CUESTIONES PRACTICA 1

Sergio Larriba Moreno y Sergio Homobono Chicharro
Pareja nº09 del grupo 1273

I-A. Midiendo tiempos con %timeit

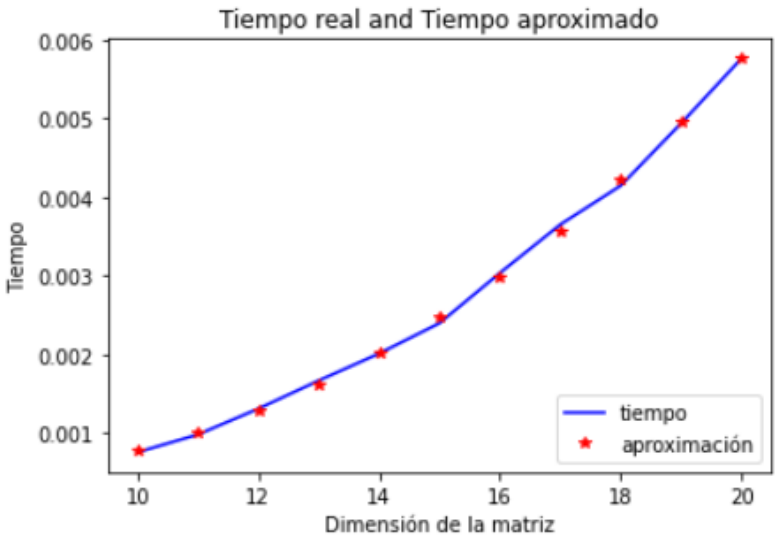
```
#I-A
def matrix_multiplication (m_1: np.ndarray, m_2: np.ndarray) :
    return np.multiply(m_1, m_2)

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])

t = np.array (l_timings)
print (t)
```

```
[[1.00000000e+01 1.10700000e-06]
 [1.10000000e+01 1.05400000e-06]
 [1.20000000e+01 1.32960000e-06]
 [1.30000000e+01 7.06899999e-07]
 [1.40000000e+01 7.16599999e-07]
 [1.50000000e+01 7.23799999e-07]
 [1.60000000e+01 7.35200001e-07]
 [1.70000000e+01 7.53800001e-07]
 [1.80000000e+01 7.52599999e-07]
 [1.90000000e+01 7.53800001e-07]
 [2.00000000e+01 7.55599999e-07]]
```

I-C. Cuestiones

1. ¿A qué funcion f se deberian ajustar los tiempos de multiplicacion de la funcion de multiplicacion de matrices?

```
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])

def fit_func_2_times(timings: np.ndarray, func_2_fit: Callable):
    if len(timings.shape) == 1:
        timings = timings.reshape(-1, 1)
    values = func_2_fit(timings[ :, 0]).reshape(-1, 1)
    times = timings[ : , 1]
    lr_m = LinearRegression()
    lr_m.fit(values, times)
    return lr_m.predict(values)
def func_2_fit(n):
    return n**3

times = np.array(l_timings)


t2 = fit_func_2_times(times, func_2_fit)

#Creo la figura y los ejes
#imprimo el tiempo de la funcion recursiva (time)
plt.plot(times[:, 0], times[: , 1], 'b', label="tiempo")
plt.plot(times[:, 0], t2[:], '*r', label="aproximación")
plt.title("%s and %s" % ('Tiempo real', 'Tiempo aproximado'))
plt.xlabel ("Dimensión de la matriz")
plt.ylabel ("Tiempo")
plt.legend (loc="lower right")
plt.show()
```



Nos da una gráfica que siempre es creciente, en este caso, es una función exponencial, más concretamente n al cubo. Esto es debido a que tendríamos tres sumatorios, los cuales irían de 0 a n de 1, por lo que al simplificar, nos queda n al cubo.

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

```
l_timing = []
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_timing.append([dim, timings.best])
s = np.array(l_timing)
print (f"Time spent without using .dot: \n\n{s}")
```

```
Time spent without using .dot:

[[1.00000000e+01 6.56100002e-07]
 [1.10000000e+01 6.65099998e-07]
 [1.20000000e+01 7.79599998e-07]
 [1.30000000e+01 7.20899999e-07]
 [1.40000000e+01 7.22300001e-07]
 [1.50000000e+01 7.09099999e-07]
 [1.60000000e+01 7.27099996e-07]
 [1.70000000e+01 7.27200000e-07]
 [1.80000000e+01 7.28700002e-07]
 [1.90000000e+01 7.66400001e-07]
 [2.00000000e+01 7.74599999e-07]]
```

```
timess = []
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)
    timess.append([dim, timings.best])
t = np.array(timess)
print (f"\nTime spent using .dot: \n\n{t}")
```

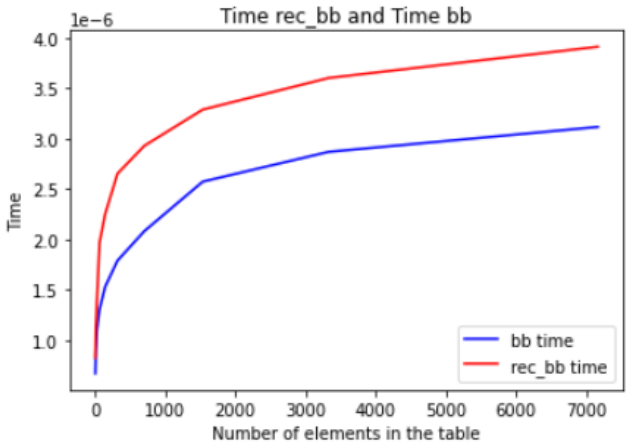
```
Time spent using .dot:

[[1.00000000e+01 6.71900000e-07]
 [1.10000000e+01 7.19200000e-07]
 [1.20000000e+01 7.03000001e-07]
 [1.30000000e+01 7.34800000e-07]
 [1.40000000e+01 6.97500002e-07]
 [1.50000000e+01 7.29300001e-07]
 [1.60000000e+01 7.29900000e-07]
 [1.70000000e+01 7.47199999e-07]
 [1.80000000e+01 7.67099999e-07]
 [1.90000000e+01 7.57499998e-07]
 [2.00000000e+01 7.91100001e-07]]
```

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?

```
#mido el tiempo de ejecucion de la funcion recursiva (rec_bb)
l_times = []
for i, size in enumerate(range(5, 15)):
    t = list(range(2**i * size))
    key = -1
    timings = %timeit -n 100 -r 10 -o -q rec_bb(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])
times = np.array(l_times)
print ("Tiempo de la funcion recursiva\n")
print(times)

#mido el tiempo de ejecucion de la funcion iterativa (bb)
l_times = []
for i, size in enumerate(range(5, 15)):
    t = list(range(2**i * size))
    key = -1
    timings = %timeit -n 100 -r 10 -o -q bb(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])
time = np.array(l_times)
print ("\nTiempo de la funcion recursiva\n")
print(time)
```

```
Tiempo de la funcion recursiva

[[5.00000e+00 1.16199e-06]
 [1.20000e+01 1.63499e-06]
 [2.80000e+01 1.98108e-06]
 [6.40000e+01 2.52469e-06]
 [1.44000e+02 2.95257e-06]
 [3.20000e+02 3.46637e-06]
 [7.04000e+02 4.01346e-06]
 [1.53600e+03 4.31336e-06]
 [3.32800e+03 4.89436e-06]
 [7.16800e+03 5.40166e-06]]

Tiempo de la funcion recursiva

[[5.00000e+00 9.25490e-07]
 [1.20000e+01 1.21309e-06]
 [2.80000e+01 1.44859e-06]
 [6.40000e+01 2.06088e-06]
 [1.44000e+02 2.62608e-06]
 [3.20000e+02 2.74068e-06]
 [7.04000e+02 3.24637e-06]
 [1.53600e+03 3.89317e-06]
 [3.32800e+03 2.86268e-06]
 [7.16800e+03 4.01587e-06]]
```

```
plt.plot(time[:, 0], time[: , 1], 'b', label="bb time")
plt.plot(times[:, 0], times[: , 1], 'r', label="rec_bb time")
plt.title("%s and %s" % ('Time rec_bb', 'Time bb'))
plt.xlabel ("Number of elements in the table")
plt.ylabel ("Time")
plt.legend (loc="lower right")
plt.show()
```

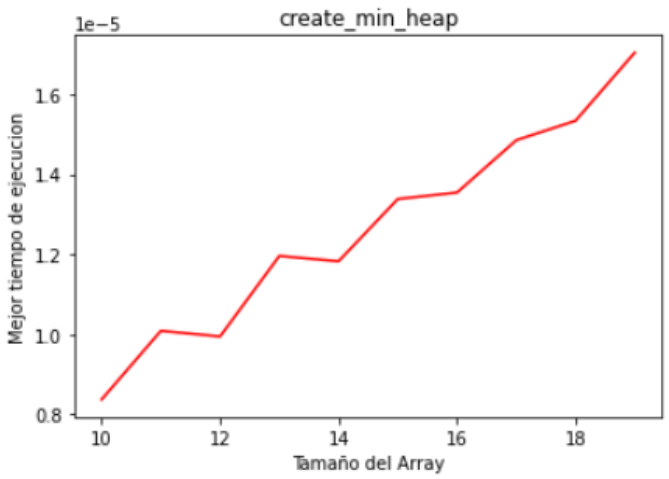


Como podemos observar en la gráfica, la función recursiva tarda más que la función iterativa y esto es debido a que las funciones iterativas, en general, son más eficientes, utilizan menos memoria y tiempo que las recursivas mientras que las funciones recursivas son caras (ineficaces) ya que ocupan mucha memoria y su ejecución emplea mucho tiempo.

II-D Cuestiones

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

```
from numpy.random.mtrand import permutation

num_Perms = 10
initialSize = 10

minHeap_Times = []

for i in range (num_Perms):
  h = np.random.default_rng()
  h = h.permutation(initialSize + i)
  t = %timeit -n 10 -r 10 -o -q create_min_heap(h)
  minHeap_Times.append([len(h), t.best])

tim = np.array(minHeap_Times)

xPoints = [item[0] for item in minHeap_Times]
yPoints = [item[1] for item in minHeap_Times]

plt.plot (xPoints, yPoints, color = 'r')

plt.title('create_min_heap')
plt.xlabel('Tamaño del Array')
plt.ylabel('Mejor tiempo de ejecucion')

plt.show()
```



Como vemos el tiempo de ejecución de la función de creación de min heaps se asemeja a una función lineal en cambio este se ajusta a $f(n) = n \log(n)$, esto se debe a que el coste de la función heapify es ($\log(n)$) y realizaremos heapify n/2 de veces por tanto el coste de la función de creación de min heaps será $O(n \log(n))$.

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

La función para el problema de selección tendrá coste máximo cuando la clave esté en la última posición del array, esto es porque habrá que crear un heap para cada elemento del array y esto tendría un coste $O(n \log(n))$ como hemos visto anteriormente. Por el contrario, si la clave se encontrase en una de las primeras posiciones del array, la función del problema de selección tendría coste $O(n)$. Por tanto, en función de K la función tendría un coste (n log (k)).

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.

Cómo creamos un min heap con tamaño K que al final tiene K primeros elementos invertidos estando el elemento que ocuparía la posición K de la raíz, y queremos obtener los elementos restantes en orden, únicamente haría falta extraer el valor de la raíz K veces dejando vacío el heap.

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 se podría. Existen distintas maneras de implementarlo, pero todas tienen un inconveniente, y es que necesitan de un sitio donde guardar los dos menores elementos. Una de las formas de implementarlo sería usando un array de tamaño dos con el menor elemento situado en la primera posición del array. O utilizar la función para el problema de selección.