NOTA:

El ejercicio 3 tiene errores que es importante entenderlos bien aplicando el Teorema Maestro.
Llegó a la complejidad pero medio de casualidad.
Lo dejo subido por que considero que la correcion esta bueno entenderla.

Nombre y Apellido	LU	NO

• Se entregó enunciado + 4 hojas resolución

Corrector:	Elías		
Nota Final / Ejs:	1	2	3
A	A	A	R

¡Suerte en la carrera! :)

Algoritmos y Estructuras de Datos II
Segundo parcial — 10 de Junio de 2022

Aclaraciones

- El parcial es a libro casi-cerrado. Solo es posible tener impreso el teorema maestro y el apunte de diseño. Además, pueden tener una hoja (2 carillas), escrita a mano, con los apuntes que se deseen.
- Cada ejercicio debe entregarse **en hojas separadas**. Las mismas deben estar numeradas.
- Incluir en esta hoja: nombre, apellido, el número de orden asignado, número de libreta.
- Cada ejercicio se calificará con Perfecto, Aprobado, Regular, o Insuficiente.
- El parcial estará aprobado si las notas de los tres ejercicios tienen al menos dos **A**.
- Los ejercicios **no** se recuperan por separado.
- Se encuentran disponibles para utilizar todas las estructuras de datos presentadas en la teórica con las operaciones y complejidades dadas en las mismas.

Ej. 1.

Proponer un algoritmo que permita ordenar un arreglo de naturales (sobre los que no se conoce nada en especial) de manera ascendente en $O(n \log d)$ en el peor caso, donde donde n es la cantidad de elementos a ordenar y d es la cantidad de elementos distintos. Justifique informalmente por qué su algoritmo cumple con lo pedido.

Ej. 2.

Se tiene una secuencia de alturas $h_1, \ldots, h_n$. Decimos que un intervalo $h_t, h_{t+1}, \ldots, h_{t+k}$ es un *edificio* si para todo $i \in [t, t+k)$, el valor $|h_i - h_{i+1}|$ no es mayor que una cierta tolerancia dada θ y además tanto $|h_t - h_{t-1}|$ como $|h_{t+k} - h_{t+k+1}|$ (en caso de existir) son mayores que θ. El *ancho* de un edificio es la cantidad de alturas que lo componen. Por ejemplo, si la secuencia de alturas es

$$[100, 101, 100, 103, 80, 77, 74, 200, 32, 30]$$

y $\theta = 3$, entonces los edificios de esta secuencia de alturas serían $[100, 101, 100, 103]$, $[80, 77, 74]$, $[200]$, $[32, 30]$, y sus anchos serían 4, 3, 1 y 2, respectivamente.

Escribir un algoritmo que tome un arreglo de enteros (que representan alturas) y una tolerancia θ y devuelva las mismas ordenadas. El orden estará dado según los anchos de los edificios en forma creciente. Tanto los edificios como las alturas dentro de cada edificio deben mantener el orden original. En el ejemplo anterior, el resultado esperado sería $[200, 32, 30, 80, 77, 74, 100, 101, 100, 103]$. La complejidad del algoritmo no debe ser peor que $O(n)$, donde n es la cantidad de alturas dada. Justifique informalmente la correctitud del algoritmo y su complejidad temporal.

Ej. 3.

Se tiene un arreglo ordenado 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 *huecos* en los valores $[14, 17, 18]$. Se pide describir un algoritmo que devuelva una lista con todos los *huecos* de un arreglo. Se puede asumir que el arreglo tiene tamaño potencia de 2.

a) Dar un algoritmo que use 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.
c) Asumiendo que $k = 1$, justificar formalmente que el algoritmo cumple con la complejidad pedida.
d) Justificar informalmente, para el caso general $k > 1$, que la complejidad es $O(k \log(n))$.

EJERCICIO 1

Idea del Algoritmo:

- Se recorre el total de elementos. O(n) /
- Se buscan y/o insertan en un dicc_AVL. O(log(d)) /
- Las claves del dicc_AVL serán los números naturales / y el valor ~~otro~~ natural que indica cuántas veces aparece. /
- Las claves pueden ordenarse con el algoritmo Merge-Sort en tiempo d. log(d) con $d \leq n$ /
- Finalmente se escriben los elementos ordenados. /

Entonces todos los elementos se guardan en un dicc_AVL como clave, teniendo de valor las veces que se encontró. /
Luego se pide el conjunto de claves y se ordena con merge sort para / luego, en orden, escribir cada valor tantas veces como indique el diccionario. /

```
ordenar_naturales(in naturales: arreglo) → res: arreglo {
    dicc_AVL = CrearDicAVL() ✓
    it_naturales = CrearIt(naturales) ✓
    while (HayProximo(it_naturales)) { //O(n) ✓
        nat_aux = dameProximo(it_naturales) //O(1) ✓
        nat_valor_actual = 1                //O(1) ✓
        if (def?(nat_aux, dicc_AVL)) { // log(d) ✓  d cantidad de posibles elementos
            nat_valor_actual = dar(dicc_AVL, nat_aux) // log(d)
                                              ↳ ¿con qué valor?
            nat_valor_actual ++;
        }
        def(dicc_AVL, nat_aux, valor_actual) //log(d) ✓
    } avanzar(it_naturales) //O(1) ✓

    claves_en_orden = Merge_Sort(darClaves(dicc_AVL)) // d.log(d) ✓
    It_claves_en_orden = CrearIt(claves_en_orden) ✓      //d ≤ n
    res = CrearArreglo() ✓
                                                      // d=n peor caso donde
    while (hayProximo(It_claves_en_orden)) { //O(d) ✓  // Todos los numeros son
        valor = dameProximo(It_claves_en_orden) //O(1) ✓  // distintos
        cantidad = dar(dicc_AVL, valor) //O(log(d)) ✓
        for (i=0; i++; i<cantidad) { //cantidad ≤ d ✓
            res.agregarAtras(valor) ✓
        }
    } avanzar(it_claves_en_orden) //O(1) ✓
    return res ✓
}
```

d ≤ n

Complejidad: $O(n).O(\log(d)) + O(d.\log(d)) + O(d).O(\log(d)) = O(n \log(d))$ ✓

EJERCICIO 2

- La idea del algoritmo/solución se basa en recorrer las alturas identificando los edificios. ✓
- Para identificar los edificios se controla no exceder la tolerancia de altura θ dada. ✓
- Luego, en un nuevo vector se crean elementos tipo tuplas de la forma:
  [([alturas edificio], ANCHO), ([alturas_edificio], ANCHO), ...] ✓
- Finalmente se ordenan por ANCHO como indica el ejercicio ✓ y se escribe el resultado de forma consecutiva. ✓
  Para este paso se puede considerar el problema como ordenar un conjunto de números naturales (los anchos) sabiendo que tengo k elementos con k < N. ✓

① Como sabés que las tuplas tienen ancho acotado por n, podés hacer un bucketSort con un arreglo de listas con n posiciones donde insertás en la posición i-ésima la tupla de edificios con ancho i ó i-1.

```
ordenar_edificios(in arreglo_alturas: arreglo, in Tolerancia: entero) → res: arreglo {
    contador_aux = 0
    edificios_aux = nuevo_arreglo()
    edificios = nuevo_arreglo()
    for (i=0; i++; i < long(arreglo_alturas)) { //recorro las alturas
        contador_aux ++;
        edificios_aux.agregar(arreglo_alturas[i])
        if (arreglo_alturas(i+1) - arreglo_alturas[i] > Tolerancia) { //cambio edificio
            Tupla aux = crear Tupla(edificios_aux, contador_aux);
            contador_aux = 0;
            edificios_aux = nuevo_Arreglo();
            edificios.agregar(Tupla_aux);
        }
    } //Hasta acá tengo el arreglo en edificios y anchos
```

Acá deberías agregar el último edificio, porque iterás n-1 veces arriba.

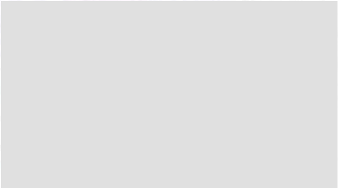
```
    edificios = orden_por_ancho (edificios); //ordeno por ancho sabiendo
                                             //que el ancho es menor a n
        (no llegue a implementar)            //Complejidad menor O(n)
    res = nuevo_arreglo()
    //escribo resultado contiguo y retorno
    for (i=0; i++; i < edificios.long()) {
        for (j=0; j++; j < π(edificios[i])) { //recorro el vector de
                                              alturas del edificio
            res.agregarAltura(π_1(edificios[i])[j]);
        }
    }

    retorno res;
}
```

Complejidad final: O(n)

dado que se recorren en total 2 veces las alturas y el orden es menor a O(n).

(Tal vez algún índice esté equivocado. Terminé en los últimos minutos) ☺

| Ejercicio 3 |



• La idea de la resolución es dividir el problema en 2 llamadas recursivas (las mitades del arreglo), teniendo como caso base que la longitud de este arreglo sea 2. Teniendo en cuenta la suposición dada de len(arreglo) % 2 = 0

Luego, en el caso base se retornan todos los huecos encontrados entre esos dos numeros.
Por ejemplo, si los numeros fueran 5 y 8 se retorna un arreglo con los huecos entre ellos: [6,7].
Si en el caso base los numeros son consecutivos (osea, no hay huecos) se retorna un arreglo vacio.
Finalmente, y dado que se cuenta con el orden inicial, se pregunta si existe hueco entre las mitades. De existir hueco las mitades se unen y se rellena con los numeros restantes. De no existir solo se unen las mitades. Las funciones para esto son:

```
unir_mitades (in arreglo_1 , in arreglo_2) → res : arreglo {
    res = crearArreglo ()
    for (i = 0; i++; i < len(arreglo_1))
        res.agregar (arreglo_1 [i])
    }
    For (i = 0 ; i++ ; i < len(arreglo_2)) {
        res.agregar (arreglo_2 [i])
    }
    return res ;
}
```

// Donde la complejidad es lineal a la longitud de los arreglos que se
// condice con los huecos encontrados k (item d)

```
unir_mitades_y_completar(in arreglo_1 ; in arreglo_2) → res: arreglo {
    res = crear_arreglo()

    for (i=0 ; i++ , i < len(arreglo_1)) {
        res.agregar(arreglo_1[i])
    }

    for ((i=0) ; i++ ; i < len(arreglo_2)) {
        res.agregar(arreglo_2[i])
    }
    ----------------------------------
    for (i = len(arreglo_1) ; i++ ; i < arreglo_2[0]) {
        res.agregar(i) ;
    }
    ----------------------------------
    return res
} // La complejidad meramente es lineal en relación a la cantidad de huecos k (item d)
```

el orden no es importante, pero se puede invertir primero para tener los huecos ordenados.

¿no? Estos serían los huecos que se encuentren en el medio? [1, 2, 5, 6] ↓ acá.

De ser así, deberías hacer...
for i = arreglo_1[ult.]+1, i++, i < arreglo_2[prim]
    res.agregar(i).

Finalmente, usando las funciones auxiliares, el algoritmo D&C es:

(a)

```
res = crear_arreglo(); ✓
dar_huecos(in arreglo_ordenado: arreglo ; int inicio, int fin) → res: arreglo {
    if (len(arreglo_ordenado) = 2) { // caso base. Además supongo len potencia de 2. ✓
        if (arreglo_ordenado[inicio] + 1 == arreglo_ordenado[inicio+1]) { // no hay huecos
            return crearArreglo(); ✓
        else {
            for (i = arreglo_ordenado[inicio] + 1 ; i++ ; i < arreglo_ordenado[i]) {
                res.agregarAtrás(i); ✓
            }
        return res; ✓
    }
```

Caso base. Conquistar (b). ✓

Donde el problema se simplificó. Conquista. (b)

(Sigue función próxima hoja)

EJERCICIO 3

③ → Explico atrás

Llamadas recursivas. DIVIDIR (b)

```
mitad_1 = dosHuecos(arreglo_ordenado, inicio, len(arreglo_ordenado)/2)
mitad_2 = dosHuecos(arreglo_ordenado, len(arreglo_ordenado)/2+1, fin)

If (Mitad_1[len(Mitad_1-1)]+1 == Mitad_2[0]) { //no hay huecos entre
                                                //las mitades
    res = [UnirMitades(Mitad_1, Mitad_2)]
} else {        (b) Union de los problemas.
    res = [Unir-Mitades-y-completar(Mitad_1, Mitad_2)]
}

return res
}
```

(c) $k=1$. La complejidad se considera $T(n)$. Luego, aplicando el Teorema Maestro: $T(n) = A \cdot T\left(\frac{n}{B}\right) + O(n^c)$

A: llamadas recursivas subproblemas

B = División/Proporción del tamaño original.

$O(n^c)$: Complejidad de Dividir y ~~Juntar~~.

Para $k=1$ la complejidad de $O(n^c)$ ~~[tachado]~~ es constante. Luego:

NO

$T(n) = O(n^c \log_b(n)) = O(\log(n))$

↑ Por el teorema del maestro cuando $\log_b(A) = c$.

↳ $O(n^k)$ con k constante no es similar $O(1)$. Sino que pasaría con $O(n^2)$?

→ Atrás

(d) Como se aclaró en los huecos auxiliares la complejidad de los costos de unir el resultado del algoritmo es lineal a la cantidad de huecos por eso cumple: $O(k \cdot \log(n))$

③ Antes de llamar recursivamente a la función deberías verificar en $O(1)$ si hay huecos en esa mitad. Si no lo hay, NO hago el llamado recursivo. De esa manera, cuando le=1 será como una búsqueda binaria.

Pensar qué pasa con

[1, 2, 3, 4, 5, 6, 7, 9].

así voy a tener

Teorema maestro. $\begin{cases} a=1 \\ b=2 \\ f(n) \in O(1) \end{cases}$

$\Rightarrow T(n) = T(n/2) + O(1)$

cumple en el caso 2 pues

$\log_2 1 = 0 = c$ y $f(n) \in \Theta(n^0 . \log^0(n)) = \Theta(1)$.

$\Rightarrow T(n) \in \Theta(\log(n))$.

Para poder cumplir la coplejidad obligatoriamente era necesario resolver un solo problema luego de aplicar la recursividad.
Aparentemente la unica forma de lograrlo era poder identificar en los vectores si tenian o no huevos previamente en O(1)

¿como?
A = [5,6,7,8,9,10]

Entoces,

A[len(A) - 1] - A[0] = len(A) -1

A[5] - A[0] = 6 - 1

10 - 5 = 5

Esto solo se cumple si no hay huecos dado que sabemos que los numeros son consecutivos.

¿Como se supoje que nos demos cuenta de esto en el examen?

No se