ICPC Lib

November 1, 2025

Contents

1 Bitmasks

1.1 enumerate_submasks.cpp

```
// Enumerate all submasks of mask in 2^(active bits of mask), which is
    optimal
// in DECREASING order
for (i64 submask = mask; submask > 0; submask = (submask - 1) & mask){
    // do stuff
}
// NOTE: needs to treat 0 submask separately
```

1.2 sos_dp.cpp

```
#include <bits/stdc++.h>
using namespace std;
#define i64 int64_t

vector<i64> sosdp(const vector<i64> A, const i64 B, bool inverse =
    false){
    assert(A.size() == (1ll << B));
    vector<i64> S = A;
    for (i64 i = 0; i < B; i++){
        // introduce possible diff at i
        for (i64 mask = (1ll << B) - 1; mask >= 0; mask--){
            if (mask & (1ll << i)){
                if (inverse){
                    S[mask] -= S[mask ^ (1ll << i)];
                } else {
                    S[mask] += S[mask ^ (1ll << i)];
                }
            }
        }
    }
    return S;
}

vector<i64> not_arr(const vector<i64> A, const i64 B){
    vector<i64> S(1ll << B);
    for (i64 mask = 0; mask < (1ll << B); mask++){
        S[mask] = A[((1ll << B) - 1) ^ mask];
    }
    return S;
}

// SOSP(A)[x] = sum(y supermask of x)A[x] = sum(y submask of ~x)A[~y]
// = SOS(~A)[~x] = ~SOS(~A)
// otherwise if SOSP(S) = A then
// ~SOS(~S) = A ==> SOS(~S) = ~A ==> ~S = SOS^{-1}(~A) ==> S = ~
    SOS^{-1}(~A)
vector<i64> supersetdp(const vector<i64> A, const i64 B, bool inverse
    = false){
    return not_arr(sosdp(not_arr(A, B), B, inverse), B);
}
```

1.3 OR convolution

Seja:

$$C[i] = \sum_{j \cup k = i} A[j] \cdot B[k]$$

Tome a SOSDP de C:

$$SOSDP(C)[i] = \sum_{(j \cup k) \subseteq i} A[j] \cdot B[k] = \sum_{j \subseteq i} A[j] \sum_{k \subseteq i} B[k] = SOSDP(A)[i] \cdot SOSDP(B)[i]$$

Logo: $C = SOSDP^{-1}(SOSDP(A) \cdot SOSDP(B))$.

1.4 AND Convolution

$$C[i] = \sum_{j \cap k = i} A[j] \cdot B[k] = \sum_{j^c \cup k^c = i^c} A[j] \cdot B[k] = \sum_{j \cup k = i^c} A^c[j] \cdot B^c[k]$$

com $A^c[j] = A[j^c]$ e $B^c[k] = B[k^c]$ para todos os conjuntos j, k. Basta aplicar então OR Convolution.

1.5 GCD Convolution

Definição:

$$C[i] = \sum_{\gcd(j,k) = i} A[j] \cdot B[k]$$

Então defina $S(A)[i] = \sum_{i|j} A[j]$ para qualquer array A. Isso pode ser computado pela própria definição em $O(n \log n)$ (somando $A[ki]$ na posição ki para $1 \leq k \leq \lfloor \frac{n}{i} \rfloor$).

$$S(C)[i] = \sum_{i | \gcd(j,k)} A[j]B[k] = \left(\sum_{i|j} A[j] \right) \left(\sum_{i|k} B[k] \right) = S(A)[i] \cdot S(B)[i]$$
$$\implies C = S^{-1}(S(A) \cdot S(B))$$

pois $i \mid \gcd(j,k) \iff i \mid j$ e $i \mid k$.

Para recuperar C podemos usar DP:

$$C[i] = \sum_{i|j} C[j] - \sum_{i|j, i<j} C[j] = S(C)[i] - \sum_{k=2}^{\lfloor n/i \rfloor} C[ki]$$

2 Combinatorics

2.1 Fórmula de Legendre

Para primo p, $v_p(n!) = \sum_{k=1}^{\infty} \left\lfloor \frac{n}{p^k} \right\rfloor$, e $v_p\left(\binom{n}{k}\right) = v_p(n!) - v_p(k!) - v_p((n-k)!)$.

Para $m = \prod p_i^{e_i}$ composto, $v_m(n!) = \min_i \left\lfloor \frac{v_{p_i}(n!)}{e_i} \right\rfloor$.

2.2 Teorema de Lucas

Para p primo e $n = n_k p^k + \ldots + n_1 p + n_0$ e $m = m_k p^k + \ldots + m_1 p + m_0$,

$$\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \mod p$$

2.3 Teorema de Sprague-Grundy e game-theory

Nim: ganha se e só se o XOR dos números da pilha é maior que 0: $a_1 \oplus \ldots \oplus a_n \neq 0$.

Estados terminais tem número de Grundy $n(v) = 0$ e não-terminais tem número de Grundy $n(v) = \text{mex}_{v \to u}\, n(u)$. Assim, um estado é **vencedor** (em um jogo em que quem não tem movimentos perde) se e só se $n(v) \neq 0$. Um único jogo é equivalente a uma **pilha de Nim com (limitados) acréscimos**.

Assim, se temos dois "sub-jogos" independentes S_1 e S_2 então seu número de Grundy é o XOR dos números de Grundy individuais, $n(S_1 + S_2) = n(S_1) \oplus n(S_2)$.

2.4 Stars and bars

Número de soluções inteiras para $x_1 + x_2 + \ldots + x_k = n$ com $x_i \geq 1$ para todo i é $\binom{n-1}{k-1}$ ($n-1$ posições - aquelas entre j e $j+1$ para $j = 1 \ldots n-1$ - para $k-1$ barras separadoras, sendo que cada posição só pode ser ocupada por no máximo uma barra)

Número de soluções inteiras para $x_1 + x_2 + \ldots + x_k = n$ com $x_i \geq 0$ para todo i é $\binom{n+k-1}{k-1}$ (n unidades, $k-1$ barras separadoras, qualquer permutação é válida).

3 count_inversions_merge_sort.cpp

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

// sorts and returns count of inversions. Modifies original array!
int inversions(vector<int>& a, int l, int r){
    if (l == r) return 0;
    const int m = l + (r - l) / 2;
    int total = inversions(a, l, m) + inversions(a, m + 1, r);
    vector<int> left, right;
    for (int i = l; i <= m; i++) left.emplace_back(a[i]);
    for (int i = m + 1; i <= r; i++) right.emplace_back(a[i]);

    int idx = l;
    int pl = 0, pr = 0;
    while (pl < left.size() || pr < right.size()){
        // we only count the inversion pairs from the perspective of
           the smaller
        // element on the right
        // 1 3 5 | 2 4
        // if right[pr] < left[pl] then it is smaller than
           [left[pl]..[left[|left|-1]]]
        // so contributes for left-pl cross-inversions
        if (pr == right.size())
            a[idx++] = left[pl++];
        else if (pl == left.size())
            a[idx++] = right[pr++];
        else {
            if (left[pl] < right[pr])
                a[idx++] = left[pl++];
            else {
                a[idx++] = right[pr++];
                total += left.size() - pl;
            }
        }
    }
    return total;
}
```

4 dsu.cpp

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

struct DSU {
    vector<int> parent, sz;
    vector<vector<int>> elements;

    DSU(int n){
        parent.resize(n);
        iota(parent.begin(), parent.end(), 0);
        sz.assign(n, 1);
        elements.resize(n);
        for (int i = 0; i < n; i++){
            elements[i].emplace_back(i);
        }
    }

    int find(int x){
        if (parent[x] != x){
            parent[x] = find(parent[x]);  // path-compression
        }
        return parent[x];
    }

    void unite(int x, int y){
        x = find(x);
        y = find(y);
        if (x == y){
            return;
        } else if (sz[x] > sz[y]) {
            swap(x, y);
        }
        sz[y] += sz[x];
        for (int z: elements[x]){
            elements[y].emplace_back(z);
        }
        parent[x] = y;
    }
};
```

5 fft.cpp

```
// CODE TAKEN FROM CP-ALGORITHMS
#include <bits/stdc++.h>
using namespace std;

using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
    int n = a.size();

    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
            swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
```

```cpp
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

6 Geometry

6.1 convex_hull.hpp

```cpp
#include "./point.hpp"
#include "./polygon.hpp"
#define i64 int64_t

// counterclockwise convex hull, can include collinear points or not
vector<pt> convex_hull(vector<pt> poly, bool INCLUDE_COLLINEAR){
    // choose bottommost point as pivot
    pt pivot = poly[0];
    const i64 n = poly.size();
    for (i64 i = 1; i < n; i++){
        if (poly[i].y < pivot.y || (poly[i].y == pivot.y && poly[i].x
             < pivot.x)){
            pivot = poly[i];
        }
    }
    sort(poly.begin(), poly.end(),
        [&pivot](pt p, pt q){return polarComp(pivot, p, q);}
    ); // pivot will be always poly[0] after sort
    // since including collinear points, we want to privilege the
       farthest collinear points
    // when coming back (e.g (0,0), (1,0), (2,0), (2,1), (2,2), (1,1))
    if (INCLUDE_COLLINEAR){
        i64 i = n - 1;
        while (i > 0 && orientation(pivot, poly[i], poly.back()) == 0){
            i--;
        }
        reverse(poly.begin() + i + 1, poly.end());
    }
    vector<pt> hull = {pivot};
    for (i64 i = 1; i < n; i++){
        while (
            hull.size() >= 2
            && (
                orientation(hull[hull.size() - 2], hull[hull.size() -
                    1], poly[i]) == 1
                || (!INCLUDE_COLLINEAR && orientation(hull[hull.size()
                    - 2], hull[hull.size() - 1], poly[i]) == 0)
            )
        ){
            // goes clockwise, break orientation
            hull.pop_back();
        }
        hull.emplace_back(poly[i]);
    }
    return hull;
}
```

6.2 point.hpp

```cpp
#include <bits/stdc++.h>
using namespace std;

#define i64 int64_t

struct pt {
    i64 x, y;

    pt(i64 x, i64 y) : x(x), y(y) {}
    pt(const pt& p) : x(p.x), y(p.y) {}

    i64 cross(pt p){
        return x * p.y - y * p.x;
    }
    i64 dot(pt p){
        return x * p.x + y * p.y;
    }
    void operator=(const pt q){
```

```
        x = q.x;
        y = q.y;
    }
};

pt operator-(pt p1, pt p2){
    return pt(p1.x - p2.x, p1.y - p2.y);
}
pt operator+(pt p1, pt p2){
    return pt(p1.x + p2.x, p1.y + p2.y);
}
pt operator*(pt p1, i64 k){
    return pt(k * p1.x, k * p1.y);
}
pt operator*(i64 k, pt p1){
    return pt(k * p1.x, k * p1.y);
}

i64 orientation(pt p, pt q, pt r){
    i64 o = (q - p).cross(r - p);
    if (o > 0){
        return -1; // counterclockwise
    } else if (o < 0){
        return 1; // clockwise
    } else {
        return 0; // collinear
    }
}

i64 ccw(pt p, pt q, pt r) { return orientation(p, q, r) < 0; }


bool is_collinear(pt p, pt q, pt r){
    return (q - p).cross(r - p) == 0;
}

// is p on segment q,r?
bool on_segment(pt p, pt q, pt r){
    // pqr is collinear
    // qp and pr have the same sign/orientation (as p in between q and
       r)
    return (
        is_collinear(p, q, r)
        && ((p - q).dot(r - p) >= 0)
    );
}

// counterclockwise polar sort
bool polarComp(pt pivot, pt p, pt q){
    auto d2 = [](pt a){
        return a.x * a.x + a.y * a.y;
    };
    i64 o = orientation(pivot, p, q);
    if (o == -1) return true;
    else if (o == 1) return false;
    else {
        // collinear => return closest point
        return d2(p - pivot) < d2(q - pivot);
    }
}

// line intersection
bool line_line_intersect(pt a, pt b, pt c, pt d){
    return (
        (b - a).cross(d - c) != 0
        || (b - a).cross(c - a) == 0
    ); // not parallel, or equal lines (ab||cd and abc collinear)
}

// line ab, segment cd
bool line_segment_intersect(pt a, pt b, pt c, pt d){
    // c and d are in different half-planes
    // ==> different orientations
    // OR any triple abc, abd is collinear
    return orientation(c, a, b) * orientation(d, a, b) != 1;
}

// https://cp-algorithms.com/geometry/check-segments-intersection.html
bool segment_segment_intersect(pt a, pt b, pt c, pt d){
    auto inter1d = [](i64 l1, i64 r1, i64 l2, i64 r2){
        if (l1 < r1) swap(l1, r1);
        if (l2 < r2) swap(l2, r2);
        return max(l1, l2) <= min(r1, r2);
    };
    if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0){
        // all four are collinear
        // reduces to 1d interval intersection
        return inter1d(a.x, b.x, c.x, d.x) && inter1d(a.y, b.y, c.y,
            d.y);
    }
    return
        // different half-planes: line ab intersects segment cd
        (orientation(a, b, c) != orientation(a, b, d))
        // different half-planes: line cd intersects segment cd
        && (orientation(c, d, a) != orientation(c, d, b))
    ;
}

// horizontal ray from a, direction +-1, intersects segment cd?
// DOES include intersection at endpoints (cd) by default,
// verified by CSES "Point in Polygon"
bool horizontal_ray_segment_intersect(pt a, pt c, pt d, i64 direction
   = 1){
    if (c.y <= a.y && a.y < d.y){  // different half-planes from ray,
       c below or at line, d above
        // now we need d in the same half-plane as the ray
           (considering line ac)
        // so the intersection happens on the ray and not on the other
           side
        if (direction == 1){
```

```
            return orientation(c, a, d) > 0;
        } else {
            return orientation(c, a, d) < 0;
        }
    } else if (d.y <= a.y && a.y < c.y){
        return horizontal_ray_segment_intersect(a, d, c, direction);
    }
    return false;
}
```

6.3 polygon.hpp

```
#include "./point.hpp"
#include "./triangle.hpp"

// shoelace formula
double area(vector<pt>& p){
    i64 a = 0;
    i64 n = p.size();
    for (i64 i = 0; i < n; i++){
        a += p[i].cross(p[(i + 1) % n]);  // area of OP[i]P[i+1]
    }
    return (double)a / 2.0f;
}

// O(log n) check for boundary/inside convex polygon
// take the first pt (lexicographically) as reference (P0), and sort
    by polar order
// since polygon is convex, then P = U P[0]P[i]P[i+1] for i=1..n-2
// do binary search to find i, and then check for being inside triangle
// check for boundaries separately
const i64 INSIDE = 1;
const i64 BOUNDARY = 0;
const i64 OUTSIDE = -1;
i64 inside_polygon_convex(pt q, vector<pt> p){
    const i64 n = p.size();
    pt pivot = p[0];
    for (pt a: p){
        if (make_pair(a.x, a.y) < make_pair(pivot.x, pivot.y)){
            pivot = a;
        }
    }
    sort(p.begin(), p.end(), [&pivot](pt a, pt b){return
       polarComp(pivot, a, b);});

    i64 l = 1, r = n - 2, i = -1;
    while (l <= r){
        i64 m = l + (r - l) / 2;
        // is in P[0]P[m]P[m+1] iff P[m]P[0]Q is ccw but P[m+1]P[0]Q
           is cw
        if (ccw(p[m], p[0], q)){
            if (!ccw(p[m + 1], p[0], q)){
                i = m;
                break;
            } else {
                l = m + 1;
            }
        } else {
            r = m - 1;
        }
    }
    // inside_triangle includes boudaries
    if (i == -1 || !inside_triangle(q, p[0], p[i], p[i + 1])) return
        OUTSIDE;
    else if (
        on_segment(q, p[i], p[i + 1])
        || (i == 1 && on_segment(q, p[0], p[1]))
        || (i == n - 2 && on_segment(q, p[0], p[n - 1]))
    ){
        return BOUNDARY;
    } else return INSIDE;
}

// Raycasting algorithm, works on all polygons, convex or not.
   Complexity: O(N)
// if the ray intersects a vertex, then it only counts if orientation
   changes
// (so it does not hit "tangentially", but cuts through)
i64 inside_polygon(pt q, vector<pt>& p){
    const i64 n = p.size();
    // draw horizontal ray
    i64 ray_intersections = 0;
    for (i64 i = 0; i < n; i++){
        // p[i]p[i+1]
        if (on_segment(q, p[i], p[(i + 1) % n])){
            return BOUNDARY;
        }
        ray_intersections += horizontal_ray_segment_intersect(q, p[i],
            p[(i + 1) % n]);
    }
    // to be inside: odd exits
    if (ray_intersections % 2 == 1){
        return INSIDE;
    } else {
        return OUTSIDE;
    }
}
```

6.4 triangle.hpp

```
#include "./point.hpp"

// pqr oriented area
// can also be calculated by shoelace
double area(pt p, pt q, pt r){
    return (double)((q - p).cross(r - p)) / 2.0f;
}

bool inside_triangle(pt q, pt p1, pt p2, pt p3){
    i64 abs_area = 0;
    pt p[3] = {p1, p2, p3};
    for (i64 i = 0; i < 3; i++){
```

```
        abs_area += abs((p[i] - q).cross(p[(i + 1) % 3] - q));
    }
    i64 total = abs((p[1] - p[0]).cross(p[2] - p[0]));
    return abs_area == total;
}
```

7 Graphs

7.1 2sat.cpp

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


namespace SAT2 {
    vector<vector<int>> build_graph(const vector<pair<int, int>>
        &clauses, int n){
        vector<vector<int>> g(2 * n);
        for (auto[x, y]: clauses){
            // (x or y) clause = (~x => y) clause and (~y => x) clause
            g[(x + n) % (2 * n)].emplace_back(y);
            g[(y + n) % (2 * n)].emplace_back(x);
        }
        return g;
    }
    vector<vector<int>> transpose(vector<vector<int>> &g){
        vector<vector<int>> gt(g.size());
        for (int u = 0; u < g.size(); u++){
            for (int v: g[u]){
                gt[v].emplace_back(u);
            }
        }
        return gt;
    }

    void forward_dfs(
        const int u, vector<vector<int>> &g, vector<bool> &vis,
            vector<int> &stack
    ){
        vis[u] = true;
        for (int v: g[u]){
            if (!vis[v]){
                forward_dfs(v, g, vis, stack);
            }
        }
        stack.emplace_back(u);
    }

    void backward_dfs(
        const int u, vector<vector<int>> &g, vector<bool> &vis,
        vector<int> &comp, int c = 0
    ){
        vis[u] = true;
        comp[u] = c;
        for (int v: g[u]){
            if (!vis[v]){
                backward_dfs(v, g, vis, comp, c);
            }
        }
    }

    pair<vector<int>, int> scc(const vector<pair<int, int>> &clauses,
        int n){
        vector<vector<int>> g = build_graph(clauses, n);
        vector<vector<int>> gt = transpose(g);

        const int s = g.size();
        vector<int> stack, comp(s);
        vector<bool> vis(s, false);
        for (int u = 0; u < s; u++){
            if (!vis[u]){
                forward_dfs(u, g, vis, stack);
            }
        }
        reverse(stack.begin(), stack.end());
        fill(vis.begin(), vis.end(), false);
        int c = 0;
        for (int u: stack){
            if (!vis[u]){
                backward_dfs(u, gt, vis, comp, c);
                ++c;
            }
        }
        return {comp, c};
    }

    pair<vector<bool>, bool> solve(const vector<pair<int, int>>&
        clauses, int n){
        // run scc (Kosaraju)
        auto[comp, num_comps] = scc(clauses, n);

        // build quotient graph
        vector<vector<int>> g_scc(num_comps);
        for (auto[x, y]: clauses){
            // (x or y) = (~x => y) = (~y => x)
            g_scc[comp[(x + n) % (2 * n)]].emplace_back(comp[y]);
            g_scc[comp[(y + n) % (2 * n)]].emplace_back(comp[x]);
        }

        // build valuation: mark as true the first one that appears
        vector<bool> valuation(n);
        for (int u = 0; u < n; u++){
            if (comp[u] < comp[u + n]){
                valuation[u] = false;
            } else {
                valuation[u] = true;
            }
        }

        // check valuation
```

```
        bool ok = true;
        for (auto[x, y]: clauses){
            bool left = ((x < n) ? valuation[x] : !valuation[x - n]);
            bool right = ((y < n) ? valuation[y] : !valuation[y - n]);
            ok &= left | right;
        }
        return {valuation, ok};
    }
};
```

7.2 bellman_ford.cpp

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

// edges: (a, b, c) = directed edge from a to b with weight c
vector<int> negative_cycle(const vector<tuple<int, int, int>> edges,
    int n){
    // virtual vertex connects to all others with weight 0
    // so ALL negative cycles are reachable!
    // for that effect we just need to set d[u] = 0 for all u
    // (we don't need an actual new vertex)
    vector<int> d(n, 0);
    vector<int> source_list;
    vector<int> prev(n, -1);
    // all paths of size s
    for (int s = 0; s <= n; s++){
        bool any = false;
        for (auto[a, b, c]: edges){
            if (d[a] + c < d[b]){
                any = true;
                prev[b] = a;
                d[b] = d[a] + c;
            }
        }
        if (!any) break;
    }

    // run a (single) new iteration of relaxation
    int target = -1;
    for (auto[a, b, c]: edges){
        if (d[a] + c < d[b]){
            target = b;
            break;
        }
    }
    if (target != -1){
        vector<int> cyc;
        // might be a cycle end or a vertex reachable from a negative
            cycle
        int curr = target;
        vector<bool> vis(n);
        while (!vis[curr]) {
            cyc.emplace_back(curr);
            vis[curr] = true;
            // we might have prev[curr] == curr
            // (in case of self negative edges!)
            curr = prev[curr];
        }
        // now curr is a visited vertex (closed the cycle!)
        cyc.emplace_back(curr);
        reverse(cyc.begin(), cyc.end());
        while (cyc.back() != cyc[0]){
            cyc.pop_back(); // remove vertices reachable from the
                cycle but not in it
        }
        return cyc;
    } else {
        return vector<int>();
    }
}
```

7.3 cycle_detection.cpp

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

#define WHITE 0
#define GRAY 1
#define BLACK 2
bool has_cycle = false;

void dfs(vector<vector<int>> &g, int u, vector<int> &color){
    if (color[u] != WHITE){
        return;
    }
    color[u] = GRAY;
    for (int v: g[u]){
        if (color[v] == GRAY){
            has_cycle = true;
        } else if (color[v] == WHITE){
            dfs(g, u, color);
        }
    }
    color[u] = BLACK;
}
```

7.4 dfs_toposort.cpp

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

namespace Graph {
    int c;

    void dfs(
        int u, vector<vector<int>> &G, vector<bool> &vis, vector<int>
            &order
    ){
        vis[u] = true;
        for (int v: G[u]){
            if (!vis[v]){
```

```
                dfs(v, G, vis, order);
            }
        }
        order[u] = c;
        --c;
    }

    vector<int> toposort(vector<vector<int>> &G){
        c = G.size() - 1;
        const int n = G.size();
        vector<bool> vis(n, false);
        vector<int> order(n);
        for (int u = 0; u < n; u++){
            if (!vis[u]){
                dfs(u, G, vis, order);
            }
        }
        // validate order
        for (int u = 0; u < n; u++){
            for (int v: G[u]){
                if (order[u] >= order[v]) {
                    return vector<int>();
                }
            }
        }
        return order;
    }
}
```

7.5 dijkstra.cpp

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

const int INF = (int)2e9;

// directed edges are pairs (weight, endpoint)
void dijkstra(vector<vector<pair<int, int>>> &adj, int dis[], int s)
{
    for (int i = 0; i < (int)adj.size(); i++) dis[i] = INF;

    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>> > pq;
    pq.emplace(0, s);
    dis[s] = 0;
    while (!pq.empty())
    {
        auto [dis_u, u] = pq.top();
        pq.pop();
        if (dis_u > dis[u])
            continue; // skip suboptimal
        // since dis[u] is always updated when better path is found,
        // dis[u] <= d for each [d, u] state on the priority queue

        for (auto [w, v] : adj[u])
        {
            if (dis[u] + w < dis[v])
            {
                dis[v] = dis[u] + w;
                pq.emplace(dis[v], v);
                // sorting by dis[v] is the same as sorting by the
                    weights w[u, v]
                // since dis[v] - w(u, v) = dis[u] for all v (constant
                    difference)
            }
        }
    }
}
```

7.6 dinic.cpp

```
// Tested on CSES - Police Chases
#include <bits/stdc++.h>
using namespace std;
#define i64 int64_t

/*
Dinic (overview):
// make residual graph (edges and backedges)
// then on each iteration
// do bfs to make layer graph
// do dfs to find blocking flow
*/

const i64 INF = (i64)2e18;
struct Edge {
    i64 u, v, c, f;
    i64 cap() const { return c - f; }
    Edge(i64 from, i64 to, i64 cap, i64 flow) : u(from), v(to),
        c(cap), f(flow) {}
};

// for edge indices 0..m-1 in original network
// we represent forward edge i as 2 * i and backedge as 2*i + 1
// thus forward and backedge of indices i1, i2 relate in this way:
// i1 ^ 1 = i2, i2 ^ 1 = i1
struct Dinic {
    i64 n, s, t;
    vector<vector<i64>> adj;  // maps vertex to edge indices
    vector<Edge> edges;
    // performance optimization to push flow possibly more than once
    // after each bfs/layer graph build
    vector<i64> cur_edge_ptr;
    // performance optimization to avoid looking at blocked nodes
    vector<i64> blocked;
    vector<i64> level;
    queue<i64> q;

    void add_edge(i64 u, i64 v, i64 c){
        // set flow at c so effective cap of back edge starts at 0
        Edge forward(u, v, c, 0), backward(v, u, c, c);
```

```
        adj[u].emplace_back(edges.size());
        edges.emplace_back(forward);
        adj[v].emplace_back(edges.size());
        edges.emplace_back(backward);
    }

    Dinic(const i64 n, i64 s, i64 t) : n(n), s(s), t(t){
        adj.resize(n);
        level.resize(n);
        cur_edge_ptr.assign(n, 0);
        blocked.assign(n, false);
        edges.reserve(2 * n);
    }

    // level vector --> implicit layer graph (by takings edges where d increases by 1)
    // pass by ref to optimize copies
    void layer_graph(){
        fill(level.begin(), level.end(), INF);
        level[s] = 0;
        q.emplace(s);
        while (!q.empty()){
            i64 u = q.front();
            q.pop();
            for (i64 id: adj[u]){
                const Edge& e = edges[id];  // u -> v
                if (e.cap() == 0) continue;  // edge does not exist
                i64 v = e.v;
                if (level[v] == INF){
                    level[v] = level[u] + 1;
                    q.emplace(v);
                }
            }
        }
        while (!q.empty()) q.pop();
    }

    // push_flow on layer dag of residual graph
    i64 push_flow(i64 u, vector<i64>& blocked, vector<i64>& level, i64 dF){
        if (u == t){
            return dF;  // already at sink
        }
        bool all_neighbors_blocked = true;  // performance optimization
        i64 total_pushed = 0;
        for (; cur_edge_ptr[u] < (i64)adj[u].size() && dF > 0; cur_edge_ptr[u]++){
            i64 id = adj[u][cur_edge_ptr[u]];
            // edges[id] is forward edge (relative to u): u -> v
            i64 v = edges[id].v;
            if (level[v] != level[u] + 1) continue;  // not dag edge
            if (edges[id].cap() == 0 || blocked[v]) continue;  // edge does not exist
            i64 pushed = push_flow(v, blocked, level,
                min(edges[id].cap(), dF));
            edges[id].f += pushed;
            edges[id ^ 1].f -= pushed;
            total_pushed += pushed;
            dF -= pushed;

            if (edges[id].cap() > 0 && !blocked[v]) {  // edge still valid
                all_neighbors_blocked = false;
            }
        }
        if (all_neighbors_blocked) blocked[u] = true;
        return total_pushed;
    }

    i64 maxflow(){
        while (true){
            layer_graph();
            if (level[t] == INF) {
                break;
            }  // no more augmenting paths
            // can't assume anyone is blocked (reuse block vector), since we might try to unblock by undoing flow
            fill(cur_edge_ptr.begin(), cur_edge_ptr.end(), 0);  // reset edge ptr
            fill(blocked.begin(), blocked.end(), false);  // reset blocked
            while (push_flow(s, blocked, level, INF)) {}
        }
        i64 maxFlow = 0;
        for (i64 id: adj[s]) {
            maxFlow += edges[id].f;
        }
        return maxFlow;
    }

    // assumes dinic was called first
    vector<i64> mincut(){
        layer_graph();
        vector<i64> partition_num(n);
        for (i64 u = 0; u < n; u++){
            if (level[u] < INF){
                partition_num[u] = 0;
            } else {
                partition_num[u] = 1;
            }
        }
        return partition_num;
    }
};
```

7.7 dinic_with_scaling.cpp

```
// Tested on CSES - Police Chases
#include <bits/stdc++.h>
using namespace std;
#define i64 int64_t
```

```cpp
// Same as dinic but with edge scaling
// Complexity is O(VE log U) where U = max edge capacity
// WE ONLY INCLUDE WHAT CHANGED
const i64 INF = (i64)(1ll << 60);  // best to use a power of 2

// level vector --> implicit layer graph (by takings edges where d
    increases by 1)
// ------------------------------------
// NOTE: when using dinic with scaling for mincut, do layer graph with
    default lim of 1
// ------------------------------------
void layer_graph(i64 lim = 1){
    fill(level.begin(), level.end(), INF);
    level[s] = 0;
    q.emplace(s);
    while (!q.empty()){
        i64 u = q.front();
        q.pop();
        for (i64 id: adj[u]){
            const Edge& e = edges[id];  // u -> v
            if (e.cap() < lim) continue;  // edge does not exist
            i64 v = e.v;
            if (level[v] == INF){
                level[v] = level[u] + 1;
                q.emplace(v);
            }
        }
    }
    while (!q.empty()) q.pop();
}

// push_flow on layer dag of residual graph
bool can_push_flow(i64 u, vector<i64>& level, i64 flow){
    if (u == t){
        return true;  // already at sink
    }
    for (; cur_edge_ptr[u] < (i64)adj[u].size(); cur_edge_ptr[u]++){
        i64 id = adj[u][cur_edge_ptr[u]];
        // edges[id] is forward edge (relative to u): u -> v
        i64 v = edges[id].v;
        if (level[v] != level[u] + 1) continue;  // not dag edge
        if (edges[id].cap() < flow) continue;  // edge does not exist
        i64 pushed = can_push_flow(v, level, flow);
        if (pushed){
            edges[id].f += flow;
            edges[id ^ 1].f -= flow;
            return true;
        }
    }
    return false;
}

i64 maxflow(){
    for (i64 lim = INF; lim >= 1; ){
        layer_graph(lim);
        if (level[t] == INF) {
            lim >>= 1;
            continue;
        }  // no more augmenting paths
        // can't assume anyone is blocked (reuse block vector), since
            we might try to unblock by undoing flow
        fill(cur_edge_ptr.begin(), cur_edge_ptr.end(), 0);  // reset
            edge ptr
        while (can_push_flow(s, level, lim)) {}
    }
    i64 maxFlow = 0;
    for (i64 id: adj[s]) {
        maxFlow += edges[id].f;
    }
    return maxFlow;
}
```

7.8 floyd_warshall.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

const int INF = (int)2e9;  // might change depending on the problem

vector<vector<int>> floydWarshall(const vector<tuple<int, int, int>>&
    edges, int n){
    vector<vector<int>> d(n, vector<int>(n, INF));
    for (int i = 0; i < n; i++){
        d[i][i] = 0;
    }
    for (auto[a, b, w]: edges){
        d[a][b] = min(d[a][b], w);
    }
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (d[i][k] < INF && d[k][j] < INF)
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }
    return d;
}
```

7.9 kosaraju.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> transpose(const vector<vector<int>> &g){
    vector<vector<int>> gt(g.size());
    for (int u = 0; u < g.size(); u++){
        for (int v: g[u]){
            gt[v].emplace_back(u);
```

```
        }
    }
    return gt;
}

void forward_dfs(
    const int u, const vector<vector<int>> &g, vector<bool> &vis,
        vector<int> &stack
){
    vis[u] = true;
    for (int v: g[u]){
        if (!vis[v]){
            forward_dfs(v, g, vis, stack);
        }
    }
    stack.emplace_back(u);
}

void backward_dfs(
    const int u, const vector<vector<int>> &g, vector<bool> &vis,
    vector<int> &comp, int c = 0
){
    vis[u] = true;
    comp[u] = c;
    for (int v: g[u]){
        if (!vis[v]){
            backward_dfs(v, g, vis, comp, c);
        }
    }
}

pair<vector<int>, int> scc(const vector<vector<int>> &g, int n){
    vector<vector<int>> gt = transpose(g);

    const int s = g.size();
    vector<int> stack, comp(s);
    vector<bool> vis(s, false);
    for (int u = 0; u < s; u++){
        if (!vis[u]){
            forward_dfs(u, g, vis, stack);
        }
    }
    reverse(stack.begin(), stack.end());
    fill(vis.begin(), vis.end(), false);
    int c = 0;
    for (int u: stack){
        if (!vis[u]){
            backward_dfs(u, gt, vis, comp, c);
            ++c;
        }
    }
    return {comp, c};
}
```

7.10 lexicographical_minimal_toposort.cpp

```
#include <bits/stdc++.h>
using namespace std;
#define i64 int64_t

// toposort with bfs, returns the lexicographically smallest
   topological order
// Runs in O(n log n) because of sorting
// Returns boolean indicating if topo order exists, and the order in
   case it does
pair<bool, vector<i64>> lexicographical_toposort(const
   vector<vector<i64>>& dag){
    const i64 n = dag.size();
    // dependencies_met[i]: how many j such that j->i
    // have received their order number
    vector<i64> dependencies_met(n, 0),
                in_degree(n, 0),
                order(n, -1);
    for (i64 u = 0; u < n; u++){
        for (i64 v: dag[u]){
            ++in_degree[v];
        }
    }
    priority_queue<i64, vector<i64>, greater<i64>> q;
    // min pq guarentees lexicographical order and topo order
    // by only pushing when all dependencies are met
    for (i64 u = 0; u < n; u++){
        if (!in_degree[u]) q.emplace(u);
    }
    i64 c = 0;
    while (!q.empty()){
        i64 u = q.top();
        order[u] = c++;
        q.pop();
        for (i64 v: dag[u]){
            ++dependencies_met[v];
            if (dependencies_met[v] == in_degree[v]){
                q.emplace(v);
            }
        }
    }
    for (i64 u = 0; u < n; u++){
        // circular dependencies exist
        if (order[u] == -1) return {false, {}};
    }
    return {true, order};
}
```

7.11 tarjan.cpp

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

const int INF = (int)2e9;

namespace Tarjan {
    vector<int> lowlink, t_in;
```

```
    vector<pair<int, int>> dfs_tree_edges;
    int timer;

    void dfs(int u, const vector<vector<int>> &g, vector<bool> &vis,
            int parent = -1){
        vis[u] = true;
        t_in[u] = lowlink[u] = timer++;
        for (int to: g[u]){
            if (to == parent) continue;
            else if (vis[to]){
                // back edge on DFS tree
                lowlink[u] = min(t_in[to], lowlink[u]);
            } else {
                dfs(to, g, vis, u);
                dfs_tree_edges.emplace_back(u, to);
                lowlink[u] = min(lowlink[to], lowlink[u]);
            }
        }
    }

    // lowlink(u) = min(t_in(u), min_{p->u}t_in(u), min_{u->to em
        tree}lowlink(to))
    vector<vector<pair<int, int>>> dfs_tree_bridges(const
        vector<vector<int>> &g) {
        const int n = g.size();
        vector<bool> vis(n, false);
        lowlink = vector<int>(n, INF);
        t_in = vector<int>(n, INF);
        dfs_tree_edges.clear();
        timer = 0;

        for (int u = 0; u < n; u++){
            if (!vis[u]){
                dfs(u, g, vis);
            }
        }

        vector<vector<pair<int, int>>> h(n);
        for (auto[v, to]: dfs_tree_edges){
            if (lowlink[to] > t_in[v]){
                h[v].emplace_back(to, 1);
            } else {
                h[v].emplace_back(to, 0);
            }
        }
        return h;
    }
}
```

8 Number Theory

8.1 fastexp.cpp

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

const int MOD = (int)1e9 + 7;  // NOTE: MIGHT CHANGE

int bexp(int a, int p){
    if (p == 0){
        return 1;
    } else {
        int m = bexp(a, p / 2);
        if (p % 2 == 1){
            return (((m * m) % MOD) * a) % MOD;
        } else {
            return (m * m) % MOD;
        }
    }
}

int inv(int a){
    return bexp(a, MOD - 2);
}
```

8.2 spf.cpp

```
/*
Given N, calculates the smallest prime factor (spf) that divides K for
    each 1 <= K <= N
If K is prime, that prime factor will be K itself
*/

#include <bits/stdc++.h>
using namespace std;

void spf(int N, int ans[]){
  // ans is a N+1 sized array
  for (int i = 0; i <= N; i++){
    ans[i] = i;
  }

  int is_prime[N+1]; // 0 means is prime, 1 means is composite
  memset(is_prime, N + 1, 0);
  is_prime[0] = 1; is_prime[1] = 1;
  for (int i = 2; i <= N; i++){
    if (is_prime[i] == 0){
      for (int mul = i; mul <= N; mul += i){
        ans[mul] = min(ans[mul], i);
        // last value will be the smallest divisor of mul greater than
            1 (which is the spf)

      }
    }
  }

}
```

9 Range Queries

9.1 bit.cpp

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

vector<int> BIT(const vector<int> &a){
    int n = a.size();
    vector<int> B(n + 1, 0);

    for (int i = 0; i < n; i++){
        update(B, i, a[i]);  // sum a[i] to position i+1 of the BIT
    }
    return B;
}

int get(const vector<int> &BIT, int i){
    int s = 0;
    while (i >= 0){
        s += BIT[i];
        // i & -i == 2^lsb(i)
        i = i ^ (i & (-i));
    }
    return s;
}

void update(vector<int> &BIT, int i, int x){
    const int n = BIT.size() - 1;
    while (i <= n){
        BIT[i] += x;
        i += i & (-i);
    }
}
```

9.2 segtree.cpp

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

// default seg tree (in this example, a Max Seg)
struct SegTree {
    vector<int> seg;
    vector<int> a;
    int n;

    SegTree(const vector<int>& a) : a(a), n(a.size()) {
        seg.resize(4 * n);
        build(0, n - 1, 0);
    }

    inline int merge(int a, int b){
        return max(a, b);  // <--- CHANGE HERE
    }

    void build(int l, int r, int node){
        if (l == r) seg[node] = a[l];
        else {
            const int m = l + (r - l) / 2;
            build(l, m, 2 * node + 1);
            build(m + 1, r, 2 * node + 2);
            seg[node] = merge(seg[2 * node + 1], seg[2 * node + 2]);
        }
    }

    int query(int l, int r){
        return _query(l, r, 0, n - 1, 0);
    }

    void update(int p, int x){
        _update(p, x, 0, n - 1, 0);
    }

    int _query(int l, int r, int tl, int tr, int node){
        if (l == tl && r == tr){
            return seg[node];
        } else {
            int tm = tl + (tr - tl) / 2;
            if (r <= tm){
                return _query(l, r, tl, tm, 2 * node + 1);
            } else if (l > tm){
                return _query(l, r, tm + 1, tr, 2 * node + 2);
            } else {
                return merge(
                    _query(l, tm, tl, tm, 2 * node + 1),
                    _query(tm + 1, r, tm + 1, tr, 2 * node + 2)
                );
            }
        }
    }

    void _update(int p, int x, int tl, int tr, int node){
        if (tl == p && tr == p){
            seg[node] = x;
            a[p] = x;
        } else {
            int tm = tl + (tr - tl) / 2;
            if (p <= tm){
                _update(p, x, tl, tm, 2 * node + 1);
            } else {
                _update(p, x, tm + 1, tr, 2 * node + 2);
            }
            seg[node] = merge(seg[2 * node + 1], seg[2 * node + 2]);
        }
    }
};
```

10 Strategy

10.1 Debugging:

- **Nem sempre o bug está na parte mais complexa do código: leia tudo, até o template**
- Verifique o valor do MOD, do INF e das constantes, dos tamanhos dos arrays.
- Criar corner cases ($n = 0, 1$, casos especiais que forçam alguma coisa, etc) e casos pequenos
- Colocar asserts para detectar índices out-of-bounds ou outros problemas
- **Minimizar estado global, separar lógica em funções menores**

10.2 Início (primeira 1h30)

- Ler questões
- Passar no máximo 10-15 min por questão
- Implementar apenas as fáceis
  - Priorizar primeiro aquelas que são simples e rápidas de implementar (em detrimento por ex. de uma aplicação trivial de um algoritmo longo, como uma SegTree ou fluxo)
- Fase mais individual
- Priorizar para implementação aquela pessoa que tem a ideia mais clara da solução

10.3 Meio

- Ler o restante da prova
- Olhar os standings para selecionar a próxima questão
- Analisar mais profundamente no papel para conseguir passar as médias/difíceis
- Evitar ficar três pessoas em uma mesma questão

10.4 Final (última 1h30)

- Foco do time todo em uma ou duas questões mais prováveis de passarem

11 Strings

11.1 rabin_karp.cpp

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

#define i64 int64_t

vector<i64> rollingHash(const vector<int>& v, i64 p, i64 m){
    const i64 n = v.size();
    vector<i64> prefH(n);
    vector<i64> p_pow(n);
    // prefH = hash(v[0...i]) = sum(j<=i) v[j] * p^j mod m
    p_pow[0] = 1;
    prefH[0] = v[0] % m;
    for (i64 i = 1; i < n; i++){
        p_pow[i] = (p_pow[i - 1] * p) % m;
        prefH[i] = (v[i] * p_pow[i] + prefH[i - 1]) % m;
    }
    return prefH;
}

// lowercase
vector<i64> rollingHash(const string& s){
    vector<int> s_repr(s.size());
    for (i64 i = 0; i < s.size(); i++){
        s_repr[i] = s[i] - 'a';
    }
    return rollingHash(s_repr, 31, (i64)1e9 + 7);
}

i64 bexp(const i64 a, const i64 p, const i64 mod){
    if (p == 0){
        return 1;
    } else {
        const i64 b = bexp(a, p / 2, mod);
        if (p % 2 == 0){
            return (b * b) % mod;
        } else {
            return (a * ((b * b) % mod)) % mod;
        }
    }
}

vector<i64> prime_powers(const i64 n, const i64 p = 31, const i64 m =
    (i64)1e9 + 7){
    vector<i64> p_pow(2 * n + 1);
    p_pow[n] = 1;
    p_pow[n - 1] = bexp(p, m - 2, m);   // fermat little theorem
    for (i64 i = 1; i <= n; i++){
        p_pow[n + i] = (p_pow[n + i - 1] * p) % m;
    }
    for (i64 i = 2; i <= n; i++){
        p_pow[n - i] = (p_pow[n - i + 1] * p_pow[n - 1]) % m;
```

```
    }
    return p_pow;
}

vector<i64> rabinKarpMatch(const string& s, const string& t){
    vector<i64> prefix_s = rollingHash(s), prefix_t = rollingHash(t);

    const i64 mod = (i64)1e9 + 7, p = 31;
    const i64 HS = prefix_s.back();  // complete hash of s
    // precompute prime powers mod M
    const vector<i64> p_pow = prime_powers(t.size(), p, mod);

    vector<i64> matches;  // starting points of matches

    const i64 offset = t.size();
    for (i64 i = 0; i + s.size() <= t.size(); i++){
        // hash of t[i...i + |s| - 1] == (hash(t[0...i + |s| - 1]) -
            hash(t[0...i - 1])) * p^-i
        const i64 h = ((prefix_t[i + s.size() - 1] - (i > 0 ?
            prefix_t[i - 1] : 0) + mod) * p_pow[offset - i]) % mod;
        if (h != HS){
            continue;
        }
        // compare
        bool ok = true;
        for (i64 j = 0; (j < s.size()) && ok; j++){
            ok = ok && (s[j] == t[i + j]);
        }
        if (ok) matches.emplace_back(i);
    }
    return matches;
}
```

11.2 suffix_array.cpp

```
// TESTED on first suffix array problem of Codeforces EDU
#include <bits/stdc++.h>
using namespace std;

#define ALPHABET 256

void csort(vector<int>& p, const vector<int>& classes, int offset){
    const int n = p.size();
    vector<int> cnt(max(ALPHABET, n), 0), ptemp(n);
    for (int i = 0; i < n; i++){
        ++cnt[classes[i]];
    }
    for (int i = 1; i < max(ALPHABET, n); i++){
        cnt[i] += cnt[i - 1];
    }
    // distribute on blocks in reverse order to make sorting stable
    // distribute according to order of k-th element (k=offset)
    for (int i = n - 1; i >= 0; i--){
        int offseted_pos = (p[i] + offset) % n;
        ptemp[--cnt[classes[offseted_pos]]] = p[i];
    }
    swap(ptemp, p);
}

pair<vector<int>, vector<vector<int>>> suffix_array(const string& s){
    int lg = 0;
    string t = s;
    t.push_back('$');
    int n = t.size();
    while ((1ll << lg) < n) ++lg;  // now 2^lg >= n
    vector<int> p(n);
    vector<vector<int>> c(lg + 1, vector<int>(n));
    for (int i = 0; i < n; i++){
        p[i] = i;
        c[0][i] = (int)t[i];
    }
    csort(p, c[0], 0);
    // check offset of size 2^k
    for (int k = 0; k <= lg; k++){
        csort(p, c[k], (1ll << k));  // sort by second, stably
        csort(p, c[k], 0);  // sort by first, stably
        if (k == lg) break;
        // redo equivalence classes
        c[k + 1][p[0]] = 0;  // first element receives class=0
        int cls = 0;
        for (int i = 1; i < n; i++){
            if (
                (c[k][p[i - 1]] < c[k][p[i]])
                || (c[k][(p[i - 1] + (1ll << k)) % n] < c[k][(p[i] +
                    (1ll << k)) % n])
            ){
                ++cls;
            }
            c[k + 1][p[i]] = cls;
        }
    }
    return {p, c};
}

int lcp(int i, int j, const vector<vector<int>>& cls){
    const int n = cls[0].size();
    int lg = 0;
    while ((1ll << lg) < n) ++lg;  // now 2^lg >= n
    int ans = 0;
    for (int k = lg; k >= 0; k--){
        if (cls[k][i] == cls[k][j]){
            i = (i + (1ll << k)) % n;
            j = (j + (1ll << k)) % n;
            ans += (1ll << k);
        }
    }
    return ans;
}

// LCS in O(N log N)
// TESTED on second suffix array problem of Step 2 of Codeforces EDU
tuple<int, int, int> lcs(string s, string t){
```

```
    auto [p, c] = suffix_array(s + "#" + t);
    // NOTE: the sentinel must be different to avoid having cyclic
        shifts breaking stuff
    const int n = s.size(), m = t.size();
    const int sz = n + 1 + m + 1;  // middle # and final $
    vector<int> lcp_array(sz - 1);
    for (int i = 0; i + 1 < sz; i++) lcp_array[i] = lcp(p[i], p[i +
        1], c);
    int lcs_sz = 0, s_idx = -1, t_idx = -1;
    for (int i = 0; i + 1 < sz; i++){
        int left = p[i], right = p[i + 1];
        if (lcp_array[i] > lcs_sz && min(left, right) < n && max(left,
            right) > n){
            // p[i] in s, p[i + 1] in t or vice-versa
            lcs_sz = lcp_array[i];
            s_idx = min(left, right);
            t_idx = max(left, right);
        }
    }
    return {lcs_sz, s_idx, t_idx};
}
```

11.3 trie.cpp

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

template<int K>  // sum of characters in the set
struct Trie {
    int children[K][26];
    int cnt[K];  // cnt of each word
    int ptr = 1;

    Trie(){
        for (int i = 0; i < K; i++){
            for (int j = 0; j < 26; j++){
                children[i][j] = -1;
            }
            cnt[i] = 0;
        }
    }

    void insert(string& u, int i=0, int node=0){
        int idx = u[i] - 'a';
        if (children[node][idx] == -1){
            // create node
            children[node][idx] = ptr++;
        }
        if (i + 1 < u.size()){
            insert(u, i + 1, children[node][idx]);
        } else {
            ++cnt[children[node][idx]];
        }
    }

    int search(string &u, int i=0, int node=0){
        if (i == u.size()){
            return cnt[node];
        } else {
            int idx = u[i] - 'a';
            if (children[node][idx] == -1){
                return 0;
            } else {
                return search(u, i + 1, children[node][idx]);
            }
        }
    }

    bool isleaf(int node){
        bool ok = true;
        for (int i = 0; i < 26; i++){
            ok = ok && children[node][i] == -1;
        }
        return ok;
    }

    void remove(string &u, int i=0, int node=0){
        int idx = u[i] - 'a';
        if (children[node][idx] != -1){
            remove(u, i + 1, children[node][idx]);
            if (isleaf(children[node][idx]) &&
                cnt[children[node][idx]] == 0){
                children[node][idx] = -1;
            }
        } else {
            // reached leaf
            cnt[node] = max(cnt[node] - 1, 0);
        }
    }
};
```

12 template.cpp

```
#define TESTCASES

#include <bits/stdc++.h>
using namespace std;
#define fastio
    ios_base::sync_with_stdio(false);cin.tie(NULL);cout.tie(NULL)
#define endl '\n'

#define i64 int64_t
#define i128 __int128
#define all(x) begin(x),end(x)
#define print(msg, v) cerr << msg; for(auto it = v.begin(); it !=
    v.end(); it++){cerr << *it << " ";} cerr << endl;
#define printgraph(msg, G) cerr << msg << endl; \
    for (i64 u = 0; u < G.size(); u++) { \
        cerr << "G[" << u << "]="; \
        for (i64 v: G[u]) { \
```

```
            cerr << v << " "; \
        } \
        cerr << endl; \
    }

template <typename T, typename U>
ostream& operator<< (ostream& out, pair<T, U> x)
{
    out << x.first << " " << x.second;
    return out;
}
template <typename T>
ostream& operator<< (ostream& out, vector<T> v)
{
    for (const auto& x: v){
        out << x << " ";
    }
    return out;
}

template <typename T>
using min_pq = priority_queue<T, vector<T>, greater<T>>;
template <typename T>
using max_pq = priority_queue<T>;

void solve(){}

signed main(){
    fastio;
    int t = 1;
#ifdef TESTCASES
    cin >> t;
#endif
    while (t--){
        solve();
    }
}
```

13 Trees

13.1 binary_lifting.cpp

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


const int N = 2e5, LOGN = 20;
int lift[LOGN + 1][N];

void binary_lifting(const vector<vector<int>> &forest){
    const int n = forest.size();
    vector<int> parent(n, -1);
    for (int u = 0; u < n; u++){
        for (int v: forest[u]){
            parent[v] = u;
        }
    }
    for (int u = 0; u < n; u++){
        lift[0][u] = parent[u];
    }
    for (int k = 1; k <= LOGN; k++){
        for (int u = 0; u < n; u++){
            lift[k][u] = lift[k - 1][lift[k - 1][u]];
        }
    }
}

int jump(int u, int k){
    for (int i = 0; i <= LOGN; i++){
        if ((k >> i) & 1){
            u = lift[i][u];
        }
    }
    return u;
}
```

13.2 centroid.cpp

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

namespace CentroidDecomposition {
    int subtree_size(int u, const vector<vector<int>>& adj,
        vector<int>& s, vector<int>& blocked, int p = -1){
        s[u] = 1;
        for (int v: adj[u]){
            if (v == p) continue;
            if (blocked[v]) continue;
            s[u] += subtree_size(v, adj, s, blocked, u);
        }
        return s[u];
    }

    // find centroid of subtree
    int find_centroid(int u, const vector<vector<int>>& adj,
        vector<int>& s, vector<int>& blocked, int total_size, int p =
        -1){
        for (int v: adj[u]){
            if (v == p) continue;
            if (blocked[v]) continue;
            if (2 * s[v] >= total_size){
                return find_centroid(v, adj, s, blocked, total_size,
                    u); // there can be only one such vertex
            }
        }
        return u;
    }

    // total_size = size of subtree rooted at root (considering
        deletions)
    // @return centroid subtree root
    int _build_centroid_tree(
```

```
        const vector<vector<int>>& adj,
        vector<int>& blocked,
        vector<vector<int>>& centroid_adj,
        vector<int>& sz,
        int total_size,
        int root = 0
    ){
        subtree_size(root, adj, sz, blocked);
        int c = find_centroid(root, adj, sz, blocked, total_size);
        blocked[c] = true;
        for (int v: adj[c]){
            if (blocked[v]) continue;
            // build centroid subtree on partition and get its root
            int c2 = _build_centroid_tree(adj, blocked, centroid_adj,
                sz, sz[v], v);
            centroid_adj[c].emplace_back(c2);
        }
        return c;
    }

    // @return pair (centroid tree root, directed centroid tree)
    pair<int, vector<vector<int>>> centroid_tree(const
        vector<vector<int>>& adj){
        const int n = adj.size();
        vector<int> blocked(n, false), sz(n);
        vector<vector<int>> centroid_adj(n);
        int master_centroid = _build_centroid_tree(
            adj, blocked, centroid_adj, sz, n
        );
        return {master_centroid, centroid_adj};
    }
}
```

13.3 hld_no_range_update.cpp

```
// HLD implementation: no range updates
// Tested on CSES Path Queries (https://cses.fi/problemset/task/1138)
// Query: sum of vertices from root to v
// Update: set value of v to x

// uses multiple seg trees. Marginally faster than a single segtree
// see hld_with_range_update for better descriptions

#include <bits/stdc++.h>
using namespace std;
#define i64 int64_t

using tree = vector<vector<i64>>;
struct HLD {
    vector<i64> heavy;
    vector<i64> sz;
    vector<i64> head;
    vector<i64> pos;
    vector<i64> parent;
    // NOTE: SegTree is the segment tree implementation
    // NOTE: SegTree needs default constructor
    vector<SegTree> segs;
    vector<i64> a;
    vector<i64> depth;

    HLD(const tree& adj, const vector<i64>& _a, i64 root = 0) : a(a) {
        const i64 n = adj.size();
        heavy.resize(n);
        sz.resize(n);
        head.resize(n);
        pos.resize(n);
        parent.resize(n);
        segs.resize(n);

        subtree(root, adj);
        dfs(root, adj);

        // build segment trees
        for (i64 h = 0; h < n; h++){
            if (h == head[h]){
                vector<i64> path;
                i64 cur = h;
                while (cur != -1 && head[cur] == h){
                    path.emplace_back(a[cur]);  // add value
                    cur = heavy[cur];
                }
                segs[h] = SegTree(path);
            }
        }
    }

    void subtree(i64 u, const tree& adj, i64 p = -1, i64 d = 0){
        sz[u] = 1;
        depth[u] = d;
        for (i64 v: adj[u]){
            if (v == p) continue;
            subtree(v, adj, u, d + 1);
            sz[u] += sz[v];
        }
    }

    void dfs(i64 u, const tree& adj, i64 p = -1, i64 h = 0, i64 d = 0){
        i64 cur_sz = 0;
        pos[u] = d;
        head[u] = h;
        parent[u] = p;
        heavy[u] = -1;
        for (i64 v: adj[u]){
            if (v == p) continue;
            if (sz[v] > cur_sz){
                cur_sz = sz[v];
                heavy[u] = v;
            }
        }
        for (i64 v: adj[u]){
            if (v == p) continue;
            if (v == heavy[u]){
```

```cpp
                dfs(v, adj, u, h, d + 1);
            } else {
                dfs(v, adj, u, v, 0);
            }
        }
    }

    // query from u to p^min(k, depth[u])(u), inclusive
    // NOTE: SUM QUERY HERE, other variations are possible
    i64 query_up(i64 u, i64 k){
        i64 s = 0;
        while (u != -1 && k >= pos[u]){
            s += segs[head[u]].query(0, pos[u]);
            u = parent[head[u]];
        }
        if (u != -1){  // k < pos[u]
            s += segs[head[u]].query(pos[u] - k, pos[u]);
        }
        return s;
    }

    i64 query_path(i64 u, i64 v){
        // NOTE: need to implement LCA separetely!
        i64 l = lca(u, v);
        return query_up(u, depth[u] - depth[l]) + query_up(v, depth[v]
            - depth[l]) - a[l];
    }

    void update(i64 u, i64 x){  // set value to x
        segs[head[u]].update(pos[u], x);
    }
};
```

13.4 hld_with_range_update.cpp

```cpp
// HLD implementation
// Tested on CSES Path Queries (https://cses.fi/problemset/task/1138)
//  and range update tested on problem L from ICPC Gran Premio de
    Mexico 2025 3ra fecha
// Query: sum of vertices from root to v
// Update: set value of v to x
// Call init_seg before using

// Implementation note: this is an implementation where
// you use a single seg-tree and the node positions are their
// time of visit in the DFS, where you **visit heavy edges first**
// (so heavy paths make contiguous ranges)
// However, this leads to a larger segtree and can be slower!
// An alternative is to use an array of segtrees (one for each path)

#include <bits/stdc++.h>
using namespace std;
#define i64 int64_t

// heavy edge is the edge with the largest subtree
// going through light (i.e not heavy) edge reduces size by half
// heavy path goes up to just the vertex before light edge
template<size_t N> struct HLD {
    const vector<vector<i64>> adj;
    vector<i64> a;
    i64 parent[N], sz[N], depth[N],
        heavy[N],   // heavy[u] = v <==> uv is heavy edge
        head[N],  // head[u] = start of heavy path that goes through u
        pos[N];   // visit pos of vertex in dfs
    i64 global_pos = 0;
    i64 n, root;

    HLD(const vector<vector<i64>>& adj, i64 root = 0) :
        adj(adj), n(adj.size()), root(root)
    {
        subtree(root);
        dfs(root);
    }

    void init_seg(const vector<i64>& vec) {
        a = vec;
        vector<i64> b(n);
        for (i64 i = 0; i < n; i++) b[pos[i]] = a[i];
        seg::init(b);
    }

    void subtree(i64 u, i64 p = -1, i64 d = 0){
        sz[u] = 1; depth[u] = d; parent[u] = p;
        for (i64 v: adj[u]){
            if (v == p) continue;
            subtree(v, u, d + 1);
            sz[u] += sz[v];
        }
    }

    void dfs(i64 u, i64 p = -1, i64 h = 0){
        i64 cur_sz = 0;
        pos[u] = global_pos++;
        head[u] = h;
        heavy[u] = -1;
        for (i64 v: adj[u]){
            if (v == p) continue;
            if (sz[v] > cur_sz){
                cur_sz = sz[v];
                heavy[u] = v;
            }
        }
        if (heavy[u] != -1) dfs(heavy[u], u, h);
        for (i64 v: adj[u]){
            if (v == p || v == heavy[u]) continue;
            dfs(v, u, v);
        }
    }

    // query from u to p^{k - 1}(u)
    // NOTE: sum query here, other variations are possible
    i64 query_up(i64 u, i64 k){
```

```cpp
    i64 s = 0;
    while (u != -1 && k >= pos[u] - pos[head[u]] + 1){  // full
        path
        s += seg::query(pos[head[u]], pos[u]);
        k -= pos[u] - pos[head[u]] + 1;
        u = parent[head[u]];
    }
    if (u != -1 && k >= 1){
        s += seg::query(pos[u] - (k - 1), pos[u]);
    }
    return s;
}

i64 lca(i64 u, i64 v){
    if (pos[v] < pos[u]) swap(u, v);
    // now u is above or in another branch
    if (head[u] == head[v]) return u;
    else return lca(u, parent[head[v]]);
}

// query when values are on vertices
i64 query_path_vertex(i64 u, i64 v){
    i64 l = lca(u, v);
    return (query_up(u, depth[u] - depth[l]) + query_up(v,
        depth[v] - depth[l]) + query_up(l, 1));
}

// query when a[i] = value for edge i->parent[i]
i64 query_path_edge(i64 u, i64 v){
    i64 l = lca(u, v);
    return query_up(u, depth[u] - depth[l]) + query_up(v, depth[v]
        - depth[l]);
}

void update_pt(i64 u, i64 x){  // set value to x
    seg::update_pt(pos[u], x);
    a[u] = x;
}

void update_path_vertices(i64 u, i64 v, i64 x){  // same stuff as
    lca
    if (pos[v] < pos[u]) swap(u, v);
    // now u is above or in another branch
    if (head[u] == head[v]) seg::update_range(pos[u], pos[v], x);
    else {
        seg::update_range(pos[head[v]], pos[v], x);
        update_path_vertices(u, parent[head[v]], x);
    }
}

void update_path_edges(i64 u, i64 v, i64 x){  // same stuff as lca
    if (pos[v] < pos[u]) swap(u, v);
    // now u is above or in another branch
    if (head[u] == head[v]) {
        if (pos[u] < pos[v]) seg::update_range(pos[u] + 1, pos[v],
            x);
    }
    else {
        seg::update_range(pos[head[v]], pos[v], x);
        update_path_edges(u, parent[head[v]], x);
    }
}

// subtree of v is entered at pos[v] and visited in contiguous
    range
i64 query_subtree(i64 u){
    return seg::query(pos[u], pos[u] + sz[u] - 1);
}
void update_subtree(i64 u, i64 x){
    seg::update_range(pos[u], pos[u] + sz[u] - 1, x);
}
};
```

13.5 lca.cpp

```cpp
// tested on CSES company queries II
#include <bits/stdc++.h>
using namespace std;
#define i64 int64_t

namespace lca {
    const i64 N = 2e5 + 100, LOGN = 20;
    i64 parent[N], level[N];
    i64 blift[LOGN + 1][N];

    void dfs(i64 u, const vector<vector<i64>>& adj, i64 p = -1, i64 d
        = 0){
        parent[u] = p;
        level[u] = d;
        for (i64 v: adj[u]){
            if (v == p) continue;
            dfs(v, adj, u, d + 1);
        }
    }
    void init(const vector<vector<i64>>& adj, i64 root = 0){
        const i64 n = adj.size();
        dfs(root, adj);
        parent[root] = root;  // fix root
        for (i64 u = 0; u < n; u++) blift[0][u] = parent[u];
        for (i64 k = 1; k <= LOGN; k++){
            for (i64 u = 0; u < n; u++){
                blift[k][u] = blift[k - 1][blift[k - 1][u]];
            }
        }
    }
    i64 up(i64 u, i64 k){
        for (i64 i = 0; i <= LOGN; i++){
            if (k & (1ll << i)){
                u = blift[i][u];
            }
```

```
        }
        return u;
    }
    i64 lca(i64 u, i64 v){
        if (level[v] < level[u]) swap(u, v);
        v = up(v, level[v] - level[u]);
        if (u == v) return u;
        // loop invariant: u and v are distinct
        // (have not reached a common ancestor yet)
        for (i64 i = LOGN; i >= 0; i--){
            if (blift[i][u] != blift[i][v]){
                // keeps loop invariant -> greedily take it!
                u = blift[i][u];
                v = blift[i][v];
            }
        }
        // at the end of the invariant we must be just one level below
            the LCA
        // otherwise we could have jumped more
        return parent[u];
    }
}
```

13.6 path_queries.cpp

```
// Tested on CSES Distance Queries
// sum queries on paths. Values are on vertices

#include <bits/stdc++.h>
using namespace std;


const int N = 2e5, LOGN = 20;
int lift[LOGN + 1][N];
int sums[LOGN + 1][N];

void binary_lifting(const vector<int>& parent, const vector<int>&
    values){
    const int n = parent.size();
    for (int u = 0; u < n; u++){
        lift[0][u] = parent[u];
        sums[0][u] = values[u];
    }
    for (int k = 1; k <= LOGN; k++){
        for (int u = 0; u < n; u++){
            lift[k][u] = lift[k - 1][lift[k - 1][u]];
            // u..p^(2^{k - 1})(u) and then p^(2^{k - 1}+1)(u) to
                p^(2^k(u))
            sums[k][u] = sums[k - 1][u] + sums[k - 1][lift[k - 1][u]];
        }
    }
}

int jump(int u, int k){
    for (int i = 0; i <= LOGN; i++){
        if ((k >> i) & 1){
            u = lift[i][u];
        }
    }
    return u;
}

// v[u] + v[p[u]] + ... + v[p^{2^k - 1}(u)]
int query_sum(int u, int k){
    int s = 0;
    for (int i = 0; i <= LOGN; i++){
        if ((k >> i) & 1){
            s += sums[i][u];
            u = lift[i][u];
        }
    }
    return s;
}

int LCA(
    int u, int v, const vector<int> &level,
    const vector<int> &parent
){
    // invariant: level[u] <= level[v]
    if (level[v] < level[u]) swap(u, v);
    v = jump(v, level[v] - level[u]);

    if (u == v){
        return u;
    }

    // loop invariant: u and v are distinct
    // (we have not reached a common ancestor yet)
    for (int i = LOGN; i >= 0; i--){
        if (lift[i][u] != lift[i][v]){
            // keeps loop invariant -> greedily take it!
            u = lift[i][u];
            v = lift[i][v];
        }
    }

    // at the end of the invariant we must be just one level below the
        LCA
    // otherwise we could have jumped more
    return parent[u];
}

int path_sum(int u, int v, const vector<int> &level, const vector<int>
    &parent){
    int lca = LCA(u, v, level, parent);

    int left = query_sum(u, level[u] - level[lca]), right =
        query_sum(v, level[v] - level[lca]);
    return left + right + sums[0][lca];
}
```