i `s.order_of_key(k)`.
Czas: $\mathcal{O}(\log n)$

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
                         tree_order_statistics_node_update>;
```

Treap.h

Opis: Implicit treap ze spychaniem. Wystarczy zmienić `push`, `pull` i dane.
Czas: $\mathcal{O}(\log n)$

```
mt19937 rng(2137);
struct node {
  int pr, sz = 1, val;
  node *l = 0, *r = 0;
  ll sum = 0;
  bool rev;
  node(int x = 0) {
    pr = rng();
    sum = val = x;
  }
  void pull() {
    sz = 1 + size(l) + size(r);
    sum = val + (l ? l->sum : 0) + (r ? r->sum : 0);
  }
  void push() {
    if (rev) {
      swap(l, r);
      if (l) l->rev ^= 1;
      if (r) r->rev ^= 1;
      rev = false;
    }
  }
  friend int size(node* a) {
    return a ? a->sz : 0;
  }
  friend pair<node*, node*> split(node* a, int k) {
    if (!a) return {0, 0};
    a->push();
    if (k <= size(a->l)) {
      auto [la, lb] = split(a->l, k);
      a->l = lb;
      a->pull();
      return {la, a};
    } else {
      auto [ra, rb] = split(a->r, k - size(a->l) - 1);
      a->r = ra;
      a->pull();
      return {a, rb};
    }
  }
  friend node* merge(node* a, node* b) {
    if (!a || !b) return a ?: b;
    a->push(); b->push();
    if (a->pr > b->pr) {
      a->r = merge(a->r, b);
      a->pull();
      return a;
    } else {
      b->l = merge(a, b->l);
      b->pull();
      return b;
    }
  }
};
```

LineSet.h

Opis: Znajduje maksimum funkcji liniowych online. Dla doubli `div(a,b) = a/b` oraz `INF = 1/.0`.
Czas: $\mathcal{O}(\log n)$

```
struct line {
  mutable ll a, b, p;
  bool operator<(const line& o) const { return a < o.a; }
  bool operator<(ll x) const { return p < x; }
};
struct line_set : multiset<line, less<>> {
  static const ll INF = LLONG_MAX;
  ll div(ll a, ll b) {
    return a / b - ((a ^ b) < 0 && a % b);
  }
  bool inter(iterator x, iterator y) {
    if (y == end()) return x->p = INF, 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 (inter(y, z)) z = erase(z);
    if (x != begin() && inter(--x, y)) inter(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p) {
      inter(x, erase(y));
    }
  }
  ll get(ll x) {
    line l = *lower_bound(x);
    return l.a * x + l.b;
  }
};
```

Grafy (4)

4.1 Przepływy

Dinic.h

Opis: Znajduje największy przepływ. Na niektórych grafach może być szybszy bez skalowania.
Czas: $\mathcal{O}(nm \log U)$

```
struct dinic {
  struct edge {
    int to, rev;
    ll cap;
  };
  int n;
  vector<vector<edge>> adj;
  vector<int> q, lvl, it;
  dinic(int _n) : n(_n), adj(n), q(n) {}
  void add_edge(int u, int v, ll cap, ll rcap = 0) {
    int i = ssize(adj[u]), j = ssize(adj[v]);
    adj[u].push_back({v, j + (u == v), cap});
    adj[v].push_back({u, i, rcap});
  }
  ll dfs(int u, int t, ll cap) {
    if (u == t || !cap) return cap;
    for (int& i = it[u]; i < ssize(adj[u]); i++) {
      edge& e = adj[u][i];
      if (lvl[e.to] == lvl[u] + 1) {
        if (ll d = dfs(e.to, t, min(cap, e.cap))) {
          e.cap -= d, adj[e.to][e.rev].cap += d;
          return d;
        }
      }
    }
    return 0;
  }
  ll flow(int s, int t, ll cap) {
    ll f = 0; q[0] = s;
    for (int b = 62; b >= 0; b--) do {
      lvl.assign(n, 0); it.assign(n, 0);
      int l = 0, r = lvl[s] = 1;
      while (l < r && !lvl[t]) {
        int u = q[l++];
        for (edge e : adj[u]) {
          if (!lvl[e.to] && e.cap >> b) {
            lvl[e.to] = lvl[u] + 1, q[r++] = e.to;
          }
        }
      }
      while (ll d = dfs(s, t, cap)) f += d, cap -= d;
    } while (lvl[t]);
    return f;
  }
};
```

MCMF.h

Opis: Znajduje największy przepływ o najmniejszym koszcie. Jeżeli są ujemne krawędzie to przed puszczeniem `flow` w `pi` trzeba policzyć najkrótsze ścieżki z `s`.
Czas: $\mathcal{O}(Fm \log n)$

```
#include <ext/pb_ds/priority_queue.hpp>
const ll INF64 = 2e18;
struct MCMF {
  struct edge {
    int from, to, rev;
    ll cap, cost;
  };
  int n;
  vector<vector<edge>> adj;
  vector<ll> dst, pi;
  __gnu_pbds::priority_queue<pair<ll, int>> q;
  vector<decltype(q)::point_iterator> it;
  vector<edge*> p;
```

```
  MCMF(int _n) {
    n = _n;
    adj.resize(n);
    pi.resize(n);
    p.resize(n);
  }
  void add_edge(int u, int v, ll cap, ll cost, ll rcap = 0) {
    int i = ssize(adj[u]), j = ssize(adj[v]);
    adj[u].push_back({u, v, j + (u == v), cap, cost});
    adj[v].push_back({v, u, i, rcap, -cost});
  }
  bool path(int s, int t) {
    dst.assign(n, INF64); it.assign(n, q.end());
    q.push({dst[s] = 0, s});
    while (!q.empty()) {
      int u = q.top().second; q.pop();
      for (edge& e : adj[u]) {
        ll d = dst[u] + pi[u] + e.cost - pi[e.to];
        if (e.cap && d < dst[e.to]) {
          dst[e.to] = d, p[e.to] = &e;
          if (it[e.to] == q.end()) {
            it[e.to] = q.push({-dst[e.to], e.to});
          } else {
            q.modify(it[e.to], {-dst[e.to], e.to});
          }
        }
      }
    }
    for (int i = 0; i < n; i++) {
      pi[i] = min(pi[i] + dst[i], INF64);
    }
    return pi[t] != INF64;
  }
  pair<ll, ll> flow(int s, int t, ll cap) {
    ll f = 0, c = 0;
    while (f < cap && path(s, t)) {
      ll d = cap - f;
      for (edge* e = p[t]; e; e = p[e->from]) d = min(d, e->cap
          );
      for (edge* e = p[t]; e; e = p[e->from]) {
        e->cap -= d, adj[e->to][e->rev].cap += d;
      }
      f += d, c += d * pi[t];
    }
    return {f, c};
  }
};
```

4.2 Skojarzenia

Matching.h

Opis: Dinic uproszczony dla grafów dwudzielnych.
Czas: $\mathcal{O}(m\sqrt{n})$

```
struct matching {
  int n, m;
  vector<vector<int>> adj;
  vector<int> pb, pa;
  vector<int> lvl, it;
  matching(int _n, int _m) {
    n = _n;
    m = _m;
    adj.resize(n);
    pb.resize(n, -1);
    pa.resize(m, -1);
    it.resize(n);
  }
  void add_edge(int u, int v) {
    adj[u].push_back(v);
  }
  bool bfs() {
    bool res = false;
    lvl.assign(n, -1);
    queue<int> q;
    for (int i = 0; i < n; i++) {
      if (pb[i] == -1) {
        q.push(i);
        lvl[i] = 0;
      }
    }
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      for (int j : adj[u]) {
        if (pa[j] == -1) {
          res = true;
        } else if (lvl[pa[j]] == -1) {
          lvl[pa[j]] = lvl[u] + 1;
          q.push(pa[j]);
        }
      }
    }
    return res;
  }
  bool dfs(int u) {
    for (auto& i = it[u]; i < ssize(adj[u]); i++) {
      int v = adj[u][i];
      if (pa[v] == -1 ||
          (lvl[pa[v]] == lvl[u] + 1 && dfs(pa[v]))) {
        pb[u] = v;
        pa[v] = u;
        return true;
      }
    }
    return false;
  }
  int match() {
    int ans = 0;
    while (bfs()) {
      it.assign(n, 0);
      for (int i = 0; i < n; i++) {
        if (pb[i] == -1 && dfs(i)) ans++;
      }
    }
    return ans;
  }
};
```

4.3 Grafy skierowane

SCC.h

Opis: Spójne są posortowane topologicznie.
Czas: $\mathcal{O}(n+m)$

```
struct SCC {
  int n, cnt = 0;
  vector<vector<int>> adj;
  vector<int> p, low, in;
  stack<int> st;
  int tour = 0;
  SCC(int _n) {
    n = _n;
    adj.resize(n);
    p.resize(n, -1);
    low.resize(n);
    in.resize(n, -1);
  }
  void add_edge(int u, int v) {
    adj[u].push_back(v);
  }
  void dfs(int u) {
    low[u] = in[u] = tour++;
    st.push(u);
    for (int v : adj[u]) {
      if (in[v] == -1) {
        dfs(v);
        low[u] = min(low[u], low[v]);
      } else {
        low[u] = min(low[u], in[v]);
      }
    }
    if (low[u] == in[u]) {
      int v = -1;
      do {
        v = st.top();
        st.pop();
        in[v] = n;
        p[v] = cnt;
      } while (v != u);
      cnt++;
    }
  }
  void build() {
    for (int i = 0; i < n; ++i) {
      if (in[i] == -1) dfs(i);
    }
    for (int i = 0; i < n; i++) p[i] = cnt - 1 - p[i];
  }
};
```

4.4 Drzewa

RerootDP.h

Opis: Oblicza DP z każdego korzenia.
Czas: $\mathcal{O}(n \log n)$

```
template<typename T>
vector<T> reroot(const auto& adj) {
  int n = ssize(adj);
  vector<int> q(n), p(n, -1);
  for (int i = 0, j = 1; i < n; i++) {
    int u = q[i];
    for (int v : adj[u]) if (v != p[u]) p[v] = u, q[j++] = v;
  }
  vector<T> dp(n), ans(n), rdp(n);
  for (int i = n - 1; i >= 0; i--) {
    int u = q[i], pi = -1;
    dp[u].init(u);
    for (int j = 0; j < ssize(adj[u]); j++) {
      if (adj[u][j] != p[u]) dp[u].merge(u, j, dp[adj[u][j]]);
      else pi = j;
    }
    ans[u] = dp[u];
    dp[u].push(u, pi);
  }
  for (int u : q) {
    auto sum = [&](T& s, int l, int r) {
      for (int i = l; i < r; i++) {
        int v = adj[u][i];
        s.merge(u, i, v != p[u] ? dp[v] : rdp[u]);
      }
```

```
    };
    auto rec = [&](auto self, int l, int r, const T& s) {
      if (l + 1 == r) {
        if (adj[u][l] != p[u]) rdp[adj[u][l]] = s;
        return;
      }
      int m = (l + r) / 2;
      T ss = s; sum(ss, l, m); self(self, m, r, ss);
      ss = s; sum(ss, m, r); self(self, l, m, ss);
    };
    T s; s.init(u);
    if (n > 1) rec(rec, 0, ssize(adj[u]), s);
    for (int i = 0; i < ssize(adj[u]); i++) {
      if (adj[u][i] != p[u]) rdp[adj[u][i]].push(u, i);
      else ans[u].merge(u, i, rdp[u]);
    }
    ans[u].push(u, -1);
  }
  return ans;
}
struct DP {
  void init(int u) {}
  void merge(int u, int i, const DP& s) {}
  void push(int u, int i) {}
};
```

Matma (5)

5.1 Arytmetyka modularna

ModInt.h

```
template<int M, int R>
struct mod {
  static const int MOD = M, ROOT = R;
  int x;
  mod(ll y = 0) : x(y % M) { x += (x < 0) * M; }
  mod operator+=(const mod& o) {
    if ((x += o.x) >= M) x -= M;
    return *this;
  }
  mod operator-=(const mod& o) {
    if ((x -= o.x) < 0) x += M;
    return *this;
  }
  mod operator*=(const mod& o) {
    x = 1ll * x * o.x % M;
    return *this;
  }
  mod operator/=(const mod& o) {
    return (*this) *= o.inv();
  }
  friend mod operator+(mod a, const mod& b) { return a += b; }
  friend mod operator-(mod a, const mod& b) { return a -= b; }
  friend mod operator*(mod a, const mod& b) { return a *= b; }
  friend mod operator/(mod a, const mod& b) { return a /= b; }
  auto operator<=>(const mod&) const = default;
  mod pow(ll n) const {
    mod a = x, b = 1;
    while (n > 0) {
      if (n % 2 == 1) b *= a;
      a *= a;
      n /= 2;
    }
    return b;
  }
  mod inv() const {
    return pow(M - 2);
  }
};
using mint = mod<998244353, 3>;
```

GCD.h

Opis: Znajduje x i y takie, że $ax + by = \gcd(a, b)$.
Czas: $\mathcal{O}(\log(\min(a, b)))$

```
ll gcd(ll a, ll b, ll& x, ll& y) {
  if (!b) return x = 1, y = 0, a;
  ll g = gcd(b, a % b, y, x);
  return y -= x * (a / b), g;
}
```

5.2 Wielomiany

NTT.h

Opis: Mnoży dwa wielomiany modulo liczba NTT-pierwsza.
Czas: $\mathcal{O}((n + m)\log(n + m))$

```
template<typename T>
void ntt(vector<T>& a, bool inv) {
  int n = ssize(a);
  vector<T> b(n);
  for (int i = n / 2; i > 0; i /= 2, swap(a, b)) {
    T w = T(T::ROOT).pow((T::MOD - 1) / n * i), m = 1;
    for (int j = 0; j < n; j += 2 * i, m *= w) {
      for (int k = 0; k < i; k++) {
        T u = a[j + k], v = a[j + k + i] * m;
        b[j / 2 + k] = u + v;
        b[j / 2 + k + n / 2] = u - v;
      }
    }
  }
  if (inv) {
    reverse(a.begin() + 1, a.end());
    T ni = T(n).inv();
    for (int i = 0; i < n; i++) a[i] *= ni;
  }
}
template<typename T>
vector<T> conv(vector<T> a, vector<T> b) {
  int s = ssize(a) + ssize(b) - 1;
  int n = 1 << (__lg(2 * s - 1));
  a.resize(n); b.resize(n);
  ntt(a, false); ntt(b, false);
  for (int i = 0; i < n; i++) a[i] *= b[i];
  ntt(a, true);
  a.resize(s);
  return a;
}
```

Conv3.h

Opis: Mnoży dwa wielomiany. Musi zachodzić $n+m \leq 2^{24}$ oraz $c_k \leq 5 \cdot 10^{25}$.
Czas: $\mathcal{O}((n + m)\log(n + m))$

```
template<typename T>
vector<T> mconv(const auto& a, const auto& b) {
  auto cp = [&](const auto& v) {
    vector<T> vv(ssize(v));
    for (int i = 0; i < ssize(v); i++) vv[i] = T(v[i].x);
    return vv;
  };
  return conv(cp(a), cp(b));
}
template<typename T>
vector<T> conv3(const vector<T>& a, const vector<T>& b) {
  using m0 = mod<754974721, 11>; auto c0 = mconv<m0>(a, b);
  using m1 = mod<167772161, 3>; auto c1 = mconv<m1>(a, b);
  using m2 = mod<469762049, 3>; auto c2 = mconv<m2>(a, b);
  m1 r01 = m1(m0::MOD).inv();
  m2 r02 = m2(m0::MOD).inv(), r12 = m2(m1::MOD).inv();
  vector<T> d(ssize(c0));
  for (int i = 0; i < ssize(c0); i++) {
    int x = c0[i].x;
    int y = ((c1[i] - x) * r01).x;
    int z = (((c2[i] - x) * r02 - y) * r12).x;
    d[i] = (T(z) * m1::MOD + y) * m0::MOD + x;
  }
  return d;
}
```

5.3 Sploty

FST.h

Opis: Wykonuje splot bitowy. n musi być potęgą dwójki.
Czas: $\mathcal{O}(n \log n)$

```
void fst(vector<mint>& a, bool inv) {
  int n = ssize(a);
  for (int i = 1; i < n; i *= 2) {
    for (int j = 0; j < n; j += 2 * i) {
      for (int k = 0; k < i; k++) {
        mint u = a[j + k], v = a[j + k + i];
        a[j + k] = u + v, a[j + k + i] = u - v; // XOR
        // a[j + k] = inv ? u - v : u + v; // AND
        // a[j + k + i] = inv ? v - u : u + v; // OR
      }
    }
  }
  // XOR
  if (inv) {
    mint ni = mint(n).inv();
    for (int i = 0; i < n; i++) a[i] = a[i] * ni;
  }
}
vector<mint> conv(vector<mint> a, vector<mint> b) {
  int n = ssize(a);
  fst(a, false); fst(b, false);
  for (int i = 0; i < n; i++) a[i] = a[i] * b[i];
  fst(a, true);
  return a;
}
```

Teksty (6)

KMP.h

Opis: `p[i]` – najdłuższy ścisły sufiks `s[0:i]` który jest prefiksem `s`.
Czas: $\mathcal{O}(n)$

```
vector<int> kmp(const string& s) {
  int n = ssize(s);
  vector<int> p(n);
  for (int i = 1; i < n; i++) {
    int j = p[i - 1];
    while (j > 0 && s[i] != s[j]) j = p[j - 1];
    p[i] = j + (s[i] == s[j]);
```

```
  }
  return p;
}
```

Manacher.h

Opis: Znajduje długość najdłuższego palindromu w każdym środku. `p[2 * i]` – środek w i, `p[2 * i + 1]` – środek między i a $i+1$.
Czas: $\mathcal{O}(n)$

```
vector<int> manacher(const string& s) {
  int n = ssize(s);
  string t(2 * n - 1, '#');
  for (int i = 0; i < n; i++) t[2 * i] = s[i];
  vector<int> p(2 * n - 1);
  for (int i = 0, l = -1, r = -1; i < 2 * n - 1; i++) {
    if (i <= r) p[i] = min(r - i + 1, p[l + r - i]);
    while (p[i] < min(i + 1, 2 * n - 1 - i)) {
      if (t[i - p[i]] != t[i + p[i]]) break;
      p[i]++;
    }
    if (i + p[i] - 1 > r) {
      l = i - p[i] + 1;
      r = i + p[i] - 1;
    }
  }
  for (int i = 0; i < 2 * n - 1; i++) {
    p[i] -= t[i - p[i] + 1] == '#';
  }
  return p;
}
```

SuffixArray.h

Opis: Jeżeli tekst ma znaki inne niż `a-z` trzeba zmienić inicjalizację.
Czas: $\mathcal{O}(n \log n)$

```
vector<int> suffix_array(const string& s) {
  int n = ssize(s);
  vector<int> p(n), cnt(26);
  for (int i = 0; i < n; i++) cnt[s[i] - 'a']++;
  for (int i = 1; i < 26; i++) cnt[i] += cnt[i - 1];
  for (int i = 0; i < n; i++) p[--cnt[s[i] - 'a']] = i;
  vector<int> rnk(n);
  for (int i = 1; i < n; i++) {
    rnk[p[i]] = s[p[i]] == s[p[i - 1]] ? rnk[p[i - 1]] : i;
  }
  cnt.resize(n);
  vector<int> np(n), nrnk(n);
  for (int len = 1; len < n; len *= 2) {
    iota(cnt.begin(), cnt.end(), 0);
    for (int i = n - len; i < n; i++) np[cnt[rnk[i]]++] = i;
    for (int i = 0; i < n; i++) {
      if (p[i] - len >= 0) {
        np[cnt[rnk[p[i] - len]]++] = p[i] - len;
      }
    }
    nrnk[np[0]] = 0;
    for (int i = 1; i < n; i++) {
      int a = np[i - 1];
      int b = np[i];
      if (max(a, b) + len < n && rnk[a] == rnk[b] &&
          rnk[a + len] == rnk[b + len]) {
        nrnk[b] = nrnk[a];
      } else {
        nrnk[b] = i;
      }
    }
    swap(p, np);
    swap(rnk, nrnk);
  }
  return p;
};
vector<int> build_lcp(const string& s, const vector<int>& sa) {
  int n = ssize(s);
  vector<int> pos(n);
  for (int i = 0; i < n; i++) pos[sa[i]] = i;
  vector<int> lcp(n - 1);
  int k = 0;
  for (int i = 0; i < n; i++) {
    if (pos[i] == 0) continue;
    while (i + k < n && s[i + k] == s[sa[pos[i] - 1] + k]) k++;
    lcp[pos[i] - 1] = k;
    k = max(0, k - 1);
  }
  return lcp;
}
```

Z.h

Opis: `f[i]` – największe k takie, że `f[i:i+k]` jest prefiksem `s`.
Czas: $\mathcal{O}(n)$

```
vector<int> z(const string& s) {
  int n = ssize(s);
  vector<int> f(n);
  f[0] = n;
  for (int i = 1, l = 0, r = 0; i < n; i++) {
    if (i <= r) f[i] = min(r - i + 1, f[i - l]);
    while (f[i] < n - i && s[i + f[i]] == s[f[i]]) f[i]++;
    if (i + f[i] - 1 > r) {
      l = i;
      r = i + f[i] - 1;
    }
  }
  return f;
}
```

Geometria (7)

7.1 Podstawy

Point.h

Opis: Podstawowy szablon do geometrii. Do wszystkich porównań należy używać `sgn`.

```
using D = ll;
const D EPS = D(1e-9);
int sgn(D x) { return (x > EPS) - (x < -EPS); }
struct P {
  D x, y;
  P operator+(P o) const { return {x + o.x, y + o.y}; }
  P operator-(P o) const { return {x - o.x, y - o.y}; }
  P operator*(D a) const { return {x * a, y * a}; }
  P operator/(D a) const { return {x / a, y / a}; }
  auto operator<=>(P o) const {
    return pair(sgn(x - o.x), sgn(y - o.y)) <=> pair(0, 0);
  }
  bool operator==(P o) const {
    return sgn(x - o.x) == 0 && sgn(y - o.y) == 0;
  }
};
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 norm(P a) { return a.x * a.x + a.y * a.y; }
auto& operator<<(auto& o, P a) {
  return o << '(' << a.x << ", " << a.y << ')';
}
```

AngleCmp.h

Opis: Sortuje punkty w kolejności CCW, zaczynając od $y < 0$. Punkt $(0, 0)$ należy do linii $x \geq 0$, $y = 0$.

```
int half(P a) {
  if (sgn(a.y) < 0) return -1;
  if (sgn(a.y) == 0 && sgn(a.x) >= 0) return 0;
  return 1;
}
bool angle_cmp(P a, P b) {
  if (half(a) != half(b)) return half(a) < half(b);
  return sgn(cross(a, b)) > 0;
}
```

LineIntersection.h

Opis: Znajduje punkt przecięcia prostych. W obliczeniach użyty jest iloczyn trzech współrzędnych.

```
P line_inter(P s1, P t1, P s2, P t2) {
  D d = cross(t1 - s1, t2 - s2);
  assert(sgn(d) != 0); // parallel
  D p = cross(t1 - s2, t2 - s2), q = cross(t2 - s2, s1 - s2);
  return (s1 * p + t1 * q) / d;
}
```

7.2 Wielokąty

ConvexHull.h

Opis: Znajduje otoczkę wypukłą w kierunku CCW. Usuwa punkty współliniowe.
Czas: $\mathcal{O}(n \log n)$

```
vector<P> convex_hull(vector<P> p) {
  if (ssize(p) <= 1) return p;
  sort(p.begin(), p.end());
  vector<P> h(ssize(p) + 1);
  int s = 0, t = 0;
  for (int it = 0; it < 2; it++) {
    for (P a : p) {
      while (t >= s + 2) {
        P u = h[t - 2], v = h[t - 1];
        if (sgn(cross(v - u, a - v)) <= 0) t--;
        else break;
      }
      h[t++] = a;
    }
    reverse(p.begin(), p.end());
    s = --t;
  }
  h.resize(t - (t == 2 && h[0] == h[1]));
  return h;
}
```

PolygonTangents.h

Opis: Znajduje najbliższe punkty styczne różne od a. Wielokąt musi być CCW i $n \geq 3$. Punkt a nie może leżeć w ściśłym wnętrzu wielokąta.
Czas: $\mathcal{O}(\log n)$

```
pair<P, P> tangents(const vector<P>& p, P a) {
  int n = ssize(p);
  P t[2];
  for (int it = 0; it < 2; it++) {
    auto dir = [&](int i) {
      P u = p[i] - a;
      P v = p[i < n - 1 ? i + 1 : 0] - a;
      D c = cross(u, v);
      if (sgn(c) != 0) return sgn(c) < 0;
```

```
      if (sgn(dot(u, v)) <= 0) return true;
      return sgn(norm(u) - norm(v)) > 0;
    };
    auto dirx = [&](int i) { return dir(i) ^ it; };
    if (dirx(0) == 1 && dirx(n - 1) == 0) {
      t[it] = p[0];
      continue;
    }
    int s[2] = {0, n - 1};
    while (s[1] - s[0] > 2) {
      int mid = (s[0] + s[1]) / 2;
      int x = dirx(mid);
      if (dirx(s[x ^ 1]) == (x ^ 1)) {
        s[x] = mid;
      } else {
        bool b = sgn(cross(p[mid] - a, p[s[1]] - a)) < 0;
        s[b ^ x ^ it ^ 1] = mid;
      }
    }
    t[it] = p[s[0] + 1 + (dirx(s[0] + 1) == 0)];
  }
  return {t[0], t[1]};
}
```

Inne (8)

GCC.h

Opis: Pragmy do dopychania kolanem. Należy wstawić przed bitsami.

```
#include <bits/allocator.h>
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
```