Integrantes:
Pablo Paillalef
Benjamín Vega

1. Conteste y explique en sus palabras:

- ¿Qué quiere decir que un problema sea NP-Hard?

Que un problema sea NP-hard, quiere decir que se le considera un problema intratable. Estos son de complejidad similar a los NP completos, pero no necesariamente pertenecen al conjunto NP.

- ¿Qué quiere decir que $P \neq NP$?

El que podamos **comprobar** fácilmente una solución no implica que podamos **encontrar** fácilmente una solución. La solución no se ha podido comprobar hasta ahora pero se cree que la respuesta es que **P ≠ NP** y no **P = NP**. Si la respuesta fuese **P = NP** entonces para cada problema en el que se puede **comprobar** fácilmente una solución entonces se puede **encontrar** fácilmente una solución.

- ¿Cómo se prueba que un problema es NP-Hard usando reducciones?

Si queremos demostrar que un problema X es NP-Hard, tomamos un problema Y NP-completo y lo intentamos reducir a X. Si conseguimos esto en tiempo polinomial, entonces X es NP-Hard.

2. Escoja 2 problemas NP-hard y descríbalos claramente.

- Suma de subconjuntos:

Es un problema de decisión. En su formulación general, hay un conjunto de números S = {a_1,...,a_n} y una suma objetivo T, y con estos datos nos hacemos la siguiente pregunta ¿Existe un subconjunto contenido en S cuya suma sea T?

Entrada: Set de números S y la suma T a la que se quiere llegar.

Salida: Verdadero o Falso, dependiendo de si la suma es posible en el conjunto de números S.

❖ El problema de la Mochila (knapsack problem):

El problema de la Mochila es un problema de optimización en el que se tiene un conjunto de objetos con un valor y peso asociados y se buscan las combinaciones cuya suma de valores sea el máximo posible sin superar un peso límite.

Entrada: Capacidad máxima de la mochila (W), lista de objetos de valor V = [v_1, v_2, v_3,.., v] y de peso P = [p_1, p_2, p_3,.., p] donde cada objeto k_i vale v_i y pesa p_i.

Salida: Lista de objetos donde $\sum_{i=1}^{m} p_i \leq W$ y $\sum_{i=1}^{m} v_i$ es el valor máximo posible sin superar el peso.

3. Demuestre que son NP-hard utilizando **reducción**. Para cada **problema B** escogido:

1. Explique **cómo** y **por qué** un problema **A** (NP-Hard) se puede reducir a su problema **B**. Sea claro, use figuras y ejemplos si es necesario.

❖ Problema del Suma de Subconjuntos:

El problema de la Cobertura de Vértices se puede reducir al problema de la Suma de Subconjuntos. Si cambiamos la pregunta a ¿Tiene un grafo G una cobertura de vértices de tamaño K? Tenemos un Grafo G(V,E) de N vértices y queremos una cobertura de vértices de tamaño K. Con esto tenemos que V = {1,2,...,N}. Podemos definir enteros a_1,...,a_n – uno para cada vértice– y b_(i,j) –uno para cada arco (i,j) que pertenezcan a E–, y por último un parámetro k' **(se puede probar que el grafo tenga una cobertura de vértices de tamaño K, si y sólo si, hay un subconjunto de enteros de tamaño k')**. Si definimos los enteros a_i y b_(i,j) de forma que tenemos un subconjunto de los a_i y b_(i,j) que sumen k', entonces: si el subconjunto de de a_i que es una cobertura de vértices en el grafo, y el subconjunto de b_(i,j) que corresponde a los arcos del grafo de forma que todos están conectados con al menos uno de los vértices en a_i. Entonces la construcción va a forzar a que la cobertura de vértices sea de tamaño K.

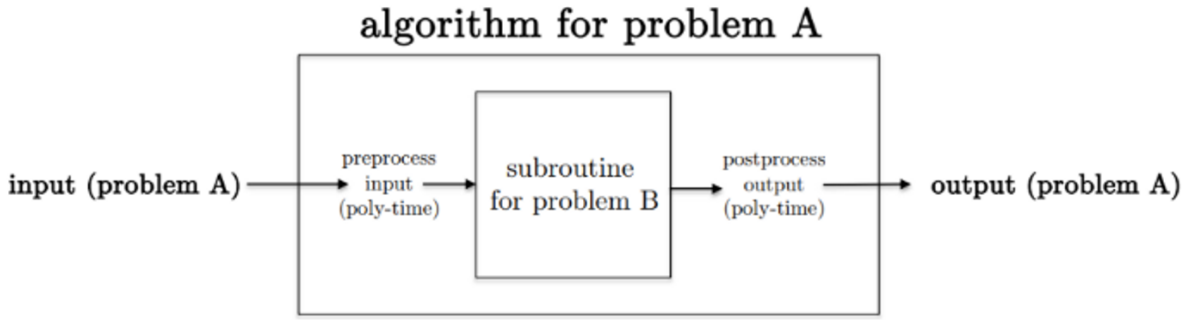
De esta forma el algoritmo de Cobertura de Vértices buscará la suma de los subconjuntos que dan una cobertura de un tamaño específico.

❖ Problema de la Mochila:

El problema de la **suma de subconjuntos** se puede reducir al problema de la **mochila** ya que se puede encontrar conjuntos que sean menores o iguales al valor

requerido y luego verificar los que son exactamente iguales al valor. Esto se puede realizar si se hace que el peso de cada objeto sea igual a su valor. De esta manera el algoritmo buscará aquellas combinaciones de objetos cuya suma de valores o peso sea igual al valor buscado.

2. Describa un algoritmo que utilice la reducción para resolver el **problema A usando el problema B** (indique claramente el **preproceso** y el **postproceso** requerido).



❖ Problema de la Suma se Subconjuntos:

El algoritmo que usaría el de la suma de subconjuntos para resolver el de la cobertura de vértices consistiría en lo siguiente:

1. Recibe el grafo G(V,E) y un parámetro K (que es el tamaño de la cobertura de vértices)
2. Para cada vértice en V y arco (i,j) en E, definimos unos enteros a_i y un b_(i,j) respectivamente. Para K definimos un entero k'.
3. Usamos el algoritmo de la suma de subconjuntos para encontrar un subconjunto en a_i que dé como resultado k'. De forma que i esté contenido en la Cobertura y en todos los enteros b_(i,j).
4. Se revisa el resultado en el post-proceso y, una vez que k' se verifique, se podrá retornar el conjunto de vértices que cumplan la Cobertura esperada.

❖ Problema de la Mochila:

Un algoritmo que podría utilizar el algoritmo de la mochila para resolver el problema de la suma de subconjuntos sería el siguiente:

1. Primero recibe el valor buscado y la lista de valores asociados a cada elemento.
2. Luego en el pre-proceso duplica la lista de valores pero esta será para los pesos de los elementos, ya que estos deben ser igual al valor de cada uno.
3. Después se utiliza el algoritmo que soluciona el problema de la mochila para obtener una combinación de objetos cuya suma de valores sea menor o igual al valor requerido.

4. Finalmente se revisa el resultado en el post-proceso y, si la suma de valores de la lista retornada es igual al valor requerido, se retornará esta lista, en caso contrario se retornará que no existe una combinación que sea igual al valor buscado.

4. Proponga y describa algoritmos sencillos que puedan resolver los 2 problemas B escogidos. Indique el tiempo de ejecución de cada uno de ellos.

- ❖ Problema de la suma se Subconjuntos:

A continuación se presenta un código en lenguaje Python que resuelve el problema de la **Suma de Subconjuntos** utilizando el algoritmo que resuelve el problema de la **Mochila.**

```python
# Algoritmo que resuelve el problema de la Suma de Subconjuntos
def SubsetSum(T, V):
    lista = Mochila(T, V, V) # Algoritmo para el problema de la Mochila

    # Si la lista no existe o está vacía, no existe una combinación que
    # cumpla la condición por lo que se retorna
    if lista is None or len(lista) == 0:
        print("No existe una combinación de valor", T)
        return

    # Se calcula el valor de la suma de valores de la lista
    sum = 0
    for i in range(len(lista)):
        sum += V[i]

    # Si la suma de los objetos es igual al valor T, se cumple la condición
    if sum == T:
        print("Existe una combinación de valor", T)
        print(lista)
    # Sino, no se cumple la condición
    else:
        print("No existe una combinación de valor", T)
```

Tiempo de ejecución:

Analizando las partes se puede ver que algoritmo tiene un tiempo de ejecución de **O(n)**:

1. Utiliza el algoritmo que resuelve el problema de la Mochila, el cual tiene un tiempo de ejecución de **O(n*w)** donde **n** es el número de objetos y **w** es la capacidad de la mochila, pero como la capacidad de la mochila es la misma que el valor **t** se puede reemplazar por **O(n*t)**.
2. Un ciclo **for** que itera **n** veces, por ello tiene un tiempo de ejecución de **O(n)**.

Con esto se obtiene la fórmula **T(n) = O(n*t) + O(n).**

$$T(n) = O(n * t) + O(n)$$
$$T(n) = O(n + n * t)$$
$$T(n) = O(n * (1 + t))$$
$$T(n) = O(n * t)$$

Esto significa que el algoritmo tiene un tiempo de ejecución de **O(n*t)**.

- Problema de la Mochila

A continuación se presenta un pseudocódigo que resuelve el problema de la **Mochila** de manera dinámica (bottom-up)**:**

```
# Algoritmo que resuelve el problema de la Mochila
Mochila(K, V, W)
    listaIndices // Lista que contendrá los índices de los objetos utilizados en la solución
    matrix = [[0 for i in range(K+1)] for i in range(len(V))] //Matriz auxiliar donde se guardan las soluciones

    // Se revisa cada objeto n
    for n in range(0, len(V)):
        // Se revisa cada peso posible
        for w in range(1, K+1):
            // Se revisa si el peso del objeto es menor al peso máximo actual
            if W[n] <= w:
                // Se guarda el valor máximo
                matrix[n][w] = max(V[n] + matrix[n-1][w-W[n]] , matrix[n-1][w])
            else:
                // Sobrepasa el peso por lo que se guarda la solución del objeto anterior
                matrix[n][w] = matrix[n-1][w]

    listaIndices = ObtenerObjetos(matrix) // Obtiene los índices de los objetos utilizados en la solución final

    return listaIndices // Se retorna la lista de índices de los objetos de la solución final
```

Tiempo de ejecución:

El **Algoritmo de la Mochila** (dinámico bottom-up) tiene un tiempo de ejecución de **O(n*w)** donde **n** es la cantidad de objetos y **w** es la capacidad de la mochila, esto se obtiene analizando las partes que lo componen:

1. Un ciclo **for** que itera **n** veces, por ello tiene un tiempo de ejecución de **O(n)**.
    1.1. Otro ciclo for que itera **w** (capacidad de la mochila) veces donde:
        1.1.1. Se guarda un valor máximo entre dos variables, lo cual tiene un tiempo de ejecución de **O(1)**.
2. Se buscan los índices de los objetos de la solución, lo que tiene un tiempo de ejecución de **O(n)**.

Con esto se obtiene la fórmula **T(n) = O(n) * O(w) * O(1) + O(n)**:

$$T(n) = O(n) * O(w) * O(1) + O(n)$$
$$T(n) = O(n * w) + O(n)$$
$$T(n) = O(n * (w + 1))$$
$$T(n) = O(n * w)$$

Por lo cual la complejidad temporal del algoritmo es **O(n*w)**.