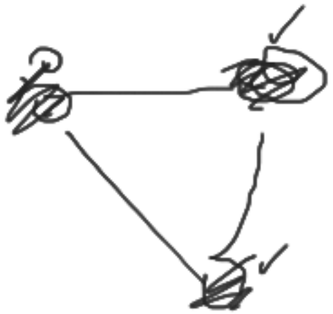
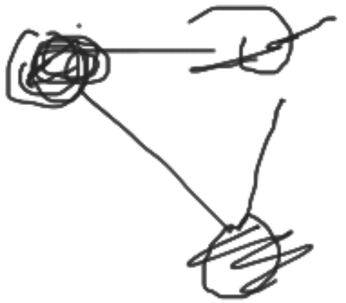
Ejercicio 1. Modificar el algoritmo DFS para que determine si el grafo de entrada es o no es conexo.

```
Explorar(v):
    visited[v] = true
    for u in adj[v]:
        if (visited[u]) continue
        Explorar(u)

componentes = 0
DFS(u):
    for v in V(G):
        if (!visited[v])
            ++componentes
            Explorar(v)
```





Pregunta 1

```
DFS(v);
if componentes > 1
    cout << no es conexo
```

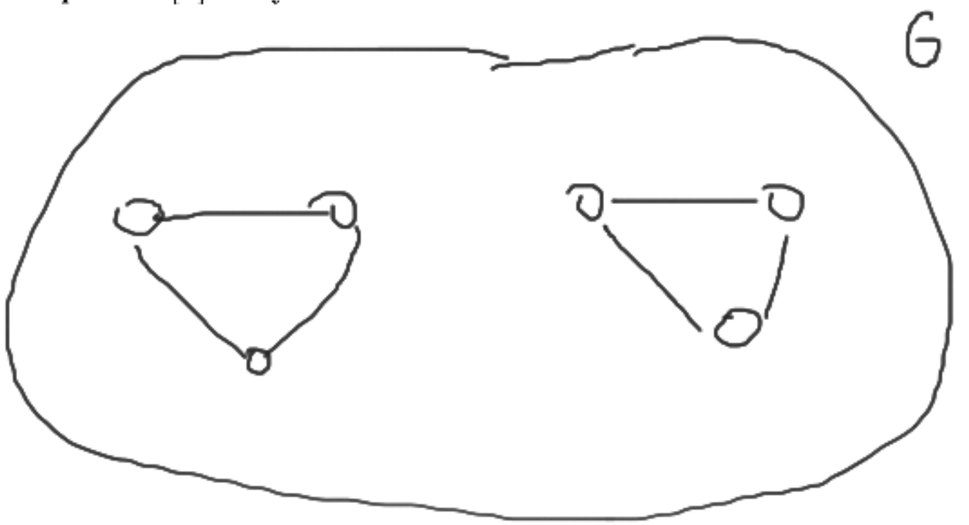
Pregunta 2

```
cout << componentes.
```

Ejercicio 2. Modificar el algoritmo DFS para que determine la cantidad de componentes del grafo.

ver pizarra anterior

Ejercicio 3. Modificar el algoritmo DFS para que llene un arreglo *componente*, indexado por los vértices del grafo, tal que *componente*$[v]$ = *componente*$[u]$ si u y v están en el mismo componente de G.




```
int componente[N];
Explorar(v):
    componente[v] = componentes
    visited[v] = true
    for u in adj[v]:
        if (visited[u]) continue
        Explorar(u)
componentes = 0
DFS(u):
    for v in V(G):
        if (!visited[v])
            ++componentes
            Explorar(v)
```

Ejercicio 5. Modificar el algoritmo BFS para que, reciba un grafo G y dos vértices s y t, y devuelva un camino mínimo entre s y t si existe. Si no existe camino debe mostrar un mensaje.

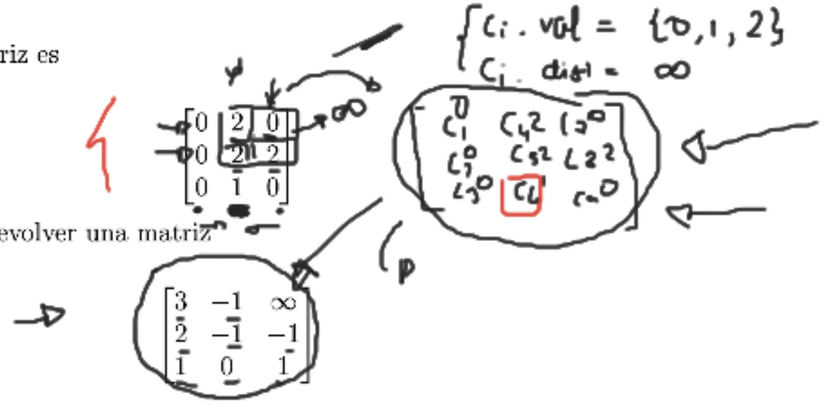

```
BFS(G, s, t):
// v: v.dist, v.p
// para cada v, v.dist = inf
s.p = null
s.dist = 0
Q = {}
enqueue(Q, S)
while Q != 0
  u = dequeue(Q)
  for w in adj[u]:
    if w.dist = inf:
      w.dist = u.dist + 1
      w.p = u
      enqueue(Q, w)
   if w = t: break;

if t.p = NULL
  if t != s
    cout << "no hay camino"
    return
  else
    return [s]
while t.p != null:
 camino.push_front(t)
 t = t.p
camino.push_front(t)
return camino
```

Ejercicio 8. Podemos representar un campo minado por una matriz de números enteros, donde el caracter 0 representa un lugar sin mina, el caracter 1 representa un lugar minado y el caracter 2 representa un lugar al cual no se puede acceder.

(a) Suponga que tenemos un campo minado con una única mina. Deseamos saber cuales son las posibilidades de poder toparse con dicha mina al moverse a partir de una posición (los movimientos no pueden ser diagonales) Haga un algoritmo que calcule cual es el menor numero de pasos que una persona debe moverse para toparse con dicha mina.

Por ejemplo, si la matriz es



su algoritmo deberá devolver una matriz

Siendo la matriz $n \times n$, su algoritmo deberá consumir tiempo $O(n^2)$.

(b)

0 = sin mina 1 = mina 2 = no hay acceso

```
MinMovimientosBuscaminas(&M, c):
  c.dist = 0
  Q = {}
  enqueue(Q, c)

  while (Q != 0):
    u = dequeue(Q)
    for celda in GetAdjacentCells(u):
      if celda.dist = inf:
        if celda.val != 2:
          celda.dist = u.dist + 1
          enqueue(Q, celda)
        else
          celda.dist = -1
  return BuildMatrix(M)
```

```
BFS(R, s):
 if s in R:
  dist[s] = 1
 else:
  dist[s] = 0
 enqueue(Q, s)
 while Q != 0:
  u = dequeue(Q)
  for w in adj[u]:
   if !visited:
    if w in R:
     dist[w] = dist[u] + 1
    else:
     dist[w] = dist[u]
    if dist[w] > k:
     continue
    if w is leaf:
     cumplen++
     continue
    else
     enqueue(Q, w)
```

Ejercicio 9. Dado un árbol T (grafo conexo sin circuitos, no dirigido y sin pesos en las aristas), un vértice s y un conjunto $R \subseteq V(T)$, decimos que una hoja v (vértice de grado uno diferente de s) de T es *k-atingible* desde s, si el (único) camino desde s hacia v contiene como máximo k vértices en R. Diseñe un algoritmo **eficiente** ($O(|V(G)|)$) que recibe un árbol G, un vértice s en G, un conjunto $R \subseteq V(G)$, un entero k y encuentra todas las hojas k-atingibles desde s en G.