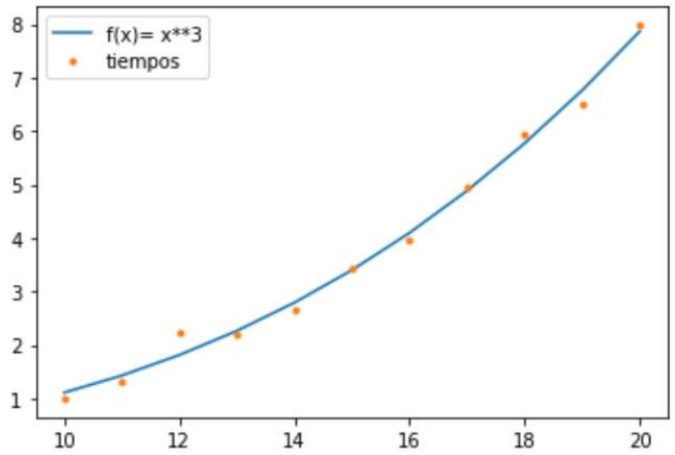
Hugo Torres Martínez
Luis Rodríguez Moreno
Grupo: 1272
Pareja: 04

CUESTIONES 1-C

1. ¿A qué función f se deberían ajustar los tiempos de multiplicación de la función de multiplicación de matrices?

Los tiempos de multiplicación de matrices se ajustan a f = n^3.



2. Calcular los tiempos de ejecución que se obtendrían usando la multiplicación de matrices a.dot(b) de Numpy y compararlos con los anteriores.

Aparte de estos dos métodos para multiplicar matrices, hemos desarrollado una función propia que ejecuta la misma operación. Adjuntamos el código de dicha función:

```
def matrix_multiplication(m1: np.ndarray, m2 : np.ndarray) :
    if len (m1[0]) == len (m2):
        m3 = [ ]
        for i in range (len (m1)):
            m3.append ([])
            for j in range (len (m2[0])):
                m3 [i].append(0)

        for i in range (len (m1)):
            for j in range (len (m2[0])):
                for k in range(len(m1[0])):
                    m3[i][j] += m1[i][j] * m2[k][j]
        return m3
    else:
        return None
```

Hemos calculado sus tiempos de ejecución y los hemos añadido también a la gráfica para compararlos con los otros dos métodos.

```
'TIEMPO DE EJECUCION FORMA MOODLE '

import numpy as np
l_timings2 = []
for i2 in range(11):
    dim2 = 10+i2
    m2 = np.random.uniform(0., 1., (dim2, dim2))
    timings2 = %timeit -o -n 1 -r 1 -q matrix_multiplication2(m2, m2)
    l_timings2.append([dim2, timings2.best])

print(l_timings2)
```

✓ 0.1s Python

```
[[10, 0.0013287100000098917], [11, 0.001455008999997176], [12, 0.001322494999996593],
[13, 0.0017739520000077391], [14, 0.0021767690000018547], [15, 0.004262194000006048],
[16, 0.0047139370000053304], [17, 0.005173350000006849], [18, 0.0054098680000009836],
[19, 0.005535119000001032], [20, 0.00724569500000823]]
```

```
'TIEMPO DE EJECUCION NUESTRA FORMA DE HACERLO'

import numpy as np
l_timings = []
for i in range(11):
    dim = 10+i
    m = np.random.uniform(0., 1., (dim, dim))
    timings = %timeit -o -n 10 -r 5 -q matrix_multiplication(m, m)
    l_timings.append([dim, timings.best])

print(l_timings)
```

✓ 2.2s Python

```
[[10, 0.0008706218000000376], [11, 0.0012361255000001847], [12, 0.0013233551000013221],
[13, 0.0016671486000007008], [14, 0.002111680200000876], [15, 0.0028051645999994435],
[16, 0.0038176681999999573], [17, 0.00434075059999941], [18, 0.004523046299999578],
[19, 0.005332734399999595], [20, 0.005824309899999491]]
```

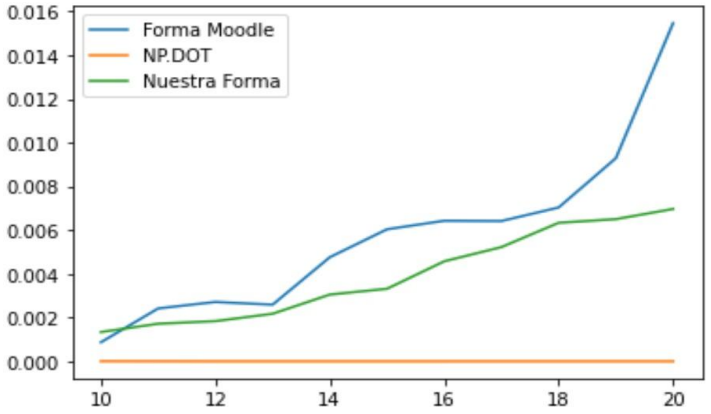
```
'TIEMPOS DE EJECUCION NP.DOT'
'Ejercicio 1b'

import numpy as np
l_timings_np = []
for i in range(11):
    dim_np = 10+i
    m_np = np.random.uniform(0., 1., (dim_np, dim_np))
    timings_np = %timeit -o -n 10 -r 5 -q np.dot(m_np, m_np)
    l_timings_np.append([dim_np, timings_np.best])


print(l_timings_np)
154
```

✓ 0.7s Python

```
[[10, 2.571199999579221e-06], [11, 3.5495000005880684e-06], [12, 3.636600000334056e-06],
[13, 2.09429999955546e-06], [14, 3.7143000000128268e-06], [15, 2.2547000000372464e-06],
[16, 3.875000000164164e-06], [17, 4.171799999141967e-06], [18, 4.241300000273895e-06],
[19, 2.79750000089507e-06], [20, 2.8486999994470352e-06]]
```

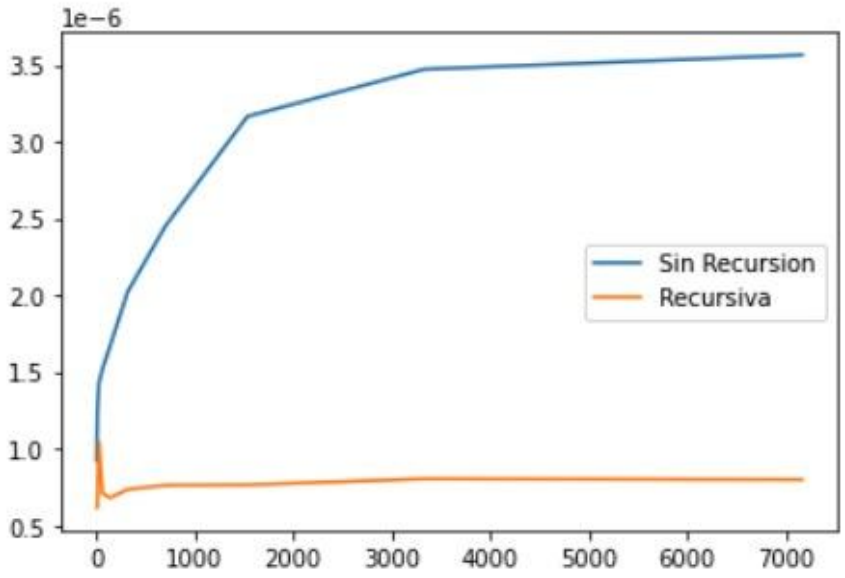
```
'Tiempo de ejecucion cuestion 2'
import matplotlib.pyplot as plt
import numpy as np
times_1 = np.asarray(l_timings_np)
times_2 = np.asarray(l_timings2)
times_3 = np.asarray(l_timings)

plt.plot(times_2[:,0], times_2[:,1],'-', label='Forma Moodle')
plt.plot(times_1[:,0] ,times_1[:,1],'-', label='NP.DOT')
plt.plot(times_3[:,0] ,times_3[:,1],'-', label='Nuestra Forma')
plt.legend()
plt.show()
```

✓ 0.4s



3. Comparar los tiempos de ejecución de las versiones recursiva e iterativa de la búsqueda binaria en su caso más costoso y dibujarlos para unos tamaños de tabla adecuados. ¿Se puede encontrar alguna relación entre ellos?

Ambas funciones tardan más o menos lo mismo con las primeras operaciones, pero conforme aumenta ese número la función recursiva consigue ahorrar mucho tiempo.



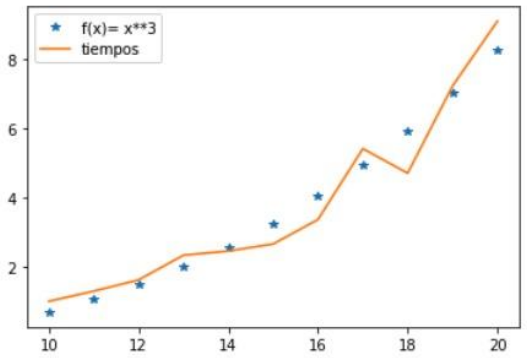
CUESTIONES 2-D

1. Analizar visualmente los tiempos de ejecución de nuestra función de creación de min heaps. ¿A qué función se deberían ajustar dichos tiempos?

Los tiempos de ejecución de creación de min heaps se ajustan a f = n^3.

```
'Tiempo de ejecucion 2A'

import matplotlib.pyplot as plt
timings3 = np.asarray(l_timings2)
ajuste = fit_func_2_times(timings3, func_2_fit)
plt.plot(timings3[ : ,0], ajuste , '*', label='f(x)= x**3')
plt.plot(timings3[:,0], timings3[:,1]/timings3[0,1], '-', label='tiempos')
plt.legend()
plt.show()
```

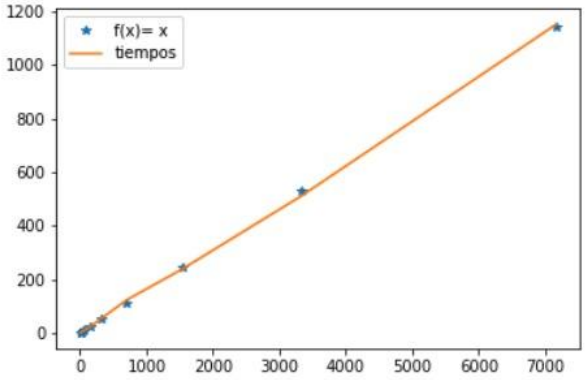
✓ 0.4s Python



2. Expresar en función de k y del tamaño del array cual debería ser el coste de nuestra función para el problema de selección.

```
import matplotlib.pyplot as plt

timings5 = np.asarray(l_times_ps)
ajuste_3 = fit_func_2_times_3(timings5, func_2_fit_3)

plt.plot(timings5[:,0], ajuste_3 , '*', label='f(x)= x')
plt.plot(timings5[:,0], timings5[:,1]/timings5[0,1], '-', label='tiempos')
plt.legend()
plt.show()
```

✓ 0.2s



3. Una ventaja de nuestra solución al problema de selección es que también nos da los primeros k elementos de una ordenación del array. Explicar por qué esto es así y cómo se obtendrían estos k elementos.

Al llamar a la función create_min_heap obtenemos la raíz del árbol, es decir, el elemento más pequeño. Al aplicar el remove de ese elemento, no solo lo extraemos y lo guardamos en un array auxiliar, sino que se aplica el min heap para el resto del array una vez extraído. De esta forma, si seguimos realizando el remove y seguimos guardando cada raíz en el array auxiliar k veces, obtendremos un array ordenado, ya que cada vez que extraemos, extraemos el elemento más pequeño.

4. La forma habitual de obtener los dos menores elementos de un array es mediante un doble for donde primero se encuentra el menor elemento y luego el menor de la tabla restante. ¿Se podrían obtener esos dos elementos con un único for sobre el array?¿Cómo?

Sí que se puede conseguir. Antes de recorrer el array, nos aseguramos que de los dos primeros elementos el más pequeño sea min1. Una vez hemos hecho estas dos asignaciones, comenzamos a recorrer el array con un bucle for. De esta forma, vamos comprobando de uno en uno los valores de min1 y min2, haciendo las comparaciones que se muestran en el código, con la ayuda de una variable auxiliar. Finalmente la función nos devuelve una tupla con los dos elementos más pequeños del array dado por argumento.

```
import numpy as np
def dos_menores_elementos(h:np.ndarray)->tuple([int, int]):

    if h[0]<h[1]:
        min1 = h[0]
        min2 = h[1]
    else:
        min1 = h[1]
        min2 = h[0]

    for i in range(2,len(h)):
        if h[i] < min1:
            aux = min1
            min1 = h[i]
            min2 = aux
        elif h[i] < min2:
            min2 = h[i]

    return (min1, min2)
```