CURSO DE PROGRAMACIÓN COMPETITIVA

GRAFOS

CURSO DE PROGRAMACIÓN COMPETITIVA URJC - 2025

Organizadores:

- Isaac Lozano (isaac.lozano@urjc.es)
- Sergio Salazar (sergio.salazar@urjc.es)
- **Adaya Ruiz** (am.ruiz.2020@alumnos.urjc.es)
- Lucas Martín (lucas.martin@urjc.es)
- **Iván Penedo** (ivan.penedo@urjc.es)
- Alicia Pina (alicia.pina@urjc.es)
- Sara García (sara.garciar@urjc.es)
- Raúl Fauste (raul.fauste@urjc.es)

Contenidos

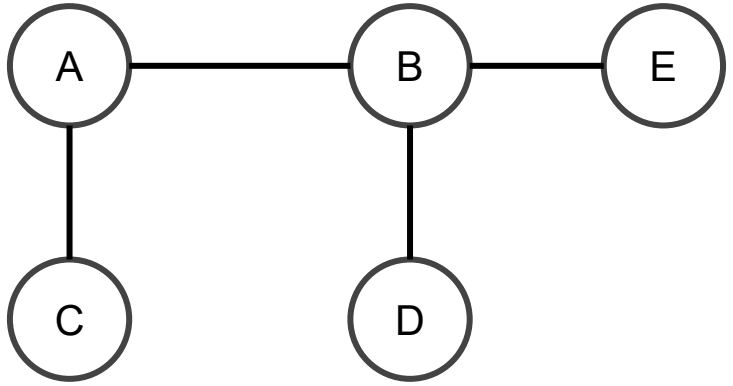
GRAFOS

- Repaso clase ED2
  - Definición, tipos y representaciones
  - Recorridos en profundidad y anchura
- Enfoques algorítmicos
  - Componentes conexas
  - Ordenamiento Topológico
  - Bipartito
- Algoritmos sobre grafos
  - Puntos de articulación
  - Distancia mínima
  - Árboles de recubrimiento
  - Estructura Union-Find

Grafos - Definición

- Definición: G = (V,E)
  - Conjunto de vértices: V={A,B,C,D,E}
  - Conjunto de aristas:

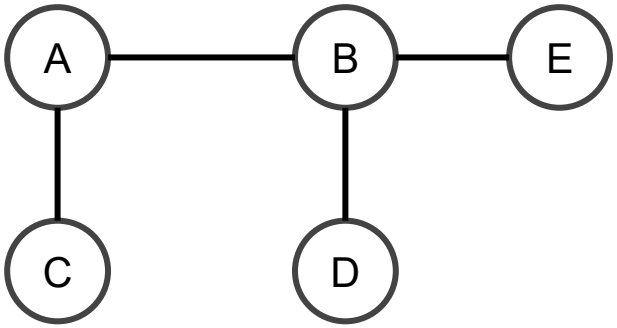
    E={(A,B),(A,C),(B,D),(B,E)}



Grafos - Tipos

- Grafo NO dirigido

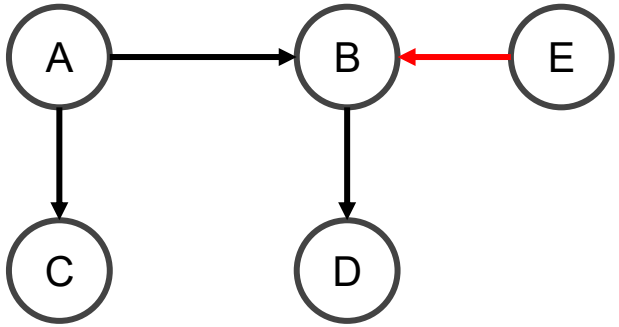
  $(A,B) \Leftrightarrow (B,A)$



E={
(A,B),(B,A),(A,C),(C,A),
(B,D),(D,B),(B,E),(E,B)}

- Grafo dirigido

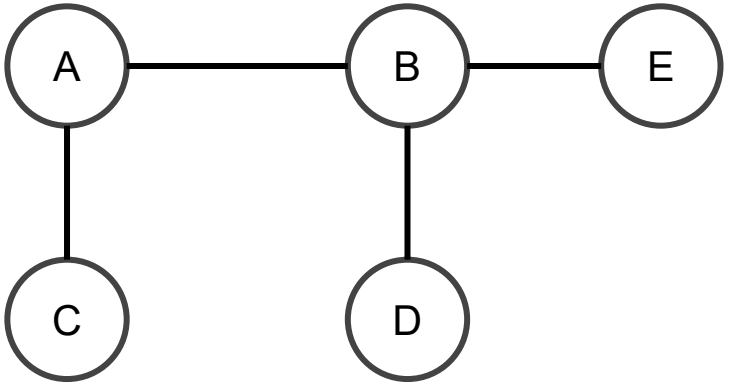
  $(E,B) \neq (B,E)$



E={
(A,B),(A,C),(B,D),(E, B)}

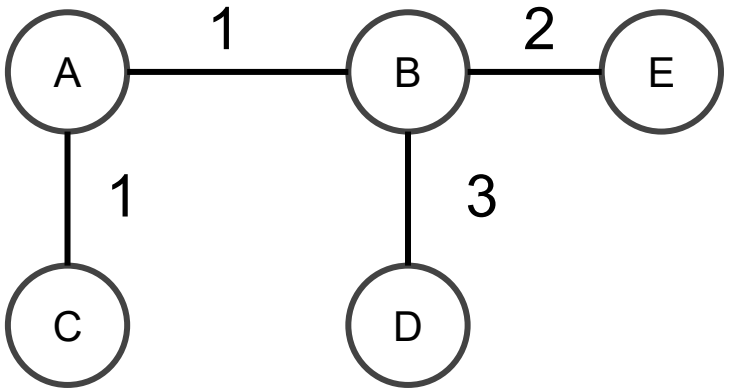
Grafos - Tipos

- Grafo no ponderado



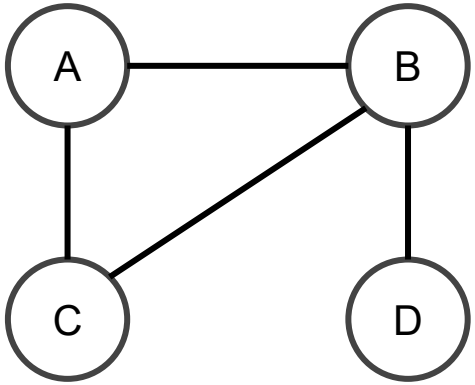
E={(A,B),(A,C),(B,D),(B,E),
(B,A),(C,A),(D,B),(E,B)}

- Grafo ponderado



E={(A,B,1),(A,C,1),(B,D,3),(B,E,2),
(B,A,1),(C,A,1),(D,B,3),(E,B,2)}

Grafos - Representaciones



Matriz de adyacencia

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

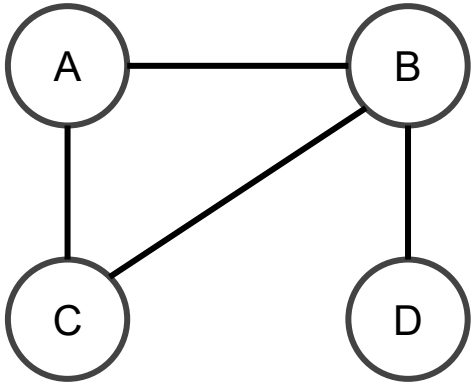
Lista de adyacencia

A	{B,C}
B	{A,C,D}
C	{A,B}
D	{B}

Lista de aristas

{
(A,B),(B,A),
(A,C),(C,A),
(B,C),(C,B),
(B,D),(D,B)
}

Grafos - Representaciones



Matriz de adyacencia

- Memoria: $O(|V|^2)$
- Acceso: O(1)
- Aristas de un vértice: O(|V|)

Caso de uso: grafos densos (N~=5.000)

Lista de adyacencia

- Memoria: O(|V|+|E|)
- Acceso: O(|V|) u O(1)
- Aristas de un vértice: O(1)

Caso de uso: grafos dispersos (N>100.000)

Lista de aristas

- Memoria: O(|E|)
- Acceso: O(|E|)
- Aristas de un vértice: O(|E|)

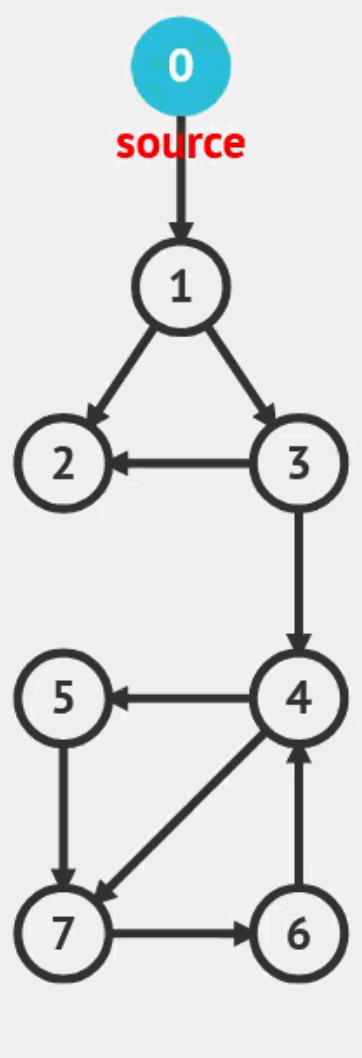
Caso de uso: procesado de aristas

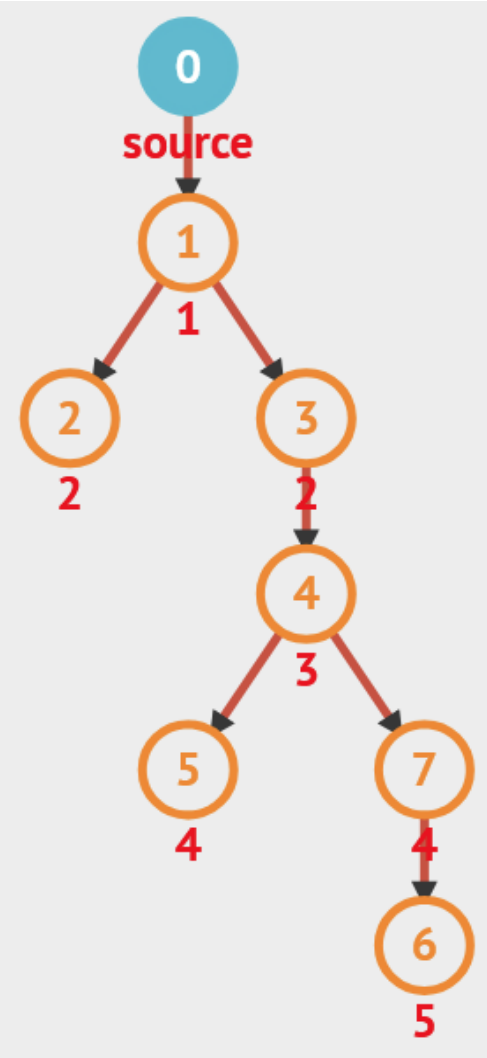
Grafos - Recorridos

Recorrer un grafo

- Recorrido en anchura
  - **BFS** - Breadth First Search
- Recurrido en profundidad
  - **DFS** - Depth First Search

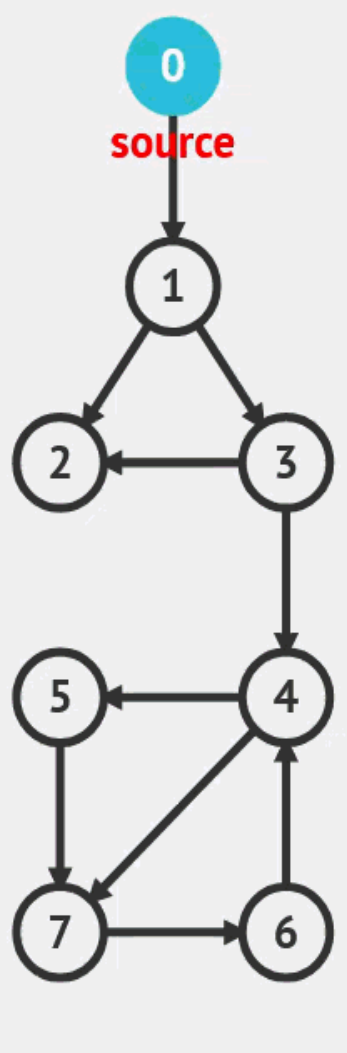
Grafos - BFS

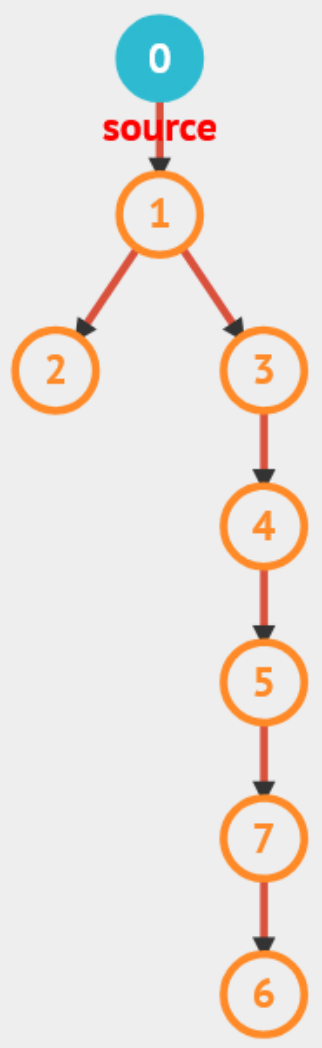





```python
from collections import deque

def bfs(grafo, inicio):
    visitados = set()
    cola = deque([inicio])
    visitados.add(inicio)

    while cola:
        actual = cola.popleft()

        for vecino in grafo[actual]:
            if vecino not in visitados:
                visitados.add(vecino)
                cola.append(vecino)
```

Grafos - DFS






```
1   def dfs(grafo, inicio):
2       visitados = set()
3       pila = [inicio]
4       visitados.add(inicio)
5
6       while pila:
7           actual = pila.pop()
8
9           for vecino in grafo[actual]:
10              if vecino not in visitados:
11                  visitados.add(vecino)
12                  pila.append(vecino)
```

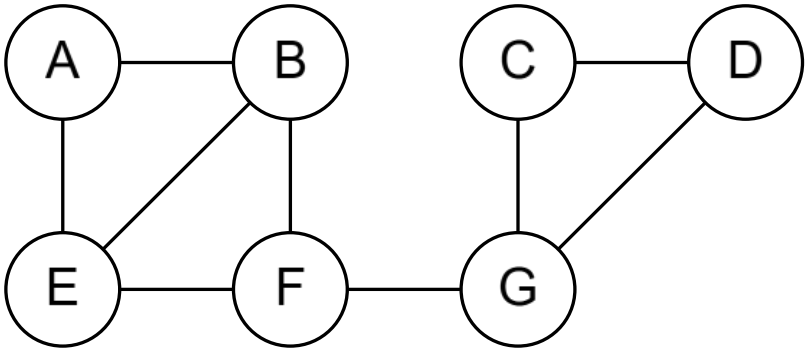
Grafos - Recorridos

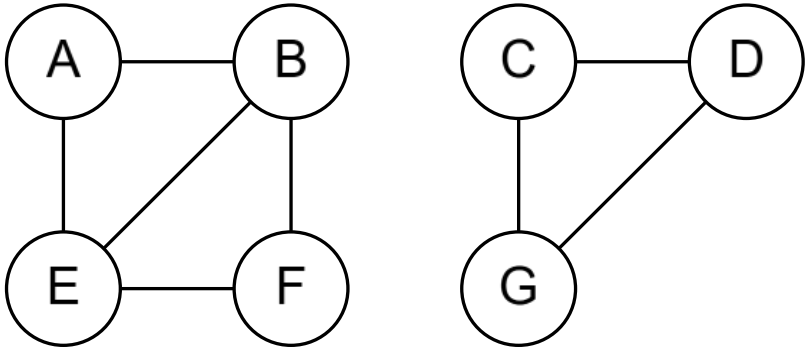
Cosas a tener en cuenta al recorrer:

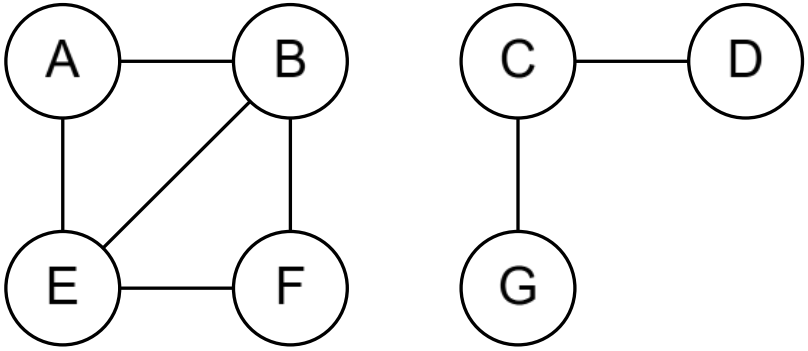
- Tenemos que llevar cuenta de por qué nodos ya hemos pasado → TLE
- Cada arista recorrida cuenta como 1 paso
- El grafo puede tener algún nodo suelto (también hay que recorrerlo)

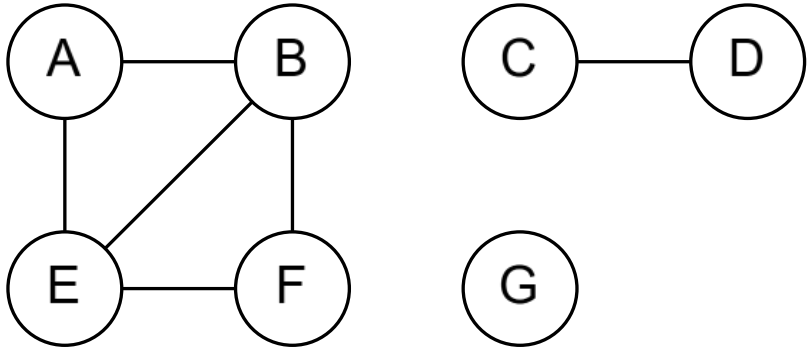
Componentes conexas

¿Qué son?



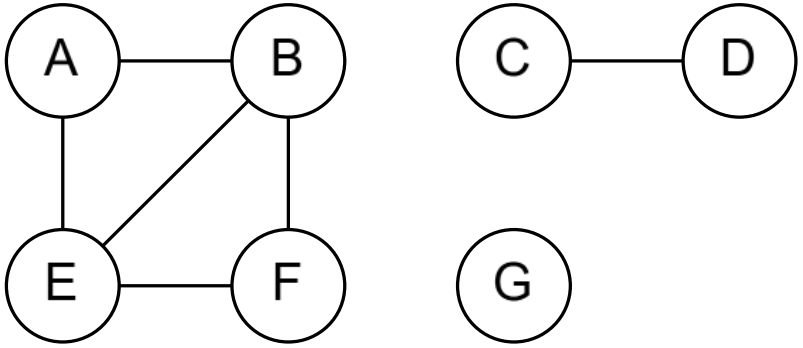






Componentes conexas

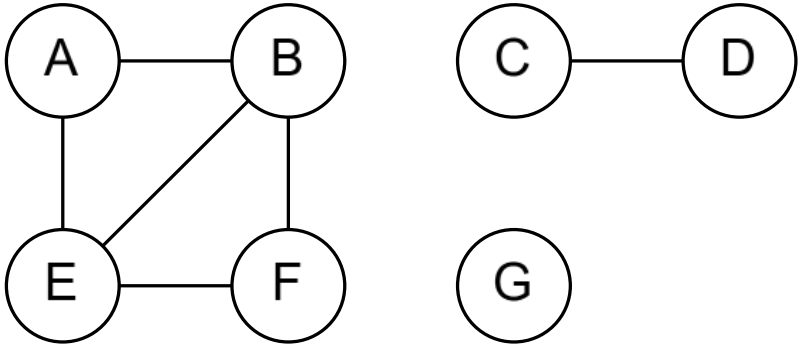
¿Como las descubrimos?




```
1    def dfs(grafo, inicio):
2        pila = [inicio]
3        visitados = set()
4        visitados.add(inicio)
5        while pila:
6            actual = pila.pop()
7            componente.append(actual)
8            for vecino in grafo[actual]:
9                if vecino not in visitados:
10                   visitados.add(vecino)
11                   pila.append(vecino)
```

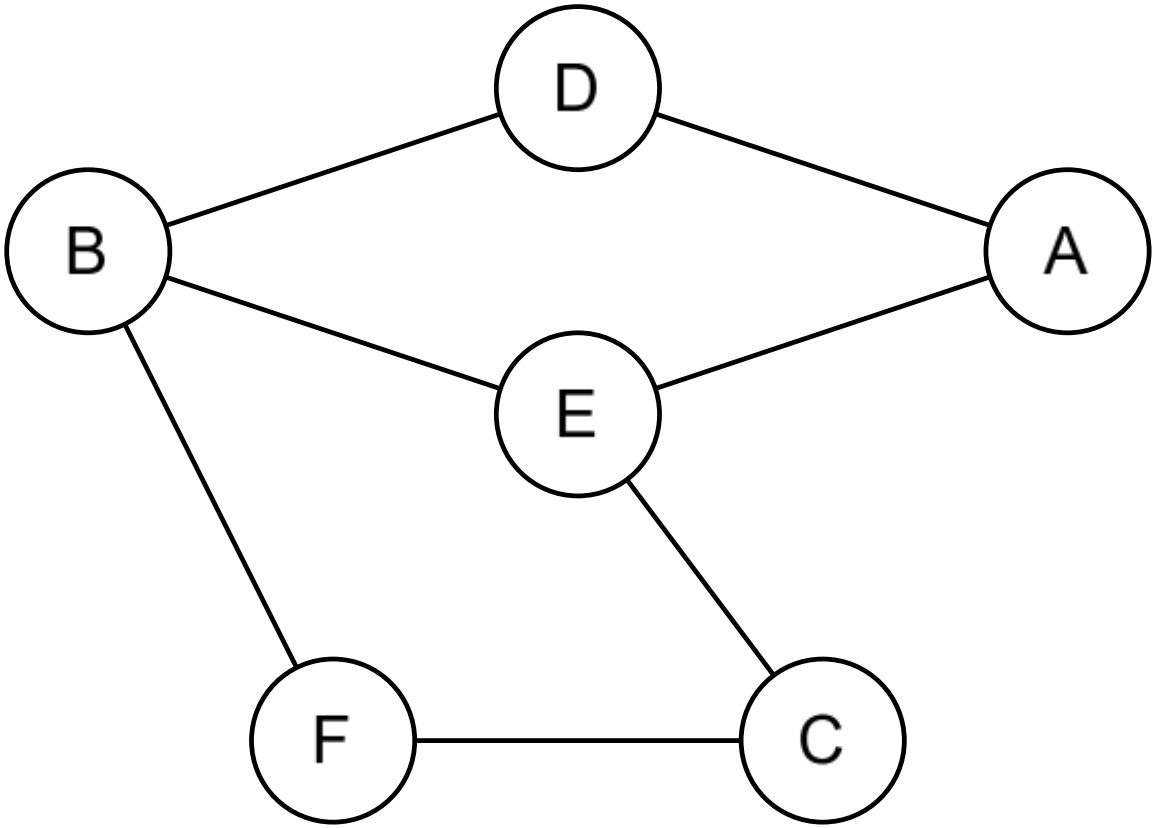
Componentes conexas

¿Como las descubrimos?




```
1   def dfs(grafo, inicio, visitados):
2       pila = [inicio]
3       visitados.add(inicio)
4       while pila:
5           actual = pila.pop()
6           for vecino in grafo[actual]:
7               if vecino not in visitados:
8                   visitados.add(vecino)
9                   pila.append(vecino)
10
11  def contar_componentes(grafo):
12      visitados, n_componentes = set(), 0
13      for nodo in grafo:
14          if nodo not in visitados:
15              dfs(grafo, nodo, visitados)
16              n_componentes += 1
17      return n_componentes
```

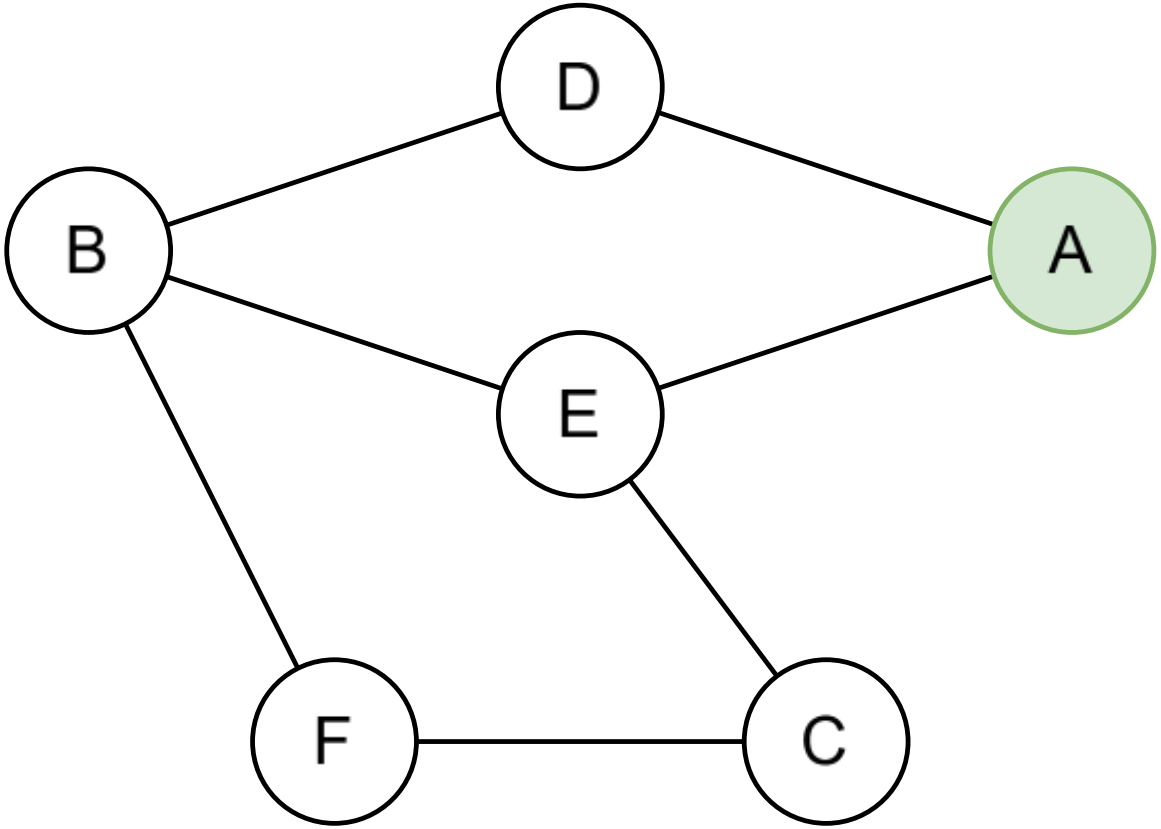
Bipartitos



Buscamos dividir los nodos en dos grupos, sin aristas entre nodos del mismo grupo.

2-Coloring: Coloreado de grafos con dos colores.

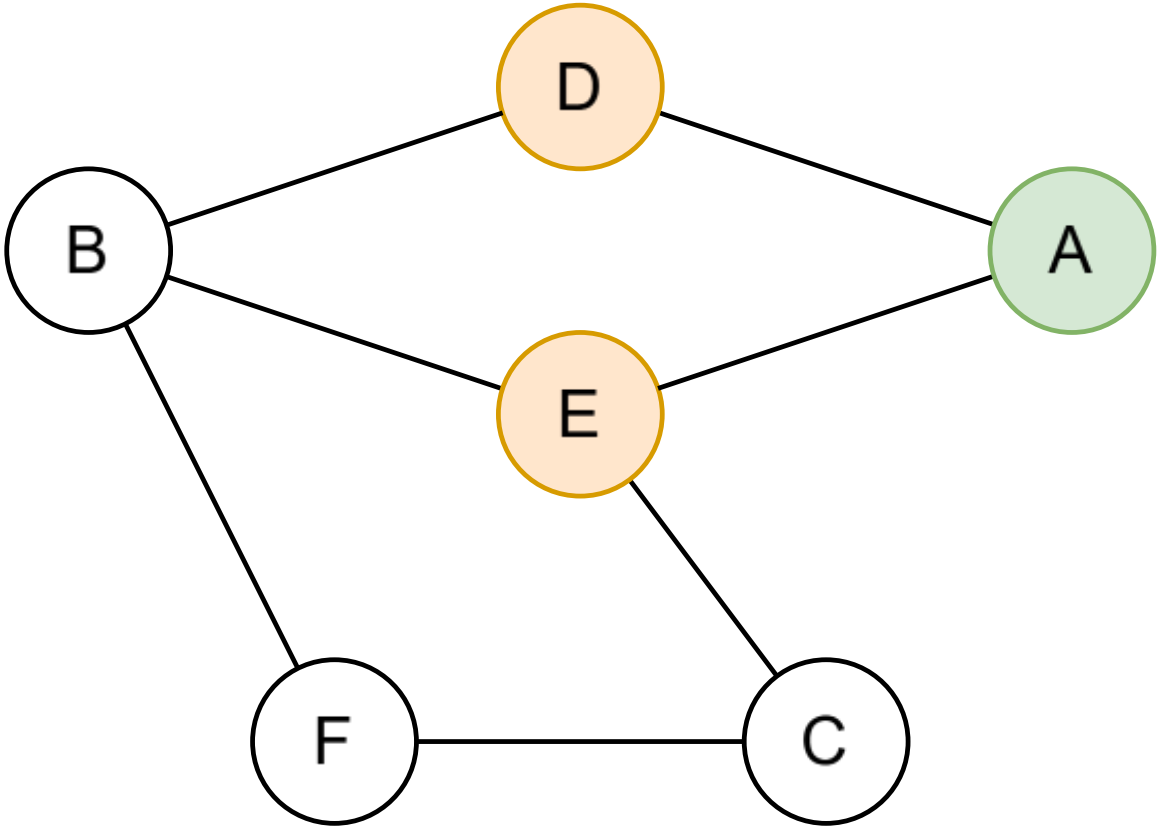
Bipartitos



Buscamos dividir los nodos en dos grupos, sin aristas entre nodos del mismo grupo.

2-Coloring: Coloreado de grafos con dos colores.

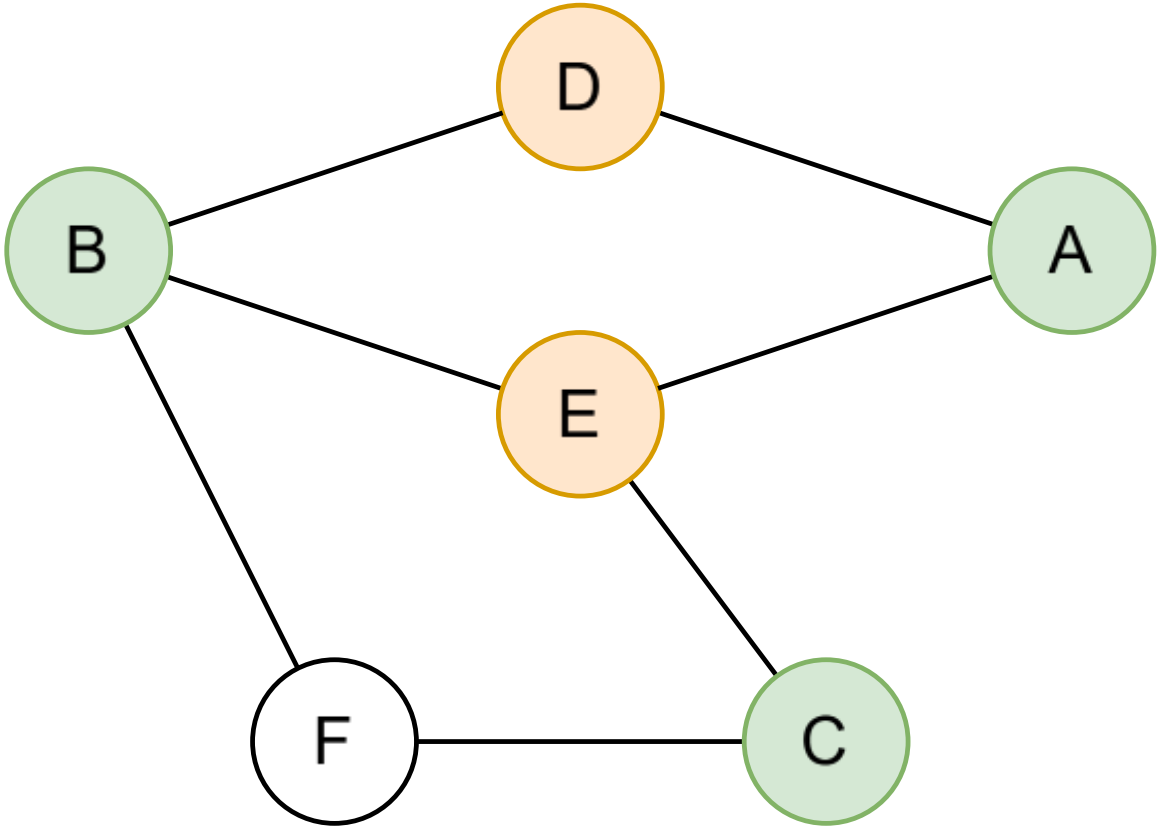
Bipartitos



Buscamos dividir los nodos en dos grupos, sin aristas entre nodos del mismo grupo.

2-Coloring: Coloreado de grafos con dos colores.

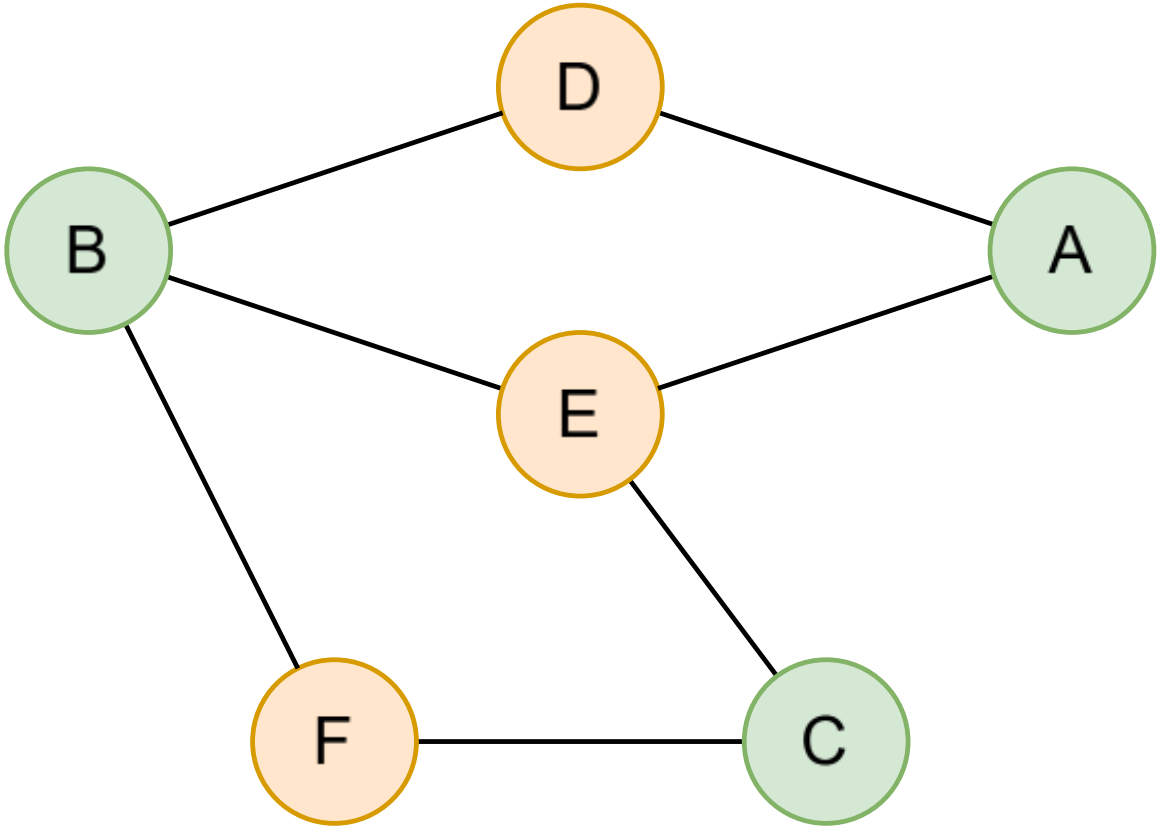
Bipartitos



Buscamos dividir los nodos en dos grupos, sin aristas entre nodos del mismo grupo.

2-Coloring: Coloreado de grafos con dos colores.

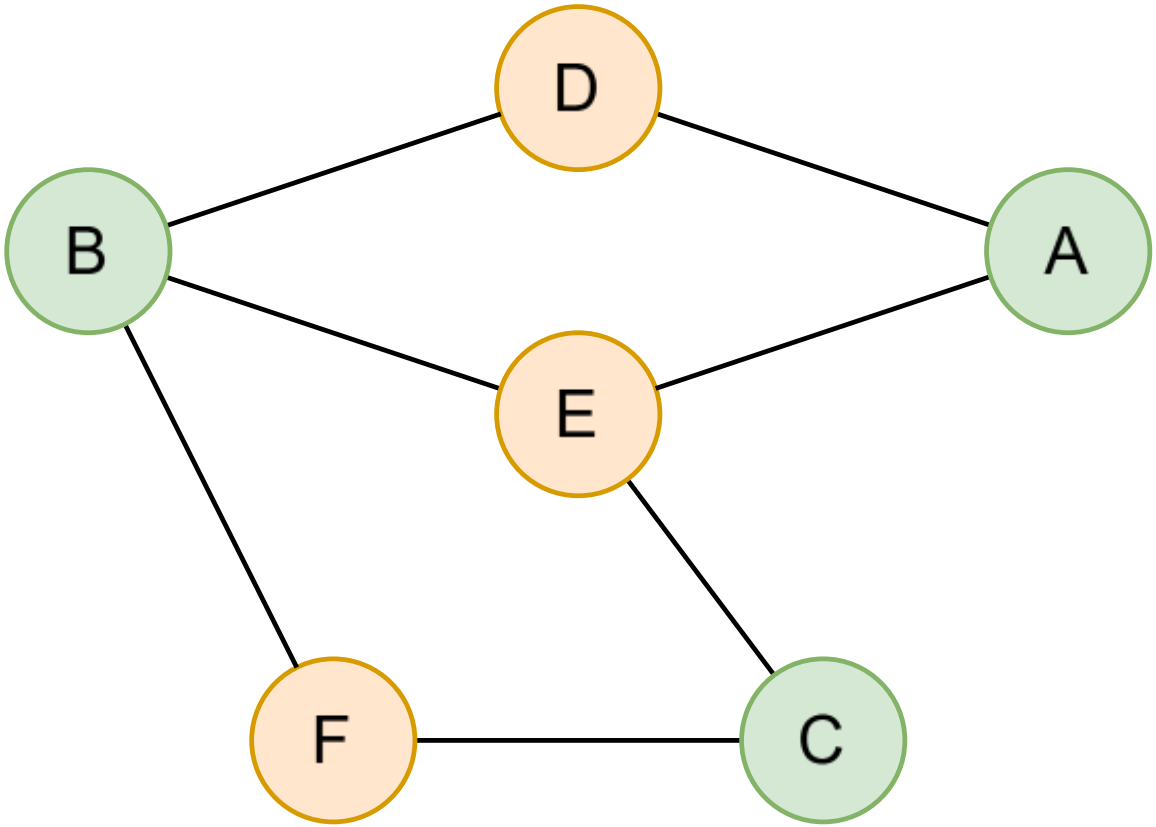
Bipartitos

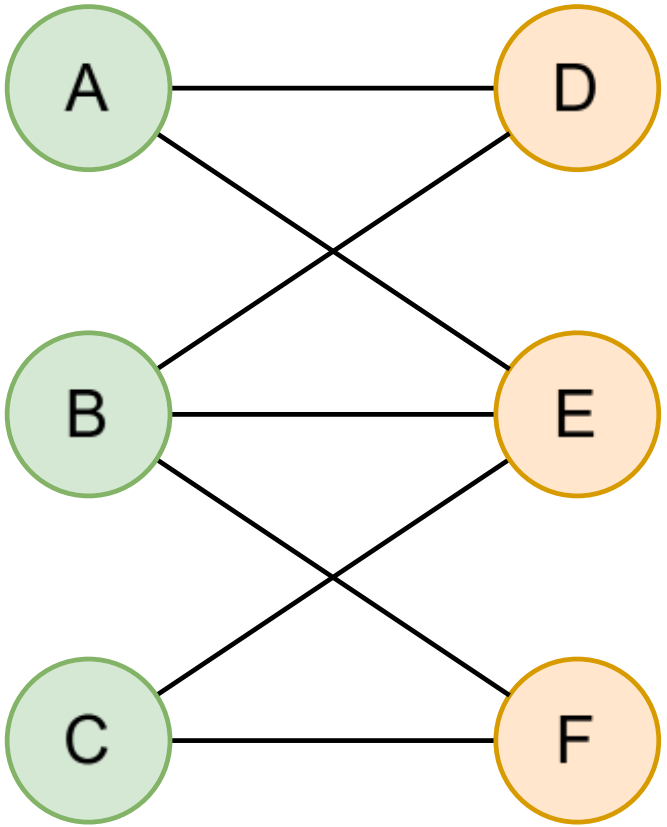


Buscamos dividir los nodos en dos grupos, sin aristas entre nodos del mismo grupo.

2-Coloring: Coloreado de grafos con dos colores.

Bipartitos

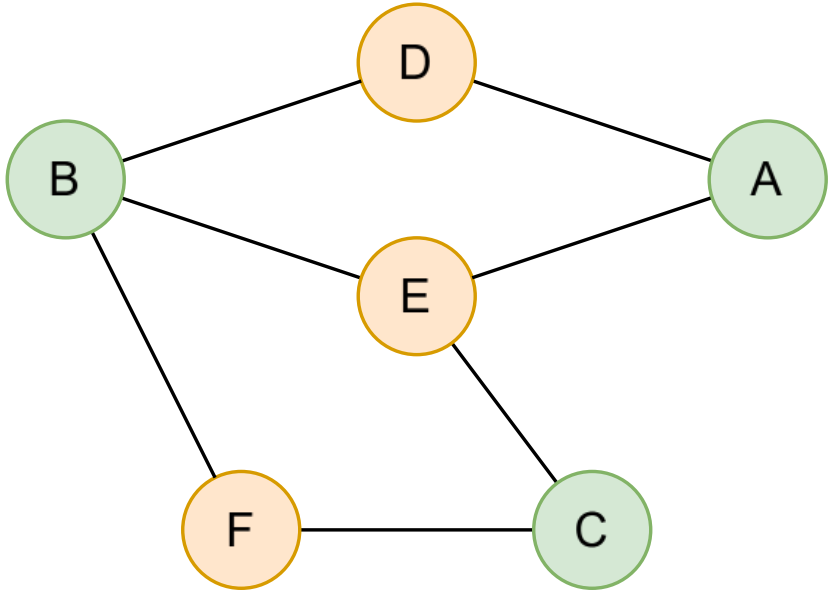




Bipartitos

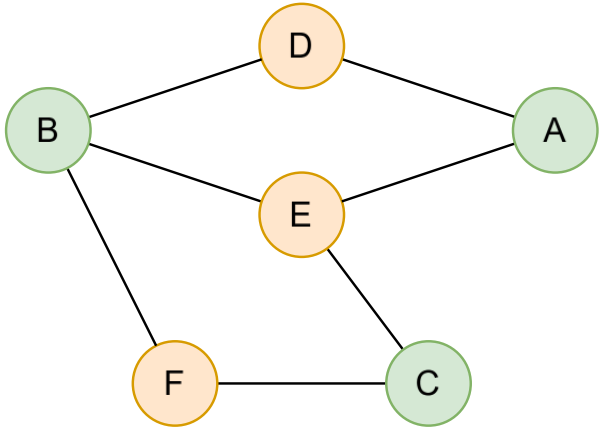
```
1   def dfs(grafo, inicio, visitados):
2       pila = [inicio]
3       visitados.add(inicio)
4       while pila:
5           actual = pila.pop()
6           for vecino in grafo[actual]:
7               if vecino not in visitados:
8                   visitados.add(vecino)
9                   pila.append(vecino)
10
11  def contar_componentes(grafo):
12      visitados, n_componentes = set(), 0
13      for nodo in grafo:
14          if nodo not in visitados:
15              dfs(grafo, nodo, visitados)
16              n_componentes += 1
17      return n_componentes
```

¿Cómo podemos obtener el coloreado?



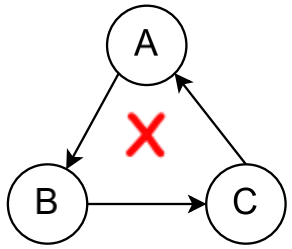
Bipartitos

```
1  def es_bipartito_dfs(grafo, inicio, colores):
2      pila, colores[inicio] = [inicio], 0
3      while pila:
4          actual = pila.pop()
5          for vecino in grafo[actual]:
6              if vecino not in colores:
7                  colores[vecino] = 1 - colores[actual]
8                  pila.append(vecino)
9              elif colores[vecino] == colores[actual]:
10                 return False
11     return True
12
13 def es_bibartito(grafo):
14     colores = {}
15     for nodo in grafo_ejemplo:
16         if nodo not in colores_globales:
17             if not es_bipartito_dfs(grafo, nodo, colores):
18                 return False
19     return True
```



Ordenamiento Topológico

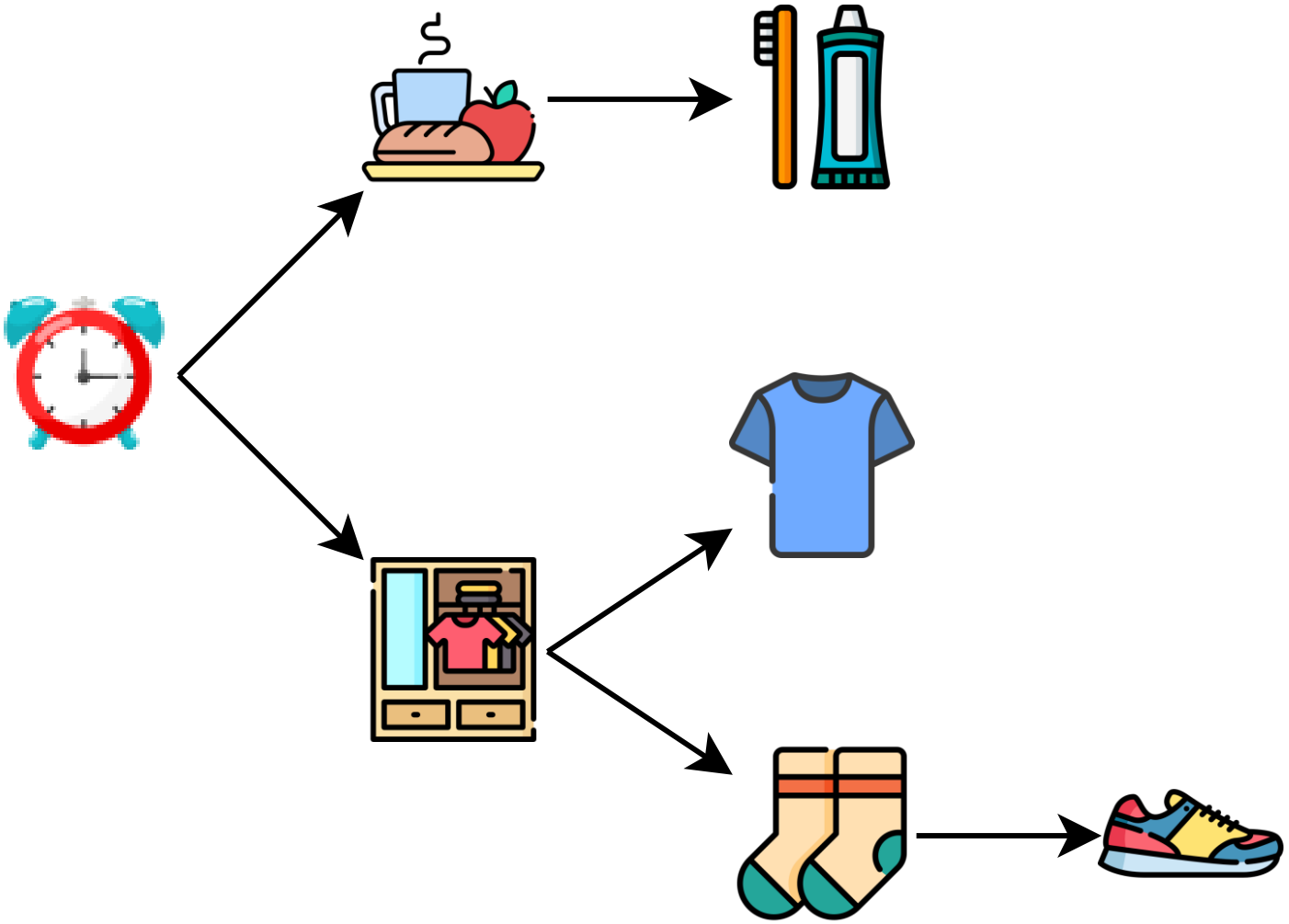
Grafos Dirigidos Acíclicos



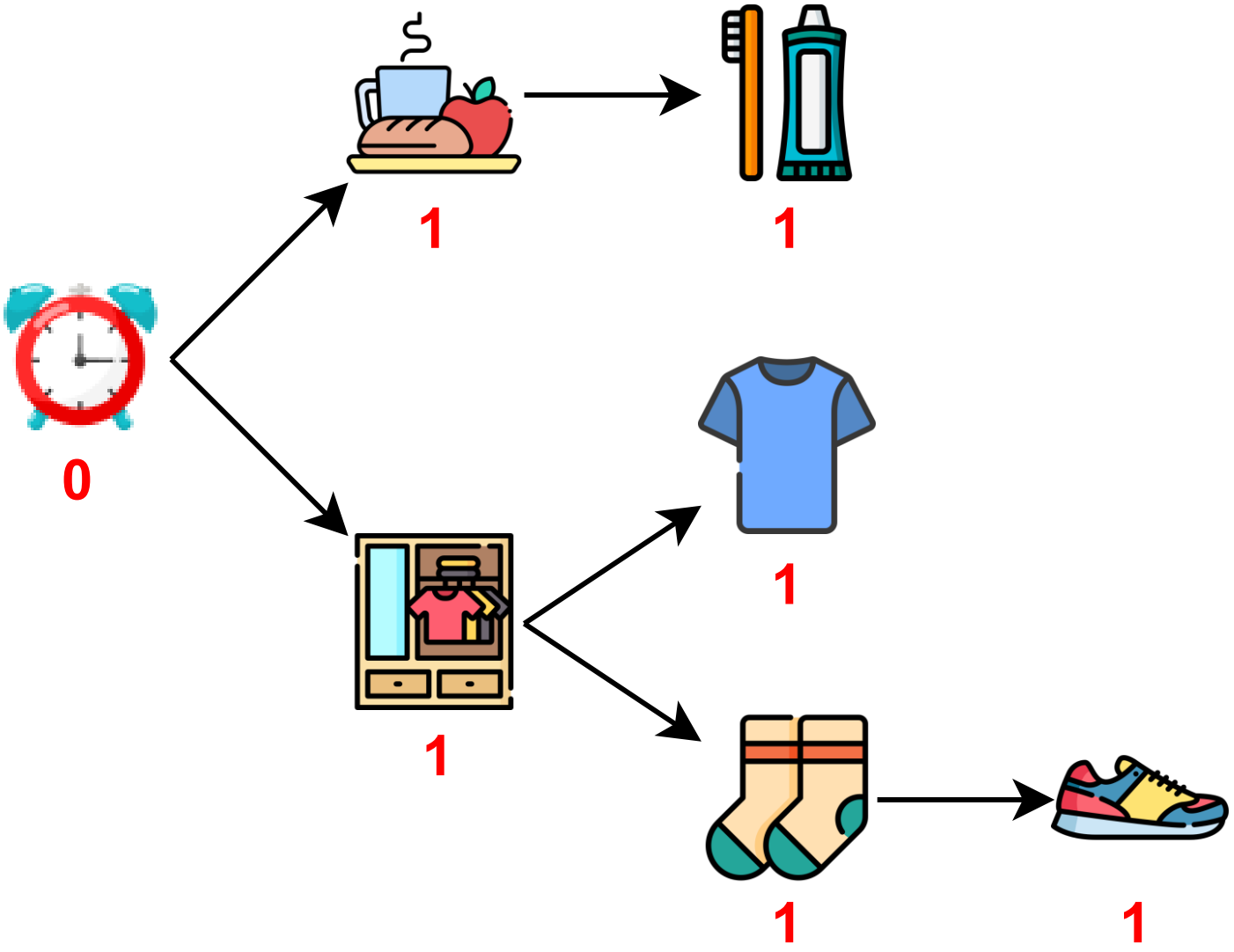
Aplicaciones practicas:

- **Prerrequisitos**: Antes de realizar B, se tiene que realizar A.
- **Gestión de proyectos**: Para poder empezar la tarea B, hay que terminar A.

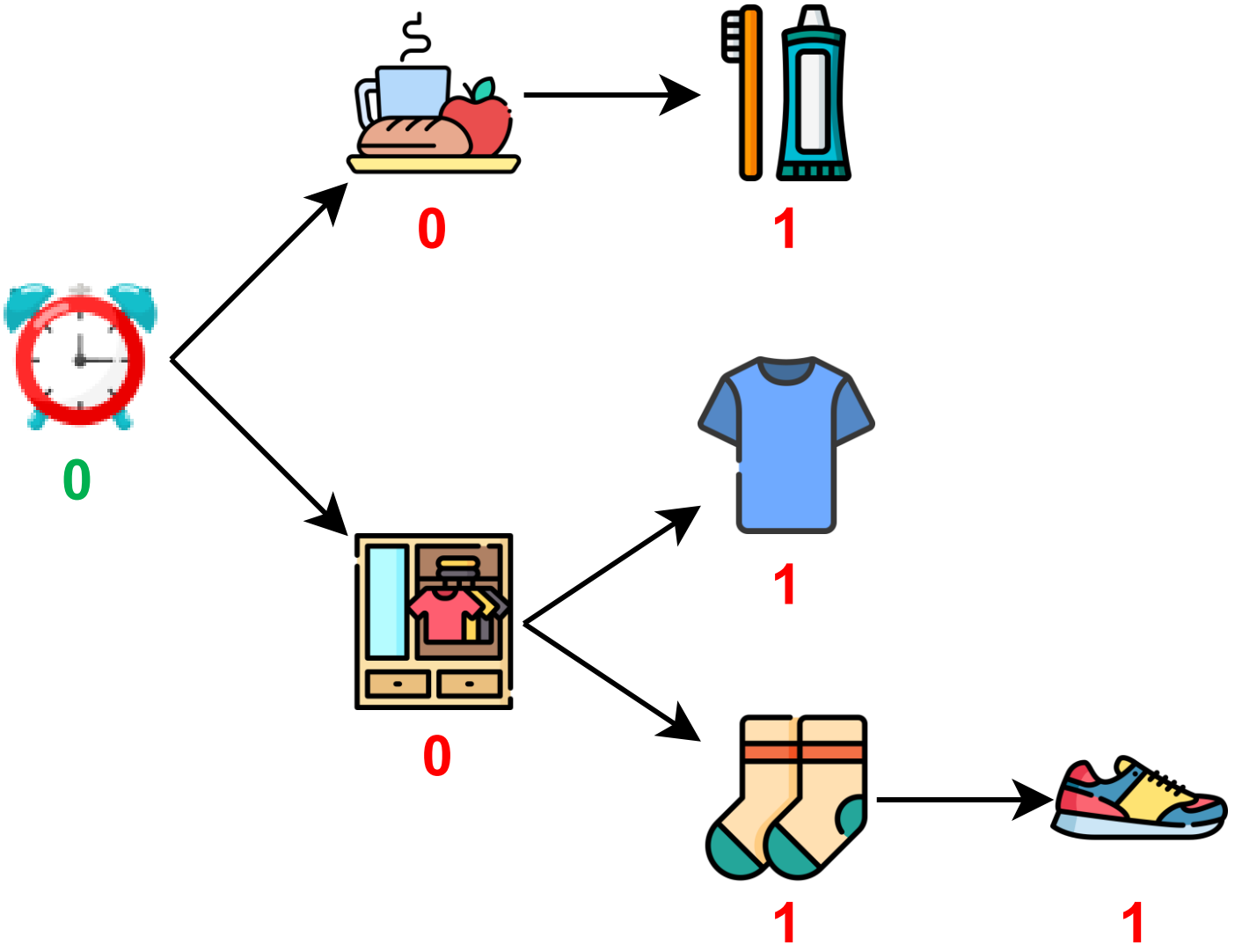
Ordenamiento Topológico



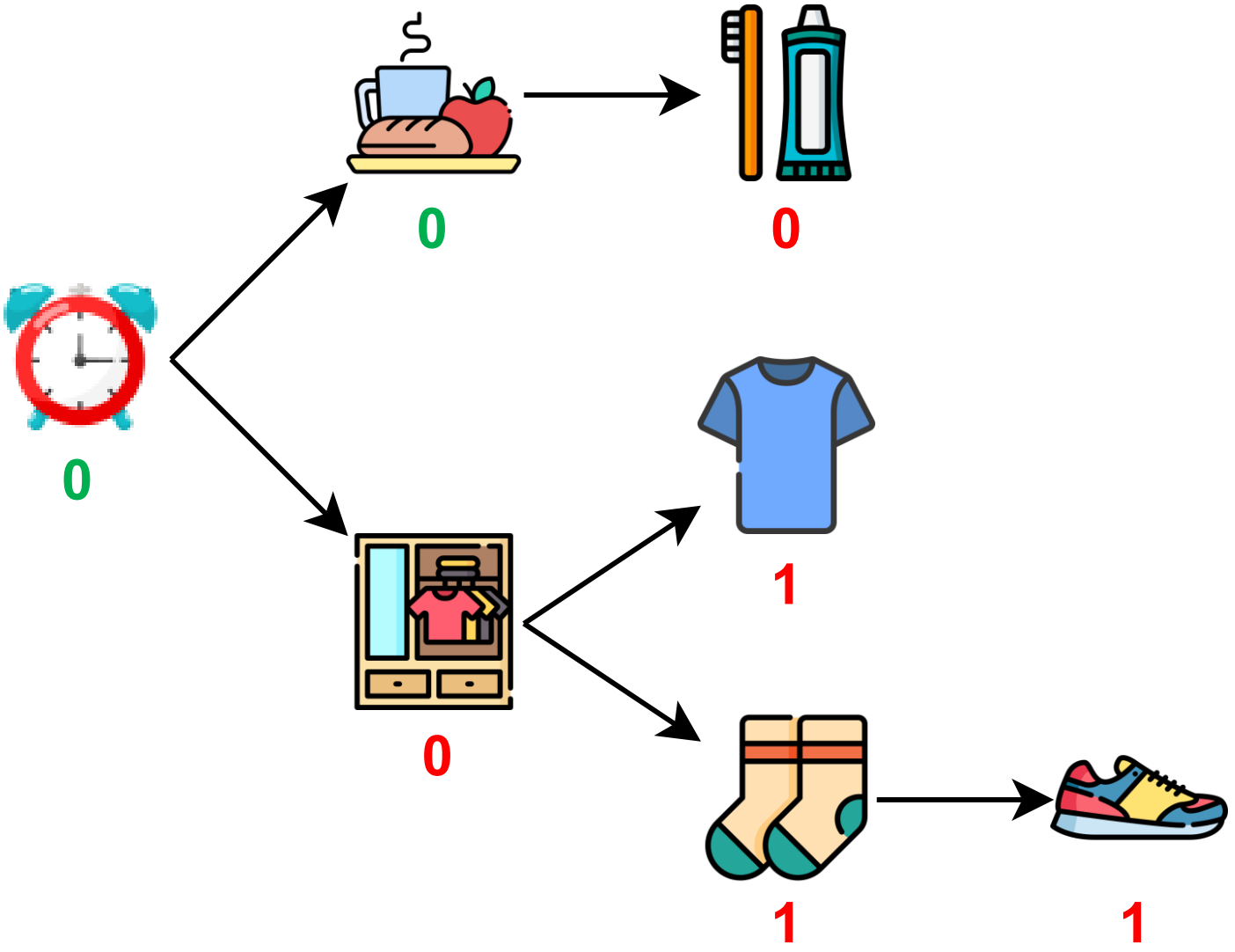
Ordenamiento Topológico



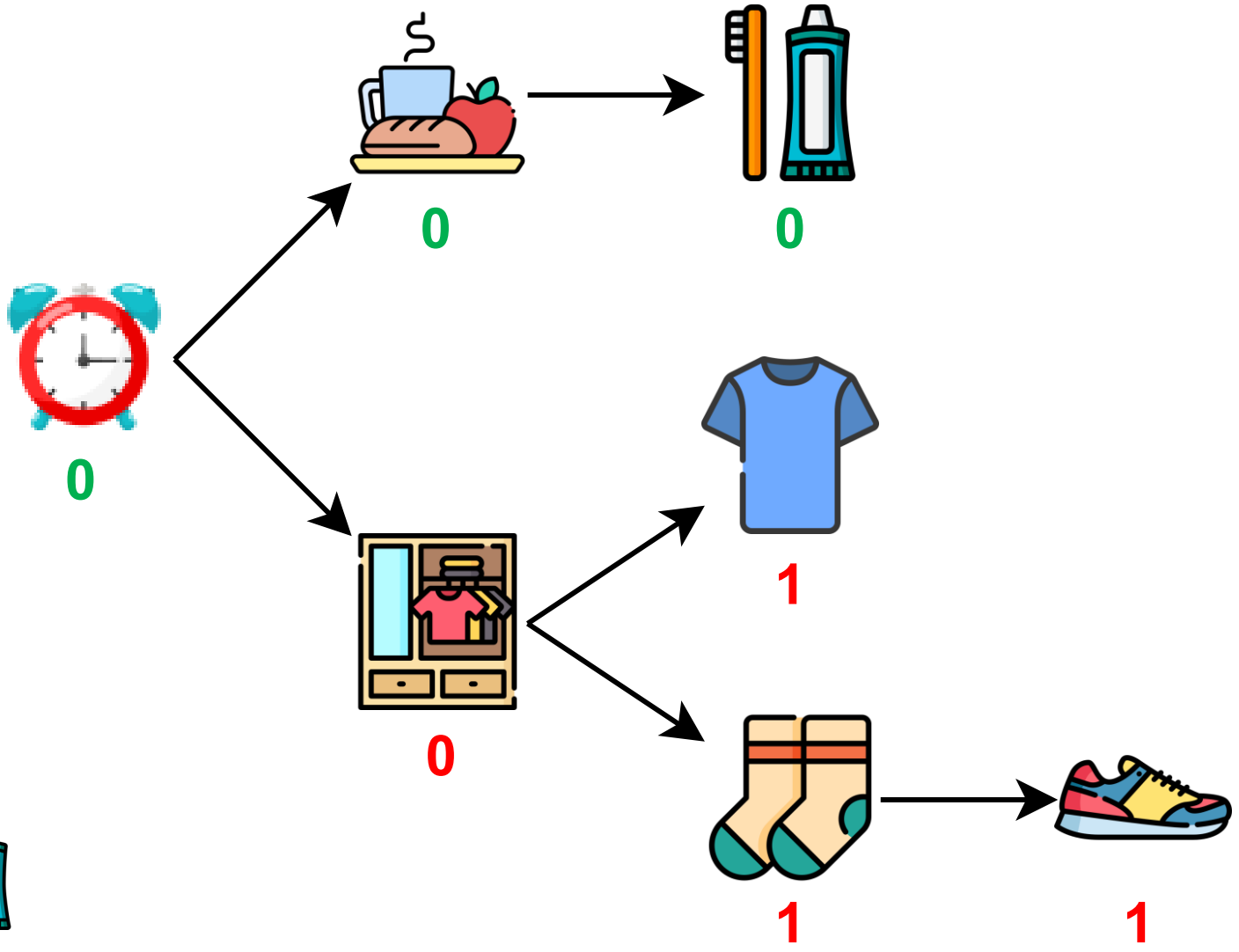
Ordenamiento Topológico



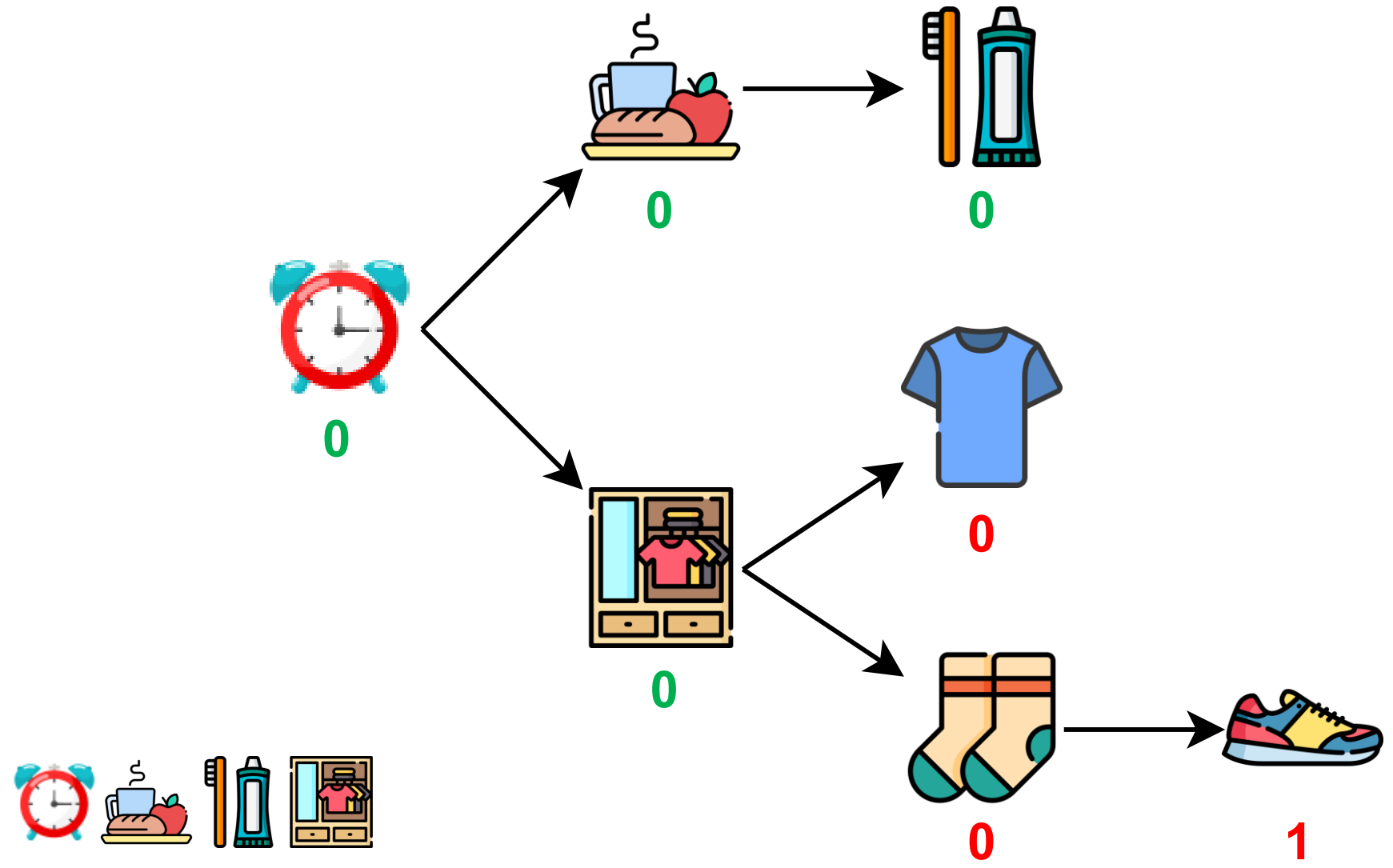
Ordenamiento Topológico



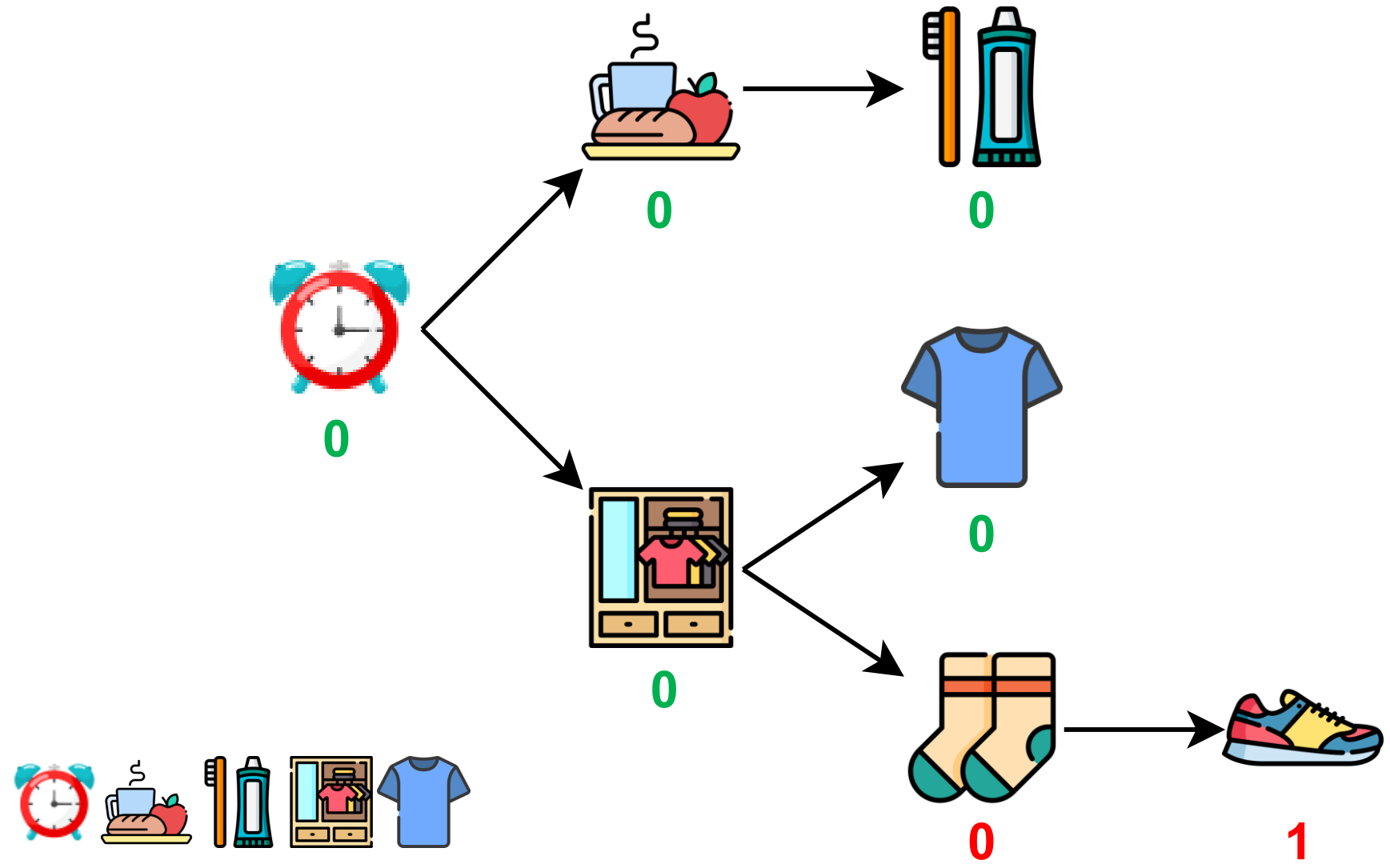
Ordenamiento Topológico



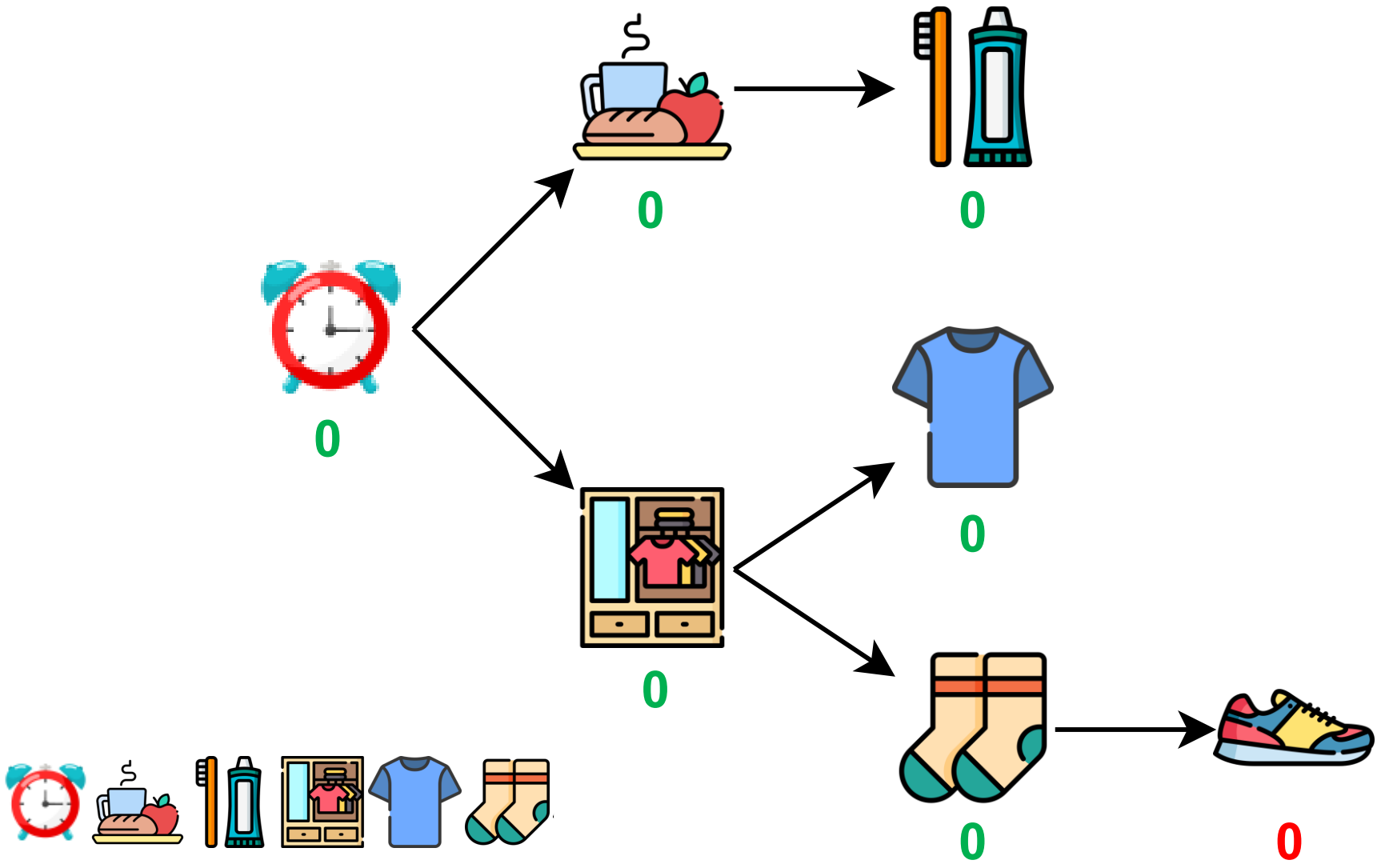
Ordenamiento Topológico



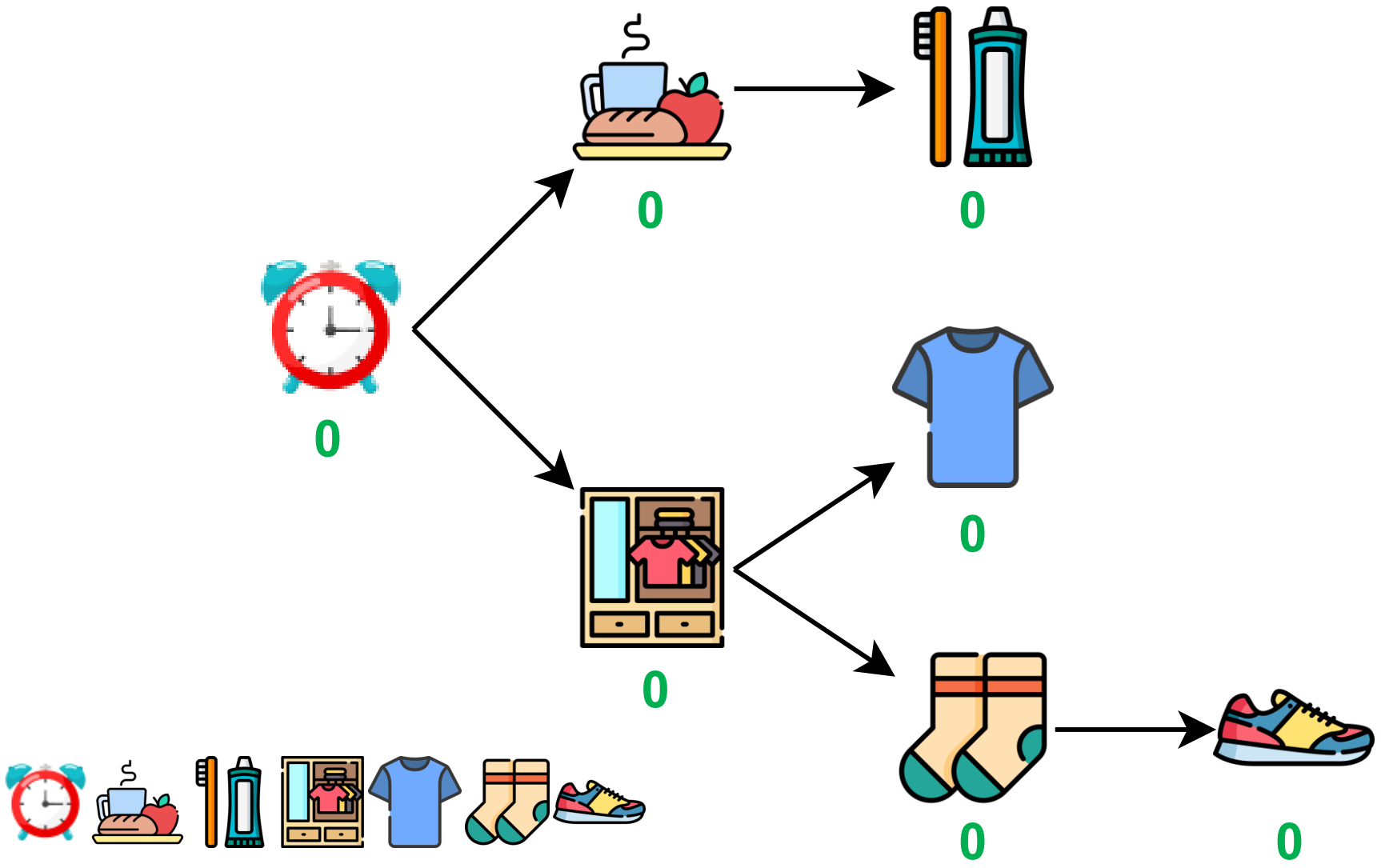
Ordenamiento Topológico



Ordenamiento Topológico



Ordenamiento Topológico



Ordenamiento Topológico

```python
import heapq

def ordenamiento_topologico(grafo):
    in_degree = ...









    orden_final = []
    cola_prioridad = [u for u in grafo if in_degree[u] == 0]
    heapq.heapify(cola_prioridad)

    while cola_prioridad:
        ...









    return orden_final
```

Ordenamiento Topológico

```python
import heapq

def ordenamiento_topologico(grafo):
    in_degree = {u: 0 for u in grafo}
    for u in grafo:
        for v in grafo[u]:
            in_degree[v] = in_degree.get(v, 0) + 1

    orden_final = []
    cola_prioridad = [u for u in grafo if in_degree[u] == 0]
    heapq.heapify(cola_prioridad)

    while cola_prioridad:
        u = heapq.heappop(cola_prioridad)
        orden_final.append(u)
        for v in grafo.get(u, []):
            in_degree[v] -= 1
            if in_degree[v] == 0:
                heapq.heappush(cola_prioridad, v)
    return orden_final
```

Problemas

El "País de los Círculos" quiere invadir el "País de los Rectángulos", el cual está representado por una cuadrícula de tamaño $N \times M$. Plantean la siguiente estrategia:

- **Día 1**: Los paracaidistas ocupan simultáneamente K celdas específicas que están débilmente protegidas.
- **Días siguientes**: En cada nuevo día, la ocupación se expande a todas las celdas adyacentes (arriba, abajo, izquierda y derecha) de las celdas ya ocupadas.

Debes calcular cuántos días en total tardará el ejército en ocupar todas las celdas de la cuadrícula de $N \times M$.

Problemas

Tienes un sistema de dependencias (un Makefile). Cuando un archivo fuente cambia, no solo se debe recompilar ese archivo, sino también todos los que dependen de él, y los que dependen de esos, y así sucesivamente, con estas reglas:

- **Dependencia Directa**: Si el archivo A depende de B, y B cambia, entonces A debe ser recompilado.
- **Orden de Compilación**: Si ambos archivos (A y B) necesitan recompilarse, B debe compilarse antes que A (primero las dependencias, luego el archivo que las usa).

Dado un archivo específico que ha cambiado, piden:

- Identificar todos los archivos afectados (directa o indirectamente).
- Listarlos en un orden válido que respete las dependencias.

Problemas

Se te entrega una cuadrícula de tamaño $N \times N$ llena de números enteros. Tu objetivo es seleccionar exactamente N celdas de la cuadrícula siguiendo dos restricciones estrictas:

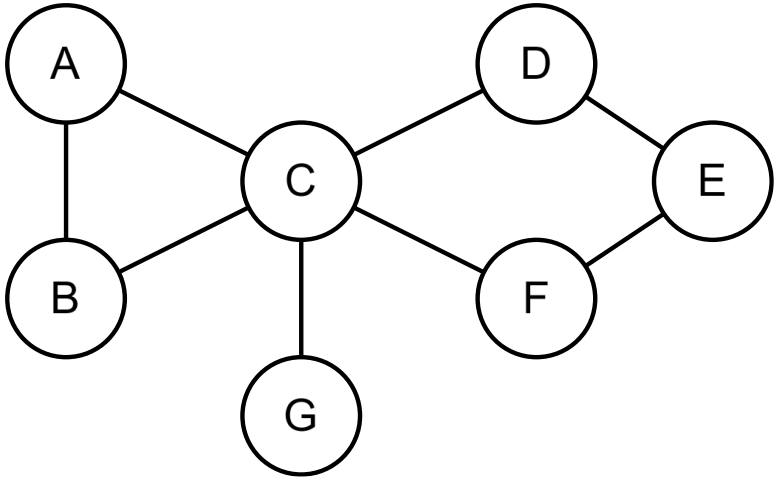
1. Debes elegir exactamente una celda por cada fila.
2. Debes elegir exactamente una celda por cada columna.

De todas las formas posibles de elegir esas N celdas, debes encontrar aquella donde el valor mínimo entre las celdas elegidas sea lo más grande posible.

Puntos de articulación

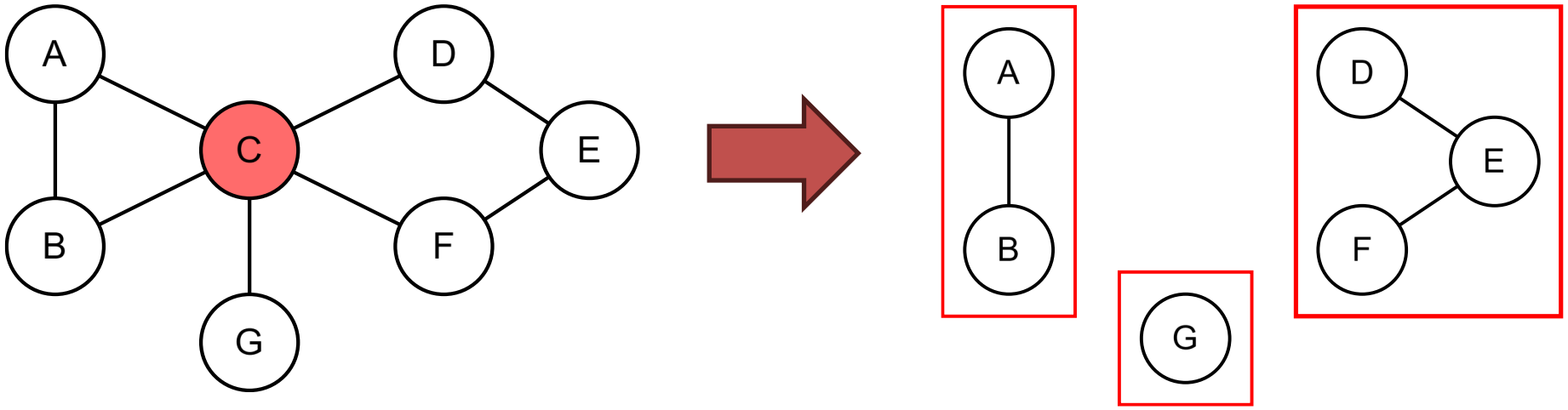
En algunos problemas nos pueden preguntar qué pasa con un grafo si uno de sus nodos desaparece

Puede haber nodos críticos para la conectividad del grafo!



Puntos de articulación

Un **punto de articulación** es un nodo que, al eliminarlo (junto con sus aristas), **aumenta** el número de **componentes conexas** del grafo.



Puntos de articulación

¿Cómo podemos detectar los puntos de articulación?

Idea:

- Explorar el grafo con **DFS**
- Comprobar si los nodos descendientes pueden llegar a otros nodos sin pasar por su padre

Puntos de articulación

Un nodo **u** es punto de articulación si existe un hijo **v** tal que:

low[v] ≥ desc[u]

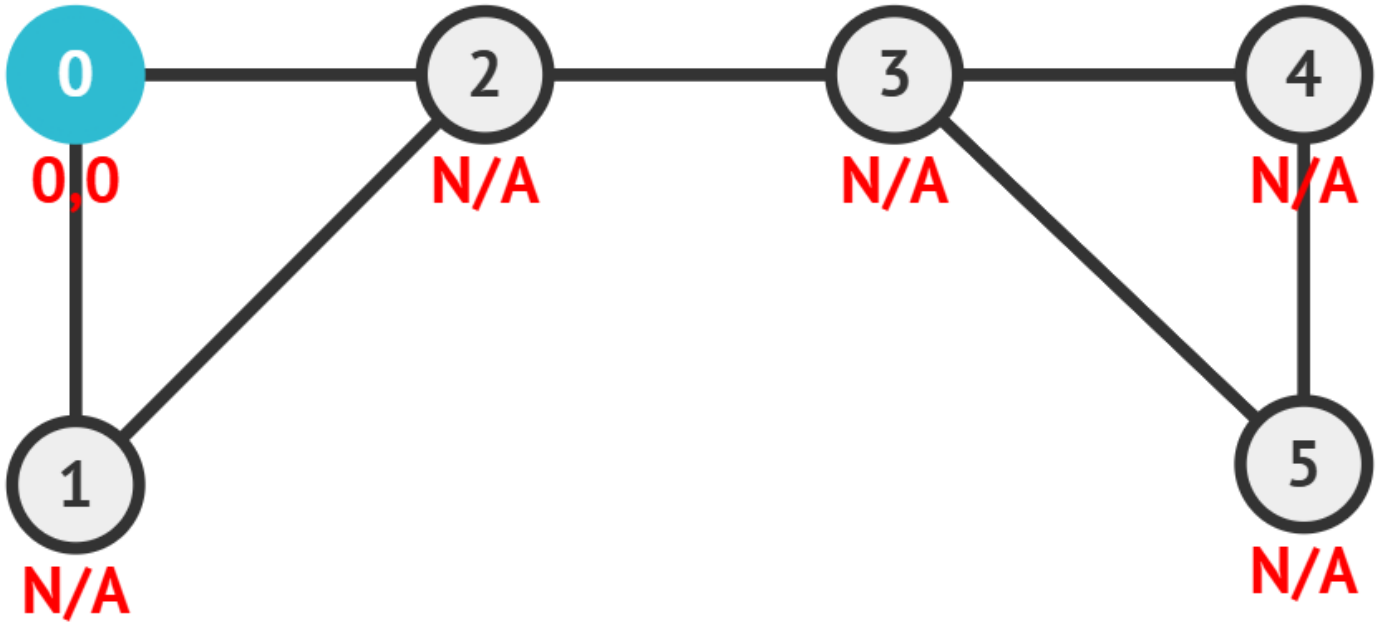
Conceptos clave:

- **desc[u]**: tiempo de descubrimiento del nodo
- **low[u]**: el tiempo de descubrimiento más bajo al que puede llegar u usando:
    - aristas del árbol DFS
    - aristas de retroceso

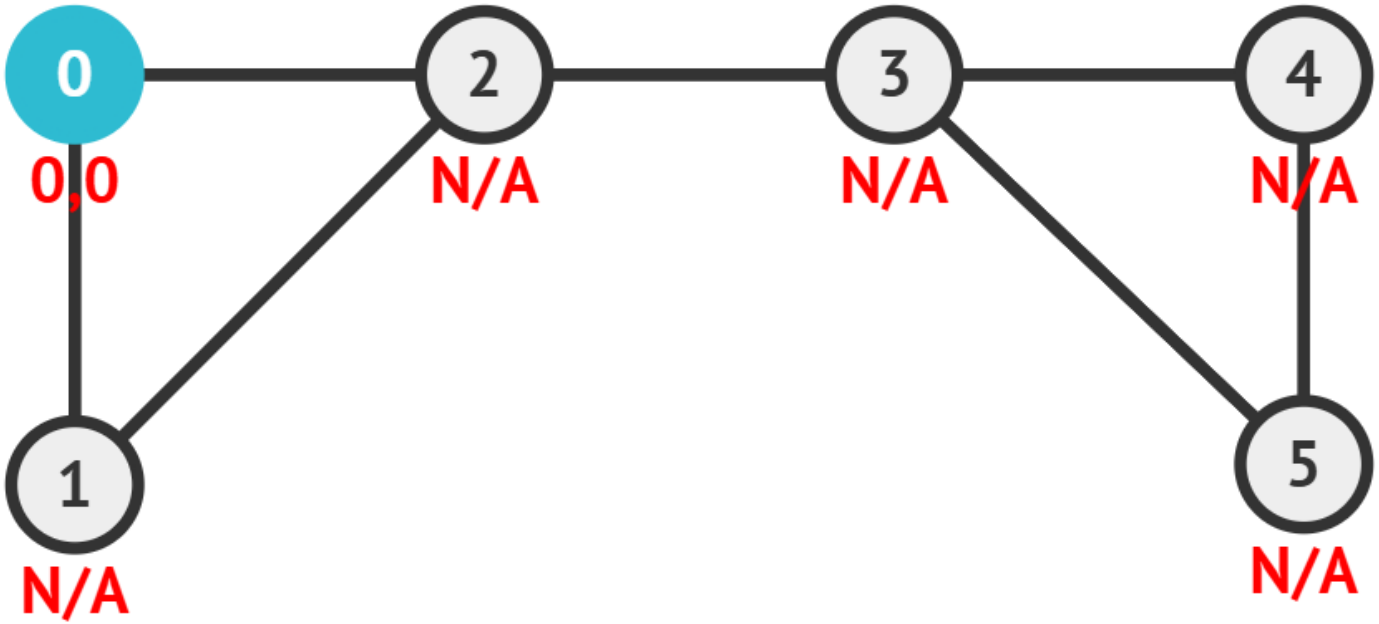
! La raíz se trata distinto: es punto de articulación si tiene **más de un hijo**

Puntos de articulación

¿Cómo obtenemos los puntos de articulación de este grafo?

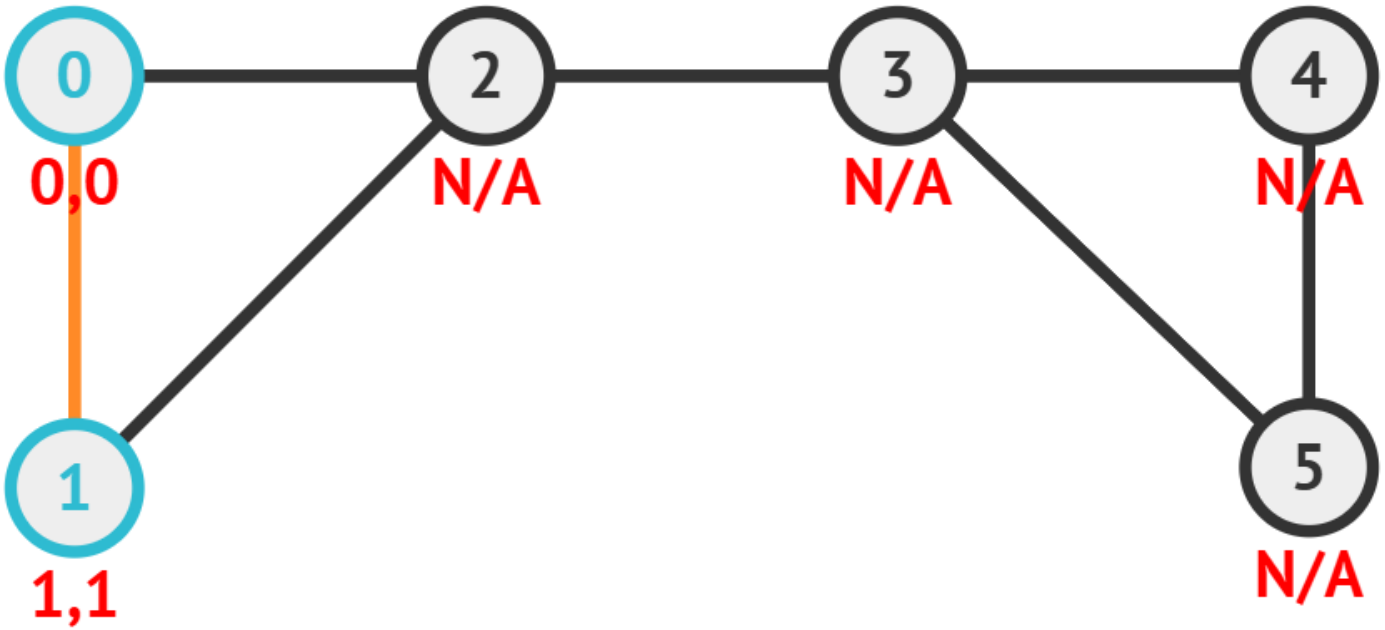


Puntos de articulación



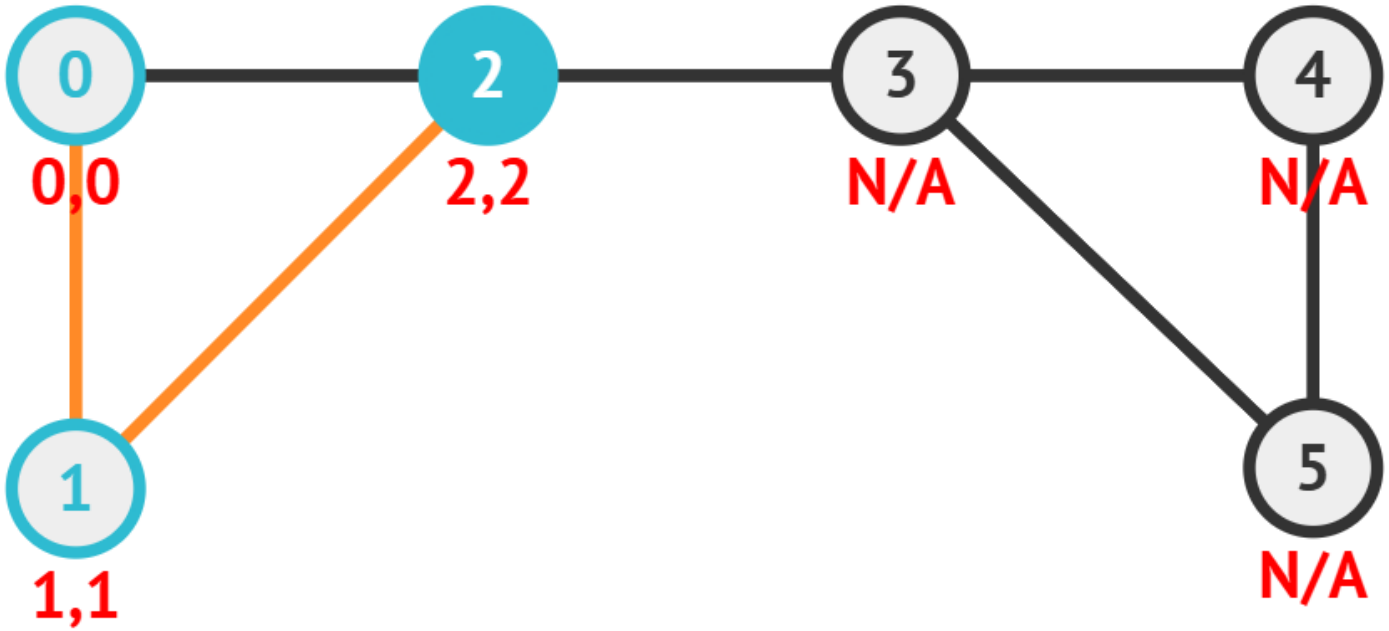
Nodo	Desc	Low
0	0	0
1	?	?
2	?	?
3	?	?
4	?	?
5	?	?

Puntos de articulación



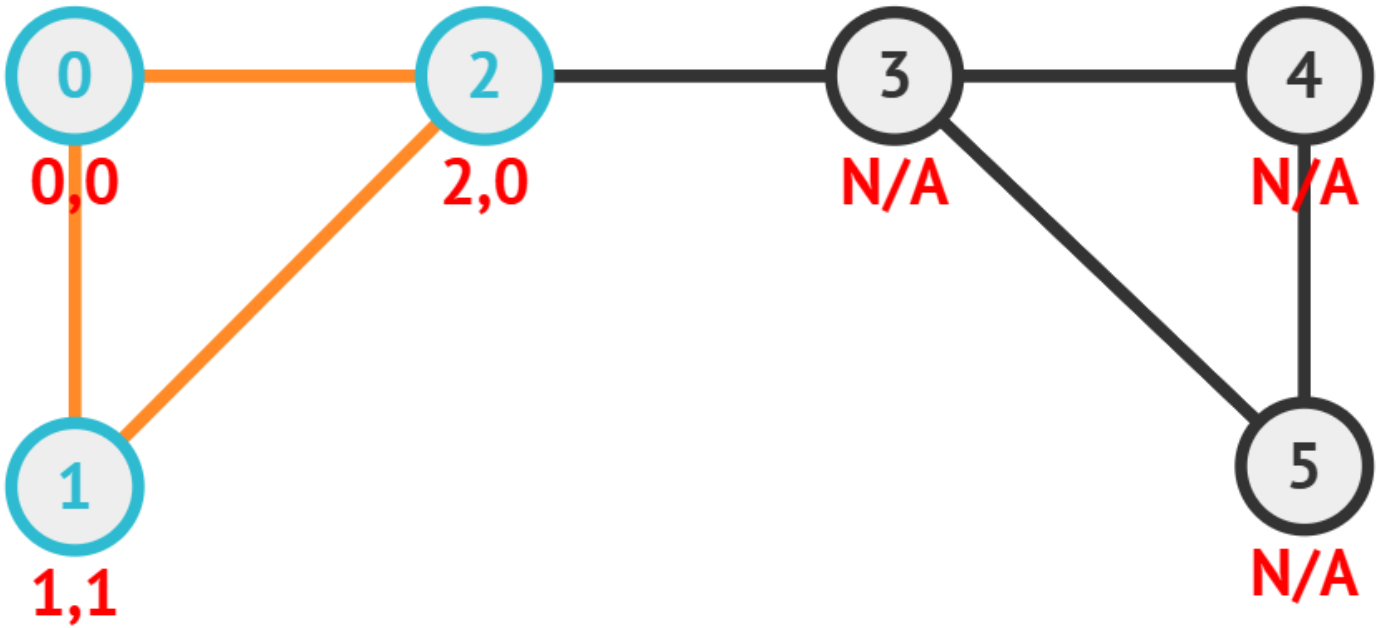
Nodo	Desc	Low
0	0	0
1	1	1
2	?	?
3	?	?
4	?	?
5	?	?

Puntos de articulación



Nodo	Desc	Low
0	0	0
1	1	1
2	2	2
3	?	?
4	?	?
5	?	?

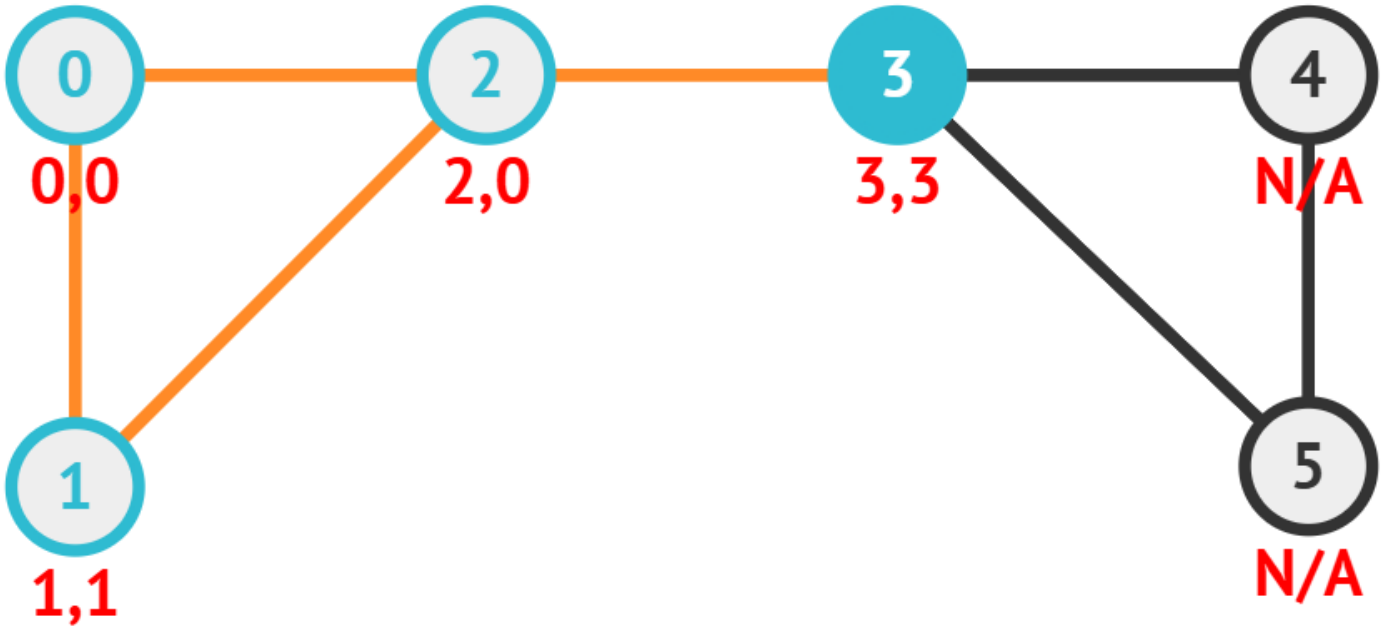
Puntos de articulación



Nodo	Desc	Low
0	0	0
1	1	1
2	2	0
3	?	?
4	?	?
5	?	?

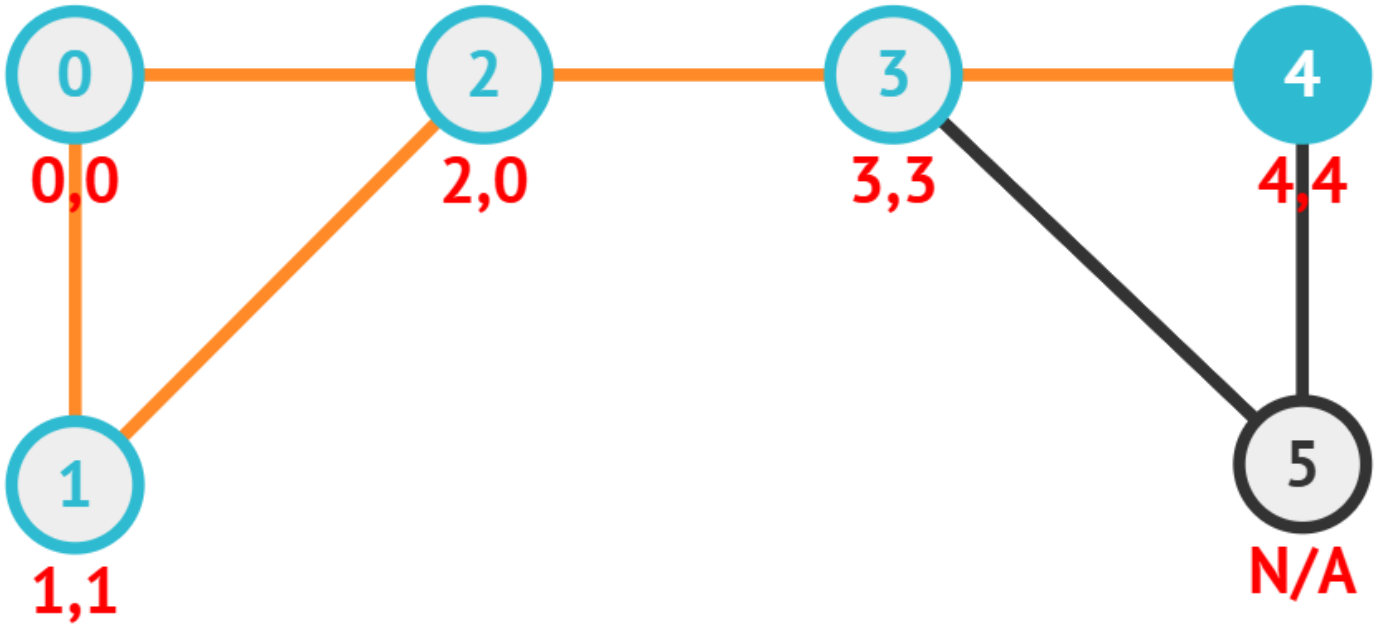
Comprobamos vecinos y si ya está visitado y no es padre:
low[u] = min(low[u], desc[adj])

Puntos de articulación



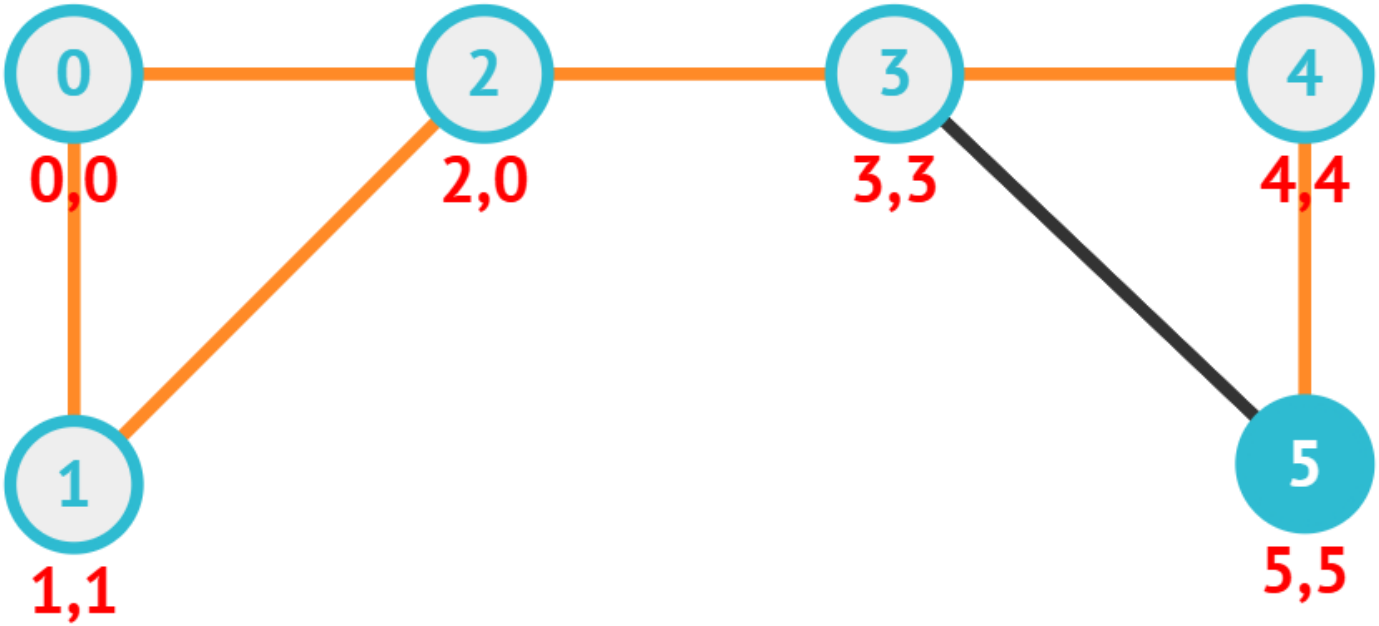
Nodo	Desc	Low
0	0	0
1	1	1
2	2	0
3	3	3
4	?	?
5	?	?

Puntos de articulación



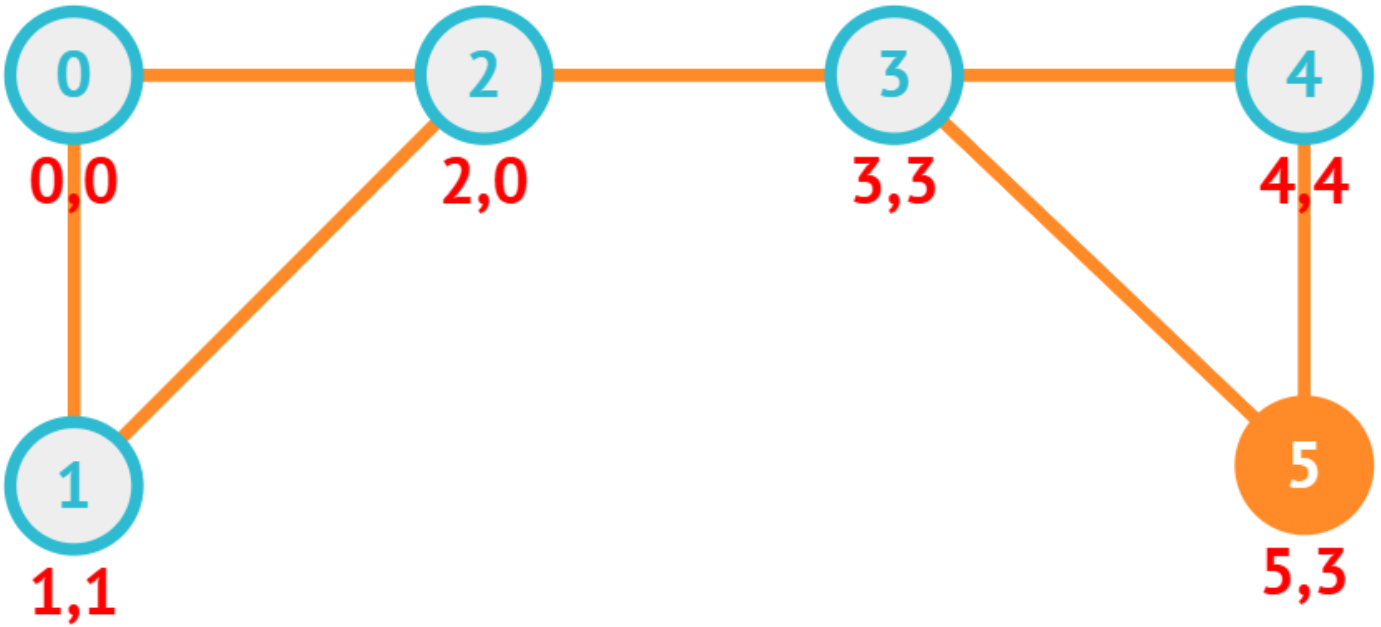
Nodo	Desc	Low
0	0	0
1	1	1
2	2	0
3	3	3
4	4	4
5	?	?

Puntos de articulación



Nodo	Desc	Low
0	0	0
1	1	1
2	2	0
3	3	3
4	4	4
5	5	5

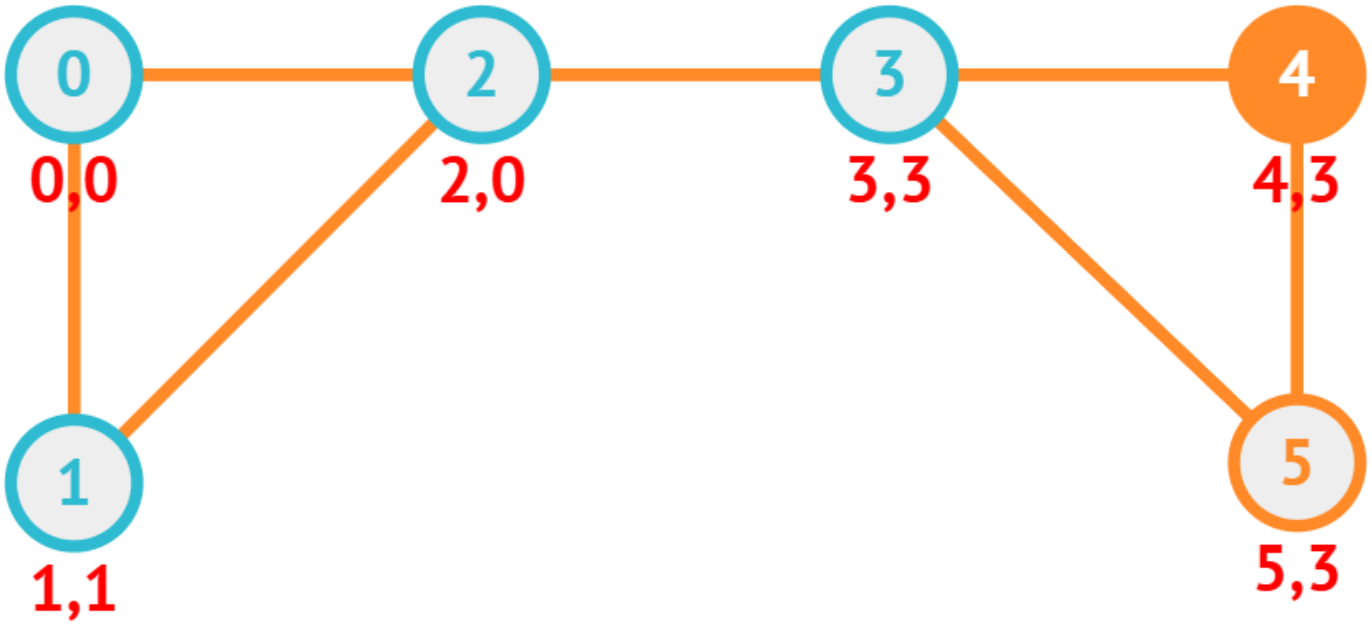
Puntos de articulación



Nodo	Desc	Low
0	0	0
1	1	1
2	2	0
3	3	3
4	4	4
5	5	3

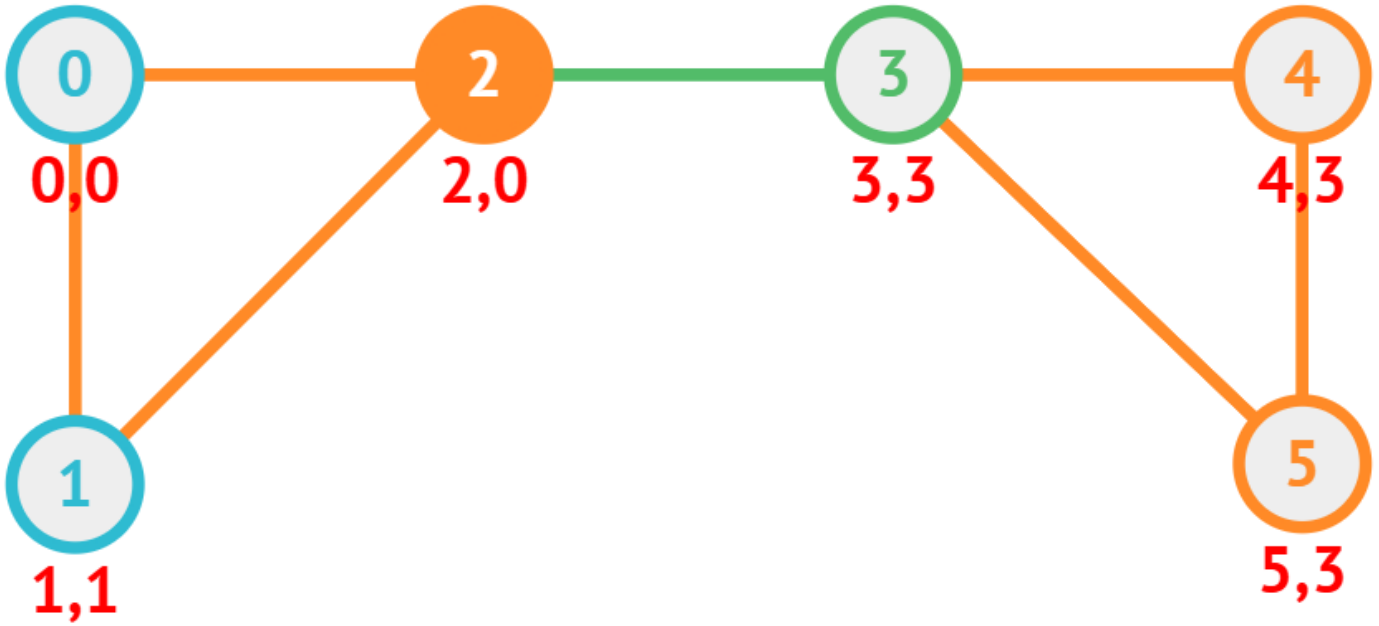
Recalculamos descubrimiento mínimo con:
low[u] = min(low[u], low[adj])

Puntos de articulación



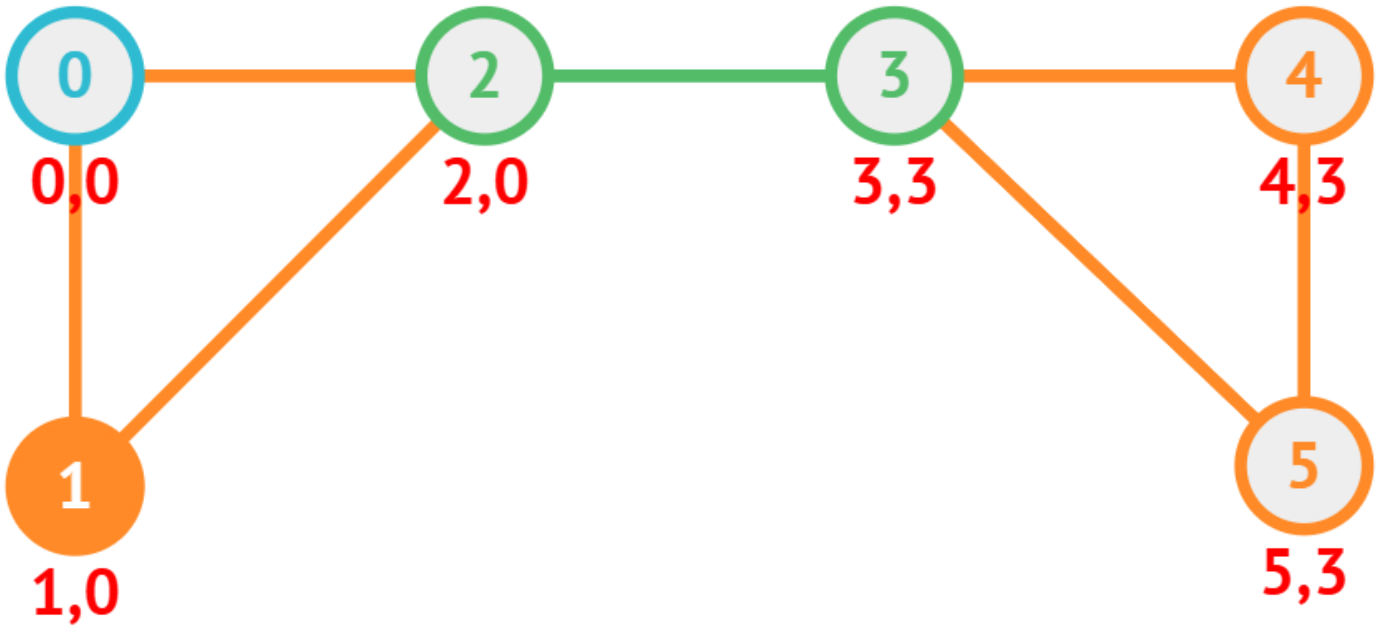
Nodo	Desc	Low
0	0	0
1	1	1
2	2	0
3	3	3
4	4	3
5	5	3

Puntos de articulación



Nodo	Desc	Low
0	0	0
1	1	1
2	2	0
3	3	3
4	4	3
5	5	3

Puntos de articulación



Un nodo **u** es punto de articulación si existe un hijo **v** tal que **low[v] ≥ disc[u]**

low[3] ≥ desc[2] → 2 es punto de articulación
low[4] ≥ desc[3] → 3 es punto de articulación

Nodo	Desc	Low
0	0	0
1	1	0
2	2	0
3	3	3
4	4	3
5	5	3

Puntos de articulación

```
def dfsRec(g, visited, v, desc, low, padre, AP, time):
    visited[v] = True
    time += 1
    desc[v] = low[v] = time
    child = 0
    for adj in g[v]:
        if not visited[adj]:
            child += 1
            padre[adj] = v
            dfsRec(g, visited, adj, desc, low, padre, AP, time)
            low[v] = min(low[v], low[adj])
            if padre[v] is None and child > 1:
                AP[v] = True
            elif padre[v] is not None and low[adj] >= desc[v]:
                AP[v] = True
        elif padre[v] != adj:
            low[v] = min(low[v], desc[adj])
    return AP
```

Caminos más cortos

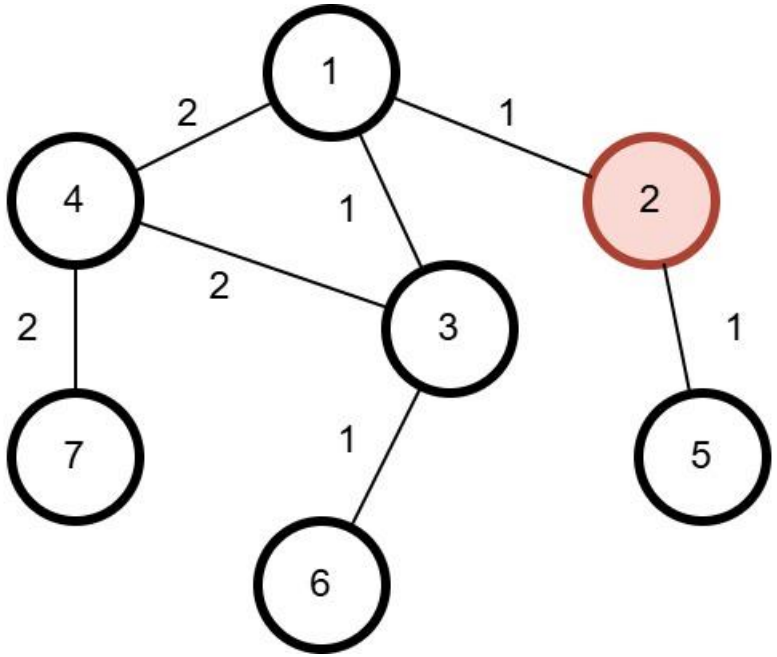
Problema:

Ha empezado una invasión de aliens y están estableciendo bases por todo el país. Los únicos lugares seguros son los que están suficientemente lejos de las bases de los aliens.

¿Cómo averiguamos cuáles son?

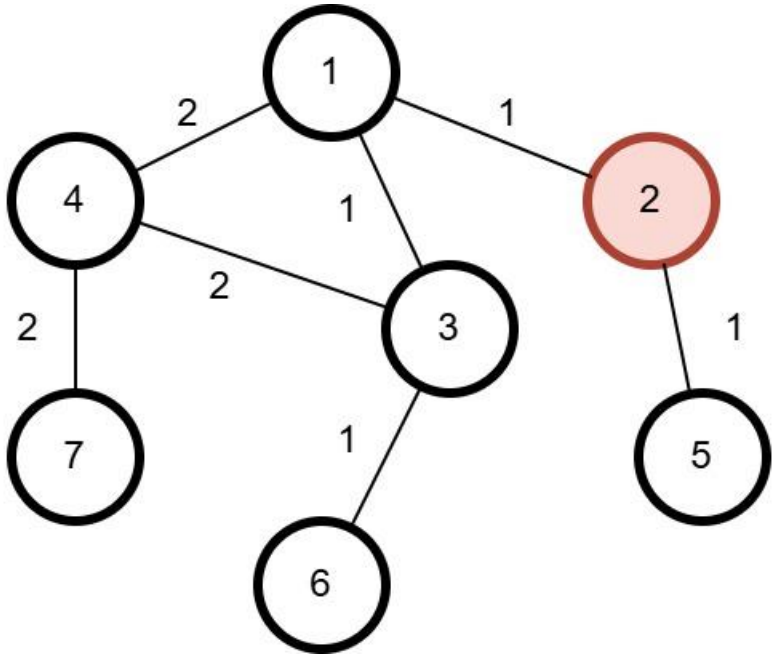
Caminos más cortos

Tenemos el mapa de ciudades con las distancias entre ellas, la ubicación de las bases de los aliens y la distancia segura, en este caso → **5**



Caminos más cortos

Para averiguar qué ciudades son seguras, tenemos que calcular la distancia más corta a cada una de las ciudades y ver si es mayor o igual a la distancia segura.



Caminos más cortos

¿Cómo calculamos las distancias?

ALGORITMO DE DIJKSTRA

Algoritmo voraz: si siempre escogemos el camino más corto, cuando lleguemos al final habremos llegado por el camino más corto

Dijkstra

Complejidad???

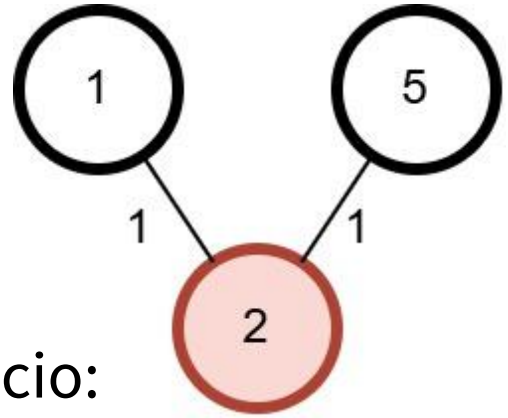
Si iteramos sobre todas las aristas en cada paso es muy costoso
→ $O(N^2)$

Cola de prioridad: guardamos los posibles nuevos caminos con su distancia

- Insertar y eliminar en PQ es O(log n)
- Por cada iteración **O(n log n)**

Dijkstra

1	∞
2	0
3	∞
4	∞
5	∞
6	∞
7	∞

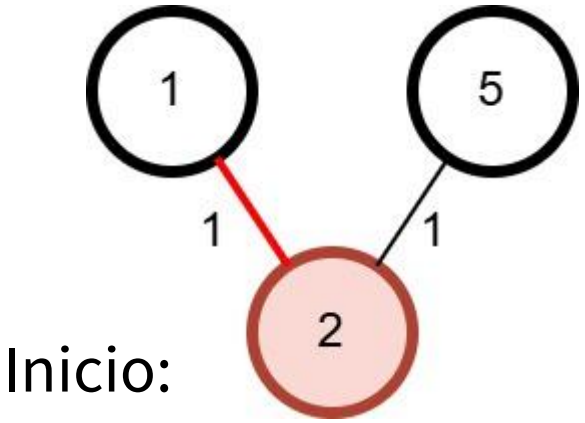


Inicio:

PQ	1,1	5,1

Dijkstra

1	1
2	0
3	∞
4	∞
5	1
6	∞
7	∞

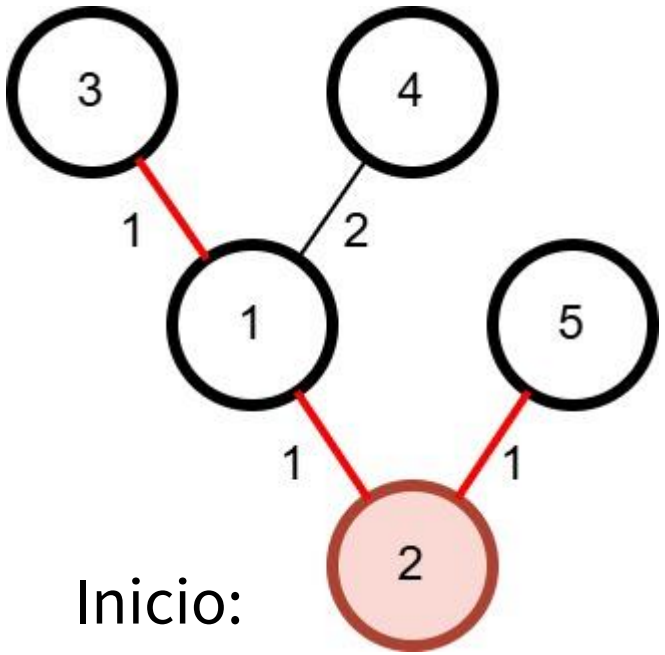


Inicio:

PQ	5,1

Dijkstra

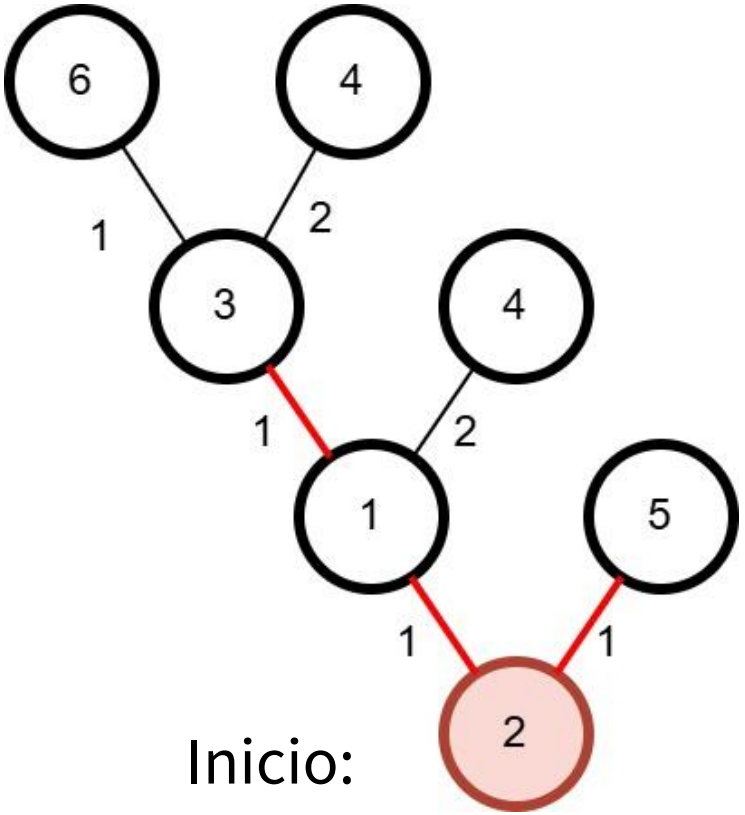
1	1
2	0
3	2
4	3
5	1
6	∞
7	∞



PQ	5,1	3,2	4,3

Dijkstra

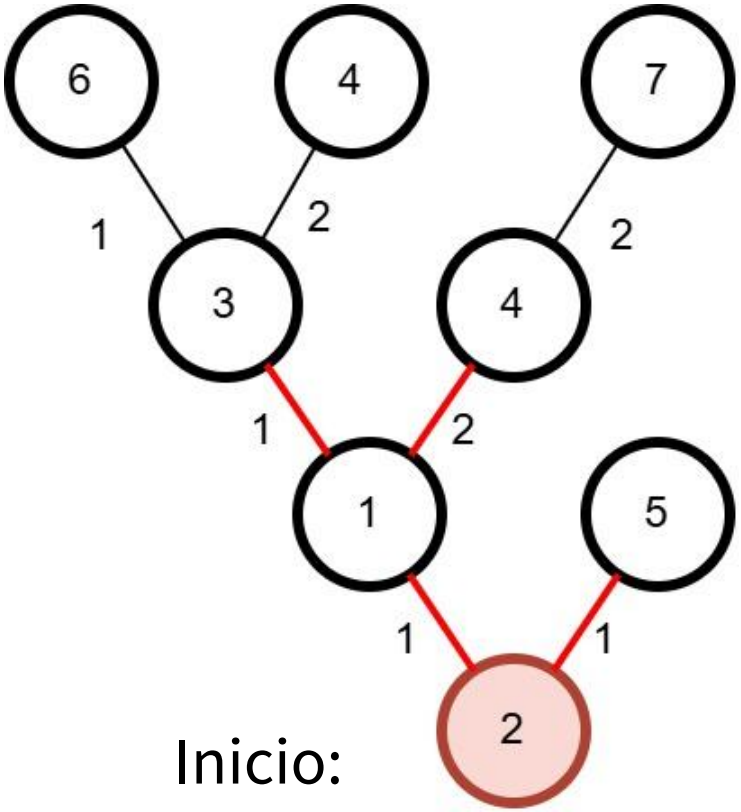
1	1
2	0
3	2
4	3
5	1
6	3
7	∞



PQ	3,2	4,3	6,3

Dijkstra

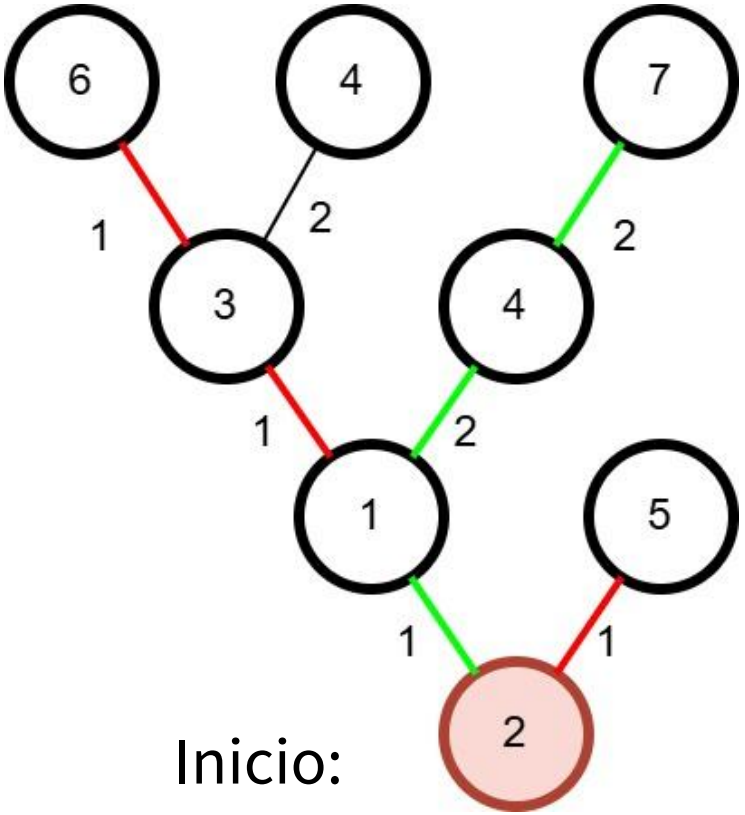
1	1
2	0
3	2
4	3
5	1
6	3
7	5



PQ	4,3	6,3	7,5

Dijkstra

1	1
2	0
3	2
4	3
5	1
6	3
7	5



PQ	

Algoritmo de Dijkstra

```
def dijkstra(graph, start):
    queue = []
    heapq.heappush(queue, (0, start))
    distances = [float('inf') for _ in range(len(graph))]
    distances[start] = 0

    while queue:
    current_distance, current_node = heapq.heappop(queue)
    if current_distance > distances[current_node]:
        continue
    for neighbor, weight in graph[current_node]:
         distance = current_distance + weight
         if distance < distances[neighbor]:
             distances[neighbor] = distance
             heapq.heappush(queue, (distance, neighbor))
    return distances
```

Dijkstra

Problema propuesto:
https://open.kattis.com/problems/invasion

Encontrar el número de ciudades seguras según los aliens van construyendo sus bases

- Primera base: ¿cuántas ciudades son seguras?
- Siguiente base: ¿cuántas ciudades siguen siendo seguras? ¿Las que lo eran en el paso anterior están a suficiente distancia de la nueva base?

Árboles de recubrimiento

¿Qué es un **árbol**?

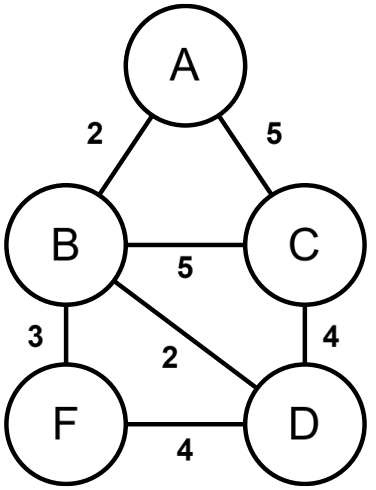
- Grafo **conexo** que **no tiene ciclos**

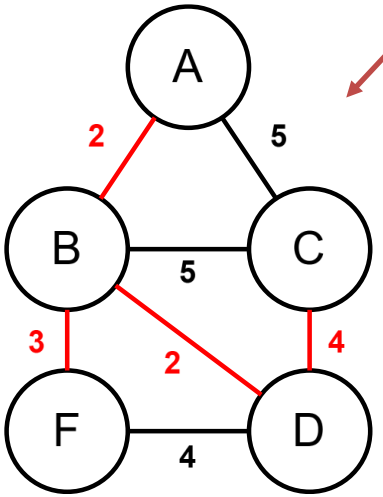
¿Qué es un **árbol de recubrimiento mínimo**?

- Subconjunto de aristas que:
  - Conecta todos los nodos
  - No forma ciclos
  - Menor peso total

Uso:

- Construcción de redes (carreteras, etc)
- Evitar ciclos





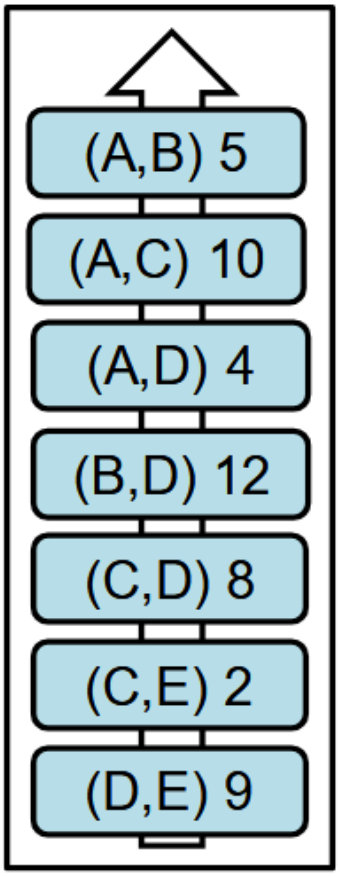
Árboles de recubrimiento - Kruskal

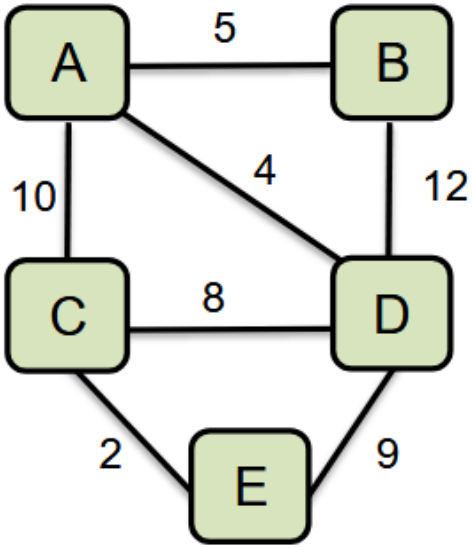
Algoritmo de Kruskal:

- Ordena las aristas de **menor** a mayor **peso**
- Empieza con cada **nodo** como una **componente separada**
- Va añadiendo **aristas** una a una SOLO si **conectan** dos **componentes distintas**
- El algoritmo termina cuando todos los nodos están conectados

Árboles de recubrimiento - Kruskal

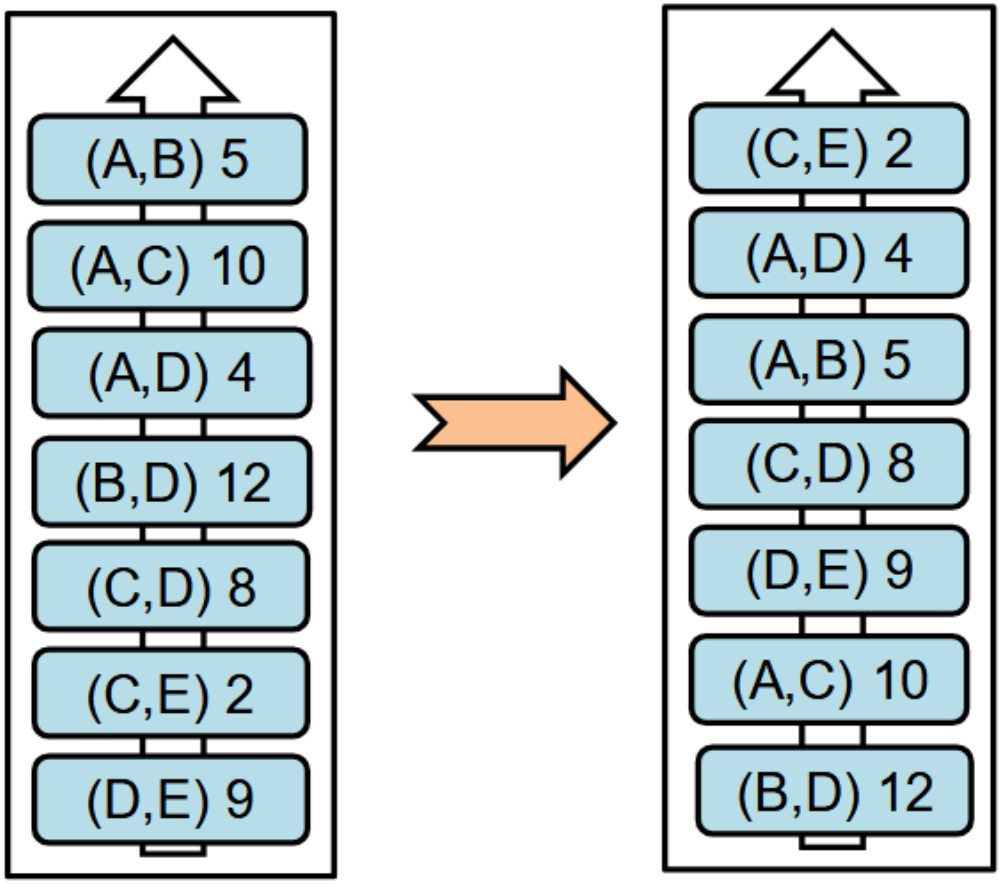
Valor: 0

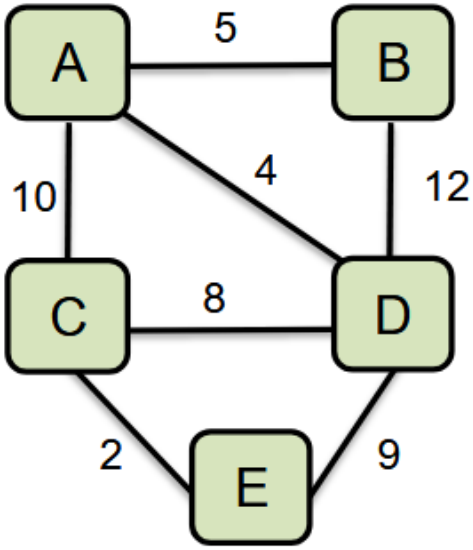




Árboles de recubrimiento - Kruskal

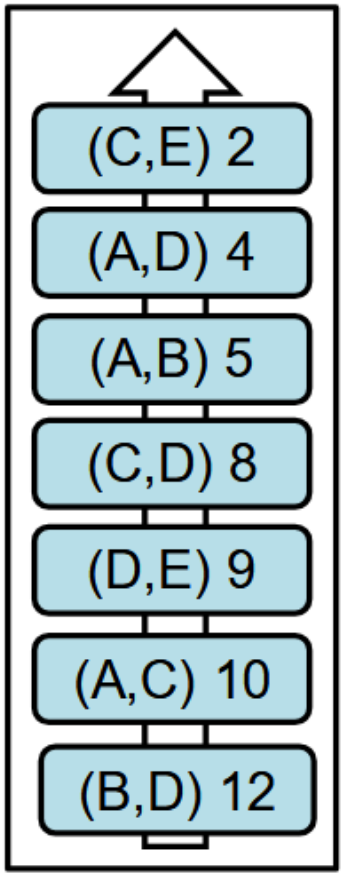
Valor: 0

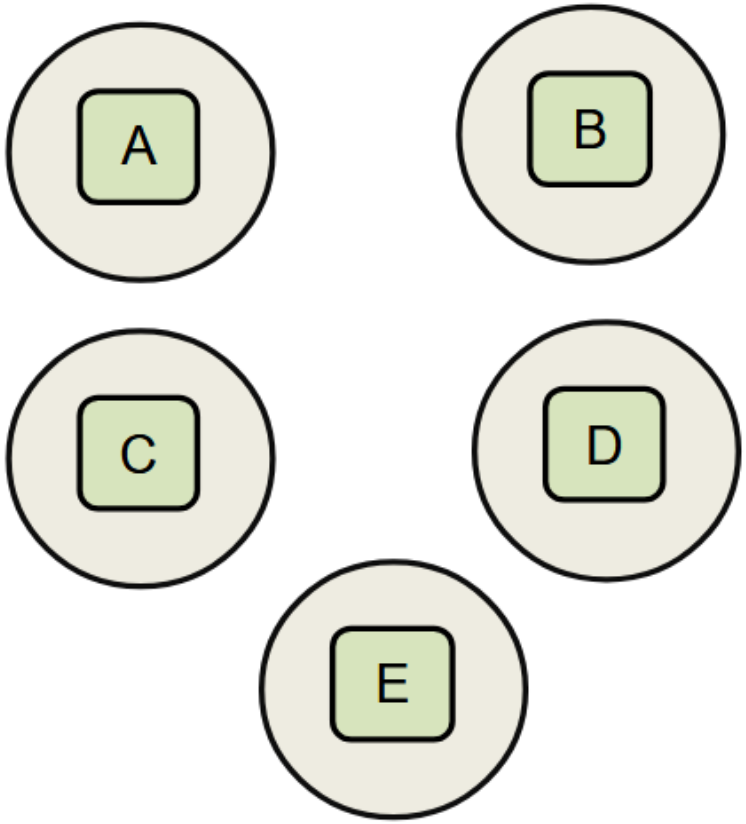


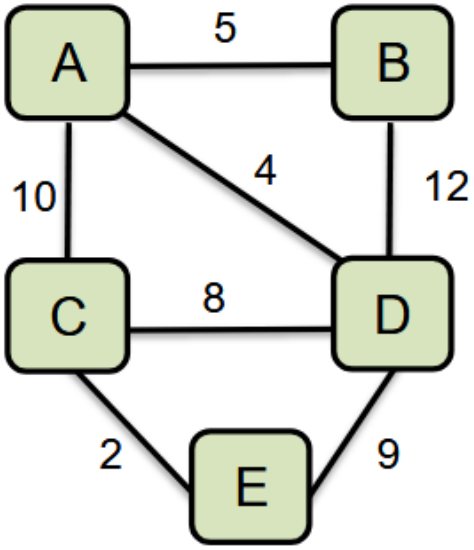


Árboles de recubrimiento - Kruskal

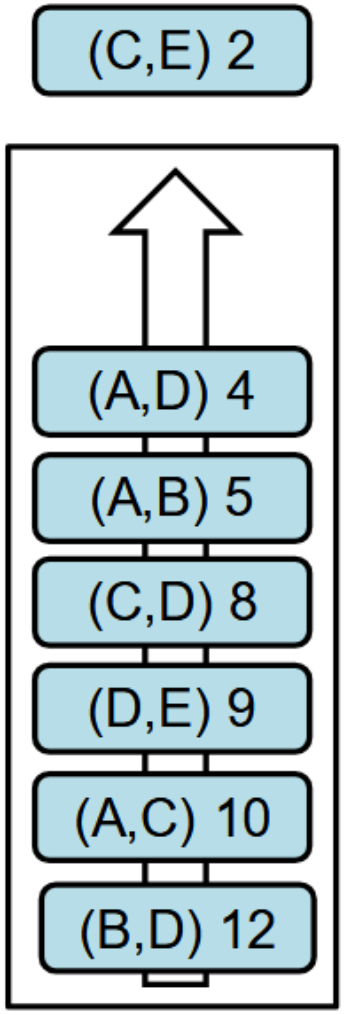
Valor: 0



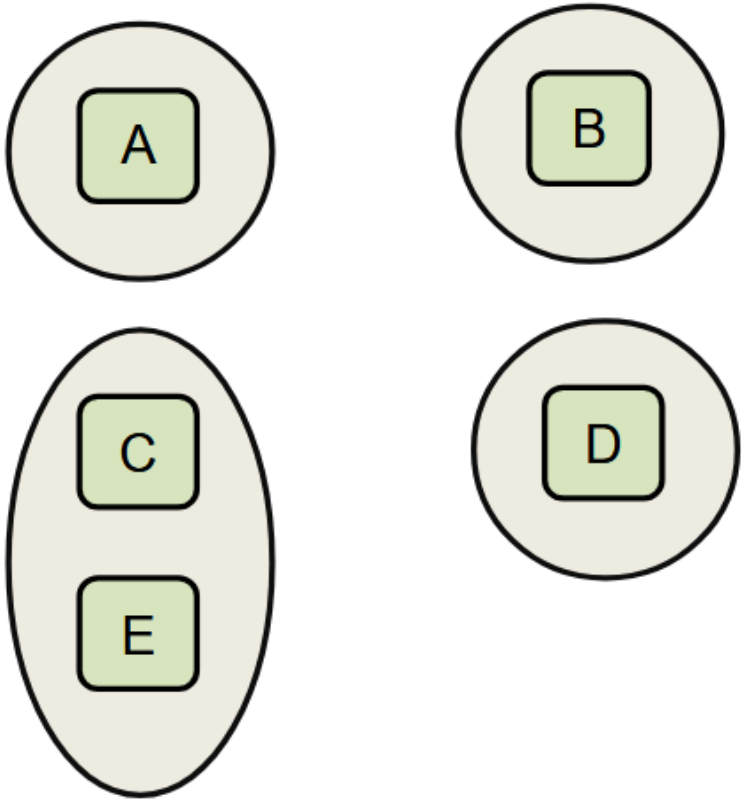


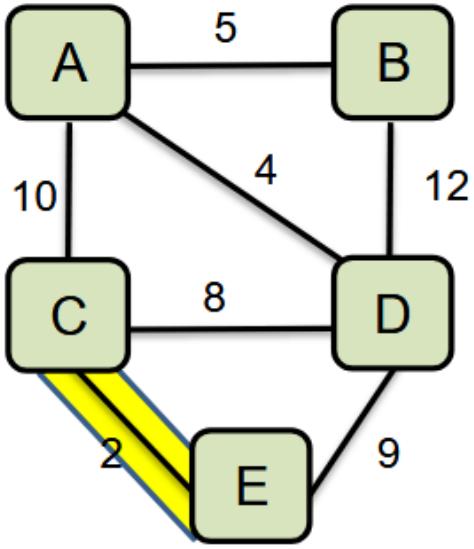


Árboles de recubrimiento - Kruskal

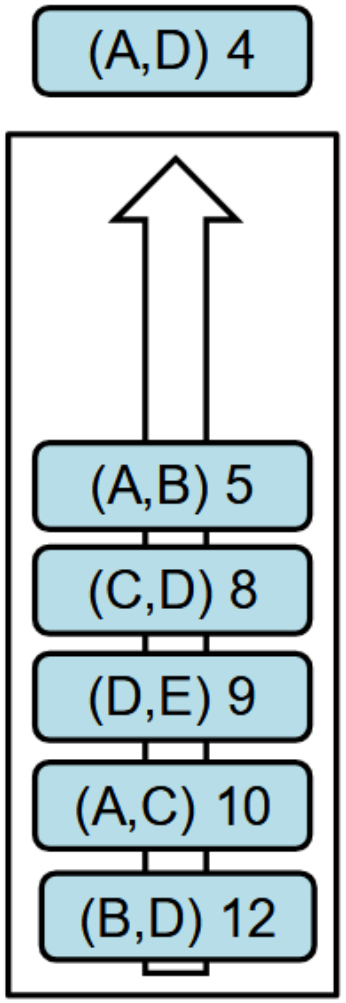


Valor: 2

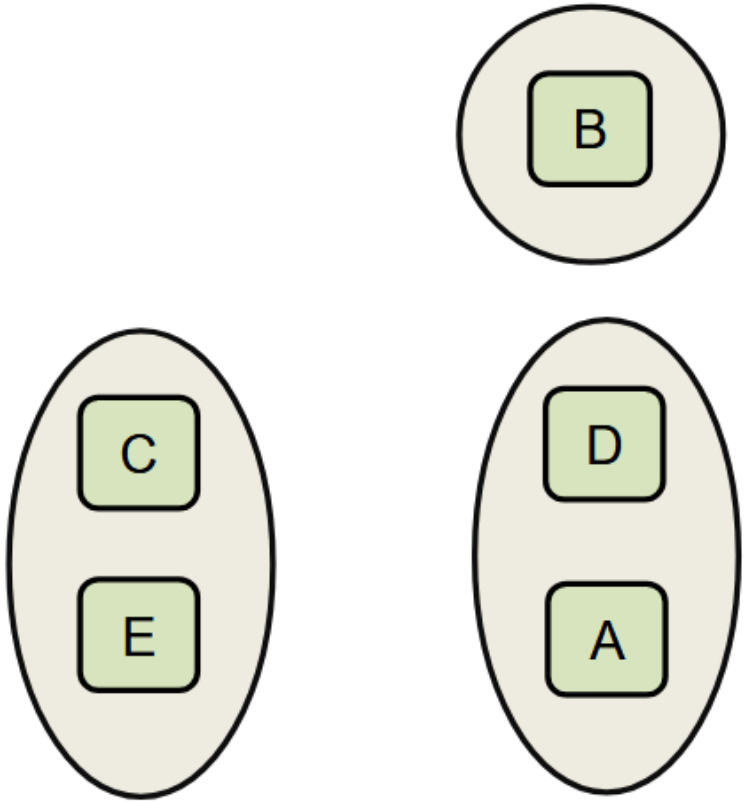


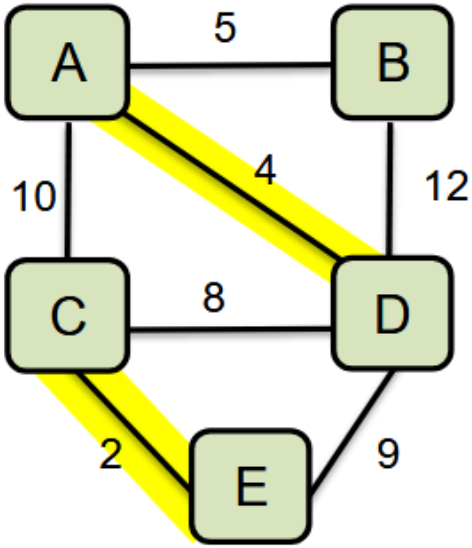


Árboles de recubrimiento - Kruskal

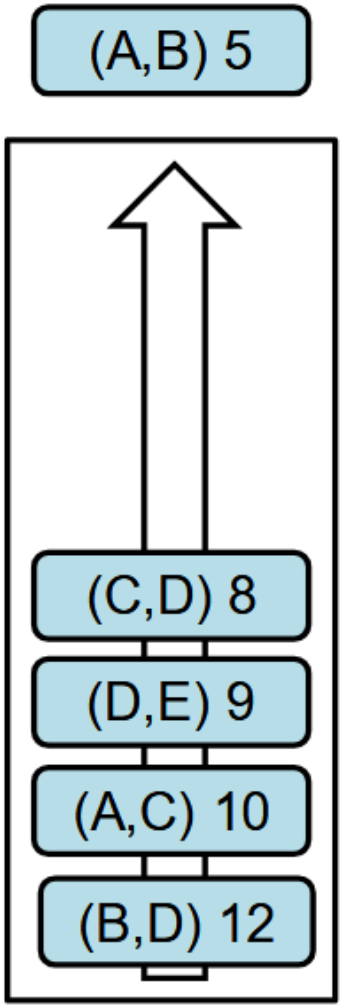


Valor: 6

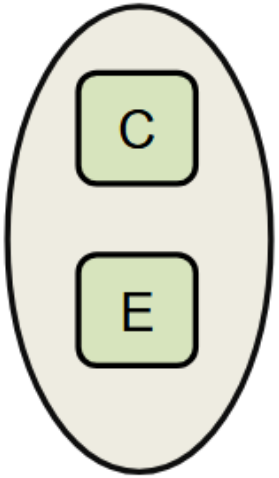


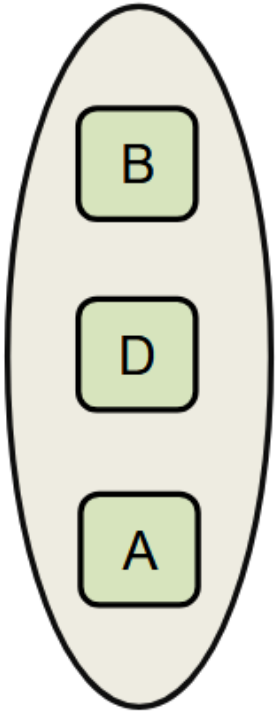


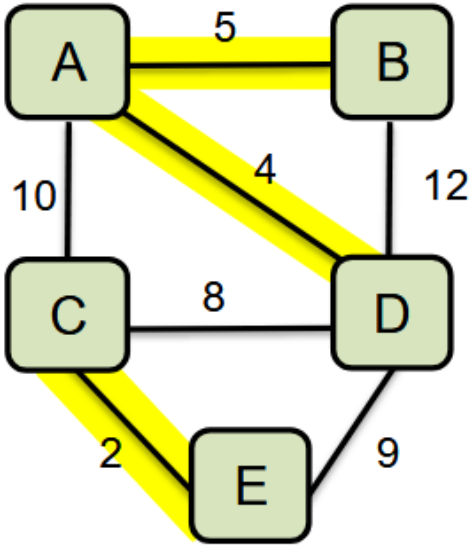
Árboles de recubrimiento - Kruskal



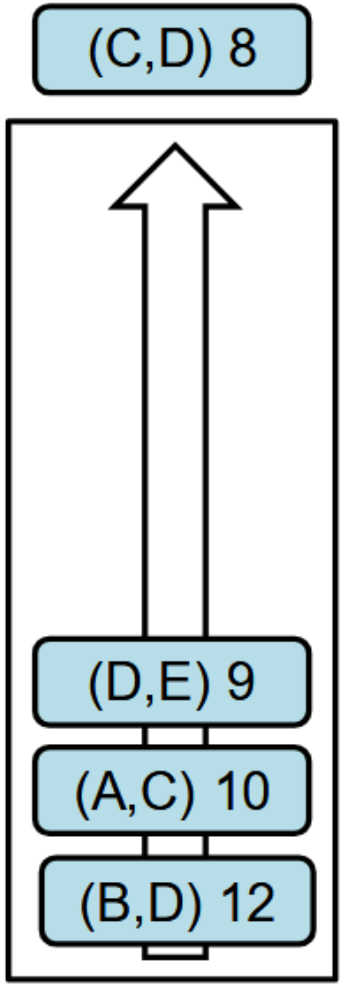
Valor: 11



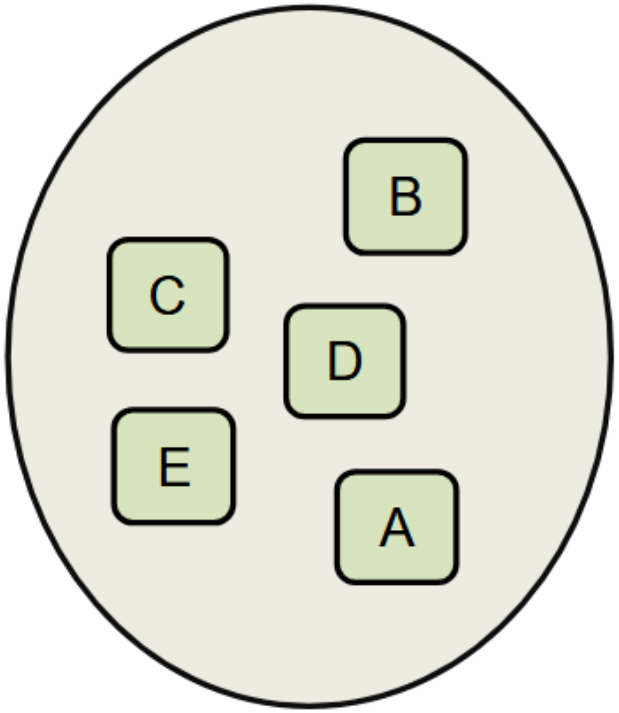


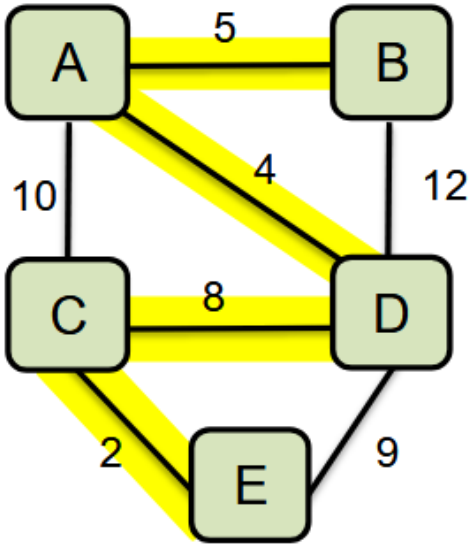


Árboles de recubrimiento - Kruskal



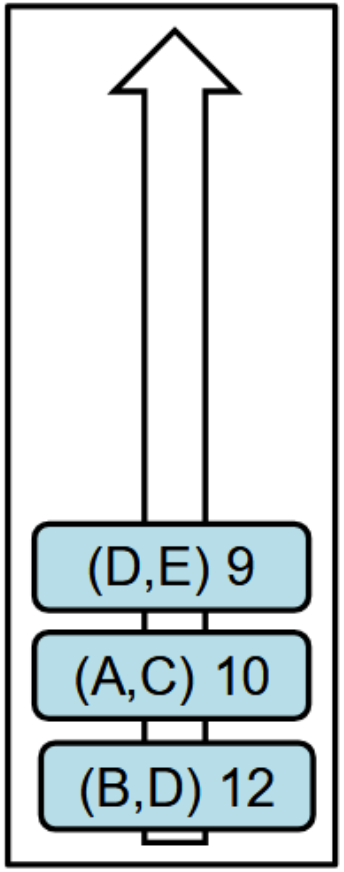
Valor: 19

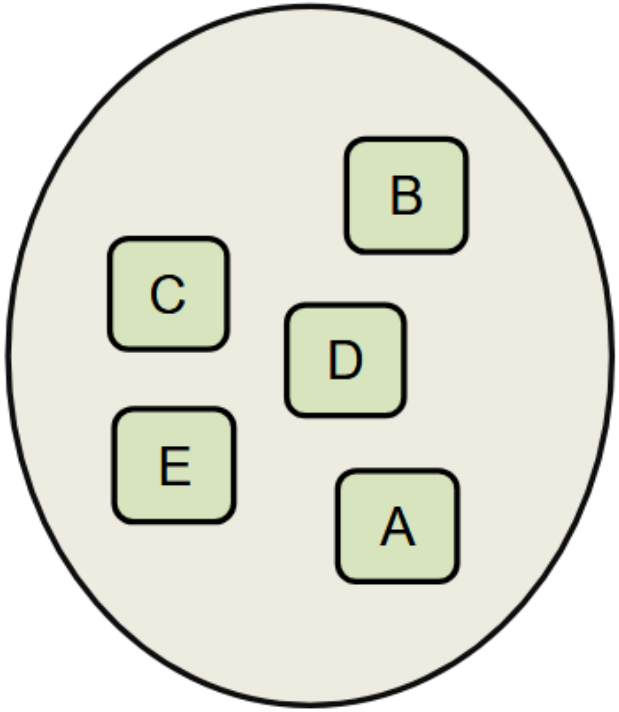


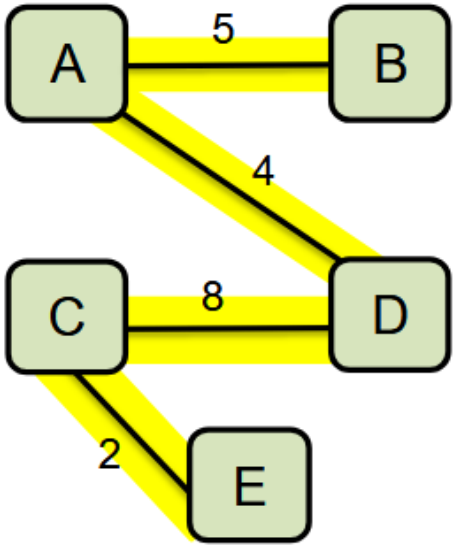


Árboles de recubrimiento - Kruskal

Valor: 19







Árboles de recubrimiento - Kruskal

```
def kruskal(num_nodes, edges):
    edges.sort(key=lambda x: x[2])  # By weight
    node_sets = [{i} for i in range(num_nodes)]
    mst = []
    cost = 0

    for u, v, weight in edges:
        index_u, index_v = -1, -1
        for i, node_set in enumerate(node_sets):
            if u in node_set: index _u = i
            if v in node_set: index _v = i
        if index _u != index _v:
            mst.append((u, v, weight))
            cost += weight
            node_sets[index_u].update(node_sets[index_v])
            node_sets.pop(index_v)

    return mst, cost
```

Árboles de recubrimiento – Kruskal (UF)

```python
def kruskal(num_nodes, edges):
    edges.sort(key=lambda x: x[2])
    dsu = DSU(num_nodes)
    mst = []
    cost = 0

    for u, v, weight in edges:
        if dsu.union(u, v):
            mst.append((u, v, weight))
            cost += weight

    return mst, cost
```

```python
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, i):
        if self.parent[i] == i: return i
        self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        root_i, root_j = self.find(i), self.find(j)
        if root_i != root_j:
            if self.rank[root_i] < self.rank[root_j]:
                self.parent[root_i] = root_j
            elif self.rank[root_i] > self.rank[root_j]:
                self.parent[root_j] = root_i
            else:
                self.parent[root_i] = root_j
                self.rank[root_j] += 1
            return True
        return False
```

Árboles de recubrimiento - Prim

Algoritmo de Prim:

- Empieza en un **nodo cualquiera** del grafo
- Construye el árbol **expandiéndose** desde ese nodo
- En cada paso añade la arista de **menor peso** que conecta el árbol con un nodo que aún no está en él
- El algoritmo termina cuando todos los nodos están conectados
- No vale para grafos no conexos!

Árboles de recubrimiento - Prim

```
import heapq
def prim(graph, start):
    queue, costs = [], [float('inf') for _ in range(len(graph))]
    visited, mst_cost = [False for _ in range(len(graph))], 0
    heapq.heappush(queue, (0, start))
    costs[start] = 0

    while queue:
        current_weight, u = heapq.heappop(queue)
        if visited[u]: continue

        visited[u] = True
        mst_cost += current_weight
        for v, weight in graph[u]:
            if not visited[v] and weight < costs[v]:
                costs[v] = weight
                heapq.heappush(queue, (weight, v))

    return mst_cost
```

¿Preguntas?

HASTA LA SEMANA QUE VIENE!

@URJC_CP

@Dijkstraidos