ALGORITMIA Y ESTRUCTURAS DE DATOS AVANZADAS

PRÁCTICA 1

ALEJANDRO RAÚL HURTADO FUERTES

CAMILO JENÉ CONDE

G-1281 Pareja 4

A lo largo de esta práctica vamos a aprender a trabajar con Min Heaps y Python. Para ello, vamos a utilizar Google Colab, que hace la función de Jupyter Notebook.

Hay dos apartados: el primer apartado tiene tres letras; y el segundo apartado 4.

APARTADO 1

a. Aprender a medir tiempos con %timeit:

En este apartado vamos a utilizar la orden mágica *%timeit* que se va a encargar de calcular el tiempo que tarda en ejecutar (en este caso) una multiplicación de matrices.

La función *matrix_multiplication* recibe dos argumentos que son ambas matrices en forma de array que utilizará para multiplicar.

Tras ejecutar el script dado, nos muestra el siguiente resultado:

```
[[10, 0.0006670700000000806],
 [11, 0.0008828307000001701],
 [12, 0.0011575586000006411],
 [13, 0.0014343392999990102],
 [14, 0.001822055799999589],
 [15, 0.0021857783000001517],
 [16, 0.0026845892999986633],
 [17, 0.0031625152999993134],
 [18, 0.0037562191999995775],
 [19, 0.00447963470000019],
 [20, 0.0051495833000000415]]
```

b. Medida de tiempos de ejecución de funciones recursivas e iterativas.

Se nos pide realizar dos funciones de búsqueda binaria: recursiva e iterativa. Éstas reciben una lista ordenada, sus índices first y last y la clave que debe buscar.

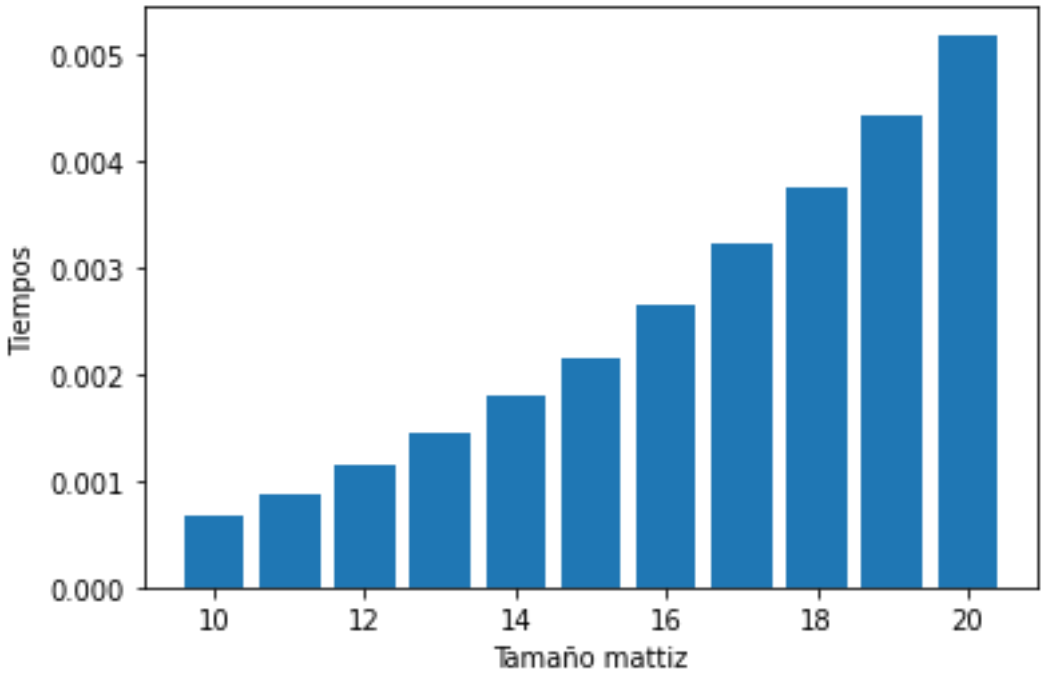
El algoritmo consiste en ir partiendo en mitades hasta encontrar el número correcto. La función recursiva se irá llamando a sí misma hasta encontrar la solución y la iterativa lo realizará dentro de un bucle.

Adecuando el script que nos piden, el resultado es el siguiente:

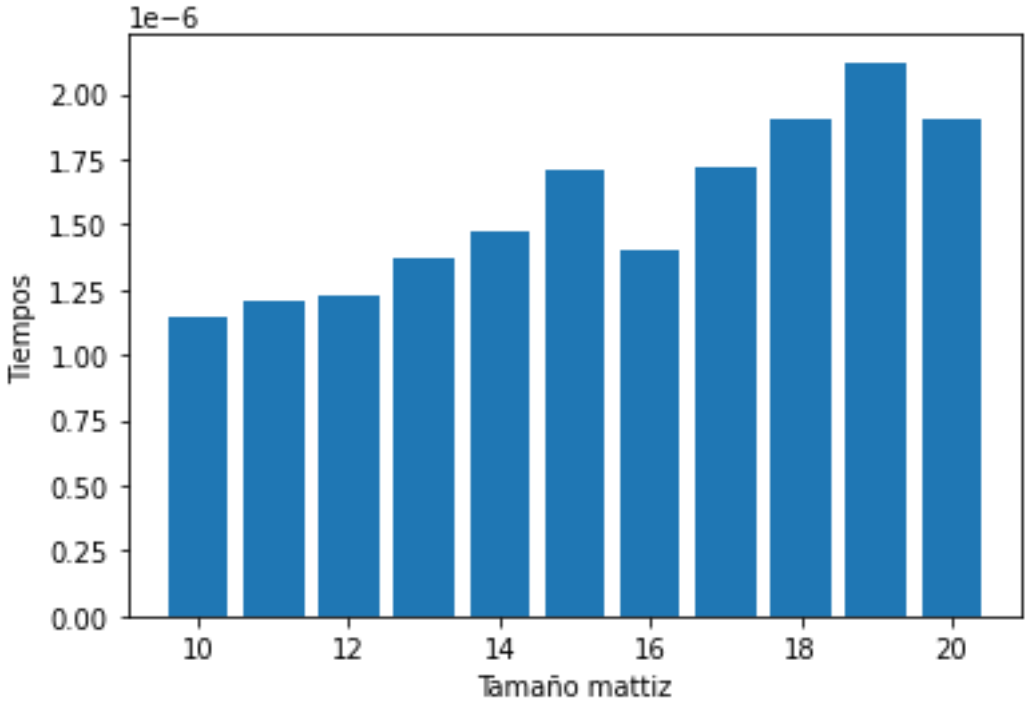
rec_bb	bb
[[5.00000e+00 6.32960e-07] [1.20000e+01 9.35060e-07] [2.80000e+01 1.68961e-06] [6.40000e+01 2.09746e-06] [1.44000e+02 1.45911e-06] [3.20000e+02 2.71426e-06] [7.04000e+02 2.97497e-06] [1.53600e+03 3.04678e-06] [3.32800e+03 3.58344e-06] [7.16800e+03 4.14630e-06]]	[[5.00000000e+00 5.94950002e-07] [1.20000000e+01 9.81910002e-07] [2.80000000e+01 1.24094000e-06] [6.40000000e+01 1.06647000e-06] [1.44000000e+02 1.25155000e-06] [3.20000000e+02 2.17394000e-06] [7.04000000e+02 2.52647000e-06] [1.53600000e+03 2.58974000e-06] [3.32800000e+03 2.70023000e-06] [7.16800000e+03 3.39745000e-06]]

c. Cuestiones.

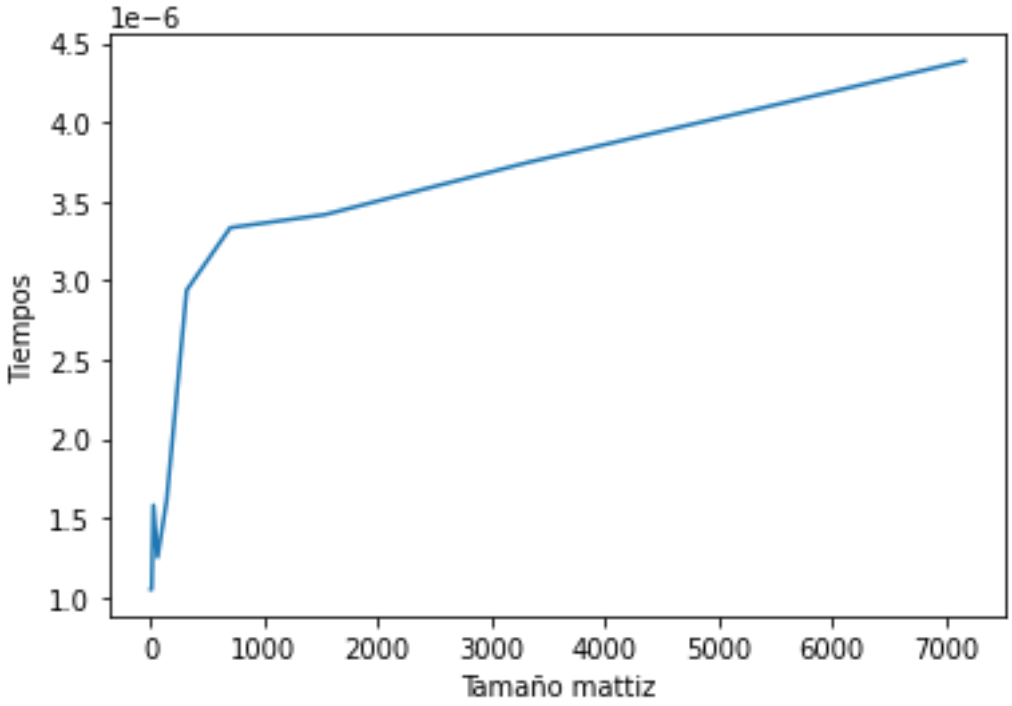
Tras ejecutar la multiplicación de matrices comprobamos que el tiempo de ejecución aumenta de forma exponencial según aumenta el tamaño de las matrices multiplicadas.



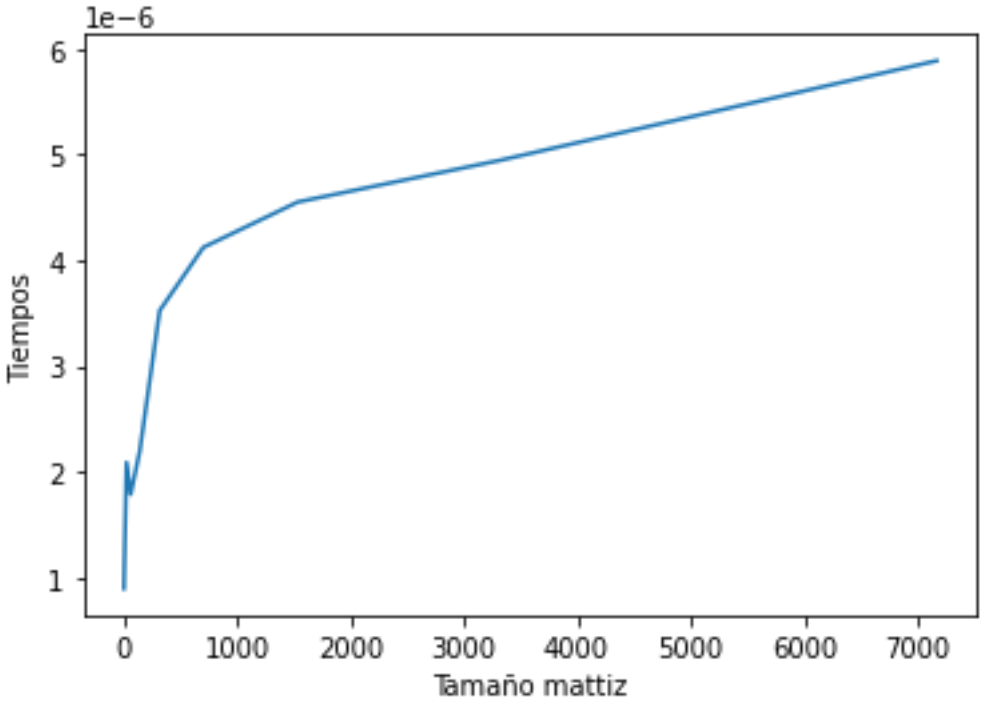
Al ejecutar el algoritmo con la función .dot() de numpy, se puede observar que el código está mucho más optimizado. Saliendo un tiempo mucho menor.



Por último, vamos a comparar las dos funciones creadas de la búsqueda binaria. Cómo se puede observar, tarda más en realizarse la iterativa que la recursiva.



Búsqueda binaria recursiva



Búsqueda binaria iterativa

APARTADO 2

a. Trabajar sobre Min Heaps y arrays de Numpy.

En este apartado se nos pide escribir tres funciones para el correcto funcionamiento de los Min Heaps: *min_heapify, insert_min_heap y create_min_heap*.

Para la comprobación del correcto funcionamiento de las funciones, hemos utilizado el script que nos dan en Moodle.

b. Colas de prioridad sobre Min Heaps.

En este apartado nos piden crear las funciones necesarias para poder trabajar con colas de prioridad (baja). El remove, debe eliminar el elemento de menor valor de prioridad con la función: *min(h)*.

Es importante saber que pq_ini() devuelve una cola None.

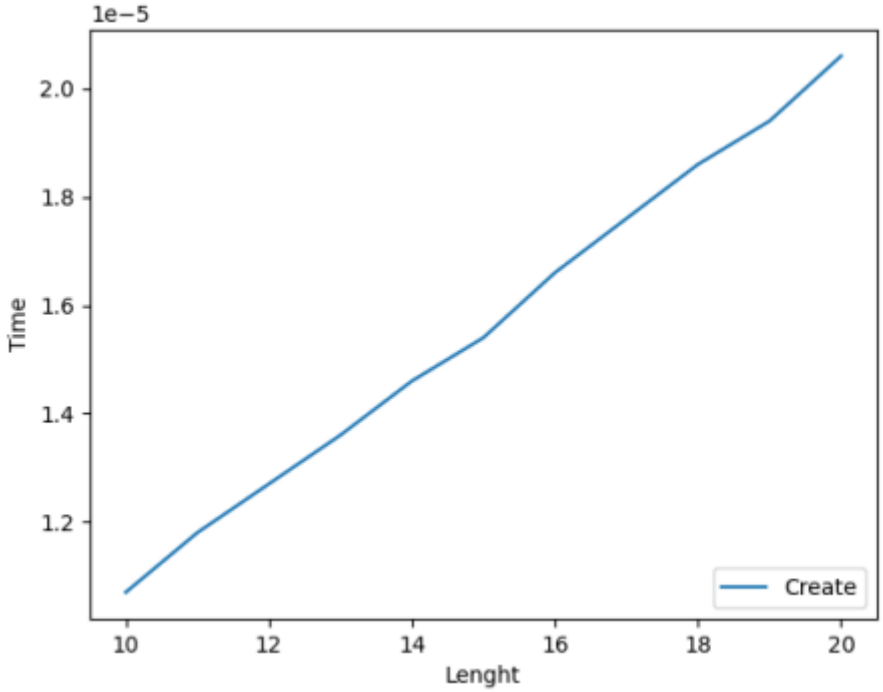
c. El problema de selección.

Debemos implementar una función que implemente el algoritmo del problema. El cuál consiste en encontrar el elemento de un array desordenado con N elementos que ocuparía el lugar k-ésimo.

Utilizando Max-Heaps creamos la función *select_min_heap()*.

d. Cuestiones.

1. La gráfica de tiempo es:



La F a ajustar es f(x) = x/2.

2. Sabemos que para crear el MaxHeap, el coste es O(n/2) y realizar heapify cuesta O(n*log(k)), resultando: O(n/2) + O(n*log(k)).
3. La función pq_remove creada, se encarga de mantener ordenado el heap. Por lo tanto, si se crease un array dónde se guarden los valores que vamos sacando, podríamos asegurar que lo resultante sería un array ordenado.
4. Sí, se podrían obtener los dos elementos, utilizando funciones creadas en la práctica. Primero realizamos el min_heapify y luego los dos remove, obteniendo los dos valores más pequeños.

```
arr = min_heapify()
for i in range(0, N):
    x = pq.remove(arr)
```

Siendo N el número de valores que queremos obtener.