Ejercicio 1

En clases se vio esta versión de Bellman-Ford, que encuentra el camino más corto (o “barato”) en un grafo direccional, acíclico.

Básicamente, el algoritmo hace |V|−1 pasadas por todas las aristas del grafo, tratando de reducirlas (o relajarlas). Sin embargo, las únicas aristas que podrían producir un cambio en d[] son aquellas que salen de un vértice cuyo d[] cambió en la pasada anterior.

```
BellmanFord(s):
    for u ∈ V :
        d[u] ← ∞; π[u] ← ∅
    d[s] ← 0
    for k = 1 … |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + cost(u, v) :
                d[v] ← d[u] + cost(u, v)
                π[v] ← u
```

Ejercicio 1

Modifica este algoritmo de manera que intente reducir aristas sólo cuando tenga sentido hacerlo.

HINT: Emplea una cola y un arreglo de booleanos que permitan detectar cuando hay que detenerse.

```
BellmanFord(s):
    for u ∈ V :
        d[u] ← ∞; π[u] ← ∅
    d[s] ← 0
    for k = 1  … |V| − 1 :
        for (u, v) ∈ E :
            if d[v] > d[u] + cost(u, v) :
                d[v] ← d[u] + cost(u, v)
                π[v] ← u
```

Ejercicio 1

Nuevos elementos:

Cola **Q**: Donde iremos almacenando los nodos que han cambiado, y que son candidatos a alterar d[]

Arreglo de bools **Encolado**[] : Arreglo que nos dirá si el nodo v se encuentra en la cola Q

Ejercicio 1

```
Bellman-Ford'(s):
    d[s] = 0                                # suponemos que todos los otros d[] son inicialmente ∞
    Q.enqueue(s)                            # suponemos que Q está inicialmente vacía
    Encolado[s] = True                      # Todos los otros casilleros de Q son False, salvo s
    while !Q.empty():                       # Iteramos mientras Q no esté vacía (y no |V| - 1 veces)
        u = Q.dequeue()
        Encolado[u] = False
        for each v in Vecinos[u]:           # Tratamos de reducir cada arista que sale de u
            if d[v] > d[u] + ω(u,v):
                d[v] = d[u] + ω(u,v)
                π[v] = u
                if !Encolado[v]:                    #si pudimos reducir (u,v), entonces
                    Q.enqueue(v)                    # guardamos v en Q(si v no está ya en Q)
                    Encolado[v] = True
```

Ejercicio 2

Dado un grafo G = (V,E), con un vector de pesos w (todos positivos) y un nodo ***s*** de partida, Al profesor Luigi le interesa el problema de encontrar el costo del camino simple (sin ciclos) **de mayor costo entre *s* y cada nodo del grafo**.

Charles Walls, ex-alumno de IIC2133, argumenta que una modificación del algoritmo de Bellman-Ford (BF) puede resolver este problema. Su razonamiento es el siguiente:

"Como el camino que buscamos no tiene ciclos, el camino más largo entre dos nodos tiene a lo más |V| - 1 aristas. De esta forma podemos primero buscar los caminos más largos de 1 arista, luego los de dos aristas, y así sucesivamente, tal como lo hace BF."

Ejercicio 2

El algoritmo propuesto por Charles es el siguiente:

```
BellmanFordn´t(s):
    for u ∈ V :
        d[u] ← −∞; π[u] ← Ø
    d[s] ← 0
    for k = 1  ...  |V| − 1 :
        for (u, v) ∈ E :
            if d[v] < d[u] + cost(u, v) :
                d[v] ← d[u] + cost(u, v)
                π[v] ← u
```

*Los únicos cambios con respecto a Bellman-Ford original son:

1. En lugar de ∞ se asigna -∞
2. En lugar de > tenemos un <.

Ejercicio 2

El profesor Luigi duda un poco de este algoritmo, por lo que te hace las siguientes preguntas:

1) Existe algún contraejemplo que muestre que este algoritmo no siempre funciona? Descríbelo
2) Debe cumplirse alguna condición para que este algoritmo funcione? Cuál es?

```
BellmanFordn´t(s):
    for u ∈ V :
        d[u] ← −∞; π[u] ← ∅
    d[s] ← 0
    for k = 1 … |V| − 1 :
        for (u, v) ∈ E :
            if d[v] < d[u] + cost(u, v) :
                d[v] ← d[u] + cost(u, v)
                π[v] ← u
```

3) El profesor afirma que hay parte del argumento del alumno que efectivamente está correcto: ***"Los caminos más largos no pueden tener más de |V|−1 aristas"***. Diga cómo es posible modificar la idea de BF para encontrar caminos más largos. Analice cuál sería la complejidad del algoritmo resultante. No es necesario que explicites el algoritmo.

Ejercicio 2

1) Existe algún contraejemplo que muestre que este algoritmo no siempre funciona? Descríbelo:

Respuesta: El contraejemplo se construye con un grafo con un ciclo en el camino hacia un nodo.

Ejercicio 2

1) Existe algún contraejemplo que muestre que este algoritmo no siempre funciona? Descríbelo:

Respuesta: El contraejemplo se construye con un grafo con un ciclo en el camino hacia un nodo.

2) Debe cumplirse alguna condición para que este algoritmo funcione? Cuál es?

Respuesta: Dado que la solución está basada en BF, podríamos pensar que encontrar el camino más caro sin ciclos en un grafo G = (V,E) con pesos w es equivalente a encontrar el camino más barato sin ciclos en un grafo G = (V,E) con −w. Como BF encontrará el camino más barato en un grafo sin ciclos negativos, debe cumplirse esta condición para que este nuevo algoritmo funcione

Ejercicio 2

3) El profesor afirma que hay parte del argumento del alumno que efectivamente está correcto: ***"Los caminos más largos no pueden tener más de $|V|-1$ aristas"***. Diga cómo es posible modificar la idea de BF para encontrar caminos más largos. Analice cuál sería la complejidad del algoritmo resultante. No es necesario que explicites el algoritmo.

Respuesta: Debemos modificar BF de forma que cada vez que se encuentra un nuevo camino hasta un nodo, se guarde explícitamente, junto a su costo (esto con el fin de poder ir actualizando cualquier solución más cara que encontremos).

Ejercicio 2

Esto significa que debemos tener una matriz d[v][i] que almacena el costo del i-ésimo camino sin ciclos hasta v que se ha encontrado. El camino a su vez, se almacena en π [v][i].

Cuando miramos (u, v) debemos usar esta arista completando cada camino hasta u para generar un nuevo camino hasta v. El costo de computar esto depende del tamaño de d[v][], que en el peor caso debe llevar la cuenta de todos los caminos posibles desde la fuente hasta v, que es exponencial en $|V|$ ($O(|V|!)$, de hecho)

El algoritmo principal, entonces, realiza $|V|-1$ iteraciones, pero cada iteración es $O(V!)$. El algoritmo es $O((|V| + 1)!)$.