1. Calcular la complejidad computacional del siguiente algoritmo

A

```
int sum = 0;
for (int n = N; n > 0; n /= 2)
  for (int i = 0; i < n; i++)
    sum++;
```

B

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
  for(int j = 0; j < i; j++)
    sum++;
```

2. Encuentre, si es posible, una solución para la siguiente ecuación de recurrencia:

A. **T(n) = 2 T(n/3) + n, T(1) = 1**

B. **T(n) = 2 T(n/2) + 1, T(1) = 1**

1A) Si armanos una tabla de iteraciones:

n	iteraciones i
N	N
N/2	N/2
N/4	N/4
N/8	N/8
.	.
.	.
.	.
1	1 -> redondeo

Es decir que el total de iteraciones es N+N/2+N/4+N/8+..., que es algo de la forma N(1+1/2+1/4+....), y por lo tanto converge a 2N, es O(N).

1B) Similar, solo que las iteraciones las podriamos ver como 1+2+4+8+...+N/8+N/4+N/2+N.

2A) A=2, B=3, f(n) = n,

Calculamos un factor C = log_b(a) = log3(2) < 1.

Como f(n) es mayor que n^C (n > n^C), T(n) = Θ(f(n)) = Θ(n)

2B) A=2, B=2, f(n) = 1,

Calculamos un factor C = log_b(a) = log2(2) = 1.

Como f(n) es menor que n^C (1 < n), T(n) = Θ(n^C) = Θ(n)

Calcular la complejidad computacional del siguiente algoritmo

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
  for (int j = 0; j < N; j++)
    sum++;
```

Encuentre, si es posible, una solución para la siguiente ecuación de recurrencia:
T(n) = 2 T(n/2) + n^2, T(1) = 1

1) En el primer for, por cada iteracion el valor i-esimo es multiplicado por dos
iteracion I valor de i

1	I	2
2	I	4
3	I	8
.	I	
.	I	
.	I	
n	I	2 a la x

2 a la x > n es decir que x > log2(n)

Por lo tanto la complejidad asintotica de dicho for sera O(log n)

El segundo se ve a simple viste que iterara n veces por lo que su complejidad asintotica es O(n)

Mientras que la linea "sum++;" es constante O(1)

Multiplicando cada complejidad por ser estos for anidados, obtenemos que
la complejidad total del bloque es 0(n*log n)

2) Recordando el Teorema Maestro tenemos que

T(n) = aT(n/b) + f(n)

Se debe cumplir que a >= 1 y b > 1

En este caso a = 2 >= 1
b = 2 > 1

Por lo tanto se cumple.

Luego observamos que n^logb(a) = n^log2(2) = n^1 = n

Esto cumple con el caso 3 del teorema maestro, donde f(n) es polinomicamente mayor que la expresion antes mencionada

Como n^2 es polinomicamente mayor que n:
Podemos decir que la expresion tendra complejidad Θ(f(n)) = Θ(n^2)

Defina una estructura para implementar una lista doblemente enlazada. Muestre la estructura e implemente las siguientes primitivas para la lista doblemente enlazada:

- lista_crear
- lista_eliminar_de_posicion

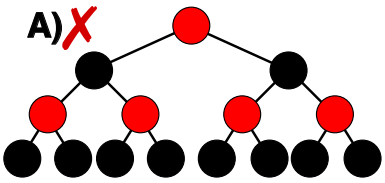
Para la lista doblemente enlazada cada nodo tendra dos campos que apuntaran uno al nodo siguiente y uno al nodo anterior

Podemos definir entonces

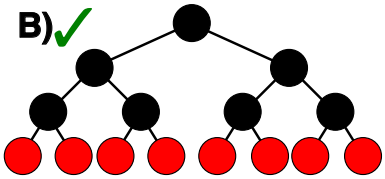
```c
// firma del destructor de elementos

typedef void(*lista_liberar_elemento)(void*);

// definicion de estructura nodo

typedef struct nodo{
    void* dato;
    struct nodo *siguiente;
    struct nodo *anterior;
} nodo_t;

// definicion de estructura lista

typedef struct lista{
    nodo_t* primero ;
    nodo_t* ultimo ;
    size_t cantidad;
    lista_liberar_elemento destructor;
}


/* Crea la lista reservando la memoria requerida
 * e inicializando campos necesarios.
 * Devuelve el puntero a la lista o NULL en caso de error
*/
lista_t* crear( lista_liberar_elemento destructor )
{
    lista_t* lista;

    if( (lista = (lista_t*) malloc(sizeof(lista_t))) == NULL )
        return NULL;

    lista->primero = NULL;
    lista->ultimo = NULL;
    lista->cantidad = 0;
    lista->destructor = destructor;

    return lista;
}

int  lista_eliminar_de_posicion(lista_t* lista, size_t posicion )
{
    if( !lista || lista->primero == NULL )
        return -1;

    nodo_t* aux_posicion = lista->primero;
    size_t i;
    // itero hasta llegar a la posicion o hasta llegar al final
    for( i=0; i<posicion && aux_posicion->siguiente; i++)
    {
        aux_posicion = aux_posicion->siguiente;
    }

    if(posicion == 0){
        lista->primero = aux_posicion->siguiente;
        if(aux_posicion->siguiente)
            aux_posicion->siguiente->anterior = NULL;
    }

    if( posicion >= lista_cantidad ){
        aux_posicion->anterior->siguiente = NULL;
        lista->ultimo = aux_posicion->anterior;
    }

    if(lista->destructor)
        lista->destructor(aux_posicion->dato);

    free(aux_posicion);

    lista->cantidad--;

    return 0;
}
```

¿Cuáles de los siguientes árboles rojo-negro son válidos para una determinada implementación? Justifique cada caso.

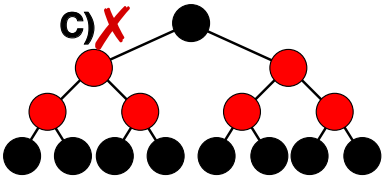
A) ✗



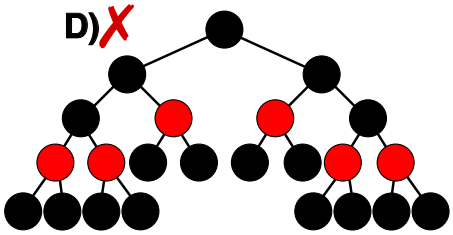
B) ✓



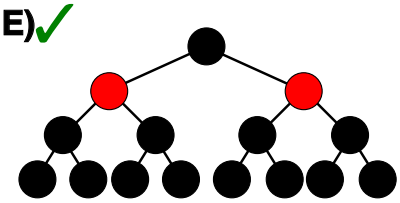
C) ✗



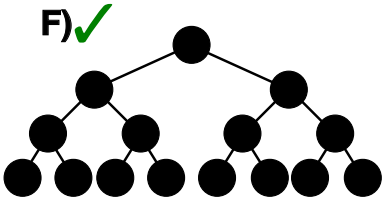
D) ✗



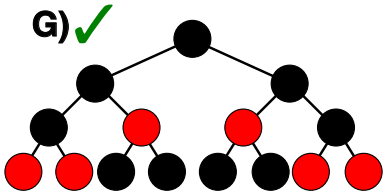
E) ✓



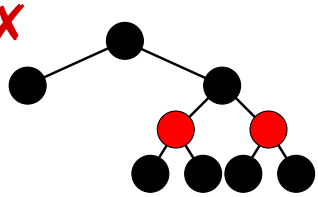
F) ✓



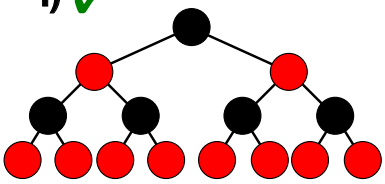
G) ✓



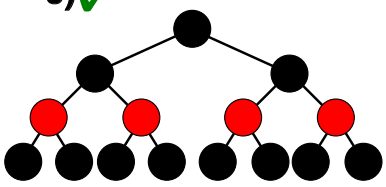
H) ✗



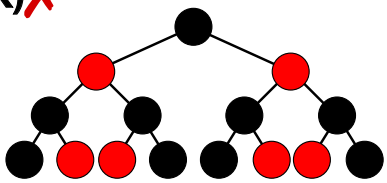
I) ✓



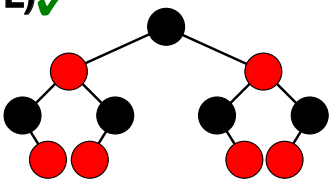
J) ✓



K) ✗



L) ✓



Requisitos para que un árbol sea árbol rojo y negro:

- Todo nodo es rojo o negro.
- La raíz es negra.
- Todas las hojas NULL son negras.
- Todo nodo rojo debe tener nodos hijos negros. No puede haber nodos rojos adyacentes.
- Cualquier camino desde un nodo dado hasta sus hojas, tiene la misma cantidad de nodos negros.

Considero en todos los casos que todas las hojas que apuntan a NULL son negras.

A. Este árbol no cumple con las propiedades del árbol rojo y negro, ya que la raíz es roja y debe ser negra.
B. Este árbol puede ser un árbol rojo y negro ya que cumple todas los requisitos para serlo.
C. Este árbol no puede ser un árbol rojo y negro ya que hay nodos rojos con hijos rojos.
D. Este árbol no puede ser un árbol rojo y negro ya que hay distinta altura negra en los caminos.
E. Este árbol puede ser un árbol rojo y negro ya que cumple todas los requisitos para serlo.
F. Este árbol puede ser un árbol rojo y negro, aunque no es óptimo ya que no hay nodos rojos. Y no se cómo se insertaron los nodos, para que queden todos negros al final.
G. Este árbol puede ser un árbol rojo y negro ya que cumple todas los requisitos para serlo.
H. Este árbol no puede ser un árbol rojo y negro porque no todos los caminos tienen la misma cantidad de nodos negros (desde la raíz hay uno que tiene un nodo más que el resto, por lo que tiene distinta altura negra).
I. Este árbol puede ser un árbol rojo y negro ya que cumple todas los requisitos para serlo.
J. Este árbol puede ser un árbol rojo y negro ya que cumple todas los requisitos para serlo.
K. Este árbol no puede ser un árbol rojo y negro ya que cualquier camino dado desde un nodo a sus hojas descendientes debería contener el mismo número de nodos negros y no es así.
L. Este árbol puede ser un árbol rojo y negro ya que cumple todas los requisitos para serlo.

OBS: - Los punteros NULL de un árbol rojo/negro no suelen dibujarse y se consideran como nodos negros.

Justifique si el siguiente vector de enteros representa un heap (maximal o minimal). ¿Qué condición debe cumplir para que lo sea?

5	8	6	2	4	3

Explique (texto o código claro) cómo reordenarlo (inplace, sin requerir reservar memoria extra) para que cumpla la condición. El algoritmo descripto debe funcionar para cualquier vector de enteros.
Aplique el algoritmo al vector dado para obtener un **heap minimal** explicando cada iteración del mismo. En cada paso de algoritmo muestre como queda el array (marque con un asterisco los elementos que se movieron en el array en ese paso).

El vector de enteros dado no representa un heap maximal ni minimal.

Un heap es un árbol binario que tiene la menor altura posible, y es casi completo.
Para que un heap sea maximal el padre de un nodo debe ser mayor o igual a sus hijos, y para que un heap sea minimal el padre de un nodo debe ser menor o igual a sus hijos.

En este caso no se cumple ninguna de estas condiciones uniformemente para todos los nodos. Hay nodos que son mayores que sus hijos, y hay nodos que son menores que sus hijos.

Para reordenar el vector y que cumpla la condición de heap, se utiliza la función heapify.

La idea del heapify es empezar desde el último nivel completo del árbol, y hacer sift down. Se reordena en el vector para que quede ordenado como un heap, en el caso de heap minimal, todos los nodos que sean padres deben ser menores o iguales que sus hijos. Se termina cuando no se puede intercambiar más nodos.

Diagrama de heapify minimal para el vector del enunciado:

Vector: 5 8 6 2 4 3

Aplico siftdown en 3, no hay movimiento ya que no tiene hijos (lo mismo sucede con 4 y 2):

Vector: 5 8 6 2 4 3

Aplico siftdown en 6, es mayor que su único hijo 3 aplico swap.

Vector: 5 8 3 2 4 6

Aplico siftdown en 8 es mayor a ambos de sus hijos, swap con 2 (el menor).

Vector: 5 2 3 8 4 6

Aplico siftdown en 5, es mayor a ambos de sus hijos swap con 2 (el menor).

Vector: 2 5 3 8 4 6

Luego, 5 es mayor que uno de sus dos hijos nuevos, swap con 4, (el menor).

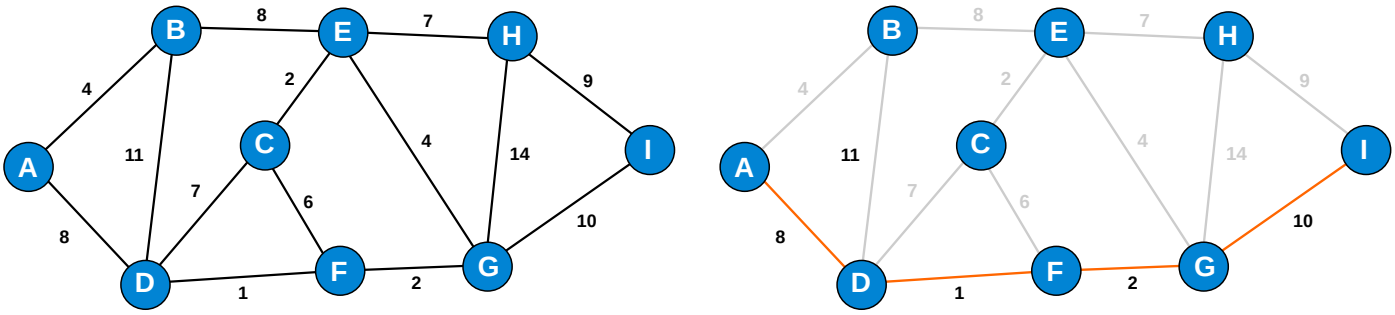
Vector: 2 4 3 8 5 6

2 es menor que 4 y 3
4 es menor que 8 y 5
3 es menor que 6

El resto no tiene hijos, cumple condición de heap.

OBS: - Se puede empezar el algoritmo desde el último nivel con hijos y ahorrarse el último.
- Para un nodo N (posición del vector), su hijo izquierdo es 2*N y su hijo derecho 2*N+1

Aplique el algoritmo de Dijkstra en el siguiente grafo (de A hacia I). Explique cada paso del algoritmo.



- Se visita el vértice NO VISITADO con menor distancia conocida desde el primer vértice, que es A, ya que la distancia de A a A es 0 y las demás infinito.
- Se calcula la distancia entre los vertices adyacentes sumando los pesos de cada uno con la distancia de A.
- Si la distancia calculada de los vértices conocidos es menor a la que está en la tabla se actualiza y tambien los vértices desde donde se llegó.
- Se marca el Vertice A como visitado.
- Se continua con el vértice no visitado con menor distancia....
- Repetir

En lo que sigue, + representa el nodo actual y * representa un nodo ya visitado:

Nodo	A+	B	C	D	E	F	G	H	I
Distancia	0	4	∞	8	∞	∞	∞	∞	∞
Anterior	A	A	-	A	-	-	-	-	-

Nodo	A*	B+	C	D	E	F	G	H	I
Distancia	0	4	∞	8	12	∞	∞	∞	∞
Anterior	A	A	-	A	B	-	-	-	-

Nodo	A*	B*	C	D+	E	F	G	H	I
Distancia	0	4	15	8	12	9	∞	∞	∞
Anterior	A	A	D	A	B	D	-	-	-

Nodo	A*	B*	C	D*	E	F+	G	H	I
Distancia	0	4	15	8	12	9	11	∞	∞
Anterior	A	A	D	A	B	D	F	-	-

Nodo	A*	B*	C	D*	E	F*	G+	H	I
Distancia	0	4	15	8	12	9	11	25	21
Anterior	A	A	D	A	B	D	F	G	G

Nodo	A*	B*	C	D*	E+	F*	G*	H	I
Distancia	0	4	14	8	12	9	11	19	21
Anterior	A	A	E	A	B	D	F	E	G

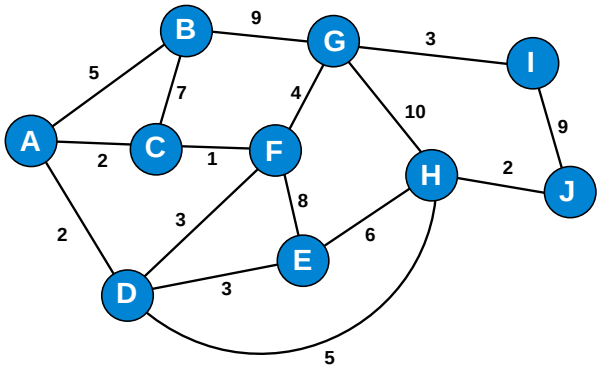
Nodo	A*	B*	C+	D*	E*	F*	G*	H	I
Distancia	0	4	14	8	12	9	11	19	21
Anterior	A	A	E	A	B	D	F	E	G

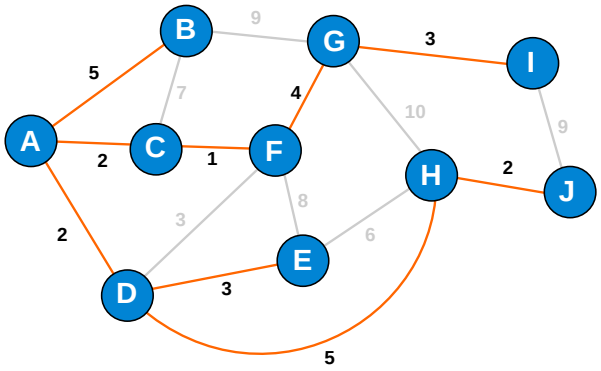
Nodo	A*	B*	C*	D*	E*	F*	G*	H+	I
Distancia	0	4	14	8	12	9	11	19	21
Anterior	A	A	E	A	B	D	F	E	G

Nodo	A*	B*	C*	D*	E*	F*	G*	H*	I*
Distancia	0	4	14	8	12	9	11	19	21
Anterior	A	A	E	A	B	D	F	E	G

Recorriendo la tabla empezando por I, vemos que el camino es I-G-F-D-A, con peso 21.

Aplique el algoritmo de Prim en el siguiente grafo. Explique cada paso del algoritmo.





El algoritmo Prim es un algoritmo greedy que permite encontrar el mínimo spanning tree.
Un árbol de expansión mínimo es aquel que pesa menos o igual que otros árboles de expansión.

El algoritmo Prim tiene los siguientes pasos:

1. Se consideran todos los nodos fuera del árbol
2. Arbitrariamente se elige un nodo del árbol
3. Se agrega al árbol el nodo adyacente con el mínimo peso.
4. Se repite 3. hasta que todos los nodos hayan sido visitados.

En el caso del grafo del enunciado:

Elijo arbitrariamente comenzar por B:

V = [B] , NV = [A,C,D,E,F,G,H,I,J]

El nodo adyacente al visitado B con mínimo peso es A, B_A = 5:

V = [B,A] , NV = [C,D,E,F,G,H,I,J]

Uno de los nodos adyacente a los visitados con mínimo peso es D, A_D = 2: (se podría haber elegido C también)

V = [B,A,D] , NV = [C,E,F,G,H,I,J]

El nodo adyacente a los visitados con mínimo peso es C, A_C = 2:

V = [B,A,D,C] , NV = [E,F,G,H,I,J]

El nodo adyacente a los visitados con mínimo peso es F, C_F = 1:

V = [B,A,D,C,F] , NV = [E,G,H,I,J]

El nodo adyacente a los visitados con mínimo peso es E, D_E = 3:

V = [B,A,D,C,F,E] , NV = [G,H,I,J]

El nodo adyacente a los nodos visitados de mínimo peso es G, F_G = 4:

V = [B,A,D,C,F,E,G] , NV = [H,I,J]

El nodo adyacente a los nodos visitados de mínimo peso es I, G_I = 3:

V = [B,A,D,C,F,E,G,I] , NV = [H,J]

El nodo adyacente a los nodos visitados de mínimo peso es H, D_H = 5:

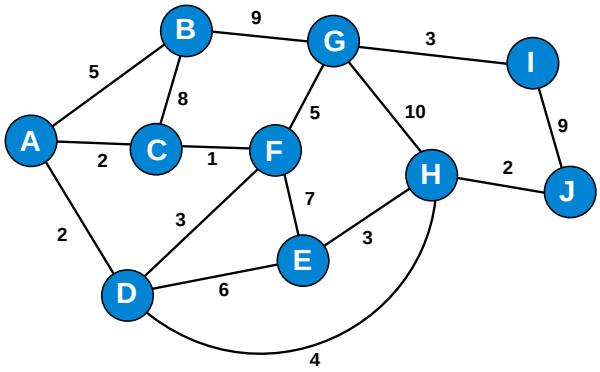
V = [B,A,D,C,F,E,G,I,H] , NV = [J]

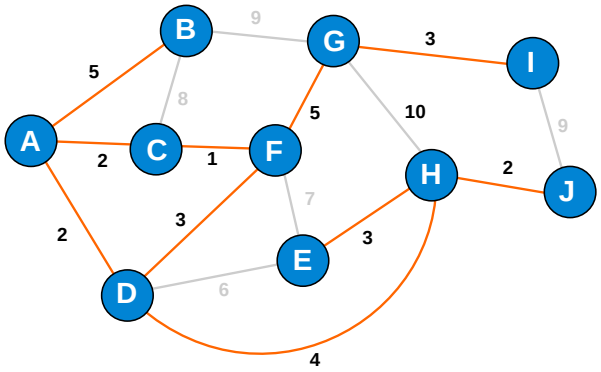
El último nodo adyacente a los visitados con mínimo peso es J, H_J = 2:

V = [B,A,D,C,F,E,G,I,H,J] , NV = [].

Y así finaliza el algoritmo, siendo el minimo spanning tree el unido por las aristas B_A, A_D, A_C, C_F, D_E, F_G, G_I, D_H, H_J.

Aplique el algoritmo de Kruskal en el siguiente grafo. Explique cada paso del algoritmo.





El algoritmo Kruskal tiene como finalidad encontrar el arbol de expansion minimo en un grafo conexo y ponderado.

Para esto primero creo un conjunto de arboles con cada nodo
{A,B,C,D,E,F,G,H,I,J}

Comienzo con la arista 1 que es la de menor peso, como se crea un nuevo arbol, uno ambos nodos y lo agrego a mi conjunto
{A,B,C-F,D,E,G,H,I,J}

Sigo con la arista 2 que une A-C
{A-C-F,B,D,E,G,H,I,J}

sigo con la arista 2 que une A-D
{D-A-C-F,B,E,G,H,I,J}

Sigo con la arista 2 que une H-J
{H-J,D-A-C-F,B,E,G,I}

Sigo con la arista 3 que une E-H
{D-A-C-F,E-H-J,B,G,I}

Sigo con la arisa 3 que une D-F
Pero esta no crea un nuevo arbol, asi que la descarto

Sigo con la arista 3 que une G-I
{G-I,D-A-C-F,E-H-J,B}

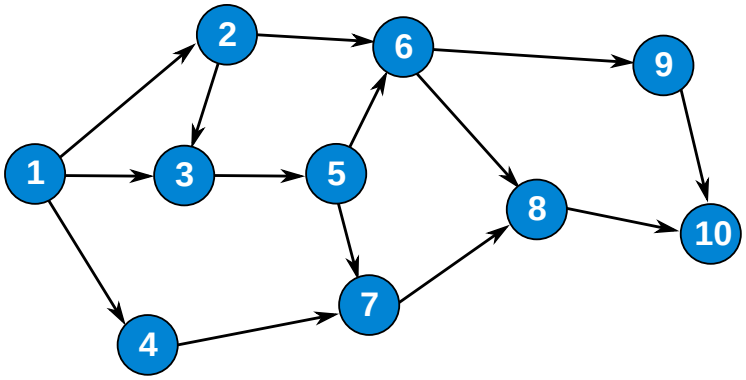
Sigo con la arista 4 que une H-D
{D-A-C-F-H-E-J,G-I,B}

Sigo con la arista 5 que une F-G
{D-A-C-F-H-E-J-G-I,B}

Sigo con la arista 5 que une A-B
{D-A-B-C-F-H-E-J-G-I}

De esta manera cada nodo quedo conectado, las demas aristas se desprecian y obtuvimos el arbol de expansion minimo.

Aplique el algoritmo de ordenamiento topológico en el siguiente grafo. Explique cada paso del algoritmo.



Existen diferentes algoritmos para hallar el órden topológico (e incluso puede haber diferentes resultados), vamos a utilizar el siguiente algoritmo visto en clase:

- Tomamos como inicial una cola con los vértices con grado de incidencia 0 (sin aristas entrantes).

Candidatos: [1]

Removemos un vértice de la cola (ponemos el mismo en la lista ordenada) y eliminamos del grafo las aristas salientes del mismo. Insertamos a la cola los nuevos vértices sin entradas.

Ordenado: [1]
Candidatos: [2,4]

Repetimos hasta quedarnos sin vértices candidatos. Si al finalizar quedan vértices sin recorrer del grafo, detectamos un ciclo en el mismo.

Ordenado: [1,2]
Candidatos: [4,3]

Ordenado: [1,2,4]
Candidatos: [3]

Ordenado: [1,2,4,3]
Candidatos: [5]

Ordenados: [1,2,4,3,5]
Candidatos: [6,7]

Ordenados: [1,2,4,3,5,6]
Candidatos: [7, 9]

Ordenados: [1,2,4,3,5,6,7]
Candidatos: [9, 8]

Ordenados: [1,2,4,3,5,6,7,9]
Candidatos: [8]

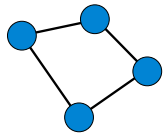
Ordenados: [1,2,4,3,5,6,7,9,8]
Candidatos: [10]

Ordenados: [1,2,4,3,5,6,7,9,8,10]
Candidatos: []

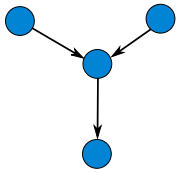
Ya se recorrieron todos los nodos, por lo tanto el órden final es [1,2,4,3,5,6,7,9,8,10]

Clasifique los siguentes grafos en: Dirigido / no dirigido, simple / no simple, completo / incompleto, cíclico / acíclico, conexo / fuertemente conexo / débilmente conexo / no conexo, árbol. Justifique.

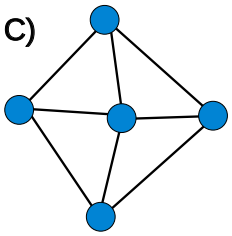
A)



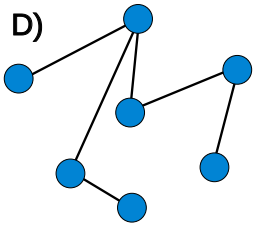
B)



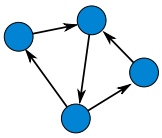
C)



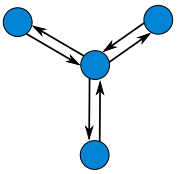
D)



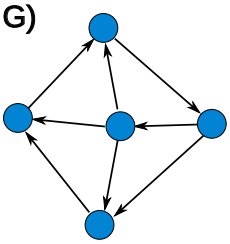
E)



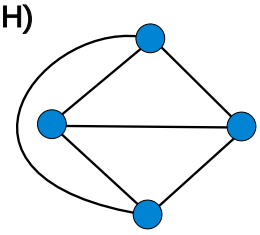
F)



G)



H)



Grafo dirigido: las aristas tienen dirección.
Grafo no dirigido: las aristas no tienen dirección.

Grafo simple: un grafo es simple cuando no posee bucles ni aristas paralelas.
Grafo no-simple: no es un grafo simple.

Grafo completo: todos los vértices tienen aristas que los unen con todos los demás.
Grafo incompleto: no es completo.

Grafo cíclico: contiene un camino entre al menos dos aristas que comienza y termina en el mismo vértice.
Grafo acíclico: no es cíclico.

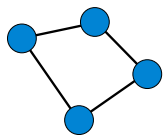
Grafo conexo: Un grafo es conexo si para cualquier par de vértices existe al menos un camino entre ellos.
Grafo fuertemente conexo: Es un grafo dirigido en el que para cada par de vértices existe un camino de A a B y de B a A.

Grafo debilmente conexo: (Grafo dirigido) Es conexo y se debería cambiar una o más aristas por aristas sin sentido para que pueda ser fuertemente conexo.
Grafo no conexo: Hay uno o más vértices aislados que no están conectados por ninguna arista.

Árbol: un grafo es un árbol si es conexo y acíclico.

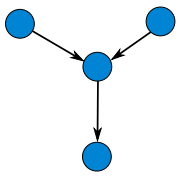
A continuación se indica si los siguientes cumplen con las definiciones anteriores:

A)



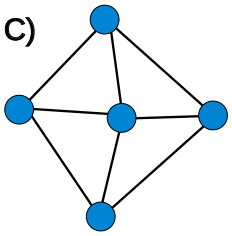
No dirigido, Simple Incompleto, Cíclico, Conexo

B)



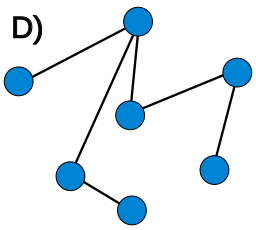
Dirigido, Simple Incompleto, Acíclico, Débilmente conexo

C)



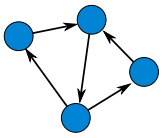
No dirigido, Simple Incompleto, Cíclico, Conexo

D)



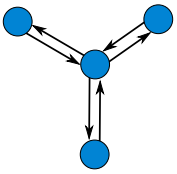
No dirigido, Simple Incompleto, Acíclico, Conexo, Árbol

E)



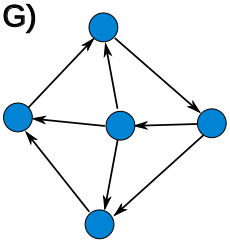
Dirigido, Simple Incompleto, Cíclico, Fuertemente conexo

F)



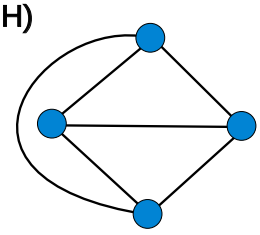
Dirigido, Simple Incompleto, Cíclico, Fuertemente conexo

G)



Dirigido, Simple Incompleto, Cíclico, Fuertemente conexo

H)



No dirigido, Simple Completo, Cíclico, conexo