ICPC Library - mlv

Contents

1 Data Structures 1
1.1 Fenwick Tree . . . 1
1.2 Prefix Sum 2D . . . 1
1.3 Segment Tree . . . 1
1.4 RMQ - Range Minimum Queries . . . 2

2 DP 2
2.1 Bitmask DP . . . 2
2.2 Bottom-Up DP . . . 3
2.3 Digit DP . . . 3
2.4 Top-Down DP . . . 3

3 Geometry 3
3.1 Convex Hull . . . 3
3.2 Point Operations . . . 4

4 Graphs 4
4.1 Bellman-Ford Algorithm . . . 4
4.2 Breadth-First Search . . . 4
4.3 Bipartite Graph Check . . . 4
4.4 Depth-First Search . . . 5
4.5 Dijkstra's Algorithm . . . 5
4.6 Disjoint Set Union (DSU) . . . 5
4.7 Cycle Detection . . . 5
4.8 Floyd-Warshall Algorithm . . . 6
4.9 Kruskal's Algorithm . . . 6
4.10 Restore Path . . . 6
4.11 Topological Sort . . . 6
4.12 LCA - Binary Lifting . . . 7
4.13 Find Bridges . . . 7

5 Math 7
5.1 Fast Exponentiation . . . 7
5.2 Linear Recurrence . . . 7
5.3 Matrix Exponentiation . . . 8
5.4 Modular Combinatorics . . . 8
5.5 Modular Inverse . . . 8
5.6 Sieve of Eratosthenes . . . 8

6 Miscellaneous 9
6.1 Backtracking . . . 9
6.2 Binary Search . . . 9
6.3 Meet in the Middle . . . 9
6.4 Monotonic Stack . . . 9

7 Strings 10
7.1 Hashing . . . 10
7.2 KMP Algorithm . . . 10
7.3 Trie . . . 10

1 Data Structures

1.1 Fenwick Tree

```
template <typename T>
class BIT{
private:
    T n;
    vector<T> bit;

public:
    BIT(const vector<T>& v){
        n = v.size();
        bit.assign(n+1, 0);

        for(int i = 0; i < n; i++){
            bit[i+1] = v[i];
        }

        for(int i = 1; i <= n; i++){
            int j = i + (i & (-i));
            if(j <= n){
                bit[j] += bit[i];
            }
        }
    }

    void update(int idx, T delta){
        ++idx;
        while(idx <= n){
            bit[idx] += delta;
            idx += idx & (-idx);
        }
    }

    T query(int idx){
        ++idx;
        T sum = 0;
        while(idx > 0){
            sum += bit[idx];
            idx -= idx & (-idx);
        }
        return sum;
    }

    T range_query(int l, int r){
        return (l == 0) ? query(r) : query(r) - query(l - 1);
    }
};
```

1.2 Prefix Sum 2D

```
// 1-based indexing (entry vector v should be 1-based as well)
vector<vector<ll>> ps(n+1, vector<ll>(n+1));
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
        ps[i][j] = v[i][j] + ps[i-1][j] + ps[i][j-1] - ps[i-1][j-1];
    }
}

auto query = [&](int xi, int yi, int xf, int yf){
    return ps[xf][yf] - ps[xf][yi-1] - ps[xi-1][yf] + ps[xi-1][yi-1];
};
```

1.3 Segment Tree

```
template <typename T>
class SegTree{

private:
    int n;
    vector<T> tree;

    T combine(T a, T b){
        return (a + b);
    }
```

```cpp
    T identity = 0;

    void build(const vector<T>& v, int node, int start, int end){
        if(start == end){
            tree[node] = v[start];
            return;
        }
        int mid = (start + end) / 2;
        build(v, 2*node, start, mid);
        build(v, 2*node+1, mid+1, end);
        tree[node] = combine(tree[2*node], tree[2*node+1]);
    }

    T query(int node, int start, int end, int l, int r){
        if(r < start or l > end){
            return identity;
        }
        if(l <= start and end <= r){
            return tree[node];
        }
        int mid = (start + end) / 2;
        T left_query = query(2*node, start, mid, l, r);
        T right_query = query(2*node+1, mid+1, end, l, r);
        return combine(left_query, right_query);
    }

    void update(int node, int start, int end, int idx, T val){
        if(start == end){
            tree[node] = val;
            return;
        }
        int mid = (start + end) / 2;
        if(idx <= mid){
            update(2*node, start, mid, idx, val);
        }
        else{
            update(2*node+1, mid+1, end, idx, val);
        }
        tree[node] = combine(tree[2*node], tree[2*node+1]);
    }

public:
    SegTree(const vector<T>& v){
        n = v.size();
        tree.resize(4*n);
        build(v, 1, 0, n-1);
    }

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

    void update(int idx, T val){
        update(1, 0, n-1, idx, val);
    }
};
```

1.4 RMQ - Range Minimum Queries

```cpp
class RMQ {
private:
    int n, K;
    std::vector<int> lg;
    std::vector<std::vector<int>> st;

public:
    RMQ(const std::vector<int>& a) {
        n = int(a.size());
        K = std::floor(std::log2(n));
        lg.assign(n+1, 0);
        for (int i = 2; i <= n; i++)
            lg[i] = lg[i/2] + 1;

        st.assign(K+1, std::vector<int>(n));

        for (int i = 0; i < n; i++)
            st[0][i] = a[i];

        for (int k = 1; k <= K; k++) {
            int len = 1 << k;
            for (int i = 0; i + len <= n; i++) {
                st[k][i] = std::min(
                    st[k-1][i],
                    st[k-1][i + (len >> 1)]
                );
            }
        }
    }

    int query(int L, int R) const {
        int len = R - L + 1;
        int k = lg[len];
        int span = 1 << k;
        return std::min(
            st[k][L],
            st[k][R - span + 1]
        );
    }
};
```

2 DP

2.1 Bitmask DP

```cpp
// setCoverDP:
// What it does: Computes the minimum cost to cover all required properties
    .
// Each item covers a set of properties (represented as a bitmask) at a
    certain cost.
// When to use: Use this function when you face a set cover problem where
    you need to select
// a subset of items to cover all properties with the minimum total cost.

struct Item {
    int mask, cost;
};

int setCoverDP(const vector<Item>& items, int m) {
    int n = items.size();
    vector<vector<int>> dp(n + 1, vector<int>(1 << m, INF));
    dp[0][0] = 0;
    for (int i = 0; i < n; i++) {
        for (int mask = 0; mask < (1 << m); mask++) {
            dp[i + 1][mask] = min(dp[i + 1][mask], dp[i][mask]);
            int new_mask = mask | items[i].mask;
            if (dp[i][mask] != INF)
                dp[i + 1][new_mask] = min(dp[i + 1][new_mask], dp[i][mask]
                    + items[i].cost);
        }
    }
    return dp[n][(1 << m) - 1];
}

// tspDP:
// What it does: Computes the minimum cost to visit all vertices in a
    complete graph starting from vertex 0.
```

```
// It uses a bitmask DP approach to solve the Traveling Salesman Problem (
    TSP) or similar path-cover problems.
// When to use: Use this function when you need to determine the shortest
    path that visits every vertex
// exactly once in problems like TSP.
int tspDP(const vector<vector<int>>& graph) {
    int n = graph.size(), N = 1 << n;
    vector<vector<int>> dp(N, vector<int>(n, INF));
    dp[1][0] = 0; // start at vertex 0, mask = 1<<0
    for (int mask = 0; mask < N; mask++) {
        if (!(mask & 1)) continue; // ensure starting vertex is visited
        for (int u = 0; u < n; u++) {
            if (!(mask & (1 << u))) continue;
            for (int v = 0; v < n; v++) {
                if (mask & (1 << v)) continue;
                dp[mask | (1 << v)][v] = min(dp[mask | (1 << v)][v], dp[
                    mask][u] + graph[u][v]);
            }
        }
    }
    int result = INF;
    for (int u = 0; u < n; u++) {
        result = min(result, dp[N - 1][u]);
    }
    return result;
}
```

2.2 Bottom-Up DP

```
int coinChangeBottomUp(const vector<int>& coins, int amount) {
    vector<int> dp(amount + 1, INF);
    dp[0] = 0;
    for (int a = 1; a <= amount; a++) {
        for (int coin : coins) {
            if (a - coin >= 0)
                dp[a] = min(dp[a], dp[a - coin] + 1);
        }
    }
    return dp[amount] == INF ? -1 : dp[amount];
}
```

2.3 Digit DP

```
ll dp[2000+3][2000+3][2][2][2];
ll solve_digit_dp(int pos, int mod, bool smaller, bool only_zero, bool
    is_even){
    if(pos == static_cast<int>(v.size())){
        return mod == 0;
    }
    if(dp[pos][mod][smaller][only_zero][is_even] != -1){
        return dp[pos][mod][smaller][only_zero][is_even] % MOD;
    }
    dp[pos][mod][smaller][only_zero][is_even] = 0;
    int tight = smaller ? 9 : v[pos];
    for(int i = 0; i <= tight; i++){
        bool new_is_smaller = smaller or (i < v[pos]);
        int new_mod = (mod*10 + i) % m;
        int new_only_zero = only_zero and i == 0;
        int new_is_even = new_only_zero ? is_even : !is_even;
        if(not new_only_zero and i == d and not new_is_even){
            continue;
        }
        if(not new_only_zero and i != d and new_is_even){
            continue;
        }
        dp[pos][mod][smaller][only_zero][is_even] = (dp[pos][mod][smaller][
            only_zero][is_even] % MOD +
            solve_digit_dp(pos+1, new_mod, new_is_smaller, new_only_zero,
                new_is_even) % MOD)
            % MOD;
    }

    return dp[pos][mod][smaller][only_zero][is_even] % MOD;
}
```

2.4 Top-Down DP

```
int lcsTopDownHelper(const string &s1, const string &s2, int i, int j,
    vector<vector<int>> &memo) {
    if (i == s1.size() || j == s2.size())
        return 0;
    if (memo[i][j] != -1)
        return memo[i][j];
    if (s1[i] == s2[j])
        memo[i][j] = 1 + lcsTopDownHelper(s1, s2, i + 1, j + 1, memo);
    else
        memo[i][j] = max(lcsTopDownHelper(s1, s2, i + 1, j, memo),
            lcsTopDownHelper(s1, s2, i, j + 1, memo));
    return memo[i][j];
}

int lcsTopDown(const string &s1, const string &s2) {
    vector<vector<int>> memo(s1.size(), vector<int>(s2.size(), -1));
    return lcsTopDownHelper(s1, s2, 0, 0, memo);
}
```

3 Geometry

3.1 Convex Hull

```
int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1)
        return;

    sort(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
```

```
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1],
                 a[i], include_collinear))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
            while (down.size() >= 2 && !ccw(down[down.size()-2], down[down.
                 size()-1], a[i], include_collinear))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    if (include_collinear && up.size() == a.size()) {
        reverse(a.begin(), a.end());
        return;
    }
    a.clear();
    for (int i = 0; i < (int)up.size(); i++)
        a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        a.push_back(down[i]);
}
```

3.2 Point Operations

```
const double PI = acos(-1);
constexpr double EPS = 1e-6;
template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<typename T>
struct Point{
    T x, y;
    Point(T x=0, T y=0) : x(x),y(y){}
    bool operator < (Point o) const { return tie(x,y) < tie(o.x,o.y); }
    bool operator == (Point o) const { return tie(x,y) == tie(o.x,o.y); }
    Point operator + (Point o) const { return Point(x+o.x,y+o.y); }
    Point operator - (Point o) const { return Point(x-o.x,y-o.y); }
    Point operator * (T k) const { return Point(x*k,y*k); }
    Point operator / (T k) const { return Point(x/k,y/k); }
    double cross(Point o) const { return x*o.y - y*o.x; }
    double cross(Point a, Point b) const { return (a-*this).cross(b-*this);
         }
    double dot(Point o) const { return x*o.x + y*o.y; }
    double dist() const { return std::sqrt(x*x + y*y); }
    double dist(Point a) const { return std::sqrt((x-a.x)*(x-a.x) + (y-a.y)
         *(y-a.y)); }
    double dist2() const { return x*x + y*y; }
    double len() const { return hypot(x,y); }
    Point perp() const { return Point(-y,x); }
    Point rotate(double a) const { return Point(x*cos(a)-y*sin(a), x*sin(a)
         +y*cos(a)); }
    int quad() { return (x<0)^3*(y<0); }
    bool ccw(Point<T> q, Point<T> r){ return (q-*this).cross(r-q) > 0;}
};

template<typename T>
Point<T> projPointLine(Point<T> a, Point<T> b, Point<T> c) { // ponto c na
     linha a - b, a.b = |a| cost * |b|
    return a + (b-a) * (b-a).dot(c-a) / (b-a).dot(b-a);
}

template<typename T>
double distancePointLine(Point<T> a, Point<T> b, Point<T> c) { // distancia
      do ponto c a linha a - b
    return c.dist(projPointLine(a, b, c));
}

template<typename T>
bool ptInSegment (Point<T> a, Point<T> b, Point<T> c) { // ponto c esta em
     um segmento a - b
```

```
    if (a == b) return a == c;
    a = a-c, b = b-c;
    return cmp(cross(a, b)) == 0 && cmp(dot(a, b)) <= 0;
}
```

4 Graphs

4.1 Bellman-Ford Algorithm

```
struct Edge {
    int a, b, cost;
};

void bellman_ford(int u)
{
    vector<int> dist(n, INF);
    dist[u] = 0;
    for (int i = 0; i < n - 1; ++i)
        for (Edge e : edges)
            if (dist[e.a] < INF)
                dist[e.b] = min(dist[e.b], dist[e.a] + e.cost);
}
```

4.2 Breadth-First Search

```
void bfs(int u){
    queue<int> q;
    q.push(u);
    vis[u] = true;

    vector<bool> vis(n);
    vector<int> dist(n);
    vector<int> pred(n);

    while(!q.empty()){
        int v = q.front();
        q.pop();

        for(int next_v:adj[v]){
            if(!vis[next_v]){
                vis[next_v] = true;
                q.push(next_v);
                d[next_v] = d[v] + 1;
                pred[next_v] = v;
            }
        }
    }
}
```

4.3 Bipartite Graph Check

```
bool can_bipartite = true;
vector<int> colors(n, -1);

void dfs_bipartite(int source, bool color){
    colors[source] = color;
    for(int next_v:adj[source]){
        if(colors[next_v] == -1){
            dfs(next_v, !color);
        }
        else if(colors[next_v] == color){
            can_bipartite = false;
            break;
```

```
        }
    }
}

void bipartite(){
    for(int i = 0; i < n; i++){
        if(colors[i] == -1){
            dfs(i, false);
        }
    }
}
```

4.4 Depth-First Search

```
void dfs(int u){
    vis[u] = true;
    for(int v:adj[u]){
        dfs(v);
    }
}
```

4.5 Dijkstra's Algorithm

```
void dijkstra(int source){
    dist.assign(n, LLONG_MAX);
    dist[source] = 0;
    priority_queue<ii, vector<ii>, greater<ii>> pq;

    pq.push({0, source});

    while(not pq.empty()){
        auto[distance, v] = pq.top();
        pq.pop();

        if(distance > dist[v]){
            continue;
        }

        for(auto [next_v, next_dist]:adj[v]){
            if(distance + next_dist < dist[next_v]){
                dist[next_v] = distance + next_dist;
                pq.push({dist[next_v], next_v});
            }
        }
    }
}
```

4.6 Disjoint Set Union (DSU)

```
class DSU{
    private:
        vector<ll> rep;
        vector<ll> size;

    public:
        DSU(ll n){
            size.assign(n, 1);
            rep.resize(n);
            for(int i = 0; i < n; i++){
                rep[i] = i;
            }
        }

        ll find(ll v){
            if(v == rep[v]){
                return v;
            }
            rep[v] = find(rep[v]);
            return rep[v];
        }

        void join(ll u, ll v){
            u = find(u);
            v = find(v);

            if(u == v){
                return;
            }

            if(size[u] < size[v]){
                swap(u, v);
            }

            rep[v] = u;
            size[u] += size[v];
        }
};
```

4.7 Cycle Detection

```
//directed graph

int n;
vector<vector<int>> adj;
vector<char> color; // 0 = unvisited, 1 = visiting, 2 = visited
vector<int> parent;
int cycle_start, cycle_end;

bool dfs_directed(int u) {
    color[u] = 1;
    for (int v : adj[u]) {
        if (color[v] == 0) {
            parent[v] = u;
            if (dfs_directed(v)){
                return true;
            }

        }
        else if (color[v] == 1) {
            cycle_end = u;
            cycle_start = v;
            return true;
        }
    }
    color[u] = 2;
    return false;
}

void find_cycle_directed() {
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int i = 0; i < n; i++) {
        if (color[i] == 0 && dfs_directed(i)){
            break;
        }
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
        return;
    }
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v])
```

```cpp
        cycle.push_back(v);

    cycle.push_back(cycle_start);

    reverse(cycle.begin(), cycle.end());
}

//undirected graph
int n;
vector<vector<int>> adj;
vector<bool> vis;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs_undirected(int u, int par) {
    vis[u] = true;
    for (int v : adj[u]) {
        if(v == par){
            continue;
        }
        if (vis[v]) {
            cycle_end = u;
            cycle_start = v;
            return true;
        }
        parent[v] = u;
        if (dfs_undirected(v, parent[v]))
            return true;
    }
    return false;
}

void find_cycle_undirected() {
    vis.assign(n, false);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int i = 0; i < n; i++) {
        if (!vis[i] && dfs(i, parent[i])){
            break;
        }
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
        return;
    }
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v])
        cycle.push_back(v);
    cycle.push_back(cycle_start);
}
```

4.8 Floyd-Warshall Algorithm

```cpp
dist.assign(n, vector<ll>(n, LLONG_MAX));
//input distances before floyd
//self distances are 0
void floyd_warshall() {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] < LLONG_MAX and dist[k][j] < LLONG_MAX) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
}
```

4.9 Kruskal's Algorithm

```cpp
sort(edges.begin(), edges.end());

ll kruskal(){
    ll total = 0;
    DSU dsu(n);

    for(auto[w, u, v] : edges){
        if(dsu.find(u) != dsu.find(v)){
            dsu.join(u, v);
            total += w;
        }
    }

    return total;
}
```

4.10 Restore Path

```cpp
if (!vis[u]) {
    cout << "No path!";
    return;
}
vector<int> path;
for (int v = u; v != -1; v = p[v])
    path.push_back(v);

reverse(path.begin(), path.end());
```

4.11 Topological Sort

```cpp
//using dfs
void dfs(int u) {
    vis[u] = true;
    for (int v : adj[u]) {
        if (!vis[v]) {
            dfs(v);
        }
    }
    answer.push_back(u);
}

void topo_sort_dfs() {
    vis.assign(n, false);
    answer.clear();
    for (int i = 0; i < n; ++i) {
        if (!vis[i]) {
            dfs(i);
        }
    }
    reverse(answer.begin(), answer.end());
}

//using bfs (khan's algorithm)
void topo_sort_bfs(){
    vector<int> dep(n);
    for(int i = 0; i < n; i++){
        for(int v:adj[i]){
            ++dep[v];
        }
    }

    queue<int> q;
    for(int i = 0; i < n; i++){
        if(dep[i] == 0){
```

```
            q.push(i);
        }
    }

    vector<int> answer;
    while(!q.empty()){
        int u = q.front();
        q.pop();
        answer.push_back(u);

        for(int v:adj[u]){
            --dep[v];
            if(dep[v] == 0 && !vis[v]){
                q.push(v);
            }
        }
    }

    if(answer.size() != n){
        cout << "cycle" << endl;
    }
}
```

4.12 LCA - Binary Lifting

```
int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

    for (int u : adj[v]) {
        if (u != p)
            dfs(u, v);
    }

    tout[v] = ++timer;
}

bool is_ancestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

4.13 Find Bridges

```
void IS_BRIDGE(int u, int v);
vector<vector<int>> adj;
int n;

vector<bool> vis;
vector<int> tin, low;
int timer;

void dfs(int u, int p){
    vis[u] = true;
    tin[u] = low[u] = timer++;
    bool parent_skipped = false;
    for(int v:adj[u]){
        if(v == p and !parent_skipped){
            parent_skipped = true;
            continue;
        }
        if(vis[v]){
            low[u] = min(low[u], tin[v]);
        }
        else{
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if(low[v] > tin[u]){
                IS_BRIDGE(u, v);
            }
        }
    }
}

void find_bridges(){
    timer = 0;
    vis.assign(n, false);
    tin.assign(n, -1);
    for(int i = 0; i < n; i++){
        if(not vis[i]){
            dfs(i, -1);
        }
    }
}
```

5 Math

5.1 Fast Exponentiation

```
ll fast_exp(ll a, ll b, ll m) {
    ll res = 1;
    a %= m;
    while (b) {
        if (b & 1){
            res = (res * a) % m;
        }
        a = (a * a) % m;
        b >>= 1;
    }
    return res;
}
```

5.2 Linear Recurrence

```
// =============================
```

```
// Fibonacci Twist using Matrices
// ==============================

/*
    Definition of the Fibonacci Twist sequence:

    ft(n) = ft(n-1) + ft(n-2) + (n-1)
    with ft(0) = 0 and ft(1) = 1

    First few terms in the sequence:
    n =  0  1  2  3   4   5   6
    ft = 0  1  2  5  10  19  34

    V(n) =
    [ ft(n)    ]
    [ ft(n-1)  ]
    [ (n-1)    ]
    [ 1        ]

    V(n) = T * V(n-1)

    T = [ 1  1  1  -1 ]
        [ 1  0  0   0 ]
        [ 0  0  1   1 ]
        [ 0  0  0   1 ]

    Base: V(1) = [ ft(1) = 1, ft(0) = 0, (n-1) = 2, constant 1 ]
*/

ll fibonacci_twist(ll n, ll mod) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    matrix fib_base = {{1, 1, 1, -1},
                       {1, 0, 0,  0},
                       {0, 0, 1,  1},
                       {0, 0, 0,  1}};

    matrix result = matrix_exponentiation(fib_base, n - 1, mod);

    ll base_case[4] = {1, 0, 2, 1};
    ll answer = 0;

    for (int i = 0; i < 4; i++) {
        answer = (answer + (base_case[i] * result[0][i]) % mod + mod) % mod
            ;
    }

    return answer;
}
```

5.3 Matrix Exponentiation

```
struct matrix {
    long long mat[2][2];
    matrix friend operator *(const matrix &a, const matrix &b){
        matrix c;
        for (int i = 0; i < 2; i++) {
          for (int j = 0; j < 2; j++) {
              c.mat[i][j] = 0;
              for (int k = 0; k < 2; k++) {
                  c.mat[i][j] += a.mat[i][k] * b.mat[k][j];
              }
          }
        }
        return c;
    }
};

matrix matrix_exp(matrix base, long long n) {
    matrix ans{ {
      {1, 0},
      {0, 1}
    } };
    while(n > 0){
        if(n & 1){
            ans = ans*base;
        }
        base = base*base;
        n >>= 1;
    }
    return ans;
}
```

5.4 Modular Combinatorics

```
ll comb(ll n, ll m){
    ll answer = 1;
    m = min(m, n-m);

    for(int i = 0; i < m; i++){
        answer = (answer * (n-i)) % MOD;
        answer = (answer * mod_inverse(i+1, MOD)) % MOD;
    }

    return answer;
}
```

5.5 Modular Inverse

```
ll inverse(ll n, ll mod){
    ll ans = 1;
    ll a = n;
    ll b = mod - 2;
    while(b){
        if(b & 1){
            ans = (ans * a) % mod;
        }
        a = (a * a) % mod;
        b >>= 1;
    }
    return ans;
}
```

5.6 Sieve of Eratosthenes

```
// find all prime numbers
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
    if (is_prime[i]) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}

vector<int> sieveDivisorCount(int N) {
    vector<int> divCount(N + 1, 0);
    for (int i = 1; i <= N; i++) {
        for (int j = i; j <= N; j += i)
            divCount[j]++;
    }
    return divCount;
}

vector<int> getDivisors(int x) {
    vector<int> divs;
    for (int i = 1; i * i <= x; i++) {
```

```
        if (x % i == 0) {
            divs.push_back(i);
            if (i != x / i)
                divs.push_back(x / i);
        }
    }
    sort(divs.begin(), divs.end());
    return divs;
}
```

6 Miscellaneous

6.1 Backtracking

```
void backtrack(vector<int>& state, vector<int>& choices, vector<bool>& used
    ) {
    if (state.size() == choices.size()) {
        process_solution(state);
        return;
    }
    for (int i = 0; i < choices.size(); i++) {
        if (!used[i] && is_valid(state, choices[i])) {
            used[i] = true;
            state.push_back(choices[i]);

            backtrack(state, choices, used);

            used[i] = false;
            state.pop_back();
        }
    }
}
```

6.2 Binary Search

```
int lower_bound_index(vector<int>& arr, int target) {
    int left = 0, right = arr.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid;
    }
    return left;
}
```

6.3 Meet in the Middle

```
void genSubsets(const vector<ll>& v, vector<ll>& subs_sum_v, ll& sum, int
     index){
    subs_sum_v.push_back(sum);

    for(int i = index; i < static_cast<int>(v.size()); ++i){
        sum += v[i];
        genSubsets(v, subs_sum_v, sum, i+1);
        sum -= v[i];
    }
}

ll meet_in_the_middle(){
    vector<ll> lo(n/2);
    vector<ll> hi(n-n/2);
    for(int i = 0; i < n/2; i++){
        cin >> lo[i];
    }
    for(int i = n/2; i < n; i++){
        cin >> hi[i-n/2];
    }

    vector<ll> subs_sum_lo;
    subs_sum_lo.reserve(pow(2, n/2));
    vector<ll> subs_sum_hi;
    subs_sum_hi.reserve(pow(2, n-n/2));

    ll sum = 0;
    genSubsets(lo, subs_sum_lo, sum, 0);
    sum = 0;
    genSubsets(hi, subs_sum_hi, sum, 0);

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

    ll ans = 0;
    for(ll v:subs_sum_hi){
        auto it_lo = lower_bound(subs_sum_lo.begin(), subs_sum_lo.end(),
            target-v);
        auto it_hi = upper_bound(subs_sum_lo.begin(), subs_sum_lo.end(),
            target-v);

        if(it_lo != subs_sum_lo.end()){
            ans += it_hi - it_lo;
        }
    }

    return ans;
}
```

6.4 Monotonic Stack

```
void monotonic_stack(){
    vector<int> v(N);
    vector<int> left(N);
    vector<int> right(N);

    for(int i = 0; i < N; i++){
        cin >> v[i];
    }

    stack<ii> stack_left;
    stack<ii> stack_right;

    stack_left.push({v[0], 0});
    stack_right.push({v[N-1], N-1});
    left[0] = -1;
    right[N-1] = -1;

    for(int i = 1; i < N; i++){
        while(not stack_left.empty() and v[i] <= stack_left.top().first){
            stack_left.pop();
        }
        left[i] = stack_left.empty() ? -1 : stack_left.top().second;
        stack_left.push({v[i], i});

        int j = N-1-i;
        while(not stack_right.empty() and v[j] <= stack_right.top().first){
            stack_right.pop();
        }
        right[j] = stack_right.empty() ? -1 : stack_right.top().second;
        stack_right.push({v[j], j});
    }
}
```

7 Strings

7.1 Hashing

```
class StringHash{
    public:
        int n;
        string s;
        int p1, p2;
        ll mod1, mod2;
        vector<ll> prefix1, prefix2;
        vector<ll> power1, power2;

        StringHash(const string &s, int p1 = 53, ll mod1 = 1e9+7)
        : s(s), p1(p1), mod1(mod1){
            n = s.size();
            prefix1.resize(n+1);
            power1.resize(n+1);

            power1[0] = 1;
            for(int i = 0; i < n; i++){
                prefix1[i+1] = (prefix1[i] + (s[i] - 'a' + 1) * power1[i])
                     % mod1;
                power1[i+1] = (power1[i]*p1) % mod1;
            }
        }

        ll getHash(int l, int r) const {
            ll h1 = (prefix1[r+1] + mod1 - prefix1[l]) % mod1;
            return h1;
        }

        bool compareSubstr(int l1, int r1, int l2, int r2) const {
            // if((r1 - l1) != (r2 - l2)){
            //     return false;
            // }
            auto hash1 = getHash(l1, r1);
            auto hash2 = getHash(l2, r2);

            if(l1 < l2){
                hash1 = (hash1 * power1[l2-l1]) % mod1;
            }
            else if (l1 > l2){
                hash2 = (hash2 * power1[l1-l2]) % mod1;
            }
            return hash1 == hash2;
        }

        bool compareSubstr(ll hash1, int l1, int r1, int l2, int r2) const
         {
            if((r1 - l1) != (r2 - l2)){
                return false;
            }
            auto hash2 = getHash(l2, r2);

            if(l1 < l2){
                hash1 = (hash1 * power1[l2-l1]) % mod1;
            }
            else if (l1 > l2){
                hash2 = (hash2 * power1[l1-l2]) % mod1;
            }
            return hash1 == hash2;
        }
    };
```

7.2 KMP Algorithm

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

7.3 Trie

```
struct TrieNode {
    int count;
    bool isEnd;
    TrieNode* children[26];

    TrieNode(){
        count = 0;
        isEnd = false;
        for(int i = 0; i < 26; i++){
            children[i] = nullptr;
        }
    }
};
struct TrieNode {
    int count;
    bool isEnd;
    TrieNode* children[26];

    TrieNode(){
        count = 0;
        isEnd = false;
        for(int i = 0; i < 26; i++){
            children[i] = nullptr;
        }
    }
};

class Trie{
private:
    TrieNode* root;

public:
    Trie(){
        root = new TrieNode();
    }

    void insert(const string& word){
        TrieNode* cur = root;
        for(char c:word){
            int idx = c - 'a';
            if(!cur->children[idx]){
                cur->children[idx] = new TrieNode();
            }
            cur = cur->children[idx];
            cur->count++;
        }
        cur->isEnd = true;
    }

    int search(const string &word){
        TrieNode* cur = root;
        for(char c:word){
            int idx = c - 'a';
            if(!cur->children[idx]){
                return false;
```

```
            }
            cur = cur->children[idx];
        }
        return cur->count;
    }
};
class Trie{
private:
    TrieNode* root;

public:
    Trie(){
        root = new TrieNode();
    }

    void insert(const string& word){
        TrieNode* cur = root;
        for(char c:word){
            int idx = c - 'a';
            if(!cur->children[idx]){
                cur->children[idx] = new TrieNode();
            }
            cur = cur->children[idx];
            cur->count++;
        }
        cur->isEnd = true;
    }

    int search(const string &word){
        TrieNode* cur = root;
        for(char c:word){
            int idx = c - 'a';
            if(!cur->children[idx]){
                return false;
            }
            cur = cur->children[idx];
        }
        return cur->count;
    }
};
```