Grafos II

Alex Blanchard Ortiz

De dónde venimos

Ya sabes hacer (Grafos I)	Hoy le agregamos
DFS y BFS sobre una lista de adyacencia	el mismo recorrido, pero con el grafo **escondido en una grilla**
distancia mínima con aristas de peso 1	distancia mínima cuando **cada arista cuesta distinto**
contar componentes conexas con un DFS	**mantener** las componentes mientras van **llegando** aristas
saber si un grafo es conexo	**conectarlo entero al mínimo costo**

Todo lo de hoy se apoya en el DFS y el BFS de la clase pasada. No hay ningún recorrido nuevo que aprender: hay grafos nuevos donde aplicarlos.

Ruta de la clase

Bloque 1 — el grafo escondido

- Una grilla es un grafo, aunque no lo parezca
- Flood fill: contar componentes sin construir la lista de adyacencia
- BFS en grilla y mapas de distancia

Bloque 2 — cuando las aristas cuestan

- Grafos ponderados y por qué el BFS deja de servir
- Dijkstra, su invariante y cómo se rompe con pesos negativos
- Reconstruir el camino, no solo su largo

Bloque 3 — conjuntos y árboles mínimos

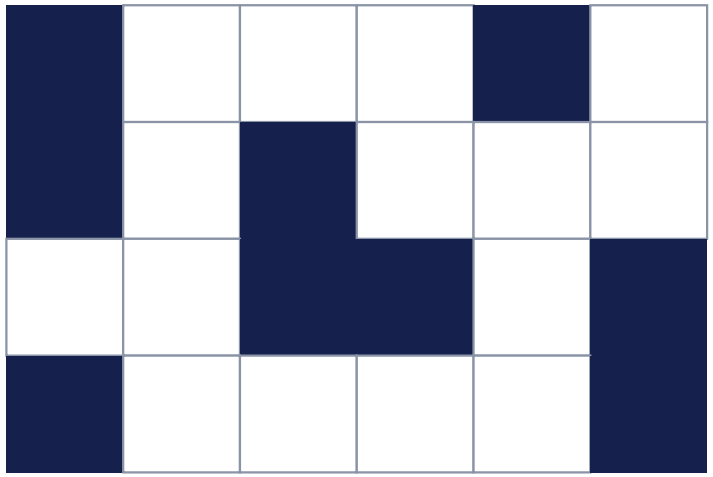
- Union-Find (DSU): componentes que cambian en el tiempo
- Árbol de cobertura mínimo y el algoritmo de Kruskal

Bloque 1 · Grillas y flood fill

1 · Grillas | 2 · Dijkstra | 3 · DSU y MST

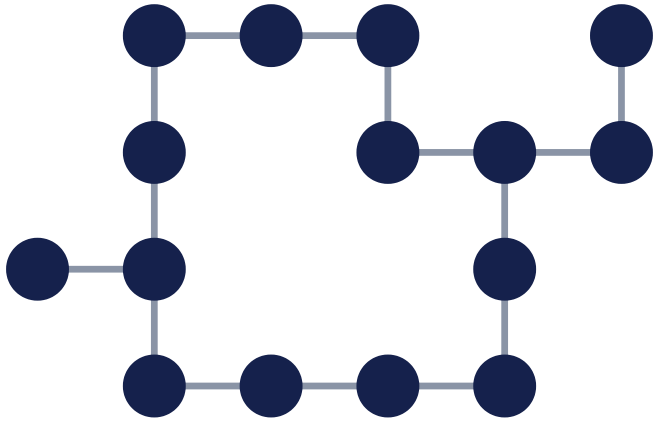
El grafo escondido en la matriz

el mapa que te dan



es el mismo objeto

el grafo que en realidad es



Cada celda **libre** es un vértice. Dos celdas libres que se tocan por un lado son una **arista**.

¿Y la lista de adyacencia?

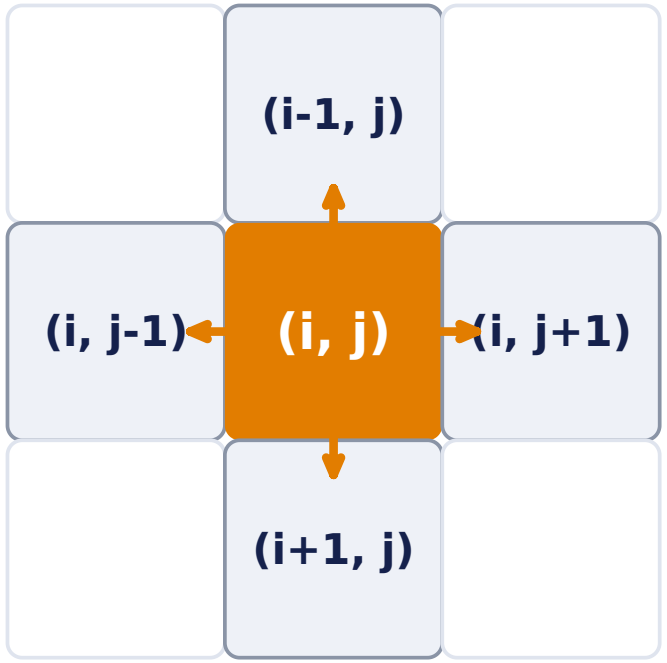
No se construye. **Nunca.**

Los vecinos de la celda (i, j) siempre son los mismos cuatro: arriba, abajo, izquierda y derecha. Calcularlos al vuelo es más barato que guardarlos.

> La grilla **es** la lista de adyacencia. El mapa que ya leíste te dice quién es vecino de quién.

Una grilla de 1000×1000 tiene 10^6 celdas y cada una mira 4 vecinos: $4 \cdot 10^6$ operaciones en total. Muy por debajo de las 10^8 que alcanzamos a hacer en un segundo.

Los vecinos: el truco de dx/dy




```
int dx[4] = {-1, 1,  0, 0};       // arriba, abajo, izquierda, derecha
int dy[4] = { 0, 0, -1, 1};       // dx cambia la fila, dy cambia la columna

for (int d = 0; d < 4; ++d) {
    int ni = i + dx[d], nj = j + dy[d];   // (ni, nj) es un vecino de (i, j)
}
```

La trampa: salirse del mapa

`i + dx[d]` puede darte -1, y `g[-1][j]` lee memoria que no es tuya. **Es el bug número uno de todo este bloque.**

```
const int MAXN = 1005;            // n, m <= 1000

int n, m;
char g[MAXN][MAXN];               // el mapa
bool vis[MAXN][MAXN];             // ¿ya pase por esta celda?

bool dentro(int i, int j) {       // el guardia de todo el bloque
    return 0 <= i && i < n && 0 <= j && j < m;
}
```

> El orden importa: primero preguntas si la celda **existe**, y recién después miras qué hay adentro.

Ejercicio en vivo 1 — Counting Rooms

Te dan el mapa de un edificio de $n \times m$ celdas. Cada celda es **piso** (`.`) o **muro** (`#`). Te puedes mover en las cuatro direcciones, siempre por el piso.

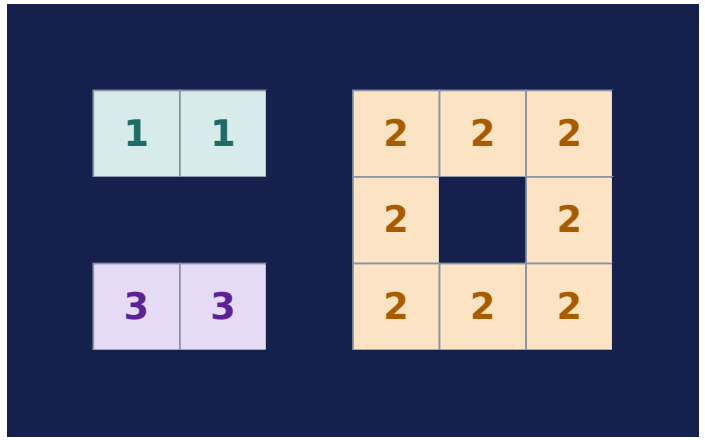
¿Cuántas habitaciones tiene el edificio?

```
Entrada          Salida
5 8              3
########
#..#...#
####.#.#
#..#...#
########
```

CSES 1192 — Counting Rooms. Restricciones: $1 \leq n, m \leq 1000$. Quedó pendiente de la clase pasada.

Paso 1: modelar

- celda de **piso** → vértice · dos celdas de piso **pegadas** → arista
- una **habitación** → una **componente conexa** de ese grafo



3 componentes conexas = 3 habitaciones

Paso 2: la idea

Contar habitaciones **es** contar componentes conexas. Y eso ya lo hicimos la clase pasada:

> Recorro todas las celdas. Cada vez que encuentro una de piso **sin visitar**, lanzo un recorrido que se come la habitación entera y sumo 1.

Es exactamente el mismo `for` de Grafos I. Lo único que cambia es de dónde salen los vecinos: antes eran `ady[u]`, ahora son las cuatro direcciones.

Al recorrido que pinta una región completa de una grilla se le dice **flood fill** (el balde de pintura del Paint).

Paso 3: leer el mapa

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

const int MAXN = 1005;
int n, m;
char g[MAXN][MAXN];
bool vis[MAXN][MAXN];

int main() {
    cin >> n >> m;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            cin >> g[i][j];
}
```

Leer `char` por `char` con `cin` se salta solo los espacios y los saltos de línea, así que no hay que preocuparse por el fin de cada fila.

Paso 4: el flood fill

```
int dx[4] = {-1, 1,  0, 0};
int dy[4] = { 0, 0, -1, 1};

void llenar(int i, int j) {
    if (!dentro(i, j)) return;        // me sali del mapa
    if (g[i][j] == '#') return;       // es muro
    if (vis[i][j]) return;            // ya estuve aca

    vis[i][j] = true;
    for (int d = 0; d < 4; ++d)
        llenar(i + dx[d], j + dy[d]);
}
```

Las tres preguntas del principio son el DFS entero. Sin ellas, la recursión no termina nunca.

Paso 5: contar las habitaciones

```
int habitaciones = 0;

for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        if (g[i][j] == '.' && !vis[i][j]) {
            llenar(i, j);              // se come la habitacion completa
            habitaciones++;
        }

cout << habitaciones << endl;
```

Complejidad: $O(n \cdot m)$. Cada celda se visita una vez y mira 4 vecinos. Con $n = m = 1000$ son $4 \cdot 10^6$ operaciones.

Solución completa

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

const int MAXN = 1005;
int n, m;
char g[MAXN][MAXN];
bool vis[MAXN][MAXN];
int dx[4] = {-1, 1, 0, 0}, dy[4] = {0, 0, -1, 1};

bool dentro(int i, int j) { return 0 <= i && i < n && 0 <= j && j < m; }

void llenar(int i, int j) {
    if (!dentro(i, j) || g[i][j] == '#' || vis[i][j]) return;
    vis[i][j] = true;
    for (int d = 0; d < 4; ++d) llenar(i + dx[d], j + dy[d]);
}

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    cin >> n >> m;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j) cin >> g[i][j];

    int habitaciones = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            if (g[i][j] == '.' && !vis[i][j]) { llenar(i, j); habitaciones++; }

    cout << habitaciones << endl;
}
```

Ojo con la recursión

Una habitación puede tener 10^6 celdas, y eso son 10^6 llamadas apiladas. En CSES pasa; en otros jueces te revienta el stack. La versión con `queue` no tiene ese riesgo:

```
void llenar_bfs(int si, int sj) {
    queue<pair<int,int>> q;
    vis[si][sj] = true;
    q.push({si, sj});

    while (!q.empty()) {
        auto [i, j] = q.front(); q.pop();
        for (int d = 0; d < 4; ++d) {
            int ni = i + dx[d], nj = j + dy[d];
            if (!dentro(ni, nj) || g[ni][nj] == '#' || vis[ni][nj]) continue;
            vis[ni][nj] = true;       // marcar AL ENCOLAR, igual que en Grafos I
            q.push({ni, nj});
        }
    }
}
```

BFS en grilla: el mapa de distancias

0	1	2	3	4	5	6	7	8
1					6		8	9
2		8	9	8	7		9	10
3		7			8	9	10	11
4	5	6	7	8	9	10	11	12

cada número es la cantidad mínima de pasos desde la celda marcada

BFS en grilla — implementación

```cpp
int dist[MAXN][MAXN];

void bfs(int si, int sj) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j) dist[i][j] = -1;   // -1 = no alcanzada

    dist[si][sj] = 0;
    queue<pair<int,int>> q;
    q.push({si, sj});

    while (!q.empty()) {
        auto [i, j] = q.front(); q.pop();
        for (int d = 0; d < 4; ++d) {
            int ni = i + dx[d], nj = j + dy[d];
            if (!dentro(ni, nj) || g[ni][nj] == '#') continue;
            if (dist[ni][nj] != -1) continue;          // ya tiene distancia
            dist[ni][nj] = dist[i][j] + 1;
            q.push({ni, nj});
        }
    }
}
```

Es el mismo BFS de Grafos I. El arreglo `dist` hace de `visitado` y de respuesta a la vez.

Para practicar: grillas

Labyrinth — CSES 1193

Salir de un laberinto desde `A` hasta `B` en la menor cantidad de pasos, e **imprimir el camino**.

Es este mismo BFS, pero además hay que recordar de dónde llegaste a cada celda. La técnica para reconstruir el camino la vemos en el bloque 2.

Monsters — CSES 1194

Escapar del laberinto sin que te pillen los monstruos, que se mueven al mismo tiempo que tú.

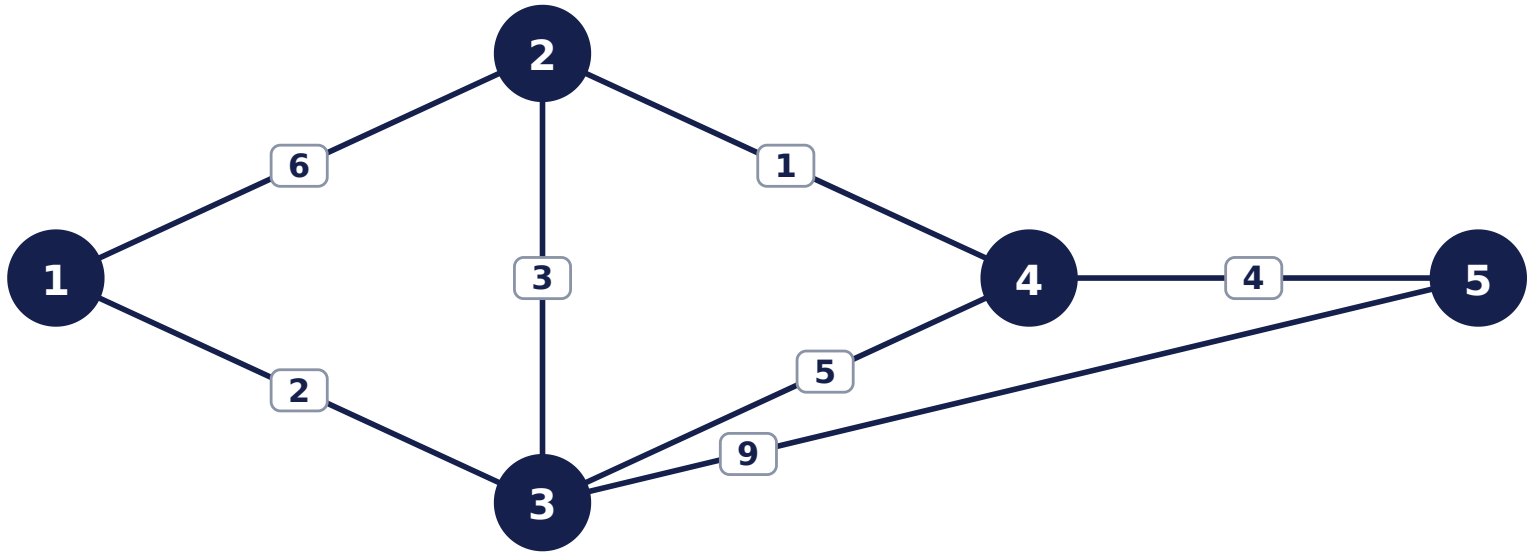
Pista: primero un BFS que parte desde **todos** los monstruos a la vez (mete todos en la cola antes de empezar), y después el tuyo.

Bloque 2 · Cuando las aristas cuestan

1 · Grillas | **2 · Dijkstra** | 3 · DSU y MST

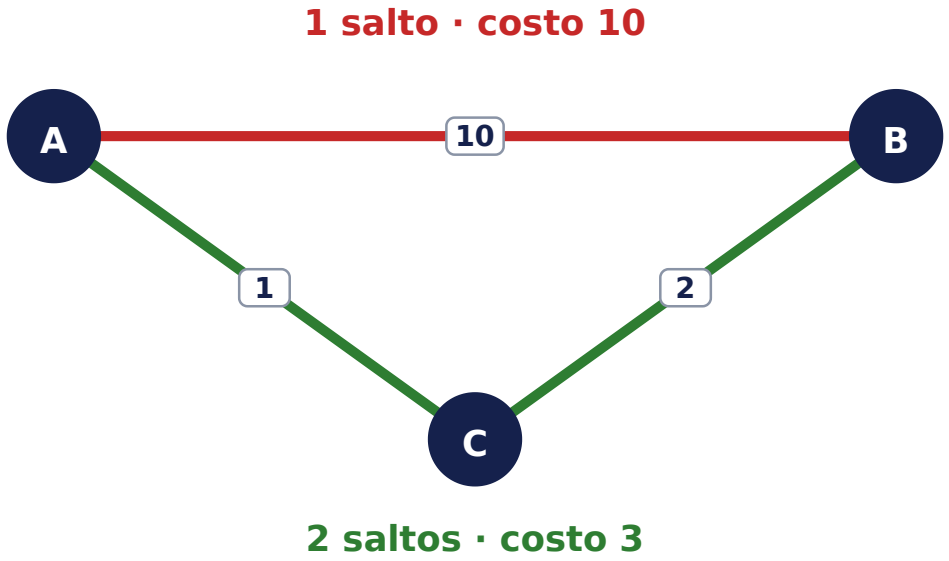
Aristas que cuestan distinto

Hasta ahora toda arista valía lo mismo: **un paso**. Pero un camino tiene kilómetros, un vuelo tiene precio, una tubería tiene capacidad.



Un grafo **ponderado** le pone un número w a cada arista: su **peso** o **costo**.

¿Y el BFS no sirve?



El BFS elegiría el camino **rojo**, porque cuenta **saltos**. Pero el barato es el **verde**.

> El BFS minimiza la cantidad de aristas. Cuando las aristas cuestan distinto, eso deja de ser la respuesta.

Lista de adyacencia ponderada

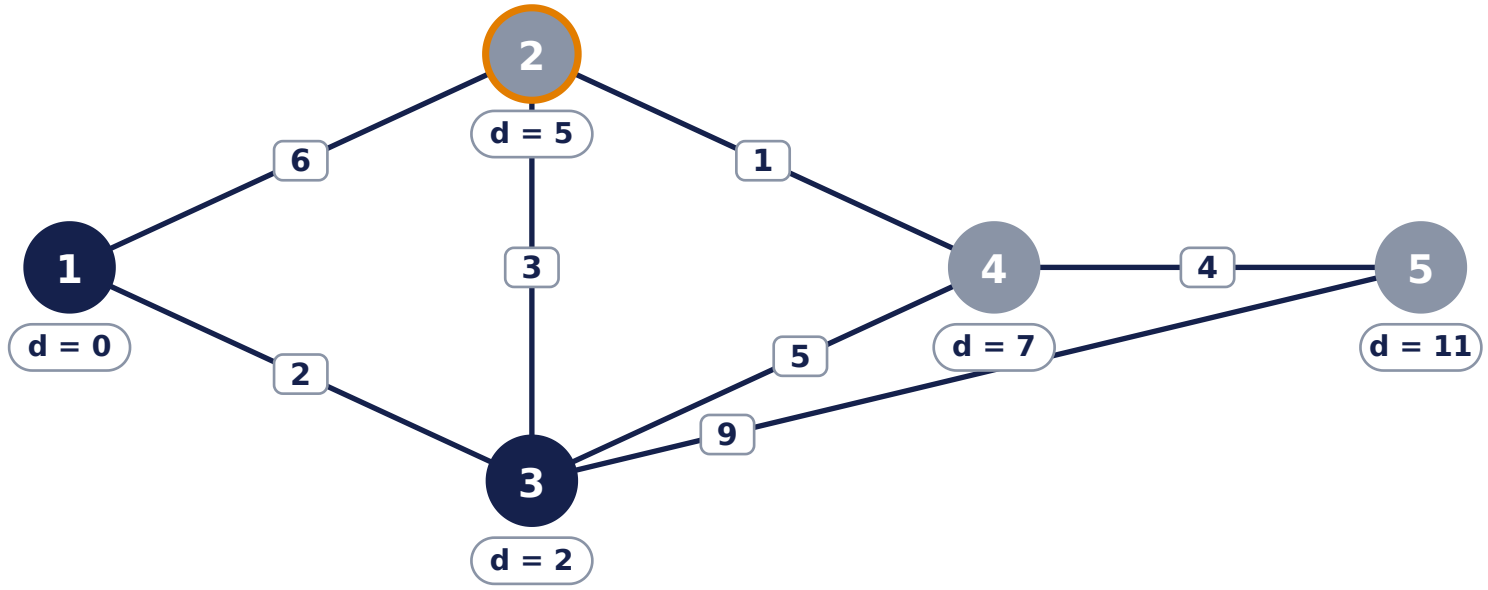
Un vecino ya no es un `int`: es un par **(a quién llego, cuánto me cuesta)**.

```cpp
const int MAXN = 100005;

int n, m;
vector<pair<int,int>> ady[MAXN];      // ady[u] = { {vecino, peso}, ... }

int main() {
    cin >> n >> m;
    for (int e = 0; e < m; ++e) {
        int u, v, w;
        cin >> u >> v >> w;
        ady[u].push_back({v, w});
        ady[v].push_back({u, w});     // borra esta linea si es un digrafo
    }
}
```

También existe la matriz ponderada: `mtz[u][v] = w`, y `INF` donde no hay arista. Sirve con la misma regla de bolsillo de Grafos I: solo cuando $V \lesssim 500$.

Dijkstra: la idea

● distancia ya **definitiva** · ● distancia **tentativa**, puede bajar · **anillo naranjo** = el próximo que cierro



Es **greedy**: de todos los pendientes, cierro siempre el que tiene la distancia tentativa más chica.

Dijkstra — paso a paso

paso		d[1]	d[2]	d[3]	d[4]	d[5]	cola (dist, nodo)
t=1	saco (0,1)	0	**6**	**2**	∞	∞	(2,3) (6,2)
t=2	saco (2,3)	0	**5**	2	**7**	**11**	(5,2) (6,2) (7,4) (11,5)
t=3	saco (5,2)	0	5	2	**6**	11	(6,2) (6,4) (7,4) (11,5)
t=4	**saco (6,2)** ✗ obsoleta	0	5	2	6	11	(6,4) (7,4) (11,5)
t=5	saco (6,4)	0	5	2	6	**10**	(7,4) (10,5) (11,5)
t=6	**saco (7,4)** ✗ obsoleta	0	5	2	6	10	(10,5) (11,5)
t=7	saco (10,5)	0	5	2	6	10	(11,5)
t=8	**saco (11,5)** ✗ obsoleta	0	5	2	6	10	vacía
final		**0**	**5**	**2**	**6**	**10**	

¿Por qué funciona?

Cuando saco de la cola el vértice con la distancia tentativa **más chica**, esa distancia **ya no va a bajar más**.

> Cualquier otro camino hacia él tendría que pasar primero por algún vértice que quedó **más lejos**, y desde ahí solo se suman pesos. Nunca va a dar menos.

Toda la garantía se apoya en una frase: **los pesos no son negativos**. Ahí está el truco, y ahí está el límite del algoritmo.

Al final del bloque vamos a ver exactamente cómo se cae este razonamiento cuando aparece un peso negativo.

Dijkstra — implementación

```cpp
const long long INF = 1e18;
long long dist[MAXN];

void dijkstra(int inicio) {
    for (int u = 1; u <= n; ++u) dist[u] = INF;
    dist[inicio] = 0;

    // min-heap: arriba queda el par con la distancia mas chica
    priority_queue<pair<long long,int>,
                   vector<pair<long long,int>>,
                   greater<pair<long long,int>>> cola;
    cola.push({0, inicio});

    while (!cola.empty()) {
        auto [d, u] = cola.top(); cola.pop();
        if (d > dist[u]) continue;              // entrada obsoleta, la ignoro

        for (auto [v, w] : ady[u])
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                cola.push({dist[v], v});
            }
    }
}
```

El mismo vértice, varias veces en la cola

Cada vez que la distancia de un vértice **baja**, lo empujo de nuevo. Las entradas viejas quedan adentro: no hay forma barata de borrarlas.

(5, 2)	✓ **la proceso**	dist[2] = 5, coinciden
(6, 2)	✗ **continue**	dist[2] = 5, quedó vieja
(6, 4)	✓ **la proceso**	dist[4] = 6, coinciden
(7, 4)	✗ **continue**	dist[4] = 6, quedó vieja

`if (d > dist[u]) continue;` es todo el arreglo: si la distancia que traía el par ya no es la del vértice, la entrada está vencida y se descarta.

Complejidad de Dijkstra

Representación	Complejidad
lista de adyacencia + `priority_queue`	$O((V + E) \log V)$
matriz, buscando el mínimo a mano en cada paso	$O(V^2)$

Con $V = 10^5$ y $E = 2 \cdot 10^5$: unas $3 \cdot 10^5 \cdot 17 \approx 5 \cdot 10^6$ operaciones. Cómodo bajo las 10^8 del segundo.

La versión $O(V^2)$ solo conviene cuando el grafo es denso y chico, con la misma regla de bolsillo de siempre: $V \lesssim 500$.

Ejercicio en vivo 2 — Shortest Routes I

Hay n ciudades y m vuelos. Cada vuelo va **en un solo sentido**, desde a hasta b, y tiene un largo c.

¿Cuál es el largo del vuelo más corto desde la ciudad 1 hasta cada una de las n ciudades?

```
Entrada          Salida
3 4              0 5 2
1 2 6
1 3 2
3 2 3
1 3 4
```

CSES 1671 — Shortest Routes I. Restricciones: $n \leq 10^5$, $m \leq 2 \cdot 10^5$, $1 \leq c \leq 10^9$.

Pasos 1 y 2: modelar y la idea

- ciudad → vértice · vuelo → **arco dirigido** con peso · largo del vuelo → peso
- "el más corto desde la ciudad 1 a todas" → Dijkstra desde el vértice 1, tal cual

La idea es directa. El problema real está en otra parte:

> $2 \cdot 10^5$ vuelos de largo 10^9 cada uno dan hasta $2 \cdot 10^{14}$. Un `int` llega hasta $2 \cdot 10^9$. **Se desborda.**

Todo lo que toque distancias va en `long long`: el arreglo `dist`, el `INF`, y los pares que van en la cola.

Ojo también con la entrada repetida: en el ejemplo hay dos vuelos de 1 a 3 (uno de largo 2 y otro de largo 4). La lista de adyacencia se los queda a los dos y Dijkstra elige solo.

Paso 3: leer el digrafo ponderado

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

const int MAXN = 100005;
const long long INF = 1e18;

int n, m;
vector<pair<int,int>> ady[MAXN];
long long dist[MAXN];

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    cin >> n >> m;
    for (int e = 0; e < m; ++e) {
        int a, b, c;
        cin >> a >> b >> c;
        ady[a].push_back({b, c});     // dirigido: SOLO en este sentido
    }
}
```

Paso 4: Dijkstra desde la ciudad 1

```
for (int u = 1; u <= n; ++u) dist[u] = INF;
dist[1] = 0;

priority_queue<pair<long long,int>,
               vector<pair<long long,int>>,
               greater<pair<long long,int>>> cola;
cola.push({0, 1});

while (!cola.empty()) {
    auto [d, u] = cola.top(); cola.pop();
    if (d > dist[u]) continue;

    for (auto [v, w] : ady[u])
        if (d + w < dist[v]) {
            dist[v] = d + w;
            cola.push({dist[v], v});
        }
}
```

Paso 5: imprimir

```
for (int u = 1; u <= n; ++u)
    cout << dist[u] << " \n"[u == n];
```

> `" \n"[u == n]` imprime un espacio entre las distancias y un salto de línea al final. Es un atajo cómodo, pero un `if` normal hace exactamente lo mismo.

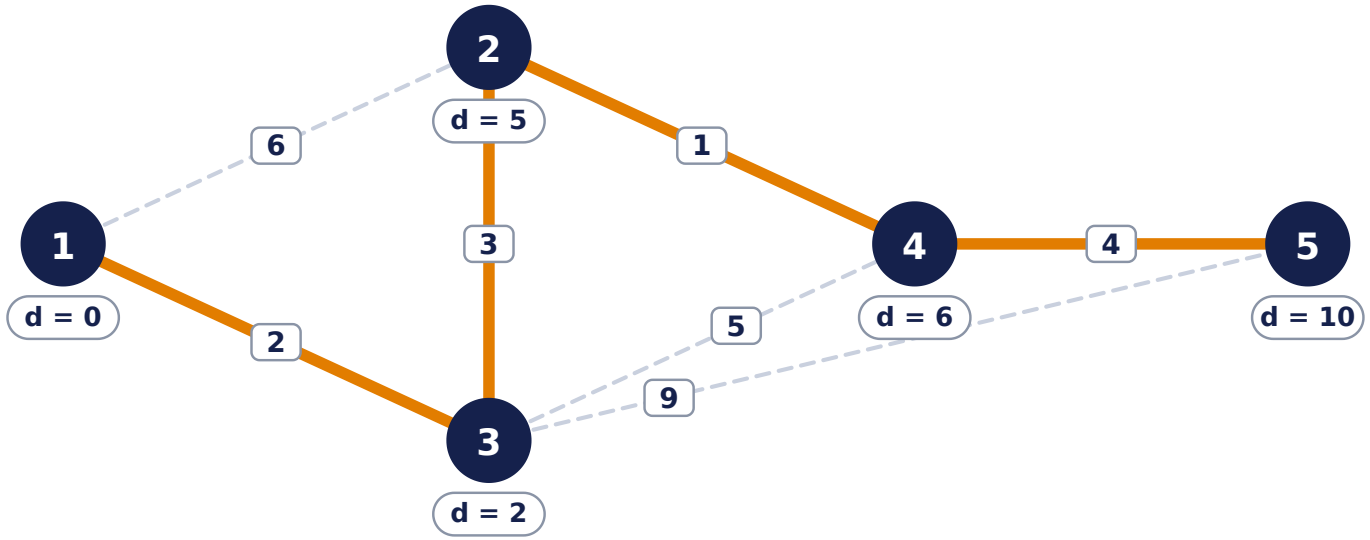
Con la ciudad 1 siempre alcanzable, `dist[1]` vale 0 y ninguna distancia queda en `INF`.

Si un problema sí permite ciudades inalcanzables, hay que decidir qué imprimir cuando `dist[u] == INF`. Nunca imprimas el `INF` crudo.

Solución completa

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

const int MAXN = 100005;
const long long INF = 1e18;

int n, m;
vector<pair<int,int>> ady[MAXN];
long long dist[MAXN];

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    cin >> n >> m;
    for (int e = 0; e < m; ++e) {
        int a, b, c; cin >> a >> b >> c;
        ady[a].push_back({b, c});
    }

    for (int u = 1; u <= n; ++u) dist[u] = INF;
    dist[1] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> cola;
    cola.push({0, 1});

    while (!cola.empty()) {
        auto [d, u] = cola.top(); cola.pop();
        if (d > dist[u]) continue;
        for (auto [v, w] : ady[u])
            if (d + w < dist[v]) { dist[v] = d + w; cola.push({dist[v], v}); }
    }

    for (int u = 1; u <= n; ++u) cout << dist[u] << " \n"[u == n];
}
```

¿Y si me piden el camino, no solo el largo?



padre	[1]	[2]	[3]	[4]	[5]
	—	3	1	2	4

Reconstruir el camino con `padre[]`

Una línea más dentro de la relajación: cada vez que mejoro `dist[v]`, anoto **por quién** llegué.

```
int padre[MAXN];                        // padre[v] = desde donde llegue a v

if (d + w < dist[v]) {
    dist[v] = d + w;
    padre[v] = u;                       // <-- la unica linea nueva
    cola.push({dist[v], v});
}
```

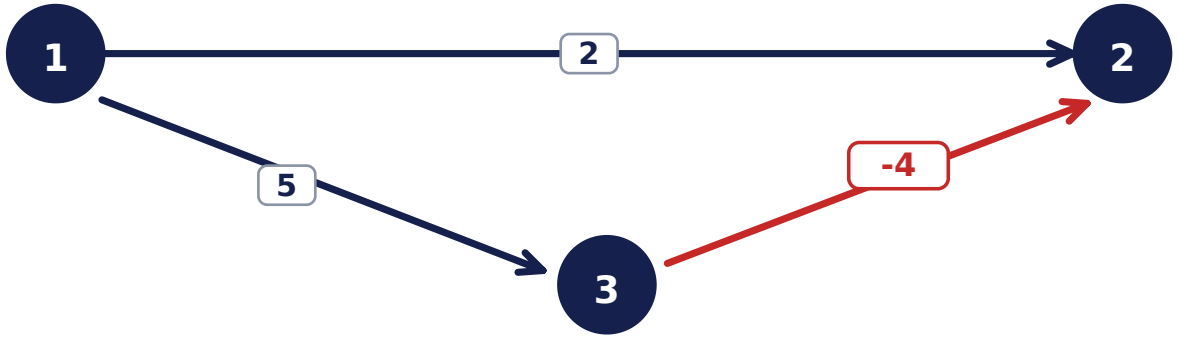
Después el camino se lee **al revés**, desde el destino hacia atrás:

```
vector<int> camino;
for (int v = destino; v != 0; v = padre[v]) camino.push_back(v);
reverse(camino.begin(), camino.end());     // ahora va del inicio al destino
```

Funciona porque `padre` arranca en `0` y el vértice inicial nunca lo cambia: ese `0` es la señal de parada. El mismo truco sirve tal cual en el BFS de una grilla.

Dijkstra se rompe con pesos negativos

Dijkstra cierra el 2 con d = 2 y no lo vuelve a mirar



pero 1 → 3 → 2 cuesta 5 - 4 = 1

Se cae justo la frase del "¿por qué funciona?": con un peso negativo, alejarse **sí** puede terminar saliendo más barato.

Existen Bellman-Ford y Floyd-Warshall

Algoritmo	¿Pesos negativos?	Complejidad	Para qué
Dijkstra	no	$O((V+E)\log V)$	un origen → todos
Bellman-Ford	sí, y detecta ciclos negativos	$O(V \cdot E)$	un origen → todos
Floyd-Warshall	sí, sin ciclos negativos	$O(V^3)$	todos → todos, $V \lesssim 500$

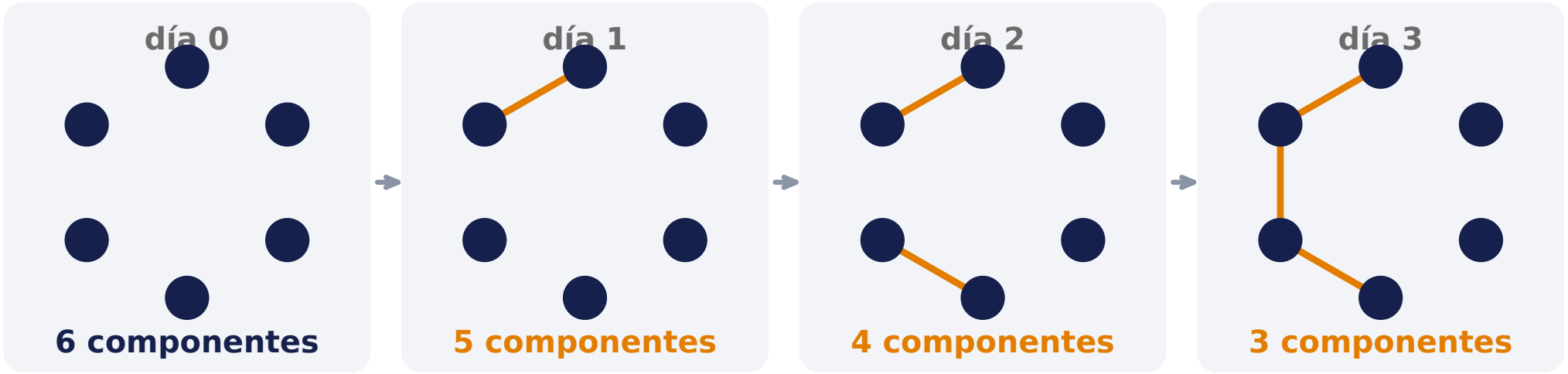
> Hoy no los vemos. Quedan acá para que sepas qué buscar el día que te aparezca un peso negativo.

Para practicar: CSES 1667 *Message Route* (BFS + `padre[]`, la versión sin pesos de lo que acabas de ver) y CSES 1195 *Flight Discount* (Dijkstra sobre el grafo duplicado: "ya usé el cupón" / "todavía no").

Bloque 3 · Conjuntos y árboles mínimos

1 · Grillas | 2 · Dijkstra | **3 · DSU y MST**

Aristas que van llegando de a poco



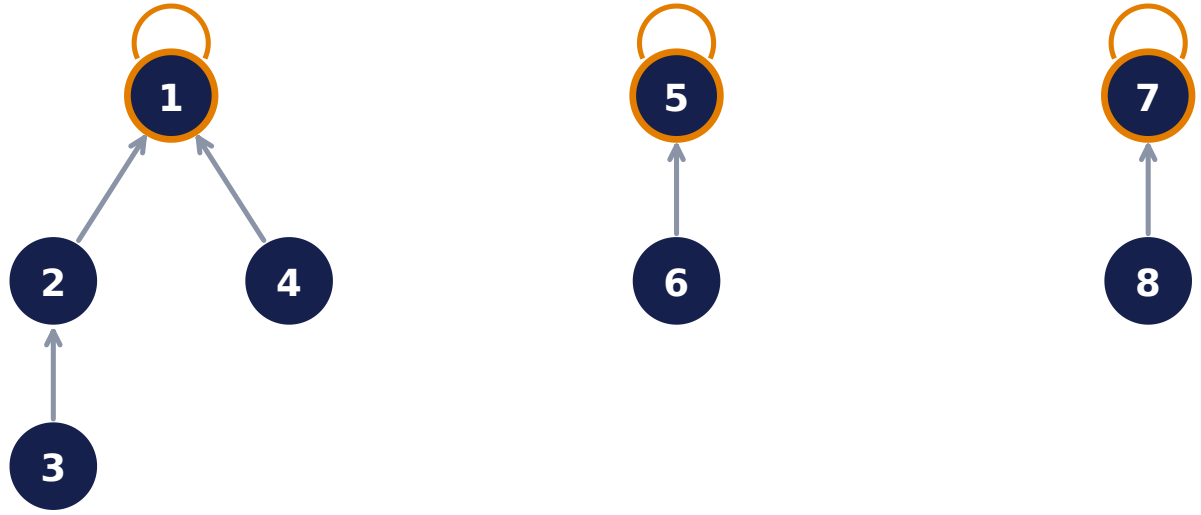
Rehacer el DFS después de cada arista cuesta $O(m \cdot (V + E))$. Con $m = 2 \cdot 10^5$ eso son más de 10^{10} operaciones: **cien veces por sobre el presupuesto**.

> Necesitamos una estructura que responda "¿estos dos están en la misma componente?" y "júntalas" sin volver a recorrer el grafo entero.

DSU: la idea del representante

Cada conjunto elige un **representante**: uno de sus elementos, siempre el mismo.

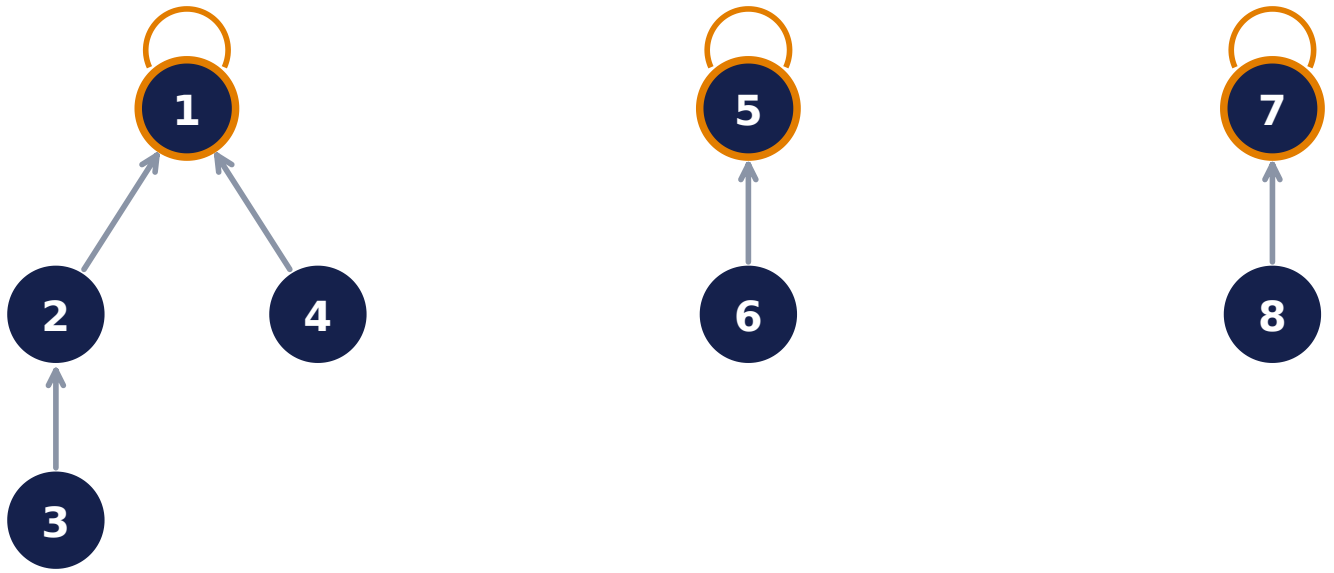
Preguntar "¿están juntos?" pasa a ser "¿tienen el mismo representante?".



Como mínimo la estructura tiene que soportar dos cosas: **unir** dos conjuntos distintos y **encontrar** el representante de un elemento.

La representación: un solo arreglo

`padre[x]` guarda **de quién cuelga** `x`. El representante es el único que cuelga de sí mismo.



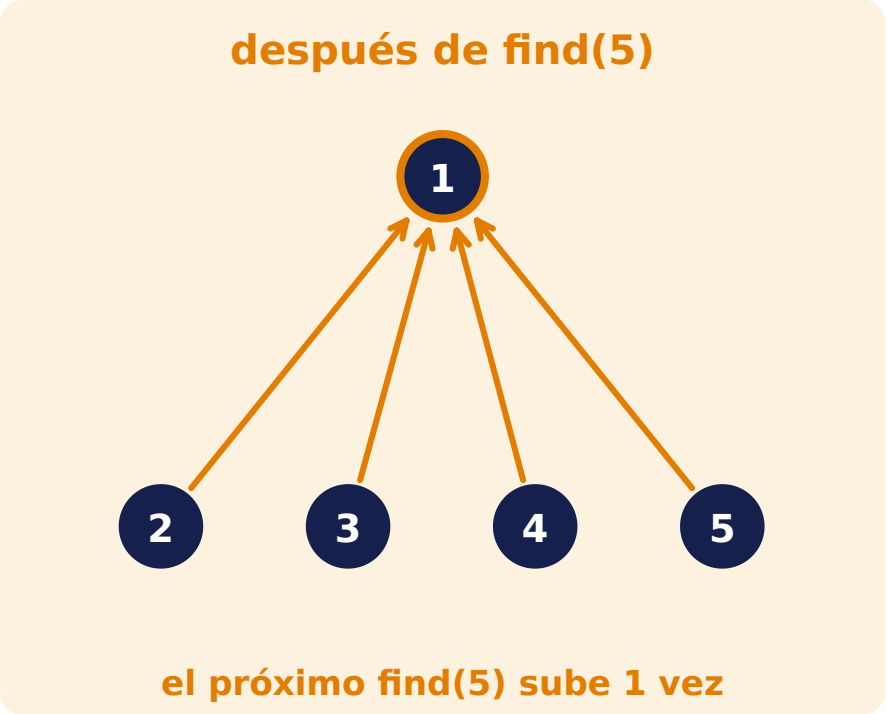
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
padre	1	1	2	1	5	5	7	7

`find` y `une`, versión ingenua

```
int padre[MAXN];

void init(int n) {
    for (int u = 1; u <= n; ++u) padre[u] = u;   // cada uno solo, en su conjunto
}

int find(int x) {
    while (padre[x] != x) x = padre[x];          // subo hasta la raiz
    return x;
}

void une(int a, int b) {
    a = find(a); b = find(b);
    if (a != b) padre[b] = a;                    // cuelgo un representante del otro
}
```

Funciona, pero `find` **puede costar** $O(n)$: nada impide que el árbol se estire hasta quedar como una lista.

El problema de la cadena, y su arreglo

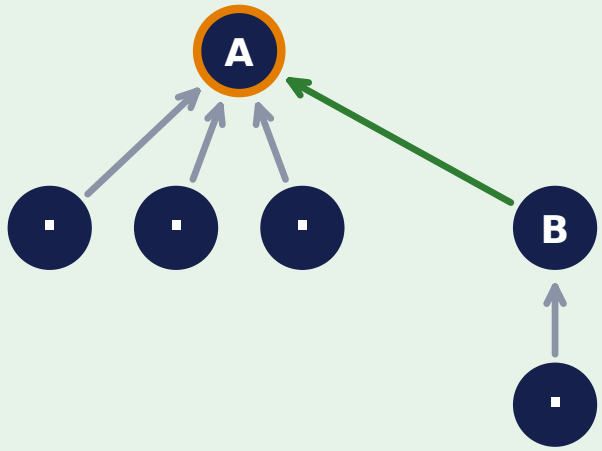


Compresión de caminos: ya que subí hasta la raíz, aprovecho de colgar de ella a todos los que pasé por el camino.

Unión por tamaño

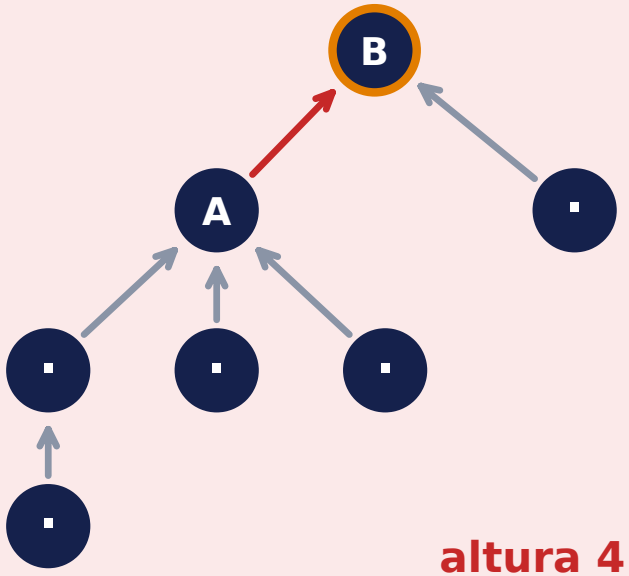
Al unir dos árboles, **el chico cuelga del grande**. Así ninguna rama crece más de lo necesario.

✓ el chico cuelga del grande



altura 3

✗ el grande cuelga del chico



altura 4

Basta con llevar un arreglo `tam[]` con el tamaño de cada conjunto y actualizarlo en cada unión.

DSU completo

```cpp
int padre[MAXN], tam[MAXN];

void init(int n) {
    for (int u = 1; u <= n; ++u) { padre[u] = u; tam[u] = 1; }
}

int find(int x) {
    if (padre[x] == x) return x;
    int raiz = find(padre[x]);     // busco la raiz
    padre[x] = raiz;               // y me cuelgo directo de ella
    return raiz;
    // las tres lineas de arriba se escriben tambien asi:
    //   return padre[x] = find(padre[x]);
}

bool une(int a, int b) {           // devuelve si de verdad unio algo
    a = find(a); b = find(b);
    if (a == b) return false;      // ya estaban en el mismo conjunto
    if (tam[a] < tam[b]) swap(a, b);
    padre[b] = a;
    tam[a] += tam[b];
    return true;
}
```

Complejidad del DSU

Operación	Con compresión + unión por tamaño
`find(x)`	$O(\alpha(n))$
`une(a, b)`	$O(\alpha(n))$

$\alpha(n)$ es la inversa de la función de Ackermann. Vale **a lo más 4** para cualquier n que quepa en un computador: en la práctica, cuéntalo como $O(1)$.

Sin ninguna de las dos optimizaciones cada operación puede costar $O(n)$; con una sola de ellas, $O(\log n)$.

Ejercicio en vivo 3 — Road Construction

Hay n ciudades y **ningún** camino. Cada día se construye un camino nuevo, y son m días en total.

Después de cada día, imprime cuántas componentes hay y el tamaño de la más grande.

Entrada	Salida
5 3 1 2 1 3 4 5	4 2 3 3 2 3

CSES 1676 — Road Construction. Restricciones: $n \leq 10^5$, $m \leq 2 \cdot 10^5$.

Paso 1: modelar

- ciudad → elemento del DSU · camino nuevo → `une(a, b)`
- componente conexa → conjunto · ciudades de una componente → `tam[]` del conjunto

> Fíjate en lo que **no** hay: ni lista de adyacencia, ni DFS, ni BFS. El grafo nunca se construye. El DSU es todo lo que se necesita.

La pregunta "¿cuántas componentes hay?" y "¿cuál es la más grande?" se responden **sin recorrer nada**, si las vamos manteniendo al día.

Paso 2: la idea

camino	¿qué pasa?	componentes	mayor
inicio	—	**5**	**1**
1 — 2	estaban separados → uno	**4**	**2**
1 — 3	estaban separados → uno	**3**	**3**
4 — 5	estaban separados → uno	**2**	**3**

Arranco con n conjuntos. Cada `une` que **de verdad** une baja el contador en 1. El mayor solo se compara contra el conjunto recién formado: **nunca decrece**.

Paso 3: el DSU con tamaño

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

const int MAXN = 100005;
int padre[MAXN], tam[MAXN];

int find(int x) {
    if (padre[x] == x) return x;
    return padre[x] = find(padre[x]);
}

int main() {
    int n, m;
    cin >> n >> m;
    for (int u = 1; u <= n; ++u) { padre[u] = u; tam[u] = 1; }
}
```

Paso 4: procesar cada día y responder

```
int comps = n, mayor = 1;

for (int e = 0; e < m; ++e) {
    int a, b;
    cin >> a >> b;
    int ra = find(a), rb = find(b);

    if (ra != rb) {                          // el camino sirve de algo
        if (tam[ra] < tam[rb]) swap(ra, rb);
        padre[rb] = ra;
        tam[ra] += tam[rb];
        comps--;
        mayor = max(mayor, tam[ra]);
    }
    cout << comps << " " << mayor << "\n";
}
```

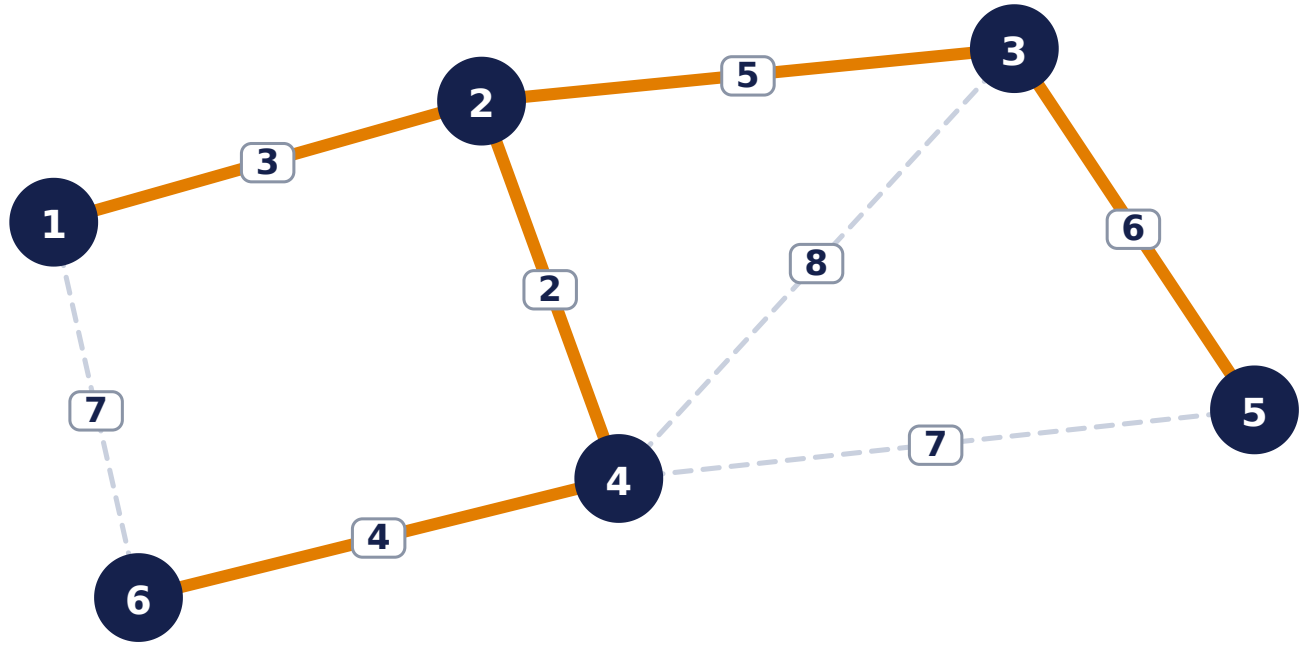
Si `ra == rb` el camino une dos ciudades que ya estaban conectadas: no cambia nada, pero igual hay que imprimir la línea.

Solución completa

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

const int MAXN = 100005;
int padre[MAXN], tam[MAXN];

int find(int x) { return padre[x] == x ? x : padre[x] = find(padre[x]); }

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n, m; cin >> n >> m;
    for (int u = 1; u <= n; ++u) { padre[u] = u; tam[u] = 1; }

    int comps = n, mayor = 1;
    for (int e = 0; e < m; ++e) {
        int a, b; cin >> a >> b;
        int ra = find(a), rb = find(b);
        if (ra != rb) {
            if (tam[ra] < tam[rb]) swap(ra, rb);
            padre[rb] = ra; tam[ra] += tam[rb];
            comps--; mayor = max(mayor, tam[ra]);
        }
        cout << comps << " " << mayor << "\n";
    }
}
```

Árbol de cobertura mínimo

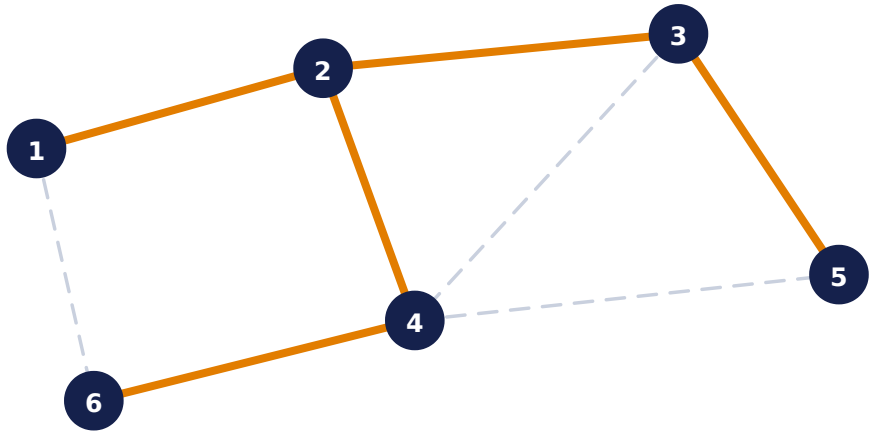
De todas las formas de dejar el grafo **conexo** usando solo sus aristas, la más barata. Con V vértices siempre son exactamente $V - 1$ aristas, y nunca sobra un ciclo: es un **árbol**.



Peso total: $2 + 3 + 4 + 5 + 6 = 20$. Las aristas punteadas quedaron fuera.

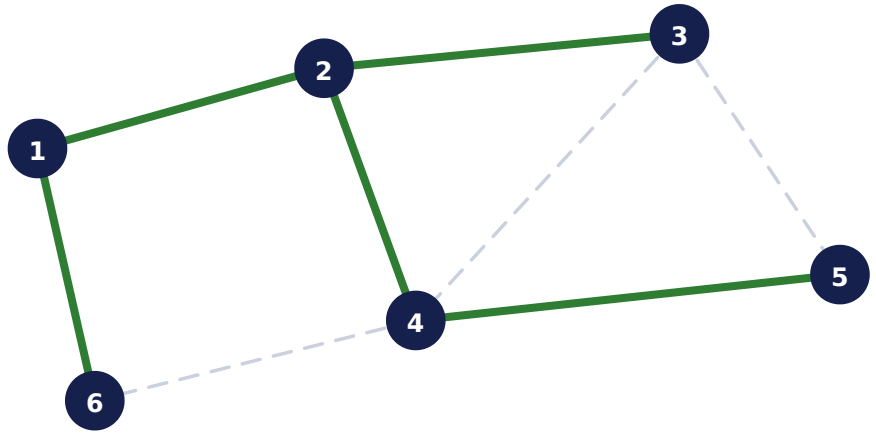
El MST no es el árbol que arma Dijkstra

MST · peso total 20



usa 4—6 (peso 4) y 3—5 (peso 6)

árbol de Dijkstra desde 1 · peso total 24



usa 1—6 (peso 7) y 4—5 (peso 7)

El MST minimiza la **suma total** de las aristas. Dijkstra minimiza la **distancia desde un origen** a cada vértice, uno por uno. Son objetivos distintos y dan árboles distintos.

Kruskal: la idea

1. Ordenar **todas** las aristas de menor a mayor peso
2. Partir con el DSU vacío: cada vértice en su propio conjunto
3. Recorrer las aristas en ese orden. Para cada arista (u, v):
   - si u y v están en **conjuntos distintos**, la tomo y los uno
   - si ya estaban **en el mismo conjunto**, la descarto: cerraría un ciclo
4. Al tomar la arista número $V - 1$, el árbol está listo

> No hace falta construir un grafo nuevo: basta con ir **sumando el peso** de las aristas tomadas y **contándolas**.

Es greedy, igual que Dijkstra, pero con otro criterio: acá la avaricia es por el peso de la arista, no por la distancia al origen.

Kruskal — paso a paso

arista	peso	¿ya estaban juntos?	acumulado	conjuntos
2 — 4	2	**no → la tomo**	**2**	{1} {2,4} {3} {5} {6}
1 — 2	3	**no → la tomo**	**5**	{1,2,4} {3} {5} {6}
4 — 6	4	**no → la tomo**	**9**	{1,2,4,6} {3} {5}
2 — 3	5	**no → la tomo**	**14**	{1,2,3,4,6} {5}
3 — 5	6	**no → la tomo**	**20**	{1,2,3,4,5,6}
1 — 6	7	**sí → la descarto**	20	{1,2,3,4,5,6}
4 — 5	7	**sí → la descarto**	20	{1,2,3,4,5,6}
3 — 4	8	**sí → la descarto**	20	{1,2,3,4,5,6}
MST		**5 aristas = V - 1**	**20**	

La lista de aristas, y Kruskal

Para ordenar por peso conviene una tercera representación: **la lista de aristas**, con el peso **primero**.

```
vector<array<int,3>> aristas(m);          // {peso, u, v}
for (int e = 0; e < m; ++e) {
    int a, b, c;
    cin >> a >> b >> c;
    aristas[e] = {c, a, b};               // el peso va PRIMERO, a proposito
}
sort(aristas.begin(), aristas.end());     // ordena por peso, sin comparador
```

`array<int,3>` se compara de izquierda a derecha. Con el peso en la posición 0, el `sort` por defecto ya hace exactamente lo que queremos.

```
long long costo = 0; int tomadas = 0;
for (auto [w, a, b] : aristas)
    if (une(a, b)) { costo += w; tomadas++; }   // une() avisa si sirvio
```

Complejidad de Kruskal

Paso	Costo
ordenar las aristas	$O(E \log E)$
recorrerlas haciendo `une`	$O(E \cdot \alpha(V)) \approx O(E)$
total	$O(E \log E)$

El `sort` domina. El DSU es prácticamente gratis: toda la gracia de $\alpha(n) \leq 4$ es que desaparece al lado del ordenamiento.

Con $E = 2 \cdot 10^5$: unas $2 \cdot 10^5 \cdot 18 \approx 4 \cdot 10^6$ operaciones.

Ejercicio en vivo 4 — Road Reparation

Hay n ciudades y m caminos, todos inservibles. Reparar el camino entre a y b cuesta c.

¿Cuál es el costo mínimo para que se pueda viajar entre cualquier par de ciudades? Si no se puede, imprime `IMPOSSIBLE`.

```
Entrada          Salida
5 6              14
1 2 3
2 3 5
2 4 2
3 4 8
5 1 7
5 4 4
```

CSES 1675 — Road Reparation. Restricciones: $n \leq 10^5$, $m \leq 2 \cdot 10^5$, $1 \leq c \leq 10^9$.

Pasos 1 y 2: modelar y la idea

- ciudad → vértice · camino → arista no dirigida · costo de reparación → peso
- "que se pueda viajar entre cualquier par, al mínimo costo" → **es el MST, textual**

> `IMPOSSIBLE` ⟺ **terminas con menos de $n-1$ aristas ⟺ el grafo original era disconexo.** No hay forma de conectar lo que nunca estuvo conectado.

Y otra vez el mismo cuidado del bloque 2: 10^5 aristas de costo 10^9 dan hasta 10^{14}. El acumulador va en `long long`.

Los pesos individuales sí caben en `int`; el que se desborda es el total.

Pasos 3 y 4: leer, ordenar, unir

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

vector<array<int,3>> aristas(m);
for (int e = 0; e < m; ++e) {
    int a, b, c;
    cin >> a >> b >> c;
    aristas[e] = {c, a, b};
}
sort(aristas.begin(), aristas.end());

for (int u = 1; u <= n; ++u) { padre[u] = u; tam[u] = 1; }

long long costo = 0;
int tomadas = 0;
for (auto [w, a, b] : aristas)
    if (une(a, b)) { costo += w; tomadas++; }

if (tomadas == n - 1) cout << costo << endl;
else                  cout << "IMPOSSIBLE" << endl;
```

Solución completa

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

const int MAXN = 100005;
int padre[MAXN], tam[MAXN];

int find(int x) { return padre[x] == x ? x : padre[x] = find(padre[x]); }

bool une(int a, int b) {
    a = find(a); b = find(b);
    if (a == b) return false;
    if (tam[a] < tam[b]) swap(a, b);
    padre[b] = a; tam[a] += tam[b];
    return true;
}

int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n, m; cin >> n >> m;
    vector<array<int,3>> aristas(m);
    for (int e = 0; e < m; ++e) {
        int a, b, c; cin >> a >> b >> c;
        aristas[e] = {c, a, b};
    }
    sort(aristas.begin(), aristas.end());
    for (int u = 1; u <= n; ++u) { padre[u] = u; tam[u] = 1; }

    long long costo = 0; int tomadas = 0;
    for (auto [w, a, b] : aristas)
        if (une(a, b)) { costo += w; tomadas++; }

    if (tomadas == n - 1) cout << costo << endl;
    else                  cout << "IMPOSSIBLE" << endl;
}
```

Cierre

El mapa: ¿qué uso para cada cosa?

Lo que necesito	La herramienta
recorrer el grafo, contar componentes	DFS o BFS
distancia mínima con aristas de peso 1	BFS
lo mismo, pero sobre una grilla	BFS con `dx` / `dy`
distancia mínima con pesos ≥ 0	**Dijkstra**
el camino en sí, no solo su largo	`padre[]` + `reverse`
¿están conectados? aristas que van llegando	**DSU**
conectar todo al mínimo costo total	**Kruskal**
hay pesos negativos	Bellman-Ford *(no visto)*

Tarea

Problema	Tema	Por dónde empezar
Counting Rooms · 1192	grillas	lo resolvimos hoy: escríbelo tú y mándalo
Labyrinth · 1193	grillas	BFS en grilla + `padre[]` para imprimir el camino
Monsters · 1194	grillas	un BFS que parte desde todos los monstruos a la vez
Message Route · 1667	caminos	es Labyrinth sin grilla: BFS + reconstrucción
Flight Discount · 1195	caminos	Dijkstra sobre el grafo duplicado: con cupón y sin
Round Trip · 1669	recorridos	pendiente de Grafos I: detectar e imprimir un ciclo
Road Reparation · 1675	MST	Kruskal directo, ojo con el `long long`

Todos son de CSES: `cses.fi/problemset/task/<número>`.

Para seguir entrenando

- **CSES Problem Set**, sección *Graph Algorithms*: está ordenada casi exactamente como estas dos clases
- **Codeforces**, tags `dfs and similar`, `shortest paths`, `dsu`, `graphs`, dificultad 1000–1400
- **AtCoder Beginner Contest**, problemas C y D

> El mejor consejo sigue siendo el de la clase pasada: aprende a escribir DFS, BFS, Dijkstra y el DSU **de memoria**. En un contest no quieres estar pensando en la sintaxis de la `priority_queue`.

Los cuatro caben en menos de 60 líneas juntos. Vale la pena tenerlos en los dedos.

¿Preguntas?