**Taller 3 – FADA**

**Julián Andrés Ibáñez Quiñones – 1866221**

**1. (35%) Análisis algoritmo Divide y vencerás.**

```
Stooge-Sort(A,i,j)
1  if A[i] > A[j]
2  then exchange A[i] ↔ A[j]
3  if i + 1 ≥ j
4  then return
5  k ← ⌊(j − i + 1)/3⌋ ▷ Round down.
6  Stooge-Sort(A,i,j − k) ▷ First two-thirds.
7  Stooge-Sort(A,i + k,j) ▷ Last two-thirds.
8  Stooge-Sort(A,i,j − k) ▷ First two-thirds again.
```

**a) (5 %) Explique la estrategia: Dividir, conquistar y combinar.**

Este algoritmo va recibir una lista, ya sea ordenada o no, y la retornará con orden ascendente, en este caso porque está configurado con signo mayor o igual (**≥**). La forma como la estrategia dividir, conquistar y combinar funciona para este algoritmo es la siguiente:

**1. Linea 1 y 2:** El algoritmo inicia comparando el elemento del indice 0 con el elemento del último indice, o mejor dicho, el elemento que está en la posición del tamaño de la lista menos 1, si el primero es mayor que el último, entonces los intercambiará de posición. Esto lo hace de manera recursiva, ya que una lista se va dividir en sublistas y en cada una de estas de aplicará este método.

Para el ejemplo enumeraremos las posiciones desde 1.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| **6** | **5** | **7** | **2** | **3** | **4** | **9** | **8** | **1** |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| **1** | **5** | **7** | **2** | **3** | **4** | **9** | **8** | **6** |

**2. Linea 3:** Verifica si la sublista tiene dos o menos elementos. Si es así, la sublista ya está ordenada y la función retorna sin hacer más operaciones.

**3. DIVIDIR:** La lista se divide entre 3 redondeando a piso.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| **1** | **5** | **7** | **2** | **3** | **4** | **9** | **8** | **6** |

Primero se opera la sublista compuesta por los dos primeros tercios, o sea, desde el indice 1 hasta el 6.

Al sacar una sublista siempre compara el primer y ultimo elemento y los cambia si es necesario.

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| **1** | **5** | **7** | **2** | **3** | **4** |

Esta lista se divide entre 3

Se obtiene esta sublista de los dos primeros tercios de la anterior.

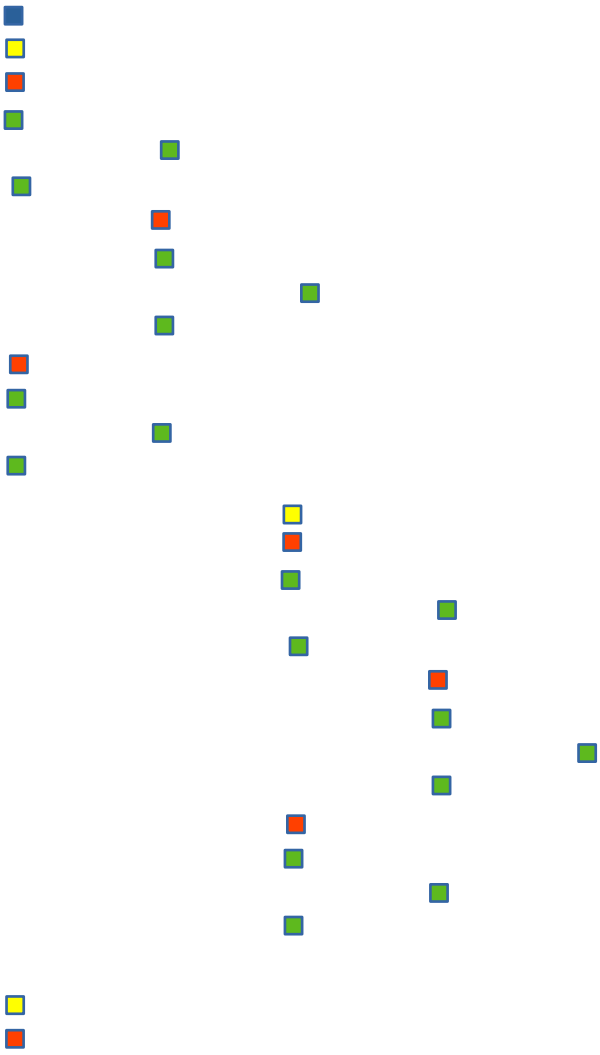
Se divide la lista amarilla entre 3 y se toma la dos primeras partes.

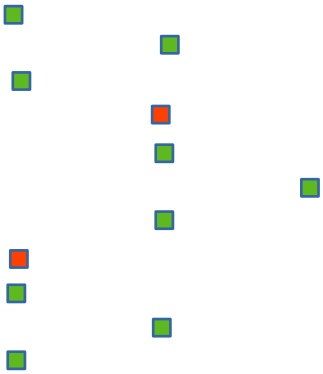
Como lo mínimo que se puede comparar son 2 elementos, esta es la sublista más pequeña.

Básicamente el algoritmo en cada sublista que se pueda dividir (esto es controlado por la linea 3 y 4 del pseudocódigo) verifica los primeros ⅔, luego los últimos ⅔ y nuevamente los primeros ⅔ (Lineas 6, 7 y 8).

A continuación ejemplo grafico explicando esto y en el orden que el algoritmo lo hace, suponiendo que va cambiando el primer y último elemento de cada sublista una vez se obtiene esta.

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | | | | | |





Luego de todo esto, la lista ya tiene que estar ordenada.

b) (10 %) Implemente el algoritmo.

```
def stooge_sort(arr, i=0, j=None):
    if j is None:
        j = len(arr) - 1

    if arr[i] >= arr[j]:
        arr[i], arr[j] = arr[j], arr[i]

    if i + 1 >= j:
        return

    k = (j - i + 1) // 3

    stooge_sort(arr, i, j - k)
    stooge_sort(arr, i + k, j)
    stooge_sort(arr, i, j – k)

#Ejemplo
arr = [6, 5, 7, 2, 3, 4,]
stooge_sort(arr)
print("Lista ordenada", arr)
```

c) (5 %) Realice pruebas con arreglos aleatorios de tamaño 10, 100, 1000, 10000.

```
import random
import time

sizes = [10, 100, 1000, 10000]

for size in sizes:
    arr = [random.randint(1, 1000) for _ in range(size)]

    start_time = time.time()
    stooge_sort(arr)
    end_time = time.time()
```

```
    print(f"Size: {size}, Time: {end_time - start_time:.6f} seconds")
```

resultado de ejecución (para 10000 toca esperar demasiado)

```
Size: 10, Time: 0.000000 seconds
Size: 100, Time: 0.094344 seconds
Size: 1000, Time: 24.118062 seconds
```

d) (15 %) Calcule la complejidad teórica del algoritmo y compárela con la complejidad práctica encontrada. Analice lo que encuentra.

La complejidad que tiene este algoritmo es de O^(nlog3/23) o más o menos  O^(n2.7095). Donde n es el tamaño de la lista.