1. A* SEARCH

```cpp
/*solo funciona con valores*/
/*
 * A* SEARCH - Búsqueda informada con heurística
 * Encuentra el camino más corto usando f(n) = g(n) + h(n).
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(b^d) peor caso?
 *   - b = branching factor (vecinos promedio)
 *   - d = profundidad de la solución
 *   - Con mala heurística, explora exponencialmente: b^d nodos
 *   - Con heurística admisible buena: cercano a O((V + E) log V) como Dijkstra
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Con mala heurística (h=0), degrada a Dijkstra: O((V+E) log V). Con
 *   heurística admisible pero poco informativa, explora muchas ramas. En el
 *   peor caso (árbol de branching factor b, profundidad d): O(b^d) nodos explorados.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V)?
 *   - Arrays g_score, f_score, parent: 3V
 *   - Priority queue: hasta V nodos en memoria
 *   - Path reconstruido: a lo más V nodos
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Priority queue: a lo más V nodos abiertos. Arrays g_score, f_score,
 *   parent: 3V elementos. Total: O(V).
 */
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int id;
    int f;
    bool operator>(const Node &other) const {
        return f > other.f;
    }
};

vector<int> a_star(int start, int goal,
                   const vector<vector<pair<int,int>>> &adj,
                   const vector<int>& h) {

    int n = adj.size();
    const int INF = 1e9;

    vector<int> g(n, INF);     // costo real acumulado
    vector<int> parent(n, -1);
    vector<bool> closed(n, false);

    priority_queue<Node, vector<Node>, greater<Node>> pq;

    g[start] = 0;
    pq.push({start, h[start]});

    while (!pq.empty()) {
        Node current = pq.top();
        pq.pop();

        int u = current.id;

        if (closed[u]) continue;
        closed[u] = true;

        if (u == goal) break;

        for (auto &edge : adj[u]) {
            int v = edge.first;
            int w = edge.second;
            // NO SE PERMITEN PESOS NEGATIVOS:
            if (w < 0) {
                throw runtime_error("A* received a negative edge weight");
            }

            int tentative = g[u] + w;
            if (tentative < g[v]) {
                g[v] = tentative;
                parent[v] = u;
                int f = tentative + h[v];
                pq.push({v, f});
            }
        }
    }

    if (g[goal] == INF) return {}; // No reachable

    // reconstruct path
    vector<int> path;
    for (int v = goal; v != -1; v = parent[v])
        path.push_back(v);
    reverse(path.begin(), path.end());

    return path;
}

int main() {
    int n, m, inicio, objetivo;
    cin >> n >> m >> inicio >> objetivo;

    vector<vector<pair<int,int>>> adj(n);

    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});
    }

    // Leer heurística (valores h para cada nodo)
    vector<int> h(n);
    for (int i = 0; i < n; i++) {
        cin >> h[i];
    }

    auto path = a_star(inicio, objetivo, adj, h);

    if (path.empty()) {
        cout << "No path found" << endl;
    } else {
        cout << "Path: ";
        for (int x : path) cout << x << " ";
        cout << endl;
    }
}
```

2. ALICE SEQUENCE

```cpp
/*
 * SEQUENCE ALIGNMENT - Needleman-Wunsch
 * Alinea dos secuencias maximizando similitud.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(nm)?
 *   - Llena matriz DP de (n+1) filas  (m+1) columnas
 *   - Cada celda calcula max de 3 opciones: O(1)
 *   - Total: (n+1)  (m+1) = O(nm)
 *   - Reconstrucción del path: O(n + m)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Llena una matriz DP de nm celdas. Cada celda calcula el máximo entre
 *   3 opciones (match, delete, insert) en O(1). Total: n  m.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(nm)?
 *   - Matriz DP completa: (n+1)  (m+1) = O(nm)
 *   - Strings alineados: O(n + m)
 *   - Optimización: si solo necesitas score, usa O(min(n,m)) con 2 filas
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Matriz DP de nm. OPTIMIZACIÓN: Si solo necesitas el score final (no la
 *   alineación), puedes usar solo 2 filas: O(min(n,m)).
 */
#include <bits/stdc++.h>
using namespace std;

struct AlignmentResult {
    int score;
    string A_aligned;
    string B_aligned;
};

// Needleman-Wunsch: global alignment
AlignmentResult needlemanWunsch(const string &A, const string &B,
                                int match = 1, int mismatch = -1, int gap = -1) {
    int n = A.size();
    int m = B.size();

    // DP matrix
    vector<vector<int>> dp(n+1, vector<int>(m+1));

    // Build DP base cases
    for(int i = 0; i <= n; i++) dp[i][0] = i * gap;
    for(int j = 0; j <= m; j++) dp[0][j] = j * gap;

    // Fill DP matrix
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            int score_diag = dp[i-1][j-1] + (A[i-1] == B[j-1] ? match : mismatch);
            int score_up   = dp[i-1][j] + gap;
            int score_left = dp[i][j-1] + gap;

            dp[i][j] = max({score_diag, score_up, score_left});
        }
    }

    // --- Reconstruction ---
    string A_al, B_al;
    int i = n, j = m;

    while(i > 0 || j > 0) {
        if(i > 0 && j > 0 &&
           dp[i][j] == dp[i-1][j-1] + (A[i-1] == B[j-1] ? match : mismatch)) {
            A_al.push_back(A[i-1]);
            B_al.push_back(B[j-1]);
            i--; j--;
        }
        else if(i > 0 && dp[i][j] == dp[i-1][j] + gap) {
            A_al.push_back(A[i-1]);
            B_al.push_back('-');
            i--;
        }
        else { // j > 0
            A_al.push_back('-');
            B_al.push_back(B[j-1]);
            j--;
        }
    }

    reverse(A_al.begin(), A_al.end());
    reverse(B_al.begin(), B_al.end());

    return { dp[n][m], A_al, B_al };
}

int main() {
    string A, B;
    cin >> A >> B;

    auto res = needlemanWunsch(A, B);
```

```
    cout << "Score: " << res.score << endl;
    if (res.A_aligned.size() > 0) {
        cout << "Alineamiento:" << endl;
        cout << res.A_aligned << endl;
        cout << res.B_aligned << endl;
    }
}
/*T(n,m)=O(nm), porque */
/*E((n+1)(m+1))=O(nm)
reconstrucción usa O(n+m)*/
```

3. BELLMAN-FORD

```
/*
 * BELLMAN-FORD - Caminos mínimos con pesos negativos
 * Encuentra distancias mínimas y detecta ciclos negativos.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(VE)?
 *   - Relaja TODAS las aristas E exactamente V-1 veces
 *   - Cada relajación es O(1): compara y actualiza dist
 *   - Total: (V-1)  E = O(VE)
 *   - Paso extra para detectar ciclo negativo: +O(E)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Imagina un grafo con ciudades (V vértices) y carreteras (E aristas).
 *   Bellman-Ford garantiza encontrar el camino más corto haciendo V-1 "pasadas"
 *   completas. ¿Por qué V-1? Porque el camino más largo posible sin ciclos
 *   usa máximo V-1 aristas. En cada pasada, revisa TODAS las E aristas para
 *   ver si puede mejorar alguna distancia.
 *   Entonces: (V-1 pasadas)  (E aristas por pasada) = VE operaciones.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V)?
 *   - Arrays dist y pred: 2O(V)
 *   - Variables auxiliares: O(1)
 *   - Nota: Vector de aristas O(E) es entrada, no espacio auxiliar del algoritmo
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Solo necesita guardar: un array dist[] con V distancias (una por vértice),
 *   un array pred[] con V predecesores (para reconstruir el camino), y unas
 *   pocas variables temporales. El vector de aristas de entrada NO se cuenta
 *   porque es el problema mismo, no memoria auxiliar del algoritmo.
 */
#include <iostream>
#include <vector>
#include <climits>
#include <algorithm>
using namespace std;

void bellmanFord(const vector<vector<int>>& aristas, int n, int inicio, int fin) {
    vector<int> dist(n, INT_MAX);
    vector<int> pred(n, -1);
    dist[inicio] = 0;

    // Optimización: detener si no hay cambios
    bool cambio = true;
    for (int i = 0; i < n - 1 && cambio; i++) {
        cambio = false;
        for (const auto& e : aristas) {
            if (dist[e[0]] != INT_MAX && dist[e[0]] + e[2] < dist[e[1]]) {
                dist[e[1]] = dist[e[0]] + e[2];
                pred[e[1]] = e[0];
                cambio = true;
            }
        }
    }

    // Verificar ciclo negativo
    for (const auto& e : aristas) {
        if (dist[e[0]] != INT_MAX && dist[e[0]] + e[2] < dist[e[1]]) {
            cout << "\nERROR: Ciclo negativo detectado\n";
            return;
        }
    }

    if (dist[fin] == INT_MAX) {
        cout << "\nNo existe camino de " << inicio << " a " << fin << "\n";
        return;
    }

    // Reconstruir camino
    vector<int> camino;
    for (int v = fin; v != -1; v = pred[v])
        camino.push_back(v);

    cout << "\n==========================================\n";
    cout << "Nodo " << inicio << " -> Nodo " << fin << "\n";
    cout << "==========================================\n";
    cout << "Costo: " << dist[fin] << "\n";
    cout << "Camino: ";
    for (int i = camino.size() - 1; i >= 0; i--) {
        cout << camino[i];
        if (i > 0) cout << " -> ";
    }
    cout << "\n==========================================\n\n";
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m, inicio;
    cin >> n >> m >> inicio;

    vector<vector<int>> aristas;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        aristas.push_back({u, v, w});
    }

    vector<int> dist(n, INT_MAX);
    vector<int> pred(n, -1);
    dist[inicio] = 0;

    // Relajar aristas V-1 veces
    for (int i = 0; i < n - 1; i++) {
        for (const auto& e : aristas) {
            if (dist[e[0]] != INT_MAX && dist[e[0]] + e[2] < dist[e[1]]) {
                dist[e[1]] = dist[e[0]] + e[2];
                pred[e[1]] = e[0];
            }
        }
    }

    // Detectar ciclo negativo
    for (const auto& e : aristas) {
        if (dist[e[0]] != INT_MAX && dist[e[0]] + e[2] < dist[e[1]]) {
            cout << "\ n    CICLO NEGATIVO DETECTADO" << endl;
            cout << "Arista problema: " << e[0] << " -> " << e[1] << " (peso: " << e[2] << ")" <<
           endl;
            return 0;
        }
    }

    for (int i = 0; i < n; i++) {
        if (dist[i] == INT_MAX) {
            cout << inicio << " -> " << i << ": INF" << endl;
        } else {
            cout << inicio << " -> " << i << ": " << dist[i] << " [";
            vector<int> camino;
            for (int v = i; v != -1; v = pred[v]) camino.push_back(v);
            reverse(camino.begin(), camino.end());
            for (int j = 0; j < camino.size(); j++) {
                cout << camino[j];
                if (j < camino.size() - 1) cout << " -> ";
            }
            cout << "]" << endl;
        }
    }
    return 0;
}
/*Complejidad:

Tiempo: O(VE) en el peor caso, pero termina antes si converge
Espacio: O(V) para distancias y predecesores*/
```

4. BINARY SEARCH

```cpp
/*
 * BINARY SEARCH - Búsqueda binaria
 *
 * ¿Por qué O(log n) temporal?
 *   En cada iteración dividimos el espacio de búsqueda a la mitad.
 *   T(n) = T(n/2) + O(1)  Por teorema maestro: O(log n)
 *   Ejemplo: n=1024    log (1024) = 10 comparaciones máximo
 *
 * ¿Por qué O(1) espacial?
 *   Solo usamos 3 variables (l, r, mid) independiente del tamaño del array.
 *   No hay recursión (versión iterativa), no hay arrays auxiliares.
 *
 * EXPLICACIÓN INTUITIVA:
 *   Imagina buscar una palabra en el diccionario. En vez de leer página
 *   por página (O(n)), abres el libro por la mitad y decides si buscar
 *   en la mitad izquierda o derecha. Repites hasta encontrar la palabra.
 *   Cada decisión elimina la mitad de las opciones  log(n) pasos.
 */

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

// Búsqueda binaria estándar - retorna índice o -1
int binary_search(vector<int>& arr, int target) {
    int l = 0, r = arr.size() - 1;

    while (l <= r) {
        int mid = l + (r - l) / 2;  // evita overflow

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            l = mid + 1;  // buscar en mitad derecha
        else
            r = mid - 1;  // buscar en mitad izquierda
    }

    return -1;  // no encontrado
}

// Lower bound: primer elemento >= target
int lower_bound_custom(vector<int>& arr, int target) {
    int l = 0, r = arr.size();

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

        if (arr[mid] < target)
            l = mid + 1;
        else
            r = mid;
    }

    return l;  // índice del primer >= target
}

// Upper bound: primer elemento > target
int upper_bound_custom(vector<int>& arr, int target) {
    int l = 0, r = arr.size();

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

        if (arr[mid] <= target)
            l = mid + 1;
        else
            r = mid;
    }

    return l;  // índice del primer > target
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test 1: Binary Search estándar
    vector<int> arr = {1, 3, 5, 7, 9, 11, 13, 15};
    int target = 7;

    int idx = binary_search(arr, target);
    cout << "Binary Search: " << target << " encontrado en índice " << idx << "\n";

    // Test 2: Lower/Upper bound
    target = 7;
    int lb = lower_bound_custom(arr, target);
    int ub = upper_bound_custom(arr, target);

    cout << "Lower bound de " << target << ": índice " << lb << " (valor " << arr[lb] << ")\n";
    cout << "Upper bound de " << target << ": índice " << ub << " (valor " << arr[ub] << ")\n";

    // Test 3: Contar elementos en rango [5, 11]
    int range_l = 5, range_r = 11;
    int count = upper_bound_custom(arr, range_r) - lower_bound_custom(arr, range_l);
    cout << "Elementos en rango [" << range_l << ", " << range_r << "]: " << count << "\n";

    // Test 4: STL equivalente
    auto it_lb = lower_bound(arr.begin(), arr.end(), 7);
    auto it_ub = upper_bound(arr.begin(), arr.end(), 7);
    cout << "\nSTL lower_bound: " << (it_lb - arr.begin()) << "\n";
    cout << "STL upper_bound: " << (it_ub - arr.begin()) << "\n";

    return 0;
}
```

5. BUBBLE SORT

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

/*
 * BUBBLE SORT
 * Intercambia repetidamente elementos adyacentes fuera de orden.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n)?
 *   - Primera pasada: compara n-1 pares
 *   - Segunda pasada: compara n-2 pares
 *   - Total: (n-1) + (n-2) + ... + 1 = O(n)
 *   - Mejor caso O(n): con flag 'swapped', si ya ordenado termina en una pasada
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Imagina ordenar cartas comparando pares adyacentes. En la primera pasada
 *   comparas n-1 pares, en la segunda n-2, y así: (n-1) + (n-2) + ... + 1 =
 *   n(n-1)/2     n /2 comparaciones. Es como hacer una burbuja que sube n veces,
 *   cada vez más corta.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(1)?
 *   - Solo usa flag 'swapped' y contadores i, j
 *   - Intercambia elementos en el mismo arreglo
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Solo necesitas una variable temporal para intercambiar dos elementos y un
 *   flag para saber si hubo cambios. Todo el ordenamiento ocurre en el mismo
 *   arreglo, sin copias.
 */

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    bubbleSort(arr);

    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}
```

6. COMBINATIONS

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

/*
 * COMBINATIONS - Combinaciones de tamaño K (Backtracking)
 * Genera C(n,k) combinaciones.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(C(n,k)k)?
 *   - Número de combinaciones: C(n,k) = n!/(k!(n-k)!)
 *   - Por cada combinación, copiar k elementos: O(k)
 *   - Total: C(n,k)  k
 *   - Peor caso cuando k=n/2: C(n,n/2)  2 / n
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Genera todas las C(n,k) = n!/(k!(n-k)!) combinaciones. Copiar cada
 *   combinación cuesta O(k). Total: C(n,k)  k.
 *   Peor caso: k = n/2, donde C(n,n/2) es máximo.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(k)?
 *   - Stack de recursión: profundidad k
 *   - Vector curr: guarda k elementos en construcción
 *   - No cuenta almacenar resultado (output)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Vector curr: k elementos. Stack de recursión: profundidad k.
 *   Total: O(k).
 */

// Combinaciones de tamaño K
void generateCombinations(vector<int>& arr, int k, int start, vector<int>& curr,
        vector<vector<int>>& result) {
    if (curr.size() == k) {
        result.push_back(curr);
        return;
    }

    for (int i = start; i < arr.size(); i++) {
        curr.push_back(arr[i]);
        generateCombinations(arr, k, i + 1, curr, result);
        curr.pop_back();
    }
}

int main() {
    int n, k;
    cin >> n >> k;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    vector<vector<int>> combinations;
    vector<int> curr;
    generateCombinations(arr, k, 0, curr, combinations);

    cout << "Total de combinaciones C(" << n << "," << k << "): " << combinations.size() << endl;
    cout << "Combinaciones:" << endl;

    for (auto& comb : combinations) {
        cout << "{ ";
        for (int x : comb) cout << x << " ";
        cout << "}" << endl;
    }

    return 0;
}
```

7. COUNT INVERSIONS

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

/*
 * COUNTING INVERSIONS (Sort-and-CountInv)
 * Cuenta pares (i,j) donde i < j pero arr[i] > arr[j] usando MergeSort.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n log n)?
 *   - Divide el arreglo recursivamente: log(n) niveles (como árbol binario)
 *   - En cada nivel, procesa todos los n elementos al hacer merge
 *   - Total: n trabajo  log(n) niveles = O(n log n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Divide el arreglo recursivamente hasta tener elementos individuales
 *   (log n niveles de división). Luego hace merge de las mitades ordenadas,
 *   comparando y contando inversiones. En cada nivel procesa todos los n
 *   elementos. Total: n  log n.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - En merge, crea arrays left y right que suman O(n) elementos
 *   - Stack de recursión: log(n) llamadas, pero memoria se reutiliza
 *   - Dominante: arrays temporales O(n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Durante el merge, crea arrays temporales 'left' y 'right' que juntos
 *   suman n elementos. Aunque el stack de recursión tiene profundidad log(n),
 *   la memoria dominante son estos arrays temporales: O(n).
 */

// Conteo de inversiones usando MergeSort
long long mergeAndCount(vector<int>& arr, int l, int m, int r) {
    vector<int> left(arr.begin() + l, arr.begin() + m + 1);
    vector<int> right(arr.begin() + m + 1, arr.begin() + r + 1);

    int i = 0, j = 0, k = l;
    long long inv_count = 0;

    while (i < left.size() && j < right.size()) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
            inv_count += (left.size() - i);
        }
    }

    while (i < left.size()) arr[k++] = left[i++];
    while (j < right.size()) arr[k++] = right[j++];

    return inv_count;
}

long long mergeSortAndCount(vector<int>& arr, int l, int r) {
    long long inv_count = 0;
    if (l < r) {
        int m = l + (r - l) / 2;
        inv_count += mergeSortAndCount(arr, l, m);
        inv_count += mergeSortAndCount(arr, m + 1, r);
        inv_count += mergeAndCount(arr, l, m, r);
    }
    return inv_count;
}

int main() {
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    long long inversions = mergeSortAndCount(arr, 0, n - 1);

    cout << "Inversiones: " << inversions << endl;
    cout << "Arreglo ordenado: ";
    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}
```

8. DFS

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

/*
 * DFS - Depth First Search (Búsqueda en Profundidad)
 * Explora profundo antes de retroceder.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(V + E)?
 *   - Visita cada vértice exactamente una vez: O(V)
 *   - Por cada vértice, revisa sus aristas salientes: O(E) en total
 *   - No repite porque marca 'visited'
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Visita cada vértice exactamente una vez (gracias al array 'visited'),
 *   y por cada vértice explora todas sus aristas salientes. Como cada arista
 *   se revisa una sola vez desde su vértice origen: V visitas + E exploraciones = O(V+E).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V)?
 *   - Array 'visited' de tamaño V
 *   - Stack de recursión en peor caso (cadena): profundidad V
 *   - Vector 'order' guarda V nodos
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Array 'visited': V booleanos. Stack de recursión: en el peor caso
 *   (grafo en cadena) profundidad V. Vector 'order': guarda los V nodos
 *   visitados. Total: O(V) elementos.
 */

// DFS - Depth First Search
vector<int> adj[100005];
bool visited[100005];
vector<int> order;

void dfs(int u) {
    visited[u] = true;
    order.push_back(u);

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

int main() {
    int n, m, inicio;
    cin >> n >> m >> inicio;

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);  // Para grafo no dirigido
    }

    dfs(inicio);

    cout << "Orden DFS desde " << inicio << ": ";
    for (int x : order) cout << x << " ";
    cout << endl;

    return 0;
}
```

9. DFS + BFS

```cpp
/*
 * DFS + BFS - Recorrido de grafos
 *
 * ¿Por qué O(V+E) temporal (ambos)?
 *   DFS/BFS visitan cada vértice V exactamente una vez: O(V).
 *   Para cada vértice, recorren sus aristas adyacentes: O(E) en total.
 *   Total: O(V + E)
 *
 * ¿Por qué O(V) espacial?
 *   - Array 'visited': V booleanos
 *   - DFS recursivo: stack de recursión hasta V de profundidad
 *   - BFS: cola puede tener hasta V elementos simultáneos
 *   - Lista de adyacencia: O(V + E) pero se cuenta aparte
 *
 * EXPLICACIÓN INTUITIVA:
 *   DFS (Depth-First Search): Explorar "en profundidad". Como un laberinto,
 *   sigues un camino hasta el fondo antes de retroceder. Usa stack/recursión.
 *
 *   BFS (Breadth-First Search): Explorar "en amplitud". Como ondas en agua,
 *   visitas todos los vecinos inmediatos antes de alejarte. Usa cola (queue).
 */

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

int n;  // número de vértices
vector<vector<int>> adj;  // lista de adyacencia
vector<bool> visited;

// DFS recursivo
void dfs(int u) {
    visited[u] = true;
    cout << u << " ";

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

// BFS iterativo
void bfs(int start) {
    queue<int> q;
    vector<bool> vis(n, false);
    vector<int> dist(n, -1);

    q.push(start);
    vis[start] = true;
    dist[start] = 0;

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

        cout << u << " ";

        for (int v : adj[u]) {
            if (!vis[v]) {
                vis[v] = true;
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }

    cout << "\nDistancias desde " << start << ": ";
    for (int i = 0; i < n; i++) {
        cout << dist[i] << " ";
    }
    cout << "\n";
}

// Contar componentes conexas
int count_components() {
    visited.assign(n, false);
    int components = 0;

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i);
            components++;
            cout << "\n";
        }
    }

    return components;
}

// Detectar ciclos usando DFS
bool has_cycle_dfs(int u, int parent) {
    visited[u] = true;
```

```cpp
    for (int v : adj[u]) {
        if (!visited[v]) {
            if (has_cycle_dfs(v, u)) return true;
        } else if (v != parent) {
            return true;  // arista de retorno = ciclo
        }
    }

    return false;
}

bool has_cycle() {
    visited.assign(n, false);
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            if (has_cycle_dfs(i, -1)) return true;
        }
    }
    return false;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Crear grafo de ejemplo
    n = 7;
    adj.assign(n, vector<int>());

    // Grafo: 0-1-2-3
    //        |   |
    //        4   5-6
    adj[0] = {1, 4};
    adj[1] = {0, 2};
    adj[2] = {1, 3, 5};
    adj[3] = {2};
    adj[4] = {0};
    adj[5] = {2, 6};
    adj[6] = {5};

    cout << "=== DFS desde 0 ===\n";
    visited.assign(n, false);
    dfs(0);
    cout << "\n\n";

    cout << "=== BFS desde 0 ===\n";
    bfs(0);
    cout << "\n";

    cout << "=== COMPONENTES CONEXAS ===\n";
    int comp = count_components();
    cout << "Total componentes: " << comp << "\n\n";

    cout << "=== DETECCIÓN DE CICLOS ===\n";
    cout << (has_cycle() ? "Tiene ciclos" : "No tiene ciclos") << "\n";

    return 0;
}
```

10. DIJKSTRA

```cpp
/*
 * DIJKSTRA - Caminos mínimos desde un origen
 * Encuentra distancias mínimas en grafos con pesos NO negativos.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O((V + E) log V)?
 *   - Cada vértice se extrae del heap exactamente una vez: V  log(V)
 *   - Por cada vértice, relaja sus aristas salientes: E aristas totales
 *   - Cada relajación puede insertar en heap: log(V)
 *   - Total: Vlog(V) + Elog(V) = O((V + E) log V)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Extrae cada vértice del min-heap exactamente una vez: V  log(V).
 *   Por cada vértice, relaja sus aristas salientes: E aristas totales.
 *   Cada relajación puede insertar/actualizar en el heap: log(V).
 *   Total: Vlog(V) + Elog(V) = (V+E) log V.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V + E)?
 *   - Grafo de adyacencia: O(E) aristas + O(V) listas
 *   - Priority queue: hasta O(V) nodos simultáneos
 *   - Arrays dist y padre: 2O(V)
 *   - Dominante: O(V + E)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Grafo de adyacencia: O(E) aristas + O(V) listas. Priority queue: hasta
 *   V nodos simultáneos. Arrays dist y padre: 2V elementos. Dominante:
 *   O(V+E) del grafo.
 */
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

struct Arista {
    int destino;
    ll peso;
};

struct Nodo {
    int vertice;
    ll distancia;
    bool operator>(const Nodo& otro) const {
        return distancia > otro.distancia;
    }
};

pair<vector<ll>, vector<int>> dijkstra(int n, vector<vector<Arista>>& grafo, int inicio) {
    vector<ll> dist(n, LLONG_MAX);
    vector<int> padre(n, -1);
    vector<bool> visitado(n, false);
    priority_queue<Nodo, vector<Nodo>, greater<Nodo>> pq;

    dist[inicio] = 0;
    pq.push({inicio, 0});

    while (!pq.empty()) {
        Nodo actual = pq.top();
        pq.pop();

        int u = actual.vertice;
        if (visitado[u]) continue;
        visitado[u] = true;

        for (const Arista& e : grafo[u]) {
            int v = e.destino;
            ll peso = e.peso;

            if (dist[u] + peso < dist[v]) {
                dist[v] = dist[u] + peso;
                padre[v] = u;
                pq.push({v, dist[v]});
            }
        }
    }
    return {dist, padre};
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m, inicio;
    cin >> n >> m >> inicio;

    vector<vector<Arista>> grafo(n);

    for (int i = 0; i < m; i++) {
        int u, v;
        ll w;
        cin >> u >> v >> w;
        grafo[u].push_back({v, w});
    }

    auto [dist, padre] = dijkstra(n, grafo, inicio);

    for (int i = 0; i < n; i++) {
        if (dist[i] == LLONG_MAX) {
            cout << inicio << " -> " << i << ": INF" << endl;
        } else {
            cout << inicio << " -> " << i << ": " << dist[i] << " [";
            vector<int> camino;
            for (int v = i; v != -1; v = padre[v]) camino.push_back(v);
            reverse(camino.begin(), camino.end());
            for (int j = 0; j < camino.size(); j++) {
                cout << camino[j];
                if (j < camino.size() - 1) cout << " -> ";
            }
            cout << "]" << endl;
        }
    }
    return 0;
}
```

11. DETERMINISTIC SELECT

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

/*
 * DSELECT - Deterministic Selection (Mediana de Medianas)
 * Garantiza O(n) en peor caso usando pivote inteligente.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n) garantizado?
 *   - Divide en grupos de 5: O(n/5) medianas
 *   - Encuentra mediana de medianas recursivamente: T(n/5)
 *   - Particiona con ese pivote: O(n)
 *   - Garantiza descartar 30% mínimo: T(0.7n) en peor caso
 *   - Recurrencia: T(n) = T(n/5) + T(0.7n) + O(n)  O(n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Median-of-medians garantiza pivote decente. Divide en grupos de 5,
 *   encuentra medianas recursivamente, usa esa mediana como pivote.
 *   Recurrencia: T(n) = T(n/5) + T(7n/10) + O(n) = O(n) garantizado
 *   (Teorema Maestro).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - Stack de recursión: en peor caso O(n) de profundidad
 *   - T(n) = T(n/5) + T(0.7n) puede llevar a profundidad lineal
 *   - Grupos de 5 se procesan in-place
 *   - No crea copias grandes del arreglo
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Stack de recursión: T(n) = T(n/5) + T(0.7n) tiene profundidad O(n) en
 *   el peor caso (no es log n porque suma de fracciones 1/5 + 0.7 = 0.9 < 1
 *   pero acumula). O(n) espacial.
 */

// DSelect - Selección determinista (Mediana de Medianas)
int partition(vector<int>& arr, int l, int r, int pivotIdx) {
    int pivot = arr[pivotIdx];
    swap(arr[pivotIdx], arr[r]);
    int storeIdx = l;

    for (int i = l; i < r; i++) {
        if (arr[i] < pivot) {
            swap(arr[i], arr[storeIdx]);
            storeIdx++;
        }
    }
    swap(arr[storeIdx], arr[r]);
    return storeIdx;
}

int findMedian(vector<int>& arr, int l, int r) {
    sort(arr.begin() + l, arr.begin() + r + 1);
    return l + (r - l) / 2;
}

int medianOfMedians(vector<int>& arr, int l, int r) {
    if (r - l < 5) return findMedian(arr, l, r);

    for (int i = l; i <= r; i += 5) {
        int subRight = min(i + 4, r);
        int medianIdx = findMedian(arr, i, subRight);
        swap(arr[medianIdx], arr[l + (i - l) / 5]);
    }

    int mid = l + (r - l) / 10;
    return medianOfMedians(arr, l, l + (r - l) / 5);
}

int dselect(vector<int>& arr, int l, int r, int k) {
    if (l == r) return arr[l];

    int pivotIdx = medianOfMedians(arr, l, r);
    pivotIdx = partition(arr, l, r, pivotIdx);

    int rank = pivotIdx - l + 1;

    if (rank == k) return arr[pivotIdx];
    else if (k < rank) return dselect(arr, l, pivotIdx - 1, k);
    else return dselect(arr, pivotIdx + 1, r, k - rank);
}

int main() {
    int n, k;
    cin >> n >> k;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    int result = dselect(arr, 0, n - 1, k);

    cout << "El " << k << "-ésimo elemento más pequeño es: " << result << endl;

    return 0;
```

```
}
```

12. FRACTIONAL KNAPSACK

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

/*
 * FRACTIONAL KNAPSACK (Greedy)
 * Maximiza valor permitiendo fracciones de items.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n log n)?
 *   - Calcular ratios valor/peso: O(n)
 *   - Ordenar n items por ratio: O(n log n)
 *   - Recorrer items tomando lo que cabe: O(n)
 *   - Dominante: O(n log n) del sort
 *
 *   EXPLICACIÓN INTUITIVA:
 *   1. Calcular ratios valor/peso: recorre n items = O(n)
 *   2. Ordenar n items por ratio descendente: O(n log n)
 *   3. Selección voraz (llenando capacidad): O(n)
 *   Dominante: ordenamiento O(n log n).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - Vector de items: O(n)
 *   - Sort in-place usa stack O(log n) en promedio
 *   - Variables capacityLeft, totalValue: O(1)
 *   - Total: O(n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Vector de items: n elementos. Sort usa stack de recursión: O(log n).
 *   Dominante: O(n) del vector.
 */

// Fractional Knapsack - Greedy
struct Item {
    int value, weight;
    double ratio;
};

bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}

double fractionalKnapsack(vector<Item>& items, int W) {
    sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;
    int currWeight = 0;

    for (auto& item : items) {
        if (currWeight + item.weight <= W) {
            currWeight += item.weight;
            totalValue += item.value;
        } else {
            int remaining = W - currWeight;
            totalValue += item.value * ((double)remaining / item.weight);
            break;
        }
    }

    return totalValue;
}

int main() {
    int n, W;
    cin >> n >> W;

    vector<Item> items(n);
    for (int i = 0; i < n; i++) {
        cin >> items[i].value >> items[i].weight;
        items[i].ratio = (double)items[i].value / items[i].weight;
    }

    double maxValue = fractionalKnapsack(items, W);

    cout << fixed << setprecision(2);
    cout << "Valor máximo: " << maxValue << endl;

    return 0;
}
```

13. GREEDY BASICS

```cpp
/*
 * N-QUEENS - Backtracking
 * Coloca N reinas en tablero NN sin que se ataquen.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(N!) con poda?
 *   - Sin poda: colocar N reinas en N casillas = combinatoria masiva
 *   - Con poda: descarta inmediatamente posiciones inválidas (diagonal, fila, columna)
 *   - Primer nivel: N opciones, segundo: ~(N-2), tercero: ~(N-4)...
 *   - Peor caso: O(N!) pero poda lo reduce drásticamente a ~O(N^N) práctico
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Sin poda serían N casillas combinadas de forma masiva. Con poda
 *   (descartando diagonales, filas, columnas atacadas): Fila 1: N opciones,
 *   Fila 2: ~N-2 opciones válidas, Fila 3: ~N-4... Aproximadamente O(N!),
 *   pero la poda reduce a algo entre N! y N^N en la práctica.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(N)?
 *   - Vector tablero: almacena columna de cada reina (N enteros)
 *   - Stack de recursión: profundidad N (una llamada por fila)
 *   - Contador de soluciones: O(1)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Vector tablero: N enteros (columna de cada reina). Stack de recursión:
 *   profundidad N (una fila por nivel). Total: O(N).
 */
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// ========== ESTRUCTURA GENÉRICA DE BACKTRACKING ==========

/*
PLANTILLA GENERAL BACKTRACKING:

1. ESTADO:
   - ¿Qué variables describen la solución parcial?
   - vector<int> solucion_actual

2. CASO BASE:
   - ¿Cuándo tenemos una solución completa?
   - if (nivel == n)  procesar solución

3. GENERAR CANDIDATOS:
   - ¿Qué opciones tenemos en este nivel?
   - for cada candidato posible

4. VERIFICAR FACTIBILIDAD:
   - ¿Es válido agregar este candidato?
   - Verificar restricciones

5. RECURSIÓN:
   - ELEGIR: agregar candidato
   - EXPLORAR: llamada recursiva
   - RETROCEDER: quitar candidato (backtrack)
*/

// ========== PATRÓN 1: ESTRUCTURA BÁSICA ==========

void backtrack_basico(int nivel, int n, vector<int>& solucion, vector<bool>& usado) {
    // 1. CASO BASE: solución completa
    if (nivel == n) {
        cout << "Solución: ";
        for (int x : solucion) cout << x << " ";
        cout << endl;
        return;
    }

    // 2. GENERAR candidatos para este nivel
    for (int candidato = 0; candidato < n; candidato++) {

        // 3. VERIFICAR factibilidad
        if (!usado[candidato]) {

            // 4. ELEGIR: hacer el movimiento
            solucion.push_back(candidato);
            usado[candidato] = true;

            // 5. EXPLORAR: recursión al siguiente nivel
            backtrack_basico(nivel + 1, n, solucion, usado);

            // 6. RETROCEDER: deshacer el movimiento
            solucion.pop_back();
            usado[candidato] = false;
        }
    }
}

// ========== PATRÓN 2: CON PODA (OPTIMIZACIÓN) ==========

int mejor_valor = -1;
vector<int> mejor_solucion;

bool puede_mejorar(int nivel, int valor_actual, int valor_restante) {
    // Poda por optimalidad: ¿es posible mejorar la mejor solución?
    return (valor_actual + valor_restante) > mejor_valor;
}

void backtrack_con_poda(int nivel, int n, vector<int>& solucion,
                        int valor_actual, const vector<int>& valores) {
    // Calcular valor restante (cota superior)
    int valor_restante = 0;
    for (int i = nivel; i < n; i++) {
        valor_restante += valores[i];
    }

    // PODA: si no podemos mejorar, retornar
    if (!puede_mejorar(nivel, valor_actual, valor_restante)) {
        return;
    }

    // Caso base: solución completa
    if (nivel == n) {
        if (valor_actual > mejor_valor) {
            mejor_valor = valor_actual;
            mejor_solucion = solucion;
        }
        return;
    }

    // Probar cada opción
    for (int decision = 0; decision <= 1; decision++) {
        int nuevo_valor = valor_actual;

        if (decision == 1) {
            solucion.push_back(nivel);
            nuevo_valor += valores[nivel];
        }

        backtrack_con_poda(nivel + 1, n, solucion, nuevo_valor, valores);

        if (decision == 1) {
            solucion.pop_back();
        }
    }
}

// ========== PATRÓN 3: PERMUTACIONES ==========
// Generar todas las permutaciones de n elementos

void generar_permutaciones(int nivel, int n, vector<int>& perm, vector<bool>& usado) {
    // Caso base
    if (nivel == n) {
        for (int x : perm) cout << x << " ";
        cout << endl;
        return;
    }

    // Probar cada elemento no usado
    for (int i = 0; i < n; i++) {
        if (!usado[i]) {
            perm.push_back(i);
            usado[i] = true;

            generar_permutaciones(nivel + 1, n, perm, usado);

            perm.pop_back();
            usado[i] = false;
        }
    }
}

// ========== PATRÓN 4: SUBCONJUNTOS (SUBSET SUM) ==========
// Generar subconjuntos que cumplan una condición

void generar_subconjuntos(int pos, int n, vector<int>& subset,
                          int suma_actual, int suma_objetivo) {
    // Caso base: solución encontrada
    if (suma_actual == suma_objetivo) {
        cout << "Subconjunto: ";
        for (int x : subset) cout << x << " ";
        cout << endl;
        return;
    }

    // Poda: si nos pasamos, retornar
    if (pos >= n || suma_actual > suma_objetivo) {
        return;
    }

    // OPCIÓN 1: INCLUIR elemento actual
    subset.push_back(pos);
    generar_subconjuntos(pos + 1, n, subset, suma_actual + pos, suma_objetivo);
    subset.pop_back();

    // OPCIÓN 2: NO INCLUIR elemento actual
    generar_subconjuntos(pos + 1, n, subset, suma_actual, suma_objetivo);
}

// ========== PATRÓN 5: COMBINACIONES ==========
// Generar combinaciones de k elementos de n

void generar_combinaciones(int inicio, int k, int n, vector<int>& combinacion) {
    // Caso base: tenemos k elementos
    if (combinacion.size() == k) {
        for (int x : combinacion) cout << x << " ";
        cout << endl;
        return;
    }

    // Probar elementos desde 'inicio' hasta n
    for (int i = inicio; i < n; i++) {
        combinacion.push_back(i);
        generar_combinaciones(i + 1, k, n, combinacion);
        combinacion.pop_back();
    }
}

// ========== PATRÓN 6: N-REINAS (CON VERIFICACIÓN AVANZADA) ==========

bool es_seguro(const vector<int>& tablero, int fila, int col) {
    for (int i = 0; i < fila; i++) {
        int col_reina = tablero[i];

        // Misma columna
        if (col_reina == col) return false;

        // Diagonal
        if (abs(col_reina - col) == abs(i - fila)) return false;
    }
    return true;
}

void nreinas(int fila, int n, vector<int>& tablero, int& soluciones) {
    if (fila == n) {
        soluciones++;
        return;
    }

    for (int col = 0; col < n; col++) {
        if (es_seguro(tablero, fila, col)) {
            tablero.push_back(col);
            nreinas(fila + 1, n, tablero, soluciones);
            tablero.pop_back();
        }
    }
}

// ========== PATRÓN 7: COLOREO DE GRAFOS ==========

bool es_color_valido(const vector<vector<int>>& grafo, const vector<int>& colores,
                     int nodo, int color) {
    // Verificar que ningún vecino tenga este color
    for (int vecino : grafo[nodo]) {
        if (vecino < nodo && colores[vecino] == color) {
            return false;
        }
    }
    return true;
}

void colorear_grafo(int nodo, int n, const vector<vector<int>>& grafo,
                    vector<int>& colores, int num_colores) {
    if (nodo == n) {
        cout << "Coloreo: ";
        for (int c : colores) cout << c << " ";
        cout << endl;
        return;
    }

    for (int color = 0; color < num_colores; color++) {
        if (es_color_valido(grafo, colores, nodo, color)) {
            colores[nodo] = color;
            colorear_grafo(nodo + 1, n, grafo, colores, num_colores);
            colores[nodo] = -1;
        }
    }
}

// ========== PATRÓN 8: SUDOKU ==========

bool es_valido_sudoku(vector<vector<int>>& tablero, int fila, int col, int num) {
    // Verificar fila
    for (int j = 0; j < 9; j++) {
        if (tablero[fila][j] == num) return false;
    }

    // Verificar columna
    for (int i = 0; i < 9; i++) {
        if (tablero[i][col] == num) return false;
```

```
    }

    // Verificar subcuadro 3x3
    int inicio_fila = (fila / 3) * 3;
    int inicio_col = (col / 3) * 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (tablero[inicio_fila + i][inicio_col + j] == num) {
                return false;
            }
        }
    }

    return true;
}

bool resolver_sudoku(vector<vector<int>>& tablero) {
    // Buscar celda vacía
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (tablero[i][j] == 0) {
                // Probar números del 1 al 9
                for (int num = 1; num <= 9; num++) {
                    if (es_valido_sudoku(tablero, i, j, num)) {
                        tablero[i][j] = num;

                        if (resolver_sudoku(tablero)) {
                            return true;
                        }

                        tablero[i][j] = 0;  // Backtrack
                    }
                }
                return false;  // No hay solución
            }
        }
    }
    return true;  // Tablero completo
}

// ========== TÉCNICAS DE OPTIMIZACIÓN ==========

/*
TÉCNICAS DE PODA:

1. PODA POR FACTIBILIDAD:
   - Eliminar ramas que violan restricciones

2. PODA POR OPTIMALIDAD:
   - Eliminar ramas que no pueden mejorar la mejor solución
   - Usar cotas superiores/inferiores

3. PODA POR SIMETRÍA:
   - Evitar explorar soluciones simétricas/equivalentes

4. ORDENAR CANDIDATOS:
   - Probar primero los más prometedores
   - Encontrar soluciones buenas rápido

5. VERIFICACIÓN TEMPRANA:
   - Detectar inconsistencias lo antes posible
*/

// ========== TEMPLATE GENÉRICO UNIVERSAL ==========

template<typename Estado>
void backtrack_generico(
    Estado& estado_actual,
    bool (*es_solucion)(const Estado&),
    void (*procesar_solucion)(const Estado&),
    vector<int> (*generar_candidatos)(const Estado&),
    bool (*es_factible)(const Estado&, int),
    void (*hacer_movimiento)(Estado&, int),
    void (*deshacer_movimiento)(Estado&, int)
) {
    // Caso base
    if (es_solucion(estado_actual)) {
        procesar_solucion(estado_actual);
        return;
    }

    // Generar candidatos
    vector<int> candidatos = generar_candidatos(estado_actual);

    for (int candidato : candidatos) {
        if (es_factible(estado_actual, candidato)) {
            // Elegir
            hacer_movimiento(estado_actual, candidato);

            // Explorar
            backtrack_generico(estado_actual, es_solucion, procesar_solucion,
                               generar_candidatos, es_factible,
                               hacer_movimiento, deshacer_movimiento);

            // Retroceder
            deshacer_movimiento(estado_actual, candidato);
        }
    }
}

int main() {
    int n;
    cin >> n;

    vector<int> tablero;
    int sol_count = 0;
    nreinas(0, n, tablero, sol_count);
    cout << sol_count << endl;

    return 0;
}
```

14. GREEDY DP

```
/*
 * DYNAMIC PROGRAMMING - Ejemplo bottom-up
 * Calcula función usando tabla DP.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(ab)?
 *   - Llena matriz de a filas  b columnas
 *   - Cada celda dp[i][j] se calcula en O(1)
 *   - Total: a  b celdas  O(1) = O(ab)
 *   - No hay recalculación gracias a memoización
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Ejemplo genérico de DP bottom-up. Llena matriz de a filas  b columnas,
 *   cada celda en O(1). Total: a  b.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(ab)?
 *   - Matriz dp[][] de tamaño (a+1)  (b+1)
 *   - Optimización posible: usar solo 2 filas  O(min(a,b))
 *   - Depende si necesitas reconstruir solución o solo valor
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Matriz dp[][] de (a+1)(b+1). Optimización posible a O(min(a,b)) usando
 *   solo 2 filas si no reconstruyes solución.
 */
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 1005;
int memo[MAXN][MAXN];
bool visitado[MAXN][MAXN];

int dp_bottomup(int n, int m) {
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    for (int i = 0; i <= n; i++) dp[i][0] = 0;
    for (int j = 0; j <= m; j++) dp[0][j] = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            dp[i][j] = max({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]});
        }
    }
    return dp[n][m];
}

int main() {
    int a, b;
    cin >> a >> b;
    int resultado = dp_bottomup(a, b);
    cout << resultado << endl;
    return 0;
}
```

15. GREEDY GREEDY

```cpp
/*
 * GREEDY - Knapsack fraccionario
 * Selecciona items por mejor ratio valor/peso.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n log n)?
 *   - Calcular ratios valor/peso: recorre n items = O(n)
 *   - Ordenar n items por ratio descendente: O(n log n)
 *   - Selección voraz: recorre items ordenados = O(n)
 *   - Dominante: O(n log n) del sort
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Calcular ratios: O(n). Ordenar por ratio: O(n log n). Selección voraz: O(n).
 *   Dominante: O(n log n) del ordenamiento.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - Vector de elementos: almacena n items = O(n)
 *   - Sort in-place usa stack de recursión: O(log n)
 *   - Dominante: O(n) del vector de items
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Vector de n items. Sort usa stack O(log n). Dominante: O(n).
 */
#include <bits/stdc++.h>
using namespace std;

struct Elemento {
    int valor, peso;
    double ratio;
};

bool comparar(const Elemento& a, const Elemento& b) {
    return a.ratio > b.ratio;
}

int main() {
    int n, capacidad;

    cin >> n >> capacidad;

    vector<Elemento> elementos(n);
    for (int i = 0; i < n; i++) {
        cin >> elementos[i].valor >> elementos[i].peso;
        elementos[i].ratio = (double)elementos[i].valor / elementos[i].peso;
    }

    sort(elementos.begin(), elementos.end(), comparar);
    int valor_total = 0, usado = 0;

    for (const auto& e : elementos) {
        if (usado + e.peso <= capacidad) {
            usado += e.peso;
            valor_total += e.valor;
        }
    }
    cout << valor_total << endl;
    return 0;
}
```

16. HUFFMAN

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

/*
 * HUFFMAN CODING (Compresión - Greedy)
 * Genera códigos de longitud variable óptimos.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n log n)?
 *   - Insertar n caracteres en min-heap: n  log(n)
 *   - Extraer 2 mínimos y crear nodo interno: se repite n-1 veces
 *   - Cada extracción/inserción: O(log n)
 *   - Total: (n-1)  2    log(n) = O(n log n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   1. Insertar n caracteres en min-heap: n  log(n)
 *   2. Extraer 2 mínimos y crear nodo interno se repite n-1 veces
 *   3. Cada extracción/inserción: O(log n)
 *   Total: nlog(n) + (n-1) 2log (n) = O(n log n).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - Árbol de Huffman: 2n-1 nodos (n hojas + n-1 internos) = O(n)
 *   - Priority queue: a lo más n nodos simultáneos
 *   - Map de códigos: n entradas
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Árbol de Huffman: 2n-1 nodos (n hojas + n-1 internos). Priority queue:
 *   máximo n nodos. Map de códigos: n entradas. Total: O(n).
 */

// Huffman Coding
struct Node {
    char ch;
    int freq;
    Node *left, *right;

    Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq;
    }
};

void printCodes(Node* root, string code, map<char, string>& codes) {
    if (!root) return;

    if (!root->left && !root->right) {
        codes[root->ch] = code;
        return;
    }

    printCodes(root->left, code + "0", codes);
    printCodes(root->right, code + "1", codes);
}

map<char, string> huffmanCoding(string text) {
    map<char, int> freq;
    for (char ch : text) freq[ch]++;

    priority_queue<Node*, vector<Node*>, Compare> pq;

    for (auto p : freq) {
        pq.push(new Node(p.first, p.second));
    }

    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();

        Node* parent = new Node('\0', left->freq + right->freq);
        parent->left = left;
        parent->right = right;

        pq.push(parent);
    }

    map<char, string> codes;
    printCodes(pq.top(), "", codes);

    return codes;
}

int main() {
    string text;
    getline(cin, text);

    map<char, string> codes = huffmanCoding(text);

    cout << "Códigos Huffman:" << endl;
    for (auto p : codes) {
        cout << p.first << ": " << p.second << endl;
    }

    // Calcular longitud de codificación
    int totalBits = 0;
    for (char ch : text) {
        totalBits += codes[ch].length();
    }

    cout << "\nLongitud original: " << text.length() * 8 << " bits" << endl;
    cout << "Longitud codificada: " << totalBits << " bits" << endl;
    cout << "Compresión: " << fixed << setprecision(2)
         << (1.0 - (double)totalBits / (text.length() * 8)) * 100 << "%" << endl;

    return 0;
}
```

17. INSERTION SORT

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

/*
 * INSERTION SORT
 * Construye el arreglo ordenado insertando elementos uno a uno.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n) promedio?
 *   - Para cada elemento (n iteraciones)
 *   - Lo compara con los anteriores hasta encontrar su lugar (promedio n/2)
 *   - Total: n  n/2 = n/2 = O( n )
 *   - Mejor caso O(n): si ya está ordenado, solo hace n comparaciones
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Como ordenar cartas en tu mano: tomas una carta nueva y la insertas en
 *   su posición correcta entre las ya ordenadas. En promedio, cada carta se
 *   compara con la mitad de las anteriores (n/2). Para n cartas: n  n/2 = n/2.
 *   MEJOR CASO O(n): Si ya está ordenado, solo verificas una vez cada elemento.
 *   PEOR CASO O(n): Si está al revés, cada elemento debe retroceder hasta el inicio.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(1)?
 *   - Inserta elementos moviéndolos en el mismo arreglo
 *   - Solo usa variable 'key' para guardar elemento temporal
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Solo necesitas una variable 'key' para guardar el elemento que estás
 *   insertando. Todo ocurre en el arreglo original.
 */

void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    insertionSort(arr);

    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}
```

18. KNAPSACK

```cpp
/*
 * KNAPSACK 0/1 - Dynamic Programming
 * Maximiza valor seleccionando ítems sin repetición.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(nW)?
 *   - n ítems × W capacidades = nW celdas en la tabla DP
 *   - Cada celda calcula max de 2 opciones: O(1)
 *   - Reconstrucción de solución: O(n) backtracking
 *   - NOTA: Es pseudo-polinómico (depende del valor W, no su tamaño en bits)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Construye una tabla DP de n ítems × W capacidades. Cada celda calcula el
 *   máximo entre "tomar el ítem" o "no tomarlo" en O(1). Total: n × W celdas.
 *   PSEUDO-POLINÓMICO: W es un valor numérico, no el tamaño de su representación.
 *   Si W=1000000, no es polinómico en la entrada (que sería log(W) bits).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(nW) o O(W) optimizado?
 *   - Tabla DP 2D: matriz de n × W
 *   - Optimización: solo necesita fila anterior → O(W)
 *   - Arrays de ítems y selected: O(n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Tabla 2D de nW. OPTIMIZACIÓN: Solo necesitas la fila anterior, reducible
 *   a O(W) si no reconstruyes la solución.
 */
```

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

// Devuelve: {valor_maximo, vector con índices de ítems usados}
pair<long long, vector<int>> knapsack(
    const vector<long long>& val,
    const vector<long long>& wt,
    long long W
) {
    int n = val.size();

    // DP 1D
    vector<long long> dp(W + 1, 0);

    // Para reconstrucción: items_en[w] guardará los ítems usados para alcanzar dp[w]
    vector<vector<int>> items_en(W + 1);

    for (int i = 0; i < n; i++) {
        for (long long w = W; w >= wt[i]; w--) {
            long long tomar = dp[w - wt[i]] + val[i];
            if (tomar > dp[w]) {
                dp[w] = tomar;
                items_en[w] = items_en[w - wt[i]];
                items_en[w].push_back(i);
            }
        }
    }

    // Buscar la mejor capacidad posible (0..W)
    long long bestValue = 0;
    long long bestW = 0;
    for (long long w = 0; w <= W; w++) {
        if (dp[w] > bestValue) {
            bestValue = dp[w];
            bestW = w;
        }
    }

    // Reconstrucción final
    return {bestValue, items_en[bestW]};
}


int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    long long W;
    cin >> n >> W;

    vector<long long> value(n), weight(n);
    for (int i = 0; i < n; i++) cin >> value[i];
    for (int i = 0; i < n; i++) cin >> weight[i];

    auto res = knapsack(value, weight, W);
    cout << "Valor máximo: " << res.first << endl;
    cout << "Items seleccionados: ";
    for (int i : res.second) cout << i << " ";
    cout << endl;

    return 0;
}
```

19. KNAPSACK 0/1

```cpp
/*
 * KNAPSACK 0/1 - Mochila clásica (DP)
 *
 * ¿Por qué O(nW) temporal?
 *   Tabla DP de n items  W capacidades. Llenamos cada celda una vez: O(nW).
 *   Cada celda toma O(1): max(no_tomar, tomar).
 *
 * ¿Por qué O(nW) espacial (versión 2D)?
 *   Matriz dp[n+1][W+1] almacena soluciones de subproblemas.
 *   Se puede optimizar a O(W) usando solo 1 fila (versión 1D).
 *
 * ¿Por qué O(W) espacial (versión optimizada)?
 *   Solo necesitamos la fila anterior para calcular la actual.
 *   Usando 1 array dp[W+1] y recorriendo de derecha a izquierda.
 *
 * EXPLICACIÓN INTUITIVA:
 *   Tienes una mochila con capacidad W y n objetos (peso, valor).
 *   Para cada objeto decides: "¿lo llevo o no?". Si lo llevas, sumas su
 *   valor pero gastas su peso. Eliges la combinación que maximiza valor
 *   sin exceder capacidad. DP evita recalcular las mismas decisiones.
 */

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

// Knapsack 0/1 - Versión optimizada O(W) espacio
ll knapsack_1d(vector<ll>& w, vector<ll>& v, ll W) {
    int n = w.size();
    vector<ll> dp(W + 1, 0);

    for (int i = 0; i < n; i++) {
        // Recorrer de derecha a izquierda para no sobreescribir
        for (ll cap = W; cap >= w[i]; cap--) {
            dp[cap] = max(dp[cap], dp[cap - w[i]] + v[i]);
        }
    }

    return dp[W];
}

// Knapsack 0/1 - Con reconstrucción de solución O(nW) espacio
pair<ll, vector<int>> knapsack_2d(vector<ll>& w, vector<ll>& v, ll W) {
    int n = w.size();
    vector<vector<ll>> dp(n+1, vector<ll>(W+1, 0));

    // Llenar tabla DP
    for (int i = 1; i <= n; i++) {
        for (ll cap = 0; cap <= W; cap++) {
            // Opción 1: no tomar item i-1
            dp[i][cap] = dp[i-1][cap];

            // Opción 2: tomar item i-1 (si cabe)
            if (cap >= w[i-1]) {
                dp[i][cap] = max(dp[i][cap], dp[i-1][cap - w[i-1]] + v[i-1]);
            }
        }
    }

    // Reconstruir solución
    vector<int> items;
    ll cap = W;
    for (int i = n; i > 0; i--) {
        if (dp[i][cap] != dp[i-1][cap]) {
            items.push_back(i-1);  // item i-1 fue tomado
            cap -= w[i-1];
        }
    }

    reverse(items.begin(), items.end());
    return {dp[n][W], items};
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test: Problema clásico
    vector<ll> w = {2, 3, 4, 5};       // pesos
    vector<ll> v = {3, 4, 5, 6};       // valores
    ll W = 8;                          // capacidad

    cout << "=== KNAPSACK 0/1 ===\n";
    cout << "Capacidad: " << W << "\n";
    cout << "Items (peso, valor):\n";
    for (int i = 0; i < w.size(); i++) {
        cout << "  Item " << i << ": (" << w[i] << ", " << v[i] << ")\n";
    }

    // Versión 1D (solo valor máximo)
    ll max_val_1d = knapsack_1d(w, v, W);
    cout << "\nValor máximo (1D): " << max_val_1d << "\n";

    // Versión 2D (con reconstrucción)
    auto [max_val_2d, items] = knapsack_2d(w, v, W);
    cout << "Valor máximo (2D): " << max_val_2d << "\n";

    cout << "\nItems seleccionados: ";
    ll total_weight = 0, total_value = 0;
    for (int idx : items) {
        cout << idx << " ";
        total_weight += w[idx];
        total_value += v[idx];
    }
    cout << "\nPeso total: " << total_weight << "/" << W << "\n";
    cout << "Valor total: " << total_value << "\n";

    // Verificación detallada
    cout << "\nDetalle de items seleccionados:\n";
    for (int idx : items) {
        cout << "  Item " << idx << ": peso=" << w[idx] << ", valor=" << v[idx] << "\n";
    }

    return 0;
}
```

20. KRUSKAL (MST)

```cpp
/*
 * KRUSKAL - MST con Union-Find optimizado
 * Construye árbol de expansión mínima ordenando aristas.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(E log E)?
 *   - Ordenar E aristas: O(E log E)
 *   - Por cada arista, hacer find() y union(): O((V))  O(1) con path compression
 *   - Procesar E aristas: E  (V)    O(E)
 *   - Dominante: O(E log E) del sort
 *   - Como V    E en grafos conexos: O(E log E) = O(E log V)
 *   -  (V) = inversa de Ackermann (prácticamente constante,  4)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Ordenar E aristas por peso: E log E. Procesar cada arista con Union-Find:
 *   E     (V). Como  (V)     4 en la práctica (función Ackermann inversa),
 *   se considera casi constante. Dominante: O(E log E) del ordenamiento.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V + E)?
 *   - Union-Find: arrays parent y rank de tamaño V
 *   - Vector de aristas (entrada): O(E)
 *   - Vector MST result: O(V-1) = O(V)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Vector de aristas: E elementos. Union-Find (parent, rank): 2V elementos.
 *   Total: O(V+E).
 */
#include <bits/stdc++.h>
using namespace std;

struct Arista {
    int u, v;
    long long peso;

    // Constructor para facilitar inicialización
    Arista(int _u, int _v, long long _peso) : u(_u), v(_v), peso(_peso) {}
};

// Estructura Union-Find optimizada
struct UnionFind {
    vector<int> padre;
    vector<int> rango;
    int componentes; // Número de componentes conexas

    // Inicializar estructura
    UnionFind(int n) {
        padre.resize(n);
        rango.resize(n, 0);
        componentes = n;
        for (int i = 0; i < n; i++) {
            padre[i] = i;
        }
    }

    // FIND con compresión de caminos
    // Complejidad: O((n)) amortizado
    int find(int x) {
        if (padre[x] != x) {
            padre[x] = find(padre[x]); // Compresión: conecta directamente a la raíz
        }
        return padre[x];
    }

    // UNION by rank (unión por rango)
    // Complejidad: O((n)) amortizado
    bool unir(int x, int y) {
        int px = find(x);
        int py = find(y);

        if (px == py) return false; // Ya están en el mismo conjunto

        // Une el árbol más pequeño al más grande
        if (rango[px] < rango[py]) {
            padre[px] = py;
        } else if (rango[px] > rango[py]) {
            padre[py] = px;
        } else {
            padre[py] = px;
            rango[px]++; // Incrementa rango solo si son iguales
        }

        componentes--;
        return true;
    }

    // Verifica si dos nodos están conectados
    bool conectados(int x, int y) {
        return find(x) == find(y);
    }

    // Retorna el número de componentes conexas
```

```
    int numComponentes() {
        return componentes;
    }
};

// Comparador para ordenar aristas por peso
bool compararArista(const Arista& a, const Arista& b) {
    return a.peso < b.peso;
}

// ALGORITMO DE KRUSKAL
// Retorna el MST y el peso total
pair<vector<Arista>, long long> kruskal(int n, vector<Arista>& aristas) {
    // Inicializar Union-Find
    UnionFind uf(n);

    // Ordenar aristas por peso (menor a mayor)
    sort(aristas.begin(), aristas.end(), compararArista);

    vector<Arista> mst;
    long long pesoTotal = 0;

    // Iterar sobre las aristas ordenadas
    for (const auto& arista : aristas) {
        // Si los vértices no están conectados, añadir arista al MST
        if (uf.find(arista.u) != uf.find(arista.v)) {
            mst.push_back(arista);
            pesoTotal += arista.peso;
            uf.unir(arista.u, arista.v);

            // Optimización: terminar cuando tenemos V-1 aristas
            if ((int)mst.size() == n - 1) break;
        }
    }

    return {mst, pesoTotal};
}

// Versión alternativa que retorna si el grafo es conexo
pair<vector<Arista>, long long> kruskalConValidacion(int n, vector<Arista>& aristas, bool& esConexo)
    {
    UnionFind uf(n);
    sort(aristas.begin(), aristas.end(), compararArista);

    vector<Arista> mst;
    long long pesoTotal = 0;

    for (const auto& arista : aristas) {
        if (uf.find(arista.u) != uf.find(arista.v)) {
            mst.push_back(arista);
            pesoTotal += arista.peso;
            uf.unir(arista.u, arista.v);

            if ((int)mst.size() == n - 1) break;
        }
    }

    // Verificar si el grafo es conexo (todas las componentes unidas)
    esConexo = (uf.numComponentes() == 1);

    return {mst, pesoTotal};
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m;
    cin >> n >> m;

    vector<Arista> aristas;
    for (int i = 0; i < m; i++) {
        int u, v;
        long long w;
        cin >> u >> v >> w;
        aristas.push_back(Arista(u, v, w));
    }

    auto [mst, peso] = kruskal(n, aristas);
    cout << "Peso MST: " << peso << endl;
    cout << "Aristas:" << endl;
    for (auto& e : mst) {
        cout << e.u << " - " << e.v << " (" << e.peso << ")" << endl;
    }
    return 0;
}
```

21. LIS

```
/*
 * LIS - Longest Increasing Subsequence (Subsecuencia creciente más larga)
 *
 * ¿Por qué O(n log n) temporal?
 *   - Iteramos n elementos: O(n)
 *   - Para cada elemento hacemos binary search en dp (tamaño  n): O(log n)
 *   - Total: n  log(n) = O(n log n)
 *
 * ¿Por qué O(n) espacial?
 *   - Array dp guarda la mejor subsecuencia de cada longitud: tamaño  n
 *   - Arrays auxiliares (parent, pos) para reconstrucción: O(n)
 *
 * EXPLICACIÓN INTUITIVA:
 *   Queremos la subsecuencia creciente más larga (no consecutiva).
 *   Ejemplo: [3,1,5,2,4]  LIS = [1,2,4] longitud 3
 *
 *   Array dp[i] = el menor elemento que termina una subsecuencia de longitud i+1.
 *   Si llega un elemento x: buscamos dónde insertarlo (binary search).
 *   Si x es mayor que todos  extendemos LIS. Si no  reemplazamos un elemento
 *   para mantener dp[] lo más pequeño posible (greedy).
 */

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

// LIS O(n log n) - solo longitud
int lis_length(vector<int>& arr) {
    vector<int> dp;

    for (int x : arr) {
        auto it = lower_bound(dp.begin(), dp.end(), x);

        if (it == dp.end()) {
            dp.push_back(x);  // x es mayor que todos  extender
        } else {
            *it = x;  // reemplazar para mantener dp[] pequeño
        }
    }

    return dp.size();
}

// LIS O(n log n) - con reconstrucción de subsecuencia
pair<int, vector<int>> lis_reconstruct(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp, parent(n, -1), pos(n);

    for (int i = 0; i < n; i++) {
        auto it = lower_bound(dp.begin(), dp.end(), arr[i]);
        int idx = it - dp.begin();

        if (it == dp.end()) {
            dp.push_back(arr[i]);
        } else {
            *it = arr[i];
        }

        pos[i] = idx;  // posición en dp

        // Buscar padre (elemento anterior en LIS)
        if (idx > 0) {
            for (int j = i - 1; j >= 0; j--) {
                if (pos[j] == idx - 1 && arr[j] < arr[i]) {
                    parent[i] = j;
                    break;
                }
            }
        }
    }

    // Reconstruir subsecuencia
    vector<int> result;
    int curr = -1;

    // Encontrar último elemento del LIS
    for (int i = n - 1; i >= 0; i--) {
        if (pos[i] == (int)dp.size() - 1) {
            curr = i;
            break;
        }
    }

    // Backtrack usando parent[]
    while (curr != -1) {
        result.push_back(arr[curr]);
        curr = parent[curr];
    }

    reverse(result.begin(), result.end());
    return {dp.size(), result};
}

// LIS variante: subsecuencia decreciente más larga (LDS)
int lds_length(vector<int>& arr) {
    vector<int> dp;

    for (int x : arr) {
        // Usamos upper_bound (en vez de lower_bound) y buscamos >= x
        auto it = lower_bound(dp.begin(), dp.end(), x, greater<int>());

        if (it == dp.end()) {
            dp.push_back(x);
        } else {
            *it = x;
        }
    }

    return dp.size();
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test 1: LIS longitud
    vector<int> arr = {3, 1, 5, 2, 4, 6, 3, 7};
    cout << "=== LIS LENGTH ===\n";
    cout << "Array: ";
    for (int x : arr) cout << x << " ";
    cout << "\n";

    int len = lis_length(arr);
    cout << "Longitud LIS: " << len << "\n\n";

    // Test 2: LIS con reconstrucción
    cout << "=== LIS RECONSTRUCT ===\n";
    auto [length, lis_seq] = lis_reconstruct(arr);
    cout << "Longitud: " << length << "\n";
    cout << "Subsecuencia: ";
    for (int x : lis_seq) cout << x << " ";
    cout << "\n\n";

    // Test 3: LDS (decreciente)
    cout << "=== LDS (Longest Decreasing Subsequence) ===\n";
    int lds_len = lds_length(arr);
    cout << "Longitud LDS: " << lds_len << "\n\n";

    // Test 4: Caso especial - array ordenado
    vector<int> sorted = {1, 2, 3, 4, 5, 6, 7, 8};
    cout << "=== ARRAY ORDENADO ===\n";
    cout << "Array: ";
    for (int x : sorted) cout << x << " ";
    cout << "\n";
    cout << "LIS: " << lis_length(sorted) << " (debe ser n=" << sorted.size() << ")\n";

    return 0;
}
```

22. MODULAR POW

```cpp
/*
 * MODULAR EXPONENTIATION - Exponenciación modular (a^b mod m)
 *
 * ¿Por qué O(log b) temporal?
 *   Elevamos al cuadrado la base y dividimos exponente a la mitad en cada paso.
 *   Ejemplo: a^13 = a^(1101) requiere  log (13) = 4 operaciones.
 *   En cada paso: 1 multiplicación + 1 módulo = O(1).
 *   Total: O(log b)
 *
 * ¿Por qué O(1) espacial?
 *   Solo usamos 3 variables (result, base, exp) independiente del exponente.
 *   Versión iterativa (no recursiva)  stack constante.
 *
 * EXPLICACIÓN INTUITIVA:
 *   Calcular a^b mod m directamente: overflow si b es grande (ej: 2^1000000).
 *   Solución: usar binario del exponente. Ejemplo: 2^13 = 2^8  2^4  2^1
 *   Vamos duplicando (a mod m,  a  mod m,  a  mod m...) y multiplicamos
 *   solo los bits "encendidos" del exponente. Log pasos en vez de b pasos.
 */

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

const int MOD = 1e9 + 7;

// Exponenciación modular O(log b)
ll mod_pow(ll base, ll exp, ll mod) {
    ll result = 1;
    base %= mod;  // evitar overflow inicial

    while (exp > 0) {
        if (exp & 1) {  // bit menos significativo = 1
            result = (result * base) % mod;
        }
        base = (base * base) % mod;  // elevar al cuadrado
        exp >>= 1;  // dividir exponente entre 2
    }

    return result;
}

// Inverso modular (cuando mod es primo): a^(-1)  a^(mod-2) (mod mod)
// Basado en Pequeño Teorema de Fermat
ll mod_inv(ll a, ll mod) {
    return mod_pow(a, mod - 2, mod);
}

// División modular: (a/b) mod m = (a  b^(-1)) mod m
ll mod_div(ll a, ll b, ll mod) {
    return (a % mod * mod_inv(b, mod)) % mod;
}

// Combinatoria modular: C(n, k) mod m
ll mod_comb(ll n, ll k, ll mod) {
    if (k > n) return 0;
    if (k == 0 || k == n) return 1;

    ll num = 1, den = 1;

    // C(n,k) = n!/(k!(n-k)!) = (n  (n-1)  ...  (n-k+1)) / (k  (k-1)  ...  1)
    for (ll i = 0; i < k; i++) {
        num = (num * ((n - i) % mod)) % mod;
        den = (den * ((i + 1) % mod)) % mod;
    }

    return (num * mod_inv(den, mod)) % mod;
}

// Factorial modular (precálculo para múltiples combinatorias)
vector<ll> fact, inv_fact;

void precompute_factorials(int n, ll mod) {
    fact.resize(n + 1);
    inv_fact.resize(n + 1);

    fact[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (fact[i-1] * i) % mod;
    }

    inv_fact[n] = mod_inv(fact[n], mod);
    for (int i = n - 1; i >= 0; i--) {
        inv_fact[i] = (inv_fact[i+1] * (i+1)) % mod;
    }
}

// Combinatoria rápida O(1) tras precálculo
ll fast_comb(int n, int k, ll mod) {
    if (k > n || k < 0) return 0;
    return (fact[n] * inv_fact[k] % mod) * inv_fact[n-k] % mod;
}

// Fibonacci modular usando matriz exponenciación O(log n)
ll fib_mod(ll n, ll mod) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    // Matriz de Fibonacci: [[1,1],[1,0]]^n
    // Simplificado usando solo potencias
    ll a = 1, b = 1;
    for (ll i = 2; i < n; i++) {
        ll c = (a + b) % mod;
        a = b;
        b = c;
    }
    return b;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test 1: Exponenciación modular
    cout << "=== MODULAR EXPONENTIATION ===\n";
    ll base = 2, exp = 1000000, mod = MOD;
    ll result = mod_pow(base, exp, mod);
    cout << base << "^" << exp << " mod " << mod << " = " << result << "\n\n";

    // Test 2: Inverso modular
    cout << "=== MODULAR INVERSE ===\n";
    ll a = 3;
    ll inv = mod_inv(a, MOD);
    cout << "Inverso de " << a << " mod " << MOD << " = " << inv << "\n";
    cout << "Verificación: " << a << "  " << inv << " mod " << MOD
         << " = " << (a * inv) % MOD << "\n\n";

    // Test 3: División modular
    cout << "=== MODULAR DIVISION ===\n";
    ll dividend = 10, divisor = 3;
    ll div_result = mod_div(dividend, divisor, MOD);
    cout << dividend << " / " << divisor << " mod " << MOD << " = " << div_result << "\n\n";

    // Test 4: Combinatoria modular
    cout << "=== MODULAR COMBINATORICS ===\n";
    ll n = 10, k = 3;
    ll comb = mod_comb(n, k, MOD);
    cout << "C(" << n << ", " << k << ") mod " << MOD << " = " << comb << "\n";
    cout << "(Esperado: 120)\n\n";

    // Test 5: Combinatoria rápida con precálculo
    cout << "=== FAST COMBINATORICS (with precompute) ===\n";
    precompute_factorials(100, MOD);
    cout << "C(100, 50) mod " << MOD << " = " << fast_comb(100, 50, MOD) << "\n";
    cout << "C(20, 10) mod " << MOD << " = " << fast_comb(20, 10, MOD) << "\n\n";

    // Test 6: Fibonacci modular
    cout << "=== FIBONACCI MODULAR ===\n";
    for (int i = 0; i <= 10; i++) {
        cout << "F(" << i << ") mod " << MOD << " = " << fib_mod(i, MOD) << "\n";
    }

    return 0;
}
```

23. MWIST MAX

```cpp
/*
 * MWIS - Maximum Weighted Independent Set (en árboles)
 * Encuentra el conjunto independiente de peso máximo.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(V)?
 *   - DFS visita cada nodo exactamente una vez: O(V)
 *   - Por cada nodo, calcula dp[u][0] y dp[u][1]: O(1)
 *   - Suma pesos de hijos: O(grado) pero total = O(E) = O(V) en árbol
 *   - Reconstrucción: O(V)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   DFS visita cada nodo exactamente una vez. Por cada nodo calcula dp[u][0]
 *   (no incluirlo) y dp[u][1] (incluirlo) sumando valores de hijos. Aunque
 *   suma sobre hijos, el total de operaciones sobre todas las aristas es
 *   O(E) = O(V) en un árbol (E = V-1).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V)?
 *   - Lista de adyacencia: O(V-1) aristas en árbol = O(V)
 *   - Arrays w, dp: O(V) cada uno
 *   - Stack de recursión: profundidad del árbol  V
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Lista de adyacencia: V-1 aristas = O(V). Arrays w, dp: 2V elementos.
 *   Stack de recursión: profundidad del árbol  V. Total: O(V).
 */
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

vector<int> adj[100005];
int w[100005];
int dp[100005][2];

void dfs(int u, int p) {
    dp[u][0] = 0;
    dp[u][1] = w[u];

    for (int v : adj[u]) {
        if (v == p) continue;

        dfs(v, u);

        dp[u][1] += dp[v][0];
        dp[u][0] += max(dp[v][0], dp[v][1]);
    }
}

int main() {
    int n;
    cin >> n;

    for (int i = 0; i < n; i++) cin >> w[i];

    for (int i = 0; i < n - 1; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    dfs(0, -1);
    int maxPeso = max(dp[0][0], dp[0][1]);
    cout << "Peso máximo: " << maxPeso << endl;

    vector<int> seleccionados;
    function<void(int, int, bool)> reconstruir = [&](int u, int p, bool tomado) {
        if (tomado) seleccionados.push_back(u);
        for (int v : adj[u]) {
            if (v == p) continue;
            if (tomado) {
                reconstruir(v, u, false);
            } else {
                reconstruir(v, u, dp[v][1] > dp[v][0]);
            }
        }
    };

    reconstruir(0, -1, dp[0][1] > dp[0][0]);
    cout << "Nodos seleccionados: ";
    for (int x : seleccionados) cout << x << " ";
    cout << endl;

    return 0;
}
```

24. PERMUTATIONS

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

/*
 * PERMUTATIONS - Generación de Permutaciones (Backtracking)
 * Genera todas las n! permutaciones.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(nn!)?
 *   - Hay exactamente n! permutaciones de n elementos
 *   - Por cada permutación, copiar el vector: O(n)
 *   - Total: n!  n = O(nn!)
 *   - Árbol de decisión: nivel i tiene (n-i+1) opciones  n! hojas
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Hay n! permutaciones de n elementos. Por cada permutación, copiarla al
 *   resultado cuesta O(n). Total: n!  n = nn!.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - Stack de recursión: profundidad n
 *   - Vector arr modificado in-place: O(n)
 *   - No cuenta almacenar resultado (eso es output)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Vector arr: n elementos. Stack de recursión: profundidad n (un nivel
 *   por posición). Total: O(n) (no contamos el output que almacena las n!
 *   permutaciones).
 */

// Generación de Permutaciones
void generatePermutations(vector<int>& arr, int l, int r, vector<vector<int>>& result) {
    if (l == r) {
        result.push_back(arr);
        return;
    }

    for (int i = l; i <= r; i++) {
        swap(arr[l], arr[i]);
        generatePermutations(arr, l + 1, r, result);
        swap(arr[l], arr[i]);  // backtrack
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    vector<vector<int>> permutations;
    generatePermutations(arr, 0, n - 1, permutations);

    cout << "Total de permutaciones: " << permutations.size() << endl;
    cout << "Permutaciones:" << endl;

    for (auto& perm : permutations) {
        for (int x : perm) cout << x << " ";
        cout << endl;
    }

    return 0;
}
```

25. PREFIX SUM

```cpp
/*
 * PREFIX SUM - Sumas acumuladas
 *
 * ¿Por qué O(n) temporal (precálculo)?
 *   Recorremos el array una vez para calcular prefix[i] = prefix[i-1] + arr[i-1].
 *   n iteraciones  O(1) cada una = O(n)
 *
 * ¿Por qué O(1) temporal (query)?
 *   La suma de rango [l, r] es: prefix[r+1] - prefix[l]
 *   Solo 1 resta y 2 accesos a array  O(1) constante
 *
 * ¿Por qué O(n) espacial?
 *   Array 'prefix' de tamaño n+1 para almacenar sumas acumuladas.
 *   prefix[i] = suma de arr[0..i-1]
 *
 * EXPLICACIÓN INTUITIVA:
 *   Imagina que tienes las distancias parciales de un viaje: [3km, 5km, 2km, 7km].
 *   Las distancias acumuladas son: [0, 3, 8, 10, 17].
 *   Para saber la distancia del tramo 13: 10km - 3km = 7km (¡sin sumar de nuevo!).
 *   Prefix sum te da "distancia total hasta i" en O(1).
 */

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

// Prefix Sum 1D
class PrefixSum1D {
private:
    vector<ll> prefix;
    int n;

public:
    PrefixSum1D(vector<ll>& arr) {
        n = arr.size();
        prefix.assign(n + 1, 0);

        // Precálculo O(n)
        for (int i = 1; i <= n; i++) {
            prefix[i] = prefix[i-1] + arr[i-1];
        }
    }

    // Query O(1): suma [l, r] (0-indexed)
    ll query(int l, int r) {
        return prefix[r+1] - prefix[l];
    }

    // Suma total
    ll total() {
        return prefix[n];
    }
};

// Prefix Sum 2D (matriz)
class PrefixSum2D {
private:
    vector<vector<ll>> prefix;
    int n, m;

public:
    PrefixSum2D(vector<vector<ll>>& mat) {
        n = mat.size();
        m = mat[0].size();
        prefix.assign(n+1, vector<ll>(m+1, 0));

        // Precálculo O(n*m)
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                prefix[i][j] = mat[i-1][j-1]
                             + prefix[i-1][j]
                             + prefix[i][j-1]
                             - prefix[i-1][j-1];
            }
        }
    }

    // Query O(1): suma rectángulo [r1,c1] a [r2,c2] (0-indexed)
    ll query(int r1, int c1, int r2, int c2) {
        r1++; c1++; r2++; c2++;  // convertir a 1-indexed

        return prefix[r2][c2]
             - prefix[r1-1][c2]
             - prefix[r2][c1-1]
             + prefix[r1-1][c1-1];
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test 1: Prefix Sum 1D
    cout << "=== PREFIX SUM 1D ===\n";
    vector<ll> arr = {3, 5, 2, 7, 1, 4};
    PrefixSum1D ps1d(arr);

    cout << "Array: [3, 5, 2, 7, 1, 4]\n";
    cout << "Suma [0, 3]: " << ps1d.query(0, 3) << " (esperado: 3+5+2+7=17)\n";
    cout << "Suma [2, 4]: " << ps1d.query(2, 4) << " (esperado: 2+7+1=10)\n";
    cout << "Suma [1, 5]: " << ps1d.query(1, 5) << " (esperado: 5+2+7+1+4=19)\n";
    cout << "Suma total: " << ps1d.total() << "\n\n";

    // Test 2: Prefix Sum 2D
    cout << "=== PREFIX SUM 2D ===\n";
    vector<vector<ll>> mat = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    PrefixSum2D ps2d(mat);

    cout << "Matriz 3x4:\n";
    for (auto& row : mat) {
        for (ll x : row) cout << setw(3) << x;
        cout << "\n";
    }

    cout << "\nSuma rectángulo [0,0] a [1,1]: " << ps2d.query(0, 0, 1, 1)
         << " (esperado: 1+2+5+6=14)\n";
    cout << "Suma rectángulo [1,1] a [2,3]: " << ps2d.query(1, 1, 2, 3)
         << " (esperado: 6+7+8+10+11+12=54)\n";
    cout << "Suma completa [0,0] a [2,3]: " << ps2d.query(0, 0, 2, 3)
         << " (esperado: 78)\n";

    return 0;
}
```

26. PRIM (MST)

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

/*
 * ALGORITMO DE PRIM - MST con Priority Queue
 * Construye árbol de expansión mínima creciendo desde un nodo inicial.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O((V + E) log V)?
 *   - Cada vértice se extrae del heap una vez: V  log(V)
 *   - Por cada vértice, revisa sus vecinos: E aristas totales
 *   - Cada inserción/actualización en heap: log(V)
 *   - Total: Vlog(V) + Elog(V) = O((V + E) log V)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Similar a Dijkstra: cada vértice entra/sale del min-heap una vez: V  log(V).
 *   Relaja todas las aristas: E  log(V). Total: (V+E) log V.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V + E)?
 *   - Grafo de adyacencia: O(V + E)
 *   - Priority queue: hasta O(E) entradas (versión lazy)
 *   - Arrays enMST, padre: O(V)
 *   - Vector aristasMST: O(V-1) = O(V)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Grafo de adyacencia: O(V+E). Min-heap: hasta V elementos. Arrays enMST,
 *   padre: O(V). Dominante: O(V+E).
 */

struct Arista {
    int destino;
    ll peso;
};

// Par: {peso, vertice}
struct Nodo {
    ll peso;
    int vertice;
    bool operator>(const Nodo& otro) const {
        return peso > otro.peso; // Min-heap
    }
};

// Función mejorada de Prim con inicio aleatorio
pair<ll, vector<pair<int, int>>> prim(int n, vector<vector<Arista>>& grafo, int inicio = -1) {
    // Si no se especifica inicio, elegir uno aleatorio
    if (inicio == -1) {
        srand(time(nullptr));
        inicio = rand() % n;
    }

    vector<bool> enMST(n, false);
    priority_queue<Nodo, vector<Nodo>, greater<Nodo>> pq;

    vector<pair<int, int>> aristasMST;
    vector<int> padre(n, -1);
    ll pesoTotal = 0;

    // Iniciar desde el vértice elegido
    pq.push({0, inicio});
    padre[inicio] = inicio;

    int aristas_procesadas = 0;

    while (!pq.empty() && aristas_procesadas < n - 1) {
        Nodo actual = pq.top();
        pq.pop();

        int v = actual.vertice;
        ll peso = actual.peso;

        if (enMST[v]) continue;

        // Agregar vértice al MST
        enMST[v] = true;
        pesoTotal += peso;

        // Guardar la arista (excepto el nodo inicial)
        if (v != inicio) {
            aristasMST.push_back({padre[v], v});
            aristas_procesadas++;
        }

        // Buscar vecino de menor costo no visitado
        for (const Arista& e : grafo[v]) {
            if (!enMST[e.destino]) {
                pq.push({e.peso, e.destino});
                padre[e.destino] = v;
            }
```

```
        }
    }

    return {pesoTotal, aristasMST};
}

// ========================================
// EJEMPLO DE USO
// ========================================

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m;
    cin >> n >> m;

    vector<vector<Arista>> grafo(n);

    for (int i = 0; i < m; i++) {
        int u, v;
        ll w;
        cin >> u >> v >> w;
        grafo[u].push_back({v, w});
        grafo[v].push_back({u, w});
    }

    auto [peso, aristas] = prim(n, grafo);
    cout << "Peso MST: " << peso << endl;
    cout << "Aristas:" << endl;
    for (auto [u, v] : aristas) {
        cout << u << " - " << v << endl;
    }

    return 0;
}
```

27. QUICKSORT

```
/*
 * QUICKSORT - Divide y Conquistar
 * Ordena dividiendo el arreglo en particiones alrededor de un pivote.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n log n)?
 *   - En cada nivel hacemos O(n) comparaciones (partition)
 *   - Con buen pivote, dividimos el problema en 2 mitades: log(n) niveles
 *   - Total: n comparaciones  log(n) niveles = O(n log n)
 *   - Peor caso O(n): pivote malo crea árbol degenerado de n niveles
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Divide y conquista usando un pivote. En cada nivel del árbol de recursión
 *   procesas todos los n elementos (particionándolos). Con un buen pivote
 *   (mediana), divides el problema a la mitad cada vez, creando log(n) niveles.
 *   Total: n trabajo  log(n) niveles = n log n.
 *   PEOR CASO O(n): Si siempre eliges el peor pivote (mínimo o máximo),
 *   creas un árbol degenerado con n niveles en lugar de log(n).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(log n)?
 *   - Cada llamada recursiva usa espacio en el stack
 *   - Con buen pivote: profundidad log(n)
 *   - No crea copias del arreglo (ordena in-place)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Cada llamada recursiva usa espacio en el stack. Con pivotes balanceados,
 *   la profundidad es log(n). No crea copias del arreglo (ordena in-place),
 *   solo guarda índices y posiciones del pivote en cada nivel.
 */
#include <bits/stdc++.h>
using namespace std;

// Función de partición - Lomuto partition scheme
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// QuickSort recursivo
void quicksort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

// Hoare partition scheme (alternativa más eficiente)
int partitionHoare(vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int i = low - 1;
    int j = high + 1;

    while (true) {
        do { i++; } while (arr[i] < pivot);
        do { j--; } while (arr[j] > pivot);

        if (i >= j) return j;

        swap(arr[i], arr[j]);
    }
}

void quicksortHoare(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionHoare(arr, low, high);
        quicksortHoare(arr, low, pi);
        quicksortHoare(arr, pi + 1, high);
    }
}

// QuickSort con pivote aleatorio (evita peor caso)
int partitionRandom(vector<int>& arr, int low, int high) {
    int randomIndex = low + rand() % (high - low + 1);
    swap(arr[randomIndex], arr[high]);
    return partition(arr, low, high);
}

void quicksortRandom(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionRandom(arr, low, high);
        quicksortRandom(arr, low, pi - 1);
        quicksortRandom(arr, pi + 1, high);
    }
}

// Función auxiliar para imprimir arrays
void printArray(const vector<int>& arr, const string& label) {
    cout << label << ": ";
    for (int x : arr) cout << x << " ";
    cout << endl;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    srand(time(0));

    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    quicksort(arr, 0, arr.size() - 1);
    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}

/*
COMPLEJIDAD:
- Temporal Promedio: O(n log n)
- Temporal Peor Caso: O(n) cuando el array está ordenado y pivote es primer/último elemento
- Espacial: O(log n) por la pila de recursión

VENTAJAS:
- In-place (no requiere espacio adicional significativo)
- Rápido en la práctica
- Cache-friendly

VARIANTES:
- Lomuto: Más simple, pivote al final
- Hoare: Más eficiente, menos swaps
- Random: Evita peor caso
*/
```

28. RANDOMIZED SELECT

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

/*
 * RSELECT - Randomized Selection
 * Encuentra el k-ésimo elemento más pequeño usando pivotes aleatorios.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n) promedio?
 *   - Con pivote aleatorio, esperamos dividir en mitades aproximadas
 *   - Trabajo: n + n/2 + n/4 + ... = n(1 + 1/2 + 1/4 + ...) = 2n = O(n)
 *   - Serie geométrica converge a constante
 *   - Peor caso O(n): si siempre elige peor pivote (muy raro con aleatorio)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Con pivote aleatorio, en promedio particiona el arreglo de forma balanceada.
 *   Recurrencia: T(n) = T(n/2) + O(n) = O(n) promedio. Peor caso O(n) si
 *   siempre eliges mal pivote.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(log n) promedio?
 *   - Cada recursión usa espacio en el stack
 *   - Con buen pivote: profundidad esperada log(n)
 *   - Peor caso O(n): si pivote siempre es malo (árbol degenerado)
 *   - No crea copias, modifica arreglo in-place
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Stack de recursión con pivotes balanceados: profundidad log(n) promedio.
 *   Peor caso: O(n) si degenerado.
 */

// RSelect - Selección aleatoria del k-ésimo elemento
int partition(vector<int>& arr, int l, int r) {
    int pivotIdx = l + rand() % (r - l + 1);
    swap(arr[pivotIdx], arr[r]);
    int pivot = arr[r];
    int i = l - 1;

    for (int j = l; j < r; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[r]);
    return i + 1;
}

int rselect(vector<int>& arr, int l, int r, int k) {
    if (l == r) return arr[l];

    int pivotIdx = partition(arr, l, r);
    int rank = pivotIdx - l + 1;

    if (rank == k) return arr[pivotIdx];
    else if (k < rank) return rselect(arr, l, pivotIdx - 1, k);
    else return rselect(arr, pivotIdx + 1, r, k - rank);
}

int main() {
    srand(time(0));

    int n, k;
    cin >> n >> k;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    int result = rselect(arr, 0, n - 1, k);

    cout << "El " << k << "-ésimo elemento más pequeño es: " << result << endl;

    return 0;
}
```

29. SCC (Kosaraju)

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

/*
 * SCC - Strongly Connected Components (Algoritmo de Kosaraju)
 * Encuentra componentes fuertemente conexas.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(V + E)?
 *   - Primera DFS en grafo original: O(V + E)
 *   - Crear grafo transpuesto: O(E) invirtiendo aristas
 *   - Segunda DFS en grafo transpuesto: O(V + E)
 *   - Total: 2(V + E) + E = O(V + E)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   1. Primera DFS en grafo original: O(V+E)
 *   2. Invertir todas las aristas: O(E)
 *   3. Segunda DFS en grafo transpuesto: O(V+E)
 *   Total: 2(V+E) + E = O(V+E) (las constantes se ignoran en Big-O).
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V + E)?
 *   - Grafo original adj: O(E) aristas
 *   - Grafo transpuesto adjT: O(E) aristas
 *   - Arrays visited, component, stack: O(V)
 *   - Dominante: O(V + E)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Grafo original 'adj': E aristas. Grafo transpuesto 'adjT': E aristas.
 *   Arrays visited, component, stack: O(V). Dominante: O(V+E) por almacenar
 *   ambos grafos.
 */

// Componentes Fuertemente Conectadas (Kosaraju's Algorithm)
vector<int> adj[100005];
vector<int> adjT[100005];
bool visited[100005];
stack<int> finishStack;
int component[100005];

void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) dfs1(v);
    }
    finishStack.push(u);
}

void dfs2(int u, int comp) {
    component[u] = comp;
    for (int v : adjT[u]) {
        if (component[v] == -1) dfs2(v, comp);
    }
}

int kosaraju(int n) {
    memset(visited, false, sizeof(visited));
    memset(component, -1, sizeof(component));

    // Primera pasada: llenar stack por orden de finalización
    for (int i = 0; i < n; i++) {
        if (!visited[i]) dfs1(i);
    }

    // Segunda pasada: encontrar componentes
    int numComponents = 0;
    while (!finishStack.empty()) {
        int u = finishStack.top();
        finishStack.pop();
        if (component[u] == -1) {
            dfs2(u, numComponents);
            numComponents++;
        }
    }

    return numComponents;
}

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adjT[v].push_back(u);  // Grafo transpuesto
    }

    int numSCC = kosaraju(n);

    cout << "Número de SCCs: " << numSCC << endl;
    cout << "Componentes:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Nodo " << i << " -> Componente " << component[i] << endl;
    }

    return 0;
}
```

30. SCHEDULING

```cpp
/*
 * CPU SCHEDULING ALGORITHMS
 * Implementa FCFS, SJF, SRTF, Round Robin y Priority.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué varía según algoritmo?
 *   - FCFS: O(n log n) ordenar por arrival + O(n) simular = O(n log n)
 *   - SJF: O(n) porque por cada tiempo busca proceso más corto disponible
 *   - SRTF: O(nT) revisa todos los procesos en cada unidad de tiempo T
 *   - Round Robin: O(nT/q) donde q=quantum, peor con quantum pequeño
 *   - Priority: O(n) similar a SJF pero busca mayor prioridad
 *
 *   EXPLICACIÓN INTUITIVA:
 *   FCFS: O(n log n) - ordena por arrival time + O(n) simula = O(n log n).
 *   SJF: O(n) - por cada unidad de tiempo busca el proceso más corto disponible.
 *   SRTF: O(nT) - revisa todos los procesos en cada tick de tiempo T.
 *   Round Robin: O(nT/q) donde q es el quantum. Con quantum pequeño, muchos
 *   cambios de contexto. Priority: O(n) - similar a SJF pero busca mayor prioridad.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - Vector de procesos: almacena n procesos con sus atributos
 *   - Cola ready (Round Robin): hasta n procesos esperando
 *   - Arrays de completion/waiting time: O(n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   O(n) para todos - vector de procesos, colas ready, arrays de tiempos.
 */
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int priority;
    int remainingTime;
    int completionTime;
    int turnaroundTime;
    int waitingTime;
};

void calculateMetrics(vector<Process>& procs) {
    for (auto& p : procs) {
        p.turnaroundTime = p.completionTime - p.arrivalTime;
        p.waitingTime = p.turnaroundTime - p.burstTime;
    }
}

void printResults(const vector<Process>& procs, const string& algorithm) {
    double avgWT = 0;
    for (const auto& p : procs) avgWT += p.waitingTime;
    cout << algorithm << " - Avg WT: " << avgWT / procs.size() << endl;
}




// FCFS - First Come First Serve
void FCFS(vector<Process> processes) {
    sort(processes.begin(), processes.end(), [](const Process& a, const Process& b) {
        return a.arrivalTime < b.arrivalTime;
    });

    int currentTime = 0;
    for (auto& p : processes) {
        if (currentTime < p.arrivalTime)
            currentTime = p.arrivalTime;
        currentTime += p.burstTime;
        p.completionTime = currentTime;
    }

    calculateMetrics(processes);
    printResults(processes, "FCFS");
}

// SJF - Shortest Job First (Non-preemptive)
void SJF(vector<Process> processes) {
    int currentTime = 0;
    int n = processes.size();
    int completed = 0;

    while (completed < n) {
        int idx = -1;
        int minBurst = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (processes[i].completionTime == 0 && processes[i].arrivalTime <= currentTime) {
                if (processes[i].burstTime < minBurst) {
                    minBurst = processes[i].burstTime;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            currentTime += processes[idx].burstTime;
            processes[idx].completionTime = currentTime;
            completed++;
        } else {
            currentTime++;
        }
    }

    calculateMetrics(processes);
    printResults(processes, "SJF (Non-preemptive)");
}

// SRTF - Shortest Remaining Time First (Preemptive SJF)
void SRTF(vector<Process> processes) {
    int currentTime = 0;
    int completed = 0;
    int n = processes.size();

    while (completed < n) {
        int minIdx = -1;
        int minRemaining = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (processes[i].arrivalTime <= currentTime && processes[i].remainingTime > 0) {
                if (processes[i].remainingTime < minRemaining) {
                    minRemaining = processes[i].remainingTime;
                    minIdx = i;
                }
            }
        }

        if (minIdx == -1) {
            currentTime++;
            continue;
        }

        processes[minIdx].remainingTime--;
        currentTime++;

        if (processes[minIdx].remainingTime == 0) {
            processes[minIdx].completionTime = currentTime;
            completed++;
        }
    }

    calculateMetrics(processes);
    printResults(processes, "SRTF (Preemptive)");
}

// Round Robin
void RoundRobin(vector<Process> processes, int quantum) {
    queue<int> readyQueue;
    int currentTime = 0;
    int completed = 0;
    int n = processes.size();
    vector<bool> inQueue(n, false);

    for (int i = 0; i < n; i++) {
        if (processes[i].arrivalTime <= currentTime) {
            readyQueue.push(i);
            inQueue[i] = true;
        }
    }

    while (completed < n) {
        if (readyQueue.empty()) {
            currentTime++;
            for (int i = 0; i < n; i++) {
                if (processes[i].arrivalTime <= currentTime && !inQueue[i] &&
                    processes[i].remainingTime > 0) {
                    readyQueue.push(i);
                    inQueue[i] = true;
                }
            }
            continue;
        }

        int idx = readyQueue.front();
        readyQueue.pop();
        inQueue[idx] = false;

        int execTime = min(quantum, processes[idx].remainingTime);
        processes[idx].remainingTime -= execTime;
        currentTime += execTime;

        for (int i = 0; i < n; i++) {
            if (processes[i].arrivalTime <= currentTime && !inQueue[i] &&
                processes[i].remainingTime > 0 && i != idx) {
                readyQueue.push(i);
                inQueue[i] = true;
            }
        }

        if (processes[idx].remainingTime == 0) {
            processes[idx].completionTime = currentTime;
            completed++;
        } else {
            readyQueue.push(idx);
            inQueue[idx] = true;
        }
    }

    calculateMetrics(processes);
    printResults(processes, "Round Robin (Quantum = " + to_string(quantum) + ")");
}

// Priority Scheduling (Non-preemptive)
void PriorityScheduling(vector<Process> processes) {
    int currentTime = 0;
    int n = processes.size();
    int completed = 0;

    while (completed < n) {
        int idx = -1;
        int highestPriority = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (processes[i].completionTime == 0 && processes[i].arrivalTime <= currentTime) {
                if (processes[i].priority < highestPriority) {
                    highestPriority = processes[i].priority;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            currentTime += processes[idx].burstTime;
            processes[idx].completionTime = currentTime;
            completed++;
        } else {
            currentTime++;
        }
    }

    calculateMetrics(processes);
    printResults(processes, "Priority Scheduling");
}

int main() {
    int n;
    cin >> n;

    vector<Process> processes(n);
    for (int i = 0; i < n; i++) {
        cin >> processes[i].id >> processes[i].arrivalTime >> processes[i].burstTime >>
           processes[i].priority;
        processes[i].remainingTime = processes[i].burstTime;
    }

    FCFS(processes);
    return 0;
}
```

31. SELECTION SORT

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

/*
 * SELECTION SORT
 * Encuentra iterativamente el mínimo y lo coloca en su posición.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n)?
 *   - Primer elemento: compara con n-1 elementos
 *   - Segundo elemento: compara con n-2 elementos
 *   - Total: (n-1) + (n-2) + ... + 1 = n(n-1)/2  n/2 = O(n)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   En cada iteración buscas el mínimo entre los elementos restantes.
 *   Primera búsqueda: n-1 comparaciones, segunda: n-2, tercera: n-3...
 *   Total: (n-1) + (n-2) + ... + 1 = n/2 comparaciones. Aunque siempre
 *   hace el mismo trabajo (no mejora con datos ordenados), es más eficiente
 *   en escrituras que Bubble Sort.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(1)?
 *   - Solo usa variables auxiliares (minIdx, i, j)
 *   - Intercambia elementos directamente en el arreglo original
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Solo guardas el índice del mínimo actual. El intercambio usa una variable
 *   temporal. Ordena in-place sin memoria extra.
 */

void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        swap(arr[i], arr[minIdx]);
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    selectionSort(arr);

    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}
```

32. SIEVE

```cpp
/*
 * SIEVE OF ERATOSTHENES - Criba de Eratóstenes (números primos)
 *
 * ¿Por qué O(n log log n) temporal?
 *   Iteramos desde 2 hasta n: O( n ).
 *   Para cada primo p, marcamos múltiplos: n/p operaciones.
 *   Suma sobre primos p    n : n    (1/2 + 1/3 + 1/5 + ...) = O(n log log n)
 *
 * ¿Por qué O(n) espacial?
 *   Array booleano is_prime[n+1] para marcar primos/compuestos: O(n)
 *
 * EXPLICACIÓN INTUITIVA:
 *   Imagina una lista de números del 2 al n. Empiezas con 2 (primo),
 *   tachas todos sus múltiplos (4,6,8,...). Sigues con 3 (primo),
 *   tachas 6,9,12,... Continúas hasta n. Los números no tachados son primos.
 *
 *   Es como un "filtro": eliminas candidatos múltiplos de primos conocidos.
 */

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

// Criba básica O(n log log n)
vector<bool> sieve(int n) {
    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]) {
            // Marcar múltiplos de i como compuestos
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }

    return is_prime;
}

// Criba que retorna lista de primos
vector<int> sieve_primes(int n) {
    vector<bool> is_prime = sieve(n);
    vector<int> primes;

    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
        }
    }

    return primes;
}

// Factorización prima de n usando O(sqrt(n))
map<ll, int> prime_factors(ll n) {
    map<ll, int> factors;

    // Probar divisores hasta n
    for (ll i = 2; i * i <= n; i++) {
        while (n % i == 0) {
            factors[i]++;
            n /= i;
        }
    }

    // Si queda n > 1, es un primo
    if (n > 1) {
        factors[n]++;
    }

    return factors;
}

// Contar divisores de n usando factorización
int count_divisors(ll n) {
    auto factors = prime_factors(n);
    int count = 1;

    // Si n = p1^a1  p2^a2  ...  pk^ak
    // divisores = (a1+1)  (a2+1)  ...  (ak+1)
    for (auto [p, exp] : factors) {
        count *= (exp + 1);
    }

    return count;
}

// GCD usando Euclides O(log(min(a,b)))
ll gcd(ll a, ll b) {
    return b ? gcd(b, a % b) : a;
}

// LCM usando GCD
ll lcm(ll a, ll b) {
    return a / gcd(a, b) * b;  // evita overflow
}

// Verificar si n es primo (trial division) O(n)
bool is_prime_check(ll n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    // Probar divisores de la forma 6k  1
    for (ll i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) {
            return false;
        }
    }

    return true;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test 1: Criba hasta 50
    cout << "=== SIEVE (primos hasta 50) ===\n";
    int n = 50;
    auto primes = sieve_primes(n);
    cout << "Primos hasta " << n << ": ";
    for (int p : primes) cout << p << " ";
    cout << "\nTotal: " << primes.size() << " primos\n\n";

    // Test 2: Factorización prima
    cout << "=== PRIME FACTORIZATION ===\n";
    ll num = 360;
    auto factors = prime_factors(num);
    cout << num << " = ";
    bool first = true;
    for (auto [p, exp] : factors) {
        if (!first) cout << "  ";
        cout << p;
        if (exp > 1) cout << "^" << exp;
        first = false;
    }
    cout << "\n\n";

    // Test 3: Contar divisores
    cout << "=== COUNT DIVISORS ===\n";
    num = 360;
    int div_count = count_divisors(num);
    cout << num << " tiene " << div_count << " divisores\n\n";

    // Test 4: GCD y LCM
    cout << "=== GCD & LCM ===\n";
    ll a = 48, b = 18;
    cout << "a=" << a << ", b=" << b << "\n";
    cout << "GCD: " << gcd(a, b) << "\n";
    cout << "LCM: " << lcm(a, b) << "\n\n";

    // Test 5: Verificar primalidad
    cout << "=== PRIMALITY TEST ===\n";
    vector<ll> test_nums = {17, 91, 97, 100, 1009};
    for (ll x : test_nums) {
        cout << x << " es " << (is_prime_check(x) ? "PRIMO" : "COMPUESTO") << "\n";
    }

    return 0;
}
```

33. SUBSETS

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

/*
 * POWER SET - Generación de Subconjuntos (Máscara Binaria)
 * Genera todos los 2  subconjuntos.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O( n2 )?
 *   - Hay exactamente 2 subconjuntos (cada elemento in/out)
 *   - Por cada máscara (2 iteraciones), revisa n bits
 *   - Total: 2     n = O( n2 )
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Hay exactamente 2 subconjuntos (cada elemento puede estar o no estar).
 *   Por cada máscara de bits, revisas n bits para construir el subset.
 *   Total: 2     n.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O( n2 )?
 *   - Almacena 2  subconjuntos
 *   - Tamaño promedio de cada subset: n/2
 *   - Total: 2     n/2 = O( n2 )
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Almacenas 2  subconjuntos, cada uno con tamaño promedio n/2.
 *   Total: 2     n/2      n2  .
 */

// Generación de Subconjuntos usando Máscara Binaria
vector<vector<int>> generateSubsets(vector<int>& arr) {
    int n = arr.size();
    vector<vector<int>> subsets;

    // 2^n subconjuntos posibles
    for (int mask = 0; mask < (1 << n); mask++) {
        vector<int> subset;
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                subset.push_back(arr[i]);
            }
        }
        subsets.push_back(subset);
    }
    return subsets;
}

int main() {
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++) cin >> arr[i];

    vector<vector<int>> subsets = generateSubsets(arr);

    cout << "Total de subconjuntos: " << subsets.size() << endl;
    cout << "Subconjuntos:" << endl;

    for (auto& subset : subsets) {
        cout << "{ ";
        for (int x : subset) cout << x << " ";
        cout << "}" << endl;
    }

    return 0;
}
```

34. TOPOLOGICAL SORT

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

/*
 * TOPOLOGICAL SORT (Ordenamiento Topológico)
 * Ordena DAG respetando dependencias.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(V + E)?
 *   - Es un DFS completo que visita todos los vértices: O(V)
 *   - Y todas las aristas exactamente una vez: O(E)
 *   - Push al stack es O(1) por nodo
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Es esencialmente un DFS completo que además guarda el orden en un stack.
 *   Misma complejidad que DFS: visita V vértices y E aristas exactamente
 *   una vez. Push al stack es O(1) por nodo.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(V)?
 *   - Array 'visited': V elementos
 *   - Stack de recursión (DFS): hasta V de profundidad
 *   - topoStack guarda los V nodos
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Array 'visited': V elementos. Stack de recursión DFS: hasta V de
 *   profundidad. topoStack: almacena los V nodos. Todo proporcional a V: O(V).
 */

// Topological Sort usando DFS
vector<int> adj[100005];
bool visited[100005];
stack<int> topoStack;

void dfs(int u) {
    visited[u] = true;

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

    topoStack.push(u);
}

vector<int> topologicalSort(int n) {
    memset(visited, false, sizeof(visited));

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }

    vector<int> result;
    while (!topoStack.empty()) {
        result.push_back(topoStack.top());
        topoStack.pop();
    }

    return result;
}

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);  // Grafo dirigido
    }

    vector<int> topo = topologicalSort(n);

    cout << "Ordenamiento topológico: ";
    for (int x : topo) cout << x << " ";
    cout << endl;

    return 0;
}
```

35. TSP

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

/*
 * TSP - Travelling Salesman Problem (Backtracking con poda)
 * Encuentra el ciclo más corto visitando todas las ciudades.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(n!) peor caso?
 *   - Sin poda: prueba todas las permutaciones de n ciudades = n!
 *   - Con poda: descarta ramas cuando costo parcial  mejor solución
 *   - En práctica: mucho mejor que n! pero sigue exponencial
 *   - Para n>15, usar DP bitmask O( n2 ) o heurísticas
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Sin poda, prueba todas las permutaciones de n ciudades: n!. Con poda
 *   (descartando rutas cuando costo parcial  mejor solución), mejora
 *   considerablemente pero sigue siendo exponencial. Para n>15 es inviable;
 *   mejor usar DP con bitmask O( n2 ) o heurísticas.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(n)?
 *   - Vector path: guarda n ciudades
 *   - Vector visited: n booleanos
 *   - Stack de recursión: profundidad n
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Vector path: n ciudades. Vector visited: n booleanos. Stack de recursión:
 *   profundidad n. Total: O(n).
 */

// TSP usando Backtracking
int n;
int dist[20][20];
int minCost = INT_MAX;
vector<int> bestPath;

void tspBacktrack(vector<int>& path, vector<bool>& visited, int currCost, int count) {
    if (count == n) {
        // Regresar al inicio
        int totalCost = currCost + dist[path.back()][path[0]];
        if (totalCost < minCost) {
            minCost = totalCost;
            bestPath = path;
        }
        return;
    }

    int curr = path.back();

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            // Poda: si ya superamos el mínimo, no continuar
            if (currCost + dist[curr][i] >= minCost) continue;

            visited[i] = true;
            path.push_back(i);
            tspBacktrack(path, visited, currCost + dist[curr][i], count + 1);
            path.pop_back();
            visited[i] = false;
        }
    }
}

int main() {
    cin >> n;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> dist[i][j];
        }
    }

    vector<bool> visited(n, false);
    vector<int> path;

    visited[0] = true;
    path.push_back(0);

    tspBacktrack(path, visited, 0, 1);

    cout << "Costo mínimo: " << minCost << endl;
    cout << "Camino: ";
    for (int x : bestPath) cout << x << " ";
    cout << bestPath[0] << endl;

    return 0;
}
```

36. TWO POINTERS

```cpp
/*
 * TWO POINTERS - Técnica de dos punteros
 *
 * ¿Por qué O(n) temporal?
 *   Cada puntero (l, r) recorre el array a lo sumo una vez.
 *   l avanza: O(n), r avanza: O(n)  total O(2n) = O(n)
 *   Nunca retrocedemos ambos punteros, solo avanzamos.
 *
 * ¿Por qué O(1) espacial (típico)?
 *   Solo usamos variables l, r, sum/count (constante).
 *   Si necesitamos set/map para ventana: O(k) donde k es tamaño ventana.
 *
 * EXPLICACIÓN INTUITIVA:
 *   Dos punteros (izquierdo l, derecho r) que avanzan inteligentemente.
 *   Ejemplo: encontrar subarray con suma = target.
 *   - Si suma < target  avanzar r (agregar más elementos)
 *   - Si suma > target  avanzar l (quitar elementos)
 *   - Si suma == target  ¡encontrado!
 *   Evita probar todas las combinaciones O(n), solo O(n) movimientos.
 */

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

// 1. Subarray con suma exacta = target
pair<int,int> subarray_sum(vector<int>& arr, int target) {
    int n = arr.size();
    int l = 0, r = 0, sum = 0;

    while (r < n) {
        sum += arr[r];

        // Contraer ventana si suma > target
        while (sum > target && l <= r) {
            sum -= arr[l++];
        }

        if (sum == target) {
            return {l, r};  // encontrado [l, r]
        }

        r++;
    }

    return {-1, -1};  // no encontrado
}

// 2. Subarray con suma >= target (mínima longitud)
int min_subarray_sum(vector<int>& arr, int target) {
    int n = arr.size();
    int l = 0, r = 0, sum = 0;
    int min_len = INT_MAX;

    while (r < n) {
        sum += arr[r];

        // Contraer al máximo mientras suma >= target
        while (sum >= target && l <= r) {
            min_len = min(min_len, r - l + 1);
            sum -= arr[l++];
        }

        r++;
    }

    return min_len == INT_MAX ? -1 : min_len;
}

// 3. Sliding window: máximo subarray con k elementos únicos
int max_k_distinct(vector<int>& arr, int k) {
    int n = arr.size();
    int l = 0, r = 0;
    int max_len = 0;
    map<int, int> freq;  // frecuencia en ventana

    while (r < n) {
        freq[arr[r]]++;

        // Contraer si > k distintos
        while ((int)freq.size() > k) {
            freq[arr[l]]--;
            if (freq[arr[l]] == 0) {
                freq.erase(arr[l]);
            }
            l++;
        }

        max_len = max(max_len, r - l + 1);
        r++;
    }
```

```
    return max_len;
}

// 4. Two Sum en array ordenado (retorna valores)
pair<int,int> two_sum_sorted(vector<int>& arr, int target) {
    int l = 0, r = arr.size() - 1;

    while (l < r) {
        int sum = arr[l] + arr[r];

        if (sum == target) {
            return {arr[l], arr[r]};
        } else if (sum < target) {
            l++;  // necesitamos suma mayor
        } else {
            r--;  // necesitamos suma menor
        }
    }

    return {-1, -1};
}

// 5. Container with most water (Leetcode 11)
int max_area(vector<int>& height) {
    int l = 0, r = height.size() - 1;
    int max_water = 0;

    while (l < r) {
        int h = min(height[l], height[r]);
        int width = r - l;
        max_water = max(max_water, h * width);

        // Mover puntero del lado más bajo (greedy)
        if (height[l] < height[r]) {
            l++;
        } else {
            r--;
        }
    }

    return max_water;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test 1: Subarray con suma exacta
    cout << "=== SUBARRAY SUM ===\n";
    vector<int> arr1 = {1, 4, 2, 5, 3, 7};
    int target = 10;
    auto [l1, r1] = subarray_sum(arr1, target);
    cout << "Array: [1, 4, 2, 5, 3, 7], target=" << target << "\n";
    cout << "Subarray [" << l1 << ", " << r1 << "]: ";
    for (int i = l1; i <= r1; i++) cout << arr1[i] << " ";
    cout << "\n\n";

    // Test 2: Mínima longitud con suma >= target
    cout << "=== MIN SUBARRAY LENGTH (suma >= target) ===\n";
    vector<int> arr2 = {2, 3, 1, 2, 4, 3};
    target = 7;
    int min_len = min_subarray_sum(arr2, target);
    cout << "Array: [2, 3, 1, 2, 4, 3], target=" << target << "\n";
    cout << "Mínima longitud: " << min_len << "\n\n";

    // Test 3: Sliding window k distintos
    cout << "=== MAX SUBARRAY con K DISTINTOS ===\n";
    vector<int> arr3 = {1, 2, 1, 2, 3, 1, 1, 3};
    int k = 2;
    int max_len = max_k_distinct(arr3, k);
    cout << "Array: [1, 2, 1, 2, 3, 1, 1, 3], k=" << k << "\n";
    cout << "Máxima longitud: " << max_len << "\n\n";

    // Test 4: Two sum en array ordenado
    cout << "=== TWO SUM (sorted) ===\n";
    vector<int> arr4 = {1, 3, 5, 7, 9, 11};
    target = 12;
    auto [a, b] = two_sum_sorted(arr4, target);
    cout << "Array: [1, 3, 5, 7, 9, 11], target=" << target << "\n";
    cout << "Par encontrado: " << a << " + " << b << " = " << target << "\n\n";

    // Test 5: Container with most water
    cout << "=== CONTAINER WITH MOST WATER ===\n";
    vector<int> heights = {1, 8, 6, 2, 5, 4, 8, 3, 7};
    int water = max_area(heights);
    cout << "Heights: [1, 8, 6, 2, 5, 4, 8, 3, 7]\n";
    cout << "Máxima agua: " << water << "\n";

    return 0;
}
```

37. UNBOUNDED KNAPSACK

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

/*
 * UNBOUNDED KNAPSACK (Mochila con Repetición - DP)
 * Puedes usar cada ítem múltiples veces.
 *
 * COMPLEJIDAD TEMPORAL:
 *   ¿Por qué O(nW)?
 *   - Para cada capacidad w de 1 a W: W iteraciones
 *   - Por cada w, prueba n ítems: n iteraciones internas
 *   - Total: W  n = O(nW)
 *   - Cada celda: max de (estado anterior, agregar ítem i)
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Para cada capacidad w (de 1 a W), prueba los n ítems. Como puedes
 *   reutilizar ítems, cada celda dp[w] depende de dp[w - peso[i]].
 *   Total: W  n.
 *
 * COMPLEJIDAD ESPACIAL:
 *   ¿Por qué O(W)?
 *   - Solo necesita array dp[] de tamaño W+1
 *   - No necesita matriz 2D como 0/1 Knapsack
 *   - Sobreescribe valores porque puede reutilizar ítems
 *
 *   EXPLICACIÓN INTUITIVA:
 *   Solo un array 1D dp[] de tamaño W+1. No necesitas matriz 2D porque
 *   puedes sobreescribir (reutilizar ítems está permitido): O(W).
 */

// Unbounded Knapsack (Mochila con Repetición)
int unboundedKnapsack(vector<int>& values, vector<int>& weights, int W) {
    vector<int> dp(W + 1, 0);

    for (int w = 1; w <= W; w++) {
        for (int i = 0; i < values.size(); i++) {
            if (weights[i] <= w) {
                dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
            }
        }
    }

    return dp[W];
}

int main() {
    int n, W;
    cin >> n >> W;

    vector<int> values(n), weights(n);
    for (int i = 0; i < n; i++) cin >> values[i];
    for (int i = 0; i < n; i++) cin >> weights[i];

    int maxValue = unboundedKnapsack(values, weights, W);

    cout << "Valor máximo: " << maxValue << endl;

    return 0;
}
```

38. UNION-FIND

```cpp
/*
 * UNION-FIND (DSU - Disjoint Set Union)
 *
 * ¿Por qué O( (n))    O(1) temporal por operación?
 *   Con Path Compression + Union by Rank:
 *   -  (n) = función inversa de Ackermann (crece extremadamente lento)
 *   - Para n   10^80:  (n)    4     prácticamente constante
 *   - Amortizado: secuencia de m operaciones toma O(m  (n))
 *
 * ¿Por qué O(n) espacial?
 *   - Array parent[n]: almacena padre de cada nodo
 *   - Array rank[n]: altura aproximada de cada árbol
 *   Total: 2n = O(n)
 *
 * EXPLICACIÓN INTUITIVA:
 *   Estructura para manejar conjuntos disjuntos dinámicamente.
 *   Operaciones: find(x) = ¿a qué conjunto pertenece x?
 *                unite(x, y) = unir conjuntos de x e y
 *
 *   Optimizaciones:
 *   - Path Compression: al buscar raíz, aplana el árbol (todos apuntan a raíz)
 *   - Union by Rank: unir árbol pequeño bajo el grande (evita árboles altos)
 *
 *   Aplicaciones: Kruskal MST, componentes dinámicas, detección de ciclos.
 */

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

class UnionFind {
private:
    vector<int> parent;
    vector<int> rank;
    int components;

public:
    UnionFind(int n) : parent(n), rank(n, 0), components(n) {
        // Inicialmente cada nodo es su propio padre
        iota(parent.begin(), parent.end(), 0);
    }

    // Find con Path Compression O((n))
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // path compression
        }
        return parent[x];
    }

    // Union by Rank O((n))
    bool unite(int x, int y) {
        int px = find(x);
        int py = find(y);

        if (px == py) return false;  // ya están en el mismo conjunto

        // Unir árbol de menor rank bajo el de mayor rank
        if (rank[px] < rank[py]) {
            swap(px, py);
        }

        parent[py] = px;

        // Si ranks iguales, aumentar rank del nuevo root
        if (rank[px] == rank[py]) {
            rank[px]++;
        }

        components--;
        return true;
    }

    // Verificar si x e y están conectados
    bool connected(int x, int y) {
        return find(x) == find(y);
    }

    // Número de componentes conexas
    int count_components() {
        return components;
    }

    // Tamaño del conjunto que contiene x
    int component_size(int x) {
        int root = find(x);
        int size = 0;
        for (int i = 0; i < (int)parent.size(); i++) {
            if (find(i) == root) size++;
        }
        return size;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // Test 1: Operaciones básicas
    cout << "=== UNION-FIND BÁSICO ===\n";
    int n = 10;
    UnionFind uf(n);

    cout << "Componentes iniciales: " << uf.count_components() << "\n";

    // Unir algunos nodos
    uf.unite(0, 1);
    uf.unite(2, 3);
    uf.unite(0, 2);  // ahora {0,1,2,3} están conectados
    uf.unite(5, 6);

    cout << "Después de 4 uniones: " << uf.count_components() << " componentes\n\n";

    // Test 2: Conectividad
    cout << "=== QUERIES DE CONECTIVIDAD ===\n";
    cout << "¿0 y 3 conectados? " << (uf.connected(0, 3) ? "SÍ" : "NO") << "\n";
    cout << "¿0 y 5 conectados? " << (uf.connected(0, 5) ? "SÍ" : "NO") << "\n";
    cout << "¿5 y 6 conectados? " << (uf.connected(5, 6) ? "SÍ" : "NO") << "\n\n";

    // Test 3: Componentes conexas
    cout << "=== COMPONENTES CONEXAS ===\n";
    map<int, vector<int>> components;
    for (int i = 0; i < n; i++) {
        components[uf.find(i)].push_back(i);
    }

    int comp_num = 1;
    for (auto& [root, nodes] : components) {
        cout << "Componente " << comp_num++ << ": {";
        for (int i = 0; i < (int)nodes.size(); i++) {
            cout << nodes[i];
            if (i < (int)nodes.size() - 1) cout << ", ";
        }
        cout << "}\n";
    }
    cout << "\n";

    // Test 4: Kruskal MST (aplicación típica)
    cout << "=== APLICACIÓN: KRUSKAL MST ===\n";

    struct Edge {
        int u, v, w;
        bool operator<(const Edge& e) const { return w < e.w; }
    };

    n = 6;
    vector<Edge> edges = {
        {0, 1, 4}, {0, 2, 3}, {1, 2, 1},
        {1, 3, 2}, {2, 3, 4}, {3, 4, 2},
        {4, 5, 6}
    };

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

    UnionFind mst_uf(n);
    vector<Edge> mst;
    int total_weight = 0;

    for (auto& e : edges) {
        if (mst_uf.unite(e.u, e.v)) {
            mst.push_back(e);
            total_weight += e.w;
        }
    }

    cout << "Grafo con " << n << " nodos, " << edges.size() << " aristas\n";
    cout << "MST aristas:\n";
    for (auto& e : mst) {
        cout << "  " << e.u << " - " << e.v << " (peso " << e.w << ")\n";
    }
    cout << "Peso total MST: " << total_weight << "\n\n";

    // Test 5: Detección de ciclos
    cout << "=== DETECCIÓN DE CICLOS ===\n";
    UnionFind cycle_uf(4);
    vector<pair<int,int>> graph_edges = {{0,1}, {1,2}, {2,0}, {2,3}};

    for (auto [u, v] : graph_edges) {
        if (cycle_uf.connected(u, v)) {
            cout << "CICLO detectado al agregar arista " << u << "-" << v << "\n";
        } else {
            cycle_uf.unite(u, v);
            cout << "Arista " << u << "-" << v << " agregada (sin ciclo)\n";
        }
    }

    return 0;
}
```

39. INPUT PARSING

```cpp
/*
 * INPUT PARSING - Técnicas de lectura de entrada
 *
 * COMPLEJIDAD:
 *   - getline + stringstream: O(n) donde n es el largo de la línea
 *   - cin >> : O(1) por token
 *
 * EXPLICACIÓN:
 *   Diferentes técnicas para leer entrada en competitivas:
 *   1. cin básico (separado por espacios)
 *   2. getline + stringstream (líneas completas)
 *   3. scanf (más rápido para gran volumen)
 *   4. Fast I/O (ios::sync_with_stdio(false))
 */

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

// ==================== TÉCNICA 1: GETLINE + STRINGSTREAM ====================
// Útil para: líneas con formato variable, parsing complejo
void tecnica_getline_stringstream() {
    cout << "=== GETLINE + STRINGSTREAM ===\n";
    cout << "Ingresa líneas 'articulo palabra num1 num2' (Ctrl+D para terminar):\n";

    struct Registro {
        string nombre;
        int a, b;
    };

    vector<Registro> registros;
    string linea;

    while (getline(cin, linea)) {
        istringstream iss(linea);
        string articulo, palabra;
        Registro r;

        if (iss >> articulo >> palabra >> r.a >> r.b) {
            r.nombre = articulo + " " + palabra;
            registros.push_back(r);
        }
    }

    cout << "\nLeído " << registros.size() << " registros:\n";
    for (const auto& r : registros) {
        cout << r.nombre << " -> " << r.a << ", " << r.b << "\n";
    }
}

// ==================== TÉCNICA 2: CIN BÁSICO ====================
// Útil para: entrada simple separada por espacios/newlines
void tecnica_cin_basico() {
    cout << "=== CIN BÁSICO ===\n";

    int n;
    cout << "Cantidad de elementos: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Ingresa " << n << " números: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "Array leído: ";
    for (int x : arr) cout << x << " ";
    cout << "\n";
}

// ==================== TÉCNICA 3: MÚLTIPLES CASOS DE PRUEBA ====================
// Patrón común: leer T casos
void tecnica_multiples_casos() {
    cout << "=== MÚLTIPLES CASOS ===\n";

    int t;
    cout << "Número de casos de prueba: ";
    cin >> t;

    while (t--) {
        int n;
        cin >> n;

        vector<int> arr(n);
        for (int i = 0; i < n; i++) {
            cin >> arr[i];
        }

        // Procesar caso
        int suma = 0;
        for (int x : arr) suma += x;

        cout << "Suma: " << suma << "\n";
    }
}

// ==================== TÉCNICA 4: LEER HASTA EOF ====================
// Útil cuando no se sabe cuántos datos vienen
void tecnica_eof() {
    cout << "=== LEER HASTA EOF ===\n";

    int x;
    vector<int> numeros;

    cout << "Ingresa números (Ctrl+D para terminar): ";
    while (cin >> x) {
        numeros.push_back(x);
    }

    cout << "Leídos " << numeros.size() << " números\n";
}

// ==================== TÉCNICA 5: LEER MATRIZ ====================
void tecnica_matriz() {
    cout << "=== LEER MATRIZ ===\n";

    int n, m;
    cout << "Dimensiones (n m): ";
    cin >> n >> m;

    vector<vector<int>> mat(n, vector<int>(m));

    cout << "Ingresa matriz " << n << "x" << m << ":\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> mat[i][j];
        }
    }

    cout << "Matriz leída:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cout << mat[i][j] << " ";
        }
        cout << "\n";
    }
}

// ==================== TÉCNICA 6: LEER GRAFO ====================
void tecnica_grafo() {
    cout << "=== LEER GRAFO ===\n";

    int n, m;  // n = nodos, m = aristas
    cout << "Nodos y aristas (n m): ";
    cin >> n >> m;

    vector<vector<int>> adj(n);

    cout << "Ingresa " << m << " aristas (u v):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);  // grafo no dirigido
    }

    cout << "Lista de adyacencia:\n";
    for (int i = 0; i < n; i++) {
        cout << i << ": ";
        for (int v : adj[i]) cout << v << " ";
        cout << "\n";
    }
}

// ==================== TÉCNICA 7: GETLINE PURO (STRINGS COMPLETOS) ====================
void tecnica_getline_puro() {
    cout << "=== GETLINE PURO ===\n";

    cin.ignore();  // limpiar buffer después de cin >>

    int n;
    cout << "Cantidad de líneas: ";
    cin >> n;
    cin.ignore();  // importante!

    vector<string> lineas;

    cout << "Ingresa " << n << " líneas:\n";
    for (int i = 0; i < n; i++) {
        string linea;
        getline(cin, linea);
        lineas.push_back(linea);
    }

    cout << "Líneas leídas:\n";
    for (int i = 0; i < lineas.size(); i++) {
        cout << i << ": " << lineas[i] << "\n";
    }
}

// ==================== TÉCNICA 8: FAST I/O ====================
void tecnica_fast_io() {
    cout << "=== FAST I/O ===\n";

    // Poner al inicio del main():
    // ios::sync_with_stdio(false);
    // cin.tie(nullptr);

    cout << "Fast I/O configurado (ver código)\n";
    cout << "Usar solo cin/cout, NO mezclar con scanf/printf\n";
}

// ==================== TÉCNICA 9: PARSING CON DELIMITADORES ====================
void tecnica_delimitadores() {
    cout << "=== PARSING CON DELIMITADORES ===\n";

    string entrada = "10,20,30,40,50";
    cout << "String: " << entrada << "\n";

    vector<int> numeros;
    stringstream ss(entrada);
    string token;

    while (getline(ss, token, ',')) {  // delimitador ','
        numeros.push_back(stoi(token));
    }

    cout << "Números parseados: ";
    for (int x : numeros) cout << x << " ";
    cout << "\n";
}

// ==================== TÉCNICA 10: LEER PARES ====================
void tecnica_pares() {
    cout << "=== LEER PARES ===\n";

    int n;
    cout << "Cantidad de pares: ";
    cin >> n;

    vector<pair<int,int>> pares;

    cout << "Ingresa " << n << " pares (x y):\n";
    for (int i = 0; i < n; i++) {
        int x, y;
        cin >> x >> y;
        pares.push_back({x, y});
    }

    cout << "Pares leídos:\n";
    for (auto [x, y] : pares) {
        cout << "(" << x << ", " << y << ")\n";
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cout << "TÉCNICAS DE INPUT PARSING\n";
    cout << "=========================\n\n";

    // Descomentar la técnica que quieras probar:

    // tecnica_getline_stringstream();
    // tecnica_cin_basico();
    // tecnica_multiples_casos();
    // tecnica_eof();
    // tecnica_matriz();
    // tecnica_grafo();
    // tecnica_getline_puro();
    // tecnica_fast_io();
    // tecnica_delimitadores();
    // tecnica_pares();

    // Ejemplo simple del enunciado:
    cout << "Ejemplo: Leer 'articulo palabra num1 num2'\n";
    cout << "Ingresa líneas (Ctrl+D para terminar):\n";

    struct Registro {
        string nombre;
        int a, b;
    };

    vector<Registro> registros;
    string linea;

    while (getline(cin, linea)) {
        istringstream iss(linea);
        string articulo, palabra;
        Registro r;

        if (iss >> articulo >> palabra >> r.a >> r.b) {
```

```
        r.nombre = articulo + " " + palabra;
        registros.push_back(r);
    }
}
```

```
    cout << "\nResultado:\n";
    for (const auto& r : registros) {
        cout << r.nombre << " -> " << r.a << ", " << r.b << "\n";
```

```
    }

    return 0;
}
```

ÍNDICE DE ALGORITMOS

Complejidades por Límites

n ≤ 10	O(n!) Backtracking, Permutations
n ≤ 20	O(2^n) Subsets, Bitmask DP
n ≤ 500	O(n^3) Floyd-Warshall
n ≤ 5000	O(n^2) DP cuadratico
n ≤ 10^5	O(n log n) Sort, Dijkstra, Binary Search
n ≤ 10^6	O(n) Two Pointers, Prefix Sum, DFS
n ≤ 10^9	O(log n) Binary Search, Mod Pow, GCD

Template Base

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
using pii = pair<int,int>;
using vi = vector<int>;
#define pb push_back
#define all(x) (x).begin(), (x).end()
#define sz(x) (int)(x).size()
const int MOD = 1e9 + 7;
void solve() { }
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t = 1; // cin >> t;
    while(t--) solve();
    return 0;
}
```