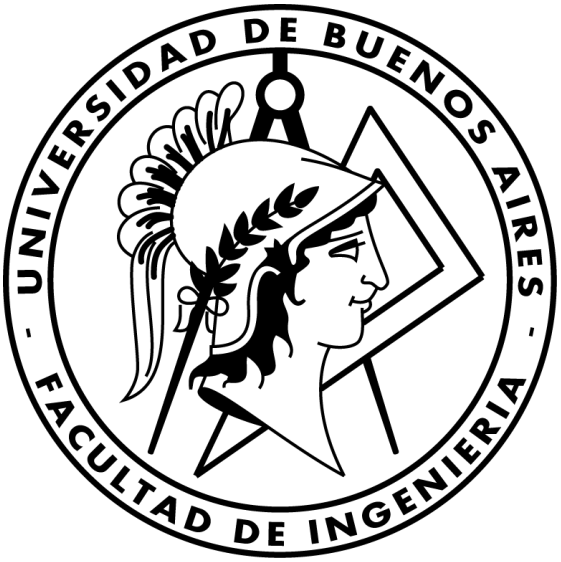


Trabajo Práctico N°2

1° Cuatrimestre 2022

75.29 / 95.06 - Teoría de Algoritmos

Los Condicionales

Nombre	Padrón
Mariano Fernández Vidal	89789
Matias Nicolas Lynch	102571
Manuel Longo Elia	102425
Azul Zaietz	102214

Parte 1: Un evento exclusivo

1- Explicar cómo se puede resolver este problema por fuerza bruta. Analizar complejidad espacial y temporal de esta solución

Este problema se puede resolver por fuerza bruta, de la siguiente forma:
Se define una lista S, en la que para cada jugador se guarda a cuantos jugadores se superó el año pasado, inicialmente todos los valores están en cero. En la lista S, la posición i corresponde al jugador i en el ranking actual.
Para determinar cada jugador a cuántos superó el año pasado, dada la lista de n jugadores, tomó el primer jugador y comparó con los restantes n-1 jugadores si el ranking del año pasado es menor al de jugador seleccionado. En caso de que se cumpla le sumo uno al jugador en la lista S. Y se procede de la misma forma para los restantes jugadores.

A continuación se muestra el pseudocódigo de la solución por fuerza bruta:

```
L = lista de n jugadores [ (nombre, ranking_año_pasado) ]
S = lista de n posiciones superadas [0, 0, ..., 0]

Desde i = 0 a n
        Desde j = i + 1 a n
                si L[ i ].ranking_año_pasado < L[ j ].ranking_año_pasado
                        S[ i ] += 1

Desde i=0 a n
        imprimir L[ i ].nombre -> i ( S[ i ] )
```

Complejidad temporal

Se puede ver que dado que se tiene que recorrer dos veces la lista de jugadores la complejidad temporal es:

$$O(n^2)$$

Complejidad espacial

Dado que solo se utiliza una lista de n elementos para guardar a cuantos jugadores superó, la complejidad espacial es:

$$O(n)$$

2- Proponer una solución utilizando la metodología de división y conquista que sea más eficiente que la propuesta anterior. (incluya pseudocódigo y explicación)

Como solución de división y conquista se utilizará un algoritmo similar al visto en clase de contar inversiones.
Dada el listado que tiene el nombre del jugador y la posición del ranking del año pasado, ordenado por el ranking actual. Se define una lista L que tendrá para cada jugador la posición del ranking del año pasado y la posición actual:

$$L = [\{ ranking\_año\_pasado, ranking\_actual\}]$$

Se define una lista S, que tendrá para cada jugador el nombre y la cantidad de rivales que superó, inicialmente en cero. Cada posición i de la lista corresponde a la posición i del jugador en el ranking actual:

$$S = [\{ nombre, cantidad\_superado \}]$$

En este caso, se considerará una inversión si dado los elementos a_i y a_j con $i < j$, se tiene:

$$a_i.ranking\_año\_pasado > a_j.ranking\_año\_pasado$$

El algoritmo consiste en ir dividiendo la lista L hasta un mínimo caso base de un elemento. Una vez que se llega al caso base se retorna la lista de un elemento y se la compara con la lista siguiente y ordenan según el ranking del año pasado.
Al comparar cada elemento de la primera lista con cada elemento de la segunda lista, si hay una inversión se suma uno a la cantidad de jugadores superados en la lista S, del jugador de la primera lista. En caso de tener una inversión se reordenan los elementos en una nueva lista. Finalmente se retorna la lista ordenada.
El algoritmo finaliza cuando se vuelve a tener la lista de tamaño original.
A continuación se muestra el pseudocódigo de la solución por división y conquista:

```
Dada la lista ordenada D

Desde i = 0 hasta D.length

  L[ i ] = { ranking_año_pasado: D[i].ranking_año_pasado, ranking_actual: i }
  S[ i ] = { nombre: D[i].nombre, cantidad_superados: i }

resultado = ordenarContar(L, S)

Desde i = 0 hasta S.length
  imprimir S[ i ].nombre -> i + 1 ( S[ i ].cantidad_superados )
```

```
ordenarContar( L, S)

  Si L.length es 1

    retorno L

  Sino

    medio = L.length / 2
    A = L[ 0 : medio]
    B = L [ medio : ]

    A = ordenarContar(A, S)
    B = ordenarContar(B, S)
    L = mergeContar(A, B, S)

    retorno L
```

```
mergeContar(A, B, S)
  L = []
  inv = 0, i = 0, j = 0

  Repetir

    a = A[ i ]
    b = B[ j ]

    Si a.ranking_año_pasado < b.ranking_año_pasado

      L[ i+j ] = a
      i += 1
      S[ a.ranking_actual ].cantidad_superados += inv

    Sino

      L[ i+j ] = b
      j += 1
      inv = inv + A.length - i

  Mientras i < A.length y j < B.length

  Desde i hasta A.lentgh
    S[ A[i].ranking_actual ].cantidad_superados += inv
    L[ i+j ] = A[i]
  Desde j hasta B.lentgh
    L[ i+j ] = B[j]

  retornar L
```

3- Realizar el análisis de complejidad temporal mediante el uso del teorema maestro.

Para analizar mediante el teorema maestro, primero se debe buscar la ecuación de recurrencia, expresada de la forma:

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

$a \geq 1$ y $b \geq 1$ constantes
f(n) una función
T(0)=cte

Para el algoritmo definido en el punto anterior se tiene que dado n jugadores se divide el problema en dos subproblemas de n/2 jugadores.

Para ordenar y contar la cantidad de inversiones, se recorre los n elementos, por lo que tiene una complejidad de O(n).

Finalmente el caso base es una operación O(1) ya que solo consiste en comparar si el tamaño de la lista es igual a 1.

Por lo tanto se puede expresar la ecuación de recurrencia como:

$$T(n) = 2T(n/2) + O(n)$$

Para obtener la complejidad se aplicará el segundo caso del teorema maestro:

$$\text{Si } f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} * \log n\right)$$

Dada la f(n)= O(n) se tiene:

$$f(n) = O(n) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

Dado que se cumple lo anterior entonces la complejidad es:

$$T(n) = \Theta\left(n^{\log_2 2} * \log n\right) = \Theta(n * \log n)$$

Por lo tanto la complejidad temporal de la solución propuesta es:

$$O(n * \log n)$$

4- Realizar el análisis de complejidad temporal desenrollando la recurrencia

Utilizando la ecuación de recurrencia definida en el punto anterior:

$$T(n) = 2T(n/2) + c\ n$$

$$T(1) = d$$

d y c constantes
n cantidad de jugadores

Desenrollando se tiene:

$$T(n) = 2T(n/2) + c\ n$$

$$T(n/2) = 2T(n/4) + c\ n/2$$

$$T(n/4) = 2T(n/8) + c\ n/4$$

…

Uniendo los resultados se llega a:

$$T(n) = 2\ (\ 2\ (\ 2\ (\ 2T(n/16) + c\ n/8\ ) + c\ n/4\ ) + c\ n/2\ ) + c\ n$$

Esta ecuación se puede expresar como la siguiente sumatoria:

$$T(n) = 2^{k} * d + \sum_{i=0}^{k} \frac{2^{i} * n}{2^{i}}$$

k es la cantidad de niveles en que se divide el problema, dado que se dividen de a mitades hasta llegar a un nivel de un elemento, k toma el valor de k = log(n). Por lo tanto la sumatoria queda de la forma:

$$T(n) = 2^{log(n)} * d + \sum_{i=0}^{log(n)} \frac{2^{i} * n}{2^{i}}$$

$$T(n) = 2^{log(n)} * d + n \sum_{i=0}^{log(n)} \left(\frac{2}{2}\right)^{i}$$

$$T(n) = 2^{log(n)} * d + n \sum_{i=0}^{log(n)} 1^{i}$$

$$T(n) = 2^{log(n)} * d + n * log(n)$$

En nuestro algoritmo d hace referencia a la constante del caso base y dado que es O(1) se lo puede cambiar por el valor 1 y utilizando la propiedad de logaritmos, la ecuación queda como:

$$T(n) = n^{log(2)} + n * log(n)$$

$$T(n) = n + n * log(n)$$

por lo tanto la complejidad queda como:

$$T(n) = O(n) + O(n * log(n)) = O(n * log(n))$$

5- Analizar la complejidad espacial basándose en el pseudocódigo.

Para la complejidad espacial se tendrán las siguientes listas:

Una lista L que tendrá para cada jugador la posición del ranking del año pasado y la posición actual. Esto genera un espacio extra de 2n que da una complejidad de O(n).
Una lista S que tendrá para cada jugador el nombre y la cantidad de rivales que superó. Esto genera un espacio extra de 2n que da una complejidad de O(n).
En cada ordenamientos sublistas de una longitud máxima de 2n elementos. Esto genera un espacio extra de 2n que da una complejidad de O(n).
Entonces, la complejidad espacial estará dada por la suma de la complejidades anteriores lo que da una complejidad espacial de O(n)

6 - Dar un ejemplo completo del funcionamiento de su solución

Como ejemplo se utilizara el caso del enunciado:

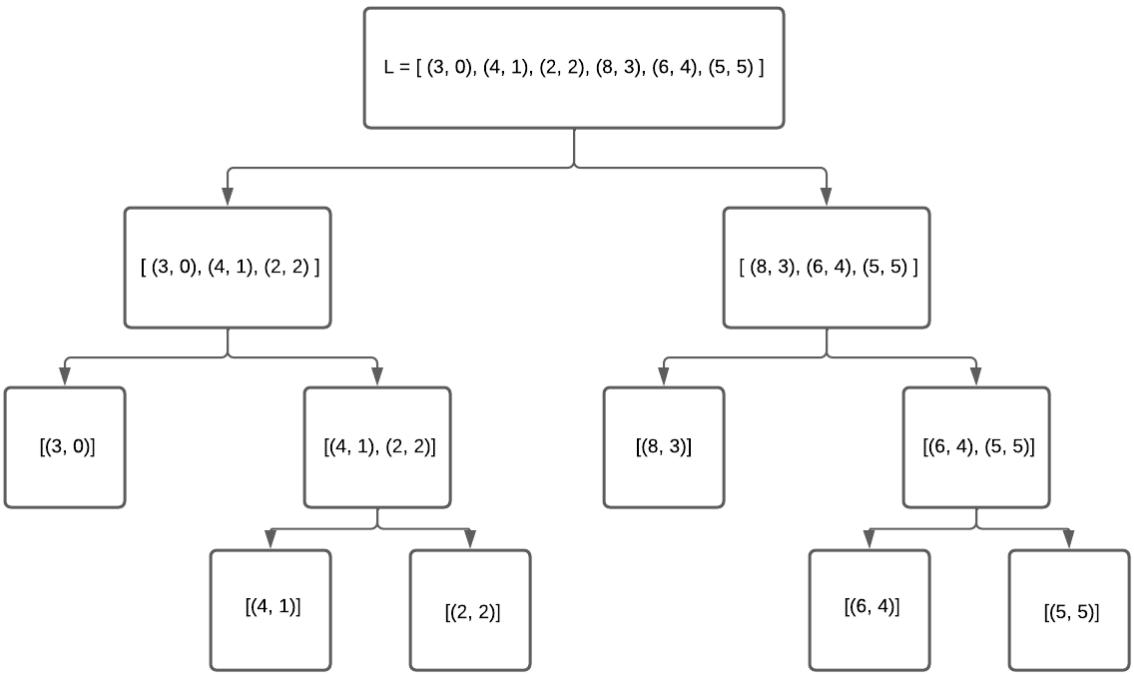
A,3 | B,4 | C,2 | D,8 | E,6 | F,5

Según el algoritmo definido, inicialmente se tienen las siguientes listas

L = [(3, 0), (4, 1), (2, 2), (8, 3), (6, 4), (5, 5)] donde la tupla representa (ranking_año_pasado, ranking_actual)

S = [(A, 0), (B, 0), (C, 0), (D, 0), (E, 0), (F, 0)] donde la tupla representa (nombre, cantidad_superados)

El algoritmo divide la lista L en dos mitades, hasta llegar al caso base de un elemento por lista



Finalmente para cada nivel del arbol se comparan las listas, se ordenan y cuentan inversiones:

Nivel 1:

[(4,1)] y [(2,2)] ⇒ [(2,2)] y [(4,1)] Se da una inversión para el B
[(6,4)] y [(5,5)] ⇒ [(5,5)] y [(6,4)] Se da una inversión para el E

S = [(A, 0), (B, 1), (C, 0), (D, 0), (E, 1), (F, 0)]

Nivel 2:

[(3,0)] y [(2,2)] y [(4,1)] ⇒ [(2,2), (3,0), (4,1)] Se da una inversión para el A
[(8,3)] y [(5,5)] y [(6,4)] ⇒ [(5,5), (6,4), (8,3)] Se da dos inversión para el D

S = [(A, 1), (B, 1), (C, 0), (D, 2), (E, 1), (F, 0)]

Nivel 3:

[(2,2), (3,0), (4,1)] y [(5,5), (6,4), (8,3)] ⇒ [(2,2), (3,0), (4,1),(5,5), (6,4), (8,3)] No hay inversiones

S = [(A, 1), (B, 1), (C, 0), (D, 2), (E, 1), (F, 0)]

Finalmente el resultado es

A ⇒ 1 (1)
B ⇒ 2 (1)
C ⇒ 3 (0)
D ⇒ 4 (2)
E ⇒ 5 (1)
F ⇒ 6 (0)

Parte 2: Ciclos negativos

Tomando como entrada de nuestro problema un grafo ponderado con valores enteros (positivos y/o negativos) dirigido donde un nodo corresponde al punto de partida, queremos conocer si existe al menos un ciclo negativo y en caso afirmativo mostrarlo en pantalla.

1- Proponer una solución al problema que utiliza programación dinámica. Incluya relación de recurrencia, pseudocódigo, estructuras de datos utilizadas y explicación en prosa.

Para analizar si existe al menos un ciclo negativo en el grafo del problema utilizando programación dinámica, podemos utilizar el algoritmo de Bellman-Ford.

Realizando una iteración principal más con el algoritmo, luego de haber encontrado el camino mínimo, podremos determinar si existe o no algún ciclo negativo en el grafo. Si en esa última iteración extra, cambia el mínimo de al menos un nodo, entonces se puede afirmar que el grafo tiene un ciclo negativo.

Relación de recurrencia

$$minPath('s', j) = 0$$

$$minPath(n_{i,0}) = \infty \ con \ n_i \neq s$$

$$minPath(n_i, j) = min\left\{ \begin{array}{l} minPath(n_i, j-1) \\ min\{minPath(n_x, j-1) + w(n_x, j-1)\} \ con \ n_x \epsilon \ pred(n_i) \end{array} \right\}$$

Pseudocódigo

```
Desde l=0 a n-1
    OPT[l][0] = 0

Desde v=0 a n-1
    OPT[0][v] = +∞

Desde l=0 a n-1 // max longitud del camino
    Desde v=1 a n // nodo
        OPT[l][v] = OPT[l-1][v]
        Por cada p predecesor de v
            si OPT[l][v] > OPT[l-1][p] + w(p,v)
                OPT[l][v] = OPT[l-1][p] +w(p,v)

Retornar OPT[n-1,n]
```

OPT[l][v] = camino mínimo de “s” al nodo nv con máxima longitud l

w(p,v) = costo de ir desde p hasta v
n = cantidad de nodos del grafo
l = longitud del camino

Estructuras de datos utilizadas

OPT[l][v] = matriz de nxn

2-Analice la complejidad temporal y espacial de su propuesta.

Llenar la matriz con infinitos tiene una complejidad algorítmica de O(V).
Obtener las aristas del grafo es un O(V + E).
Iterar V veces por todas las aristas para completar la matriz con los pesos (operaciones O(1) sobre cada una de las aristas) genera un O(V*E).
Luego la iteración extra para analizar la existencia de un ciclo negativo agregaría un O(E).

Por lo tanto, la complejidad algorítmica final sería O(V*E)

3- Programe la solución

```
INFINITO = float('inf')
import sys

def main():
    resultado = encontrar_ciclo_negativo(sys.argv[1], int(sys.argv[2]))
    print(resultado)

def parsearGrafo(tramos):
    grafo = {}
    nodos = []
    recorridos = {}
    for t in tramos:
        data = t.split()[0].split(',')
        if data[0] not in nodos:
            nodos.append(data[0])
            recorridos[data[0]] = ''
        if len(data) == 1:
            grafo['s'] = data[0]
        elif len(data) == 3:
            grafo[data[0]+data[1]] = int(data[2])
            if data[1] not in nodos:
                nodos.append(data[1])
                recorridos[data[1]] = ''
    return grafo, nodos, recorridos

def encontrar_ciclo_negativo(fileName, n):
    l, v = (n, n)
    OPT = [[0 for i in range(l)] for j in range(v + 1)]
```

```
    for l in range(n + 1):
        for v in range(1, n):
            OPT[l][v] = INFINITO

    f = open(fileName, 'r')
    lines = f.readlines()
    grafo, nodos, recorridos = parsearGrafo(lines)
    f.close()

    for l in range(0, n + 1):
        for v in range(1, n):
            OPT[l][v] = OPT[l-1][v]
            for p in range(len(nodos)):
                if OPT[l][v] > OPT[l-1][p] + w(p,v, grafo, nodos):
                    OPT[l][v] = OPT[l-1][p] + w(p,v, grafo, nodos)
                    recorridos[nodos[p]] =  nodos[v]
                    if l == n:
                        return imprimir_ciclo_negativo(recorridos,
nodos[p], grafo)

    return "No existen ciclos negativos en el grafo"

def imprimir_ciclo_negativo(recorridos, nodo_i, grafo):
    nodo_f = ''
    ciclo = ''
    nodo_anterior = nodo_i
    costo = 0
    while(nodo_f != nodo_i):
        ciclo += recorridos[nodo_anterior]
        costo += grafo[nodo_anterior + recorridos[nodo_anterior]]
        nodo_anterior = recorridos[nodo_anterior]
        nodo_f = nodo_anterior
    return "Existe al menos un ciclo negativo en el grafo. {} -> costo:
{} ".format(ciclo, costo)

def w(p,v, grafo, nodos):
    tramo = nodos[p] + nodos[v]
    try:
        return grafo[tramo]
    except KeyError:
        return INFINITO

main()
```

4- Determine si su programa tiene la misma complejidad que su propuesta teórica.

En el comienzo de la función *encontrar_ciclo_negativo()* es donde se llenan las matrices con infinito, esto es O(V).
Luego, las aristas del grafo se obtienen en la función *parsear_grafo().* Siendo esta una operación O(V + E) siendo V la cantidad de nodos y E la cantidad de aristas.
También, como se dijo anteriormente, iterar V veces por todas las aristas para completar la matriz con los pesos genera un O(V*E). Esto se realiza en *encontrar_ciclo_negativo().*
Podemos decir como conclusión, que nuestro algoritmo cumplió la complejidad esperada.

Parte 3: Un poco de teoría

1- Hasta el momento hemos visto 3 formas distintas de resolver problemas. Greedy, división y conquista y programación dinámica.

a) **Describa brevemente en qué consiste cada una de ellas.**
b) **¿Cuál es la mejor de las 3? ¿Podría elegir una técnica sobre las otras?**

a. Los tres tipos de algoritmos tienen una cosa en común: dividen el problema global en subproblemas más pequeños para trabajar y luego unir resultados. Los del tipo greedy, por un lado, trabajan los subproblemas aislados, buscando siempre la solución óptima local, y están ordenados de tal manera que la solución óptima de un subproblema va a llevar a la solución óptima del siguiente. Si el problema lo permite, se llega finalmente a una solución óptima. Este tipo de algoritmo es generalmente más fácil de programar, y de seguir, ya que es bastante directo, pero requiere un tipo de solución particular para que resulte la solución óptima, y no siempre es tan sencillo de probar que ésta es la óptima global.

Los algoritmos de división y conquista, por otro lado, trabajan el problema de adentro para afuera, dividiendo en subproblemas cada vez más pequeños, hasta un caso base de salida, y luego analizan hacia afuera los subproblemas entre sí para llegar a la solución final. Estos algoritmos también tienen la ventaja de que son simples de entender y seguir, aunque el análisis de la complejidad puede tornarse complejo debido al uso de la recursión.

Finalmente, los algoritmos de programación dinámica consisten en dividir el problema en subproblemas, al igual que los otros tipos, pero con una diferencia de jerarquía. Esto significa que el resultado de un subproblema, puede ser utilizado para el subproblema mayor. Esto lo hace guardando los resultados conseguidos, para no tener que volver a calcularlos nuevamente. Este tipo de algoritmos es más difícil de desarrollar, ya que hay que tener muy claro cuales son los subproblemas y cómo se relacionan entre sí, y es el tipo más rápido a la hora de resolver problemas de optimización.

b. Es muy difícil elegir uno de los tres como “mejor” que los otros. Hay que analizar el problema antes de elegir un tipo de algoritmo para utilizar por sobre los otros. Habría que analizar tanto la complejidad temporal como la espacial, el tamaño del problema, los recursos disponibles, entre otras cosas. En un problema de optimización pequeño, por ejemplo, se puede utilizar algoritmos greedy (si el problema se puede resolver de manera óptima de esta manera) o de programación dinámica, dependiendo la memoria disponible a la hora de ejecutarse.

Es decir, en términos generales, no existe un tipo de algoritmo “mejor” que otro, sino que es necesario analizar el problema en cuestión, y las condiciones de desarrollo antes de tomar una decisión. En ese caso sí, es posible que exista una mejor alternativa entre los algoritmos posibles.

2- Un determinado problema puede ser resuelto tanto por un algoritmo Greedy, como por un algoritmo de Programación Dinámica. El algoritmo greedy realiza N^3 operaciones sobre una matriz, mientras que el algoritmo de Programación Dinámica realiza N^2 operaciones en total, pero divide el problema en N^2 subproblemas a su vez, los cuales debe ir almacenando mientras se ejecuta el algoritmo. Teniendo en cuenta los recursos computacionales involucrados (CPU, memoria, disco) ¿Qué algoritmo elegiría para resolver el problema y por qué?

En caso de contar con buena memoria y disco, elegiría el algoritmo de programación dinámica, ya que existe una diferencia enorme entre N^2 y N^3 en términos de complejidad temporal, pero en caso de tener un sistema más lento, iría por el de tipo greedy, para evitar problemas de disco que pueden llegar a interrumpir la ejecución del programa.

REENTREGA

Las correcciones que se hicieron previas a esta entrega fueron:

1- Se propone utilizar bellman ford con una iteración adicional para poder encontrar si existe ciclos negativos, es correcto aunque no se explica cuántas iteraciones se deben hacer, se explica vagamente porque con una iteración extra se encuentra si hay ciclos negativos.

2- Se presenta la relación de recurrencia, es correcta aunque debe corregir el valor de minPath(ni,0).

3- Se adjunta pseudocódigo se realiza un análisis de complejidad en base a los nodos y a las aristas, como es la complejidad (en el peor caso) dependiendo de los nodos solamente?. ¿Cómo es la complejidad espacial dependiendo de los nodos solamente?

Comentarios programa

4- No respeta el pasaje de parámetros del programa ya que hay que pasarle la cantidad de nodos. el código no funciona se probó con el siguiente ejemplo

A
A,B,2
B,C,3
C,D,4
D,E,-7
E,A,-3
devuelve que no existen ciclos negativos

CORRECCIONES

“Se propone utilizar bellman ford con una iteración adicional para poder encontrar si existe ciclos negativos, es correcto aunque no se explica cuántas iteraciones se deben hacer, se explica vagamente porque con una iteración extra se encuentra si hay ciclos negativos.”

1- Proponer una solución al problema que utiliza programación dinámica. Incluya relación de recurrencia, pseudocódigo, estructuras de datos utilizadas y explicación en prosa.

Para analizar si existe al menos un ciclo negativo en el grafo del problema utilizando programación dinámica, podemos utilizar el algoritmo de Bellman-Ford.

Se realiza una iteración principal más con el algoritmo, ya que si hay un ciclo negativo en un grafo, incluso después de realizar N iteraciones (siendo N el número de vértices que hay en un grafo) podemos actualizar **el valor del costo del camino mínimo** (si se hicieron las N iteraciones el costo no podrá variar). Esto sucede porque para cada iteración, atravesar el ciclo negativo siempre disminuye el costo de la ruta más corta.

“Se presenta la relación de recurrencia, es correcta aunque debe corregir el valor de minPath(ni,0).”

2- Proponer una solución al problema que utiliza programación dinámica. Incluya relación de recurrencia, pseudocódigo, estructuras de datos utilizadas y explicación en prosa.

$$minPath('s', j) = 0$$

$$minPath(n_{i,0}) = \infty \ con \ n_i \neq s$$

$$minPath(n_i, j) = min\left\{ \begin{array}{l} minPath(n_i, j-1) \\ min\{minPath(n_x, j-1) + w(n_x, j-1)\} \ con \ n_x \epsilon \ pred(n_i) \end{array} \right\}$$

donde j = número de nodos del grafo

ni = nodo inicial, se busca el camino mínimo que recorra todo el grafo.

“Se adjunta pseudocódigo se realiza un análisis de complejidad en base a los nodos y a las aristas, como es la complejidad (en el peor caso) dependiendo de los nodos solamente?. ¿Cómo es la complejidad espacial dependiendo de los nodos solamente?”

Llenar la matriz con infinitos tiene una complejidad algorítmica de O(V).
En el peor de los casos, donde todos los nodos están conectados entre sí, obtenerlos tendrá una complejidad de O(V + V*(V-1))
Iterar V veces por todas las aristas para completar la matriz con los pesos (operaciones O(1) sobre cada una de las aristas) genera un O(V * V*(V-1)).
Luego la iteración extra para analizar la existencia de un ciclo negativo agregaría un O(V *(V-1)).
Siendo la complejidad final O(V*V*(V-1)).

“No respeta el pasaje de parámetros del programa ya que hay que pasarle la cantidad de nodos. el código no funciona se probó con el siguiente ejemplo”

Se modificó el código, el mismo está en bellman-ford.py.

4- Determine si su programa tiene la misma complejidad que su propuesta teórica.

La implementación propuesta cumple con la complejidad algorítmica esperada.
Completar la matriz de distancias con infinitos es O(V):

```
for v in nodos:
       dist[v] = INFINITO
```

Luego, la función obtener aristas es un O(V + E)

```
def obtener_aristas(fileName):
   f = open(fileName, 'r')
   tramos = f.readlines()
   f.close()
   grafo = {}
   nodos = []
   for t in tramos:
       data = t.split()[0].split(',')
       if data[0] not in nodos:
           nodos.append(data[0])
       if len(data) == 1:
           grafo['origen'] = data[0]
       elif len(data) == 3:
           grafo[data[0] + data[1]] = int(data[2])
           if data[1] not in nodos:
               nodos.append(data[1])
   return grafo, nodos
```

Después iteramos V veces sobre todas las aristas: O(V * E)

```
      for tramo, peso in aristas.items():
           if tramo != 'origen':
               v = tramo[0]
               w = tramo[1]
               if dist[v] + peso < dist[w]:
                   padre[w] = v
                   dist[w] = dist[v] + peso
```

Por eso cumple con la complejidad esperada: O(V * E)