

# Taller 2b – IIC2115

Matías Gaete Silva – [mzgaete@uc.cl](mailto:mzgaete@uc.cl)

## Ejercicio 1: profundidad de un árbol binario

Escriba un programa que calcule mediante recursión, la profundidad de un árbol binario, es decir, la cantidad de arcos que tiene el camino más largo desde el nodo raíz hasta un nodo hoja. Empaque el algoritmo en una función que reciba como argumento una lista de tuplas de 3 elementos, que denotan el nodo padre, hijo izquierdo e hijo derecho, respectivamente. Cada nodo se identificará de manera única con un número natural. La ausencia de un hijo será representada a través de `None`, mientras que los nodos hoja no tendrán asociada una tupla con ellos como nodo padre. Puede asumir que la primera tupla de la lista representa al nodo raíz. La función debe retornar un número natural, que indique la profundidad del árbol. Un ejemplo de ejecución del algoritmo es el siguiente:

**Código**

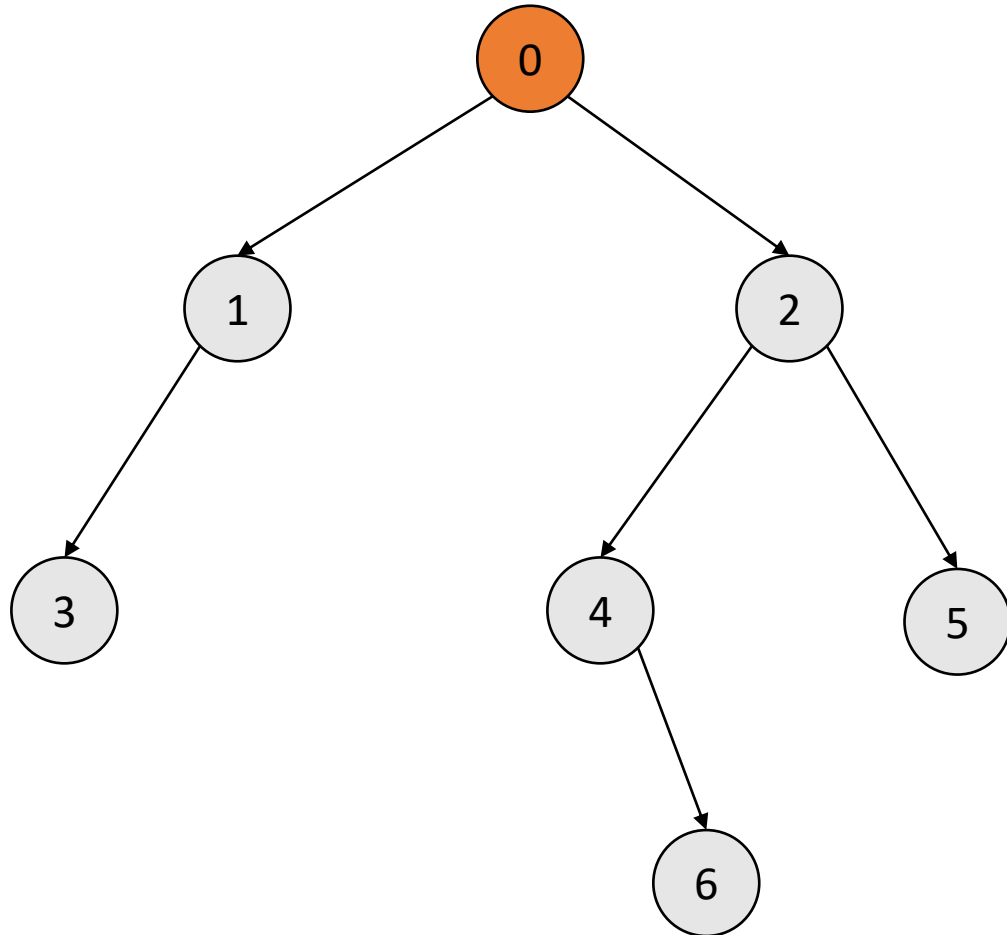
```
arbol = [(0,1,2), (4,None,6), (1,3,None), (2,4,5)]  
  
profundidad = profundidad_arbol_binario(arbol)  
  
print(profundidad)
```

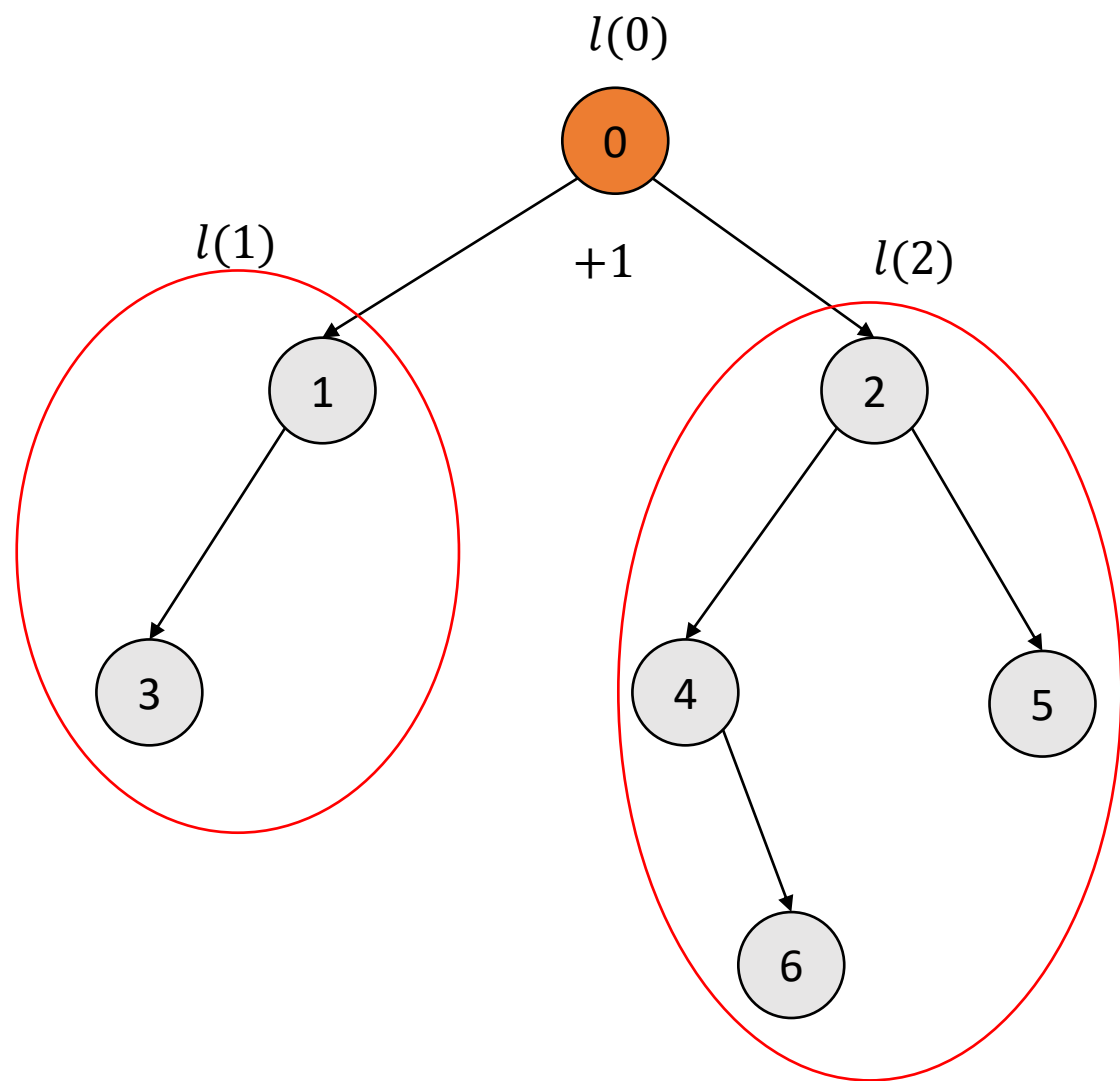
**Salida**

3

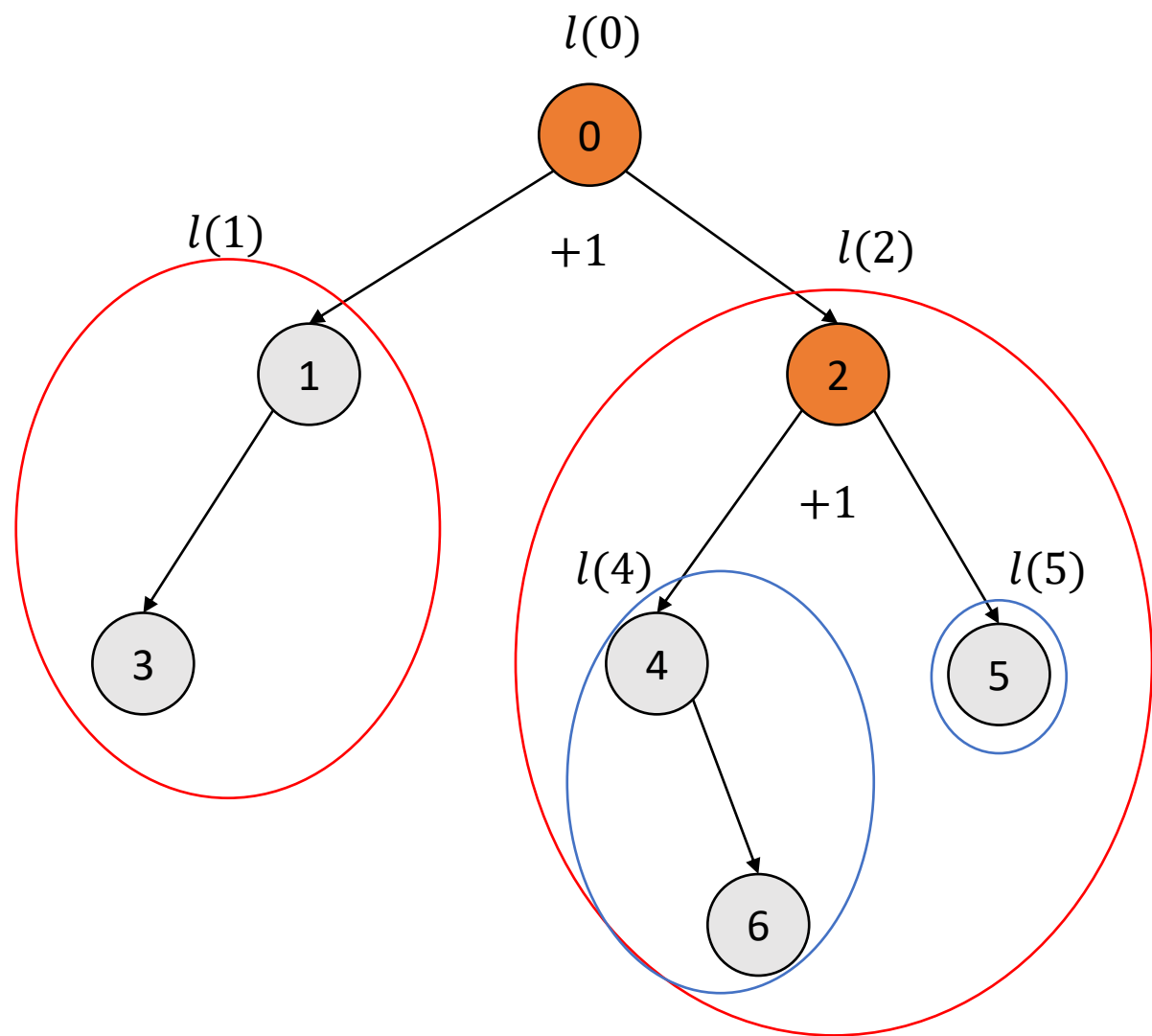
$l(i)$ : profundidad del árbol binario con raíz  $i$   
Queremos encontrar  $l(0)$

$l(0) = ?$



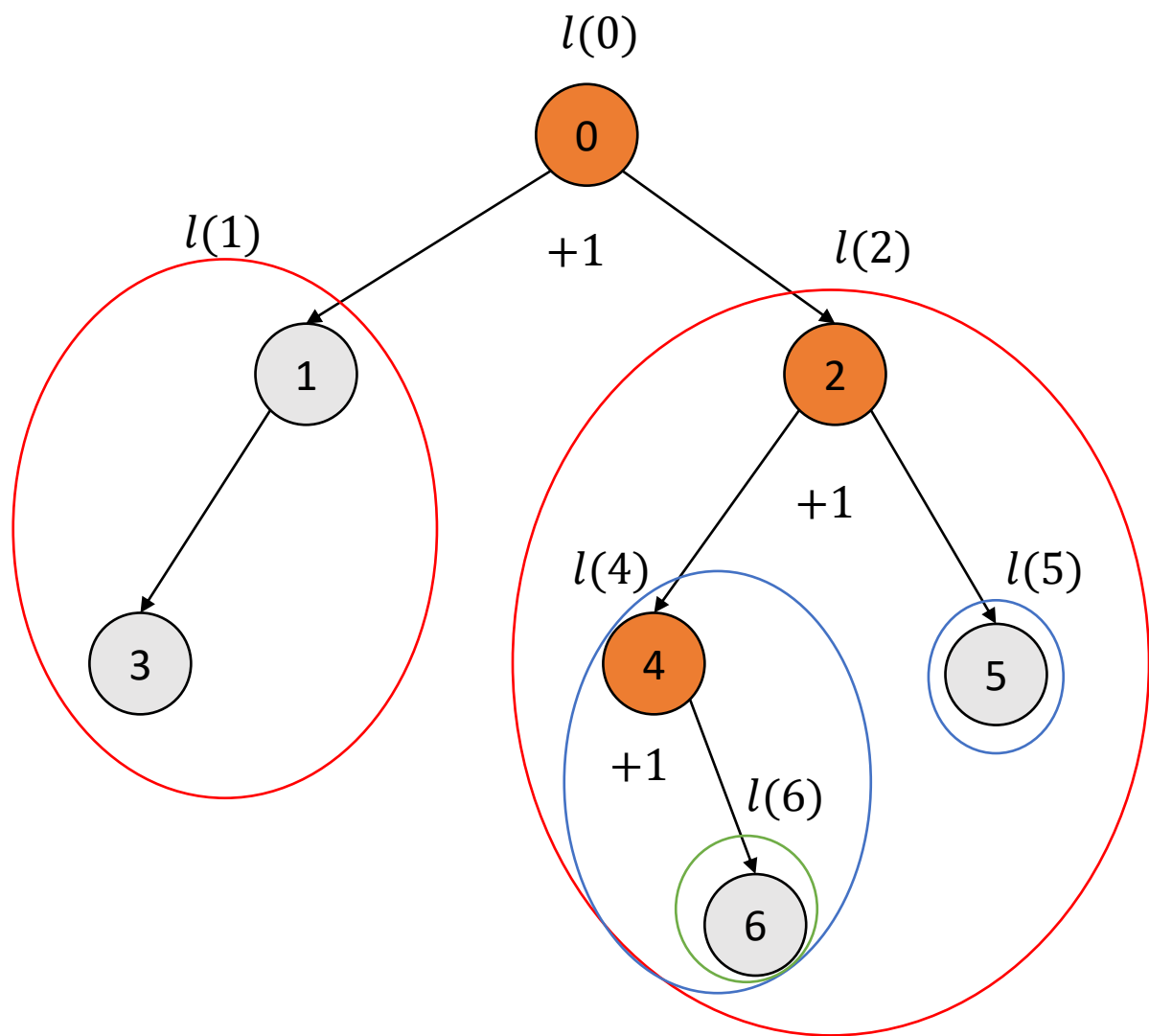


$$l(0) = \max\{l(1), l(2)\} + 1$$



$$l(0) = \max\{l(1), l(2)\} + 1$$

$$l(2) = \max\{l(4), l(5)\} + 1$$

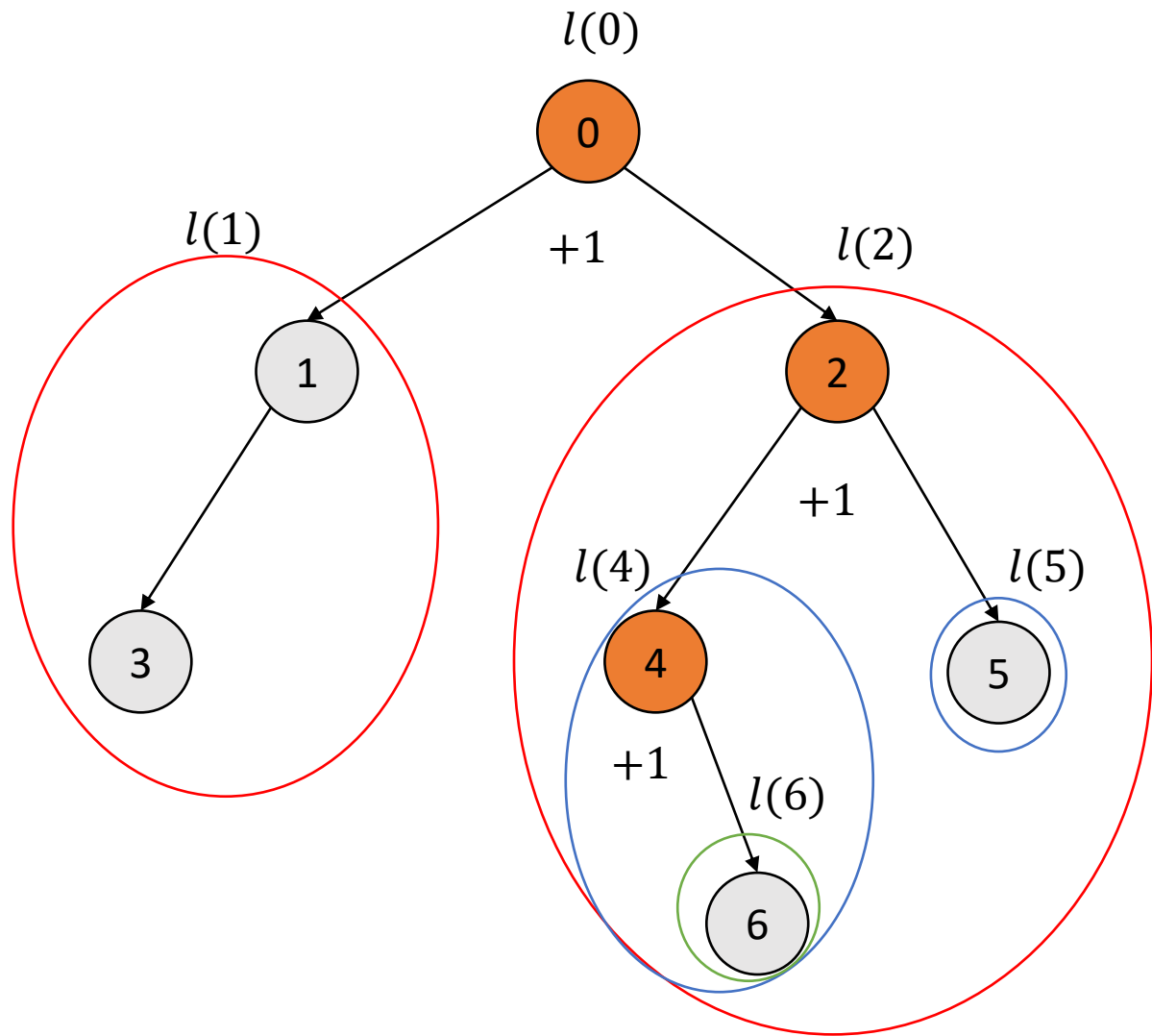


$$l(0) = \max\{l(1), l(2)\} + 1$$

$$l(2) = \max\{l(4), l(5)\} + 1$$

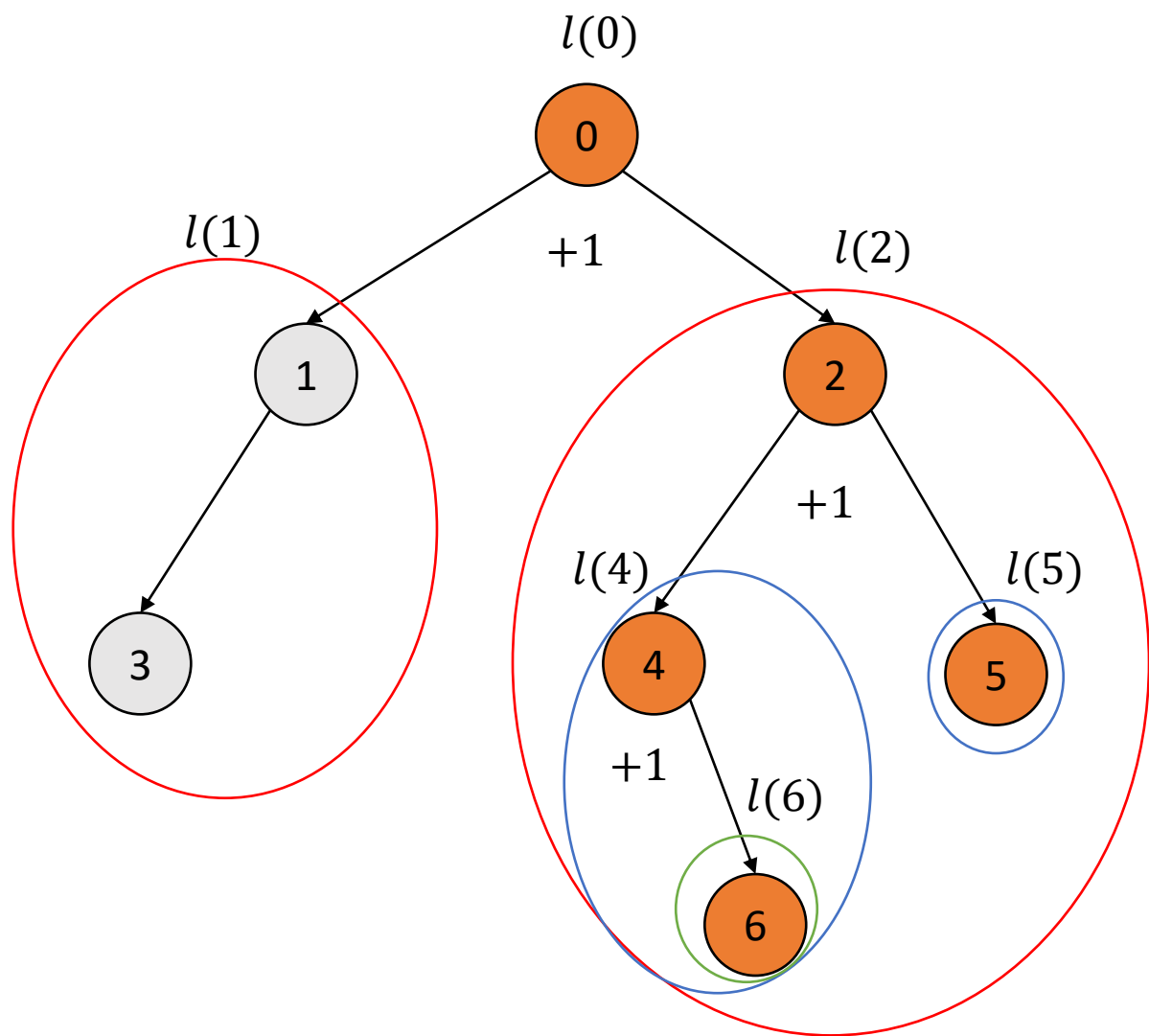
$$l(4) = \max\{l(6)\} + 1 = l(6) + 1$$

¿ $l(6)$  y  $l(5)$ ?



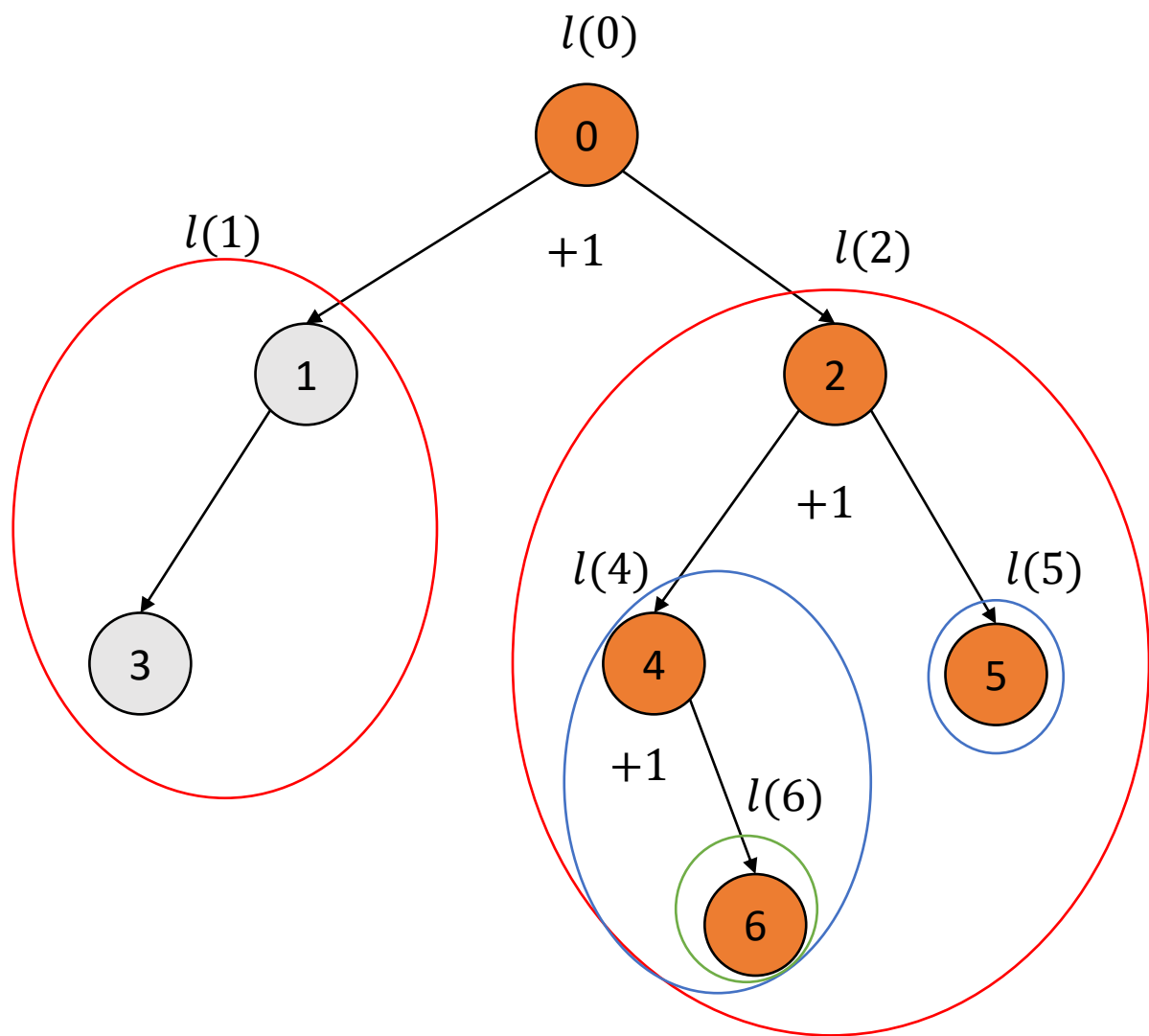
$$\begin{aligned}l(0) &= \max\{l(1), l(2)\} + 1 \\l(2) &= \max\{l(4), l(5)\} + 1 \\l(4) &= \max\{l(6)\} + 1 = l(6) + 1\end{aligned}$$

¿ $l(6)$  y  $l(5)$ ?  
Caso base! Si  $j$  es  
nodo hoja, entonces  
 $l(j) = 0$

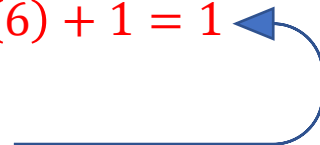


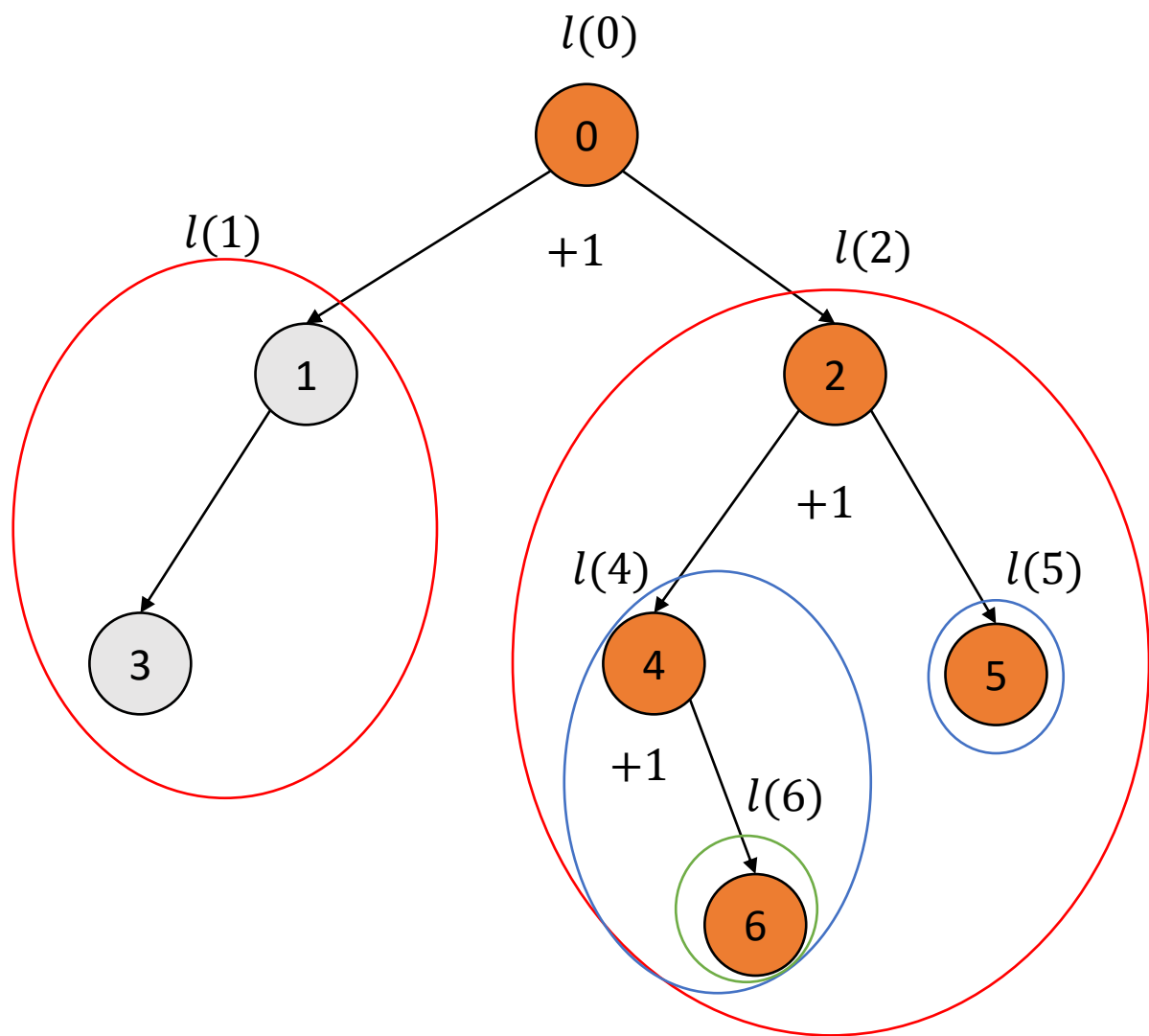
$$\begin{aligned}l(0) &= \max\{l(1), l(2)\} + 1 \\l(2) &= \max\{l(4), l(5)\} + 1 \\l(4) &= \max\{l(6)\} + 1 = l(6) + 1 \\l(5) &= 0 \\l(6) &= 0\end{aligned}$$





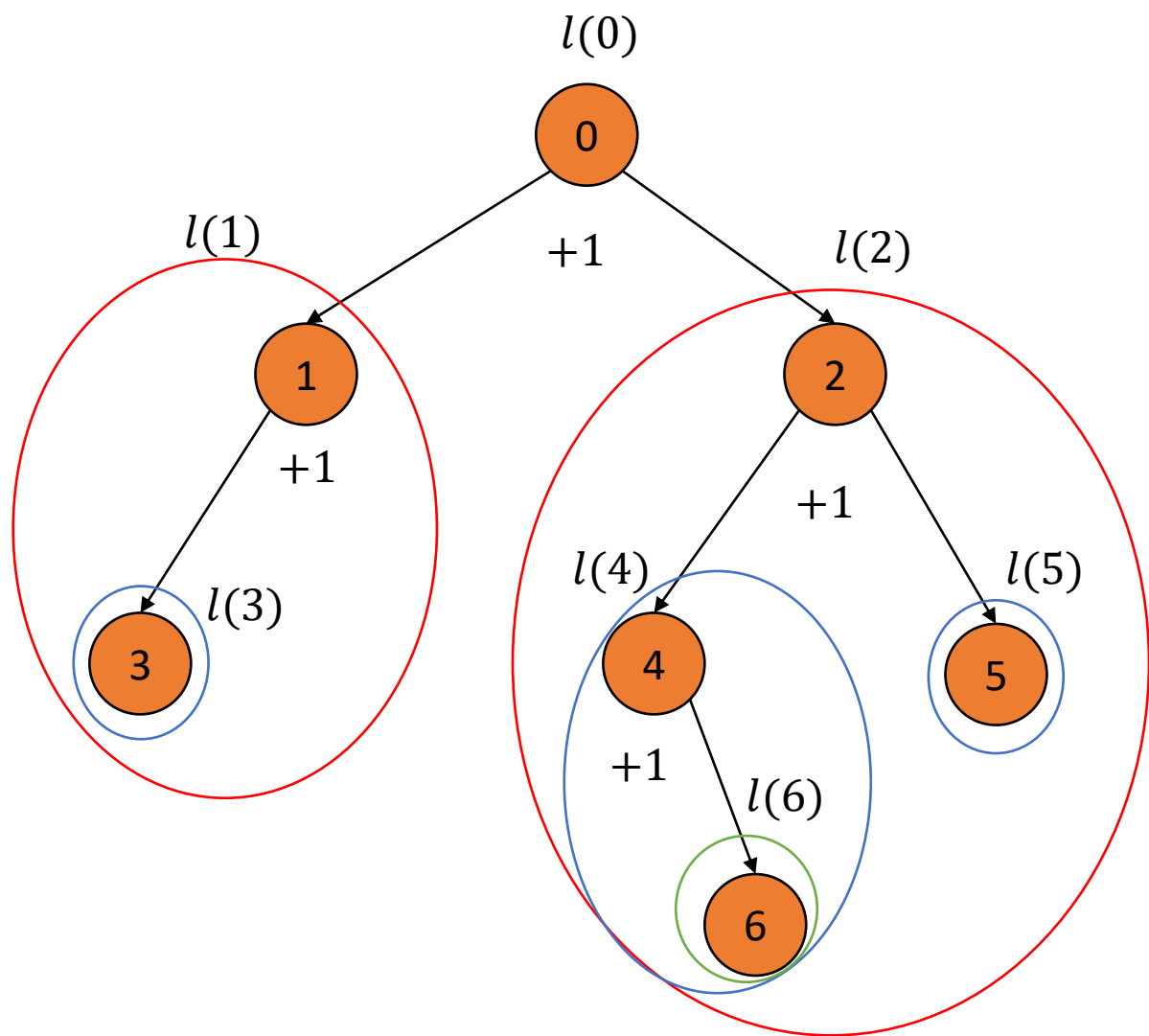
$$\begin{aligned}l(0) &= \max\{l(1), l(2)\} + 1 \\l(2) &= \max\{l(4), l(5)\} + 1 \\l(4) &= \max\{l(6)\} + 1 = l(6) + 1 = 1 \\l(5) &= 0 \\l(6) &= 0\end{aligned}$$



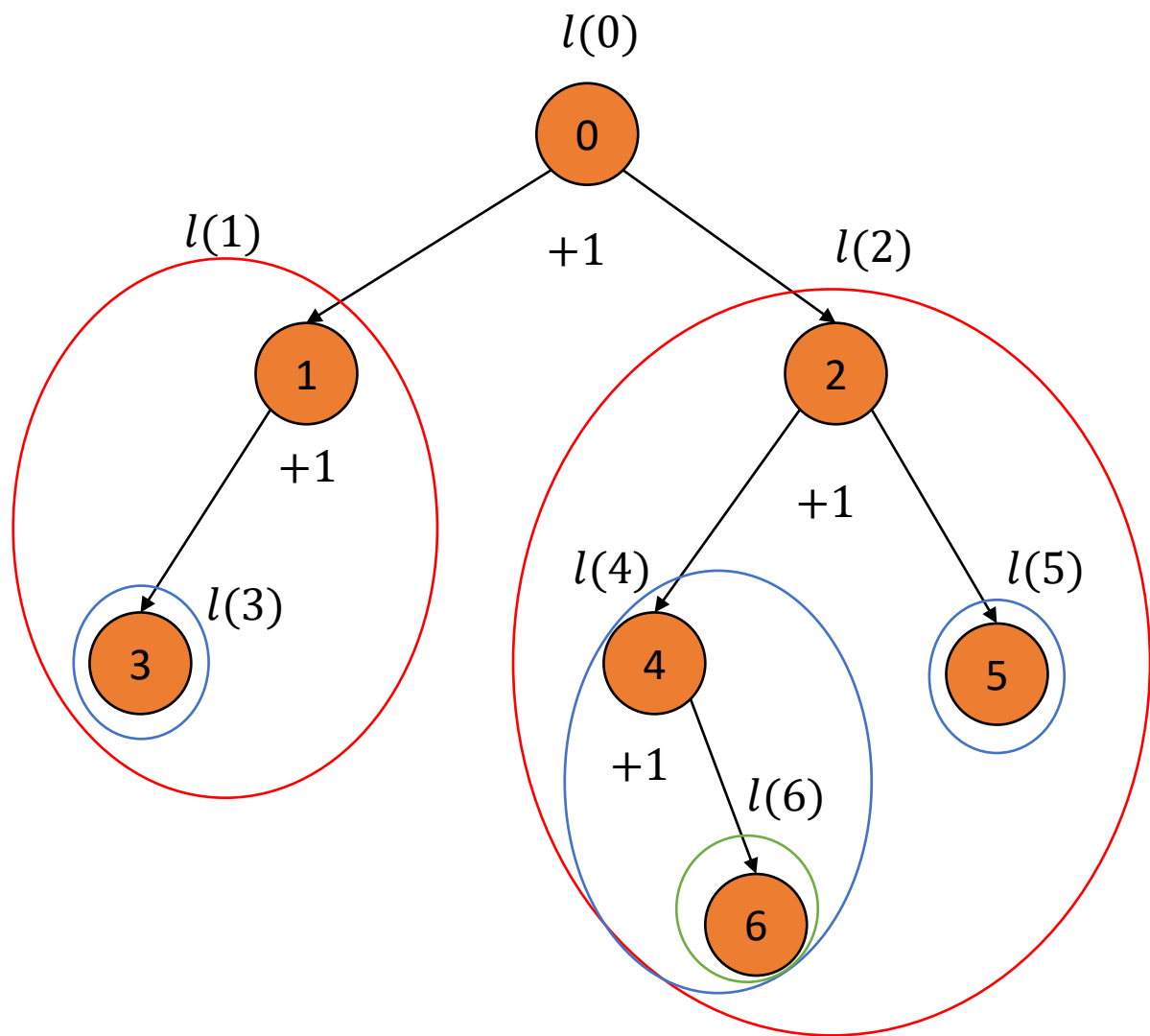


$$\begin{aligned}
 l(0) &= \max\{l(1), l(2)\} + 1 \\
 l(2) &= \max\{l(4), l(5)\} + 1 = 2 \\
 l(4) &= \max\{l(6)\} + 1 = l(6) + 1 = 1 \\
 l(5) &= 0 \\
 l(6) &= 0
 \end{aligned}$$

←



$$\begin{aligned}
 l(0) &= \max\{l(1), l(2)\} + 1 \\
 l(2) &= \max\{l(4), l(5)\} + 1 = 2 \\
 l(4) &= \max\{l(6)\} + 1 = l(6) + 1 = 1 \\
 l(5) &= 0 \\
 l(6) &= 0 \\
 l(1) &= \max\{l(3)\} + 1 = l(3) + 1 \\
 l(3) &= 0
 \end{aligned}$$



$$\begin{aligned}
 l(0) &= \max\{l(1), l(2)\} + 1 = 3 \\
 l(2) &= \max\{l(4), l(5)\} + 1 = 2 \\
 l(4) &= \max\{l(6)\} + 1 = l(6) + 1 = 1 \\
 l(5) &= 0 \\
 l(6) &= 0 \\
 l(1) &= \max\{l(3)\} + 1 = l(3) + 1 = 1 \\
 l(3) &= 0
 \end{aligned}$$

$$l(i) = \begin{cases} \max\{l(\text{hijo}_{izq}(i)), l(\text{hijo}_{der}(i))\} + 1, & i \neq \text{hoja} \\ 0, & i = \text{hoja} \end{cases}$$

Encontrar  $l(\text{raiz})$

$$l(i) = \begin{cases} \max\{l(hijo_{izq}(i)), l(hijo_{der}(i))\} + 1, & i \neq hoja \\ 0, & i = hoja \end{cases}$$

Encontrar  $l(raiz)$

Llevemos esto a la programación... ¿Qué necesitamos?

- Tener acceso a los hijos izquierdo y derecho de cada nodo.
- Tener acceso a la raíz.
- Formar estructura de árbol.

$$l(i) = \begin{cases} \max\{l(hijo_{izq}(i)), l(hijo_{der}(i))\} + 1, & i \neq hoja \\ 0, & i = hoja \end{cases}$$

Encontrar  $l(raiz)$

Llevemos esto a la programación... ¿Qué necesitamos?

- Tener acceso a los hijos izquierdo y derecho de cada nodo.
- Tener acceso a la raíz.
- Formar estructura de árbol.

Crear clase `Nodo` cuyos objetos tengan como atributos a su hijo izquierdo y derecho.

```
def __init__(self):
    self.izq = None
    self.der = None
```

```
def agregar_hijos...
```

$$l(i) = \begin{cases} \max\{l(hijo_{izq}(i)), l(hijo_{der}(i))\} + 1, & i \neq hoja \\ 0, & i = hoja \end{cases}$$

Encontrar  $l(raiz)$

Llevemos esto a la programación... ¿Qué necesitamos?

- Tener acceso a los hijos izquierdo y derecho de cada nodo.
- Tener acceso a la raíz.
- Formar estructura de árbol.

Es el primer elemento de la primera tupla de la lista 😊

$$l(i) = \begin{cases} \max\{l(hijo_{izq}(i)), l(hijo_{der}(i))\} + 1, & i \neq hoja \\ 0, & i = hoja \end{cases}$$

Encontrar  $l(raiz)$

Llevemos esto a la programación... ¿Qué necesitamos?

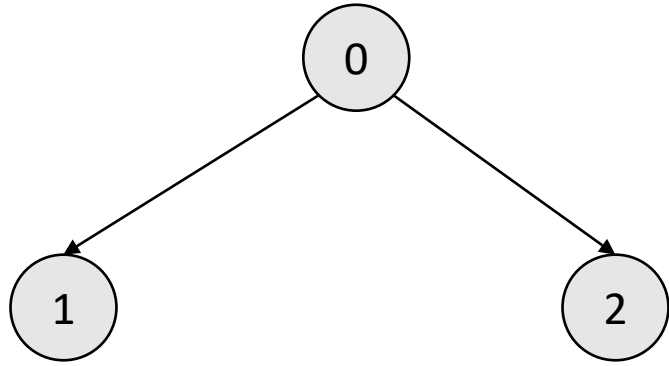
- Tener acceso a los hijos izquierdo y derecho de cada nodo.
- Tener acceso a la raíz.
- Formar estructura de árbol.

Analicemos el formato del input:  $[(0, 1, 2), (4, \text{None}, 6), (1, 3, \text{None}), (2, 4, 5)]$

Al recorrer cada elemento de la lista, sabemos cuál es el nodo padre y cuáles son sus nodos hijos, pero...

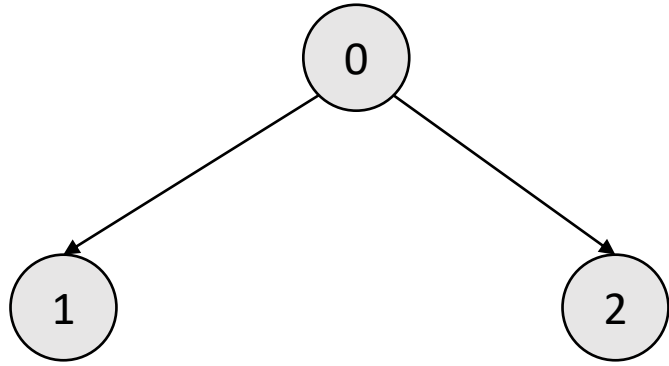


[ (0, 1, 2) , (4, None, 6) , (1, 3, None) , (2, 4, 5) ]



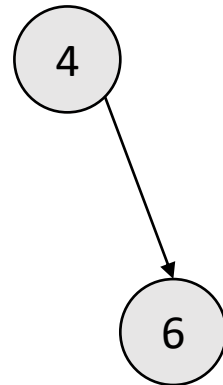
Creamos el nodo 0, el 1 y 2 y los asignamos como hijos al nodo 0. Además, solo para este caso, sabemos que 0 será la raíz.

[ (0, 1, 2), (4, None, 6), (1, 3, None), (2, 4, 5) ]

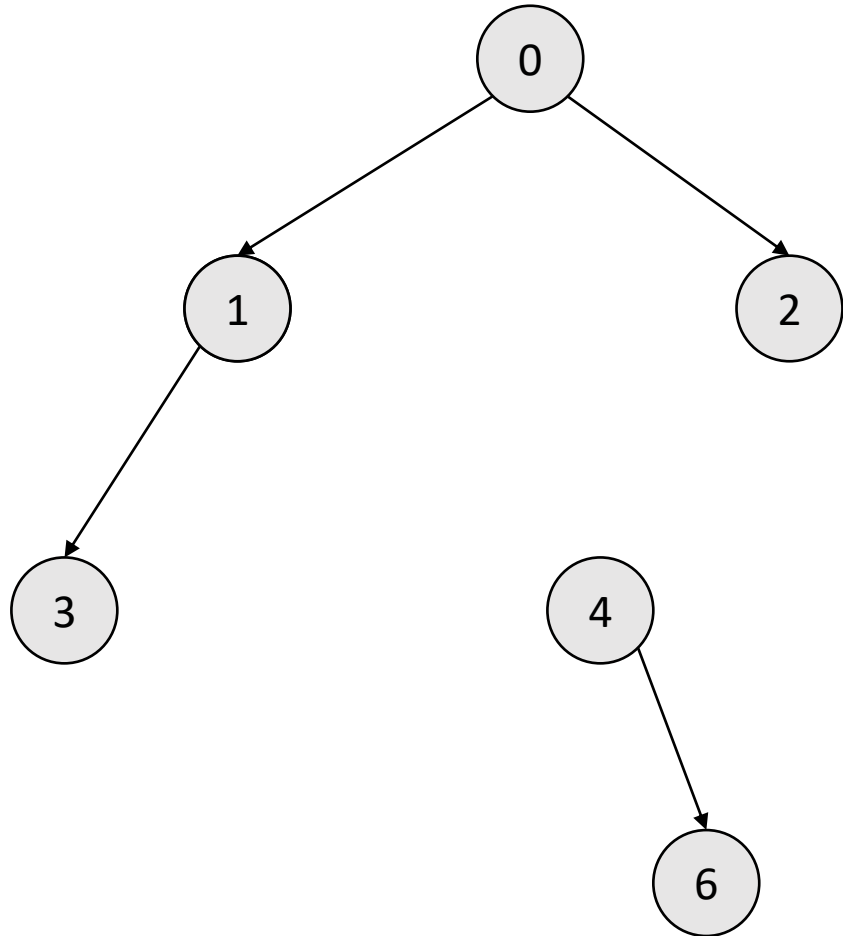


Creamos el nodo 4 y el 6 y lo asignamos como hijo derecho al nodo 4.

¿El nodo 4 será hijo del nodo 1? ¿O del nodo 2?  
¿O de otro nodo?  
Aún no podemos saberlo...

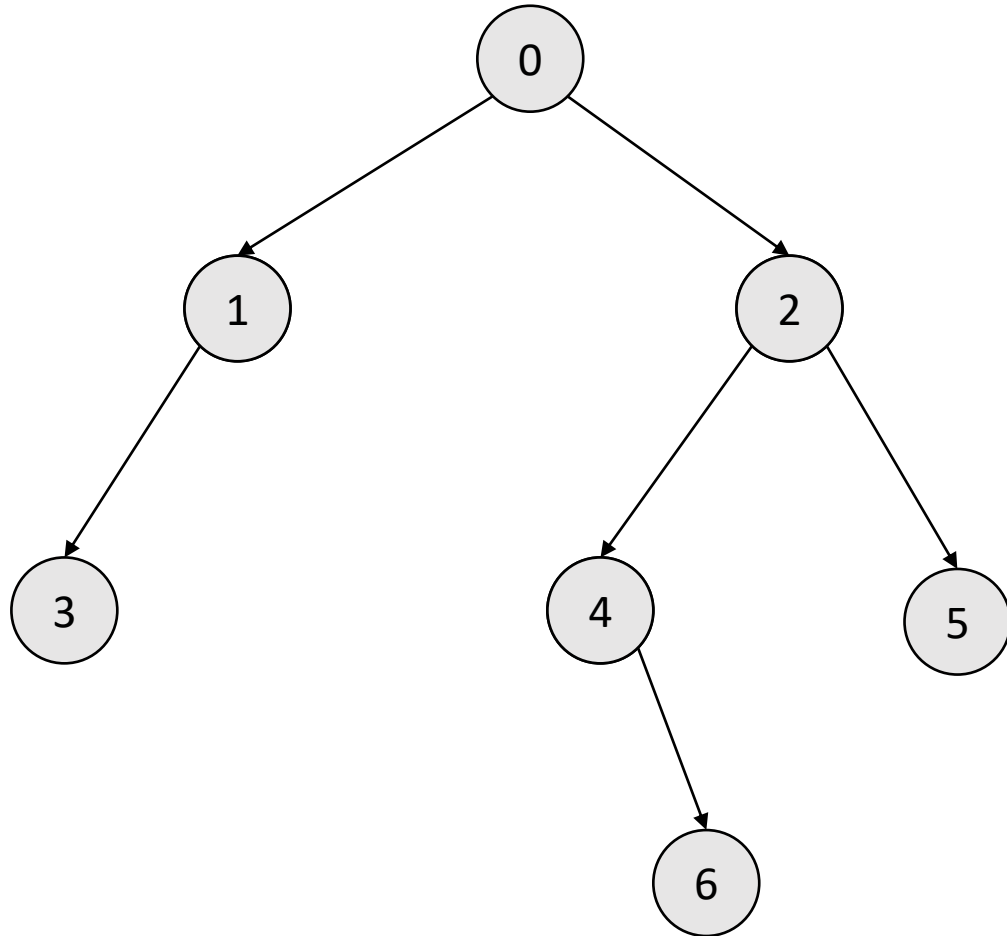


[ (0, 1, 2) , (4, None, 6) , (1, 3, None) , (2, 4, 5) ]



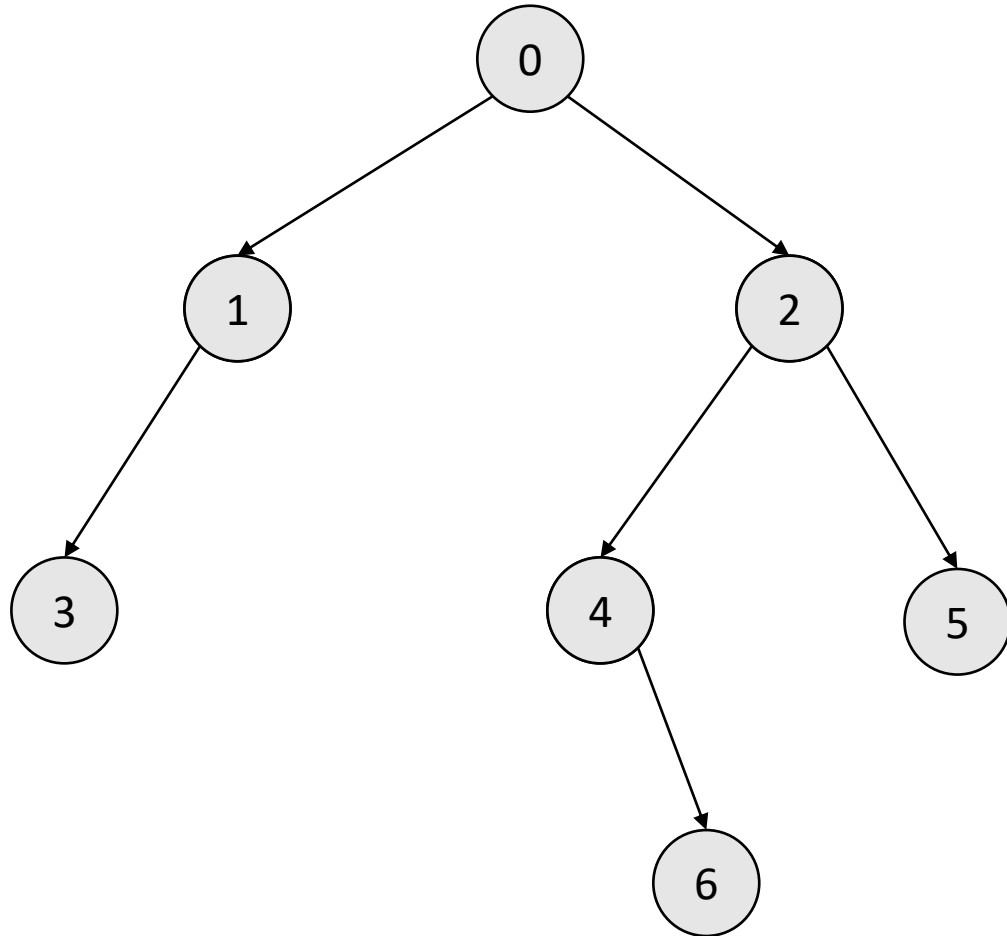
El nodo 1 ya fue creado. Creamos el nodo 3 y lo asignamos como hijo izquierdo al nodo 1.

[ (0,1,2) , (4,None,6) , (1,3,None) , (2,4,5) ]



El nodo 2 y 4 ya fueron creados. Creamos el nodo 5 y asignamos a este nodo junto con el nodo 4 como hijos derecho e izquierdo al nodo 2 respectivamente.

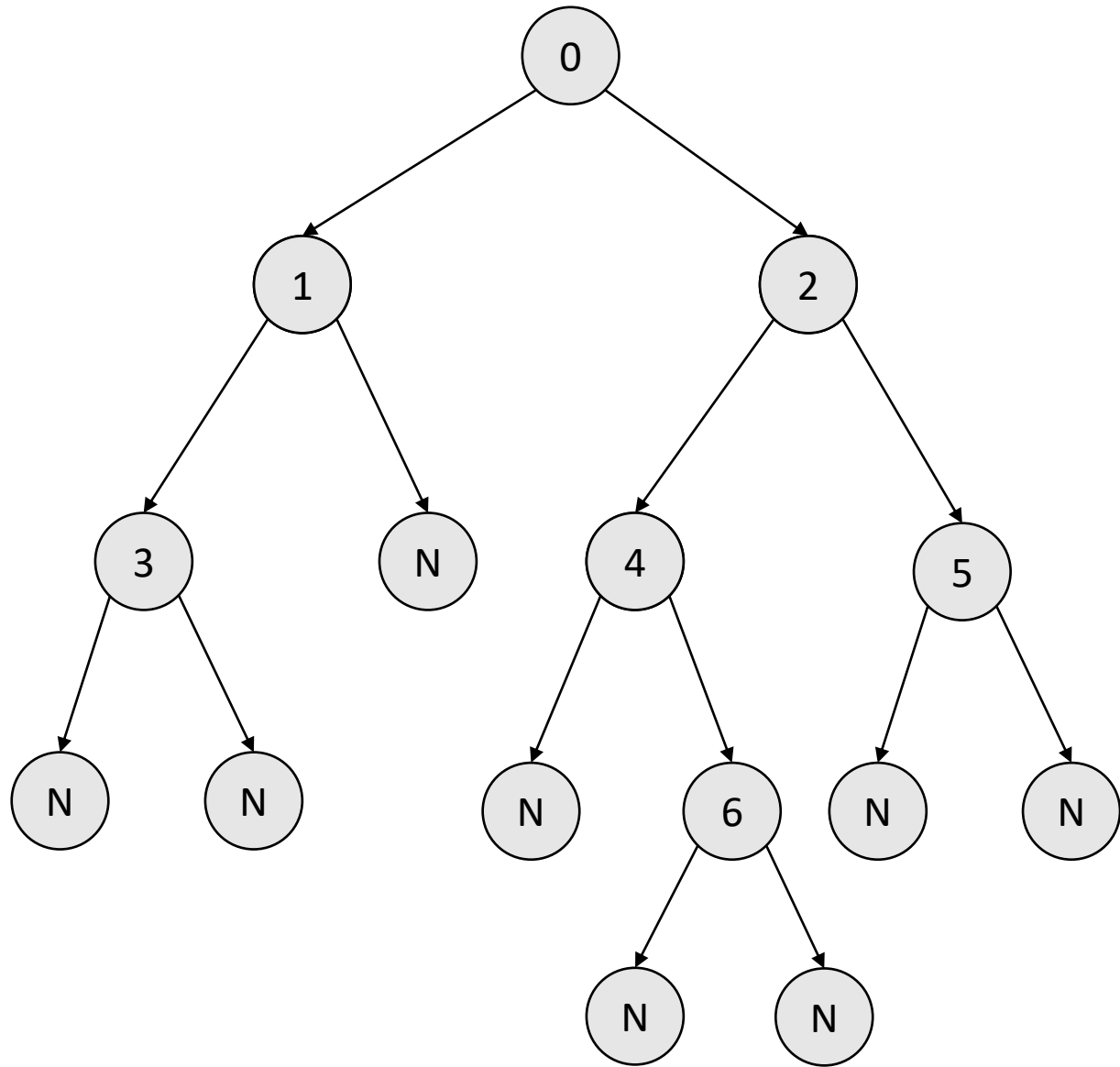
```
[ (0,1,2) , (4,None,6) , (1,3,None) , (2,4,5) ]
```



Idea: Crear clase `ArbolBinario` que tenga como atributos un diccionario, cuyas llaves sean el id y valor el Nodo, y el nodo raíz.

Definir método que reciba los tres elementos de la tupla para agregarlos al árbol. Si el padre o el hijo derecho o el hijo izquierdo no han sido creados, crearlos (siempre que no sean `None`). Asignar hijo izquierdo y derecho al padre.

Un pequeño detalle...



Todos los nodos tendrán asignado un hijo izquierdo y derecho, aunque en realidad no lo tengan (son `None`).  
Nuevo caso base: si  $i$  es `None`, retornar  $-1$ .

$$l(i) = \begin{cases} \max\{l(i.izq), l(i.der)\} + 1, & i \neq \text{None} \\ -1, & i = \text{None} \end{cases}$$

Encontrar  $l(\text{raiz})$

## Ejercicio 2: grupos de estudio

Considere un conjunto de estudiantes, donde cada uno tiene asociado el puntaje que obtuvo en la última prueba rendida. Asumiendo que este puntaje corresponde a un número natural positivo, escriba un programa basado en *backtracking*, que dado el conjunto de alumnos y un número natural positivo  $K$ , entregue  $K$  grupos de estudio disjuntos, tales que la suma de los puntajes de los alumnos en cada grupo sea la misma. El retorno debe ser una lista de listas, donde cada sublista representa un grupo de estudio y contiene los índices de los alumnos que la componen. Un ejemplo de ejecución del algoritmo es el siguiente:

### Código

```
puntajes = [7,3,5,12,2,1,5,3,8,4,6,4]
K = 5
grupos = grupos_estudio(puntajes, K)
print(grupos)
```

### Salida

```
[[4,10,9], [8,11], [1,5,2,7], [3], [0,6]]
```

Antes del *backtracking*, analicemos el problema...

Si tengo números  $n_1, \dots, n_m$  y quiero formar  $K$  grupos de estos números que sumen lo mismo, ¿cuánto debe sumar cada grupo?



Antes del *backtracking*, analicemos el problema...

Si tengo números  $n_1, \dots, n_m$  y quiero formar  $K$  grupos de estos números que sumen lo mismo, ¿cuánto debe sumar cada grupo?

$$\frac{\sum_{i=1}^m n_i}{K}$$

Antes del *backtracking*, analicemos el problema...

Si tengo números  $n_1, \dots, n_m$  y quiero formar  $K$  grupos de estos números que sumen lo mismo, ¿cuánto debe sumar cada grupo?

$$\frac{\sum_{i=1}^m n_i}{K}$$

Si los  $n_i$  son naturales y la sumatoria anterior no es divisible por  $K$ , ¿se pueden formar los  $K$  grupos?

Antes del *backtracking*, analicemos el problema...

Si tengo números  $n_1, \dots, n_m$  y quiero formar  $K$  grupos de estos números que sumen lo mismo, ¿cuánto debe sumar cada grupo?

$$\frac{\sum_{i=1}^m n_i}{K}$$

Si los  $n_i$  son naturales y la sumatoria anterior no es divisible por  $K$ , ¿se pueden formar los  $K$  grupos?

No

```
puntajes = [6,4,1,5,3]  
K = 3
```

```
suma_puntajes = 19  
suma_puntajes / K = 6.33333...
```



```
return [[], [], []]
```

```
puntajes = [6,4,1,5,2]  
K = 3
```

```
suma_puntajes = 18  
target = suma_puntajes / K = 6  
grupos = [[] for i in range(K)]  
backtracking...  
return grupos
```

Tomamos un elemento de `puntajes` e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, debemos volver un paso atrás. Si no quedan elementos por colocar, tenemos una configuración válida.

```
target = 6
```

```
puntajes = [6,4,1,5,2]
```

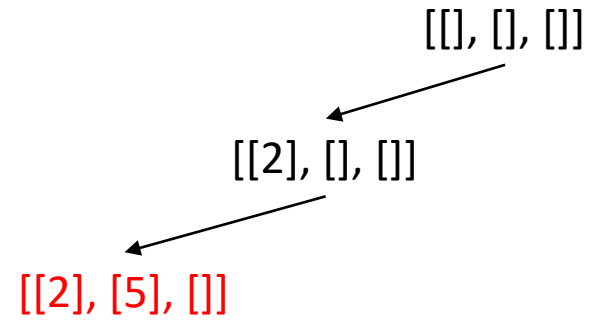
```
[[], [], []]
```

```
target = 6  
puntajes = [6, 4, 1, 5, 2]
```

`[]`, `[]`, `[]`  
↙  
`[2]`, `[]`, `[]`

```
puntajes = [6, 4, 1, 5]
```

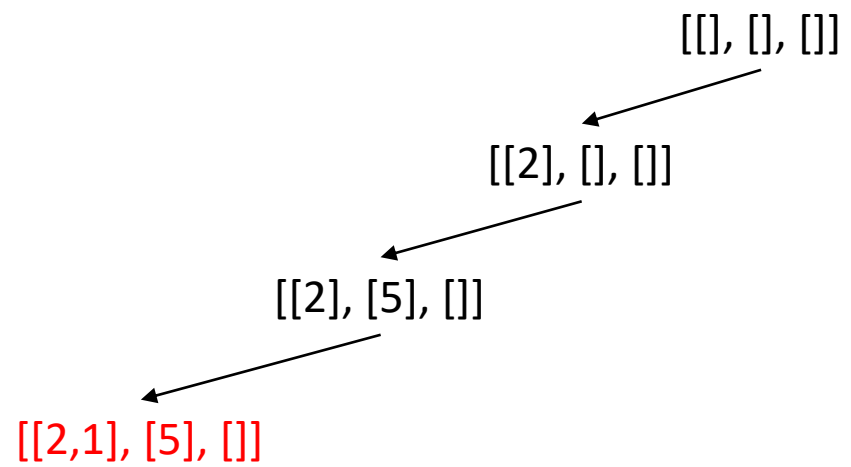
```
target = 6  
puntajes = [6, 4, 1, 5]
```



```
puntajes = [6, 4, 1]
```

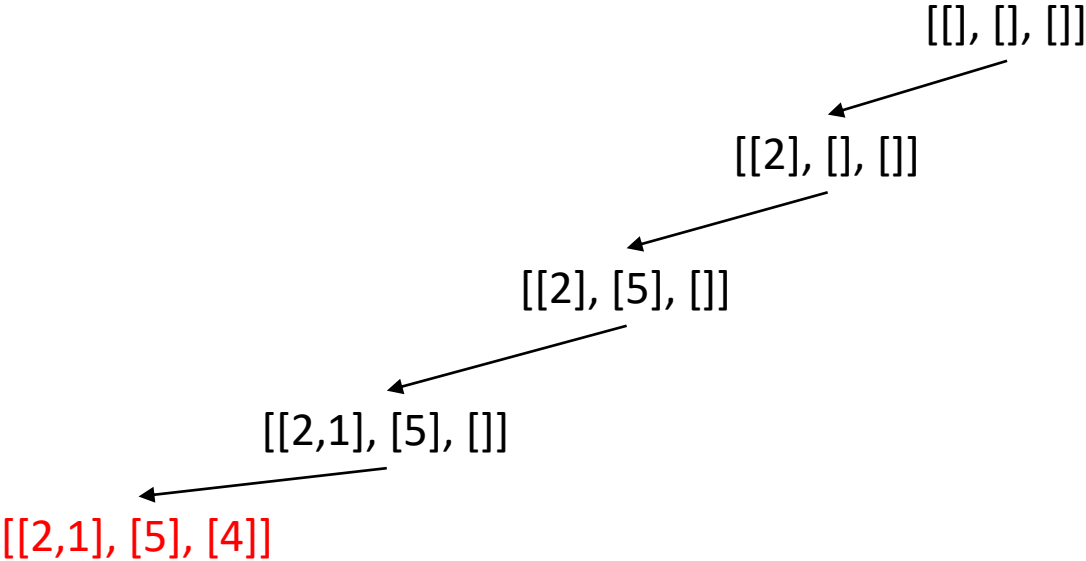


```
target = 6  
puntajes = [6, 4, 1]
```



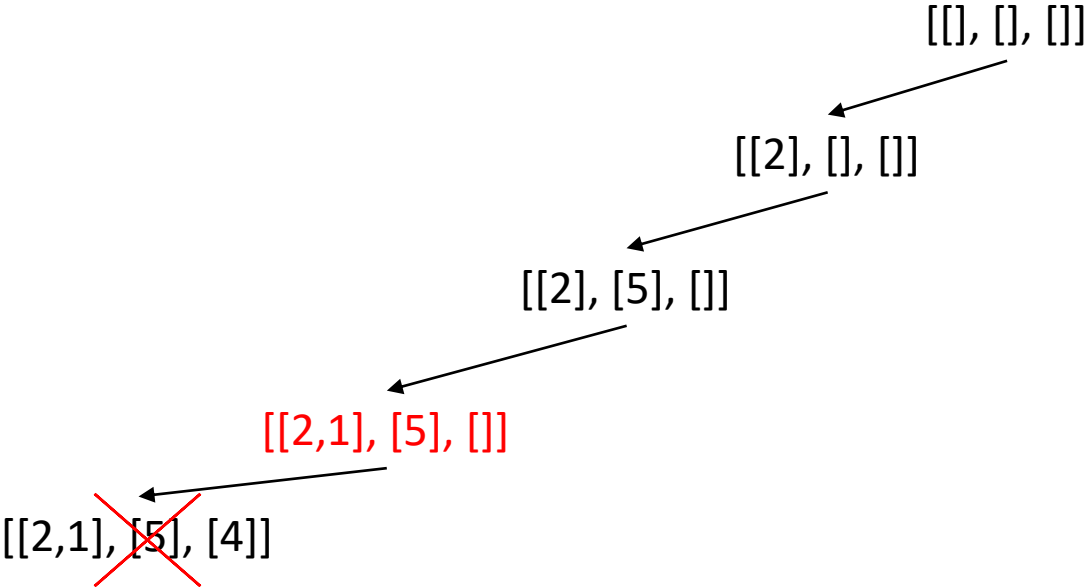
```
puntajes = [6, 4]
```

```
target = 6
puntajes = [6, 4]
```



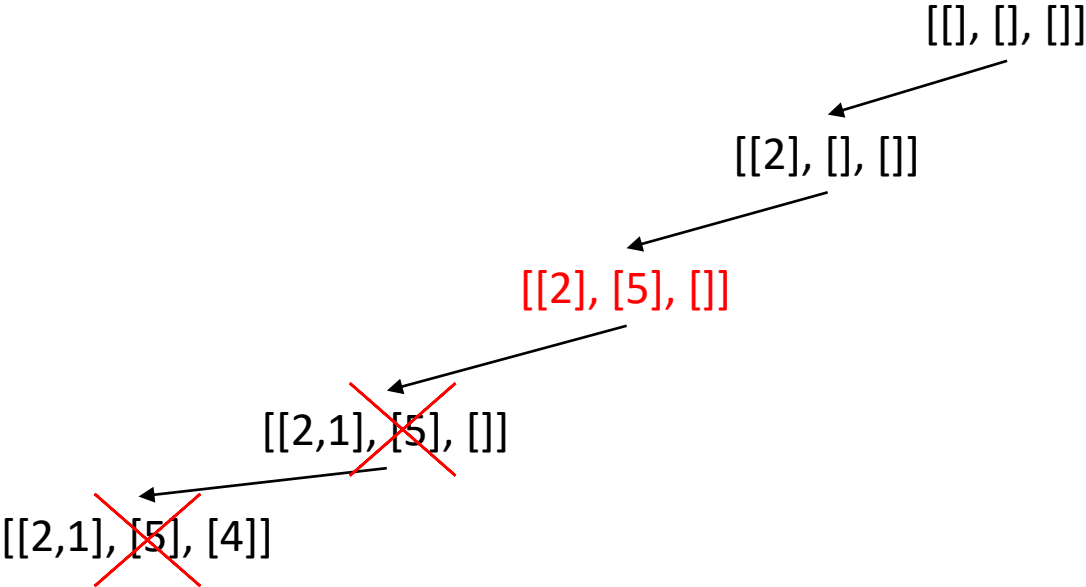
```
puntajes = [6]
```

```
target = 6
puntajes = [6, 4, 1]
```



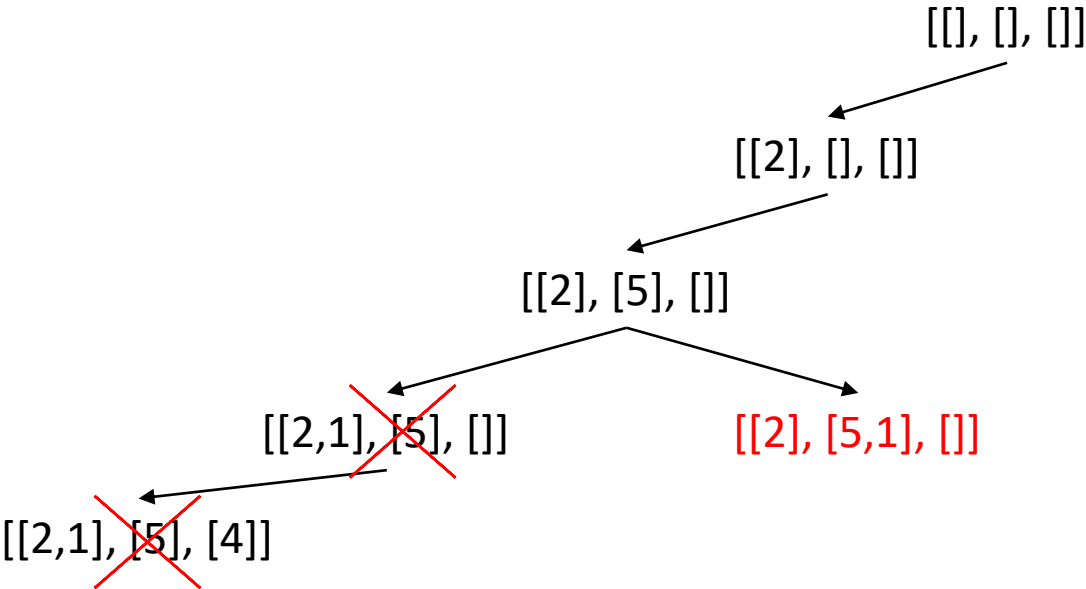
```
puntajes = [6, 4]
```

```
target = 6
puntajes = [6,4,1,5]
```



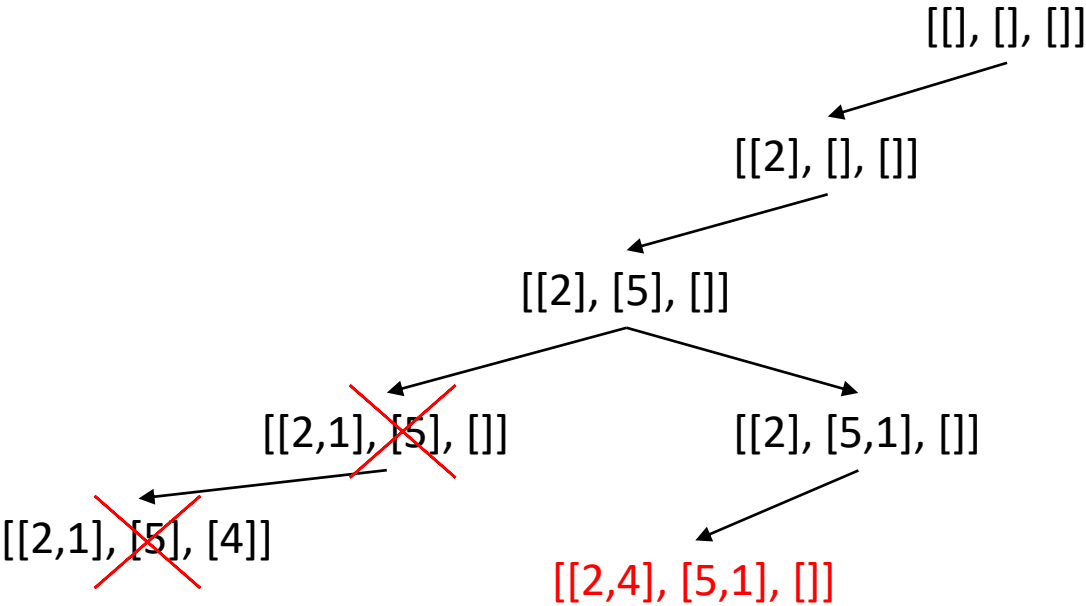
```
puntajes = [6,4,1]
```

```
target = 6
puntajes = [6,4,1]
```



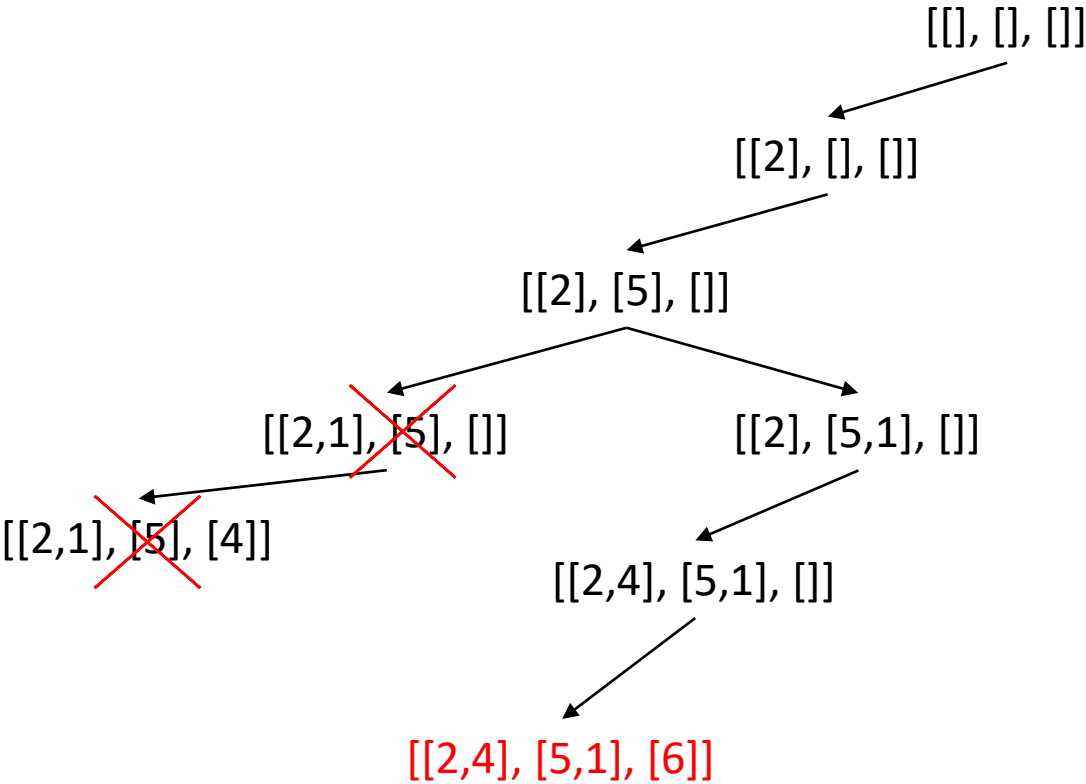
```
puntajes = [6,4]
```

```
target = 6
puntajes = [6, 4]
```



```
puntajes = [6]
```

```
target = 6
puntajes = [6]
```



```
puntajes = []
```

Tomamos un elemento de `puntajes` e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, debemos volver un paso atrás. **Si no quedan elementos por colocar, tenemos una configuración válida.**

```
def backtrack(puntajes, grupos, target):  
    if len(puntajes) == 0:  
        return True  
    ...
```



Tomamos un elemento de `puntajes` e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, debemos volver un paso atrás. Si no quedan elementos por colocar, tenemos una configuración válida.

```
def backtrack(puntajes, grupos, target):  
    if len(puntajes) == 0:  
        return True  
    puntaje = puntajes.pop()  
    ...
```

Tomamos un elemento de puntajes e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, debemos volver un paso atrás. Si no quedan elementos por colocar, tenemos una configuración válida.

```
def backtrack(puntajes, grupos, target):  
    if len(puntajes) == 0:  
        return True  
    puntaje = puntajes.pop()  
    for grupo in grupos:  
        ...  
    ...
```

Tomamos un elemento de puntajes e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, debemos volver un paso atrás. Si no quedan elementos por colocar, tenemos una configuración válida.

```
def backtrack(puntajes, grupos, target):  
    if len(puntajes) == 0:  
        return True  
    puntaje = puntajes.pop()  
    for grupo in grupos:  
        if sum(grupo) + puntaje <= target:  
            ...  
    ...
```

Tomamos un elemento de `puntajes` e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, debemos volver un paso atrás. Si no quedan elementos por colocar, tenemos una configuración válida.

```
def backtrack(puntajes, grupos, target):
    if len(puntajes) == 0:
        return True
    puntaje = puntajes.pop()
    for grupo in grupos:
        if sum(grupo) + puntaje <= target:
            grupo.append(puntaje)
            if backtrack(puntajes, grupos, target):
                return True
    ...
...
```

Tomamos un elemento de `puntajes` e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... **Si el elemento no se puede colocar en ningún grupo, no es una configuración válida**, debemos volver un paso atrás. Si no quedan elementos por colocar, tenemos una configuración válida.

```
def backtrack(puntajes, grupos, target):
    if len(puntajes) == 0:
        return True
    puntaje = puntajes.pop()
    for grupo in grupos:
        if sum(grupo) + puntaje <= target:
            grupo.append(puntaje)
            if backtrack(puntajes, grupos, target):
                return True
    ...
    return False
```

Tomamos un elemento de `puntajes` e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, **debemos volver un paso atrás**. Si no quedan elementos por colocar, tenemos una configuración válida.

```
def backtrack(puntajes, grupos, target):
    if len(puntajes) == 0:
        return True
    puntaje = puntajes.pop()
    for grupo in grupos:
        if sum(grupo) + puntaje <= target:
            grupo.append(puntaje)
            if backtrack(puntajes, grupos, target):
                return True
            grupo.pop()
    puntajes.append(puntaje)
    return False
```

Tomamos un elemento de `puntajes` e iteramos sobre los grupos. Verificamos si podemos colocarlo en el primer grupo. Si es así, lo colocamos y pasamos a trabajar con esta nueva configuración y con los puntajes restantes. Si no, verificamos si podemos colocarlo en el segundo grupo... Si el elemento no se puede colocar en ningún grupo, no es una configuración válida, debemos volver un paso atrás. Si no quedan elementos por colocar, tenemos una configuración válida.

```
def backtrack(puntajes, grupos, target):
    if len(puntajes) == 0:
        return True
    puntaje = puntajes.pop()
    for grupo in grupos:
        if sum(grupo) + puntaje <= target:
            grupo.append(puntaje)
            if backtrack(puntajes, grupos, target):
                return True
            grupo.pop()
    puntajes.append(puntaje)
    return False
```

```
def es_estado_valido(estado):
    # revisa si el estado es valido

def es_estado_solucion(estado):
    # revisa si es posible entregar una solución de manera trivial

def actualizar_estado(estado, movida):
    # actualiza el estado del mundo en base a la movida

def deshacer_ultima_movida(estado, movida):
    # deshace la última movida y vuelve al estado previo

def resolver(estado):
    if not es_estado_valido(estado):
        return False
    if es_estado_solucion(estado):
        return True
    else:
        for movida in movidas:
            actualizar_estado(estado, movida)
            if resolver(estado):
                return True
            else:
                deshacer_ultima_movida(estado, movida) # backtracking
        return False
```

Dos detalles:

1. ¿Es necesario sumar el puntaje de cada grupo cada vez que iteremos?
2. Nos piden formar los grupos con los índices...

```
def backtrack(puntajes, grupos, target):  
    if len(puntajes) == 0:  
        return True  
    puntaje = puntajes.pop()  
    for grupo in grupos:  
        if sum(grupo) + puntaje <= target:  
            grupo.append(puntaje)  
            if backtrack(puntajes, grupos, target):  
                return True  
            grupo.pop()  
    puntajes.append(puntaje)  
    return False
```

Ineficiente. Mejor llevar un contador para la suma de cada grupo.



```
def grupos_estudio(puntajes, K):
    suma_puntajes = sum(puntajes)
    if suma_puntajes % K != 0:
        return [[] for i in range(K)]
    target = suma_puntajes // K
    grupos = [[] for i in range(K)]
    sumas_grupos = [0]*K
    backtracking(puntajes, sumas_grupos, grupos, target)
    return grupos

def backtracking(puntajes, sumas_grupos, grupos, target):
    if len(puntajes) == 0:
        return True
    puntaje = puntajes.pop()
    for i, suma_actual in enumerate(sumas_grupos):
        if suma_actual + puntaje <= target:
            sumas_grupos[i] += puntaje
            grupos[i].append(puntaje)
            if backtracking(puntajes, sumas_grupos, grupos, target):
                return True
            sumas_grupos[i] -= puntaje
            grupos[i].pop()
    puntajes.append(puntaje)
    return False
```

Dos detalles:

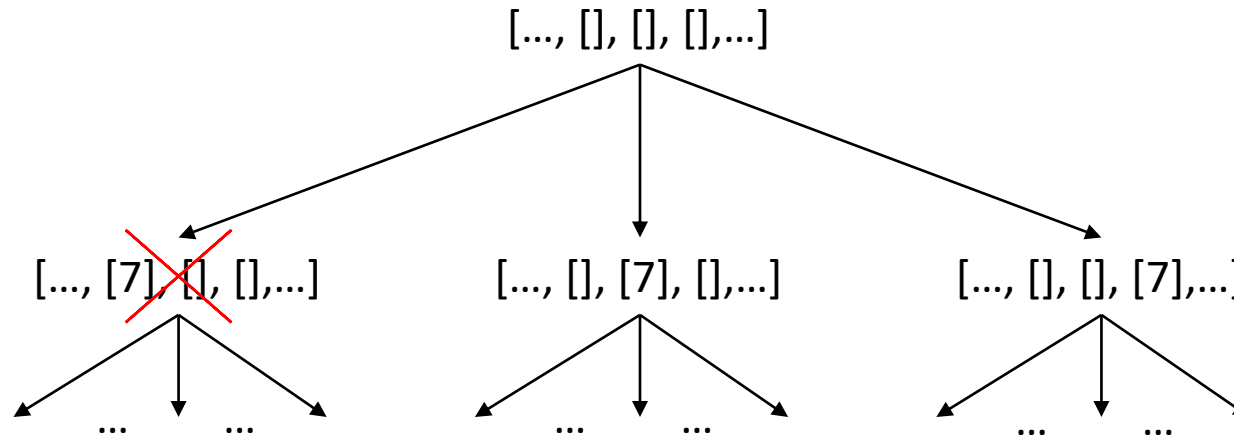
1. ¿Es necesario sumar el puntaje de cada grupo cada vez que iteremos?
2. Nos piden formar los grupos con los índices...

```
def backtrack(puntajes, sumas_grupos, grupos, target):  
    if len(puntajes) == 0:  
        return True  
    puntaje = puntajes.pop()  
    for i, suma_actual in enumerate(sumas_grupos):  
        if suma_actual + puntaje <= target:  
            sumas_grupos[i] += puntaje  
            grupos[i].append(puntaje)  
            if backtrack(puntajes, sumas_grupos, grupos, target):  
                return True  
            sumas_grupos[i] -= puntaje  
            grupos[i].pop()  
    puntajes.append(puntaje)  
    return False
```

Siempre sacamos el último elemento de `puntajes`, por lo que su índice es `len(puntajes) - 1`

```
def backtracking(puntajes, sumas_grupos, grupos, target):  
    if len(puntajes) == 0:  
        return True  
    pos = len(puntajes)-1  
    puntaje = puntajes.pop()  
    for i, suma_actual in enumerate(sumas_grupos):  
        if suma_actual + puntaje <= target:  
            sumas_grupos[i] += puntaje  
            grupos[i].append(pos)  
            if backtracking(puntajes, sumas_grupos, grupos, target):  
                return True  
            sumas_grupos[i] -= puntaje  
            grupos[i].pop()  
    puntajes.append(puntaje)  
    return False
```

## Mejora importante




Si tras agregar un puntaje llegamos a una configuración inválida y al hacer *backtracking* el grupo queda vacío, no es necesario volver a agregar ese puntaje en los grupos que siguen. Por ejemplo, si se llega a la configuración [..., [1,4], [5], [3], ...] y es inválida, es innecesario formar la configuración [..., [1,4], [3], [5], ...] (que pertenece a otra rama de búsqueda).

```
def backtrack(puntajes, sumas_grupos, grupos, target):
    if len(puntajes) == 0:
        return True
    pos = len(puntajes)-1
    puntaje = puntajes.pop()
    for i, suma_actual in enumerate(sumas_grupos):
        if suma_actual + puntaje <= target:
            sumas_grupos[i] += puntaje
            grupos[i].append(pos)
            if backtrack(puntajes, sumas_grupos, grupos, target):
                return True
            sumas_grupos[i] -= puntaje
            grupos[i].pop()
        if not suma_actual:
            break
    puntajes.append(puntaje)
    return False
```

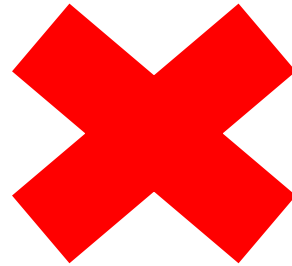
**Otras mejoras:** si ordenamos los números de menor a mayor y el último elemento es mayor que `target`, no se pueden formar los grupos. Si no ocurre lo anterior, podemos preguntarnos si hay elementos iguales a `target` (podemos revisar los números ordenados de derecha a izquierda), en cuyo caso podemos formar pre grupos que contengan solo estos elementos cada uno y así reducir la cantidad de números y grupos a formar.

```
puntajes = [1, 4, 6, 2, 3, 4]
K = 4
```




```
target = 5
```

```
puntajes_ordenados = [1, 2, 3, 4, 4, 6]
```



**Otras mejoras:** si ordenamos los números de menor a mayor y el último elemento es mayor que `target`, no se pueden formar los grupos. Si no ocurre lo anterior, podemos preguntarnos si hay elementos iguales a `target` (podemos revisar los números ordenados de derecha a izquierda), en cuyo caso podemos formar pre grupos que contengan solo estos elementos cada uno y así reducir la cantidad de números y grupos a formar.

```
puntajes = [1, 5, 5, 2, 3, 4]
K = 4
```



```
target = 5
```

```
puntajes_ordenados = [1, 2, 3, 4, 5, 5]
pre_grupos = [[5], [5]]
```

Ahora usamos backtracking para formar el resto de grupos, pero con los siguientes parámetros:

```
puntajes = [1, 2, 3, 4]
K = 2
```

Si se pueden formar los grupos, juntar grupos con `pre_grupos` y retornar.

**Otras mejoras:** si ordenamos los números de menor a mayor y el último elemento es mayor que `target`, no se pueden formar los grupos. Si no ocurre lo anterior, podemos preguntarnos si hay elementos iguales a `target` (podemos revisar los números ordenados de derecha a izquierda), en cuyo caso podemos formar pre grupos que contengan solo estos elementos cada uno y así reducir la cantidad de números y grupos a formar.

**Consideración:** al ordenar la lista, se pierden los índices. ¿Cómo podemos abordar esto? (Investigar método `numpy.argsort` de la librería `numpy`)