Subset Equality Problem

Luis Alejandro Rodríguez Otero
Matcom
September 21, 2024

Contents

1 Problemática

1.1 ¿De qué trata el problema?

Dado un conjunto S de números enteros no negativos, el problema consiste en identificar si existe una partición del conjunto S en dos nuevos conjuntos X y Y, de tal forma que la suma de todos los elementos que pertenecen a X sea igual a la suma de todos los elementos que pertenecen a Y.

De manera formal el problema se define de la siguiente manera: Sea $S = s_1, s_2, ..., s_n$ se deben encontrar $X = x_1, x_2, ..., x_a$ y $Y = y_1, y_2, ..., y_b$ tal que $a + b = n$, y:

$$X \cup Y = S, X \cap Y = \emptyset, \sum_{i=0}^{a} x_i = \sum_{j=0}^{b} y_j$$

1.2 Entrada

Un array S de números enteros no negativos.

1.3 Salida

`True` o `False` si es posible encontrar las particiones que cumplan la condición expuesta anteriormente, y en caso positivo devolver tambien dichas particiones. Si existe más de un par de particiones válidas se puede devolver cualquiera.

2 Demostración de NP Completitud

Para demostrar que un problema es NP completo debemos demostrar que dicho problema es tanto NP, como NP hard. Vamos a hacer esas demostraciones a continuación.

2.1 NP

Un problema es NP si, dada una solución candidata, es posible verificar en tiempo polinomial si la solución es correcta o no. En este caso una solución candidata está formada por los conjuntos de enteros X y Y. Para verificar si la solución es correcta primero se debe comprobar si $X \cup Y = S$ y si $X \cap Y = \emptyset$, dicha comprobación puede hacerse en $O(n)$ simplemente comprobando si no hay elementos repetidos en los conjuntos y en estos están

presentes todos los elementos de S. Aquí aclarar que los elementos se están comparando por referencia y no por valor, es decir, los elementos de X y Y no se comparan por sus valores numéricos sino por su referencia en S, por ejemplo, si hay un 5 en X y otro en Y entonces se tiene que cumplir que hay dos elementos 5 en S. Luego se verifica que $\sum_{i=0}^{a} x_i = \sum_{j=0}^{b} y_j$ lo cual puede comprobarse haciendo un recorrido lineal por ambos conjuntos lo que tiene una complejidad de $O(a+b) = O(n)$.

Por tanto es posible verificar la correctitud de una solución candidata en tiempo polinomial ($O(n)$), por lo que queda demostrado que el problema pertenece al conjunto de los problemas NP.

2.2 NP Hard

Subset Equality problem es un caso particular de otro problema llamado Subset sum problem. En este se recibe un conjunto de entrada S y un entero k, y lo que hay que buscar es un subconjunto de S que cumpla que la suma de todos sus elementos es exactamente k. Es fácil ver que Subset Equality problem en S es quivalente a resolver Subset sum con S y $k = N \div 2$ donde $N = \sum_{i=0}^{n} x_i$ donde $x_i \in S$ para todo i. Por tanto lo que vamos a demostrar es que Subset sum es NP hard.

Para demostrar que Subset Sum es NP hard vamos a realizar una reducción de un problema conocido que sabemos que es NP hard a este. El problema que vamos a utilizar es **3-Sat**, en este problema se tiene una expresión booleana en forma normal conjuntiva y cada cláusula de esta tiene 3 variables, se debe encontrar una asignación a las variables (1 o 0) que haga verdadera la expresión.

Sea una expresión booleana con n variables x_i y m cláusulas c_j. Por cada variable x_i vamos a construir los números t_i y f_i cada uno de $n+m$ dígitos de la siguiente manera:

- El i-ésimo dígito de t_i y f_i es 1.
- Para j con $n+1 \leq j \leq n+m$, el j-ésimo dígito de t_i es 1 si la variable x_i está en la clásula c_{j-n}.
- Para j con $n+1 \leq j \leq n+m$, el j-ésimo dígito de f_i es 1 si la variable $\neg x_i$ está en la clásula c_{j-n}.
- El resto de dígitos de t_i y f_i son 0.

Por ejemplo, para la siguiente expresión Booleana:

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

Los t_i y f_i quedan de la siguiente manera:

	i			**j**			
Número	1	2	3	1	2	3	4
t_1	1	0	0	1	0	0	1
f_1	1	0	0	0	1	1	0
t_2	0	1	0	1	0	1	0
f_2	0	1	0	0	1	0	1
t_3	0	0	1	1	1	0	1
f_3	0	0	1	0	0	1	0

Ahora para cada cláusula c_j vamos a construir los números x_i y y_i también de $n + m$ dígitos, de forma que el único dígito igual a 1 en las dos variables será el $(n + j)$-ésimo, el resto serán 0. Siguiendo el ejemplo de la expresión booleana anterior los números que se forman son los siguientes:

	i			**j**			
Número	1	2	3	1	2	3	4
x_1	0	0	0	1	0	0	0
y_1	0	0	0	1	0	0	0
x_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
x_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
x_4	0	0	0	0	0	0	1
y_4	0	0	0	0	0	0	1

Finalmente vamos a crear un último número s de también $n + m$ dígitos de forma que los n primeros números son 1, y el resto de m números son 3.

Definamos una instacia de Subset sum donde el conjunto de entrada está formado por todos los números que hemos construido excepto s, quien será el parámetro k, es decir, la suma requerida. Vamos a demostrar que si se tiene una asignación de las variables que satisface la expresión (o sea se resuelve el 3-SAT), entonces existe un subconjunto de números que satisface la instancia de subset sum que se creó.

Teniendo la asignación que resuelve el 3-SAT vamos a armar el subconjunto de la siguiente forma:

- Tomamos t_i si la variable x_i es `True`
- Tomamos f_i si la variable x_i es `False`
- Tomamos x_j si la cláusula c_j tiene a lo sumo 2 variables `True`
- Tomamos y_j si la cláusula c_j tiene exactamente 1 variable `True`

Según el ejemplo que hemos estado siguiendo los números escogidos si consideramos que x_1, x_2 y x_3 son `True` son los siguientes:

	i			**j**			
Número	1	2	3	1	2	3	4
t_1	1	0	0	1	0	0	1
t_2	0	1	0	1	0	1	0
t_3	0	0	1	1	1	0	1
x_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
x_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
x_4	0	0	0	0	0	0	1
s	1	1	1	3	3	3	3

Vamos a demostrar por qué la suma de los elementos del conjunto siempre es igual a s si lo construimos de esa manera:

Primeramente vamos a analizar los números construidos de forma tabular como hemos hecho hasta ahora, de modo que para resolver el Subset sum las primeras n columnas deben sumar 1 y el resto de las m columnas debe sumar 3. La primera parte se cumple ya que para cada i se escoge a t_i o a f_i, nunca a ambos ya que la variable x_i solo tiene un valor en la asignación del 3-SAT, mientras que los números x_j y y_j no tienen 1 en esas posiciones, por tanto no influyen en la suma. Luego las n primeras columnas suman 1.

Ahora notemos que en cada cláusula c_j debe haber al menos una variable en `True` para satisfacer la expresión, esto en nuestra tabla se traduce en que, sin contar los número x_j y y_j, cada una de las j columnas con $n+1 \leq j \leq n+m$ suma al menos 1, ya que cada t_i aporta un 1 a la columna j si aparece x_i en c_j, y cada j_i si aparece $\neg x_i$, además recordemos que solo tomamos t_i o f_i de forma tal que x_i o $\neg x_i$ sean `True`. Además es fácil ver que dicha suma

es a lo sumo 3, ya que solo hay 3 variables en cada cláusula. Por tanto para cada columna j hay 3 casos:

- Entre las primeras n filas hay acumulada una suma de 1 en la columna j: Esto significa que la cláusula c_j tiene exactamente una variable `True`, por lo que se tomó el elemento y_j que aporta 1 a la suma de la columna. También se cumple que en la cláusula c_j hay a lo sumo 2 variables `True` (solo hay 1) por lo que tambien se tomó el elemento x_j que aporta 1 a la suma de la columna. Luego la suma de la columna j es $1 + 1 + 1 = 3$

- Entre las primeras n filas hay acumulada una suma de 2 en la columna j: Esto significa que la cláusula c_j tiene exactamente dos variable `True`, por lo que no se tomó el elemento y_j pero si el elemento x_j, ya que se cumple que hay a lo sumo 2 variables `True`. Por tanto la suma de la columna j es $2 + 1 = 3$

- Entre las primeras n filas hay acumulada una suma de 3 en la columna j: Esto significa que la cláusula c_j tiene exactamente tres variable `True`, por lo que no se tomarán ni los elementos y_i, ni x_i. Por tanto la suma de la columna j se mantiene en 3.

Nótese que las únicas variables que pueden influir en la suma de una columna j son las que se han analizado en cada caso, y todas aportan a lo sumo solo 1 a la suma. Además todas estas operaciones (tomar números del conjunto y comprobar en qué cláusulas está cada variable) se pueden hacer en tiempo polinomial. Por tanto queda demostrado que si existe una solución al problema de 3-SAT, podemos construir a partir de esta una solución al Subset sum problem en tiempo polinomial. Vamos a hacer la demostración ahora en el otro sentido. Vamos a demostrar que si existe una solución al Subset sum entonces podemos construir una solución para el 3-SAT.

Teniendo un subconjunto C de números que suman s vamos a hacer la asignación para el 3-SAT de la siguiente manera: Asignamos `True` a la variable x_i si t_i pertenece a C, mientras que asignamos `False` a la variable x_i si f_i pertenece a C. Primeramente notemos que para un mismo valor de i no pueden pertenecer simultáneamente t_i y f_i a C, ya que sino alguno de los primeros n dígitos sumaría 2, lo que contradice que C sea una solución para el Subset sum, esto significa que a cada variable x_i se le está asignando un único valor. Ahora notemos que, incluso si todos los x_j y y_j pertenecen

al C, la suma de las m columnas restantes será a lo sumo 2, por tanto para que C sea solución se tienen que haber escogido los t_i y los f_i de manera que cada uno aporte como mínimo 1 a la suma de cada una de estas columnas. Por la manera en que se han construido todos los números, esto se traduce a que en cada cláusula va a haber al menos una variable en `True` lo que hace que se satisfaga la ecuación, y por tanto la asignación realizada resuelve el 3-SAT. Comprobar los elementos de C para asignar valores a los x_i se pude hacer en tiempo lineal con un simple recorrido de C.

Finalmente queda demostrado que se puede construir una solución para el 3-SAT a partir de una para el Subset sum en tiempo polinomial, por lo que como 3-SAT es NP hard entonces Subset Sum es también NP hard. Lo que a su vez implica que Subset Equality problem es NP hard. Finalmente como Subset Equality problem es NP, y ademas NP hard, es también NP completo.

3 Solución de fuerza bruta

La solución de fuerza bruta del ejercicio es en verdad bastante trivial, se generan todos los posibles subconjuntos de S y para cada uno se chequea que la suma de todos sus elementos es igual a $total\_sum \div 2$ donde $total\_sum$ es la suma de todos los elementos de S. Se devuelve el primer subconjunto que cumpla esa condición y el resto de S en otro subconjunto. La suma de ambos subconjuntos es $total\_sum \div 2$ por tanto son iguales. De no cumplirse la igualdad para ningún subconjunto de S entonces se devuelve `False`, ya que para esa entrada el problema no tiene solución.

3.1 Podas

Aplicamos dos podas al algoritmo para aumentar ligeramente su eficiencia. La primera es un chequeo que se realiza sobre S al inicio, se comprueba si la suma total de los elementos del conjunto es impar, si esto se cumple entonces el problema no tiene solución, ya que sería imposible armar dos subconjuntos que sumen igual. La segunda poda es que no se generan subconjuntos de todos los tamaños (de 1 a n), sino que solo generamos subconjuntos de tamaño a lo sumo la mitad de n. Esto es correcto ya que para cualquier manera de dividir S en X y Y se tiene que cumplir que la longitud de uno de estos debe ser a lo sumo la mitad de n, siendo n la longitud de S, ya que en caso contrario las longitudes de X y Y sumarían más de n, lo cual no es correcto. Por tanto si para un conjunto de tamaño menor que la mitad de n se cumple que la suma de sus elementos es la deseada, el resto

del conjunto sumará lo mismo sin importar su tamaño, por lo que no hace falta generarlo. La primera poda permite resolver algunas instancias sin solución del problema en orden lineal, mientras que la segunda no reduce la complejidad temporal total pero si logra que se realicen muchas menos operaciones.

3.2 Complejidad temporal y espacial

Para resolver el problema se utilizó el módulo `Combinations` de la biblioteca `Itertools`, este genera todas las combinaciones posibles del array de entrada pero no las almacena simultáneamente por lo que la complejidad espacial no pasa de $O(n)$ que es el tamaño del array de entrada.

En cuanto a la complejidad temporal, la operación con más costo del programa es la de generar todos los posibles subconjuntos de un conjunto, dicha complejidad es $O(2^n)$, ya que cada elemento tiene 2 opciones: Estar o no en un conjunto, por tanto para cada elemento hay 2 posibilidades, y como son n elementos entonces hay 2^n posibles subconjuntos. A cada uno de estos subconjuntos además se le hace una comprobación lineal (chequear el valor de la suma de sus elementos), por tanto en términos estrictos la complejidad temporal del algoritmo es de $O(2^n \cdot n)$.

4 Solución polinomial con DP

4.1 Idea general

Vamos a utilizar un enfoque de programación dinámica (dp) para resolver algunas instancias del problema en tiempo polinomial. Sea $target$ la mitad de la suma de todos los elementos del conjunto S (conjunto de entrada), vamos a crear una lista de dp de tamaño $target + 1$, donde la invariante que mantendrá dicha lista será la siguiente: $dp[i]$ es `True` si es posible armar un subconjunto de S tal que la suma de sus elementos es igual a i, de lo contrario es `False`. De esta manera en $dp[target]$ estará guardado si es posible encontrar un subconjunto cuyos elementos sumen $target$, lo cual soluciona el Subset Equality problem. En caso positivo reconstruimos los subconjuntos solución y los devolvemos.

4.2 Explicación del código

Al igual que en la solución de fuerza bruta, la primera comprobación que se hace es si la suma total de S es impar, en cuyo caso el problema no tiene

solución. Posteriormente se crean las listas dp y $prev$ ambas de tamaño $target + 1$ que vamos a explicar a continuación.

La lista de dp se construye de la siguiente manera: $dp[0] =$ `True`, ya que siempre es posible lograr una suma de 0 con el subconjunto vacío. Luego para cada elemento num de S y para cada i ($i \leq target$) de mayor a menor, $dp[i]$ se hace `True` si $dp[i - num]$ es `True`, aquí lo que estamos diciendo es que si hay una manera de construir un subconjunto que sume $i - num$ entonces podemos crear uno que sume i simplemente añadiendo num a dicho subconjunto.

La lista $prev$ nos va a ayudar en la reconstrucción de los subconjuntos solución. Esta se actualiza cada vez que se hace `True` una posición en el bucle del dp. Concretamente, lo que se hace es que cuando $dp[i - num]$ es `True`, luego de actualizar el dp se coloca en $prev[i]$ el valor num. Esto lo que significa es que con el número que se consiguió construir un subconjunto de suma i fue con num. De esta forma para reconstruir el conjunto solución vamos consultando los valores de $prev$ de la siguiente manera: Primero vemos con que número num_1 se logró armar una suma $target$, luego veamos con que número num_2 se armó una suma $target - num_1$, luego una suma $target - num_1 - num_2$ y asi sucesivamente hasta que $target - num_1 - num_2 - ... - num_a = 0$, con esto entonces ya podemos construir un subconjunto $X = num_1, num_2, ..., num_a$ tal que la suma de todos ellos es $target$. El otro subconjunto solución es simplemente el resto de S que tambien sumará $target$. Finalmente si fue posible construir ambos subconjuntos se devuelve `True` y los subconjuntos solución, mientras que en caso contrario solo se devuelve `False`.

4.3 Demostración formal

Vamos a demostrar las propiedades de subestructura óptima y subproblemas superpuestos, seguido de la demostración de la correctitud del algoritmo.

4.3.1 Subestructura óptima

Esta propiedad se refiere a que la solución de un problema se pueda construir a partir de las soluciones de subproblemas más pequeños. En este caso cada problema depende de un subproblema: Para calcular si es posible armar un subconjunto con suma i, comprobaremos si es posible armar un conjunto de suma $i - num$, donde $num \in S$. Esto se hace para cada elemento de S y se cumple lo siguiente:

Para el elemento j-ésimo de S lo que se está calculando realmente es qué

sumas pueden lograrse con num_j dado que estas fueron las que se lograron con subconjuntos formados por los primeros $j-1$ elementos de S. Esto pasa por que con cada número se actualiza la tabla de dp. Por ejemplo con el primer elemento de S la tabla está en `False` completa por lo que solo se actualizará la posición $dp[num_0]$, ya que la única suma que se puede lograr con un único número es precisamente ese número. Ya para el siguiente número, el algoritmo va a tener en cuenta lo que se logró sumar con el caso anterior, o los casos anteriores (por que la lista de dp se va actualizando), y así para cada elemento. Por tanto para el problema de hayar las sumas posibles que se logran con el número num_i son necesarias las respuestas de los subproblemas que calcularon las sumas que se alcanzaron usando los anteriores i números. Por tanto se construyen soluciones a problemas utilizando soluciones de subproblemas más pequeños, luego se cumple la propiedad de subestructura óptima.

4.3.2 Subproblemas superpuestos

Esta propiedad se refiere a que el mismo subproblema se puede estar calculando en varias instancias de la ejecución. En este caso el subproblema que se puede estar calculando varias veces es: Para este elemento de S, es posible construir una suma de i dado que las sumas que se han logrado armar hasta ahora son las que hay en la lista de dp. Concretamente el algoritmo lidia con esto chequeando siempre si $dp[i]$ ya está en `True`, en cuyo caso se salta a la siguiente iteración. Esto ocurre, por ejemplo, cuando ya se logró armar una suma de i con elementos anteriores pero ahora, por la forma en se ejecuta el programa, toca volver a chequear si se puede conseguir la misma suma. Sin embargo, a efectos del problema ya no es necesario volver a resolver ese subproblema, si ya se logró armar una suma de i no es necesario volver a calcular si se puede volver a formar con otros elementos, esto no alterará la solución del problema, y por eso, el algortimo evita volver a repetir el cómputo saltando a la siguiente operación. Por tanto este es un subproblema que, sin el enfoque del dp, requeriría de recalcularse repetidas veces, por tanto, se cumple la propiedad de los subproblemas superpuestos.

4.3.3 Correctitud del dp

Vamos a demostrar que la respuesta al Subset Equality problem se encuentra en $dp[target]$ al final de la ejecución. Para esto debemos demostrar que la lista de dp se está construyendo correctamente. Vamos a hacer inducción en las posibles sumas que calcula el dp:

Caso base: $i = 0$

Para $i = 0$ se actualiza $dp[0]$ = `True` desde el principio, ya que en todo momento se puede alcanzar una suma de 0 con el subconjunto vacío. Por tanto $dp[0]$ está bien calculado en todo momento.

Hipótesis de inducción

Vamos a suponer que para todo i, $(i < target)$ la posición $dp[i]$ indica si es posible lograr una suma de i con los elementos de S.

Paso inductivo

Recordemos que en la demostración de subestructura óptima vimos que lo que se calcula en cada iteración es, que sumas pueden lograrse con el número actual dado que se han conseguido estas con los anteriores. Por tanto para cada elemento de S se ha comprobado si $dp[target - num]$ es `True`. En cualquier momento si esto se cumple significa que existe al menos un subconjunto de elementos de S al cual si añadimos num se soluciona el problema, y por hipótesis de inducción sabemos que $dp[target - num]$ está correctamente calculado, por tanto, la lista de dp se está construyendo bien en todas sus posiciones, lo que significa que la respuesta al problema se encuentra en la última posición, que indica si fue posible o no construir un subconjunto de S cuyos elementos sumen $target$.

Luego en la explicación del código vimos que con la lista $prev$ se construye un rastro de números que se puede seguir de la forma ahí indicada para reconstruir un subconjunto de S con suma $target$, y que se devuelven como subconjuntos solución este y el resto de S, donde ambos suman $target$. Por tanto el algoritmo devuelve, si los hay, dos subconjuntos de S donde la suma de cada uno de los elementos de los conjuntos por separado es igual.

4.4 Complejidad temporal y espacial

La lista de dp se construye con un tamaño de $target + 1$ (al igual que $prev$) donde $target$ es la mitad de la suma de todos los elementos de S. Esta se llena recorriendo todas sus posiciones por cada elemento de S, lo que da una complejidad temporal de $O(n \cdot target)$ y una complejidad espacial de $O(target)$. Por tanto esta solución es más eficiente que la solución por fuerza bruta solo en los casos donde $target < 2^n$, es decir, en los casos donde la suma de los elementos de S no es demasiado grande.