Ejercicio extra D&C:

Se tiene un arreglo ordenado sin repetición de *n* números naturales consecutivos con *k* huecos, es decir, en el arreglo están presentes todos los elementos dentro de un rango determinado salvo una cantidad *k* de ellos. Por ejemplo, el arreglo A= [11, 12, 13, 15, 16, 19, 20, 21] tiene tres huecos, [14, 17, 18]. Se pide encontrar los valores de todos los faltantes de un arreglo.

a) Dar un algoritmo que utilice la técnica de Divide and Conquer y resuelva el problema en tiempo O(k log n) en el peor caso.
b) Marcar claramente qué partes del algoritmo se corresponden a dividir, conquistar y unir subproblemas.

Resolución:

Algo que podemos pensar primero es como determinar si hay huecos o no. Eso es fácil si comparamos la diferencia entre los valores extremos del arreglo y la longitud de este de la siguiente manera:

```
def huecos?(a):
    k=len(a)
    return k != a[k-1]-a[0] + 1                    #O(1)
```

Notar que esta función nos permite en tiempo constante determinar si hay buenos en cualquier sub-arreglo.

La cuestión ahora es como hacer el dividir y conquistar. Una idea es dividir el arreglo por la mitad y en cada una de ellas determinar si hay huecos. Una vez que determinamos que intervalos tienen huecos

```
def buscarhuecos(a):
    intervalos = obtenerintervalos(a)       #O(k * log n)
    return formarhuecos(intervalos)         #O(k)
```

Lo importante es reconocer cuales son los casos base. Las instancias de un subproblema que se puede resolver directamente.

La forma de justificar la complejidad es considerar el árbol de operaciones con cada llamada recursiva y ver que a lo sumo hay *log(n)* niveles con *k* operaciones como máximo en cada nivel.

obtenerIntervalos(**in** a: arreglo(nat)) $\rightarrow$ *res*:*lista*($< nat, nat >$)

```
1:  res ← vacia()                                                          ▷ O(1)
2:  res_izq ← vacia()                                                      ▷ O(1)
3:  res_med ← vacia()                                                      ▷ O(1)
4:  res_der ← vacia()                                                      ▷ O(1)
5:  izq ← a[0...tam(a)/2 − 1]                                              ▷ O(1)
6:  med ← a[tam(a)/2 − 1...tam(a)/2]                                       ▷ O(1)
7:  der ← a[tam(a)/2...tam(a) − 1]                                         ▷ O(1)
8:  if tam(a) == 2 then                                  ▷ Conquer: Caso Base 1
9:      agregar(res, < a[0], a[1] >)                     ▷ Aqui ya se que hay hueco
10:     return res
11: else
12:     if tam(a) == 3 then   ▷ Conquer: Caso Base 2, para arreglos de longitud impar
13:         if hayHuecos?([ a[0], a[1] ]) then
14:             agregar(res, < a[0], a[1] >)
15:         end if
16:         if hayHuecos?([ a[1], a[2] ]) then
17:             agregar(res, < a[1], a[2] >)
18:         end if
19:         return res
20:     end if
21: end if
22: if hayHuecos?(izq) then   ▷ Divide: Caso Recursivo: chequeo mitades y centro en busca de huecos
23:     res_izq ← obtenerIntervalos(izq)
24: end if
25: if hayHuecos?(der) then
26:     res_der ← obtenerIntervalos(der)
27: end if
28: if hayHuecos?(med) then
29:     res_med ← obtenerIntervalos(med)
30: end if
31: res ← res_izq + +res_med + +res_der
32: return res
```

formarHuecos(**in** l : $lista(< nat, nat >)$) $\rightarrow$ *res*:*lista*(*nat*)

```
1:  res ← vacia()                                                          ▷ O(1)
2:  it ← crearIt(l)                                                        ▷ O(1)
3:  while haySiguiente(it) do
4:      izq ← π1(siguiente(it))
5:      der ← π2(siguiente(it))
6:      j ← izq + 1
7:      while j ! = der do
8:          agregar(j, res)
9:          j + +
10:     end while
11: end while
12: return res
```