UNIVERSIDAD DE CONCEPCION
FACULTAD DE CIENCIAS
FISICAS Y MATEMATICAS
DEPARTAMENTO DE INGENIERIA MATEMATICA

---

1. Pruebe las siguientes proposiciones.

   a) $2^{n+1} = \Theta(2^n)$ y $2^{2^{n+1}} \neq \Theta(2^{2^n})$.

   b) $n = O(n \log n)$ y $n^{\log n} = O(n!)$.

   c) $o(f(n)) \cap \omega(f(n)) = \emptyset$.

   d) $O(3^{n^2}) \neq 3^{O(n^2)}$.

   Soln:

   a) $2^{n+1} = \Theta(2^n)$ y $2^{2^{n+1}} \neq \Theta(2^{2^n})$.
   (2.5 ptos.) Para el primer caso, notemos que:

   $$\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = \lim_{n\to\infty} \frac{2 \cdot 2^n}{2^n} = 2 > 0$$

   De lo anterior y del Listado 1 Problema 5 se prueba que $2^{n+1} = \Theta(2^n)$.
   (2.5 ptos.) Para el segundo caso, notemos que:

   $$\lim_{n\to\infty} \frac{2^{2^{n+1}}}{2^{2^n}} = \lim_{n\to\infty} 2^{2\cdot 2^n - 2^n} = \lim_{n\to\infty} 2^{2^n} = \infty$$

   Es por ello y del Listado 1 Problema 4 a) se tiene que $2^{2^{n+1}} \neq O(2^{2^n})$, lo cual prueba que $2^{2^{n+1}} \neq \Theta(2^{2^n})$.

   b) $n = O(n \log n)$ y $n^{\log n} = O(n!)$.
   (2 ptos.) Notemos que $\forall n \geq 2$, $\log n \geq 1$ y por consiguiente $n \log n \geq n$. Así, tomando $n_0 = 2$ y $c = 1$ en la definición de O grande se prueba que $n = O(n \log n)$.
   (3 ptos.) Para el segundo caso, del Listado 1 Problema 3 sabemos que $2^n = O(n!)$, es decir,

   $$\exists c_1 > 0, \exists n_1 \in \mathbb{N} : 2^n \leq c_1 n! \qquad \forall n \geq n_1 \tag{1}$$

   Además, se obtiene que $\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$, lo cual corresponde a la definición de o chica e implicando que $\log n = o(\sqrt{n})$. Así, podemos tomar $c_2 = 1$ particular y por definición de o chica existirá $n_2 \in \mathbb{N}$ ($n_2 = 16$) tal que

   $$\log n \leq c_2 \sqrt{n} \qquad \forall n \geq n_2 \tag{2}$$

   Por lo tanto, basta tomar $n_3 = \text{máx}\{n_1, n_2\}$ y aplicando (2) y (1) respectivamente se llega a la definición de $n^{\log n} = O(n!)$. En efecto,

   $$n^{\log n} = (2^{\log n})^{\log n} \leq (2^{\sqrt{n}})^{\sqrt{n}} = 2^n \leq c_1 n! \qquad \forall n \geq n_3$$

c) $o(f(n)) \cap \omega(f(n)) = \emptyset$.

(5 ptos) Por el absurdo, supongamos que $\exists g(n) \in o(f(n)) \cap \omega(f(n))$. De la definición de o chica y ω chica, respectivamente,

$$\forall c_1 > 0, \exists n_1 \in \mathbb{N} : g(n) \leq c_1 f(n) \qquad \forall n \geq n_1 \tag{3}$$

$$\forall c_2 > 0, \exists n_2 \in \mathbb{N} : f(n) \leq c_2 g(n) \qquad \forall n \geq n_2 \tag{4}$$

Tomando $n_3 = \text{máx}\{n_1, n_2\}$ y $c_1 = 1/c_2$, de (3) y (4) se tiene que:

$$f(n) \leq c_2 g(n) \leq c_1 c_2 f(n) = f(n) \qquad \forall n \geq n_3$$

$$\Rightarrow 1 \leq c_2 \frac{g(n)}{f(n)} \leq 1 \qquad \forall n \geq n_3$$

Lo cual contradice que $g(n)$ sea o chica de $f(n)$ debido a que la definición equivalente de o chica nos dice que el siguiente límite es cero y por lo anterior da distinto de cero.

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{\substack{n\to\infty \\ n>n_3}} \frac{g(n)}{f(n)} = \frac{1}{c_2} \neq 0$$

Por lo tanto, $o(f(n)) \cap \omega(f(n)) = \emptyset$.

d) $O(3^{n^2}) \neq 3^{O(n^2)}$.

(5 ptos.) Sea $g(n) = 2n^2 \in O(n^2)$ y notemos que $3^{g(n)} \in 3^{O(n^2)}$. Notemos que:

$$\lim_{n\to\infty} \frac{3^{g(n)}}{3^{n^2}} = \lim_{n\to\infty} 3^{2\cdot n^2 - n^2} = \lim_{n\to\infty} 3^{n^2} = \infty$$

Así, del Listado 1 Problema 4 a) se tiene que $3^{g(n)} \notin O(3^{n^2})$ y en particular $O(3^{n^2}) \neq 3^{O(n^2)}$.

2. Para cada uno de los siguientes problemas, muestre un algoritmo polinomial que lo resuelva. Justifique su respuesta.

   a) ACYCLIC GRAPH: Dado $G = (V, E)$ un grafo no dirigido ¿Es G un grafo sin ciclos?

   b) ODD CYCLE: Dado $G = (V, A)$ un grafo dirigido ¿Existe un ciclo de largo impar en G?

Soln:

a) **(10 ptos.)** El siguiente algoritmo está basado en el hecho que todo ciclo de G que contiene una arista $\{u, v\}$ tiene un camino entre u y v que no contiene a la arista $\{u, v\}$. Luego, el siguiente algoritmo permite resolver el problema de determinar si un grafo (no dirigido) es acíclico o no.

Algorithm Acyclic(G)

Require: $G = (V, E)$ un grafo (no dirigido)

```
for all {u, v} ∈ E do
    if PATH(G − {u, v}, u, v) = s then
        return n
    end if
end for
return s
```

El número de operaciones elementales del algoritmo Acyclic está dado por el ciclo for que en el peor caso son $O(|E|)$ iteraciones y en cada iteración se ejecuta el algoritmo PATH, visto en clase, cuyo tiempo de ejecución es $O(|V| + |E|)$ (o $O(|V|^2)$) y se determina $G - \{u, v\}$ que puede hacerse (dependiendo de la implementación de G) en $O(1)$ operaciones elementales. Luego, el tiempo de ejecución del algoritmo es $O(|E| \cdot (|V| + |E|))$, lo que es polinomial en el tamaño de G.

b) **(10 ptos.)** El siguiente algoritmo OddCycle usa el siguiente resultado conocido en digrafo:

Proposición: *Sea $G = (V, A)$ un digrafo fuertemente conexo no trivial. G tiene un ciclo de largo impar si y sólo si G es bipartito, es decir existe $\{V^1, V^2\}$ partición de V tal que V^1 y V^2 son juntos de vértices independientes, i.e. no hay arcos de A cuyos vértices extremos estén en un mismo conjunto de la partición.*

Luego, la idea del algoritmo OddCycle es primero chequear que no hay bucles y luego que hay al menos una componente fuertemente conexa no trivial. Para cada componente fuertemente conexa no trivial se chequea que sea bipartita. Una forma de hacer esto último es fijar un vértice s cualquiera de la componente y determinar lo que están a distancia par e impar desde s. Luego, la componente no es bipartita (por lo tanto tiene un ciclo de largo impar) si y sólo si hay algún arco entre los vértices de distancia par o entre los vértices de distancia impar.

Algorithm OddCycle(G)

```
Require: G = (V, A) un digrafo
(r, i, comp) ← SCONNECTED(G)
for all u ∈ V do
    if (u, u) ∈ A then
        return s
    end if
end for
if i = |V| then
    return n
end if
j ← 1
while j < n do
    if |comp[i]| ≥ 2 then
        Seleccionar s ∈ comp[j]
        (d, π) ← BFS(G, s)
        for all u, v ∈ comp[i] do
            if d[u] y d[v] son pares y {(u, v), (v, u)} ∩ A ≠ ∅ then
                return s
            end if
            if d[u] y d[v] son impares y {(u, v), (v, u)} ∩ A ≠ ∅ then
                return s
            end if
        end for
    end if
    j ← j + 1
end while
return n
```

El tiempo de ejecución del algoritmo OddCycle está dado principalmente por el ciclo While que ejecuta $O(|V|)$ iteraciones en el peor caso. En cada iteración se ejecuta BFS, que ejecuta $O(|A|+|V|)$ operaciones elementales en el peor caso, y el ciclo for de

la línea 15 que son $O(|V^2|)$ iteraciones en el peor caso. Las operaciones en las líneas 16 y 19 pueden realizarse con $O(|A|)$ operaciones elementales. De aquí, el ciclo While ejecuta $O(|V|) \cdot (O(|A| + |V|) + O(|V^2|) \cdot O(|A|))$ que es $O(|V|^5)$ operaciones elementales en el peor caso. Por otro lado, el algoritmo SCONNECTED ejecuta $O(|V|^3)$ operaciones elementales y el ciclo for de la línea 2 puede ser ejecutado en $O(|V|)$ operaciones elementales en el peor caso. Las otras operaciones pueden ser ejecutadas en $O(|1|)$ operaciones elementales. Por lo tanto el algoritmo OddCycle ejecuta en el peor caso $O(|V|^5)$ operaciones elementales, lo que significa que es polinomial en el tamaño de la entrada.

3. Sea $G = (V, A)$ un digrafo. Se dice que G tiene un **ordenamiento topológico** de sus vértices si ellos pueden ser ordenados en una secuencia $v_1, v_2, \ldots, v_n$, donde $V = \{v_1, \ldots, v_n\}$ y tal que si $(v_i, v_j) \in A$, entonces $i < j$.

   a) Pruebe que G tiene un ordenamiento topológico de sus vértices si y sólo si G es acíclico.

   b) Construya un algoritmo polinomial que reciba como entrada un digrafo G y retorne un orden topológico de sus vértices cuando exista.

   c) Programe en Matlab o Python el algoritmo definido en *b*) que recibe la matriz de adyacencia de un digrafo $G = (V, A)$ cualquiera con $V = \{1, \ldots, n\}$ y retorne un orden topológico de los vértices cuando éste existe y retorne *no existe* en caso contrario.

Soln:

a) **(2 ptos.)** ($\Longrightarrow$) Supongamos que $v_1, v_2, \ldots, v_n$ es un ordenamiento topológico de los vértices de G. Sea $C : v_{i_1}, \ldots, v_{i_k}, v_{i_1}$ un ciclo de G. Luego, como $\forall j = 1, \ldots, k, (v_{i_j}, v_{i_{j+1}}) \in A$, con $v_{i_{k+1}} = v_{i_1}$, entonces se tiene que:

$$i_1 < i_2 < \cdots < i_k < i_1,$$

lo cual es una contradicción.

($\Longleftarrow$) Para probar esta implicancia usaremos el siguiente básico y bien conocido resultado de digrafos:

Proposición: *Sea $G = (V, A)$ un digrafo. Si $\forall v \in V,\ d^-(v) \geq 1$ ó $\forall v \in V,\ d^+(v) \geq 1$, entonces G tiene un ciclo.*

Luego, si G es acíclico, entonces $\exists v_1 \in V$ tal que $d^-_G(v_1) = 0$. Definamos el digrafo $G_2 := G - v_1$. Como, G_2 es también acíclico (pues es subgrafo de G) entonces $\exists v_2 \in V - \{v_1\}$ tal que $d^-_{G_2}(v_2) = 0$. De esta forma, podemos construir de manera recursiva una secuencia de los vértices de $V : v_1, v_2, \ldots 1, v_n$ tal que $\forall j = 2, \ldots, n,\ d^-_{G_j}(v_j) = 0$ con $G_j := G_{j-1} - v_{j-1} = G - \{v_1, \ldots, v_{j-1}\}$ donde $G_1 := G$. Mostremos que $v_1, v_2, \ldots, v_n$ es un orden topológico de G. En efecto si $(v_i, v_j) \in A$, entonces $v_i \notin V(G_j)$, pues como $v_j \in V(G_j)$, entonces en caso contrario $d^-_{G_j}(v_j) \geq 1$, lo cual es una contradicción con $d^-_{G_j}(v_j) = 0$. Por lo tanto, como $v_i \notin V(G_j)$, entonces $i < j$.

b) **(8 ptos.)** El siguiente algoritmo ToplogicalOrder está basado en el resultado anterior. Se usa el algoritmo Acyclic visto en prática que determina si un digrafo es acíclico o no. El tiempo de ejecución está dado por el algoritmo Acyclic que ejecuta $O(|V|^3)$ operaciones elementales y por el ciclo while que ejecuta $O(|V|)$ iteraciones en el peor

caso. Determinar si $d_G^-(v) = 0$ y calcular $G - v$ pueden ser hechos en $O(|V|)$ operaciones elementales en el peor caso. El resto de las operaciones son $O(1)$. Luego, el tiempo de ejecución de TopologicalOrder es $O(|V|^3) + O(|V|) \cdot O(|V|)$ lo que es $O(|V|^3)$, ie el algoritmo es polinomial.

Algorithm TopologicalOrder(G)

Require: $G = (V, A)$ un digrafo
1: **if** $Acyclic(G) = n$ **then**
2: **return** n
3: **end if**
4: $T \leftarrow Null$, $i \leftarrow 1$
5: **while** $i \leq n$ **do**
6: Seleccionar $v \in V(G)$ tal que $d_G^-(v) = 0$.
7: $T[i] \leftarrow v$
8: $G \leftarrow G - \{v\}$
9: $i \leftarrow i + 1$
10: **end while**
11: **return** T