UH Random Team Reference

Contents

1. TEMPLATE

1.1. MAIN.

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

#define MAX(a, b) (a > b) ? a : b
#define MIN(a, b) (a < b) ? a : b
#define int long long
#define vi vector<int>
#define pii pair<int, int>
#define vii vector<pii>

using namespace std;

void solve()
{
}

int32_t main()
{
   ios_base::sync_with_stdio(0);
   cin.tie(0);

   int t;
   cin >> t;

   for (int i = 0; i < t; i++)
   {
      solve();
   }

   return 0;
}
```

2. MATH

2.1. NTT.

```
using ll = long long;

const ll mod = (119 << 23) + 1, root = 62; // 998244353

ll qp(ll b, ll e)
{
   ll ans = 1;
   for (; e; b = b * b % mod, e /= 2)
      if (e & 1)
         ans = ans * b % mod;
   return ans;
}

void ntt(vector<ll> &a, vector<ll> &rt, vector<ll> &rev, int n)
{
   for (int i = 0; i < n; i++)
      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)
         for (int j = 0; j < k; j++)
         {
            ll z = rt[j + k] * a[i + j + k] % mod, &ai = a[i + j];
            a[i + j + k] = (z > ai ? ai - z + mod : ai - z);
            ai += (ai + z >= mod ? z - mod : z);
         }
}

vector<ll> convolve(const vector<ll> &a, const vector<ll> &b)
{
   if (a.empty() || b.empty())
      return {};

   int s = a.size() + b.size() - 1, B = 32 - __builtin_clz(s), n = 1 << B;

   vector<ll> L(a), R(b), out(n), rt(n, 1), rev(n);
   L.resize(n), R.resize(n);

   for (int i = 0; i < n; i++)
      rev[i] = (rev[i / 2] | (i & 1) << B) / 2;

   ll curL = mod / 2, inv = qp(n, mod - 2);
   for (int k = 2; k < n; k *= 2)
```

```
   {
      ll z[] = {1, qp(root, curL /= 2)};
      for (int i = k; i < 2 * k; i++)
         rt[i] = rt[i / 2] * z[i & 1] % mod;
   }

   ntt(L, rt, rev, n);
   ntt(R, rt, rev, n);
   for (int i = 0; i < n; i++)
      out[-i & (n - 1)] = L[i] * R[i] % mod * inv % mod;

   ntt(out, rt, rev, n);

   return {out.begin(), out.begin() + s};
}
```

2.2. MATRIX POW.

```
const int MAXN = 2;

struct Matrix
{
   ll mat[MAXN][MAXN];
};

Matrix operator*(const Matrix &a, const Matrix &b)
{
   Matrix c;
   for (int i = 0; i < MAXN; ++i)
      for (int j = 0; j < MAXN; ++j)
         c.mat[i][j] = 0;

   for (int i = 0; i < MAXN; i++)
   {
      for (int k = 0; k < MAXN; k++)
      {
         if (a.mat[i][k] == 0)
            continue;
         for (int j = 0; j < MAXN; j++)
         {
            c.mat[i][j] += a.mat[i][k] * b.mat[k][j];
         }
      }
   }
   return c;
}
Matrix operator^(Matrix &base, ll e)
{
   Matrix c;
   for (int i = 0; i < MAXN; i++)
      for (int j = 0; j < MAXN; j++)
         c.mat[i][j] = (i == j);
   while (e)
   {
      if (e & 1ll)
         c = c * base;
      base = base * base;
      e >>= 1;
   }
   return c;
}
```

2.3. FFT.

```
struct point
{
   double x, y;
   point(double x = 0, double y = 0) : x(x), y(y) {}
};

point operator+(const point &a, const point &b)
{
   return {a.x + b.x, a.y + b.y};
}
point operator-(const point &a, const point &b)
{
   return {a.x - b.x, a.y - b.y};
}
point operator*(const point &a, const point &b)
{
   return {a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x};
}
```

```
point operator/(const point &a, double d) { return {a.x / d, a.y / d}; }

void fft(vector<point> &a, int sign = 1)
{
   int n = a.size(); // n should be a power of two
   double theta = 8 * sign * atan(1.0) / n;
   for (int i = 0, j = 1; j < n - 1; ++j)
   {
      for (int k = n >> 1; k > (i ^= k); k >>= 1)
         ;
      if (j < i)
         swap(a[i], a[j]);
   }
   for (int m, mh = 1; (m = mh << 1) <= n; mh = m)
   {
      int irev = 0;
      for (int i = 0; i < n; i += m)
      {
         point w = point(cos(theta * irev), sin(theta * irev));
         for (int k = n >> 2; k > (irev ^= k); k >>= 1)
            ;
         for (int j = i; j < mh + i; ++j)
         {
            int k = j + mh;
            point x = a[j] - a[k];
            a[j] = a[j] + a[k];
            a[k] = w * x;
         }
      }
   }
   if (sign == -1)
      for (auto &p : a)
         p = p / n;
}

vector<point> convolve(vector<point> &a, vector<point> &b)
{
   int n = a.size();
   int m = b.size();
   int k = n + m;
   while (k != (k & -k))
      k += (k & -k);
   while (a.size() < k)
      a.push_back(point(0, 0));
   while (b.size() < k)
      b.push_back(point(0, 0));

   fft(a, 1);
   fft(b, 1);

   vector<point> c(k);
   for (int i = 0; i < k; i++)
      c[i] = a[i] * b[i];

   fft(c, -1);

   return c;
}
```

2.4. FWHT.

```
using ll = long long;

const int mod = 1e9 + 7;

template <const int _mod_>
struct mod_int
{
   static const int mod = _mod_;
   int val;

   mod_int(long long v = 0)
   {
      if (v < 0)
         v = v % mod + mod;
      if (v >= mod)
         v %= mod;
      val = v;
   }

   static int mod_inv(int a, int m = mod)
   {
      int g = m, r = a, x = 0, y = 1;
      while (r != 0)
      {
         int q = g / r;
         g %= r;
         swap(g, r);
```

```
        x -= q * y;
        swap(x, y);
     }
     return x < 0 ? x + m : x;
  }

  explicit operator int() const { return val; }

  mod_int &operator+=(const mod_int &other)
  {
     val += other.val;
     if (val >= mod)
        val -= mod;
     return *this;
  }

  mod_int &operator-=(const mod_int &other)
  {
     val -= other.val;
     if (val < 0)
        val += mod;
     return *this;
  }

  static unsigned fast_mod(uint64_t x, unsigned m = mod)
  {
#if !defined(_WIN32) || defined(_WIN64)
     return x % m;
#endif
     // Optimized mod for Codeforces 32-bit machines.
     // x must be less than 2^32 * m for this to work, so that x / m fits in
     // a 32-bit integer.
     unsigned x_high = x >> 32, x_low = (unsigned)x;
     unsigned quot, rem;
     asm("divl %4\n"
        : "=a"(quot), "=d"(rem)
        : "d"(x_high), "a"(x_low), "r"(m));
     return rem;
  }

  mod_int &operator*=(const mod_int &other)
  {
     val = fast_mod((uint64_t)val * other.val);
     return *this;
  }

  mod_int &operator/=(const mod_int &other) { return *this *= other.inv(); }

  friend mod_int operator+(const mod_int &a, const mod_int &b)
  {
     return mod_int(a) += b;
  }
  friend mod_int operator-(const mod_int &a, const mod_int &b)
  {
     return mod_int(a) -= b;
  }
  friend mod_int operator*(const mod_int &a, const mod_int &b)
  {
     return mod_int(a) *= b;
  }
  friend mod_int operator/(const mod_int &a, const mod_int &b)
  {
     return mod_int(a) /= b;
  }

  mod_int &operator++()
  {
     val = val == mod - 1 ? 0 : val + 1;
     return *this;
  }

  mod_int &operator--()
  {
     val = val == 0 ? mod - 1 : val - 1;
     return *this;
  }

  mod_int operator++(int)
  {
     mod_int a = *this;
     ++*this;
     return a;
  }
  mod_int operator--(int)
  {
     mod_int a = *this;
     --*this;
     return a;
  }
  mod_int operator-() const { return val == 0 ? 0 : mod - val; }
  mod_int inv() const { return mod_inv(val); }

  bool operator==(const mod_int &other) const { return val == other.val; }
  bool operator!=(const mod_int &other) const { return val != other.val; }
```

```
   bool operator<(const mod_int &other) const { return val < other.val; }
   bool operator>(const mod_int &other) const { return val > other.val; }

   template <typename T>
   bool operator<(const T &other) const
   {
      return val < other;
   }

   template <typename T>
   bool operator>(const T &other) const
   {
      return val > other;
   }

   friend string to_string(const mod_int &m) { return to_string(m.val); }
   friend mod_int abs(const mod_int &m) { return mod_int(m.val); }

   friend ostream &operator<<(ostream &stream, const mod_int &m)
   {
      return stream << m.val;
   }
   friend istream &operator>>(istream &stream, mod_int &m)
   {
      return stream >> m.val;
   }
};

// Notar que se uso este tipo de datos entero, que implementa todas las
// operaciones basicos en el sistem residual modulo 10^9+7 para hacer mas
// faciles las implementaciones si se quiere implementarlo, se puede dejar en
// long long, y modificar las operaciones para mantener los resultados teneindo
// en cuenta el modulo a usar.
using T = mod_int<mod>;

void HADAMARD_XOR(vector<T> &a, bool inverse = false)
{
   int n = a.size();
   for (int k = 1; k < n; k <<= 1)
   {
      for (int i = 0; i < n; i += 2 * k)
      {
         for (int j = 0; j < k; j++)
         {
            T x = a[i + j];
            T y = a[i + j + k];
            a[i + j] = x + y;
            a[i + j + k] = x - y;
         }
      }
   }

   if (inverse)
   {
      T q = 1 / static_cast<T>(n);

      for (int i = 0; i < n; i++)
      {
         a[i] *= q;
      }
   }
}

void HADAMARD_AND(vector<T> &a, bool inverse = false)
{
   int n = a.size();
   for (int k = 1; k < n; k <<= 1)
   {
      for (int i = 0; i < n; i += 2 * k)
      {
         for (int j = 0; j < k; j++)
         {
            T x = a[i + j];
            T y = a[i + j + k];
            if (inverse)
            {
               a[i + j] = -x + y;
               a[i + j + k] = x;
            }
            else
            {
               a[i + j] = y;
               a[i + j + k] = x + y;
            }
         }
      }
   }
}

void HADAMARD_OR(vector<T> &a, bool inverse = false)
{
   int n = a.size();
   for (int k = 1; k < n; k <<= 1)
   {
```

```
      for (int i = 0; i < n; i += 2 * k)
      {
         for (int j = 0; j < k; j++)
         {
            T x = a[i + j];
            T y = a[i + j + k];
            if (inverse)
            {
               a[i + j] = y;
               a[i + j + k] = x - y;
            }
            else
            {
               a[i + j] = x + y;
               a[i + j + k] = x;
            }
         }
      }
   }
}

// Las demas operaciones a nivel de bit tienen una implementacion semejante
template <typename T>
vector<T> FWHT_XOR(vector<T> a, vector<T> b)
{
   bool eq = (a == b);
      int n = 1;

      while (n < (int)max(a.size(), b.size()))
      {
         n <<= 1;
      }

      a.resize(n);
      b.resize(n);

      HADAMARD_XOR(a);

      if (eq)
         b = a;
      else
         HADAMARD_XOR(b);

      for (int i = 0; i < n; i++)
      {
         a[i] *= b[i];
      }

      HADAMARD_XOR(a, true);

      return a;
   }
```

3. GEOMETRY

3.1. BASICS.

```
db DEG_to_RAD(db d) { return d*M_PI / 180.0; }
db RAD_to_DEG(db r) { return r*180.0 / M_PI; }

db EPS = 1e-9;

struct point {
   db x, y;
   point() { x = y = 0.0; }
   point(db _x, db _y) : x(_x), y(_y) {}

   bool operator < (const point &other) const {
      if (fabs(x-other.x) > EPS)
      return x < other.x;
      return y < other.y;
      }

      bool operator == (const point &other) const {
         return (fabs(x-other.x) < EPS) && (fabs(y-other.y) < EPS);
      }

      db dist(const point &other) {
         return hypot(x-other.x, y-other.y);
      }

      point rotate(db theta) {
         db rad = DEG_to_RAD(theta);
         return point(x*cos(rad) - y*sin(rad), x*sin(rad) + y*cos(rad));
      }
   };
```

```
struct line {
   db a, b, c;
   line() {}
   line(db _a, db _b, db _c) : a(_a), b(_b), c(_c) {}

   void pointsToLine(const point &p1, const point &p2) {
      if (fabs(p1.x-p2.x) < EPS) {
         a = 1.0;
         b = 0.0;
         c = -p1.x;
      }
      else {
         a = -(db)(p1.y-p2.y) / (p1.x-p2.x);
         b = 1.0;
         c = -(db)(a*p1.x) - p1.y;
      }
   }

   void pointSlopeToLine(point p, db m) {
      a = -m;
      b = 1.0;
      c = -((a * p.x) + (b * p.y));
   }

   bool areParallel(const line &other) {
      return (fabs(a-other.a) < EPS) && (fabs(b-other.b) < EPS);
   }

   bool areSame(const line &other) {
      return areParallel(other) && (fabs(c-other.c) < EPS);
   }

   bool areIntersect(const line &other, point &p) {
      if (areParallel(other)) return false;
      p.x = (other.b*c - b*other.c) / (other.a*b - a*other.b);
      if (fabs(b) > EPS) p.y = -(a*p.x + c);
      else p.y = -(other.a*p.x + other.c);
      return true;
   }
};

struct vec{
   db x, y;
   vec(db _x, db _y) : x(_x), y(_y) {}
   vec(const point &a, const point &b) : x(b.x - a.x), y(b.y - a.y) {}
   vec scale(db s) {
      return vec(x*s, y*s);
   }

   point translate(const point &p) {
      return point(x+p.x, y+p.y);
   }

   db dot(vec a, vec b) { return a.x*b.x + a.y*b.y; }

   db norm_sq(vec v) { return v.x*v.x + v.y*v.y; }

   db angle(const point &a, const point &o, const point &b) {
      vec oa = vec(o, a), ob = vec(o, b);
      return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
   }

   db cross(vec a, vec b) { return a.x*b.y - a.y*b.x; }

   bool ccw(point p, point q, point r) {
      return cross(vec(p, q), vec(p, r)) > EPS;
   }

   bool collinear(point p, point q, point r) {
      return fabs(cross(vec(p, q), vec(p, r))) < EPS;
   }

   db distToLine(point p, point a, point b) {
      vec ap = vec(a, p), ab = vec(a, b);
      db u = dot(ap, ab) / norm_sq(ab);
      point c = ab.scale(u).translate(a);
      return c.dist(p);
   }

   db distToLineSegment(point p, point a, point b) {
      vec ap = vec(a, p), ab = vec(a, b);
      db u = dot(ap, ab) / norm_sq(ab);
      if (u < 0.0) {
         point c = point(a.x, a.y);
         return c.dist(p);
      }
      if (u > 1.0) {
         point c = point(b.x, b.y);
         return c.dist(p);
      }
      return distToLine(p, a, b);
   }
```

```
};

struct circle {
   point c;
   db r;
   circle(const point &_c, db _r) : c(_c), r(_r) {}

   int inside(const point &p) {
      db dist = c.dist(p);
      return dist < r ? 1 : (fabs(dist-r) < EPS ? 0 : -1);
   }

   point inCircle(point p1, point p2, point p3) {
      line l1, l2;
      double ratio = p1.dist(p2) / p1.dist(p3);
      point p = vec(p2, p3).scale(ratio / (1+ratio)).translate(p2);
      l1.pointsToLine(p1, p);
      ratio = p2.dist(p1) / p2.dist(p3);
      p = vec(p1, p3).scale(ratio / (1+ratio)).translate(p1);
      l2.pointsToLine(p2, p);
      point c;
      l1.areIntersect(l2, c);
      return c;
   }
};
```

3.2. POLYGON.

```
struct polygon {
   vector<point> P;
   polygon(const vector<point> &_P) : P(_P) {}

   db perimeter() {
      db ans = 0.0;
      for (int i = 0; i < (int)P.size()-1; ++i)
         ans += P[i].dist(P[i+1]);
      return ans;
   }

   db area() {
      db ans = 0.0;
      for (int i = 0; i < (int)P.size()-1; ++i)
         ans += (P[i].x*P[i+1].y - P[i+1].x*P[i].y);
      return fabs(ans) / 2.0;
   }

   bool isConvex(const vector<point> &P) {
      int n = (int)P.size();
      if (n <= 3) return false;
      bool firstTurn = vec().ccw(P[0], P[1], P[2]);
      for (int i = 1; i < n-1; ++i)
         if (vec().ccw(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]) != firstTurn)
            return false;
      return true;
   }

   int insidePolygon(point pt) {
      int n = (int)P.size();
      if (n <= 3) return -1;
      bool on_polygon = false;
      for (int i = 0; i < n-1; ++i)
         if (fabs(pt.dist(P[i]) + pt.dist(P[i+1]) - P[i].dist(P[i+1])) < EPS)
            on_polygon = true;
      if (on_polygon) return 0;
      double sum = 0.0;
      for (int i = 0; i < n-1; ++i) {
      if (vec().ccw(pt, P[i], P[i+1]))
         sum += vec().angle(P[i], pt, P[i+1]);
      else
         sum -= vec().angle(P[i], pt, P[i+1]);
      }
      return fabs(sum) > M_PI ? 1 : -1;
   }
};
```

4. DATA STRUCTURE

4.1. SEGMENT TREE.

```
class SegmentTree
{
private:
   vi values;

   vi p_values;
   int n;

   int left(int p) { return p << 1; };

   int right(int p) { return (p << 1) + 1; }

   int simple_node(int index) { return values[index]; }

   int prop(int x, int y) { return x + y; }

   void build(int p, int l, int r)
   {
      if (l == r)
      {
         p_values[p] = simple_node(l);
         return;
      }

      build(left(p), l, (l + r) / 2);
      build(right(p), (l + r) / 2 + 1, r);

      p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
   }

   void set(int p, int l, int r, int i, int v)
   {
      if (l == r)
      {
         values[l] = v;
         p_values[p] = simple_node(l);
         return;
      }

      if (i <= (l + r) / 2)
         set(left(p), l, (l + r) / 2, i, v);
      else
         set(right(p), (l + r) / 2 + 1, r, i, v);

      p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
   }

   int query(int p, int l, int r, int lq, int rq)
   {
      if (lq <= l && r <= rq)
         return p_values[p];

      int l1 = l, r1 = (l + r) / 2;
      int l2 = (l + r) / 2 + 1, r2 = r;

      if (l1 > rq || lq > r1)
         return query(right(p), l2, r2, lq, rq);
      if (l2 > rq || lq > r2)
         return query(left(p), l1, r1, lq, rq);

      int lt = query(left(p), l1, r1, lq, rq);
      int rt = query(right(p), l2, r2, lq, rq);

      return prop(lt, rt);
   }

public:
   SegmentTree(vi &a)
   {
      values = a;
      n = a.size();
      p_values.assign(4 * n, 0);
      build(1, 0, n - 1);
   }

   int query(int i, int j) { return query(1, 0, n - 1, i, j); }

   void set(int i, int v) { set(1, 0, n - 1, i, v); }

   int get(int i) { return values[i]; }
};
```

4.2. DISJOINT SET UNION.

```
struct dsu {
   vi p;
   void init(int n) {
      p = vi(n, -1);
   }
   int get(int x) {
      if (p[x] < 0)
         return x;
      return p[x] = get(p[x]);
   }
   void unite(int a, int b) {
      a = get(a);
      b = get(b);
      if (a != b) {
         if (p[a] > p[b])
            swap(a, b);
         p[a] += p[b];
         p[b] = a;
      }
   }
};
```

4.3. PBDS.

```
#include <bits/extc++.h> // pbds
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
         tree_order_statistics_node_update>
   ost;

int main()
{
  int n = 9;
  int A[] = {2, 4, 7, 10, 15, 23, 50, 65, 71}; // as in Chapter 2
  ost tree;
  for (int i = 0; i < n; ++i) // O(n log n)
    tree.insert(A[i]);
  // O(log n) select
  cout << *tree.find_by_order(0) << "\n"; // 1-smallest = 2
  cout << *tree.find_by_order(n - 1) << "\n"; // 9-smallest/largest = 71
  cout << *tree.find_by_order(4) << "\n"; // 5-smallest = 15
  // O(log n) rank
  cout << tree.order_of_key(2) << "\n"; // index 0 (rank 1)
  cout << tree.order_of_key(71) << "\n"; // index 8 (rank 9)
  cout << tree.order_of_key(15) << "\n"; // index 4 (rank 5)
  return 0;
}
```

4.4. SEGMENT TREE LAZY.

```
class SegmentTreeLazy
{
private:
   vi values;
   vector<bool> lazy;
   vi l_values;
   vi p_values;
   int n;

   int left(int p) { return p << 1; };

   int right(int p) { return (p << 1) + 1; }

   int simple_node(int index) { return values[index]; }

   int prop(int x, int y) { return x + y; }

   int prop_lazy(int x, int y) { return x + y; }

   int prop_lazy_up(int x, int y, int s) { return x + y * s; }

   void update_lazy(int p, int l, int r)
   {
      if (l == r)
      {
         values[l] = prop_lazy(values[l], l_values[p]);
```

```
   }

   p_values[p] = prop_lazy_up(p_values[p], l_values[p], r - l + 1);
}

void propagate_lazy(int p, int l, int r)
{
   lazy[p] = false;

   if (l == r)
      return;

   l_values[left(p)] = lazy[left(p)]
                     ? prop_lazy(l_values[left(p)], l_values[p])
                     : l_values[p];
   l_values[right(p)] = lazy[right(p)]
                      ? prop_lazy(l_values[right(p)], l_values[p])
                      : l_values[p];

   lazy[left(p)] = true;
   lazy[right(p)] = true;
}

void build(int p, int l, int r)
{
   if (l == r)
   {
      p_values[p] = simple_node(l);
      return;
   }

   build(left(p), l, (l + r) / 2);
   build(right(p), (l + r) / 2 + 1, r);

   p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
}

void set(int p, int l, int r, int i, int v)
{
   if (lazy[p])
   {
      update_lazy(p, l, r);
      propagate_lazy(p, l, r);
   }

   if (l == r)
   {
      values[l] = v;
      p_values[p] = simple_node(l);
      return;
   }

   if (i <= (l + r) / 2)
      set(left(p), l, (l + r) / 2, i, v);
   else
      set(right(p), (l + r) / 2 + 1, r, i, v);

   p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
}

int query(int p, int l, int r, int lq, int rq)
{
   if (lazy[p])
   {
      update_lazy(p, l, r);
      propagate_lazy(p, l, r);
   }

   if (lq <= l && r <= rq)
      return p_values[p];

   int l1 = l, r1 = (l + r) / 2;
   int l2 = (l + r) / 2 + 1, r2 = r;

   if (l1 > rq || lq > r1)
      return query(right(p), l2, r2, lq, rq);
   if (l2 > rq || lq > r2)
      return query(left(p), l1, r1, lq, rq);

   int lt = query(left(p), l1, r1, lq, rq);
   int rt = query(right(p), l2, r2, lq, rq);

   return prop(lt, rt);
}

void set_rank(int p, int l, int r, int lq, int rq, int value)
{
   if (lazy[p])
   {
      update_lazy(p, l, r);
      propagate_lazy(p, l, r);
   }

   if (l > rq || lq > r)
```

```
        return;

    if (lq <= l && r <= rq)
    {
        lazy[p] = true;
        l_values[p] = value;
        update_lazy(p, l, r);
        propagate_lazy(p, l, r);
        return;
    }

    set_rank(left(p), l, (l + r) / 2, lq, rq, value);
    set_rank(right(p), (l + r) / 2 + 1, r, lq, rq, value);

    p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
  }

  int get(int p, int l, int r, int i)
  {
    if (lazy[p])
    {
        update_lazy(p, l, r);
        propagate_lazy(p, l, r);
    }

    if (l == r)
        return values[i];

    if (i <= (l + r) / 2)
        return get(left(p), l, (l + r) / 2, i);

    return get(right(p), (l + r) / 2 + 1, r, i);
  }

public:
  SegmentTreeLazy(vi &a)
  {
    values = a;
    n = a.size();
    p_values.assign(4 * n, 0);
    lazy.assign(4 * n, false);
    l_values.assign(4 * n, 0);
    build(1, 0, n - 1);
  }

  int query(int i, int j) { return query(1, 0, n - 1, i, j); }

  void set(int i, int v) { set(1, 0, n - 1, i, v); }

  void set_rank(int i, int j, int v) { set_rank(1, 0, n - 1, i, j, v); }

  int get(int i) { return get(1, 0, n - 1, i); }
};
```

4.5. AVL.

```
struct avl {
  int key;
  int height;
  int size;
  avl *left;
  avl *right;

  avl(int k) {
    key = k;
    height = 1;
    size = 1;
    left = NULL;
    right = NULL;
  }

  int getBalance() {
    int leftHeight = 0;
    int rightHeight = 0;

    if (left != NULL)
        leftHeight = left->height;

    if (right != NULL)
        rightHeight = right->height;

    return leftHeight - rightHeight;
  }

  void updateSize() {
    int leftSize = 0;
    int rightSize = 0;
```

```
    if (left != NULL)
      leftSize = left->size;

    if (right != NULL)
      rightSize = right->size;

    size = leftSize + rightSize + 1;
  }

  void updateHeight() {
    int leftHeight = 0;
    int rightHeight = 0;

    if (left != NULL)
      leftHeight = left->height;

    if (right != NULL)
      rightHeight = right->height;

    height = max(leftHeight, rightHeight) + 1;
  }

  avl *rotateLeft() {
    avl *newRoot = right;
    right = newRoot->left;
    newRoot->left = this;
    updateHeight();
    newRoot->updateHeight();
    return newRoot;
  }

  avl *rotateRight() {
    avl *newRoot = left;
    left = newRoot->right;
    newRoot->right = this;
    updateHeight();
    newRoot->updateHeight();
    return newRoot;
  }

  avl *balance() {
    updateHeight();
    updateSize();
    int balance = getBalance();

    if (balance == 2) {
      if (left->getBalance() < 0)
        left = left->rotateLeft();
      return rotateRight();
    }

    if (balance == -2) {
      if (right->getBalance() > 0)
        right = right->rotateRight();
      return rotateLeft();
    }

    return this;
  }

  avl *insert(int k) {
    if (k < key) {
      if (left == NULL)
        left = new avl(k);
      else
        left = left->insert(k);
    }
    else {
      if (right == NULL)
        right = new avl(k);
      else
        right = right->insert(k);
    }

    return balance();
  }

  avl *findMin() {
    if (left == NULL)
      return this;
    else
      return left->findMin();
  }

  avl *removeMin() {
    if (left == NULL)
      return right;
    left = left->removeMin();
    return balance();
  }

  avl *remove(int k) {
    if (k < key)
```

```
         left = left->remove(k);
      else if (k > key)
         right = right->remove(k);
      else {
         avl *leftChild = left;
         avl *rightChild = right;

         delete this;

         if (rightChild == NULL)
            return leftChild;

         avl *min = rightChild->findMin();
         min->right = rightChild->removeMin();
         min->left = leftChild;
         return min->balance();
      }

      return balance();
   }

   int getRank(int k) {
      if (k < key) {
         if (left == NULL)
            return 0;
         else
            return left->getRank(k);
      }
      else if (k > key) {
         if (right == NULL)
            return 1 + left->size;
         else
            return 1 + left->size + right->getRank(k);
      }
      else
         return left->size;
   }

   int getKth(int k) {
      if (k < left->size)
         return left->getKth(k);
      else if (k > left->size)
         return right->getKth(k - left->size - 1);
      else
         return key;
   }

   static avl *join(avl *left, avl *right) {
      if (left->height < right->height) {
         right->left = join(left, right->left);
         return right->balance();
      }
      else if (left->height > right->height) {
         left->right = join(left->right, right);
         return left->balance();
      }
      else {
         avl *min = right->findMin();
         min->right = right->removeMin();
         min->left = left;
         return min->balance();
      }
   }

   pair<avl *, avl *> split(int k) {
      if (k < key) {
         pair<avl *, avl *> p = left->split(k);
         left = p.second;
         return {p.first, join(this, left)};
      }
      else {
         pair<avl *, avl *> p = right->split(k);
         right = p.first;
         return {join(this, right), p.second};
      }
   }
};
```

4.6. ABI.

```
class Abi
{

private:
   vi p;
   int _size;

   int ls_one(int i) { return i & (-i); }
```

```
public:
   Abi(int n)
   {
      _size = n;
      p.assign(n + 1, 0);
   }

   int rsq(int k)
   {
      int sum = 0;

      for (int i = k; i > 0; i -= ls_one(i))
      {
         sum += p[i];
      }
      return sum;
   }

   int sum(int a, int b) { return rsq(b) - rsq(a - 1); }

   void adjust_sum(int k, int v)
   {
      for (int i = k; i < p.size(); i += ls_one(i))
         p[i] += v;
   }

   int size()
   {
      return _size;
   }
};
```

4.7. SQRT DECOMPOSITION.

```
struct sqd {
   int n;
   int b;
   vi a;
   vi bsum;
   sqd(vi &a) {
      n = a.size();
      b = sqrt(n);
      this->a = a;
      bsum.assign(b + 1, 0);
      for (int i = 0; i < n; i++)
         bsum[i / b] += a[i];
   }
   void update(int i, int v) {
      bsum[i / b] += v - a[i];
      a[i] = v;
   }
   int query(int l, int r) {
      int sum = 0;
      for (int i = l; i <= r; i++)
         if (i % b == 0 && i + b - 1 <= r) {
            sum += bsum[i / b];
            i += b - 1;
         } else
            sum += a[i];
      return sum;
   }
};
```

5. STRING

5.1. SUFFIX ARRAY.

```
class SuffixArray
{

public:
   SuffixArray(string s)
   {
      n = s.size() + 1;
      s_value = s + "$";

      ra.assign(n, 0);
```

```
      sa.assign(n, 0);
      temp_ra.assign(n, 0);
      temp_sa.assign(n, 0);

      construct_sa();
      build_lcp();
   }

   int size() { return n; }

   int get_int(int i) { return sa[i]; }

   int cant_match(string p)
   {
      pii ans = matching(p);

      if (ans.first == -1 && ans.second == -1)
         return 0;

      return ans.second - ans.first + 1;
   }

   int get_lcp(int i) { return plcp[sa[i]]; }

   int cant_substr() { return v_cant_substr; }

   string get_str(int i) { return s_value.substr(sa[i], n - sa[i] - 1); }

private:
   string s_value;
   int n;
   int v_cant_substr;

   vi ra;
   vi sa;
   vi c;
   vi temp_ra;
   vi temp_sa;
   vi phi;
   vi plcp;

   void counting_sort(int k)
   {
      int sum = 0;
      int maxi = max((int)300, n);

      c.assign(maxi, 0);

      for (int i = 0; i < n; i++)
         c[i + k < n ? ra[i + k] : 0]++;

      for (int i = 0; i < maxi; i++)
      {
         int tx = c[i];
         c[i] = sum;
         sum += tx;
      }

      for (int i = 0; i < n; i++)
         temp_sa[c[sa[i] + k < n ? ra[sa[i] + k] : 0]++] = sa[i];

      for (int i = 0; i < n; i++)
         sa[i] = temp_sa[i];
   }

   void construct_sa()
   {
      int k, r;

      for (int i = 0; i < n; i++)
      {
         ra[i] = s_value[i];
         sa[i] = i;
      }

      for (k = 1; k < n; k <<= 1)
      {
         counting_sort(k);
         counting_sort(0);

         temp_ra[sa[0]] = r = 0;

         for (int i = 1; i < n; i++)
            temp_ra[sa[i]] = (ra[sa[i]] == ra[sa[i - 1]] && ra[sa[i] + k] == ra[sa

         for (int i = 0; i < n; i++)
            ra[i] = temp_ra[i];

         if (ra[sa[n - 1]] == n - 1)
            break;
      }
   }

   pii matching(string p)
```

```
    {
        int l = 0;
        int r = n - 1;
        int p_size = p.size();

        string comp;

        while (l < r)
        {
            int m = (l + r) / 2;

            comp = s_value.substr(sa[m], min(n - sa[m], p_size));

            if (comp >= p)
                r = m;
            else
                l = m + 1;
        }

        comp = s_value.substr(sa[l], min(n - sa[l], p_size));

        if (comp != p)
            return {-1, -1};

        int ans_l = l;

        l = 0;
        r = n - 1;

        while (l < r)
        {
            int m = (l + r) / 2;

            comp = s_value.substr(sa[m], min(n - sa[m], p_size));

            if (comp > p)
                r = m;
            else
                l = m + 1;
        }

        comp = s_value.substr(sa[r], min(n - sa[r], p_size));

        if (comp != p)
            r--;

        int ans_r = r;

        return {ans_l, ans_r};
    }

    void build_lcp()
    {
        phi.assign(n, 0);
        plcp.assign(n, 0);

        phi[0] = -1;

        for (int i = 1; i < n; i++)
            phi[sa[i]] = sa[i - 1];

        int l = 0;
        int q = 0;
        for (int i = 0; i < n; i++)
        {
            if (phi[i] == -1)
            {
                plcp[i] = 0;
                continue;
            }

            while (s_value[i + l] == s_value[phi[i] + l])
                l++;

            plcp[i] = l;
            q += l;
            l = max(l - 1, (int)0);
        }

        v_cant_substr = n * (n - 1) / 2 - q;
    }
};
```

5.2. Z FUNCTION.

```
// Z[i] is the length of the longest substring
// starting from S[i] which is also a prefix of S.

vi z_function(string s)
{
```

```
    int n = (int)s.length();
    vi z(n);

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

// suff[i] = length of the longest common suffix of s and s[0..i]
vi suffixes(const string &s)
{
    int n = s.length();

    vi suff(n, n);

    for (int i = n - 2, g = n - 1, f; i >= 0; --i)
    {
        if (i > g && suff[i + n - 1 - f] != i - g)
            suff[i] = min(suff[i + n - 1 - f], i - g);
        else
        {
            for (g = min(g, f = i); g >= 0 && s[g] == s[g + n - 1 - f]; --g)
                ;
            suff[i] = f - g;
        }
    }

    return suff;
}
```

5.3. KMP PF.

```
vi prefix_function(string p)
{
    vi pf(p.size());

    pf[0] = 0;
    int k = 0;

    for (int i = 1; i < p.size(); i++)
    {
        while (k > 0 && p[k] != p[i])
            k = pf[k - 1];

        if (p[k] == p[i])
            k++;

        pf[i] = k;
    }

    return pf;
}

vi kmp(string t, string p)
{
    vi result;
    vi pf = prefix_function(p);
    int k = 0;

    for (int i = 0; i < t.size(); i++)
    {
        while (k > 0 && p[k] != t[i])
            k = pf[k - 1];

        if (p[k] == t[i])
            k++;

        if (k == p.size())
        {
            result.push_back(i - (p.size() - 1));
            k = pf[k - 1];
        }
    }

    return result;
}
```

5.4. TRIE.

```
class Trie
{
private:
   int cant_string;
   int cant_string_me;
   int cant_node;
   char value;
   Trie *children[alphabet];

public:
   Trie(char a)
   {
      cant_string = 0;
      cant_node = 1;
      cant_string_me = 0;
      value = a;

      for (int i = 0; i < alphabet; i++)
         children[i] = NULL;
   }

   pair<Trie *, int> search(string s)
   {
      Trie *node = this;
      int i = 0;

      while (i < s.size() && node->children[s[i] - first_char] != NULL)
      {
         node = node->children[s[i] - first_char];

         i++;
      }

      return {node, i};
   }

   void insert(string s)
   {
      int q = s.size() - search(s).second;

      Trie *node = this;

      for (int i = 0; i < s.size(); i++)
      {
         node->cant_node += q;
         if (node->children[s[i] - first_char] == NULL)
         {
            node->children[s[i] - first_char] = new Trie(s[i]);
            q--;
         }

         node = node->children[s[i] - first_char];
         node->cant_string_me++;
      }

      node->cant_string++;
   }

   void eliminate(string s)
   {
      if (!contains(s))
         return;

      Trie *node = this;
      int q = 0;

      for (int i = 0; i < s.size(); i++)
      {
         if (node->children[s[i] - first_char] == NULL)
         {
            node->children[s[i] - first_char] = new Trie(s[i]);
         }

         if (node->children[s[i] - first_char]->cant_string_me == 1)
         {
            node->children[s[i] - first_char] = NULL;

            q = s.size() - i;
            break;
         }

         node = node->children[s[i] - first_char];
         node->cant_string_me--;

         if (i == s.size() - 1)
            node->cant_string--;
      }

      node = this;
```

```
      for (int i = 0; i < s.size() - q + 1; i++)
      {
         node->cant_node -= q;
         node = node->children[s[i] - first_char];
      }
   }

   bool contains(string s)
   {
      auto q = search(s);
      return q.second == s.size() && q.first->cant_string >= 1;
   }

   int cant_words_me() { return cant_string_me; }

   int cant_words() { return cant_string; }

   Trie *get(char a) { return children[a - first_char]; }

   int size() { return cant_node; }
};
```

6. GRAPH

6.1. TOPOLOGICAL SORT.

```
vector<int> topoSort(int V, vector<int> adj[])
{
   vector<int> in(V);
   vector<int> resp;

   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < adj[i].size(); j++)
      {
         in[adj[i][j]]++;
      }
   }

   queue<int> q;

   for (int i = 0; i < V; i++)
   {
      if (in[i] == 0)
         q.push(i);
   }

   while (q.size() != 0)
   {
      int n = q.front();
      q.pop();

      for (int i = 0; i < adj[n].size(); i++)
      {
         in[adj[n][i]]--;

         if (in[adj[n][i]] == 0)
            q.push(adj[n][i]);
      }

      resp.push_back(n);
   }

   return resp;
}
```

6.2. DIJSKTRA.

```
int infinite = (int)1e9;

// O(V^2)
vector<int> dijkstra1(int V, vector<vector<int>> adj[], int S)
{
   vector<int> d;
   d.assign(V, infinite);
   d[S] = 0;

   vector<bool> mask;
```

```
   mask.assign(V, false);

   for (int i = 0; i < V; i++)
   {
      int m = infinite;
      int act = -1;

      for (int j = 0; j < V; j++)
      {
         if (mask[j])
            continue;

         if (m > d[j])
         {
            m = d[j];
            act = j;
         }
      }

      for (int j = 0; j < adj[act].size(); j++)
      {
         if (d[act] + adj[act][j][1] < d[adj[act][j][0]])
         {
            d[adj[act][j][0]] = d[act] + adj[act][j][1];
         }
      }

      mask[act] = true;
   }

   return d;
}
```

```
// O((V+E)log(E))
vi dijkstra2(int V, vii adj[], int S)
{
   vector<int> d;

   d.assign(V, infinite);
   d[S] = 0;

   priority_queue<pair<int, int>> q;
   q.push({d[S], S});

   while (!q.empty())
   {
      int act = q.top().second;
      int m = abs(q.top().first);
      q.pop();

      if (m > d[act])
         continue;

      for (int j = 0; j < adj[act].size(); j++)
      {
         if (d[act] + adj[act][j].second < d[adj[act][j].first])
         {
            d[adj[act][j].first] = d[act] + adj[act][j].second;
            q.push({-d[adj[act][j].first], adj[act][j].first});
         }
      }
   }

   return d;
}
```

6.3. BRIDGE EDGES.

```
vector<bool> visited;
vector<int> t;
vector<int> low;
set<pair<int, int>> bridges;

void dfs_bridges(vector<int> adj[], int n, int p, int q)
{
   t[n] = q;
   low[n] = q++;
   visited[n] = true;

   int j = 0;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!visited[adj[n][i]])
      {
         dfs_bridges(adj, adj[n][i], n, q);
         low[n] = min(low[adj[n][i]], low[n]);
         j++;
```

```
      }
      else if (adj[n][i] != p)
      {
         low[n] = min(t[adj[n][i]], low[n]);
      }
   }

   if (t[n] == low[n] && p != -1)
   {
      bridges.insert({min(n, p), max(n, p)});
   }
}

set<pair<int, int>> bridge_edges(int V, vector<int> adj[])
{
   visited.assign(V, false);
   t.assign(V, -1);
   low.assign(V, -1);
   bridges = set<pair<int, int>>();

   for (int i = 0; i < V; i++)
   {
      if (!visited[i])
      {
         dfs_bridges(adj, i, -1, 1);
      }
   }

   return bridges;
}
```

6.4. PRIM.

```
int spanningTreePrim(int V, vector<vector<int>> adj[])
{

   priority_queue<pair<int, int>> q;

   vector<bool> mask;
   mask.assign(V, false);
   mask[0] = true;

   int cost = 0;

   for (int i = 0; i < adj[0].size(); i++)
   {
      q.push({-adj[0][i][1], adj[0][i][0]});
   }

   while (q.size() != 0)
   {
      auto aux = q.top();
      q.pop();

      int k = aux.second;
      if (mask[k])
         continue;

      mask[k] = true;
      cost += abs(aux.first);

      for (int i = 0; i < adj[k].size(); i++)
      {
         if (!mask[adj[k][i][0]])
         {
            q.push({-adj[k][i][1], adj[k][i][0]});
         }
      }
   }

   return cost;
}
```

6.5. KRUSKAL.

```
class ufds
{
private:
   vector<int> p, rank, sizeSet;
   int disjoinSet;

public:
   ufds(int n)
```

```
   {
      p.assign(n, 0);
      rank.assign(n, 0);
      sizeSet.assign(n, 1);
      disjoinSet = n;
      for (int i = 0; i < n; i++)
      {
         p[i] = i;
      }
   }

   int find(int n)
   {
      if (n == p[n])
         return n;
      p[n] = find(p[n]);
      return p[n];
   }

   bool isSameSet(int i, int j) { return find(i) == find(j); }

   void unionSet(int i, int j)
   {
      if (!isSameSet(i, j))
      {
         disjoinSet--;
         int x = find(i);
         int y = find(j);
         if (rank[x] > rank[y])
         {
            p[y] = x;
            sizeSet[x] += sizeSet[y];
         }
         else
         {
            p[x] = y;
            sizeSet[y] += sizeSet[x];
            if (rank[x] == rank[y])
               rank[y]++;
         }
      }
   }

   int numDisjoinset() { return disjoinSet; }

   int sizeofSet(int i) { return sizeSet[find(i)]; }
};

// Function to find sum of weights of edges of the Minimum Spanning Tree.
int spanningTreeKruskal(int V, vector<vector<int>> adj[])
{
   ufds dsu(V);

   vector<pair<int, pair<int, int>>> a;

   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < adj[i].size(); j++)
      {
         a.push_back({adj[i][j][1], {i, adj[i][j][0]}});
      }
   }

   sort(a.begin(), a.end());

   int cost = 0;

   for (int i = 0; i < a.size(); i++)
   {
      if (!dsu.isSameSet(a[i].second.first, a[i].second.second))
      {
         cost += a[i].first;

         dsu.unionSet(a[i].second.first, a[i].second.second);
      }
   }

   return cost;
}
```

6.6. BELLMAN FORD.

```
int infinite = (int)1e9;

vector<int> bellman_ford(int V, vector<vector<int>> &edges, int S)
{
   vector<int> d;
   d.assign(V, infinite);
```

```
    d[S] = 0;

    for (int i = 0; i < V - 1; i++)
    {
        for (int j = 0; j < edges.size(); j++)
        {
            if (d[edges[j][0]] + edges[j][2] < d[edges[j][1]])
            {
                d[edges[j][1]] = d[edges[j][0]] + edges[j][2];
            }
        }
    }

    for (int j = 0; j < edges.size(); j++)
    {
        if (d[edges[j][0]] + edges[j][2] < d[edges[j][1]])
        {
            vector<int> resp(1);
            resp[0] = -1;

            return resp;
        }
    }

    return d;
}
```

6.7. DINIC.

```
template <typename flow_type>
struct dinic
{
    struct edge
    {
        size_t src, dst, rev;
        flow_type flow, cap;
    };

    int n;
    vector<vector<edge>> adj;

    dinic(int n) : n(n), adj(n), level(n), q(n), it(n) {}

    void add_edge(size_t src, size_t dst, flow_type cap, flow_type rcap = 0)
    {
        adj[src].push_back({src, dst, adj[dst].size(), 0, cap});
        if (src == dst)
            adj[src].back().rev++;
        adj[dst].push_back({dst, src, adj[src].size() - 1, 0, rcap});
    }

    vector<int> level, q, it;

    bool bfs(int source, int sink)
    {
        fill(level.begin(), level.end(), -1);
        for (int qf = level[q[0] = sink] = 0, qb = 1; qf < qb; ++qf)
        {
            sink = q[qf];
            for (edge &e : adj[sink])
            {
                edge &r = adj[e.dst][e.rev];
                if (r.flow < r.cap && level[e.dst] == -1)
                    level[q[qb++] = e.dst] = 1 + level[sink];
            }
        }
        return level[source] != -1;
    }

    flow_type augment(int source, int sink, flow_type flow)
    {
        if (source == sink)
            return flow;
        for (; it[source] != adj[source].size(); ++it[source])
        {
            edge &e = adj[source][it[source]];
            if (e.flow < e.cap && level[e.dst] + 1 == level[source])
            {
                flow_type delta = augment(e.dst, sink, min(flow, e.cap - e.flow));
                if (delta > 0)
                {
                    e.flow += delta;
                    adj[e.dst][e.rev].flow -= delta;
                    return delta;
                }
            }
        }
        return 0;
    }
```

```
   flow_type max_flow(int source, int sink)
   {
      for (int u = 0; u < n; ++u)
         for (edge &e : adj[u])
            e.flow = 0;
      flow_type flow = 0;
      flow_type oo = numeric_limits<flow_type>::max();

      while (bfs(source, sink))
      {
         fill(it.begin(), it.end(), 0);
         for (flow_type f; (f = augment(source, sink, oo)) > 0;)
            flow += f;

      } // level[u] = -1 => source side of min cut
      return flow;
   }
};
```

6.8. CENTROID DESCOMPOSITION.

```
const int MAXN = 2e5 + 5;

vi ady[MAXN];

bitset<MAXN> is_centroid;

int sz[MAXN], ct_par[MAXN];
void centroid_dfs(int node, int parent)
{
   sz[node] = 1;
   for (int &nxt : ady[node])
   {
      if (is_centroid[nxt] || nxt == parent)
         continue;
      centroid_dfs(nxt, node);
      sz[node] += sz[nxt];
   }
}
int get_centroid(int node, int parent, int tree_sz)
{
   for (int nxt : ady[node])
   {
      if (is_centroid[nxt] || nxt == parent)
         continue;
      if (sz[nxt] * 2 > tree_sz)
         return get_centroid(nxt, node, tree_sz);
   }
   return node;
}
void centroid_decomp(int node, int parent = -1)
{
   centroid_dfs(node, -1);
   int tree_sz = sz[node];
   int centroid = get_centroid(node, -1, tree_sz);
   is_centroid[centroid] = 1;
   ct_par[centroid] = parent;

   for (int &child : ady[centroid])
   {
      if (is_centroid[child])
         continue;
      centroid_decomp(child, centroid);
   }
}

void solve()
{
}

int32_t main()
{
   ios_base::sync_with_stdio(0);
   cin.tie(0);

   int t;
   cin >> t;

   for (int i = 0; i < t; i++)
   {
      solve();
   }

   return 0;
}
```

6.9. LOWER BOUND FLOW.

```
template <typename T>
struct dinic
{
   struct edge
   {
      int src, dst;
      T low, cap, flow;
      int rev;
   };

   int n;
   vector<vector<edge>> adj;

   dinic(int n) : n(n), adj(n + 2) {}

   void add_edge(int src, int dst, T low, T cap)
   {
      adj[src].push_back({src, dst, low, cap, 0, (int)adj[dst].size()});
      if (src == dst)
         adj[src].back().rev++;
      adj[dst].push_back({dst, src, 0, 0, 0, (int)adj[src].size() - 1});
   }

   vector<int> level, iter;

   T augment(int u, int t, T cur)
   {
      if (u == t)
         return cur;
      for (int &i = iter[u]; i < (int)adj[u].size(); ++i)
      {
         edge &e = adj[u][i];
         if (e.cap - e.flow > 0 && level[u] > level[e.dst])
         {
            T f = augment(e.dst, t, min(cur, e.cap - e.flow));
            if (f > 0)
            {
               e.flow += f;
               adj[e.dst][e.rev].flow -= f;
               return f;
            }
         }
      }
      return 0;
   }

   int bfs(int s, int t)
   {
      level.assign(n + 2, n + 2);
      level[t] = 0;
      queue<int> Q;
      for (Q.push(t); !Q.empty(); Q.pop())
      {
         int u = Q.front();
         if (u == s)
            break;
         for (edge &e : adj[u])
         {
            edge &erev = adj[e.dst][e.rev];
            if (erev.cap - erev.flow > 0 && level[e.dst] > level[u] + 1)
            {
               Q.push(e.dst);
               level[e.dst] = level[u] + 1;
            }
         }
      }
      return level[s];
   }

   const T oo = numeric_limits<T>::max();

   T max_flow(int source, int sink)
   {
      vector<T> delta(n + 2);

      for (int u = 0; u < n; ++u) // initialize
         for (auto &e : adj[u])
         {
            delta[e.src] -= e.low;
            delta[e.dst] += e.low;
            e.cap -= e.low;
            e.flow = 0;
         }

      T sum = 0;
      int s = n, t = n + 1;

      for (int u = 0; u < n; ++u)
      {
```

```
         if (delta[u] > 0)
         {
            add_edge(s, u, 0, delta[u]);
            sum += delta[u];
         }
         else if (delta[u] < 0)
            add_edge(u, t, 0, -delta[u]);
      }

      add_edge(sink, source, 0, oo);
      T flow = 0;

      while (bfs(s, t) < n + 2)
      {
         iter.assign(n + 2, 0);
         for (T f; (f = augment(s, t, oo)) > 0;)
            flow += f;
      }

      if (flow != sum)
         return -1; // no solution

      for (int u = 0; u < n; ++u)
         for (auto &e : adj[u])
         {
            e.cap += e.low;
            e.flow += e.low;
            edge &erev = adj[e.dst][e.rev];
            erev.cap -= e.low;
            erev.flow -= e.low;
         }

      adj[sink].pop_back();
      adj[source].pop_back();

      while (bfs(source, sink) < n + 2)
      {
         iter.assign(n + 2, 0);
         for (T f; (f = augment(source, sink, oo)) > 0;)
            flow += f;
      } // level[u] == n + 2 ==> s-side

      return flow;
   }
};
```

6.10. FLOYD WARSHALL.

```
int infinite = (int)1e8;

void shortest_distance(vector<vector<int>> &matrix)
{
   for (int k = 0; k < matrix.size(); k++)
   {
      for (int i = 0; i < matrix.size(); i++)
      {
         for (int j = 0; j < matrix[0].size(); j++)
         {
            matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]);
         }
      }
   }
}

void find_path_k(vector<vector<bool>> &matrix, int k)
{
   for (int x = 0; x < k; x++)
   {
      for (int i = 0; i < matrix.size(); i++)
      {
         for (int j = 0; j < matrix[0].size(); j++)
         {
            matrix[i][j] = matrix[i][j] || (matrix[i][x] && matrix[x][j]);
         }
      }
   }
}
```

6.11. DFS BFS.

```
void dfs_g(int n, int c, vi adj[], vector<bool> &visited, vi &cc)
{
   visited[n] = true;
   cc[n] = c;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!visited[adj[n][i]])
         dfs_g(adj[n][i], c, adj, visited, cc);
   }
}

void dfs_t(int n, int p, int d, vi adj[], vi &deep)
{
   deep[n] = d;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (p != adj[n][i])
         dfs_t(adj[n][i], n, d + 1, adj, deep);
   }
}

vi bfs(int node, int n, vi adj[])
{
   vi result(n);
   vector<bool> visited;
   visited.assign(n, false);

   queue<int> q;
   visited[node] = true;

   q.push(node);

   while (q.size() != 0)
   {
      int w = q.front();
      q.pop();

      for (int i = 0; i < adj[w].size(); i++)
      {
         if (!visited[adj[w][i]])
         {
            q.push(adj[w][i]);
            result[adj[w][i]] = result[w] + 1;
            visited[adj[w][i]] = true;
         }
      }
   }

   return result;
}
```

6.12. SCC TARJANS.

```
stack<int> q;
vector<bool> mask;
vector<int> cc_list;

void g_transp(int V, vector<int> adj[], vector<int> new_adj[])
{
   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < adj[i].size(); j++)
      {
         new_adj[adj[i][j]].push_back(i);
      }
   }
}

void dfs_visit(int n, vector<int> adj[], int cc)
{
   mask[n] = true;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!mask[adj[n][i]])
         dfs_visit(adj[n][i], adj, cc);
   }

   if (cc == -1)
      q.push(n);
   else
```

```
   {
      cc_list[n] = cc;
   }
}

void tarjans(int V, vector<int> adj[])
{
   vector<int> new_adj[V];
   g_transp(V, adj, new_adj);

   mask.assign(V, false);
   cc_list.assign(V, -1);

   for (int i = 0; i < V; i++)
   {
      if (mask[i])
         continue;

      dfs_visit(i, adj, -1);
   }

   for (int i = 0; i < V; i++)
      mask[i] = false;

   int ind = 0;

   while (q.size() != 0)
   {
      int act = q.top();

      q.pop();

      if (!mask[act])
      {
         dfs_visit(act, new_adj, ind);
         ind++;
      }
   }
}
```

6.13. MIN COST MAX FLOW.

```
template <typename flow_type, typename cost_type> struct min_cost_max_flow {
      struct edge {
            size_t src, dst, rev;
            flow_type flow, cap;
            cost_type cost;
      };

      int n;
      vector<vector<edge>> adj;

      min_cost_max_flow(int n) : n(n), adj(n), potential(n), dist(n), back(n) {}

      void add_edge(size_t src, size_t dst, flow_type cap, cost_type cost) {
            adj[src].push_back({src, dst, adj[dst].size(), 0, cap, cost});
            if (src == dst)
                  adj[src].back().rev++;
            adj[dst].push_back({dst, src, adj[src].size() - 1, 0, 0, -cost});
      }

      vector<cost_type> potential;

      inline cost_type rcost(const edge &e) {
            return e.cost + potential[e.src] - potential[e.dst];
      }

      void bellman_ford(int source) {
            for (int k = 0; k < n; ++k)
                  for (int u = 0; u < n; ++u)
                        for (edge &e : adj[u])
                              if (e.cap > 0 && rcost(e) < 0)
                                    potential[e.dst] += rcost(e);
      }

      const cost_type oo = numeric_limits<cost_type>::max();

      vector<cost_type> dist;
      vector<edge *> back;

      cost_type dijkstra(int source, int sink) {
            fill(dist.begin(), dist.end(), oo);

            typedef pair<cost_type, int> node;
            priority_queue<node, vector<node>, greater<node>> pq;

            for (pq.push({dist[source] = 0, source}); !pq.empty();) {
```

```
            node p = pq.top();
            pq.pop();

            if (dist[p.second] < p.first)
                continue;
            if (p.second == sink)
                break;

            for (edge &e : adj[p.second])
                if (e.flow < e.cap && dist[e.dst] > dist[e.src] + rcost(e)) {
                    back[e.dst] = &e;
                    pq.push({dist[e.dst] = dist[e.src] + rcost(e), e.dst});
                }
        }

        return dist[sink];
    }

    pair<flow_type, cost_type> max_flow(int source, int sink) {
        flow_type flow = 0;
        cost_type cost = 0;

        for (int u = 0; u < n; ++u)
            for (edge &e : adj[u])
                e.flow = 0;

        potential.assign(n, 0);
        dist.assign(n, 0);
        back.assign(n, nullptr);

        bellman_ford(source); // remove negative costs

        while (dijkstra(source, sink) < oo) {
            for (int u = 0; u < n; ++u)
                if (dist[u] < dist[sink])
                    potential[u] += dist[u] - dist[sink];

            flow_type f = numeric_limits<flow_type>::max();

            for (edge *e = back[sink]; e; e = back[e->src])
                f = min(f, e->cap - e->flow);
            for (edge *e = back[sink]; e; e = back[e->src])
                e->flow += f, adj[e->dst][e->rev].flow -= f;

            flow += f;
            cost += f * (potential[sink] - potential[source]);
        }
        return {flow, cost};
    }
};
```

6.14. ARTICULATION POINT.

```
vector<bool> visited;
vi t;
vi low;
vector<bool> art;

void dfs_art(vi adj[], int n, int p, int q)
{
   t[n] = q;
   low[n] = q++;
   visited[n] = true;

   int j = 0;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!visited[adj[n][i]])
      {
         dfs_art(adj, adj[n][i], n, q);
         low[n] = min(low[adj[n][i]], low[n]);
         j++;

         if (low[adj[n][i]] >= t[n] && p != -1)
         {
            art[n] = true;
         }
      }
      else if (adj[n][i] != p)
      {
         low[n] = min(t[adj[n][i]], low[n]);
      }
   }

   if (p == -1)
   {
      art[n] = j >= 2;
   }
```

```
}

void articulationPoints(int V, vi adj[])
{
   visited.assign(V, false);
   t.assign(V, -1);
   low.assign(V, -1);
   art.assign(V, false);
      for (int i = 0; i < V; i++)
      {
         if (!visited[i])
         {
            dfs_art(adj, i, -1, 1);
         }
      }
   }
```