Trabajo Práctico 2

Teoría de Algoritmos - 1c 2025

La 12 devs

Integrantes:

Franco Bustos 110759

Camila Aleksandra Kotelchuk 110289

Lorenzo Kwiatkowski 110239

Bruno Pezman 110457

Joaquin Miranda Iglesias 97500

Fecha: 28/06/2025

Lenguaje: Python

Entrega: 3

Índice:

Parte 1: La subasta de pizza

Explique brevemente cada una de las soluciones. Defina la función costo y límite cuando corresponda.

Backtracking:

Primero determinamos todas las posibles ubicaciones factibles. Para esto, no se considera la ganancia, sólo se considera que si ubicamos a alguien en la mesa, no esté sentado al lado de alguien que no quiere (los llamamos “enemigos”).

Una vez obtenidas las ubicaciones factibles, generamos las rotaciones posibles para cada una y calculamos la ganancia de cada rotación. Durante este proceso se van comparando las ganancias y guardando la que genera la máxima ganancia hasta el momento.

Una vez terminado, queda la máxima ganancia de todas las ubicaciones factibles, junto a la ubicación que produce ese resultado.

Branch and bound:

Dado que tengo n invitados y m porciones de pizza, con n > m, creo una matriz de ofertas oferta[i][j], donde cada entrada representa el monto que el invitado “i” y está dispuesto a pagar por la porción “j”:

Invitado	Porción 1	Porción 2	Porción 3	Porción 4	Porción 5
1	25	20	11	29	23
2	33	21	16	12	45
3	22	15	56	32	34
4	30	18	43	23	21
5	12	46	89	22	23
6	18	44	32	34	29

La pizza se encuentra en una mesa circular, por lo que cada porción tiene una posición fija, y sus vecinos están dados por la estructura de la mesa. Cada invitado puede acceder a una única porción, y existen restricciones de enemistad: ciertos pares de invitados no pueden sentarse en porciones adyacentes.

El objetivo es asignar porciones a invitados de manera tal que:

- Ningún par de enemigos está sentado en porciones adyacentes.
- Se maximice la ganancia total (suma de ofertas aceptadas).

Para resolver este problema, implementamos un algoritmo de Branch and Bound con búsqueda en profundidad (DFS). Representamos las decisiones parciales como nodos de un árbol, donde cada nodo indica una asignación parcial de porciones a invitados. Cada vez que asignamos una porción a un invitado, descendemos un nivel en el árbol. Antes de expandir un nodo, verificamos:

1. Que la asignación no siente a un invitado junto a un enemigo.
2. Que la **cota superior** (función de costo estimada) desde ese nodo sea mayor que la mejor solución encontrada hasta el momento.

La función de costo se define como: funcion_costo = ganancia_actual + mejor estimación posible entre los invitados no asignados. Y se calcula considerando, para cada porción libre, la mejor oferta posible entre los invitados restantes, sin todavía chequear restricciones.

Si alguna de estas condiciones no se cumple, se **poda** la rama correspondiente.

Una vez que se asignan las m porciones (se alcanza un nodo hoja), se evalúa la ganancia total de la solución.

Dar el pseudocódigo y estructuras de datos a utilizar en ambas propuestas.

Estructuras de datos:

- **Backtracking**:
  - **oferta** (dict[str, list[str]]): diccionario donde cada clave es un invitado y el valor una lista con las ofertas para cada porción.
  - **enemigos** (dict[str, dict[str]]): diccionario donde cada clave es un invitado y el valor un diccionario con sus enemistades.
  - **invitados** (list[str]): lista con los nombres de todos los invitados.
  - **disponibles** (set[str]): set con los invitados aún disponibles para asignar durante la solución parcial.
  - **parcial** (list[str]): lista de invitados ya asignados durante la solución parcial.
  - soluciones (list[list[str]]): lista que almacena todas las posibles soluciones completas y válidas.
  - **rotaciones_vistas** (set[tuple[str]]): set que guarda las posibles permutaciones ya evaluadas para evitar repeticiones.
  - **mejor_rotacion** (tuple[str]): tupla que almacena la rotación de mejor ganancia hasta el momento.
  - **mejor_ganancia** (int): entero que guarda la suma de las ganancias de la mejor solución lograda.
- **Branch and bound**:
  - **oferta[i][j]**: matriz de enteros que representa cuánto ofrece el invitado “i” por la porción “j” ej.: oferta[1][1] = 25
  - **enemigos**: Conjunto de tuplas(i, j) que indica los invitados que no pueden sentarse juntos ej.: ((1, 2), (4, 5), (3, 7))
  - **adyacente[j] = {(j - 1 + m), (j + 1)}**: conjunto de los índices de las porciones adyacentes a la porción j en la mesa circular. Ejemplo: si m = 6 y j = 0, entonces 0 tiene como adyacentes a 5 y a 1 por lo que (j - 1 + m) sería (0 - 1 + 6) = 5 y (j + 1) sería (0 + 1) = 1, siendo adyacente[0] = {5, 1}.

Pseudocódigo - Backtracking

Sea **m** la cantidad de porciones de pizza a repartir.

Sea **invitados** el conjunto de personas que ofertan por las porciones.

Sea **enemigos** un diccionario donde enemigos[A] es el conjunto de personas enemistadas con A.

Sea **ofertas** un diccionario donde ofertas[A][i] representa la oferta de A por la porción i (posición en la ronda).

Sea **parcial** una lista que representa una solución parcial (secuencia de invitados seleccionados hasta el momento).

Sea **disponibles** conjunto de invitados que aún no fueron incluidos en la solución parcial.

def backtrack(m, invitados, enemigos):

Inicializar lista de **soluciones**

def bt(parcial, disponibles):

Si **parcial** tiene **m** elementos: #Caso solución completa

Si el primero y el último son enemigos → **descartar (return)**

Guardar y agregar al conjunto de **soluciones**

Terminar (return)

Para cada **invitado** en **disponibles**: #Caso solución parcial

Si **parcial** no está vacío y hay enemistad con el último → **saltar (continue)**

Agregar invitado a **parcial**

Llamar **recursivamente** con el invitado **quitado** de **disponibles**

Quitar **invitado** de **parcial**

Llamar **bt([], invitados)**

Devolver soluciones

Sea **solución** una solución particular válida

Sea **mejor_ganancia** la mayor ganancia encontrada hasta el momento

Sea **mejor_rotación** la rotación con mayor ganancia encontrada hasta el momento

Sea **rotación** una rotación circular generada en base a alguna solución válida.

Sea **rotaciones** una lista que contiene todas las rotaciones generadas por una solución válida.

Sea **`rotacionesVistas`** un set() donde se guardarán todas las rotaciones generadas por cada una de las soluciones (sin repetir)

Sea **ganancia** la suma de cada uno de los lugares ocupados por las personas en una rotación

ObtenerMaximaGananciaYMesaMaxima(solucionesFactibles, ofertas)

Inicializar mejor_ganancia = -1, mejor_rotación vacía

Para cada solución en solucionesFactibles :

// Verificar si ya se ha calculado alguna rotación de esta solución

si solución en rotacionesVistas:

continuar // Si ya se ha procesado esta solución, no procesamos sus rotaciones

rotaciones = generar_rotaciones(solucion) // Generar todas las rotaciones posibles de la solución

Para cada rotacion en rotaciones:

// Calcular la ganancia de la rotación

ganancia = 0

Para cada **persona** en rotacion:

ganancia += ofertas[**persona**][posicion de la persona en rotacion]

Si ganancia > mejor_ganancia :

mejor_ganancia = ganancia

mejor_rotación = rotacion

// Después de calcular la ganancia, agregar la rotación al conjunto de vistas

rotacionesVistas.add(rotacion)

return mejor_ganancia, mejor_rotación

---

def main():

Leer archivo de ofertas y obtener (**m**, **ofertas**)

Leer archivo de enemigos y obtener el diccionario **enemigos**

Obtener la lista de **invitados** a partir de las claves de ofertas

Llamar **backtrack** con (**m**, **invitados**, **enemigos**) → **soluciones**

Inicializar **mejor_ganancia** = -1 , **mejor_distribucion** = lista vacía

mejor_ganancia, mejor_distribucion = ObtenerMaximaGananciaYMesaMaxima(**soluciones**)

Mostrar resultados finales:

Si **mejor_distribucion** no está vacía:

Imprimir **ganancia máxima** y la lista de **invitados ganadore**

Si no:

Imprimir que no se encontró **ninguna** asignación válida

Pseudocódigo - Branch and bound:

Sea **m** la cantidad de porciones de pizza a repartir.

Sea **asignaciones_parciales** las asignaciones de invitados a porciones hechas hasta el momento

Sea **porciones_libres** las porciones que todavía no fueron asignadas a ningún invitado

Sea **invitados_libres** los invitados a los que todavía no se les asignó una porción de pizza

Sea **ganancia_actual** la ganancia obtenida hasta el momento con las asignaciones ya realizadas

```
def branch_and_bound(asignaciones_parciales, porciones_libres, invitados_libres, ganancia_actual):
   si longitud(asignaciones_parciales) == m:
      actualizar_mejor_solucion(ganancia_actual)

   por cada porcion en porciones_libres:
      por cada invitado en invitados_libres:
         si es_valida(porcion, invitado, asignaciones_parciales):
            nueva_ganancia = ganancia_actual + oferta[invitado][porcion]
            cota = calcular_cota_superior(nueva_ganancia, porciones_libres - {porcion}, invitados_libres - {invitado})

            si cota > mejor_ganancia_global:
               asignaciones_parciales[porcion] = invitado
               branch_and_bound(asignaciones_parciales, porciones_libres - {porcion}, invitados_libres - {invitado}, nueva_ganancia)
            quitar la asignación de la porción actual de asignaciones_parciales

def es_valida(porcion, invitado, asignaciones_parciales):
```

```
    por cada vecino en adyacente[porcion]:
        si vecino pertenece a asignaciones_parciales:
            otro_invitado = asignaciones_parciales[vecino]
            si (invitado, otro_invitado) pertenece a enemigos o (otro_invitado, invitado) pertenece a enemigos:
                devolver Falso
    devolver Verdadero


def calcular_cota_superior(ganancia_actual, porciones_libres, invitados_libres):
    mejores_ofertas = []
    por cada porcion en porciones_libres:
        mejor_oferta = max(oferta[i][porcion] para i en invitados_libres)
        mejores_ofertas.append(mejor_oferta)
    return ganancia_actual + sum(mejores_ofertas)
```

Realice el análisis de complejidad temporal y espacial de ambas soluciones. Compararlas.

- **Backtracking:**

Análisis de Complejidad (Caso general: n > m)

Contexto

- Hay n invitados y m porciones de pizza (con n>mn > mn>m).
- Se deben seleccionar m invitados y ubicarlos en la mesa de manera que no se sienten junto a sus enemigos (ni en posiciones consecutivas ni en forma circular).
- Luego, se busca la mejor ganancia posible considerando rotaciones de la disposición.

Complejidad temporal de Backtracking

1. **Total de combinaciones posibles (sin restricciones)-**

   Primero se elige un subconjunto de m invitados de entre n, y luego se permutan sus ubicaciones:

$$\binom{n}{m} \cdot m! = \frac{n!}{(n-m)!}$$

- ❖ En el peor caso, sin restricciones, el algoritmo genera $\frac{n!}{(n-m)!}$ combinaciones completas.
- ❖ Cada paso de backtracking realiza chequeos de enemigos entre personas adyacentes (lineales y circulares), lo cual es O(1).
- ❖ Luego, la complejidad temporal de generar todas las ubicaciones factible es:

$O(\frac{n!}{(n-m)!})$

2. Cálculo de ganancias (con rotaciones)

Por cada solución válida:

- Se generan m rotaciones.
- Cada rotación se evalúa en O(m)

Entonces la Complejidad Temporal resultante final es: $O(\frac{n!}{(n-m)!} * m^2)$.

3. Complejidad Temporal Total

De los ítems 1 y 2 vemos que el algoritmo termina teniendo una complejidad total de $O(\frac{n!}{(n-m)!} * m^2)$, ya que el cálculo de ganancias tiene una complejidad temporal mayor.

Complejidad Espacial de Backtracking

- Al hacer el backtracking, se almacenan soluciones válidas de longitud m. Eso resulta:

  $O(\frac{n!}{(n-m)!} * m)$.

- Respecto al cálculo de ganancias, las rotaciones se almacenan en un conjunto rotacionesVistas, que puede llegar a contener hasta:

  $O(\frac{n!}{(n-m)!} * m)$.

Como vemos, la complejidad espacial total termina siendo $O(\frac{n!}{(n-m)!} * m)$.

Branch and bound:

- ❖ Sea **n** la cantidad total de invitados.
- ❖ Sea **m** la cantidad total de porciones.

Complejidad temporal de B&B

El algoritmo aplica **poda** cuando:

1. La asignación genera un conflicto entre enemigos adyacentes.

2. La **cota superior** es menor o igual a la mejor solución encontrada.

Estas podas eliminan gran parte del espacio de búsqueda, pero **en el peor caso**, si no hay enemigos y las ofertas están todas parejas (sin oportunidad de poda), se recorren todos los nodos igual que en el caso sin poda.

Entonces, el costo total en tiempo queda acotado por:

$$O\left(\binom{n}{m} . m!\right)$$

Donde “n sobre m“ es la cantidad de formas de elegir m invitados entre n posibles, sin importar el orden y, una vez que decidí qué m invitados participan entre los n, me falta decidir en qué orden se asignan a las m porciones.

Complejidad espacial de B&B

El espacio necesario está dado principalmente por:

- Los m niveles del árbol;
- asignaciones_parciales: hasta m elementos;
- porciones_libres: hasta m elementos;
- invitados_libres: hasta n elementos;
- oferta[i][j]: matriz de n x m;
- enemigos: depende de la entrada, por ejemplo k pares;
- adyacente[j]: lista de tamaño m donde cada entrada tiene 2 vecinos

En total, la complejidad espacial es:

$$O(n . m + m + n + k) = O(n . m)$$

Brinde un ejemplo simple paso a paso del funcionamiento de las soluciones.

Backtracking:

Datos:

-Porciones: m = 4

-Invitados: Ana, Beto, Carla, Damián

-Ofertas:

INVITADO	PORCION 1	PORCION 2	PORCION 3	PORCION 4
Ana	9	1	1	1
Beto	1	9	1	1
Carla	1	1	9	1
Damián	1	1	1	9

Enemigos:

- **Carla y Damián**

Simulación paso a paso:

BACKTRACK

Estado inicial:
parcial = []
disponibles = {Ana, Beto, Carla, Damián}

Paso 1 – Elijo Ana:
parcial = [Ana]
disponibles = {Beto, Carla, Damián}

Paso 2 – Desde Ana, intento:
-válido- Beto:
parcial = [Ana, Beto]
disponibles = {Carla, Damián}

Paso 3 – Desde Beto, intento:
-válido- Carla:
parcial = [Ana, Beto, Carla]
disponibles = {Damián}

Paso 4 – Desde Carla, intento:
-inválido- Damián es enemigo directo → descartado

-back- Vuelvo atrás a parcial = [Ana, Beto]

Paso 3 – Desde Beto, intento:
-válido- Damián:
parcial = [Ana, Beto, Damián]
disponibles = {Carla}

Paso 4 – Desde Damián, intento:
-inválido- Carla es enemigo directo → descartado

-back- Vuelvo atrás a parcial = [Ana, Beto] → luego a [Ana]

Paso 2 – Desde Ana, intento:
-válido- Carla:
parcial = [Ana, Carla]
disponibles = {Beto, Damián}

Paso 3 – Desde Carla, intento:
-válido- Beto:
parcial = [Ana, Carla, Beto]
disponibles = {Damián}

Paso 4 – Desde Beto, intento:
 -válido- Damián:
 parcial = [Ana, Carla, Beto, Damián]

Verificación final:

- Carla y Damián → en 1 y 3 → no vecinos
- Ana y Damián → no enemigos
  -válida- => Solución válida

[SOLUCIÓN VÁLIDA] Ana, Carla, Beto, Damián

-back- Backtrack a parcial = [Ana, Carla]

Paso 3 – Desde Carla, intento:
 -inválido- Damián es enemigo directo → descartado

-back- Backtrack a parcial = [Ana]

Paso 2 – Desde Ana, intento:
 -válido- Damián:
 parcial = [Ana, Damián]
 disponibles = {Beto, Carla}

Paso 3 – Desde Damián, intento:
 -válido- Beto:
 parcial = [Ana, Damián, Beto]
 disponibles = {Carla}

Paso 4 – Desde Beto, intento:
 -válido- Carla:
 parcial = [Ana, Damián, Beto, Carla]

Verificación final:

- Damián y Carla → en 1 y 3 → no vecinos
- Ana y Carla → no enemigos
  -válida- => Solución válida

[SOLUCIÓN VÁLIDA] Ana, Damián, Beto, Carla

Y así paso a paso se construyen todas las soluciones que cumplen con el requisito de no contener enemigos vecinos, resultando asi..

-Soluciones válidas encontradas-

1. Ana, Carla, Beto, Damián
2. Ana, Damián, Beto, Carla
3. Beto, Carla, Ana, Damián
4. Beto, Damián, Ana, Carla

5. Carla, Ana, Damián, Beto
6. Carla, Beto, Damián, Ana
7. Damián, Ana, Carla, Beto
8. Damián, Beto, Carla, Ana

ROTACIONES

-Ganancia por rotación (con ofertas)-

Solución 1: Ana, Carla, Beto, Damián-> ¿Está en rotaciones vistas? NO
Genero rotaciones:

-Rotación 0: Ana, Carla, Beto, Damián → 9 + 1 + 1 + 9 = 20-> máx gan
-Rotación 1: Carla, Beto, Damián, Ana → 1 + 9 + 1 + 1 = 12-> 20 > 12
-Rotación 2: Beto, Damián, Ana, Carla → 1 + 1 + 1 + 1 = 4-> 20 > 4
-Rotación 3: Damián, Ana, Carla, Beto → 1 + 1 + 9 + 1 = 12-> 20 > 12

=> máxima ganancia: 20, combinación: Ana, Carla, Beto, Damián

- MÁXIMA GANANCIA GENERAL: 20, COMBINACIÓN: Ana, Carla, Beto, Damián -

Solución 2: Ana, Damián, Beto, Carla-> ¿Está en rotaciones vistas? NO
Genero rotaciones:

-Rotación 0: Ana, Damián, Beto, Carla → 9 + 1 + 1 + 1 = 12-> máx gan
-Rotación 1: Damián, Beto, Carla, Ana → 1 + 9 + 9 + 1 = 20-> 12 < 20
-Rotación 2: Beto, Carla, Ana, Damián → 1 + 1 + 1 + 9 = 12-> 20 > 12
-Rotación 3: Carla, Ana, Damián, Beto → 1 + 1 + 1 + 1 = 4-> 20 > 4

=> máxima ganancia: 20, combinación: Damián, Beto, Carla, Ana

¿ Máxima ganancia > MÁXIMA GANANCIA GENERAL? NO => No actualiza valor

Solución 3: Beto, Carla, Ana, Damián-> ¿Está en rotaciones vistas? SI
No se generan rotaciones, se ignora solución. (Sol. 2, Rot 2)

Solución 4: Beto, Damián, Ana, Carla-> ¿Está en rotaciones vistas? SI
No se generan rotaciones, se ignora solución. (Sol. 1, Rot 2)

Solución 5: Carla, Ana, Damián, Beto-> ¿Está en rotaciones vistas? SI
No se generan rotaciones, se ignora solución. (Sol. 2, Rot 3)

Solución 6: Carla, Beto, Damián, Ana-> ¿Está en rotaciones vistas? SI
No se generan rotaciones, se ignora solución. (Sol. 1, Rot 1)

Solución 7: Damián, Ana, Carla, Beto-> ¿Está en rotaciones vistas? SI
No se generan rotaciones, se ignora solución. (Sol. 1, Rot 3)

Solución 8: Damián, Beto, Carla, Ana-> ¿Está en rotaciones vistas? SI
No se generan rotaciones, se ignora solución. (Sol. 2, Rot 1)

No hay más soluciones para analizar, entonces devuelve los datos almacenados:

-> MÁXIMA GANANCIA: 20, COMBINACIÓN: Ana, Carla, Beto, Damián <-

Branch and bound:

Datos del problema:

- Invitados: n = 5;
- Porciones: m = 3;
- Invitados disponibles: {1, 2, 3, 4, 5}
- Ya se asignó una porción al invitado 2 con una ganancia de 21 por lo que quedan dos porciones libres y los invitados restantes son {1, 3, 4, 5}.
- Enemigos: (2, 4) y (3, 5)

Distribución en la mesa: como la porción 2 ya fue ocupada por el invitado 2 y la mesa tiene las porciones 1, 2 y 3, la porción 1 es adyacente a 3 y 2; la porción 2 es adyacente a 1 y 3; y la porción 3 es adyacente a 1 y 2.

Matriz de ofertas:

Invitado	Porción 1	Porción 2	Porción 3
1	25	20	11
2	33	21	16
3	22	15	56
4	30	18	43
5	12	46	89

Ya asigné la porción 2 al invitado 2, con una ganancia de **21**. Luego debo calcular la cota superior, es decir, la mejor ganancia posible para las 2 porciones restantes (1 y 3) con los invitados que todavía no fueron asignados (1, 3, 4 y 5):

- Para la porción 1 el invitado 4 ofrece la mejor oferta que es **30**;
- Para la porción 3 el invitado 5 ofrece la mejor oferta que es **89**.

Entonces, cota_superior = 21 (ganancia_actual) + 30 (mejor oferta para la porción 1) + 89 (mejor oferta para la porción 3) = **140**.

Ahora, debo comprobar si se verifican las condiciones pedidas:

- Que no se siente a un invitado junto con uno de sus enemigos:

  Comienzo por la mejor oferta para la opción 1 (invitado 4 con oferta 30): intento asignar la porción 1 al invitado 4, pero como el invitado 4 es enemigo del invitado 2 y la porción 1 es adyacente a la porción 2 **podo** esta rama por enemigo adyacente.

  Sigo con la siguiente mejor oferta para la porción 1 (invitado 1 con oferta 25): como el invitado 1 no es enemigo de nadie que esté sentado en la mesa, esto está permitido.

  Luego, intento asignar la porción 3 al invitado 5 que es la de mayor ganancia y como el invitado 5 no es enemigo de 2, está permitido.

  Entonces, tendría la porción 2 con el invitado 2 (21); la porción 1 con el invitado 1 (25); y la porción 3 con el invitado 5 (89) que da un total de **135**.

También podría probar otra asignación como darle la porción 1 al invitado 3 (22) y la porción 3 al invitado 5 (89), pero en esta caso **podaría** pues 3 y 5 son enemigos y las porciones 1 y 3 son adyacentes.

- Que la cota superior (función de costo) sea mayor a la mejor solución encontrada hasta el momento: por ejemplo si ya tuviéramos una mejor solución de **85** (mejor ganancia encontrada en una rama anterior), como **140 > 85**, no podamos esta rama y la seguimos explorando. Si en algún caso la cota superior es una rama menor o igual a la mejor solución encontrada se podaría dicha rama pues sabemos que de allí no podríamos obtener una mejor solución.

A continuación, utilizando DFS, se sigue asignando las porciones restantes a los invitados según la mejor oferta disponible hasta llegar a un nodo hoja donde se procede a evaluar la ganancia total de la solución: ganancia_total = 135. Además, como este valor es mayor a 85, actualizamos la **mejor solución encontrada** con esta ganancia.

Determine si los programas tienen la misma complejidad que su propuesta teórica.

Backtracking:

Complejidad Temporal:

1. En el código, respecto al backtracking, en cada paso suceden 2 cosas:

a. Si la solución parcial tiene m elementos, se chequea en O(1) si el primero y el último son enemigos. Al momento de agregar la solución, no queda otra que volver a copiar la lista (esto es específico de python, para que no se pierda la referencia al salir de la función), por lo que agregar la solución es O(m).

b. Para cada invitado se chequea enemistad (también en O(1)). Antes de cada llamado recursivo, hay que copiar la lista de invitados removiendo al invitado actual (no podemos modificar la lista sobre la que estamos iterando). La copia es O(n).

Unificando, considerando n (personas) > M (porciones) y que el costo del backtracking es generar todas las permutaciones la complejidad del backtracking será de O(permutaciones)*O(n). Es decir:

$O(\frac{n!}{(n-m)!} * n)$.

2. Respecto al cálculo de la complejidad temporal de Ganancias en el código, para cada solucion se realiza:

a. Chequeo si la solucion esta en rotaciones_vistas: esto es O(1) porque rotaciones_vistas es un set.

b. Generar las rotaciones: para cada posicion dentro de la ubicacion, hay que rotar la lista. Rotar la lista usando deque tiene una complejidad O(m). Luego, generar las rotaciones será O(m^2).

c. Para cada rotacion, se suman las ganancias de todos los integrantes de la mesa. Obtener la ganancia de un integrante es O(1) porque es un diccionario. Sumar todas las ganancias de la rotacion termina siendo O(m). Como tengo O(m) rotaciones para una solucion, esta parte termina siendo O(m^2).

En el peor de los casos hay $\frac{n!}{(n-m)!}$ ubicaciones factibles, por lo que al hacer $O(m^2)$ operaciones por cada solucion, la complejidad temporal total de obtener la ganancia máxima termina siendo:

$O(\frac{n!}{(n-m)!} * m^2)$.

3. Si bien $n > m$, si consideramos que $m^2 >> n$, el código termina teniendo una complejidad temporal total de:

$O(\frac{n!}{(n-m)!} * m^2)$

que es idéntica a la calculada para el pseudocódigo.

Complejidad Espacial:

1. Obtención de las ubicaciones factibles: cada paso del backtracking utiliza las siguientes estructuras.
    a. disponibles (OrderedDict): 'n' personas. Luego, es O(n). Lo mismo con 'disp'.
    b. parcial(lista): 'm' posiciones. Luego, ocupa O(m).

    Como esto se utiliza para $\frac{n!}{(n-m)!}$ soluciones, la complejidad espacial de esta sección termina siendo:

    $O(\frac{n!}{(n-m)!} * n\ )$.

2. Respecto al cálculo de la máxima ganancia, se utilizan las siguientes estructuras:
    a. mejor_rotacion (lista): 'm' posiciones. Luego, ocupa O(m).
    b. rotaciones_vistas (set): 'n! / (n-m)!' posiciones. Cada rotacion tiene 'm' posiciones. Luego, ocupa $O(\frac{n!}{(n-m)!} * m)$.
    c. rotaciones (lista de listas): hay 'm' rotaciones, y cada rotación tiene a su vez 'm' posiciones. Luego, ocupa $O(m^2)$.

    Como para cada solucion genero una lista de rotaciones, termino usando

    $O(\frac{n!}{(n-m)!} * m^2)$ de espacio.

3. Por lo tanto, la complejidad espacial total termina siendo $O(\frac{n!}{(n-m)!} * m^2)$, que es $O(m)$ veces mayor a la calculada para el pseudocódigo.

Branch and bound:

Complejidad Temporal:

Estructuras de datos del código:

- ofertas[i][j]: matriz de tamaño n × m (cuánto ofrece el invitado i por la porción j)
- restricciones: diccionario de n claves (strings), cada una con un set de hasta n-1 invitados

En el código, la funcion branch_and_bound explora las combinaciones de asignar m invitados a m porciones. Se realiza una búsqueda DFS con poda y en cada nivel k se prueban los invitados todavía no asignados (como no se permiten repeticiones, se generan permutaciones de m invitados entre n).

Por otro lado en cada nodo, cada vez que se prueba una asignación parcial, se verifica si es válida (recorre m porciones asignadas para ver si hay enemigos sentados en posiciones adyacentes); se construye la cota superior (recorre todos los invitados no asignados en O(n), recorre todas las porciones no asignadas en O(m), y por cada porción encuentra la mejor oferta --> O(n x m)) y se hacen operaciones en O(1) como asignación[k] = i y asignados.add(i).

Todo esto da una complejidad temporal total de:

$$O\left(\binom{n}{m} .m! \ . \ n \ . \ m\right)$$

$$O\left(\binom{n}{m} .m!\right)$$

Mientras que la del pseudocódigo era:

Complejidad Espacial:

Algunas de las estructuras utilizadas son: la matriz ofertas de n x m O(n . m), el diccionario de restricciones con n claves y n o menos enemigos O(n^2), la lista de asignación de largo m O(m) y el set de asignamos de tamaño como mucho m O(m). Es por esto que la complejidad espacial total es: O(n^2 + n . m)

Mientras que la del pseudocódigo era: O(n . m)

Parte 2: El traslado de información secreta

Propuesta

Generamos un grafo G=(V,E) dirigido en el que los nodos representan las ciudades, y las aristas, las conexiones aéreas entre ellas. Para modelar la restricción de que como máximo s agentes pasen por una misma ciudad, dividimos cada ciudad en dos nodos: uno de entrada C_{IN} y uno de salida C_{OUT}, conectados entre sí por una arista de capacidad S. Así, cada ciudad puede ser atravesada por a lo sumo S agentes.

Luego, agregamos al grafo dos nodos especiales: uno fuente F y uno sumidero T. Desde F, agregamos una arista de capacidad 1 para cada ciudad donde se encuentra un agente. Para cada vuelo entre dos ciudades A y B, agregamos dos aristas:

- $A_{OUT} \rightarrow B_{IN}$
- $B_{OUT} \rightarrow A_{IN}$

Ambas con capacidad infinita (o suficientemente grande), ya que no hay restricción explícita en los vuelos.

Para cada ciudad C que contiene un centro de investigación, agregamos una arista desde C hasta T, con capacidad 1, ya que cada centro recibe solo un agente.

Ejecutamos el algoritmo de Ford-Fulkerson con búsqueda de caminos aumentantes mediante BFS (Edmonds-Karp) sobre este grafo. Si el flujo máximo obtenido es igual a la cantidad de agentes (o centros, que es lo mismo), entonces es posible realizar el envío de información de forma segura.

Para determinar las rutas individuales de cada agente, reconstruimos los caminos desde la fuente F al sumidero T en el grafo residual, siguiendo las aristas que tienen flujo positivo. Cada ruta encontrada representa el camino que un agente debe seguir hasta llegar a un centro de investigación.

Pseudocódigo

```
Funcion encontrar_camino_aumentante(capacidades, flujo, s, t)
    Crear diccionario padre con clave s y valor None
    Crear diccionario cuello_botella con clave s y valor infinito
    Crear cola y agregar s

    Mientras la cola no esté vacía
        u = extraer el primer nodo de la cola
        Para cada v en las capacidades de u
            capacidad_residual = capacidades[u][v] - flujo[u][v]
            Si capacidad_residual > 0 y v no está en padre
                Asignar padre[v] = u
                Asignar cuello_botella[v] = mínimo entre cuello_botella[u] y capacidad_residual
                Si v es igual a t
                    Crear lista vacía camino
                    nodo = t
                    Mientras nodo no sea igual a s
                        Agregar nodo a camino
                        nodo = padre[nodo]
                    Agregar s a camino
                    Invertir la lista camino
                    Devolver camino y cuello_botella[t]
                Agregar v a la cola
    Devolver None y 0

Funcion ford_fulkerson(G, s, t)
    Crear diccionario flujo vacío para cada nodo en G
    Para cada nodo u en G
        Para cada vecino v en G[u]
            Inicializar flujo[u][v] = 0
            Inicializar flujo[v][u] = 0

    Crear diccionario capacidades vacío para cada nodo en G
    Para cada nodo u en G
        Para cada vecino v en G[u]
            Asignar capacidades[u][v] = G[u][v]['capacity']

    Inicializar max_flujo = 0

    Mientras True
        camino, cuello = encontrar_camino_aumentante(capacidades, flujo, s, t)
        Si camino es None
            Salir del bucle
        Para i desde 0 hasta longitud(camino) - 2
            u = camino[i]
            v = camino[i + 1]
            flujo[u][v] += cuello
            flujo[v][u] -= cuello
        max_flujo += cuello

    Devolver max_flujo, flujo

armado_red_de_flujo(n, s, ciudades, espias)
    Crear grafo G dirigido
```

```
Definir fuente como "F"
Definir sumidero como "T"
Agregar fuente y sumidero al grafo G

Crear lista de vuelos vacía
Crear conjunto de ciudades vacío
Leer ciudades
   Para cada línea en ciudades
      origen, destino = dividir la línea por coma
      Agregar (origen, destino) a vuelos
      Agregar origen y destino a ciudades

Crear lista de espías y centros vacía
Leer espias
   Para cada línea en espias
      _, ciudad = dividir la línea por coma
      Si el índice es menor a n
         Agregar ciudad a espías
      Sino
         Agregar ciudad a centros

Crear conjunto de ciudades_con_centro con los centros

Para cada ciudad en ciudades
   Si la ciudad está en ciudades_con_centro
      Agregar nodo ciudad al grafo
   Sino
      Agregar nodos ciudad_in y ciudad_out al grafo
      Agregar borde de capacidad s entre ciudad_in y ciudad_out

Para cada (origen, destino) en vuelos
   Si origen está en ciudades_con_centro
      origen_nodo = origen
   Sino
      origen_nodo = origen_out

   Si destino está en ciudades_con_centro
      destino_nodo = destino
   Sino
      destino_nodo = destino_in

   Agregar borde con capacidad infinita entre origen_nodo y destino_nodo
   Agregar borde con capacidad infinita entre destino_nodo y origen_nodo

Para cada espía en espías
   Si espía no está en ciudades_con_centro
      Agregar borde de capacidad 1 entre fuente y espía_in
   Sino
      Agregar borde de capacidad 1 entre fuente y espía

Para cada centro en centros
   Agregar borde de capacidad 1 entre centro y sumidero

Devolver grafo G, fuente, sumidero
```

```
Armado_de_ruta(G, flujo, fuente, sumidero)
   rutas = lista vacía
   flujo_copia = copia profunda de flujo

   Para cada vecino en G[fuente]:
      Si flujo_copia[fuente][vecino] <= 0:
         Continuar

      ciudades_ruta = lista vacía
      actual = vecino
      flujo_copia[fuente][vecino] -= 1
      prev_ciudad = None

      Mientras actual ≠ sumidero:
         Si actual es fuente o sumidero:
            ciudad = None
         Sino si actual termina en "_in" o "_out":
            ciudad = parte antes de "_"
         Sino:
            ciudad = actual

         Si ciudad no es None y ciudad ≠ prev_ciudad:
            ciudades_ruta.agregar(ciudad)
            prev_ciudad = ciudad

         Para cada siguiente en G[actual]:
            Si flujo_copia[actual][siguiente] > 0:
               flujo_copia[actual][siguiente] -= 1
               actual = siguiente
               Romper

      rutas.agregar(ciudades_ruta)

   Retornar rutas

Traslado(n, s, ciudades, espias)
   (G, fuente, sumidero) = Armado_red_de_flujo(n, s, ciudades, espias)
   (flujo_max, flujo) = Ford_fulkerson(G, fuente, sumidero)

   Si flujo_max ≠ n:
      Imprimir "Es imposible lograr el objetivo"
      Retornar

   rutas = armado_de_ruta(G, flujo, fuente, sumidero)

   Para i desde 0 hasta longitud(rutas) - 1:
      camino = rutas[i]
      Imprimir "Espia ", i + 1, ",",
```

Optimalidad

Supongamos que existe una solución mejor a la asignación de espías a los centros de investigación que NO utilice el modelo de red de flujo máximo propuesto. Esto implicaría que podríamos asignar todos los espías de manera eficiente sin violar las restricciones de capacidad

de las ciudades, sin que ninguna ciudad tenga más de s espías y sin que los espías se "crucen" o utilicen un vuelo más de lo permitido.

El algoritmo de flujo máximo que usamos es capaz de resolver este problema de asignación óptimamente, garantizando la máxima cantidad de espías asignados sin violar ninguna restricción de capacidad ni la condición de que cada espía debe ser trasladado. Si existiera una solución mejor, entonces el flujo máximo calculado por el algoritmo sería menor que n (el número de espías), ya que el flujo máximo es el número de espías que efectivamente podemos trasladar respetando todas las restricciones. Sin embargo, esto contradice nuestra suposición inicial de que todos los espías son asignados correctamente y se cumplen todas las restricciones.
Por lo tanto, la suposición de una solución mejor es falsa y, por ende, la solución es la óptima.

Análisis de Complejidad

Sea n la cantidad de agentes y n la cantidad de centros.
El grafo residual tendrá un vértice más por cada ciudad salvo la que poseen centro.
Por ende si tomamos a cada agente ubicado en una ciudad tendremos 2n vértices y le tenemos que sumar otros n por la cantidad de centros. Entonces V = 3n
Sea E la cantidad de vuelos como ejes siendo esta mayor a V.

Temporal

- Armado de la red: O(V+E)
- Buscar caminos de aumento (BFS): O(V+E)
- Armado de las rutas: O(V+E)
- Flujo máximo: O(E * |C|) si llamamos C a la suma de todas las Ce de los ejes que salen de la fuente.
- El algoritmo de FF terminará en, como mucho, C iteraciones.
- Complejidad total:

$$O(3n + E) + O(3n + E) + O(3n + E) + O(E * |C|) = O(E * |C|)$$

Espacial

- Para armar el grafo residual: O(V + E)
- Diccionarios flujo y capacidades: O(E)
- Estructuras auxiliares (padre, cuello de botella, cola de BFS): O(V)
- Complejidad total: O(3n + E) + O(E) + 3O(3n) = O(3n + E)

Programar la solución. Incluya la información necesaria para su ejecución. Compare la complejidad de su algoritmo con la del programa.

Para poder ejecutar correctamente la solución debe tener instalar la librería networkx. Para hacerlo debe ejecutar en el directorio el siguiente comando:

```
$ pip install networkx
```

En cuanto a la complejidad resultante no difiere de la explicada en pseudocódigo dado que se implementaron las mismas estructuras para resolver el problema.

¿Es posible expresar su solución como una reducción polinomial? En caso afirmativo explique cómo y en caso negativo justifique su respuesta.

Si, dado que podemos reducir el problema a uno de flujo máximo que pertenece a la clase P y es resoluble en tiempo polinomial para capacidades enteras.

Transformamos la instancia del problema en un grafo dirigido con capacidades donde:

- Cada espía es vertice conectada a la fuente general.
- Cada centro de investigación es un vertice conectado al sumidero general.
- Cada ciudad se transforma (si no es centro) en un par de nodos in y out con una arista de capacidad s.
- Cada vuelo es una arista de capacidad infinita (bidireccional).

Esto se reduce a: Problema de flujo máximo con:

- Capacidad 1 por espía.
- Capacidad 1 por centro.
- Capacidad s por ciudad.
- Capacidad ∞ en los vuelos.

Entonces, si podemos resolver el problema de flujo máximo en redes, también podemos resolver el problema propuesto, ya que este se reduce en tiempo polinomial al primero. En otras palabras, el problema planteado no es más difícil que el problema de flujo máximo; de hecho, es al menos tan difícil como este, lo cual justifica que usar dicha reducción es una forma válida y eficiente de resolverlo.

Parte 3: Las 2 jornadas de capacitación

Demostrar que dada una posible solución que nos brindan, se puede fácilmente determinar si se puede cumplir o no con la tarea solicitada.

Sea **m** la cantidad de tipos de profesionales.
Sea **n** la cantidad de equipos de trabajo.
Restricciones:

- Sea **p** la máxima cantidad de personas por jornada
- No puede pasar que todos los miembros de un equipo de trabajo asista todos juntos en un mismo día

Sea **sc** la solución a la tarea solicitada. Será una tupla que contiene dos conjuntos, una con los invitados del día 1 y otra con los del día 2.

```
certificador(m, n, sc, p, equipos):
   dia1, dia2 = sc

   si tamaño(dia1) > p o tamaño(dia2) > p:                O(1)
      retornar falso
   si tamaño(dia1) > m o tamaño(dia2) > m:                O(1)
      retornar falso

   para cada invitado en dia1:                            O(m)
      si invitado está en dia2:
```

```
        retornar falso

    para cada equipo en equipos:
        si todos los miembros del equipo están en dia1:          O(n*m)
            retornar falso
        si todos los miembros del equipo están en dia2:
            retornar falso

    retornar verdadero
```

Hemos hecho un certificado que determina en tiempo polinomial si se puede cumplir con la tarea solicitada.

Demostrar que si desconocemos si es posible lograr una solución, es difícil poder afirmar que existe. Utilizar para eso el problema "Set splitting problem" (suponiendo que sabemos que este es NP-C).

"Set splitting problem": Dada una familia F de subconjuntos de un conjunto finito S, ¿existe una partición de S en dos subconjuntos S1 y S2 tales que ninguno de los elementos de F esté completamente en S1 o S2?

Llamemos al problema JDC (Jornadas De Capacitación). Queremos determinar que JDC es NP-C. Para ello, procederemos a demostrar que pertenece a NP y a NP-H.

Habiendo resuelto el 1), podemos afirmar que es NP dado que existe una solución que certifique que el problema se puede resolver en tiempo polinomial. Entonces, intentaremos demostrar que SSP("Set splitting problem") es reducible polinomialmente a JDC.

Dada una instancia I de SSP con:

- Sea un conjunto S
- Sea C={C1,C2,...,Ck} una colección de subconjuntos de S.

Transformación:

- Cada elemento de S es un tipo de profesional
- Cada subconjunto Ci representa un equipo de trabajo.
- Ignoramos por ahora la restricción de capacidad p, o la seteamos como p = |S|(sin límite real).

Demostrar que el problema "Set splitting problem" pertenece a NP-C. (Para la demostración se le solicita investigar. La misma se puede realizar partiendo desde "3-SAT" y pasando por "NAE-3-SAT").

Para demostrar que SSP pertenece a NP-C, hay que demostrar que pertenece a NP y NP-H.
Dado un conjunto S = {s1, s2, …, sn}, y otro conjunto C = {c1, c2, …, cm} tal que cada Ci es un subconjunto de S, buscamos verificar si existe una partición de S en A y B tal que cada Ci contiene al menos un elemento en A y otro en B. Suponiendo que recibimos una posible partición de S en A y B, para verificar su validez recorremos cada Ci y para cada uno comprobamos que tenga al menos un elemento en A y al menos uno en B. Como hay M conjuntos y tienen como máximo N elementos, esta verificación tomará un tiempo máximo de O(N.M) por lo que comprobamos que set-splitting se puede verificar en tiempo polinomial por lo tanto pertenece a NP.

Para comprobar que SSP pertenece a NP-H, vamos a reducirlo al problema *NAE 3-SAT*, que es NP-H, siendo este una variante del problema 3-SAT. En el problema *NAE 3-SAT*, se tiene una fórmula con cláusulas de 3 variables, todas positivas (sin negaciones), la cual se cumple si en cada cláusula no todos los valores son iguales (o sea, no todos verdaderos ni todos falsos; tiene que haber al menos uno distinto.

Supongamos que tenemos una fórmula φ de este tipo, con un conjunto de variables X. Por cada cláusula de la forma {x1, x2, x3} que aparece en la fórmula, armamos un conjunto {x1, x2, x3}, y los juntamos todos en una colección C. Así armamos una instancia del problema Set Splitting: el conjunto de elementos es X, y queremos dividir X en dos grupos de forma tal que cada conjunto de C tenga al menos un elemento en cada grupo.

La idea clave es que asignar valores booleanos a las variables es lo mismo que dividir el conjunto X en dos partes: las que son verdaderas van a un lado, las que son falsas al otro.

Una cláusula de la fórmula se cumple en NAE 3-SAT si no todos sus valores son iguales —y eso pasa si, al dividir las variables en dos grupos, el conjunto correspondiente a esa cláusula no está todo en un solo grupo. O sea, si se "separa" correctamente.

Por eso, si podemos dividir el conjunto X de forma tal que todos los conjuntos de C están divididos entre los dos grupos, entonces la fórmula original también se puede satisfacer —y viceversa.

Como la transformación de la fórmula a los conjuntos se puede hacer rápido (en tiempo lineal), y el problema de NAE 3-SAT es NP-HARD, entonces también lo es Set Splitting. Y como además Set Splitting está en NP, concluimos que Set Splitting es NP-C.

Una persona afirma tener un método eficiente para responder si es posible o no cualquiera sea la instancia. Utilizando el concepto de transitividad y la definición de NP-C explique qué ocurriría si se demuestra que la afirmación es correcta.

Un problema es NP-completo si está en NP y es al menos tan difícil como cualquier otro problema en NP, es decir, todo problema en NP puede reducirse a él en tiempo polinomial.

Esto significa que, si encontramos una solución eficiente (en tiempo polinomial) para un problema NP-completo, automáticamente podríamos resolver todos los problemas de NP eficientemente.

La transitividad de las reducciones en teoría de la complejidad dice que: Si A se reduce a B, y B se reduce a C, entonces A se reduce a C. Esto implica que si un problema conocido (digamos 3SAT) se puede reducir a este problema de las jornadas de capacitación, y este problema se puede resolver eficientemente, entonces se podría resolver también 3SAT eficientemente.

Si alguien encuentra un método eficiente (en tiempo polinomial) que responde correctamente para toda instancia de este problema, y si se demuestra que este problema es NP-completo (cosa que es plausible dada su estructura combinatoria y restricciones de partición), entonce se habría demostrado que P = NP, lo cual, todavía no se comprobó, y de comprobarse, revolucionaria la teoría informática.

Un tercer problema al que llamaremos X se puede reducir polinomialmente al problema de “Las 2 jornadas de capacitación”, qué podemos decir acerca de su complejidad?

Podemos decir que, en términos de complejidad, el problema X seguro no es más difícil de resolver que el problema de “Las 2 jornadas de capacitación”. Si el problema de las jornadas se puede resolver en tiempo polinomial, entonces X también. Por ende, concluimos que el problema de “Las 2 jornadas de capacitación” es al menos tan difícil de resolver como X.

Referencias

Demaine, E., Bosboom, J., & Lynch, J. (2019). *6.892: Algorithmic Lower Bounds, Spring 2019 – Problem Set 3 Solutions*. Massachusetts Institute of Technology (MIT).

Correcciones 1era entrega

En la parte 1: Cambios generales.
Branch & Bound

1. Explicación de la solución:

Dado que tengo n invitados y m porciones de pizza, con n > m, creo una matriz de ofertas oferta[i][j], donde cada entrada representa el monto que el invitado “i” y está dispuesto a pagar por la porción “j”:

Invitado	Porción 1	Porción 2	Porción 3	Porción 4	Porción 5
1	25	20	11	29	23
2	33	21	16	12	45
3	22	15	56	32	34
4	30	18	43	23	21
5	12	46	89	22	23
6	18	44	32	34	29

La pizza se encuentra en una mesa circular, por lo que cada porción tiene una posición fija, y sus vecinos están dados por la estructura de la mesa. Cada invitado puede acceder a una única porción, y existen restricciones de enemistad: ciertos pares de invitados no pueden sentarse en porciones adyacentes.

El objetivo es asignar porciones a invitados de manera tal que ningún par de enemigos esté sentado en porciones adyacentes y se maximice la ganancia total (suma de ofertas aceptadas).

Para resolver este problema, implementamos un algoritmo de Branch and Bound con búsqueda en profundidad (B&BDFS). Representamos las decisiones parciales como nodos de un árbol, donde cada nodo indica una asignación parcial de porciones a invitados. En cada nivel del árbol decidimos si asignamos una porción a un invitado válido o si la dejamos sin asignar. Además, apilamos los nodos en una estructura de pila priorizando el orden según su cota superior (función de costo estimada), permitiendo explorar primero los caminos más prometedores. Antes de expandir un nodo, verificamos: que la asignación no siente a un invitado junto a un enemigo y que la **cota superior** desde ese nodo sea mayor que la mejor solución encontrada hasta el momento. La función de costo se define como: funcion_costo = ganancia_actual + mejor estimación posible entre los invitados no asignados. Y se calcula considerando, para cada porción libre, la mejor oferta posible entre los invitados restantes, sin todavía chequear restricciones. Si alguna de estas condiciones no se cumple, se **poda** la rama correspondiente.

Una vez que se asignan las m porciones (se alcanza un nodo hoja), se evalúa la ganancia total de la solución. Si es mejor que la actual, se actualiza la mejor solución global.

2. Estructuras de datos y pseudocódigo:
   - **oferta[i][j]**: matriz de enteros (lista de listas) que representa cuánto ofrece el invitado “i” por la porción “j” ej.: oferta[1][1] = 25
   - **enemigos**: Set de tuplas(i, j) que indica los invitados que no pueden sentarse juntos ej.: ((1, 2), (4, 5), (3, 7))
   - **adyacente[j] = {(j - 1 + m), (j + 1)}**: Diccionario que representa el conjunto de los índices de las porciones adyacentes a la porción j en la mesa circular. Ejemplo: si m = 6 y j = 0, entonces 0 tiene

como adyacentes a 5 y a 1 por lo que (j - 1 + m) sería (0 - 1 + 6) = 5 y (j + 1) sería (0 + 1) = 1, siendo adyacente[0] = {5, 1}.

- **nodo**: diccionario que representa una asignación parcial. Cada nodo en la pila contiene la asignación (lista con los índices de los invitados a cada porción), los invitados_asignados (conjunto de índices de invitados ya asignados), la porcion_actual (índice de la porción que se está procesando), la ganancia_actual (suma de ofertas acumuladas) y la cota_superior (valor de la cota superior desde ese nodo).
- **pila**: lista de los nodos que aún no fueron procesados.
- **mejor_ganancia**: entero que contiene la mejor ganancia encontrada hasta el momento.
- **mejor_asignacion**: lista que guarda la asignación de invitados que obtuvo la mejor ganancia.

Sea m la cantidad de porciones de pizza a repartir.

Sea oferta[i][j] la matriz de enteros con las ofertas de los invitados

Sea enemigos el conjunto de pares (i, j) de invitados que no pueden estar en porciones adyacentes

Sea adyacente[j] el conjunto de índices de porciones adyacentes a la porción j

```
def branch_and_bound():

  crear una pila vacía

  asignaciones_parciales = []
 porciones_asignadas = set()
 invitados_asignados = set()
 ganancia_actual = 0
 porcion_actual = 0

  cota = calcular_cota_superior(ganancia_actual, porciones_asignadas, invitados_asignados)

  apilar (asignaciones_parciales, porciones_asignadas, invitados_asignados, ganancia_actual,
 porcion_actual, cota)

  mejor_ganancia_global = 0
 mejor_asignacion = []

  Mientras la pila no esté vacía:

    Desapilar (asignaciones_parciales, porciones_asignadas, invitados_asignados, ganancia_actual,
 porcion_actual, cota)
    Si porcion_actual == m:

        si ganancia_actual > mejor_ganancia_global:
            mejor_ganancia_global = ganancia_actual
            mejor_asignacion = copia de asignaciones_parciales

        continuar

      Por cada invitado entre 1 y n que no esté en invitados_asignados:

        si es_valida(porcion_actual, invitado, asignaciones_parciales):

          asignaciones_parciales[porcion_actual] = invitado
         porciones_asignadas.agregar(porcion_actual)
         invitados_asignados.agregar(invitado)

          nueva_ganancia = ganancia_actual + oferta[invitado][porcion_actual]
```

nueva_cota = calcular_cota_superior(nueva_ganancia, porciones_asignadas, invitados_asignados)

Si nueva_cota > mejor_ganancia_global:

apilar (copia de asignaciones_parciales, copia de porciones_asignadas, copia de invitados_asignados, nueva_ganancia, porcion_actual + 1, nueva_cota)

asignaciones_parciales[porcion_actual] = vacio
porciones_asignadas.remover(porcion_actual)
invitados_asignados.remover(invitado)

agregado: no asignar dicha porción a nadie

nueva_cota = calcular_cota_superior(ganancia_actual, porciones_asignadas, invitados_asignados)

Si nueva_cota > mejor_ganancia_global:

apilar (copia de asignaciones_parciales, copia de porciones_asignadas, copia de invitados_asignados, ganancia_actual, porcion_actual + 1, nueva_cota)

devolver **mejor_asignacion**, **mejor_ganancia_global**

```
def es_valida(porcion, invitado, asignaciones_parciales):
    por cada vecino en adyacente[porcion]:
     si asignaciones_parciales[vecino] ≠ vacio:
        otro_invitado = asignaciones_parciales[vecino]
        si (invitado, otro_invitado) pertenece a enemigos o (otro_invitado, invitado) pertenece a enemigos:
            devolver Falso
    devolver Verdadero

def calcular_cota_superior(ganancia_actual, porciones_asignadas, invitados_asignados):
    porciones_libres = {j por j en rango(m) si j no está en porciones_asignadas}
    invitados_libres = {i por i en rango(n) si i no está en invitados_asignados}
    mejores_ofertas = []
    Para cada porcion en porciones_libres:
        si hay al menos un invitado libre:
            mejor = max(oferta[i][porcion] para i en invitados_libres)
            mejores_ofertas.agregar(mejor)
    devolver ganancia_actual + suma(mejores_ofertas)
```

3. Análisis de Complejidad:
   - Sea **n** la cantidad total de invitados.
   - Sea **`m`** la cantidad total de porciones de pizza.
   - Sea **oferta[i][j]** la matriz de ganancias de tamaño n x m

Complejidad temporal

El algoritmo aplica **poda** cuando la asignación genera un conflicto entre enemigos adyacentes y la **cota superior** es menor o igual a la mejor solución encontrada. Estas podas eliminan gran parte del espacio de búsqueda, pero **en el peor caso**, si no hay enemigos y las ofertas están todas parejas (sin oportunidad de poda), se recorren todos los nodos igual que en el caso sin poda. Si cada porción pudiera ir a cualquier invitado sin restricciones se podrían hacer hasta n^m asignaciones. Luego, por cada nodo de árbol, es_valida cuesta O(1) ya que la cantidad de adyacentes es constante (en una mesa circular, cada porción tiene 2 porciones adyacentes), y calcular_cota_superior cuesta O((m - k) . (n - k)) donde k es la cantidad de porciones ya asignadas (para cada porción libre buscamos el máximo sobre los invitados libres) que en el peor de los casos es O(m . n) Entonces, el costo total en tiempo queda acotado por: $O(n^m \cdot m \cdot n)$

Complejidad espacial

El espacio necesario está dado principalmente por el tamaño de los elementos almacenados en la pila y la cantidad máxima de elementos simultáneos en la pila.

Cada entrada en la pila contiene:

- asignaciones_parciales: lista de hasta m elementos;
- porciones_asignadas: conjunto de hasta m elementos;
- invitados_asignados: conjunto de hasta n elementos;
- ganancia_actual: número;
- porcion_actual: número;
- cota: número.

Por lo que el tamaño total por entrada en la pila es de: O(m + n)

Cada nivel de la pila corresponde a una porción a decidir, por lo que la profundidad máxima es m. Sin embargo, al considerar no asignar una porción a nadie el árbol puede estar aún más ramificado (la cantidad de nodos que se encuentran de manera simultánea en la pila depende del orden de exploración). En el peor de los casos se almacenan todas las combinaciones parciales activas, pero como se van desapilando nodos una vez explorados, el máximo real de elementos simultáneos en la pila es aproximadamente O(m . n)

En total, la complejidad espacial es:

- Por nodo: O(m + n)
- Máximo de nodos simultáneos en la pila: O(m . n)

$$O(mn(m + n)) = O(m^2n + mn^2)$$

4. Funcionamiento:

Datos del problema:

- Invitados: n = 5;
- Porciones: m = 3;

- Invitados disponibles: {1, 2, 3, 4, 5}
- Ya se asignó la porción 2 al invitado 2 con una ganancia de 21 por lo que quedan dos porciones libres y los invitados restantes son {1, 3, 4, 5}.
- Enemigos: (2, 4) y (3, 5)

Distribución en la mesa: como la porción 2 ya fue ocupada por el invitado 2 y la mesa tiene las porciones 1, 2 y 3, la porción 1 es adyacente a 3 y 2; la porción 2 es adyacente a 1 y 3; y la porción 3 es adyacente a 1 y 2.

Matriz de ofertas:

Invitado	Porción 1	Porción 2	Porción 3
1	25	20	11
2	33	21	16
3	22	15	56
4	30	18	43
5	12	46	89

En mi ejemplo ya se ha asignado la porción 2 al invitado 2, que nos daba una ganancia de 21. Con esto, se construye el primer estado del algoritmo que es el punto de partida de la exploración. Este estado inicial se coloca en una pila, que es la que luego se utiliza para explorar las diferentes posibilidades de asignación.

Paso 1 - estado inicial:
- La porción 2 ha sido asignada al invitado 2;
- Las porciones 1 y 3 están sin asignar;
- La ganancia actual es 21;
- Siguiente paso: asignar la porción 1 (coloco dicho estado en la pila para ser explorado)

```
pila = [
  (
    asignaciones_parciales = [nada, 2, nada],
    porciones_asignadas = {1},
    invitados_asignados = {2},
    ganancia_actual = 21,
    porcion_actual = 0 # es el índice,
    cota = 140 # 21 + 30 + 89 (mejor oferta posible para cada porción e invitado restantes)
  )
]
```

Paso 2 - exploración de la porción 1: se comienza sacando de la pila el estado actual y se evalúan todas las posibles asignaciones válidas de la porción 1 considerando los invitados que aún no fueron asignados y descartando aquellos que generan conflicto. Entre los candidatos posibles el invitado que mejor ganancia ofrece por la porción 1 es el 4 con una ganancia de 30. Sin embargo, este invitado es enemigo del invitado 2 quien ya fue asignado a la porción 2, adyacente a la porción 1. Por lo tanto, asignar la porción 1 al invitado 4 no está permitido y se descarta este camino.
El siguiente mejor candidato es el invitado 1 que ofrece una ganancia de 25. Este a su vez no es enemigo del invitado 2 por lo que la asignación es válida y se procede a generar un nuevo estado con dicha asignación parcial.
- La porción 2 ha sido asignada al invitado 2;
- La porción 1 ha sido asignada al invitado 1;
- La porción 3 está sin asignar;

- La ganancia actual es 21 + 25 = 46;
- Siguiente paso: asignar la porción 3 (empujo dicho estado a la pila para seguir explorándolo)

```
pila = [
  (
    asignaciones_parciales = [1, 2, nada],
    porciones_asignadas = {0, 1},
    invitados_asignados = {1, 2},
    ganancia_actual = 46,
    porcion_actual = 2,
    cota = 135 # 21 + 25 + 89
  )
]
```

Paso 3 - exploración de la porción 3: se comienza sacando de la pila el nuevo estado que corresponde a asignar la porción 3 entre los invitados aún no asignados que son el 3, el 4 y el 5, evaluando cuál de ellos ofrece la mayor ganancia para dicha porción. Quien ofrece la mayor ganancia para dicha porción es el invitado 5 con un total de 89, que como además no tiene conflicto alguno con los invitados ya asignados ni es enemigo de ninguno, la asignación es válida.

- La porción 2 ha sido asignada al invitado 2;
- La porción 1 ha sido asignada al invitado 1;
- La porción 3 ha sido asignada al invitado 5;
- La ganancia total es 21 + 25 + 89 = 135;

```
pila = [
  (
    asignaciones_parciales = [1, 2, 5],
    porciones_asignadas = {0, 1, 2},
    invitados_asignados = {1, 2, 5},
    ganancia_actual = 135,
  )
]
```

Paso 4 - finalización de la exploración: mientras haya estados en la pila, el algoritmo va a seguir funcionando por lo que si existieran otros caminos posibles también serían explorados, siempre descartando aquellos que no pueden superar la mejor solución encontrada hasta el momento (en este caso 135). En el ejemplo descrito se llegó a la solución óptima sin necesidad de seguir explorando. Si en otro camino parcial la cota superior no superará 135, ese camino sería podado y su exploración desestimada.

5. Programa: En el zip

6. Comparación de complejidades:

Sea **n** la cantidad total de invitados y `m` la cantidad total de porciones de pizza.

Complejidad temporal del programa

leer_ofertas: lee un archivo con las ofertas de cada invitado por cada porción en O(n), pero por cada línea convierte los strings a enteros dando un total de **O(n x m)**

leer_restricciones: lee un archivo con las relaciones de enemistad entre los invitados en O(n), pero cada línea puede tener hasta **n – 1** enemigos, siendo en el peor caso **O(n^2)**

adyacentes(j, m): devuelve el conjunto de posiciones adyacentes en la mesa circular en **O(1)** pues son cálculos constantes

es_valida: verifica si es válido asignar una porción a un invitado con respecto a sus enemigos adyacentes. Recorre cada una de las **m** porciones y por cada porción ya asignada chequea si hay conflicto buscando en restricciones en O(1) siendo en el peor de los casos **O(m)**

cota_superior: calcula la mejor ganancia posible para podar el árbol de búsqueda. Como hay n invitados y m porciones, busca en no_asignados en O(n) y en porciones_restantes en O(m) y para cada porción no asignada toma el máximo de los valores ofrecidos, danto un total de **O(m x n)**

banch_and_bound_dfs: intenta asignar cada invitado a cada porción y, dado que cada porción tiene 2 opciones (la asigno o no), y hay m porciones las combinaciones posibles son **O(**$(n+1)^m$**)** [es n + 1 ya que las m porciones pueden asignarse a cualquiera de los n invitados (n) o a nadie (+ 1)]. Además, en cada nodo se evalúan todos los invitados no asignados en O(n), se calcula la cota superior en O(m x n) y se ejecuta es_valida en O(n), dando por cada nodo un costo de **O(n x m)**

Entonces, el costo total en tiempo queda acotado por: $O((n+1)^m \,.\, m \,.\, n)$

Mientras que en el pseudocódigo era: $O(n^m \,.\, m \,.\, n)$

Complejidad espacial de B&B

leer_ofertas: **O(n x m)**

leer_restricciones: **O(**n^2**)**

adyacentes(j, m): **O(1)**

es_valida: **O(1)**

cota_superior: guarda las mejores ganancias de hasta m elementos **O(m)**

banch_and_bound_dfs: es una pila de estados con profundidad m **O(**$(n+1)^m$**)** donde cada estado ocupa **O(m + n)** de espacio por la asignación y el set de asignados

Entonces, el costo total en espacio queda acotado por: $O((n+1)^m \,.\, (m+n))$

Mientras que en el pseudocódigo era: $O(m^2 n + m n^2)$

Backtracking

Explicación de la solución:

Primero vamos generando una a una las distintas **soluciones válidas**. Para esto, no se considera la ganancia, sólo se considera que si **ubicamos** a **alguien** en la **mesa**, no esté sentado **al lado** de **alguien** que **no quiere** (los llamamos "**enemigos**"). Y ante la **obtención** de una **solución válida completa**, se **chequea** que la misma **no** se una "**rotación**" de alguna **solución válida ya encontrada**, y por consecuencia generar las **mismas rotaciones**. Gracias a esto nos ahorramos el calcular la **rotación de ganancia máxima** de soluciones **repetidas**. Teniendo en cuenta esto, una vez obtenidas las **ubicaciones factibles**, generamos las **rotaciones posibles** para **cada una** y calculamos la **ganancia** de **cada rotación**. Durante este **proceso** se van **comparando** las **ganancias** y **guardando** la que genera la **máxima ganancia** hasta el momento.

Una vez terminado, queda la **máxima ganancia** de todas las **ubicaciones factibles**, junto a la **ubicación** que produce ese **resultado**.

ACLARACIÓN:

El **programa** cuenta con **dos manera** de ejecutarlo..

python subasta_bt.py ofertas.txt enemigos.txt -> Modo **normal**, sin prints extra

python subasta_bt.py ofertas.txt enemigos.txt -d -> Modo **debug**, con información más detallada

Pseudocódigo - Backtracking

PSEUDOCÓDIGO:

Sea **m** la cantidad de porciones de pizza a repartir.

Sea **invitados** el conjunto de personas que ofertan por las porciones.

Sea **enemigos** un diccionario donde enemigos[A] es el conjunto de personas enemistadas con A.

Sea **ofertas** un diccionario donde ofertas[A][i] representa la oferta de A por la porción i (posición en la ronda).

Sea **parcial** una lista que representa una solución parcial (secuencia de invitados seleccionados hasta el momento).

Sea **disponibles** un conjunto de invitados que aún no fueron incluidos en la solución parcial.

Sea **vacío** un espacio de asignación sin invitados.

def backtrack(m, invitados, enemigos):

Inicializar lista de **soluciones** y set de **soluciones normalizadas**

def bt(parcial, disponibles):

Si **parcial** tiene **m** elementos: #Caso solución completa

Si el primero y el último no son espacios **vacío** y son enemigos → **descartar (return)**

Normalizar la **solución parcial**

Si no está en el conjunto de **soluciones normalizadas**:

Guardar y agregar al conjunto de **soluciones normalizadas**

Terminar (return)

Para cada **invitado** en **disponibles**: #Caso solución parcial

Si **parcial** contiene asignaciones, el anterior no es un espacio **vacío** y hay enemistad con él → **saltar (continue)**

Agregar invitado a **parcial**

Llamar **recursivamente** con el invitado **quitado** de **disponibles**

Quitar **invitado** de **parcial**

Si **parcial** contiene asignaciones:

Agregar un espacio **vacío** a parcial

Llamar **bt(parcial, disponibles)**

Quitar **invitado** de **parcial**

Llamar **bt([], invitados)**

Devolver soluciones

Sea **solución** una solución particular válida

Sea **mejor_ganancia** la mayor ganancia encontrada hasta el momento

Sea **mejor_rotación** la rotación con mayor ganancia encontrada hasta el momento

def mejor_rotacion_y_ganancia(solución, ofertas):

Inicializar **mejor_ganancia** = -1, **mejor_rotación** vacía

Para cada **rotación** posible de **solución**:

Calcular **ganancia** sumando la oferta de cada invitado en su posición

Si es la mayor hasta ahora → actualizar **mejor_ganancia** y **mejor_rotación**

Devolver **mejor_ganancia** y **mejor_rotación**

Sea **min_i** el indice (**nombre**) de menor valor (**alfabéticamente**)

Sea **indices_válidos** una lista que contendrá los **indices** donde el **invitado** no es **vacío**

def normalizar(solución):

Guardar **indices_válidos** todos los **índices** de **invitados** que no sean **vacíos**.

Si **indices_válidos** no contiene **índices** -> Devolvemos tal cual la **solución** (Todos los invitados **vacíos**)

Buscar el índice del **menor valor** en **indices_válidos** → **min_i**

Rotar **solución** para que empiece en **min_i**

Devolver la **rotación** como **tupla**

---

def main():

Leer archivo de ofertas y obtener (**m**, **ofertas**)

Asignamos ganancia 0 en cualquier posición para **vacío** en **ofertas**

Leer archivo de enemigos y obtener el diccionario **enemigos**

Obtener la lista de **invitados** a partir de las claves de ofertas

Llamar **backtrack** con (**m**, **invitados**, **enemigos**) → **soluciones**

Inicializar **mejor_ganancia** ← -1 , **mejor_distribucion** ← lista vacía

Para cada **solución** en **soluciones**:

(**rotacion**, **ganancia**) ← **mejor_rotacion_y_ganancia**(**solución**, **ofertas**)

Si **ganancia** > **mejor_ganancia**:

mejor_ganancia ← **ganancia**

mejor_distribucion ← **rotacion**

Mostrar resultados finales:

Si **mejor_distribucion** no está vacía:

Imprimir **ganancia máxima** y la lista de **invitados ganadore**

Si no:

Imprimir que no se encontró **ninguna** asignación válida

Estructuras de datos:

- **ofertas (dict[str,list[int]])**: diccionario donde cada clave es el invitado, y el valor la lista con las ofertas para cada porción. Se agrega también una clave llamada "VACIO" para considerar el caso en que no se selecciona un invitado para una porción específica, su clave es una lista de ceros.

- **enemigos (dict[str, set[str]])**: diccionario de sets. Sus claves son los nombres de los invitados, mientras que el valor de cada clave es un set con los enemigos.

- **invitados (list[str])**: lista con los nombres de los invitados

- **mejor_ganancia (int)**: entero que guarda la suma de las ganancias de la mejor solución lograda

- **mejor_distribucion (list[str]):** lista que tiene la rotación de mejor ganancia lograda

- **indices_validos (list[int])**: lista de posiciones no vacías para una dada solución

- **soluciones (list[list[str]])**: lista de listas con las soluciones factibles

- **soluciones_vistas (set(str))**: set con las soluciones que ya fueron generadas. Se utiliza solo para validar en O(1) si una solución ya fue analizada.

- **parcial (list[str])**: lista de invitados ya asignados durante la solución parcial

- **disponibles (OrderedDict[str,None])**: OrderedDict cuyas claves son los invitados disponibles para agregar. Se usa en lugar de un set ya que a diferencia del set, mantiene el orden en que se agregan las claves .

Análisis de Complejidad:

Complejidad basada en el Pseudocodigo. Caso general (n > m):

Contexto

- Hay n invitados y m porciones de pizza (con n>m).
- Se deben seleccionar 'k' invitados ($0 \leq k \leq m$) y ubicarlos en la mesa de manera que no se sienten junto a sus enemigos (ni en posiciones consecutivas ni en forma circular) y considerando que pueden existir posiciones vacías.
- Luego, se busca la mejor ganancia posible considerando rotaciones de la disposición.

Complejidad temporal de Backtracking

1. **Total de combinaciones posibles (sin restricciones)**

   Primero se elige un subconjunto de k invitados de entre n. Además, hay que elegir un subconjunto k de los m lugares en la mesa para ubicarlos, y luego se permutan sus ubicaciones:

   $$\sum_{k=0}^{m} \binom{m}{k}\binom{m}{k} \cdot k! = \sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!}$$ (cantidad de soluciones factibles)

   En el peor caso, sin restricciones, el algoritmo genera $\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!}$ combinaciones completas.

   Cada paso de backtracking realiza chequeos de enemigos entre personas adyacentes (lineales y circulares), lo cual es O(1). Además, se debe "normalizar" la solución parcial. Para esto se busca el mínimo por orden alfabético entre los invitados de la solución, y se reordena la solución, en O(m).

   Luego, la complejidad temporal de generar todas las ubicaciones factibles es: $O(\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!})$.

2. **Cálculo de ganancias (con rotaciones)**

   Por cada solución válida: Se generan m rotaciones y cada rotación se evalúa en O(m)

Entonces la Complejidad Temporal resultante final es: $O(\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!} * m^2)$.

3. Complejidad Temporal Total

De los ítems 1 y 2 vemos que el algoritmo termina teniendo una complejidad total de

$O(\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!} * m^2)$, ya que el cálculo de ganancias tiene una complejidad temporal mayor.

Complejidad Espacial de Backtracking

- Haciendo backtracking, se almacenan soluciones válidas de longitud m. Eso resulta $O(\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!} * m)$**.**
- Respecto al cálculo de ganancias, para cada rotación y solución siempre utilizo las mismas variables. A lo sumo utilizo $O(m)$ de espacio para guardar la máxima rotación hasta el momento, y para guardar la máxima resultante. Luego, termino usando $O(m)$**.**
- Como vemos, la complejidad espacial total termina siendo $O(\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!} * m)$**.**

Determine si los programas tienen la misma complejidad que su propuesta teórica.

Backtracking:

Complejidad Temporal:

1. En el código, antes de iniciar el backtracking, se crea un OrderedDict a partir de la lista de invitados y la variable con la que representamos el dejar vacío un lugar. Eso se hace en O(n), con n invitados. Respecto al backtracking, en cada paso suceden 2 cosas:

a. Si la solución parcial tiene m elementos, se chequea en O(1) si el primero y el último son enemigos. Luego se normaliza la solución (se la reordena a partir del mínimo) en O(m). Se chequea en O(1) si la solución ya fue vista, y en caso de que no, se la agrega. Al momento de agregar la solución, no queda otra que volver a copiar la lista (esto es específico de python, para que no se pierda la referencia al salir de la función), por lo que agregar la solución es O(m). Luego, el costo temporal de esta parte es O(m).

b. Para cada invitado se chequea enemistad (también en O(1)). Antes de cada llamado recursivo, hay que copiar el OrderedDict de invitados (disponibles) removiendo al invitado actual (no podemos modificar el OrderedDict sobre el que estamos iterando). La copia es O(n).

Unificando, considerando n (personas) > m (porciones) y que el costo del backtracking es generar todas las permutaciones la complejidad del backtracking será de O(permutaciones)*O(n).

La cantidad de permutaciones, considerando que dejar todos los lugares vacíos también puede ser una solución válida, será la siguiente (explicada durante el análisis del pseudocódigo):

$$\sum_{k=0}^{m} \binom{m}{k}\binom{m}{k} \cdot k! = \sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!}.$$

Luego, la complejidad temporal del cálculo de las soluciones factibles será:

$O(n \cdot \sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!})$, que es O(n) veces mayor a la vista en el pseudocódigo. Esto se debe a que al momento de hacer la llamada recursiva, hay que copiar el OrderedDict eliminando al

invitado de la estructura (y esto se termina haciendo ‘n’ veces) y no fue considerado en el pseudocódigo.

2. Respecto al cálculo de la complejidad temporal de Ganancias en el código, para cada solución se generan las m rotaciones y se calcula la ganancia para cada una de ellas. El costo para una rotación es el siguiente:

a. Generar una rotación: se rota a la vez que se copia, esto es O(m).
b. Cálculo de ganancia de la rotación: Se recorre cada elemento dentro de la rotación, sumando los valores de cada posición de la mesa. Obtener el valor de la oferta es O(1) para cada posición. Como son m posiciones, termina siendo O(m).

A partir de lo desarrollado, como tengo ‘m’ rotaciones posibles, la complejidad temporal para una solución será de O(m^2). Resta extender esto para todas las soluciones.

En el peor de los casos hay $\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!}$ ubicaciones factibles, por lo que al hacer $O(m^2)$ operaciones por cada solución, la complejidad temporal para obtener la ganancia máxima termina siendo:

$$O(m^2 \cdot \sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!}).$$

3. Finalmente, si bien $n > m$, es razonable suponer que $m^2 >> n$. Luego, la complejidad temporal total resulta:

$O(m^2 \cdot \sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!})$, que es idéntica a la calculada para el pseudocódigo.

Complejidad Espacial:

1. Obtención de las ubicaciones factibles: cada paso del backtracking utiliza las siguientes estructuras.
   a. disponibles (OrderedDict): ‘n’ personas. Luego, es O(n).
   b. parcial(lista): ‘m’ posiciones. Luego, ocupa O(m).
   c. norm (lista): ‘m’ posiciones. Ocupa O(m).

   Esto se utiliza para $\sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!}$ soluciones. Además, también se utilizan las estructuras “soluciones” (lista de listas, donde cada solución tiene ‘m’ posiciones) y “soluciones_vistas” (set, a lo sumo ocupa lo mismo que la cantidad total de soluciones). Considerando que $n > m$, la complejidad espacial de esta sección termina siendo:

   $$O(n \cdot \sum_{k=0}^{m} \frac{n!\cdot m!}{k!\cdot(m-k)!\cdot(m-k)!}).$$

   Se observa que este valor es distinto al calculado para el pseudocódigo, ya que para el pseudocódigo no se considero que cada vez que se hace una llamada recursiva, se copia el OrderedDict ‘disponibles’ (restandole un elemento), ocupando a lo sumo O(n) porque hay n invitados, y $n > m$.

2. Respecto al cálculo de la máxima ganancia, para cada solucion se utilizan las siguientes estructuras:
   a. mejor_rot (lista): ‘m’ posiciones. Luego, ocupa O(m).
   b. max_ganancia (int): O(1).

c. m (int): O(1).
d. rotada (lista): O(m).
e. ganancia (int): O(1).
f. mejor_ganancia (int): O(1).
g. mejor_distribucion (lista): O(m).

Cada vez que genero una rotación estoy reutilizando las mismas variables, y lo mismo para cada solución, por lo cual se termina utilizando una complejidad $O(m)$ de espacio para esta parte.

3. Por lo tanto, la complejidad espacial total termina siendo $O(n \cdot \sum_{k=0}^{m} \frac{n! \cdot m!}{k! \cdot (m-k)! \cdot (m-k)!})$. Por lo explicado al momento de calcular la complejidad espacial de la obtención de todas las ubicaciones factibles, la complejidad espacial total termina siendo mayor a la calculada para el pseudocódigo.

Ejemplo simple:

Datos

-Porciones: m = 4
-Invitados: Ana, Beto, Carla, Damián
-Ofertas:

INVITADO	PORCION 1	PORCION 2	PORCION 3	PORCION 4
Ana	9	1	1	1
Beto	1	9	1	1
Carla	1	1	9	1
Damián	1	1	1	9

Enemigos:

- **Carla y Damián**

Paso a paso (resumen)

1. **El algoritmo genera todas las permutaciones posibles de 4 invitados, pero solo acepta aquellas en las que no haya enemigos consecutivos (ni linealmente ni circularmente).**
2. **Se normaliza cada solución válida para evitar contar duplicados rotacionales.**
3. **Se calcula, para cada solución válida, la mejor rotación posible (la que maximiza la suma de las ofertas según la posición).**
4. **Se selecciona la de mayor ganancia.**

Simulación paso a paso:

BACKTRACK

Estado inicial:
parcial = []
disponibles = {Ana, Beto, Carla, Damián}

Paso 1 — Elijo Ana:
 parcial = [Ana]
 disponibles = {Beto, Carla, Damián}

Paso 2 — Desde Ana, intento:
 -válido- Beto:
 parcial = [Ana, Beto]
 disponibles = {Carla, Damián}

Paso 3 — Desde Beto, intento:
 -válido- Carla:
 parcial = [Ana, Beto, Carla]
 disponibles = {Damián}

Paso 4 — Desde Carla, intento:
 -inválido- Damián es enemigo directo → descartado

Paso 5 - Como no hay más invitados disponibles, inserto un espacio VACIO

 -válido- VACIO:
 parcial = [Ana, Beto, Carla, VACIO]

Verificación final:

- Ana y VACIO → no enemigos
  -válida- => Solución válida

Normalizada (empieza con Ana):
 [SOLUCIÓN VÁLIDA] Ana, Beto, Carla, VACIO

(no está en las soluciones ya halladas, -Guardamos-)

-back- Vuelvo atrás a parcial = [Ana, Beto, Carla]

Paso 4 — Desde Carla, ya intenté todo → regreso

-back- Vuelvo atrás a parcial = [Ana, Beto]

Paso 3 — Desde Beto, intento:
 -válido- Damián:
 parcial = [Ana, Beto, Damián]
 disponibles = {Carla}

Paso 4 — Desde Damián, intento:
 -inválido- Carla es enemigo directo → descartado

Paso 5 - Como no hay más invitados disponibles, inserto un espacio VACIO
-válido- VACIO:
 parcial = [Ana, Beto, Damián, VACIO]

Verificación final:

- Ana y VACIO → no enemigos
  -válida- => Solución válida

Normalizada (empieza con Ana):
[SOLUCIÓN VÁLIDA] Ana, Beto, Damián, VACIO

(no está en las soluciones ya halladas, -Guardamos-)

-back- Vuelvo atrás a parcial = [Ana, Beto] → luego a [Ana]

Paso 2 — Desde Ana, intento:
-válido- Carla:
parcial = [Ana, Carla]
disponibles = {Beto, Damián}

Paso 3 — Desde Carla, intento:
-válido- Beto:
parcial = [Ana, Carla, Beto]
disponibles = {Damián}

Paso 4 — Desde Beto, intento:
-válido- Damián:
parcial = [Ana, Carla, Beto, Damián]

Verificación final:

- Carla y Damián → en 1 y 3 → no vecinos
- Ana y Damián → no enemigos
  -válida- => Solución válida

Normalizada (empieza con Ana):
[SOLUCIÓN VÁLIDA] Ana, Carla, Beto, Damián

(no está en las soluciones ya halladas, -Guardamos-)

-back- Backtrack a parcial = [Ana, Carla]

...

Y así se continúa explorando todas las combinaciones posibles:

- En cada paso, se prueba insertar a cada invitado disponible que no sea enemigo del último ya ubicado.
- Si no se puede insertar a nadie sin violar las restricciones, se prueba con un espacio VACÍO.
- Cuando se completa una asignación de longitud `m`, se verifica que el último y el primero no sean enemigos (por ser vecinos en la mesa circular).
- Luego, la solución se normaliza (rotándola para que comience con el primer invitado no vacío) y se verifica que no haya sido registrada antes.
- Si es nueva y válida, se guarda como una solución.

Este proceso se repite recursivamente, haciendo backtracking cada vez que se llega a un camino inválido o ya completo, hasta recorrer todas las configuraciones posibles.

ROTACIONES

-Ganancia por rotación (con ofertas)-

Solución 1: Ana, Beto, Carla, VACIO
-Rotación 0: Ana, Beto, Carla, VACIO → 9 + 9 + 9 + 0 = 27-> máx gan
-Rotación 1: VACIO, Ana, Beto, Carla → 0 + 1 + 1 + 1 = 3-> 27> 3
-Rotación 2: Carla , VACIO, Ana, Beto → 1 + 0 + 1 + 1 = 3-> 27> 3
-Rotación 3: Beto , Carla , VACIO, Ana→ 1 + 1 + 0 + 1 = 3-> 27> 3

=> máxima ganancia: 27, combinación: Ana, Beto, Carla, VACIO

- MÁXIMA GANANCIA GENERAL: 27, COMBINACIÓN: Ana, Beto, Carla, VACIO-

Solución 2: Ana, Beto, Damian, VACIO
-Rotación 0: Ana, Beto, Damian, VACIO → 9 + 9 + 1 + 0 = 19->máx gan
-Rotación 1: VACIO, Ana, Beto, Damian → 0 + 1 + 1 + 9 = 11-> 19> 11
-Rotación 2: Damian, VACIO, Ana, Beto → 1 + 0 + 1 + 1 = 3-> 19> 3
-Rotación 3: Beto , Damian, VACIO, Ana→ 1 + 1 + 0 + 1 = 3-> 19> 3

=> máxima ganancia: 19, combinación: Ana, Beto, Damian, VACIO

¿ máxima ganancia > MÁXIMA GANANCIA GENERAL? NO => No actualiza valor

Solución 3: Ana, Carla, Beto, Damián
-Rotación 0: Ana, Carla, Beto, Damián → 9 + 1 + 1 + 9 = 20->máx gan
-Rotación 1: Damián , Ana, Carla, Beto→ 1 + 1 + 9 + 1 = 12-> 20> 12
-Rotación 2: Beto, Damián, Ana, Carla→ 1 + 1 + 1 + 1 = 4-> 20> 4
-Rotación 3: Carla, Beto, Damián, Ana→ 1 + 9 + 1 + 1 = 12-> 20> 12

=> máxima ganancia: 20, combinación: Ana, Carla, Beto, Damián

¿ máxima ganancia > MÁXIMA GANANCIA GENERAL? NO => No actualiza valor

Y así se repite el proceso con cada una de las soluciones generadas por el backtracking. (Si se corre el programa en modo debug (con -d al final), los prints dentro del programa permitirán ver más detalladamente las soluciones y distintas rotaciones posibles para cada una y sus ganancias. Dada la cantidad de soluciones y sus rotaciones no era viable desarrollar el paso a paso de cada una, ya que sería demasiado extenso).

Para cada una:

- Se prueban todas las rotaciones posibles de la combinación circular.
- En cada rotación, se calcula la ganancia total usando las ofertas correspondientes a cada posición.
- Se guarda la rotación con mayor ganancia dentro de esa solución.
- Si esta ganancia supera la máxima ganancia global encontrada hasta el momento, se actualiza tanto el valor como la combinación óptima.

Este procedimiento garantiza que se encuentre la mejor combinación posible entre todas las rotaciones válidas, considerando tanto las restricciones como la maximización de las ofertas.

Finalmente como no hay más soluciones a las cuales aplicarle sus respectivas rotaciones, se obtiene la MÁXIMA GANANCIA GENERAL con valor: 27 y COMBINACIÓN: [Ana, Beto, Carla, VACIO]

Correcciones parte 2: Pseudocódigo y código

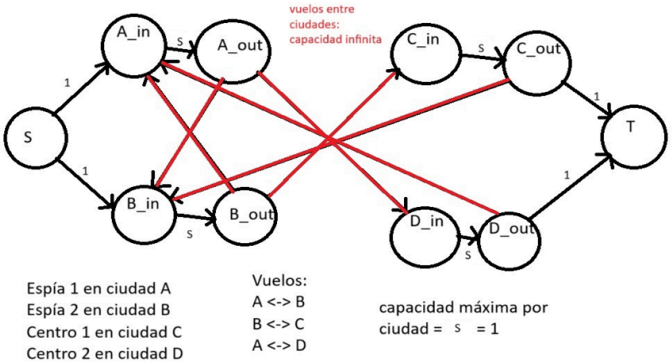
(...) Luego, agregamos al grafo dos nodos especiales: uno fuente F y uno sumidero T. Desde F, agregamos una arista hacia cada ciudad que tiene espías, con una capacidad igual a la cantidad de espías que hay en esa ciudad. Para cada vuelo entre dos ciudades A y B, agregamos dos aristas:
(...)

Ambas con capacidad infinita (o suficientemente grande), ya que no hay restricción explícita en los vuelos.

Para cada ciudad C que contiene un centro de investigación, agregamos una arista desde C hasta T, con capacidad igual al número de centros ubicados en esa ciudad, ya que cada centro recibe solo un agente.

Ejecutamos el algoritmo de flujo máximo provisto por la función *maximum_flow* de la librería networkx. Si el flujo máximo obtenido es igual a la cantidad de agentes (o centros, que es lo mismo), entonces es posible realizar el envío de información de forma segura.

Para determinar las rutas individuales de cada agente, reconstruimos los caminos desde la fuente F al sumidero T en el grafo resultante, siguiendo las aristas que tienen flujo positivo. Cada ruta encontrada representa el camino que un agente debe seguir hasta llegar a un centro de investigación.



Pseudocódigo

#las funciones “encontrar_camino_aumentante” y “ford_fulkerson” fueron diseñadas originalmente para implementar Ford-Fulkerson manualmente, pero fueron reemplazadas por la función maximum_flow de la librería networkx para garantizar mayor robustez y evitar la saturación prematura de caminos.

```
Funcion armado_red_de_flujo(n, s, ciudades, espias)
    Crear grafo G dirigido
    Definir fuente como "F"
    Definir sumidero como "T"
    Agregar F y T como nodos en G

    Crear lista de vuelos vacía y conjunto de ciudades vacío
    Leer archivo de ciudades
```

```
        Para cada línea
            Dividir por coma en origen y destino
            Agregar (origen, destino) a vuelos
            Agregar origen y destino al conjunto de ciudades

    Crear listas vacías espias y centros
    Leer archivo de espias
        Para cada línea
            _, ciudad = dividir por coma
            Si índice < n → agregar a espías
            Sino → agregar a centros

    Para cada ciudad en ciudades
        Agregar nodos ciudad_in y ciudad_out al grafo
        Agregar arista de ciudad_in a ciudad_out con capacidad s

    Para cada (origen, destino) en vuelos
        Agregar aristas:
            origen_out → destino_in (capacidad infinita)
            destino_out → origen_in (capacidad infinita)

    Para cada ciudad con espías:
        Si ya existe arista F → ciudad_in, incrementar capacidad
        Sino, crear arista F → ciudad_in con capacidad 1

    Para cada ciudad con centros:
        Si ya existe arista ciudad_out → T, incrementar capacidad
        Sino, crear arista ciudad_out → T con capacidad 1

    Devolver G, F, T

Funcion armado_de_ruta(G, flujo, fuente, sumidero)
    Crear lista vacía rutas
    Crear copia flujo_copia del diccionario flujo

    Para cada vecino en G[fuente]:
        Mientras flujo_copia[fuente][vecino] > 0:
            Inicializar lista ciudades_ruta vacía
            actual = vecino
            flujo_copia[fuente][vecino] -= 1
            prev_ciudad = None

            Mientras actual ≠ sumidero:
                Si actual termina en "_in" o "_out":
                    ciudad = parte antes de "_"
                Sino si actual es fuente o sumidero:
                    ciudad = None
                Sino:
                    ciudad = actual

                Si ciudad ≠ prev_ciudad y no es None:
                    Agregar ciudad a ciudades_ruta
                    prev_ciudad = ciudad

                Para cada siguiente en G[actual]:
                    Si flujo_copia[actual][siguiente] > 0:
                        flujo_copia[actual][siguiente] -= 1
                        actual = siguiente
                        Romper
```

```
            Si no se encontró siguiente → salir del bucle

        Agregar ciudades_ruta a rutas

    Retornar rutas

# Implementación actual: utilizamos la función maximum_flow de networkx, que implementa el algoritmo de Edmonds-Karp para encontrar el flujo máximo.

Funcion traslado(n, s, ciudades, espias)
    (G, fuente, sumidero) = armado_red_de_flujo(n, s, ciudades, espias)
    (flujo_max, flujo) = maximum_flow(G, fuente, sumidero)

    Si flujo_max ≠ n:
        Imprimir "Es imposible lograr el objetivo"
        Retornar

    rutas = armado_de_ruta(G, flujo, fuente, sumidero)

    Para i desde 0 hasta longitud(rutas) - 1:
        camino = rutas[i]
        Imprimir "Espía ", i + 1, ",", camino
```

(...)
(...) *Párrafo o texto que se encuentra igual a la primera entrega.*

Análisis de Complejidad

Sea n la cantidad de agentes y la cantidad de centros; m la cantidad de ciudades; s la cantidad máxima de agentes que pueden pasar por una ciudad; f el número de vuelos.

Temporal

- Armado de grafos: 2m (ciudades) y m + 2f + n (espías) + n (centros) = O(m + f + n)
- Flujo máximo. Utilizamos *maximum_flow*, que implementa el algoritmo de Edmonds-Karp. Su complejidad genérica es O(V*E^2), lo que se traduce en: O((m+n)*(m+n+f)^2)
- Reconstrucción de rutas:O(n .(m + f))
- Complejidad total: O((m+n)(m+f+n)2)

Espacial

- Para armar los grafos (nodos y aristas): O(m + f + n)
- Diccionarios flujo, capacidades y estructuras auxiliares (padres, cuello de botella y cola): O(m + f + n)
- Reconstrucción de rutas: un camino por espía, máximo m ciudades: O(m.n)
- Complejidad total: O(m+f+n) + O(nm) = O(nm + f)

Programar la solución. Incluya la información necesaria para su ejecución. Compare la complejidad de su algoritmo con la del programa.

Para poder ejecutar correctamente la solución debe tener instalar la librería networkx. Para hacerlo debe ejecutar en el directorio el siguiente comando:

```
$ pip install networkx
```

Esta librería se utiliza tanto para la construcción del grafo como para la resolución del problema mediante la función *maximum_flow*, que implementa el algoritmo de Edmonds-Karp para encontrar el flujo máximo en

redes con capacidades enteras. En consecuencia, la complejidad del programa implementado coincide con la propuesta en el pseudocódigo teórico de las secciones anteriores. Las estructuras de datos utilizadas (diccionarios y estructuras internas del grafo) permiten operaciones en tiempo constante promedio, y el uso de DiGraph de networkx no introduce sobrecostos que requieran ser computados. Se conserva así una complejidad:

Temporal: O((m + n)(m + f + n)^2) y Espacial: O(nm + f)

De esta forma, el análisis teórico se mantiene válido también para la implementación práctica.

¿Es posible expresar su solución como una reducción polinomial? En caso afirmativo explique cómo y en caso negativo justifique su respuesta.

Sí. Podemos reducir el problema planteado a uno de flujo máximo, que pertenece a la clase P y es resoluble en tiempo polinomial para capacidades enteras.

Transformamos la instancia del problema en un grafo dirigido con capacidades de la siguiente forma:

- Se agrega un nodo fuente y un nodo sumidero.
- Cada espía se representa como una arista de capacidad 1 desde la fuente hacia la ciudad en la que se encuentra.
- Cada centro de investigación se representa como una arista de capacidad 1 desde la ciudad donde se encuentra hacia el sumidero.
- Cada ciudad se transforma en un par de nodos in y out, conectados por una arista de capacidad s.
- Cada vuelo se representa con dos aristas (A_out → B_in y B_out → A_in) de capacidad infinita (o suficientemente grande).

Esto se reduce a un problema de flujo máximo con: capacidad 1 por espía, capacidad 1 por centro, capacidad s por ciudad y capacidad ∞ por vuelo. La transformación se realiza en tiempo polinomial y preserva la validez de las soluciones. Por lo tanto, se trata de una reducción polinomial válida.Una vez calculado el flujo máximo, cada unidad de flujo que llega desde la fuente hasta el sumidero representa un espía que logró llegar a un centro. Podemos reconstruir el camino seguido por cada espía observando las aristas con flujo positivo. Así, una solución del problema de flujo se puede transformar directamente en una solución válida para el problema original de “El traslado de información secreta”.

Correcciones parte 3: Los primeros 3 incisos

Demostrar que dada una posible solución que nos brindan, se puede fácilmente determinar si se puede cumplir o no con la tarea solicitada.

Sea **m** los tipos de profesionales.
Sea **n** los equipos de trabajo.
Sea **p** la máxima cantidad de personas por jornada
Sea **sc** la solución a la tarea solicitada. Será una tupla que contiene dos conjuntos, una con los invitados del día 1 y otra con los del día 2.

```
certificador(m, n, sc, p):
    dia1, dia2 = sc

    si tamaño(dia1) > p o tamaño(dia2) > p:                    O(1)
        retornar false

    para cada invitado en dia1:                                O(m)
```

```
            si invitado está en dia2:
            retornar false

    para cada equipo en n:
            si todos los miembros del equipo están en dia1:        O(n*m)
            retornar false
            si todos los miembros del equipo están en dia2:
            retornar false

    cantidad_tipos = len(m)
    visitados = Set()
    para cada invitado en dia1 and dia2:                           O(n)
            If invitado.tipo not in visitados:
                    visitados.add(tipo)
                    cantidad_tipos--

    if cantidad_tipos != 0:                                        O(1)

            return false
    retornar true
```

Hemos hecho un certificado que determina en tiempo polinomial si se puede cumplir con la tarea solicitada.

Demostrar que si desconocemos si es posible lograr una solución, es difícil poder afirmar que existe. Utilizar para eso el problema "Set splitting problem" (suponiendo que sabemos que este es NP-C).

Llamemos al problema JDC (Jornadas De Capacitación). Queremos determinar que JDC es NP-C. Para ello, procederemos a demostrar que pertenece a NP y a NP-H. Habiendo resuelto el 1), podemos afirmar que es NP dado que existe una solución que certifique que el problema se puede resolver en tiempo polinomial. Entonces, intentaremos demostrar que SSP ("Set splitting problem") es reducible polinomialmente a JDC.

Transformación I_{SSP} a I_{JDC}

Sea un conjunto S
Sea C={C1,C2,...,Ck} una colección de subconjuntos de S.
Por cada elemento de S tenemos un profesional
Por cada subconjunto de C tenemos un equipo
La capacidad máxima por jornada P está determinada por |S|/2

Para la complejidad, crear un profesional por cada elemento de S: O(S), crear un equipo por cada subconjunto de C: O(C) y asignar a cada equipo sus integrantes (según los elementos del subconjunto): $O(S \cdot C)$. En conjunto, la transformación tiene complejidad polinomial $O(S \cdot C)$

Transformación Sjdc a Sssp

Sea S1 el conjunto de elementos de S que fueron asignados a la primera jornada.
Sea S2 el conjunto de elementos de S asignados a la segunda jornada.
Entonces S =S1 U S2, y S1 ∩ S2=∅, es una partición de S.
Si existe una partición de los elementos de S en dos subconjuntos S1 y S2 tal que ningún subconjunto Ci ∈ C está completamente contenido en S1 ni en S2, entonces esa misma partición puede interpretarse como una asignación de profesionales a jornadas tal que ningún equipo esté totalmente en una jornada. Esto es una solución válida para JDC.
Por lo tanto, una solución para SSP implica una solución para JDC, y viceversa.

Esta transformación tiene complejidad O(S), ya que solo se necesita inspeccionar la asignación de cada profesional a una jornada y traducirla a una partición de S. Hemos demostrado que SSP es reducible polinomialmente a JDP, por lo que JDP ∈ NP-H y como demostramos que también pertenece a NP, JDP ∈ NP-C.

Demostrar que el problema "Set splitting problem" pertenece a NP-C. (Para la demostración se le solicita investigar. La misma se puede realizar partiendo desde "3-SAT" y pasando por "NAE-3-SAT").

Para demostrar que SSP pertenece a NP-C, hay que demostrar que pertenece a NP y NP-H. Arranquemos demostrando que SSP ∈ NP. Dado un conjunto S = {s1, s2, …, sn}, y otro conjunto C = {c1, c2, …, cm} tal que cada Ci es un subconjunto de S, buscamos verificar si existe una partición de S en A y B tal que cada Ci contiene al menos un elemento en A y otro en B.

```
Certificador(S, C, partición):
    A, B = partición

    para cada subconjunto Ci en C:
        tiene_en_A = False
        tiene_en_B = False

        para cada elemento x en Ci:
            si x está en A:
                tiene_en_A = True
            si x está en B:
                tiene_en_B = True

        si no (tiene_en_A y tiene_en_B):
            return False

    return True
```

Como hay M conjuntos y tienen como máximo N elementos, esta verificación tomará un tiempo máximo de O(N*M) por lo que comprobamos que set-splitting se puede verificar en tiempo polinomial por lo tanto pertenece a NP.

Para comprobar que SSP pertenece a NP-H, vamos a demostrar que el problema *NAE 3-SAT* es reducible polinomialmente a SSP, pasando antes por 3-SAT.*NAE 3-SAT es* una variante del problema 3-SAT, en el que cada cláusula tiene exactamente 3 literales (variables o sus negaciones). Una cláusula está satisfecha si no todos los literales tienen el mismo valor booleano. Es decir, al menos un valor debe ser verdadero y al menos uno debe ser falso. En principio, probaremos que *3-SAT* es reducible polinomialmente a *NAE 3-SAT.* Sabemos que 3-SAT es NP-C por lo que NAE 3-SAT debería ser tanto o más difícil de resolver que 3-SAT. Demostremos que NAE 3-SAT es NP y NP-H.

Para demostrar que NAE 3-SAT es NP, dado una fórmula φ con cláusulas de 3 valores booleanos (positivos o negados) y una asignación de valores booleanos a las variables, el certificador debe verificar que cada cláusula tiene exactamente 3 literales y en cada cláusula, no todos los literales tengan el mismo valor booleano.

Certificador(φ, asignación):

```
Para cada cláusula C en φ:
        valores = [ ]
        para cada x en C:
```

```
                    si x es una variable positiva:
                        valores.agregar(asignación[L])

                    si x es una variable negada:
                        valores.agregar(NOT asignación[L])

            si todos los valores son verdaderos o todos son falsos:
                return False
    return True
```

Hay m cláusulas, cada una con exactamente 3 literales → Tiempo total: O(m). Verificar si una lista de 3 booleanos es “todos iguales” se hace en tiempo constante. Por lo tanto, el certificado se verifica en tiempo polinomial, y concluimos que: NAE 3-SAT es NP

Vamos con la reducción:

Transformación de la instancia 3-SAT en NAE 3-SAT

Sea una instancia φ del problema 3-SAT donde: Cada cláusula Ci=(l1 ∨ l2 ∨ l3) contiene exactamente 3 literales lj, que pueden ser variables o negaciones.

Crear nueva fórmula φ' vacía

Crear nueva variable dummy b = FALSE

Al forzar b = False, la cláusula (l1, l2, l3, False) se cumple si y solo si no todos los literales l1, l2, l3 son False (o sea, al menos uno es True), lo cual es exactamente la semántica de 3-SAT.

Para cada cláusula C en la instancia 3-SAT:

Sea C = (l1 ∨ l2 ∨ l3)

Agregar cláusula NAE(l1, l2, l3, b) a φ'

retornar φ'

Siendo m la cantidad de cláusulas, para cada una en φ, agregamos 1 cláusula NAE-4 (4 literales), y eso se hace en tiempo constante → O(m)

Transformación de la solución NAE 3-SAT en la solución 3-SAT

Para cada cláusula de la solución NAE 3-SAT se ignora la variable dummy *b* y se toma la asignación del resto de las variables como solución para la instancia original de 3-SAT.

La solución de la instancia NAE-3-SAT con b = False garantiza que cada cláusula (l1 ∨ l2 ∨ l3) de la fórmula original está satisfecha. Por lo tanto, la misma asignación (excluyendo b) es una solución válida para 3-SAT. La transformación de la solución es lineal con respecto a la cantidad de cláusulas O(m), ya que solo se necesita verificar la validez de cada cláusula NAE y traducirla de vuelta a su forma original.

Demostramos que 3-SAT es reducible polinomialmente a NAE-3-SAT, por ende, NAE-3-SAT es NP-H y como también es NP, NAE-3-SAT es NP-C.

Ahora, vamos a probar que SSP es NP-H demostrando que NAE-3-SAT se puede reducir polinomialmente a SSP.

Transformación de la instancia NAE 3-SAT en la instancia SSP

Sea el conjunto de variables X={x1,x2,...,xn}

Sea el conjunto de cláusulas C={C1,C2,...,Cm}, donde cada cláusula Ci=(V1 ∨ V2 ∨ V3) contiene tres valores booleanos

Por cada Vj que aparece (ya sea xi o xi negada), se crea un elemento de un conjunto S

Por cada cláusula Ci, se crea un subconjunto {V1,V2,V3} ∈ F siendo F un conjunto finito

Para garantizar consistencia entre cada variable y su negación, se agregan subconjuntos adicionales:
Para cada variable xi, se agrega el subconjunto {xi, xi negada} ∈ F

Complejidad: Por cada cláusula se agrega un subconjunto ⇒ O(m) y por cada variable se agrega un subconjunto de consistencia ⇒ O(n). Por lo tanto, la complejidad Total: O(n + m)

Transformación de la solución SSP la solución NAE 3-SAT

Sea la solución de SSP una partición de S en A ∪ B tal que ningún subconjunto de F esté completamente dentro de uno solo

Definir una asignación booleana para cada variable xi:

Si xi ∈ A y xi negada ∈ B, asignar xi = True

Si xi ∈ B y xi negada ∈ A, asignar xi = False

En caso contrario (ambos en el mismo conjunto): no es solución válida

Si la solución SSP cumple lo pedido, es una solución válida

Revisar cada par xi, xi negada ⇒ O(n) y evaluar cláusulas si se desea validar la solución ⇒ O(m). En conjunto, la complejidad total: O(n + m)

Hemos probado que *NAE 3-SAT* es reducible polinomialmente a SSP´, por ende, SSP es NP-H, y como también es NP, podemos afirmar que SSP es NP-C.

Correcciones 2da entrega

Parte1

Reentregamos el programa 1 dentro del zip dado que un error de tipeo en el archivo no permitía correrlo correctamente.

Parte2

En esta sección solamente entregamos la implementación del código de ford_fulkerson y la búsqueda de los caminos de aumento en el zip. Finalmente la librería “networkx” la utilizamos únicamente para la representación del grafo dirigido, mientras que la lógica de flujo máximos fue programada manualmente mediante Ford-Fulkerson con BFS (Edmonds-Karp). En conclusión, la complejidad del programa implementado coincide con la expresada en el análisis anterior. Se conserva así una complejidad temporal de O((m + n)·(m + f + n)^2) y una complejidad espacial de O(nm + f).

Parte3

Corregimos el certificador:

Sea **tipos** los m tipos de profesionales.
Sea **equipos** los n equipos de trabajo.
Sea **p** la máxima cantidad de personas por jornada
Sea **sc** la solución a la tarea solicitada. Será una tupla que contiene dos conjuntos, una con los invitados del día 1 y otra con los del día 2.

```
certificador(tipos, equipos, sc, p):
	dia1, dia2 = sc
	si tamaño(dia1) > p o tamaño(dia2) > p:                          O(1)
			retornar false

	para cada invitado en dia1:                                      O(m)
			si invitado está en dia2:
			retornar false
	para cada equipo en equipos:
		miembros = equipo
			si todos los miembros del equipo están en dia1:                O(n*m)
			retornar false
			si todos los miembros del equipo están en dia2:
			retornar false
		si un miembro de miembros no está ni en el día 1 ni en el dia 2
			retornar false
	cantidad_tipos = len(tipos)
	visitados = Set()
	para cada invitado en dia1 and dia2:                          O(n)
		If invitado.tipo not in visitados:
			visitados.add(tipo)
			cantidad_tipos--
	if cantidad_tipos != 0:                                       O(1)
		return false
	retornar true
```

Corregimos la reducción:

Transformación I_{SSP} a I_{JDC}

Creamos dos jornadas de capacitación por cada partición S1,S2 de S del problema SSP
Sea C={C1,C2,...,Ck} una colección de subconjuntos de S.
Creamos un tipo de profesional ti por cada elemento de S
Creamos un equipo y le agregamos miembros, cada miembro un tipo de profesional, por cada subconjunto de C
Sea P la capacidad máxima de miembros por cada jornada de capacitación

Complejidad:O(m) crear un tipo de profesión por cada miembro creado, O(1) por crear las jornadas, O(n*m) crear un equipo de trabajo e incluir los tipos profesionales ⇒ TOTAL: O(n * m)

Si hay una asignación de profesionales a dos jornadas tal que ningún equipo esté todo en un mismo día cumpliendo la restricción P, entonces la partición de profesionales es válida para el problema SSP. Entonces obtener un si para SSP es equivalente a que exista una partición S_1, S_2 de S tal que ningún subconjunto C_j esté totalmente en S_1 y en S_2.

Transformación S_{JDC} a S_{SSP}

Recorremos cada jornada y transformamos cada tipo de profesional de cada equipo en un elemento C_i de un un subconjunto C de S. Además, transformamos cada jornada en una partición S_1 y S_2

Complejidad: O(n*m) por iterar cada miembro siendo este un tipo de profesional, de cada equipo de cada jornada.

Hemos comprobado que SSP es reducible polinomialmente a JDC, por ende, JDC pertenece a NP-H y como también es NP ⇒ JDC es NP-C

Corregimos la *reducción polinomial de 3-SAT a NAE 3-SAT*

Transformación de la instancia 3-SAT en NAE 3-SAT

Sea una instancia φ del problema 3-SAT donde: Cada cláusula Ci=(l1 ∨ l2 ∨ l3) contiene exactamente 3 literales lj, que pueden ser variables o negaciones.

Crear nueva fórmula φ' vacía

Crear constante b = FALSE

Al forzar b = False, la cláusula (l1, l2, l3, False) se cumple si y solo si no todos los literales l1, l2, l3 son False (o sea, al menos uno es True), lo cual es exactamente la semántica de 3-SAT.

Para cada cláusula C en la instancia 3-SAT:

Sea C = (l1 ∨ l2 ∨ l3)

Agregar cláusula NAE(l1, l2, l3, b) a φ'

Complejidad: Siendo m la cantidad de cláusulas, para cada una en φ, agregamos 1 cláusula NAE-4 (4 literales), y eso se hace en tiempo constante → O(m)

La solución de la instancia NAE-3-SAT con b = False garantiza que cada cláusula (l1 ∨ l2 ∨ l3) de la fórmula original está satisfecha. Por lo tanto, la misma asignación (excluyendo b) es un “SÍ” para 3-SAT

Transformación de la solución NAE 3-SAT en la solución 3-SAT

Para cada cláusula de la solución NAE 3-SAT se ignora la constante *b* y se toma la asignación del resto de las variables como solución para la instancia original de 3-SAT.

Complejidad: La transformación de la solución es lineal con respecto a la cantidad de cláusulas O(m), ya que solo se necesita verificar la validez de cada cláusula NAE y traducirla de vuelta a su forma original.

Demostramos que 3-SAT es reducible polinomialmente a NAE-3-SAT, por ende, NAE-3-SAT es NP-H y como también es NP, NAE-3-SAT es NP-C.