🟥 1) GEOMETRÍA BÁSICA (Vectores, cross product, dot product)

```
// Estructura para representar un punto en 2D
struct Punto {
    long long x, y;

    // Constructor
    Punto(long long x=0, long long y=0) : x(x), y(y) {}

    // Suma de vectores (puntos)
    Punto operator + (const Punto &o) const {
        return {x + o.x, y + o.y};
    }

    // Resta de vectores
    Punto operator - (const Punto &o) const {
        return {x - o.x, y - o.y};
    }

    // Producto escalar → mide cuán alineados están dos vectores
    long long dot(const Punto &o) const {
        return x*o.x + y*o.y;
    }

    // Producto cruzado → indica giro (izquierda/derecha) y área
    long long cross(const Punto &o) const {
        return x*o.y - y*o.x;
    }

    // Norma al cuadrado → distancia² desde (0,0)
    long long norma2() const {
        return x*x + y*y;
    }
};
```

📌 ¿En qué se usa?

- Saber si tres puntos hacen **giro left/right** → cross product ✔
- Encontrar orientaciones → Convex Hull, segment intersection
- Detectar colinealidad
- Distancias sin usar `double`
- Problemas de polígonos, áreas, vectores, geometría computacional.

🟧 2) CRIBA DE PRIMOS (Sieve + menor primo divisor)

```
struct Primos {
    vector<int> lista;     // lista de todos los primos
    vector<int> minP;      // menor primo que divide a cada número
    vector<bool> esPrimo;  // marca de primalidad

    Primos(int n) { criba(n); }

    void criba(int n) {
        esPrimo.assign(n+1, true);
        minP.assign(n+1, 0);
        esPrimo[0] = esPrimo[1] = false;

        for (int i = 2; i <= n; i++) {

            // Si sigue marcado como primo → lo agrego
            if (esPrimo[i]) {
                lista.push_back(i);
                minP[i] = i;  // el menor primo que divide a i es i
mismo
            }
```

```
            // Marco múltiplos de i
            for (long long j = 1LL * i * i; j <= n; j += i) {
                if (esPrimo[j]) {
                    esPrimo[j] = false; // no es primo
                    minP[j] = i;        // menor primo divisor
                }
            }
        }
    }
};
```

📌 ¿En qué se usa?

- Problemas que requieren testear primalidad muchas veces
- Factorización rápida
- Cantidad de divisores, sumatoria de divisores
- Construir funciones multiplicativas (phi, mu, etc.)
- Muy común en concursos (Codeforces, CSES, ICPC)

🟨 3) EXPONENCIACIÓN BINARIA

Sirve para calcular potencias grandes rápido.

```
// Calcula a^e % mod en O(log e)
long long binpow(long long a, long long e, long long mod) {
    long long r = 1;   // resultado

    while (e > 0) {
        if (e & 1)
            r = (r * a) % mod; // si el bit está prendido → multiplico
```

```
        a = (a * a) % mod;      // elevo a al cuadrado
        e >>= 1;                // avanzo un bit en e
    }

    return r;
}
```

📌 ¿En qué se usa?

- Modular exponentiation
- Combinatoria (nCk mod p)
- Inversos modulares (a^(p-2) mod p)
- Criptografía
- Problemas con potencias enormes

🟩 4) BINARY SEARCH (búsqueda binaria)

🟢 Versión clásica

```
int bs(const vector<int> &v, int x) {
    int l = 0, r = (int)v.size() - 1;

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

        if (v[m] == x) return m;    // encontrado
        if (v[m] < x) l = m + 1;    // busco derecha
        else r = m - 1;             // busco izquierda
    }
    return -1; // no encontrado
}
```

🟣 Binary search sobre respuesta

```
while (lo < hi) {
    long long mid = (lo + hi) / 2;

    if (can(mid))      // si mid funciona
        hi = mid;      // pruebo algo menor
    else
        lo = mid + 1;  // necesito algo mayor
}
```

📌 ¿En qué se usa?

- Tiempo mínimo necesario
- Máximo valor posible con restricciones
- Decidir si un valor es válido y binary search sobre él
- Ejemplos:
    - Factory Machines
    - Array Division
    - Minimum capacity, Maximum feasible value

🟦 5) MERGESORT

```
// Algoritmo divide-conquista para ordenar en O(n log n)
void merge_sort(vector<int> &a, int l, int r) {

    if (l >= r) return;

    int m = (l + r) / 2;

    // Ordeno la mitad izquierda y derecha
```

```
    merge_sort(a, l, m);
    merge_sort(a, m+1, r);

    // Mezclo las dos mitades ya ordenadas
    vector<int> tmp;
    int i = l, j = m + 1;

    // Comparo elementos para mantener orden
    while (i <= m && j <= r) {
        if (a[i] < a[j]) tmp.push_back(a[i++]);
        else tmp.push_back(a[j++]);
    }

    // Agrego restos
    while (i <= m) tmp.push_back(a[i++]);
    while (j <= r) tmp.push_back(a[j++]);

    // Copio de vuelta al array original
    for (int k = 0; k < tmp.size(); k++)
        a[l + k] = tmp[k];
}
```

📌 ¿En qué se usa?

- Ordenar muy rápido
- Contar inversiones
- Algoritmos divide & conquer
- Mezclar listas ordenadas

🟫 6) DSU (Disjoint Set Union / Union-Find)

Basado en tu archivo Kruskal.cpp

Kruskal

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

    // Inicializa n conjuntos separados
    DSU(int n) {
        parent.resize(n);
        sz.assign(n, 1);

        for (int i = 0; i < n; i++)
            parent[i] = i; // cada nodo es su propio líder
    }

    // Encuentra el representante del conjunto (con path compression)
    int find(int x) {
        if (x == parent[x]) return x;
        return parent[x] = find(parent[x]); // compresión de caminos
    }

    // Une los conjuntos de a y b
    void unite(int a, int b) {
        a = find(a);
        b = find(b);

        if (a == b) return; // ya están conectados

        // attach smaller tree under bigger tree
        if (sz[a] < sz[b]) swap(a, b);

        parent[b] = a;
        sz[a] += sz[b];
    }
};
```

📌 ¿En qué se usa?

- Saber si dos nodos están conectados
- Unificaciones dinámicas
- Minimum Spanning Tree (Kruskal)
- Componentes conectadas
- Union-find es uno de los algoritmos MÁS usados en CP

🟪 7) KRUSKAL (MST)

Usa DSU para armar un árbol de expansión mínima.

```
struct Edge {
    int u, v;
    long long w;
};

// n = nodos
// edges = lista de aristas
long long kruskal(int n, vector<Edge> &edges) {

    // ordeno las aristas por peso
    sort(edges.begin(), edges.end(),
         [](auto &a, auto &b){ return a.w < b.w; });

    DSU dsu(n);
    long long mst = 0;
    int used = 0;

    // recorro en orden creciente
    for (auto &e : edges) {

        // si unir u y v no forma ciclo
        if (dsu.find(e.u) != dsu.find(e.v)) {
            dsu.unite(e.u, e.v);
```

```
            mst += e.w; // agrego a MST
            used++;
        }
    }

    if (used != n-1) return -1; // grafo no conexo
    return mst;
}
```

📌 ¿En qué se usa?

- Redes de costo mínimo
- Infraestructura, caminos, cables, carreteras
- Selección de aristas de mínimo peso sin ciclos
- Problemas tipo:
    - “Conectar ciudades al menor costo”
    - “Unificar servidores con cables”

🟫 8) DIJKSTRA (camino mínimo)

Basado en tu archivo Dijsktra.cpp

Dijsktra

```
struct Edge { int to; long long w; };

vector<long long> dijkstra(int n, vector<vector<Edge>> &g) {
    const long long INF = LLONG_MAX;

    vector<long long> dist(n, INF);
    priority_queue<
```

```
        pair<long long,int>,
        vector<pair<long long,int>>,
        greater<pair<long long,int>>
    > pq;

    // empiezo desde el nodo 0
    dist[0] = 0;
    pq.push({0,0});

    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();

        // si tengo un mejor camino, ignoro este
        if (d != dist[u]) continue;

        // relajo vecinos
        for (auto &e : g[u]) {
            if (dist[u] + e.w < dist[e.to]) {
                dist[e.to] = dist[u] + e.w;
                pq.push({dist[e.to], e.to});
            }
        }
    }

    return dist;
}
```

📌 ¿En qué se usa?

- Rutas mínimas
- GPS, mapas
- Grafos con pesos positivos
- Tiempo/costo mínimo en trayectorias

🟦 9) SEGMENT TREE (para sumas o máximos)

```
struct SegTree {
    int n;
    vector<long long> st;

    SegTree(int n) : n(n) {
        st.assign(4*n, 0);
    }

    // Construye el árbol
    void build(vector<long long> &a, int p, int l, int r) {
        if (l == r) {
            st[p] = a[l];
            return;
        }
        int m = (l + r) / 2;

        build(a, p*2, l, m);
        build(a, p*2+1, m+1, r);

        st[p] = st[p*2] + st[p*2+1];
    }

    // Consulta suma en [i, j]
    long long query(int p, int l, int r, int i, int j) {
        if (j < l || r < i) return 0; // completamente fuera
        if (i <= l && r <= j) return st[p]; // completamente dentro

        int m = (l+r)/2;
        return query(p*2, l, m, i, j)
             + query(p*2+1, m+1, r, i, j);
    }
```

```
    // Actualiza posición idx con valor val
    void update(int p, int l, int r, int idx, long long val) {
        if (l == r) {
            st[p] = val;
            return;
        }
        int m = (l+r)/2;

        if (idx <= m) update(p*2, l, m, idx, val);
        else update(p*2+1, m+1, r, idx, val);

        st[p] = st[p*2] + st[p*2+1];
    }
};
```

📌 ¿En qué se usa?

- Queries de rangos con actualizaciones dinámicas
- RMQ (range min/max query)
- Problemas de subarreglos, intervalos, estructuras persistentes
- Esencial en:
    - CSES Range Queries
    - Codeforces segment tree tasks