Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Curso: Algoritmos - Tareas
Estudiante: John Alejandro Pastor Sandoval

TAREA 1:

```python
import math

def actualizar_papa(papa_act, cred_act, nota_alg, cred_alg):

    total_pactual = papa_act * cred_act
    total_pnuevo = total_pactual + (nota_alg * cred_alg)
    cred_total = cred_act + cred_alg
    nuevo_papa = total_pnuevo / cred_total
    influencia = nuevo_papa - papa_act
    return nuevo_papa, influencia

papa_actual = float(input("Papa Actual: ")  )        # PAPA actual
creditos_actuales = int(input("Creditos acumlados hasta este semestre:
"))      # Total de créditos acumulados hasta el momento
creditos_algoritmos = 3      # Créditos de la materia Algoritmos

# Generamos la lista de resultados para notas de 0 a 5 (incrementos de
0.5)
resultados = []
for nota_alg in [i * 0.5 for i in range(0, 11)]:
    nuevo_papa, influencia = actualizar_papa(papa_actual,
creditos_actuales, nota_alg, creditos_algoritmos)
    resultados.append((nota_alg, nuevo_papa, influencia))

# Mostrar los resultados
print(resultados)
print("Nota Algoritmos | Nuevo PAPA | Influencia")
for nota_alg, nuevo_papa, influencia in resultados:
    print(f"{nota_alg:>14.1f} | {nuevo_papa:>10.2f} |
{influencia:>10.2f}")
```

```
Papa Actual: 3.8
Creditos acumlados hasta este semestre: 100
[(0.0, 3.6893203883495147, -0.11067961165048512), (
Nota Algoritmos | Nuevo PAPA | Influencia
            0.0 |       3.69 |      -0.11
            0.5 |       3.70 |      -0.10
            1.0 |       3.72 |      -0.08
            1.5 |       3.73 |      -0.07
            2.0 |       3.75 |      -0.05
            2.5 |       3.76 |      -0.04
            3.0 |       3.78 |      -0.02
            3.5 |       3.79 |      -0.01
            4.0 |       3.81 |       0.01
            4.5 |       3.82 |       0.02
            5.0 |       3.83 |       0.03
```

Calculo probabilidades:

```python
def prob_final(n,c,p_min):
    for i in range(n+1):
        p_indiv = (1 - (1-(i/50))**c)
        p_total = p_indiv**n
        if p_total >= p_min:
            return i,round(p_total,2)
print(prob_final(50,32,0.85))
```

```
(9, 0.92)
```

Tarea 2:

Estructura	¿Para qué sirve? (Definición)	¿Cuándo se debe usar?	Operaciones
Arreglo estático (int [],double []..)	Para almacenar elementos del mismo tipo de manera continua.	Cuando se debe guardar una lista de elementos. Modificar y acceder a elementos en posiciones específicas.	Definir una posición (a[i] = x)
Arreglo dinámico (Vector)	Podemos almacenar elementos de manera secuencial, creciendo automáticamente.	no se sabe cuantos elementos habrá. Se accede a los elementos por índices. No quieres tener problemas con el tamaño para añadir o borrar	append,arr[i],insert,remove,pop

Queue	Almacenar datos de forma temporal y recuperarlos en un orden inverso(FIFO)	Simular cómo funciona una cola real Memoria eficiente	enqueue,deque, peek,isEmpty
Stack	Almacenar datos de forma temporal y recuperarlos en un orden de llegada(LIFO)	para evaluar expresiones matemáticas Memoria eficiente Llamadas a funciones (pila de ejecución)	push,pop,peek,isFull,isEmpty
Deque	Una cola doble que permite insertar y eliminar al principio y al final	Operaciones rápidas en el principio y final Darle prioridad a una tarea por sí urge	append,appendleft,pop,popleft,peek
Diccionario como HashTable (HashMap)	Asocia claves únicas a valores sin un orden específico.	Se usa cuando se necesita acceder rápido a valores usando una clave	insert,search,delete
Diccionario como árbol (Map)	Un diccionario pero este si esta ordenado por claves, basado en un árbol balanceado	Recorre las claves en orden para llegar a un valor	insert,search,delete
Set como HashTable (HashSet)	Un conjunto sin orden donde los elementos son únicos	Se usa para saber si elementos existen o no	add,remove,union
Set como árbol (Set)	Un conjunto que sí está ordenado de elementos únicos usando un árbol.	Elementos únicos ordenados	add,remove,union
Cola de prioridad o heap (priority_queue)	Una cola que siempre accede al elemento con prioridad	Se usa si se necesita una cola con tareas urgentes imprevistas	insert,delete,peek,isEmpty

Tarea 3.

Solución en palabras:

Para solucionar el problema principalmente usamos una bandera o puntero que se va moviendo en el string meow, así comparamos letra a letra con el string *S*. En un primer paso comprobamos si el string *S* comienza con una *'m'* y termina con *'w'*, si el string cumple con esto, se procede a realizar un for que primeramente revisa si el String contiene una letra

diferente a las de *'meow'*, si no contiene nada diferente, procede a funcionar el bucle donde comparamos los índices del String *S* , en el momento en que la primera letra *m* cambie a una diferente se actualizará el flag, empezando a comparar los caracteres con la segunda letra de *'meow'* y así sucesivamente hasta encontrar algún error en la concatenación de las letras, si no se encuentra ningún error se imprime 'YES' en caso contrario se imprime 'NO'

Código:

```
def is_meow(s):
    s = s.lower()
    meow = ['m', 'e', 'o', 'w']
    flag = 0  # flag to know which letter is comparing

    if not s or s[0] != meow[0] or s[len(s)-1] != meow[3]:
        return "NO"  # Must start with 'm' and end with 'w'

    for i in range(1, len(s)):
        if s[i] not in 'meow':
            return "NO"  # No works a different char from meow

        if s[i] != s[i-1]:
            flag += 1
            if flag >= 4 or s[i] != meow[flag]:
                return "NO"

    if flag == 3: return "YES"  # Must have read all four letters
```

Casos de prueba:

```
# Examples:
print(is_meow("meow"))  # YES
print(is_meow("mMeOooOw"))  # YES
print(is_meow("Mweo"))     # NO
print(is_meow("meowmeow")) # NO
print(is_meow("MEEEOOOOOOOOooWw"))  # YES
print(is_meow("MEEeeOOo"))     # NO
print(is_meow("meoww")) # YES
```

Tarea 4.

Código 1:

```
// Suma de un arreglo

int n;            //Tiempo,memoria = O(1)
cin >> n;   //Tiempo,memoria = O(1)
```

```
vector <int> arr(n);    //Tiempo,memoria =O(1),O(N)
for(int i=0; i<n; i++){      //Tiempo,memoria = O(N),O(1)
    cin >> arr[i];      //O(N)
}
int suma = 0;           //Tiempo,memoria = O(1),O(1)
for(int i=0; i<n; i++){      //Tiempo,memoria = O(N), O(1)
    suma += arr[i];     //Tiempo, memoria =O(N) DEL FOR
}
cout << suma << endl;  //Tiempo,memoria = O(1),O(1)
//COMPLEJIDAD TOTAL MEMORIA : O(N)
//COMPLEJIDAD TOTAL TIEMPO : O(N)
```

Codigo 2:

```
// For anidado
Tiempo,memoria

int n, m;         //O(1), O(1)
cin >> n >> m;          //O(1),O(1)
int res = 0;            //O(1),O(1)
for(int i=0; i<n; i++){      //O(N),O(1)
    for(int j=0; j<m; j++){  //O(M),O(1)
        res += i*j;          //O(N) * O(M) DEL DOBLE FOR
    }
}
cout << res << endl;         //O(1),O(1)
//COMPLEJIDAD TOTAL MEMORIA : O(1)
//COMPLEJIDAD TOTAL TIEMPO : O(N*M)
```

Código 3:

```
// For en una condicion
Tiempo, memoria
int w;            //O(1),O(1)
cin >> w;   //O(1),O(1)
if(w%2 == 0){     //O(1),O(1)
    cout << "Es par" << endl;      //O(1),O(1)
}
else{
    for(int i=0; i<w; i++){  //O(W),O(1)
        cout << i << endl;   //O(W) DEL FOR
    }
}
//COMPLEJIDAD TOTAL MEMORIA : O(1)
```

```
//COMPLEJIDAD TOTAL TIEMPO : O(W) si llega al else,O(1) si cumple el if
```

Código 4:

```
// Esto que?
Tiempo,memoria
int n;            //O(1),O(1)
cin >> n;   //O(1),O(1)
for(int i=0; i<n; i++){            //O(N),O(1)
    int j = 1;                //O(N) DEL FOR,O(1)
    while(j<n){               //O(N),O(1)
        j *= 2;               //O(N) DEL WHILE
    }
}
//COMPLEJIDAD TOTAL MEMORIA : O(1)
//COMPLEJIDAD TOTAL : O(N*N)
```

Código 5:

```
// Indices
Tiempo,memoria
int n;            //O(1),O(1)
cin >> n;   //O(1),O(1)
vector <int> arr(n);   //O(1),O(N)
vector <vector <int>> indices(n);        //O(1),O(N)
for(int i=0; i<n; i++){            //O(N),O(1)
    // en este problema leemos n números que están entre 0 y n-1.
    // Es decir, nos garantizan 0 <= arr[i] < n.
    cin >> arr[i];     //O(N),O(1)
    indices[arr[i]].push_back(i); //O(N),O(1)
}

for(int x=0; x<n; x++){            //O(N),O(1)
    int suma = 0;      //O(N),O(1)
    for(int j=0; j<indices[x].size(); j++){   //O(N),O(1)
        suma += indices[x][j];                //O(N*N) DEL DOBLE FOR

    }
    cout << suma << endl;    //O(N)
}
//COMPLEJIDAD TOTAL MEMORIA : O(N)
//COMPLEJIDAD TOTAL TIEMPO : O(N*N)
```

Código 6:

```
// Serie armonica
Tiempo, memoria
```

```
int n;            //O(1),O(1)
cin >> n;   //O(1),O(1)
int ans = 0;      //O(1),O(1)
for (int i = 1; i < n; i++) {           //O(N),O(1)
  for (int j = 1; j < n; j += i) {   //O(N),O(1)
    ans += i*j;         //  O(N LOG N)
  }
}
//COMPLEJIDAD TOTAL MEMORIA : O(1) Ó O(N)
//COMPLEJIDAD TOTAL TIEMPO : O(N LOG N)
// En este ultimo use chatgpt porque no entendia muy bien la parte
final, ya que revisando los bucles el segundo
// depende de los valores de I
```

Tarea 5.

Para cada una de las siguientes funciones f(n) y tiempo t, determine el tamaño máximo de n, asumiendo que se pueden hacer 10^8 operaciones por segundo.

Tarea5_algoritmos.ipynb

und	seg
minuto	60
hora	3600
dia	86400
mes	2592000
año	31104000
siglo	3110400000

$lg_2(n) = 10^8 \rightarrow n = 2^{10^8}$

	1 seg	1 min	1 hora	1 día	1 mes	1 año	1 siglo
$lg(n)$	2^{10^8}	$2^{(10^9*6)}$	$2^{(10^{10}*36)}$	$2^{(10^{10}*864)}$	$2^{(10^{11}*2592)}$	$2^{(10^{11}*31104)}$	$2^{(10^{13}*31104)}$
$\sqrt{n}$	$1 \cdot 10^{16}$	$3.6 \cdot 10^{19}$	$1.30 \cdot 10^{23}$	$7.46 \cdot 10^{25}$	$6.72 \cdot 10^{28}$	$9.67 \cdot 10^{30}$	$9.67 \cdot 10^{34}$
n	$1 \cdot 10^{8}$	$6 * 10^{9}$	$3.6 * 10^{11}$	$8.64 \cdot 10^{13}$	$2.59 * 10^{14}$	$3.11 * 10^{15}$	$3.11 \cdot 10^{17}$
$nlg(n)$	$4.52 \cdot 10^{6}$	$2.17 \cdot 10^{8}$	$1.08 \cdot 10^{10}$	$2.29 \cdot 10^{11}$	$6.1 \cdot 10^{12}$	$6.77 \cdot 10^{13}$	$5.94 \cdot 10^{15}$
n^2	$1 \cdot 10^{4}$	$7.75 \cdot 10^{4}$	$6 \cdot 10^{5}$	$2.94 \cdot 10^{6}$	$1.61 \cdot 10^{7}$	$5.58 \cdot 10^{7}$	$5.58 \cdot 10^{8}$
n^3	$4,64 \cdot 10^{2}$	$1.82 \cdot 10^{3}$	$7.11 \cdot 10^{3}$	$2.05 \cdot 10^{4}$	$6.38 \cdot 10^{4}$	$1.46 \cdot 10^{5}$	$6.78 \cdot 10^{5}$
2^n	26.57	32.48	38.39	42.97	47.88	51.47	58.11
$n!$	11	12	14	15	16	17	19

Tarea 6.

- Para la primera parte usare la función recursiva que permuta las letras en una cadena mostrando todos los posibles resultados permutando todas las letras.
  Enlace: ∞ Tarea6_Algoritmos.ipynb

```python
def permut(word, prefix=""):
    if len(word) == 0:
        print(prefix)
        return
    for i in range(len(word)):
        new_word = word[:i] + word[i+1:]
        permut(new_word, prefix + word[i])

permut("ab")
print()
permut("abcd")
```

Primer caso: En este caso usamos la cadena “ab” la cual la cantidad de cadenas permutadas serán 2! = 2

```
permut("ab", "")
├── i=0 → word[0]='a'
│   └── permut("b", "a")
│       └── i=0 → word[0]='b'
│           └── permut("", "ab") → ✅ imprime ab
├── i=1 → word[1]='b'
    └── permut("a", "b")
        └── i=0 → word[0]='a'
            └── permut("", "ba") → ✅ imprime ba
```

Segundo caso: En este caso usamos la cadena “abc” la cual la cantidad de cadenas permutadas serán 3! = 6

```
permut("abc", "")
├── i=0 → 'a'
│   └── permut("bc", "a")
│       ├── i=0 → 'b'
│       │   └── permut("c", "ab")
│       │       └── i=0 → 'c'
│       │           └── permut("", "abc") → ✅
│       └── i=1 → 'c'
│           └── permut("b", "ac")
│               └── i=0 → 'b'
│                   └── permut("", "acb") → ✅
├── i=1 → 'b'
│   └── permut("ac", "b")
│       ├── i=0 → 'a'
│       │   └── permut("c", "ba")
│       │       └── i=0 → 'c'
│       │           └── permut("", "bac") → ✅
│       └── i=1 → 'c'
│           └── permut("a", "bc")
│               └── i=0 → 'a'
│                   └── permut("", "bca") → ✅
```

```
└── i=2 → 'c'
    └── permut("ab", "c")
        ├── i=0 → 'a'
        │   └── permut("b", "ca")
        │       └── i=0 → 'b'
        │           └── permut("", "cab") → ✅
        └── i=1 → 'b'
            └── permut("a", "cb")
                └── i=0 → 'a'
                    └── permut("", "cba") → ✅
```

- Elegí un ejemplo de búsqueda binaria aplicado al tiempo mínimo que gastan x trabajadores haciendo una cantidad w de trabajo. En este caso usamos la búsqueda binaria para encontrar entre un tiempo low= 0 y un tiempo high que será igual a la cantidad de tiempo que le tomaría al trabajador más lento realizar todas las unidades de trabajo. Con esto ya tenemos los límites establecidos y en vez de iterar por cada cantidad de tiempo hasta encontrar la adecuada, usaremos la búsqueda binaria para ir buscándola de forma más eficiente, descartando resultados de manera más rápida.
  Enlace: Tarea6_Algoritmos.ipynb

```python
def puede_completar(trabajadores, W, tiempo):
    total = 0
    for v in trabajadores:
        total += int(v * tiempo)
        if total >= W:
            return True
    return False

def busqueda_binaria_tiempo(trabajadores, W):
    low, high = 0, max(W / min(trabajadores), 1) * 2  # un límite superior inicial (arbitrario)
    resultado = high

    for _ in range(100):  # 100 iteraciones para precisión suficiente
        mid = (low + high) / 2
        if puede_completar(trabajadores, W, mid):
            resultado = mid
            high = mid
        else:
            low = mid
    return resultado

trabajadores = [1.5, 2.0, 1.0,4.0]  # velocidades de trabajo
W = 30  # unidades de trabajo total

tiempo_min = busqueda_binaria_tiempo(trabajadores, W)
print(f"Tiempo mínimo para completar {W} unidades: {tiempo_min:.2f} unidades de tiempo")
```

Tarea 7. **Tarea7_Algoritmos.ipynb**

- Estudie, haga el código de Insertion Sort y pruebe su correctitud con una invariante de ciclo.

  R/ El insertion Sort es un algoritmo de ordenamiento que funciona insertando los elementos de una lista desordenada en el orden correcto, comparando un elemento en la lista con todos los de la izquierda, buscando si hay elementos más grandes que el seleccionado y cambiandolos, así hasta encontrar la posición correcta de cada elemento hasta ordenar la lista. Para probar la correcta eliminación revisamos los 3 pasos:

  Inicialización: Antes de iniciar el ciclo, i tiene el valor de 1, por lo tanto j - 1 = 0; Lo que implica que el arreglo A tiene solo un elemento, A[0], que está ordenado.

  Mantenimiento: Si el arreglo A es A[0,...,i], el algoritmo inserta A[i] en la posición correcta dentro del arreglo desplazando los elementos mayores a A[i] a la derecha. Esto mantiene la invarianza.

  Terminación: cuando el ciclo termina en que i es igual a la longitud del arreglo, la invariante garantiza que el arreglo A[0,..., n - 1] está ordenado demostrando la funcionalidad del algoritmo

- Estudie mergesort y quicksort, y realice el código de cualquiera de los dos en el lenguaje de programación que desee.

  R/ El mergesort es un algoritmo de ordenamiento muy conocido por su eficiencia y estabilidad, funciona usando la idea de divide y vencerás dividiendo el arreglo en subarreglos ordenándolos por aparte y uniendolos al final. El proceso es dividir el arreglo en mitades recursivamente hasta que no se pueda dividir más, cada subarreglo es ordenado usando el algoritmo merge sort, los subarreglos ordenados son unidos en orden hasta que se unan todos los elementos de los subarreglos.

  El quicksort al igual que el mergesort se basa en la idea de divide y vencerás, el cual elige un elemento “pivote” y subdivide el arreglo de forma que los elementos menores al pivote se organizan a la izquierda y los mayores a la derecha generando subarreglos de 1 o 0 elementos ordenados recursivamente, y de esta misma forma vuelve a unir los elementos hasta que se unan todos ordenadamente.

Tarea 8. **Tarea7_Algoritmos.ipynb**

Radix Sort ordena números dígito por dígito, desde el menos significativo al más significativo, usando un algoritmo de ordenamiento estable en cada paso. Para hacerlo hacemos lo siguiente:

1. Encuentra el número con más dígitos.

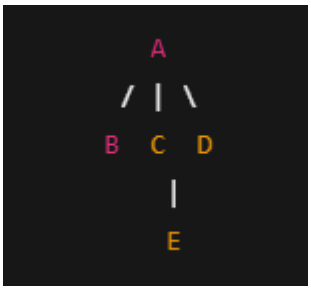
2. Ordena todos los números según el dígito de las unidades.

3. Luego según el dígito de las decenas, centenas, etc.

4. Repite hasta procesar todos los dígitos.

En el collaboratory adjunto se encuentra una implementación basada en un código de Geek For Geeks.

Tarea 9. Articulos, arbol binario
R/ Un grafo es un conjunto de nodos que se conectan atados por un conjunto de ejes que conectan pares de nodos distintos. Un árbol es una colección de nodos y ejes(conexiones) en el árbol solamente existe un camino conectando dos nodos diferentes, si hubiera más caminos sería un grafo y no un árbol. Un árbol se diferencia de un grafo si esencialmente no tiene ciclos, y un camino conecta un para de nodos.
Un árbol se puede representar en un arreglo definiendo la relación de cada nodo con su padre. Cada posición del arreglo representa un nodo, y el valor almacenado en esa posición nos indica el índice su padre, si el nodo es la raíz le asignamos el valor de -1. Un ejemplo es:



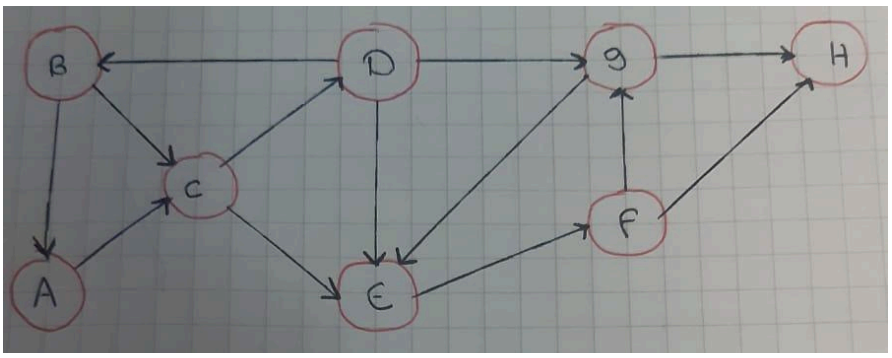
El arreglo de ese árbol sería $Arr = [-1, 0, 0, 0, 3]$ donde:

- Nodo **A** (índice 0) es la raíz = -1
- Nodo **B** tiene como padre a **A** (índice 0)
- Nodo **C** tiene como padre a **A**
- Nodo **D** tiene como padre a **A**
- Nodo **E** tiene como padre a **D** (índice 3)

Tarea 10

1. **Representación de un grafo.** Invente y dibuje a mano (papel o digital) un grafo no dirigido sin pesos con por lo menos 6 nodos y al menos 7 aristas. Y escriba las siguientes representaciones:

a. Dibuje el grafo



b. Representación directa por conjuntos (lista de nodos y aristas)
   - Conjunto de nodos: $V = \{A, B, C, D, E, F, G, H\}$
   - Conjunto de aristas:
     $A = \{(B, A), (B, C), (B, D), (C, A), (C, D), (C, E), (D, E), (D, G), (E, F), (E, G), (F, G), (F, H), (G, H)\}$

c. Matriz de adyacencia.

	A	B	C	D	E	F	G	H
A	0	0	1	0	0	0	0	0
B	1	0	1	0	0	0	0	0
C	0	0	0	1	1	0	0	0
D	0	1	0	0	1	0	1	0
E	0	0	0	0	0	1	0	0
F	0	0	0	0	0	0	1	1
G	0	0	0	0	0	0	0	1
H	0	0	0	0	0	0	0	0

d. Lista de adyacencia
   - A → C
   - B → A, C
   - C → D, E
   - D → B, E, G
   - E → F
   - F → G, H
   - G → H
   - H → N/A

1. **Representando movimientos con arreglos de delta.** Identifique todos los movimientos únicos y represente cada uno con un cambio en fila (Δfila) y un cambio

en columna (Δcolumna). Organice estos cambios en dos arreglos separados para los siguientes escenarios.

a. Caballo de ajedrez: El caballo de ajedrez se mueve en dos casillas en una dirección vertical u horizontal, y una casilla en dirección perpendicular, por ejemplo si se mueve 2 arriba se puede mover 1 a la derecha o izquierda.
   - $\Delta Fila = [-2, -1, 1, 2, 2, 1, -1, -2]$
   - $\Delta Columna = [1, 2, 2, 1, -1, -2, -2, -1]$

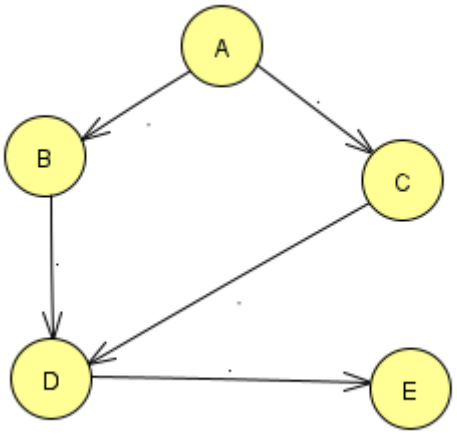
b. Pieza, personaje o mecanismo de algun juego que usted conozca (tablero 2D o 3D): En pokemon rojo fuego un juego 2D con un mapa de cuadrícula, el personaje cuenta con 4 direcciones (arriba,abajo,izquierda,derecha)

   - $\Delta Fila = [-1, 1, 0, 0]$
   - $\Delta Columna = [0, 0, -1, 1]$
   - Arriba = $(-1, 0)$
   - Abajo = $(1, 0)$
   - Izquierda =$(0, -1)$
   - Derecha =$(0, 1)$

Tarea 11:

1. Orden topológico

a. Dibuje un grafo dirigido y calcule un orden topológico válido.



$$A \Rightarrow B \Rightarrow C \Rightarrow D \Rightarrow E$$

b. El orden topológico solo se puede calcular en DAGs(Directed Acyclic Graphs) sin embargo los algoritmos de cálculo de orden topológico se pueden usar para identificar si un grafo tiene ciclos. Explique cómo identificar si existe un ciclo para
   - El algoritmo de Khan basado en bfs:El algoritmo de Kahn funciona seleccionando los vértices en el mismo orden en que aparecen en un ordenamiento topológico. El primer paso consiste en identificar una lista de nodos iniciales, es decir, aquellos que no tienen aristas entrantes (no reciben ninguna arista de otros nodos). Estos nodos se insertan en un conjunto o estructura (como una cola), desde donde se procesan para construir el orden topológico.

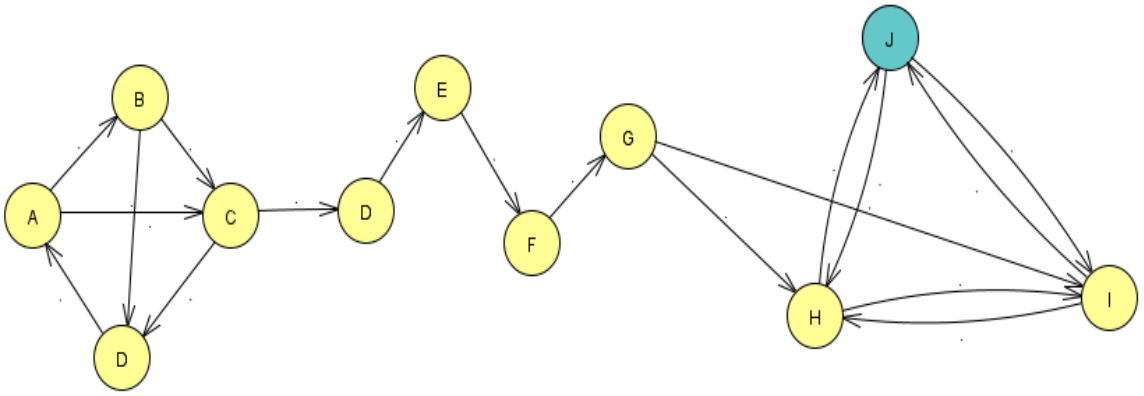
```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

   - El algoritmo basado en dfs: El algoritmo de ordenamiento topológico usando DFS (búsqueda en profundidad) recorre el grafo explorando completamente

cada rama antes de retroceder. Durante este recorrido, se marca cada nodo como visitado, y una vez que se han procesado todos sus sucesores, el nodo se añade a una pila o lista. Al final, el orden topológico se obtiene invirtiendo el contenido de esa pila. Este método también permite detectar ciclos si durante el recorrido se encuentra un nodo que ya está en proceso (es decir, se ha iniciado su exploración pero no ha terminado), lo que indica la presencia de un ciclo en el grafo.

```
L ← Empty list that will contain the sorted nodes
while exists nodes without a permanent mark do
    select an unmarked node n
    visit(n)

function visit(node n)
    if n has a permanent mark then
        return
    if n has a temporary mark then
        stop   (graph has at least one cycle)

    mark n with a temporary mark

    for each node m with an edge from n to m do
        visit(m)

    mark n with a permanent mark
    add n to head of L
```

2. Componentes fuertemente conectados. Dibuje un grafo dirigido conexo con dos componentes fuertemente conexas y con las siguientes características
Un componente debe tener 4 nodos y al menos 6 arcos.
La otra componente debe tener 3 nodos y al menos 6 arcos
Debe tener al menos 3 nodos que no hagan parte de ninguna de las dos componentes fuertemente conexas.


J
B
E
G
A
C
D
F
H
I
D