Trabajo Práctico n.º 2 - Teoría de Algoritmos

$Cuatrimestre$	1
$Año$	2022
$Estudiantes$	Aldana Rastrelli (98.408) Guido Ernesto Bergman (104.030) Ivan Loyarte (97.213) Ramiro Andrés Fernández Márquez(105.595) Santiago Ronchi (101.043)
$Fecha\ de\ Entrega$	25 de Abril de 2022
$N^\circ\ de\ entrega$	2da
$Grupo$	Grupo 45

Índice

1. Parte 1: Un evento exclusivo

💡 Todos los años la asociación de un importante deporte individual profesional realiza una preclasificación de los n jugadores que terminaron en las mejores posiciones del ranking para un evento exclusivo. En la tarjeta de invitación que enviarán suelen adjuntar el número de posición en la que está actualmente y a cuantos rivales superó en el ranking (únicamente entre los invitados). Contamos con un listado que tiene el nombre del jugador y la posición del ranking del año pasado. Ese listado está ordenado por el ranking actual.

▼ Ejemplo

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

A → Ranking actual 1 → superó a 1 entre los preclasificados (C)
B → Ranking actual 2 → superó a 1 entre los preclasificados (C)
C → Ranking actual 3 → superó a 0 entre los preclasificados (-)
D → Ranking actual 4 → superó a 2 entre los preclasificados (E y F)
E → Ranking actual 5 → superó a 1 entre los preclasificados (F)
F → Ranking actual 6 → superó a 0 entre los preclasificados (-)

En este caso el problema debería retornar:

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

1.1.

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

La solución por fuerza bruta propuesta consiste en recorrer los jugadores, comparando para cada uno su posición en el año anterior con la de todos los que se encuentran después de él en la lista. Cada vez que la posición en el año anterior de un jugador sea mayor que la que tenia otro que actualmente se encuentra después en la lista, se puede decir que el primero

supero al segundo en el rankig del año actual. En ese caso, se incrementara un contandor de la cantidad de jugadores a los que superó.

Entendiendose como n a la cantidad de jugadores en el ranking del año actual, esta resolución tiene una complejidad temporal $O(n^2)$, ya que para cada uno de los n jugadores tendrá que recorrer todos los siguientes, que son a lo sumo $n-1$. La complejidad espacial es $O(n)$ ya que es necesario guardar un contador de la cantidad de jugadores que superó cada uno de los n jugadores.

1.2.

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

La solución propuesta es similar al mergesort: consiste en dividir el arreglo de los jugadores por mitades, ordenarlas recursivamente respecto de la posición del año anterior y mergear las mitades ordenadas. Lo que se agrega, es que se guardará en la clase `Jugador` la cantidad de **inversiones** que tuvo cada uno, entendiendose que una inversión ocurre cuando un jugador que tenia una posición peor que otro en el ranking del año anterior lo supero en el ranking del año actual. En el algoritmo, estos casos ocurren cuando la posicón del año anterior de alguien de la `listaIzq` es mayor a la de alguien de la `listaDer`.

Pseudocódigo

```
# Jugador que tiene atributos: nombre, posición anterior e inversiones.
jugadores_año_pasado = [ Jugador(i) for i in len(input) ]

mergesort_and_count(jugadores_año_pasado)
imprimir_inversiones(jugadores_año_pasado)

def mergesort_and_count(jugadores):
      si len(lista) < 2:
        devolver lista
      medio = len(lista) // 2
      # Llamamos de forma recursiva partiendo a la lista en dos mitades
      izq = mergesort_and_count(lista[:medio])
      der = mergesort_and_count(lista[medio:])
      devolvemos merge_and_count(izq,der)

def merge_and_count(listaIzq, listaDer):
      seteamos índices i,j = 0
      Creamos una lista resultado = []

      mientras que i < len(listaIzq) y j < len(listaDer):
```

```
        jugador_a = listaIzq[i]
        jugador_b = listaDer[j]
        si (jugador_a.posicion_anterior < jugador_b.posicion_anterior):
          resultado.append(jugador_a)
          i += 1
        de lo contrario:
          resultado.append(jugador_b)
          jugador_a.inversiones += 1
          j += 1

      resultado += listaIzq[i:]
      resultado += listaDer[j:]

      return resultado
```

1.3.

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

Tomando *n* como la cantidad de jugadores en el año actual.

Sean

$$T(n) = \underbrace{a \times T(\frac{n}{b})}_{\text{a subproblemas con n/b elementos}} + \overbrace{f(n)}^{función\ de\ merge}$$

$$a \geq 1\ y\ b > 1\ (ctes)$$

La Ecuación de Recurrencia queda:

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + C * n & \text{if } n > 2 \\ C & \text{if } n = 2 \end{cases}$$

Ya que en cada paso, el problema se separa en 2 subproblemas y la cantitidad de elementos se reduce a la mitad.

Por lo tanto $a = b = 2$ y $f(n) = n$

📌 Como $\Theta(n^{log_b a}) = \Theta(n^1) = \Theta(n) \Rightarrow f(n)$ está acotada inferior y superiormente por $\Theta(n^{log_b a})$. En consecuencia, $T(n)$ está acotada inferior y superiormente por $nlogn$:

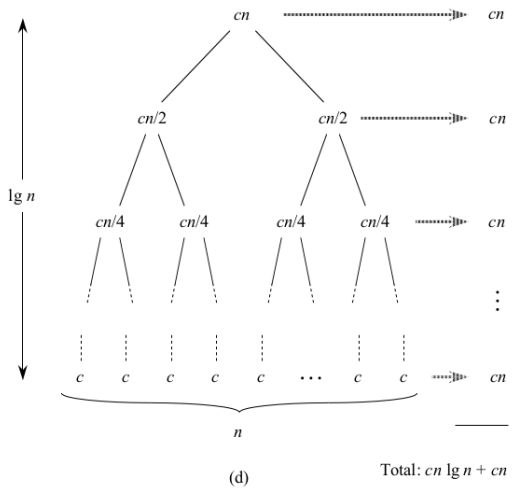
$$T(n) = \Theta(nlogn)$$

1.4.

💡 Realizar el análisis de complejidad temporal desenrollando la recurrencia

Se puede ver que en cada nivel el tamaño de los subproblemas se divide a la mitad pero se multiplica por 2 la cantidad de subproblemas:

- En el primer nivel hay un subproblema de tamaño n
- En el segundo nivel hay 2 subproblemas de tamaño $n/2$
- En el tercer nivel hay 4 subproblemas de tamaño $n/4$
- En el último nivel hay n subproblemas de tamaño 1



Definiendo s como el tamaño de los subproblemas y m como la cantidad de subproblemas, se puede acotar el tiempo de que se tarda en resolver el nivel i por:

$$T = c \times (s \times m) = c \times (\frac{n}{2^i} \times 2^i) = cn$$

Así, cada nivel requiere un tiempo de cn.

Hay $logn$ niveles, ya que se nececita dividir el problema $log_2 n$ veces para reducirlo desde un tamaño n a un tamaño 2. Por lo tanto, el tiempo total requerido es $cn \cdot logn$, es decir que la complejidad temporal es: $O(nlogn)$

1.5.

> 💡 Analizar la complejidad espacial basándose en el pseudocódigo.

Se usan las estructuras `jugadores_año_pasado` y `jugadores_año_actual`, que guardan una entrada por cada jugador, es decir son $O(n)$.

Las listas `izq` y `der` tienen a lo sumo $n/2$ elementos y la lista `resultado` tiene a lo sumo n, por lo que su complejidad espacial es $O(n)$.

Dado que el resto de las variables utlizadas son $O(1)$, la complejidad del algoritmo es $O(n)$.

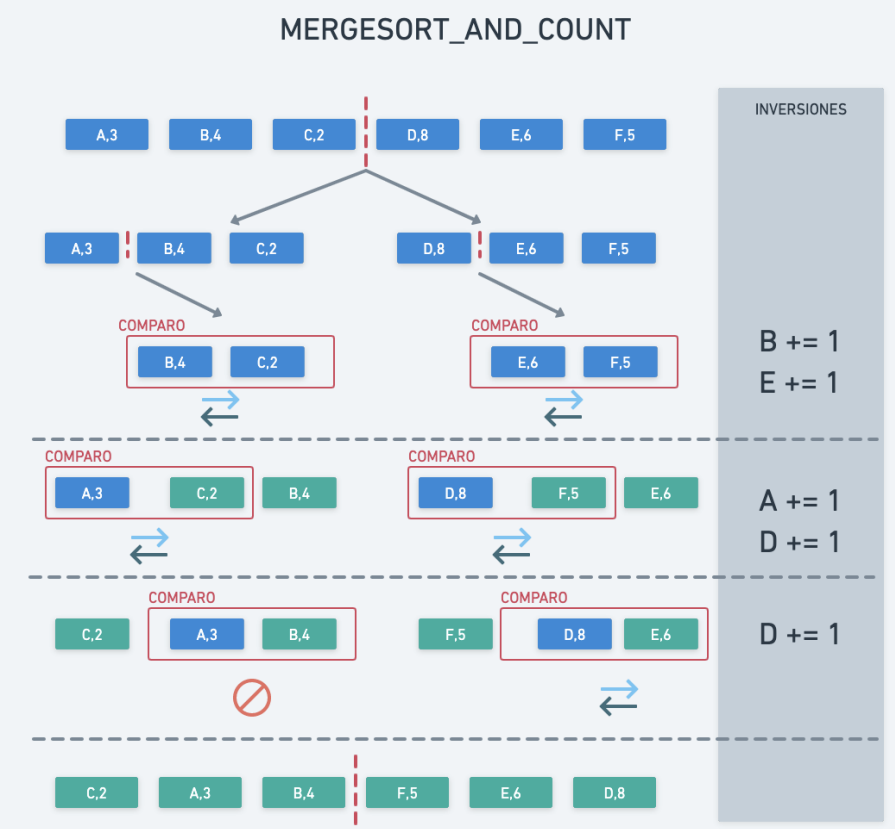
1.6.

> 💡 Dar un ejemplo completo del funcionamiento de su solución

A partir del siguiente ejemplo:

```
A → 1, B → 2, C → 3, D → 4, E → 5, F → 6
```

Se obtiene:



NOMBRE	INVERSIONES TOTALES
A	1
B	1
C	0
D	2
E	1
F	0

2. Parte 2: Ciclos negativos

💡 La detección de ciclos negativos tiene una variedad de aplicaciones en varios campos. Por ejemplo en el diseño de circuitos electrónicos VLSI, se requiere aislar los bucles de retroalimentación negativa. Estos corresponden a ciclos de costo negativo en el grafo de ganancia del amplificador del circuito. 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.

💡 **Formato de los archivos:**

El programa debe recibir por parámetro el path del archivo donde se encuentra el grafo. El archivo con el grafo es un archivo de texto donde la primera línea corresponde al nodo inicial. Luego continúa con una línea por cada eje direccionado del grafo con el formato: ORIGEN,DESTINO,PESO.

Ejemplo: “grafo.txt”

```
B
D,A,-2
B,A,3
D,C,2
C,D,-1
B,E,2
E,D,-2
A,E,3
...
```

Debe resolver el problema y retornar por pantalla la solución.

En caso de no existir ciclos negativos: “No existen ciclos negativos en el grafo”

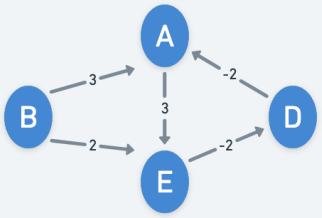
En caso de existir ciclos negativos: “Existe al menos un ciclo negativo en el grafo. A,E,D → costo: -1”

1.

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

El algoritmo propuesto consiste en una variante del algoritmo de $Bellman - Ford$. Este algoritmo permite detectar si existen ciclos negativos a los cuales se puede llegar desde un nodo t en un grafo.

Lo primero que se hará es iterar n veces buscando el valor de la distancia desde el nodo $origen$ hasta todos los otros nodos. Esto se hará tomando a los n nodos en cada iteración como “origen” y calculando a partir de la distancia del mismo, la distancia de sus aristas. Luego de n iteraciones, si no existiesen ciclos negativos, las distancias finales son los costos de los caminos mínimos desde el nodo de origen hasta cada uno de los otros.



Iteración	A	B	D	E	Nodo origen relativo
Inicialización	inf	0	inf	inf	B
0	3	0	inf	2	B
	3	0	inf	2	A
	3	0	0	2	E
	-2	0	0	2	D
1	-2	0	0	2	B
	-2	0	0	1	A
	-2	0	-1	1	E
	-3	0	-1	1	D
2	-3	0	-1	1	B
	-3	0	-1	0	A
	-3	0	-2	0	E
	-4	0	-2	0	D
3	-4	0	-2	0	B
	-4	0	-2	-1	A
	-4	0	-3	-1	E
	-5	0	-3	-1	D
Distancias Finales	-5	0	-3	-1	

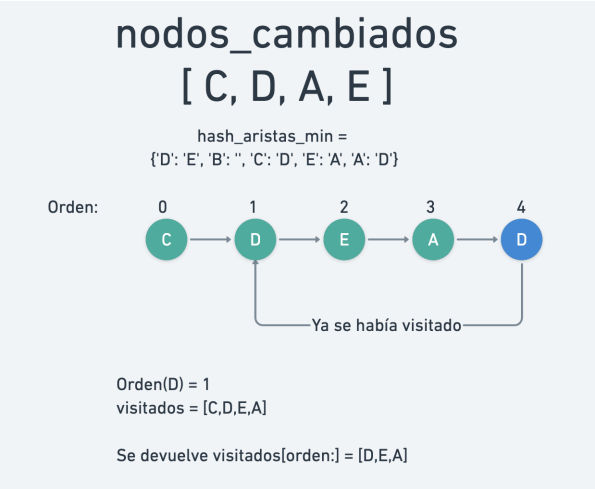
Ejemplo

Ahora, si se hiciese una iteración adicional (total de iteraciones $n + 1$), y el grafo no tuviese ciclos negativos, el valor de las distancias no cambiaría. Este cambiaria solamente en caso de que haya ciclos negativos. Lo anterior se debe a que si hubo algún cambio, significaría que se encontró

Distancias Finales	-5	0	-3	-1	
	A	B	C	D	Nodo origen relativo
Iteración extra	-5	0	-3	-1	B
	-5	0	-3	-2	A
	-5	0	-4	-2	E
	-6	0	-4	-2	D

algún camino con
al menos n aristas que permite llegar a t con menor costo que el mejor camino encontrado usando una arista menos; y en un grafo con n nodos, si hay un camino con al menos n aristas necesariamente contiene un ciclo. Este ciclo debe tener entonces costo negativo, ya que si tuviera costo positivo ó 0 , se podría haber encontrado un camino en iteraciones anteriores que tenga como máximo el mismo costo que el nuevo camino que se acaba de encontrar.

En el ejemplo se ve que los valores sí van a cambiar con una iteración extra. Lo que se hará es, con esa iteración extra, comparar con el resultado de las n iteraciones y ver qué nodos cambiaron: aquellos que hayan cambiado su valor, serán a los que se puede llegar desde el nodo inicial pasando por un ciclo negativo.
Si no se encuentran diferencias entre las distancias obtenidas con n iteraciones y las obtenidas con $n+1$, no existen ciclos negativos en el gráfico.



Una vez que se encuentran los nodos cuyas distancias cambiaron, recorremos sus aristas hasta encontrar una repetición: ese es el ciclo negativo.Luego, se calcula el costo del mismo y se devuelven ambos.

Estructuras de datos utilizadas

- `hash_aristas_min`: El hash guarda el último nodo a partir del cual se modificó el valor de una distancia. Por ejemplo, $\{D : E\}$ significa que el valor de D se cambió por última vez

desde el nodo E. Esto nos permite recrear luego el camino por el cual podría llegar a generarse un ciclo negativo.

- `distancias`: Es un arreglo que tiene dos elementos:

DISTANCIAS_ANTERIOR	Diccionario en donde se van guardando los valores de las distancias de cada uno de los nodos respecto del nodo origen. Se utiliza para las primeras n iteraciones, quedando las distancias finales almacenadas en esta fila.
DISTANCIAS_ULTIMA	Diccionario en donde se guardan los valores de las distancias de cada uno de los nodos en la iteración $n + 1$ (la iteración extra para encontrar ciclos negativos).

  Luego se comparan ambos diccionarios y, si se encuentran diferencias, se procede a buscar el ciclo negativo.

- `ciclo`: lista que guarda los nodos que participan en un ciclo negativo (de encontrarse).

Relación de recurrencia

$$M[i, v] = min(M[i-1, v], min_{w \epsilon V}(M[i-1, w] + costoVW))$$

Siendo $M[i, v]$ la distancia para llegar al nodo v en la iteración i y $costoVW$ el costo de la arista que va desde v a w.

i debe ser mayor a cero y para $i = 0$ se define:

$$M[0, t] = 0$$

$$M[0, v] = \infty$$

Siendo t el nodo inicial y v cualquier otro nodo que no sea ese.

Pseudocódigo

```
# Pre: recibe el grafo y el nodo incial
def encontrar_ciclos_negativos(grafo, nodo_nicial)
""" Devuelve una lista con los nodos del ciclo negativo y su costo.
Si no tiene ciclo negativo, devuelve [], None """

  n = cantidad de nodos en el grafo
  hash_aristas_min = diccionario
  distancias_anterior = diccionario { nodos: {aristas: inf}}
  distancias_ultima_iteracion = diccionario

  for i = 1..n
    for nodo v de grafo.nodos:
      costo_vertice = distancias_anterior[v]
```

```
      if costo aún no asignado:
        continuar

      for arista in grafo.aristas[v]:
        peso = peso(v,arista) # variable auxiliar
        costo_min_anterior = distancias_anterior[v][arista]
        costo_nuevo = costo_min_anterior + peso

      if costo_min_anterior > costo_nuevo:
        actualizar la distancia entre v y arista en distancias_anterior con costo-nuevo
        hash_aristas_min[v] = arista

    # iteración extra para encontrar ciclos negativos:
    distancias_ultima_iteracion = distancias_anterior

    for nodo v de grafo.nodos:
      costo_vertice = distancias_ultima_iteracion[v]
      if costo aún no asignado:
        continuar

      for arista in grafo.aristas[v]:
        peso = peso(v,arista) # variable auxiliar
        costo_min_anterior = distancias_ultima_iteracion[v][arista]
        costo_nuevo = costo_min_anterior + peso

      if costo_min_anterior > costo_nuevo:
        actualizar la distancia entre v y arista en distancias_anterior con costo-nuevo
        hash_aristas_min[v] = arista

  nodos_cambiados = obtener nodos cambiados entre distancias_anterior y distancias_ultima_iteracion

  if no cambiaron los nodos:
    devolver ciclo vacío y costo None

  ciclo_negativo = encontrar_ciclo_en(nodos_cambiados,con hash_aristas_min)
  costo = calcular_costo(ciclo_negativo, grafo)

  return ciclo_negativo, costo
```

```
def encontrar_ciclo_en(nodos_cambiados,hash_aristas_min):
  nodos_visitados = []
  orden = {}

  n0 = primer nodo de nodos_cambiados

  Loop:
    si n está en nodos_visitados:
      devolver nodos_visitados + n

    De lo contrario, guardar n en nodos_visitados
    n0 = arista mínima de n guardada en hash_aristas_min

  devolver []
```

```
def calcular_costo(ciclo_negativo, grafo):
  costo = 0
  por nodo in ciclo_negativo, hasta la primer repetición:
    costo += grafo.peso(nodo, nodo_siguiente)

  return costo
```

2.

💡 Analice la complejidad temporal y espacial de su propuesta.

$Temporal$

$O(n * m)$, con n cantidad de nodos y m cantidad de aristas

La función encontrar_ciclos_negativos tiene complejidad $O(n * m)$ ya que las líneas que inicializan los diccionarios tienen complejidad $O(n)$ mientras que los ciclos *for* tienen complejidad $O(n * m)$: recorren n veces todas las aristas que salen de todos los nodos, que en total son m.

Las funciones encontrar_ciclo_en y calcular_costo tienen complejidad temporal $O(n)$, ya que en en el peor de los casos tienen que recorrer todos los nodos.

Por lo tanto, la complejidad temporal del algoritmo es $O(n * m)$.

$Espacial$

$O(n)$

El tamaño de las estructuras de datos utilizadas (hash_aristas_min, distancias y ciclo) es, en el peor de los casos, proporcional a la cantidad de nodos que tiene el grafo.

3.

💡 Programe la solución

Se adjunta el código en la entrega. Instrucciones para correrlo:

```
>>python main.py test.txt
```

4.

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

$Temporal$

$O(n * m)$

El algoritmo de Bellman-Ford tiene una complejidad de $O(n * m)$ mientras que el armado del ciclo negativo (postorder) tiene una complejidad de a lo sumo $O(n)$. Por lo tanto, la complejidad temporal final es de $O(n * m)$.

$Espacial$

$O(n)$

Las estructuras utilizadas en el programa mantienen la complejidad temporal de las del pseudocódigo.

3. Parte 3: Un poco de teoría

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

1. Describa brevemente en qué consiste cada una de ellas
2. ¿Cuál es la mejor de las 3? ¿Podría elegir una técnica sobre las otras?
2. Un determinado problema puede ser resuelto tanto por un algoritmo Greedy, como por un algoritmo de División y Conquista. 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?

Pista: probablemente no haya una respuesta correcta para este problema, solo justificaciones correctas

1.

Greedy

La estrategia de programación greedy consiste en dividir al problema en subproblemas y luego resolver cada uno de ellos mediante un criterio de selección greedy. Esto quiere decir que no tiene en cuenta las repercusiones a largo plazo ni las soluciones de otros subproblemas, sino que elige mirando solamente el estado actual y local.

División y Conquista

En la división y conquista se separa recursivamente al problema en subproblemas hasta llegar a un caso base, en el cual el tamaño del subproblema es lo suficentemente chico como para poder resolverlo. Luego se combian las soluciones de los subproblemas para obtener la solción del problema original.

Programación Dinámica

La programación dinámica consiste en separar el problema un suproblemas y “memorizar” el resultado de los que ya fueron resueltos para evitar tener que volver a calcularlo cuando los subproblemas se repiten. Muchas veces esto produce una complejidad temporal menor en comparación con la fuerza bruta pero se sacrifica complejidad espacial.

2.

No hay una técnica superior entre ellas, sino que cada una es utilizada en el contexto de un problema. Las caracteristicas del problema determinan cual o cuales de las estrategias se pueden aplicar para resolverlo. Esto hace que, al no haber una naturaleza que generalice todos los problemas por igual, no haya una elección correcta entre una de esas 3 sin información del problema particular a resolver.

En los casos en los que un problema se pueda resolver con más de una estrategia, se podrá comparar la complejidad temporal y espacial de las estrategias para determinar cual usar.

3.

La elección pasa por si se quiere minimizar el uso de recursos de memoria por parte de la computadora, en cuyo caso convendría elegir el algoritmo Greedy, o si se quiere preservar el menor tiempo de ejecución posible, con menos uso del procesador aunque implique usar más memoria, donde habría que elegir el algoritmo de Programación Dinámica.

Correcciones primera entrega

Nota Entrega: Parte 1: 9

Parte 2: Reentrega

Parte 3: 7

Corrector: Ernesto

comentarios sobre las correcciones

Comentarios Parte 1

Correcta la solución por fuerza bruta y sus complejidades.

El pseudocódigo es muy implementativo, la idea del pseudocódigo es remarcar el algoritmo, copias del estilo deep copy son algo que no atañe al algoritmo en cuestión.

Solución por pseudocódigo correcta y dan una explicación que acompaña al pseudocódigo para que sea más legible.

Utilizan correctamente el teorema maestro.

Dan una explicación correcta con un árbol para la complejidad desenrrollando la recurrencia.

La complejidad espacial es correcta.

Buenisimo el ejemplo! Además de ser correcto es muy bonito.

Comentarios Parte 2

Explicación enreverada pero del algoritmo, no explican como quieren llegar a encontrar el ciclo. (-2)

Muestran las estructuras de datos que utilizan, hay una estructura llamada distancias que es una matriz de 1x2, no entiendo bien su función.

Relación de recurrencia erronea, hay que especificar el paso para que tenga sentido. No se puede calcular la recurrencia "iterativamente". (-2)

El pseudocódigo no se entiende, utiliza una estructura de datos "peso" que no se inicializa en ningun lado. La idea del pseudocódigo es sacarse de encima los problemas de sintaxis de los lenguajes de programación para poder enfocarnos en el algoritmo, pero aca el problema es que no se entiende. (Reentrega)

Comentarios programa

No cumple con el formato, hay que poner el nombre del archivo de prueba dentro del código.

A

A,B,2

J,B,-4

H,K,2

B,D,4

E,H,8

C,F,-3

I,L,3

B,I,-2

C,E,-5

F,G,-3

K,J,0

G,J,2

B,C,2

En este ejemplo el ciclo se puede encontrar como: B,C,F,G,J,B con valor -6.

Pero el programa devuelve "C,B,J,G,F → costo: -6". No es un ciclo válido, solo presenta los nodos que participan

Las complejidades son correctas

Comentarios Parte 3

Correctas las definiciones de los 3 tipos de soluciones.

Buena explicación de por qué no hay un tipo mejor que el otro.

Explicación muy concisa de las cosas a tener en cuenta para cada uno de las soluciones (Greedy vs dinámica)