Subset Sum Problem

Eric Luis López Tornas C-411

Contents

1 Definición del problema **3**

2 Demostración de NP-completitud **3**

3 Fuerza Bruta **5**

4 Dinámica Pseudopolinomial **5**

5 Anexos **7**
5.1 Código de la solución fuerza bruta 7
5.2 Código de la solución con programación dinámica 7

1 Definición del problema

Dado un conjunto de números enteros positivos $S = \{a_1, a_2, \ldots, a_n\}$ y un entero positivo T, determinar si existe un subconjunto $S' \subseteq S$ tal que la suma de los elementos de S' sea igual a T, es decir:

$$\sum_{a_i \in S'} a_i = T$$

Entrada: Un conjunto de números enteros S y un entero T.

Salida: "Sí" si existe un subconjunto $S' \subseteq S$ tal que la suma de sus elementos sea igual a T, de lo contrario "No".

2 Demostración de NP-completitud

Para demostrar que el Subset Sum es NP-completo, debemos demostar que es NP y NP-Duro. La demostración de que se trata de un problema NP es trivial ya que dado un subconjunto A con $|A| = n$, se puede verificar en tiempo polinómico O(n) si la de los elementos $\sum_{a_i \in A} a_i = T$.

Para la demostración de que Subset Sum es NP-Duro usaremos otro problema NP-Duro para hacer una reducción polinómica al primero. A continuación, mostramos que 3SAT $\leq$ Subset Sum. Sea F una fórmula en 3CNF con n variables $x_1, x_2, \ldots, x_n$ y m cláusulas $C_1, C_2, \ldots, C_m$. Para cada variable x_i, construimos dos enteros decimales positivos b_{x_i} y $b_{\overline{x}_i}$, que representan los dos literales x_i y $\overline{x}_i$, respectivamente. Cada b_{x_i} ($b_{\overline{x}_i}$) contiene $m + n$ dígitos. Sea $b_{x_i}[k]$ ($b_{\overline{x}_i}[k]$) el k-ésimo dígito contando desde la derecha de b_{x_i} ($b_{\overline{x}_i}$). Definimos lo siguiente:

$$b_{x_i}[k] = b_{\overline{x}_i}[k] = \begin{cases} 1 & \text{si } k = i, \\ 0 & \text{en otro caso.} \end{cases}$$

Esto se usa para registrar la "identidad" de la variable x_i. Para registrar la relación entre literales y cláusulas, definimos:

$$b_{x_i}[n+j] = \begin{cases} 1 & \text{si } x_i \text{ aparece en la cláusula } C_j, \\ 0 & \text{en otro caso.} \end{cases}$$

y

$$b_{\overline{x}_i}[n+j] = \begin{cases} 1 & \text{si } \overline{x}_i \text{ aparece en la cláusula } C_j, \\ 0 & \text{en otro caso.} \end{cases}$$

Finalmente, definimos $2m+1$ enteros positivos adicionales $c_j, \hat{c_j}$ para $1 \leq j \leq m$ y L como sigue:

$$c_j[k] = \hat{c_j}[k] = \begin{cases} 1 & \text{si } k = n+j, \\ 0 & \text{en otro caso.} \end{cases}$$

$$L = \underbrace{3 \ldots 3}_{m} \underbrace{1 \ldots 1}_{n}.$$

Ahora, empleamos esta construcción para demostrar que F tiene una asignación satisfactoria si y solo si existe un subconjunto $A' \subseteq A = \{b_{i,j} | 1 \leq n, j = 0, 1\} \cup \{c_j, \hat{c_j} | 1 \leq j \leq m\}$ tal que la suma de todos los enteros en A' es igual a L.

Primero, supongamos que F tiene una asignación satisfactoria σ y conformemos el subconjunto A'. Para cada variable x_i, se incluye b_{x_i} en A' si $x_i = 1$ bajo la asignación σ, o se incluye $b_{\overline{x}_i}$ en A' si $x_i = 0$ bajo la asignación σ, como esto es una asignación válida solo puede ocurrir uno y solo uno de los dos casos. Para cada cláusula C_j, se incluye tanto c_j como $\hat{c}_j$ en A' si C_j contiene exactamente un literal satisfecho bajo la asignación σ; incluye solo c_j en A' si C_j contiene exactamente dos literales satisfechos bajo la asignación σ; y no se incluye ni c_j ni $\hat{c}_j$ en A' si los tres literales en C_j están satisfechos bajo la asignación σ.

Por el proceso descrito, el conjunto A' obtenido cumple la condición de que la suma de todos los números en A' es igual a L.

Supongamos que existe un subconjunto A' de A tal que la suma de todos los números en A' es igual a L. Dado que $L[i] = 1$ para $1 \leq i \leq n$, A' contiene exactamente uno de b_{x_i} y $b_{\overline{x}_i}$. Define una asignación σ estableciendo:

$$x_i = \begin{cases} 1 & \text{si } b_{x_i} \in A', \\ 0 & \text{si } b_{\overline{x}_i} \in A'. \end{cases}$$

Afirmamos que σ es una asignación satisfactoria para F. De hecho, para cualquier cláusula C_j, como $L[n+j] = 3$, debe haber un b_{x_i} o $b_{\overline{x}_i}$ en A' cuyo $(n+j)$-ésimo dígito desde la izquierda sea 1. Esto significa que hay un literal con asignación 1 que aparece en C_j, es decir, que hace que C_j esté satisfecha.

3 Fuerza Bruta

Dado un conjunto $S = \{S_1, S_2, ..., S_n\}$ y un entero T, el algoritmo explora todos los subconjuntos de S mediante un enfoque de backtracking. Por cada estado, se siguen los siguientes pasos:

- Se verifica si el subconjunto actual tiene algún elemento que cumpla con la condición de sumar T.
- Se realiza una llamada recursiva para considerar los elementos restantes del conjunto.
- El algoritmo retrocede para explorar otras combinaciones posibles, volviendo al estado anterior.

Complejidad Temporal

El algoritmo evalúa todos los subconjuntos posibles de S. Dado que cada subconjunto tiene una suma que se obtiene durante el proceso de generación, la complejidad temporal es $O(2^n)$.

4 Dinámica Pseudopolinomial

El enfoque consiste en mantener un arreglo unidimensional de tamaño $T + 1$, donde cada entrada $dp[j]$ es un valor booleano que indica si es posible obtener la suma j utilizando un subconjunto de los elementos de S. Se itera sobre cada elemento $s_i \in S$ y se actualiza dp en orden inverso, es decir, desde T hasta el valor s_i, para evitar sobreescribir valores previamente marcados como verdaderos en el mismo paso. Si $dp[j - s_i]$ es verdadero, entonces se establece $dp[j] = \text{True}$.

Correctitud del Algoritmo

Demostraremos la correctitud del algoritmo por inducción sobre los elementos del conjunto S que han sido procesados.

Caso base: Justo antes de procesar el primer elemento de S, se tiene que $dp[0] = \text{True}$. Esto es correcto, ya que el subconjunto vacío siempre suma 0. Por lo tanto, todos los números que pueden ser representados por el subconjunto vacío están marcados como verdaderos en dp, es decir, solo el valor 0.

Paso inductivo: Supongamos que al procesar los primeros $i - 1$ elementos, $dp[j]$ es verdadero si podemos formar la suma j con un subconjunto de $\{s_1, s_2, \ldots, s_{i-1}\}$. Ahora, al procesar el elemento s_i, para cada j tal que $T \geq j \geq s_i$, si $dp[j - s_i]$ es verdadero, significa que podemos formar la suma $j - s_i$ con los primeros $i - 1$ elementos, y por lo tanto, agregando s_i, podemos formar la suma j.

Por lo tanto, al finalizar el algoritmo, $dp[T]$ será verdadero si y solo si es posible formar la suma T con un subconjunto de S, lo que garantiza la correctitud del algoritmo.

Complejidad del Algoritmo

Aunque la implementación sugiere que la complejidad temporal es $O(n \cdot T)$, no estamos ante un algoritmo polinomial en el tamaño de la entrada. La complejidad del algoritmo debe analizarse en función de la longitud de la entrada.

Supongamos que los valores de los elementos de T y S están en el rango $[1, \ldots, 2^k]$, lo que implica que se necesitan aproximadamente k bits para representar cada número. El tamaño total de la entrada está determinado por n, la cantidad de elementos en S, y k, el número de bits necesarios para representar los valores en T y S. Por lo tanto, el tamaño real de la entrada es $O(n \cdot 2^k)$, un término exponencial en función de k.

5 Anexos

5.1 Código de la solución fuerza bruta

```python
def _backtrack(S, index, subsequence, aux, T):
    # Caso base: si ya procesamos todos los enteros de S
    if index == len(S):
        return T == aux

    # Excluimos el elemento en la posici n actual
    x = _backtrack(S, index + 1, subsequence, aux, T)

    # Incluimos el elemento en la posici n actual
    subsequence.append(S[index])
    y = _backtrack(S, index + 1, subsequence, aux + S[index], T)
    subsequence.pop()

    return x or y

def backtrack(S, T):
    return _backtrack(S, 0, [], 0, T)
```

5.2 Código de la solución con programación dinámica

```python
def subset_sum_dp(S, T):
    dp = [False] * (T + 1)
    dp[0] = True

    for s in S:
        # Recorremos en orden inverso
        for j in range(T, s - 1, -1):
            if dp[j - s]:
                dp[j] = True

    return dp[T]
```