## Enunciado

En este ejercicio vamos a resolver el problema de suma de subconjuntos con la técnica de backtracking. Dado un multiconjunto $C = \{c_1, ..., c_n\}$ de números naturales y un natural $k$, queremos determinar si existe un subconjunto de $C$ cuya sumatoria sea $k$. Vamos a suponer fuertemente que $C$ está ordenado de alguna forma arbitraria pero conocida (i.e., $C$ está implementado como la secuencia $c_1, ..., c_n$ o, análogamente, tenemos un iterador de $C$). Las soluciones (candidatas) son los vectores $a = (a_1, ..., a_n)$ de valores binarios; el subconjunto de $C$ representado por $a$ contiene a $c_i$ si y sólo si $a_i = 1$. Luego, a es una solución válida cuando $\sum_{i=1}^{n} a_i c_i = k$. Asimismo, una solución parcial es un vector $p = (a_1, ..., a_i)$ de números binarios con $0 \leq i \leq n$. Si $i < n$, las soluciones sucesoras de $p$ son $p \oplus 0$ y $p \oplus 1$, donde $\oplus$ indica la concatenación.

### a) Escribir el conjunto de soluciones candidatas para $C = \{6, 12, 6\}$ y $k = 12$.

Soluciones candidatas: $\{(0,0,0), (1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1), (0,1,1), (1,1,1)\}$ o sea, toda combinación posible.

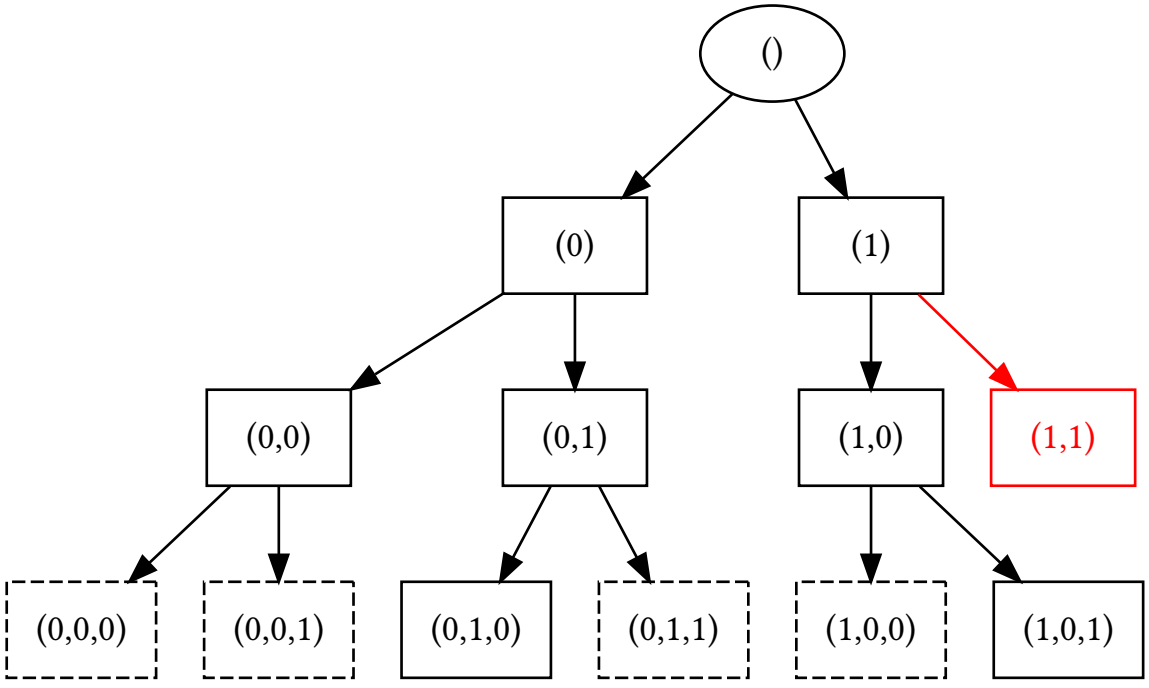
### b) Escribir el conjunto de soluciones válidas para $C = \{6, 12, 6\}$ y $k = 12$.

Conjunto de soluciones válidas: $\{(1,0,1), (0,1,0)\}$

### c) Escribir el conjunto de soluciones parciales para $C = \{6, 12, 6\}$ y $k = 12$.

Conjunto de soluciones parciales: $\{(), (1), (0), (0,0), (0,1), (1,0), (1,1)\}$

### d) Dibujar el árbol de *backtracking* correspondiente al algoritmo descrito arriba para $C = \{6, 12, 6\}$ y $k = 12$, indicando claramente la relación entre las distintas componentes del árbol y los conjuntos de los incisos anteriores.



## e) (SimpleTex no lo entiende y no lo voy a escribir a mano)

Sea $\mathcal{C}$ la familia de todos los multiconjuntos de números naturales. Considerar la siguiente función recursiva ss: $\mathcal{C} \times \mathbb{N} \to \{V, F\}$ (donde $\mathbb{N} = \{0, 1, 2, \dots\}$, $V$ indica verdadero y $F$ falso):

$$\text{ss}(\{c_1, \dots, c_n\}, k) = \begin{cases} k = 0 & \text{si } n = 0 \\ \text{ss}(\{c_1, \dots, c_{n-1}\}, k) \vee \text{ss}(\{c_1, \dots, c_{n-1}\}, k - c_n) & \text{si } n > 0 \end{cases}$$

Convencerse de que $\text{ss}(C, k) = V$ si y sólo si el problema de subconjuntos tiene una solución válida para la entrada $C, k$. Para ello, observar que hay dos posibilidades para una solución válida $a = (a_1, \dots, a_n)$ para el caso $n > 0$: o bien $a_n = 0$ o bien $a_n = 1$. En el primer caso, existe un subconjunto de $\{c_1, \dots, c_{n-1}\}$ que suma $k$; en el segundo, existe un subconjunto de $\{c_1, \dots, c_{n-1}\}$ que suma $k - c_n$.

**Me convencí!**

## f)

Convencerse de que la siguiente es una implementación recursiva de ss en un lenguaje imperativo y de que retorna la solución para $C$, $k$ cuando se llama con $C, |C|, k$. ¿Cuál es su complejidad?

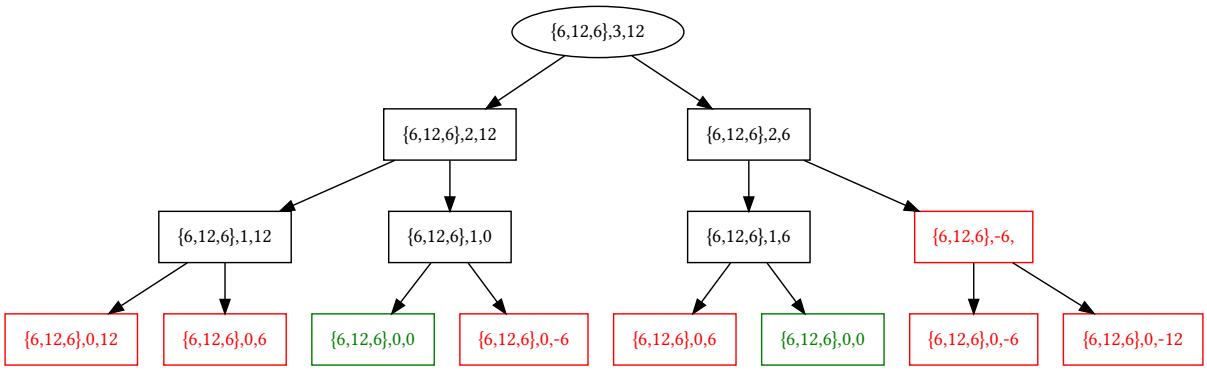
```
1) subset_sum(C, i, j): // implementa ss({c1, . . . , ci}, j)

2) Si i = 0, retornar (j = 0)

3) Si no, retornar subset_sum(C, i - 1, j) v subset_sum(C, i - 1, j - C[i])
```

Me convencí, luego, su complejidad es tal que

2) es $O(1)$

3) Cada subset_sum se extiende a 2 llamadas recursivas por recursión, y hay $|C|$ recursiones, luego tengo $2^{|C|}$ llamadas, cada una de tamaño $O(1)$ por lo que es $O\left(2^{|C|}\right)$

## Dibujar el árbol de llamadas recursivas para la entrada $C = \{6, 12, 6\}$ y $k = 12$, y compararlo con el árbol de backtracking.




Podemos notar que el árbol es análogo al de backtracking, con la diferencia de que no hay podas de ningún tipo tal que no haga recursiones de más.

## h)

Considerar la siguiente regla de factibilidad : $p = (a_1, ..., a_i)$ se puede extender a una solución válida sólo si $\sum_{q=1}^{i} a_q c_q \leq k$. Convencerse de que la siguiente implementación incluye la regla de factibilidad.

```
1) subset_sum(C, i, j): // implementa ss({c1, . . . , ci}, j)
2) Si j < 0, retornar falso // regla de factibilidad
3) Si i = 0, retornar (j = 0)
4) Si no, retornar subset_sum(C, i − 1, j) v subset_sum(C, i − 1, j − C[i])
```

**Rta:** Convencido 😀

## i)

Definir otra regla de factibilidad, mostrando que la misma es correcta; no es necesario implementarla.

**Sol:** Una regla de factibilidad podría ser

## j)

Modificar la implementación para imprimir el subconjunto de C que suma k, si existe. Ayuda: mantenga un vector con la solución parcial p al que se le agregan y sacan los elementos en cada llamada recursiva; tenga en cuenta de no suponer que este vector se copia en cada llamada recursiva, porque cambia la complejidad.

```python
def ss(C, i, j):
  if j < 0:
    return False
  if i = 0:
    return j == 0
  return ss(C, i-1, j) or ss(C, i-1, j-C[i])
```