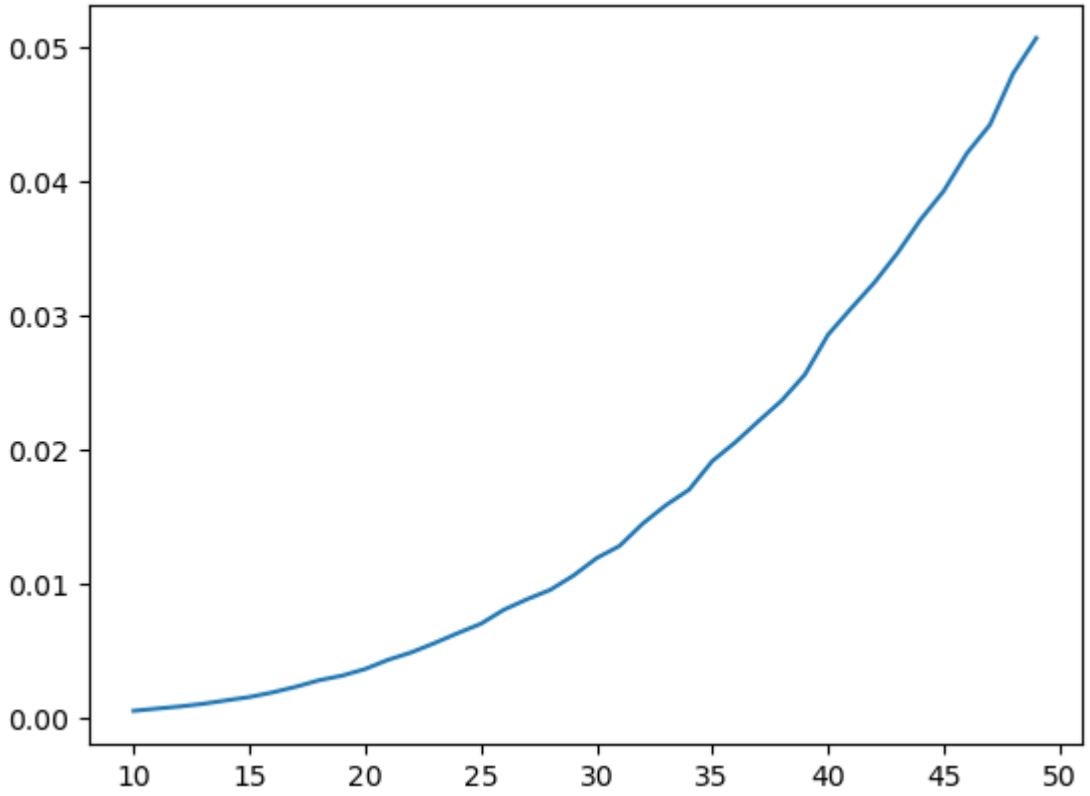
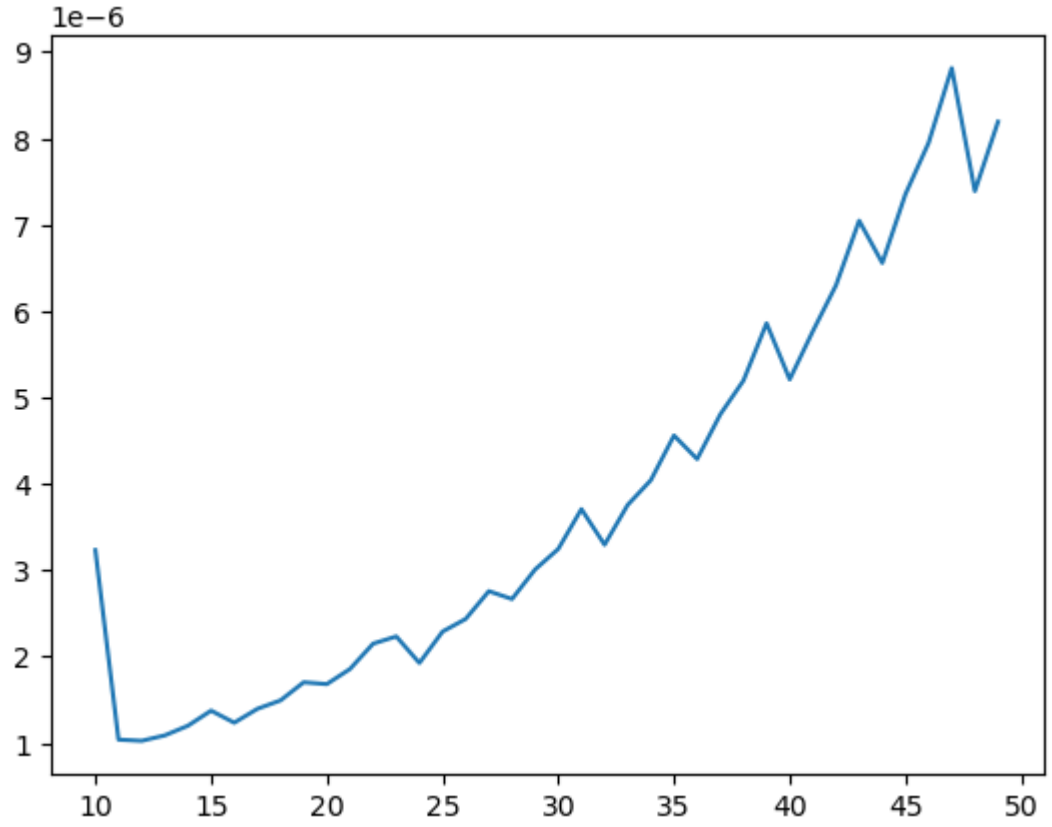
Practica 1
Pareja 11
Marcos Alonso Pardo y Daniel Cruz Navarro

Cuestiónes IC
1.
A la función a la que se debe aproximar la función es a n al cubo, ya que recorre tres bucles anidados de tamaño n. Al hacer que la función retorne n al cubo el ajuste nos da `[0.99258455 1.3009011  1.67069466 2.10755405 2.6170681]`
2.
A continuación, tenemos dos gráficas, en ambas el eje y representa los segundos y el eje x representa la longitud de ambos lados de la matrix(matriz cuadrada).



Gráfica de matrix_multiplication(), nuestra función.

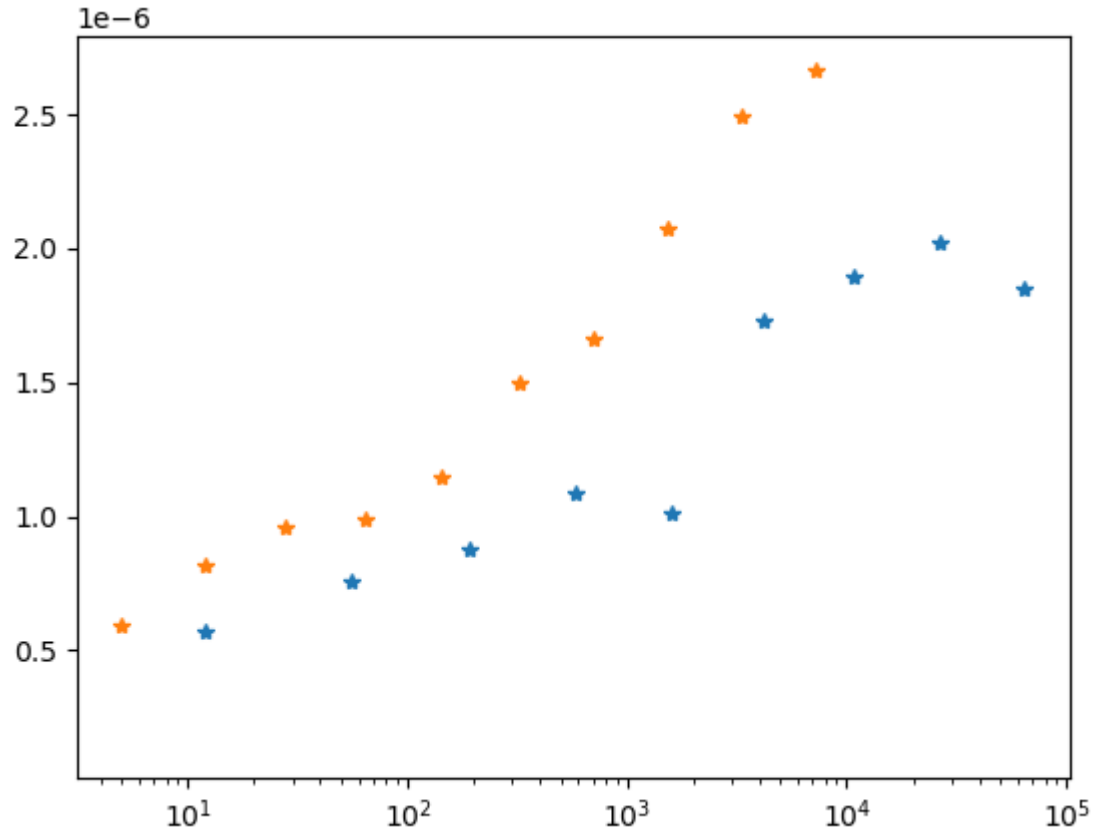


Gráfica de a.dot de numpy

Como se observa, en ambas tenemos un crecimiento exponencial con una curva parecida, aunque en a.dot() se observan diversos picos mientras matrix_multiplication() si es una curva casi total, sin embargo, a.dot() es mucho más rápida en cuanto a tiempos de ejecución, sus tiempos están en torno a los 10^-6 segundos mientras que matrix_multiplication() llega a tardar en su peor caso hasta unos 0,05 segundos(frente a los 9*10^-6 de a.dot()). Podríamos concluir que matrix_multiplication() es bastante lenta, sobre todo si la comparamos con a.dot() de numpy.

3.

A continuación vamos a comparar las funciones bb() y rec_bb(), la misma función de búsqueda binaria en modo iterativo y recursivo respectivamente. Abajo tenemos una gráfica comparando los tiempos de ejecución de ambas funciones(eje y) según el tamaño de la lista(eje x). Los puntos amarillos se corresponden con rec_bb() y azules con bb().-



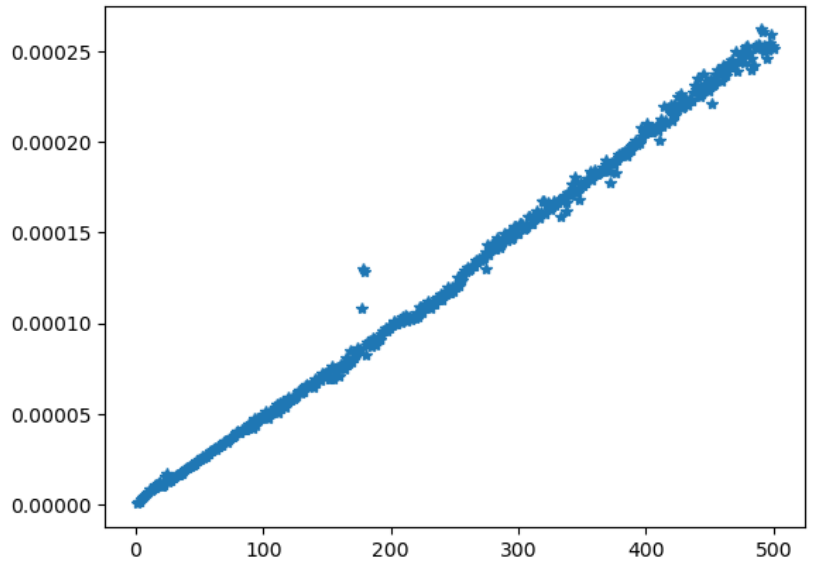
Como esperábamos, los tiempos de ejecución crecen de forma logarítmica, log(n), siempre crecen más pero cada vez crecen menos. Por ejemplo, se observa un crecimiento parecido en los tramos 100-1000 y 1000-10000 cuando el segundo es mucho más grande.

También se percibe que la función bb() es algo más rápida que rec_bb() en todos sus puntos, además de que tiene un gasto menor de memoria.

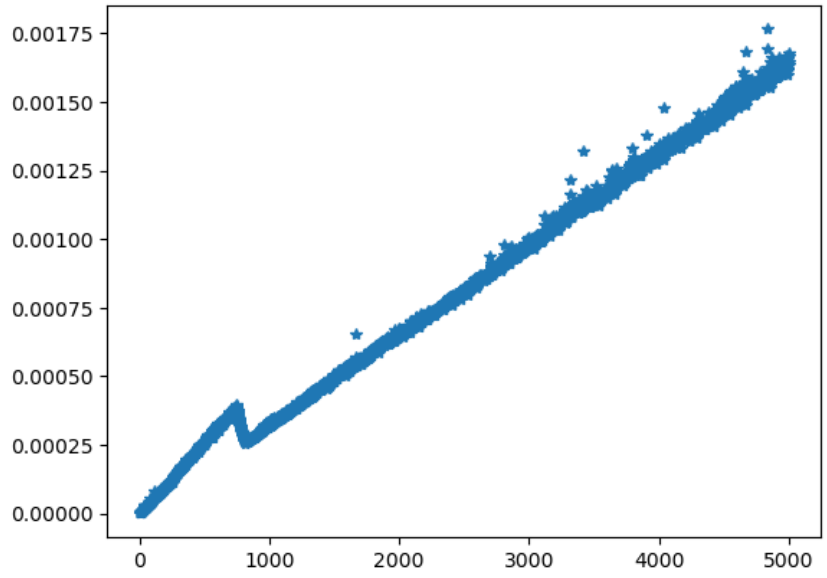
Cuestiones IID

1.

En la teoría, debería salirnos una función de la forma f = O(nlogn) o f = (n*logn)/2, siendo n el número de la longitud de la lista. Esto se debe a que min_heapify() que es O(logn) está dentro de create_min_heap() que tiene un bucle que corresponde a O(n).



En la práctica, en los tiempos de ejecución nos sale una función más parecida a f = O(n)



2.
Nuestra función de select_min_heap() consta de dos bucles, uno que es la función create_min_heap(), con f = O(nlogn) y otro bucle que se ejecuta k veces y tiene la función pq_remove() anidada, que es de O(logn), siendo este segundo bucle equivalente a f = O(k * log(n)), nuestra función al completo quedaría como f = nlogn + klogn, lo que es lo mismo a f = O(nlogn) ya que siempre se cumple k <= n

3.
La función select_min_heap() nos da lo que hay en el índice k, con esta información , haríamos un bucle que pase por todos los elementos en el cual iríamos comprobando si estos son menores que el número que hay en el índice k, si esto es así significa que el elemento está en los k primeros números.
4.
Si, utiilizando dos variables min1 y min2, recorreríamos la lista y haríamos dos comparaciones, si el elemento es menor que ambas variables min1 pasa a ser el nuevo elemento y min2 pasa a ser el antiguo min1, si el elemento es menor que min2 solo, se intercambian el elemento y min2.