ALGORITMOS Y ESTRUCTURAS DE DATOS: PRÁCTICA 1

INTRODUCCIÓN.

Esta primera práctica nos ha introducido rápidamente a las bases de Python, lo que nos permitirá en un futuro utilizar estos recursos aprendidos, para crear códigos mucho más complejos.

Las bases de Python a las que nos referimos son las listas, útiles siempre que trabajemos con datos concretos que queremos buscar, las funciones, que permiten modularizar y simplificar un gran problema, en subproblemas más sencillos, y fáciles de resolver (además, podrían ser reutilizadas en un futuro), y la recursión, que en ocasiones facilita ciertos problemas conocidos como Divide y vencerás, llamando a la función automáticamente.

En los ficheros listas.py, funciones.py y recursion.py, se han implementado las funciones para resolver los problemas, tal y como se pide. Como se ha comentado previamente en la descripción de cada fichero, hemos querido simplificar las funciones al máximo, para centrarnos en el problema que nos interesa. Es decir, no nos hemos centrado en problemas secundarios, como cerciorarnos de que se introducirán los datos en la manera en que exigen las funciones (listas y números en estos casos, lo cual se comprobaría utilizando métodos de cadenas y booleanos), sino que suponemos que se han introducido los datos adecuados, y por ello, simplemente se devuelven, o bien listas con los datos exigidos, booleanos, o listas vacías (si el elemento no se encuentra en la lista).

También se ha de tener en cuenta, que hemos tratado de crear funciones, lo más económicas y eficientes posibles, aplicando la Búsqueda lineal, teniendo en cuenta que se exige que los datos que se introducen sean dos: una lista y un f(x), es decir, no hemos podido utilizar recursos conocidos como, por ejemplo, la Búsqueda binaria (dado que se han de implementar 4 inputs: una lista, un f(x), un número elemento inicial y uno final; y, además, en caso de que se mezclen caracteres alfanuméricos, no sería posible ordenarla correctamente), o el Backtracking (que hubiera sido mucho más ineficiente).

Una vez creadas estas funciones, hemos querido analizar la eficiencia de dos en concreto: la search_loop, y la search_comprh, las cuales realizan el mismo trabajo, pero como comentaremos más adelante, una es más eficiente que la otra.

OBJETIVOS.

Como se ha introducido, los objetivos primarios de esta parte de la práctica (la parte 3), han sido:

1) Medir la eficiencia de las funciones ***search_loop***, y ***search_comprh***, las cuales habíamos creado nosotros previamente. Para ello, hemos medido específicamente la media del tiempo de ejecución, y su desviación típica.
2) Comparar cuál de las dos funciones es más eficiente, explicar el por qué, tanto para el caso peor, como para el caso promedio.

1. ANÁLISIS DE LOS GRÁFICOS.

Tanto la ***search_loop***, como la ***search_comprh***, son funciones que devuelven *True* si hay por lo menos un elemento de la lista (lst) que cumple el predicado f, y *False*, si no lo hay.

Sin querer entrar en detalle de su código, (ambas se han adjuntado en el fichero funciones.py), sabemos que, en ambos casos, se recorre la lista de longitud variable, y se comprueba, elemento a elemento, si alguno de ellos coincide con el parámetro (f(x)) que se exige (éste es modificado con *lambda x*), sin embargo, la función que incorpora el bucle, *loop*, para, en cuanto encuentra un sólo elemento que coincide con la descripción, y en cambio, la función *comprehension*, debe recorrerse toda la lista, dado que carece de bucle, aun habiendo comprobado si existe al menos un elemento con el valor a buscar. Esto será clave para el análisis de los gráficos.

1.1. *CASO PROMEDIO.*

El tiempo efectivo de ejecución de un algoritmo depende de muchos factores, según la función a analizar. En nuestro caso, será el tamaño de la lista, es decir, menor tiempo de ejecución se necesitará, cuanto menor sea la lista a recorrer.

El temporizador que tiene Python incorporado, ***basic_time(array,val)***, devuelve el tiempo de ejecución en milisegundos. Por tanto, es importante tener en cuenta que, en una lista de corto tamaño, aun suponiendo el caso peor, el tiempo de ejecución es casi cero. Por tanto, hemos creado un algoritmo (NSTAT) que genera, una serie de listas de longitudes crecientes (hasta alcanzar los 10100 elementos), cuyos elementos (números) son completamente aleatorios, así como la aleatoriedad del elemento a buscar en la lista (este input será un entero entre -1000 y 1000). En cuanto se termina de recorrer la lista, se genera otra lista, con otro input, y así sucesivamente, hasta haber alcanzado, en nuestro caso, una lista de longitud 10100.

Por tanto, este algoritmo permite predecir que es estadísticamente elevada la probabilidad de que el elemento a buscar, efectivamente, se encuentre en la lista generada aleatoriamente, (será encontrado antes o después, pero muy probablemente sea encontrado). Esto es conocido como caso promedio. Cabe destacar, que hemos forzado al programa a que trabaje con números (evitando cualquier otro tipo de signos) y siempre con enteros, ya que la probabilidad de encontrar *floating points*, sería prácticamente 0, es decir, sería tomado como un caso peor.

Queremos estudiar las funciones de dos maneras: calculando la media y la varianza:

La **media** es, en nuestro caso, el sumatorio de los tiempos (tiempo que se tardaba en comprobar si el elemento estaba, o no, en la lista), entre el número de veces que se ha aplicado NSTAT:

$$T_M(n) = \frac{1}{\texttt{NSTAT}} \sum_{i=1}^{\texttt{NSTAT}} t_i^{(n)}$$

Ahora bien, la teoría predice que la búsqueda lineal muestra una gráfica, lógicamente, lineal, dado que el tiempo de ejecución medio, se podría calcular como:

$$T(n) = \frac{c_1 + c_2}{2} n + c_2$$

Y aun tendiendo *t* a infinito, la función permanece igual:

$$\lim_{n\to\infty} \frac{T(n)}{n} \lim_{n\to\infty} \frac{\frac{c_1+c_2}{2} n + c_2}{n} = \frac{c_1 + c_2}{2}$$

Por tanto, que la gráfica resulte lineal, significa que, a mayor longitud de la lista, mayor es el tiempo que tardará en encontrarse el valor específico.

La **varianza** por el otro lado, es una medida de dispersión. Eso significa que pretende capturar en qué medida los datos están en torno a la media, por lo que un gráfico que muestre la curva muy elevada en el eje y, implica que los datos se encuentran muy alejados del valor medio, y que la media simplemente ha salido por la compensación entre dos valores muy alejados el uno del otro, y viceversa. Se calcula como la raíz cuadrada de la varianza, la cual se calcula como:

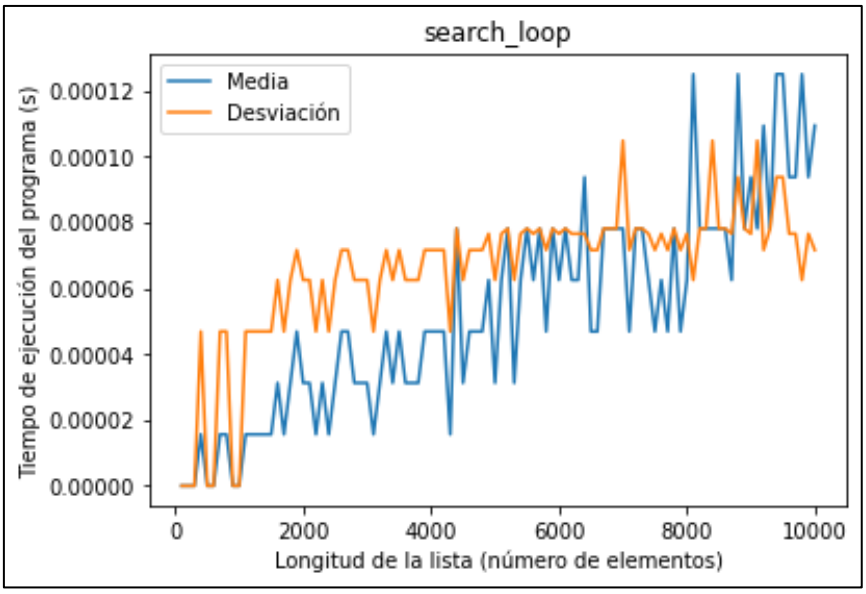
$$\sigma^2(n) = \frac{1}{\texttt{NSTAT}} \sum_{i=1}^{\texttt{NSTAT}} (t_i^{(n)} - T_M(n))^2$$

Es decir, la raíz cuadrada del sumatorio de las distancias de los tiempos reales, a los tiempos dados por la media.

Es lógico pensar que, dada una serie de listas con datos completamente aleatorios, la media se ha calculado a base de equilibrar tiempos muy altos, con tiempos muy bajos. Ahora lo visualizaremos una serie de gráficas.

1.1.1. *FUNCIÓN SEARCH_LOOP*

Recordemos que la función en cuestión se ha programado de tal forma que se van a recorrer las diferentes listas, y una vez encontrado el elemento, se devolverá un *True* (o *False* si no coinciden). También hemos predicho que obtendremos una curva prácticamente lineal, pues cuanto mayor es la lista, mayor es el tiempo de ejecución.

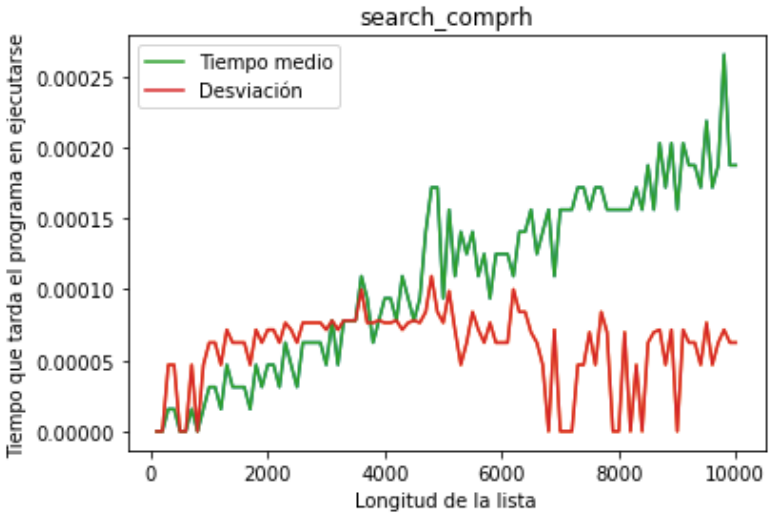


Obviando la forma de sierra de ambas curvas (debido a la aleatoriedad de los elementos de la lista y el input), podemos comprobar visualmente que la curva del tiempo medio (línea azul) tiende a ascender, según se elonga la lista, es decir, crea una **curva lineal**.

Por el otro lado, la curva de la desviación (línea naranja), parte de valores bajos (poca diferencia entre los máximos y mínimos), y según aumenta la longitud de la lista, aumenta la varianza, hasta quedar prácticamente constante a mitad de la gráfica, lo que significa que se ha alcanzado un punto en el que los tiempos reales quedarán muy alejados de la media.

1.1.2. *FUNCIÓN SEARCH_COMPRH*

En el caso de la función search_comprh, debemos tener en cuenta que, aunque cumple la misma función que el bucle, en este caso, se recorre siempre la lista, desde el elemento 0, hasta el elemento len(lst), haya encontrado el valor a buscar, o no. Obtendremos una curva, visiblemente más lineal que la de la función search_loop.

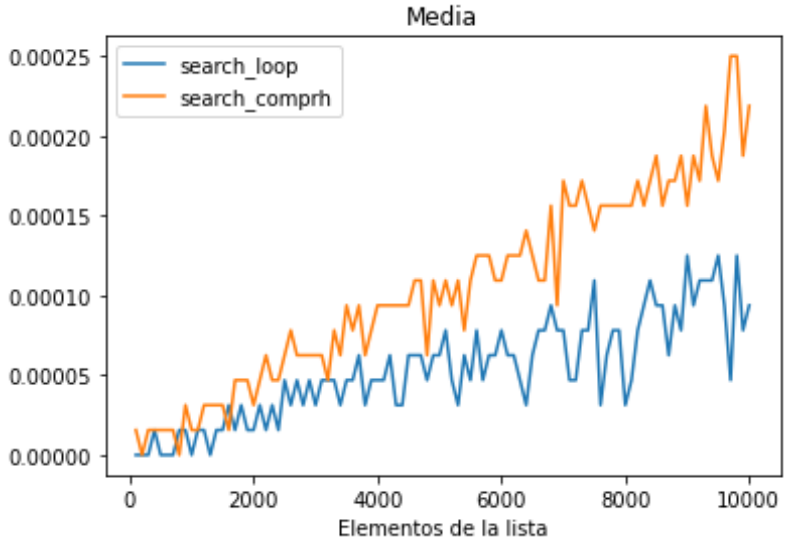


El tiempo medio es visiblemente **lineal** (curva verde), ya que, al contrario de lo que ocurría con el bucle, esta función no permite cambiar de lista al encontrar el elemento que busca, sino que debe recorrerlas por completo (por tanto, son proporcionales el tiempo que tarda en ejecutarse el programa, y la longitud de las listas).

La desviación, en cambio, vemos que tiene picos de bajada según aumenta la longitud de la lista, puesto que los valores obtenidos del tiempo real medido, no se alejan tanto de la media, que también será bastante alta.

1.1.3. *CONCLUSIÓN DE LOS DATOS OBTENIDOS EN LOS CASOS PROMEDIOS*

En el gráfico que se adjunta, se comparan los tiempos medios de ambas funciones, caracterizando la search_loop con el color azul, y la search_comprh con el naranja. Mediante esta gráfica se comprueba rápidamente, cuál de las dos funciones tarda, en tiempo medio, menos en ejecutarse.

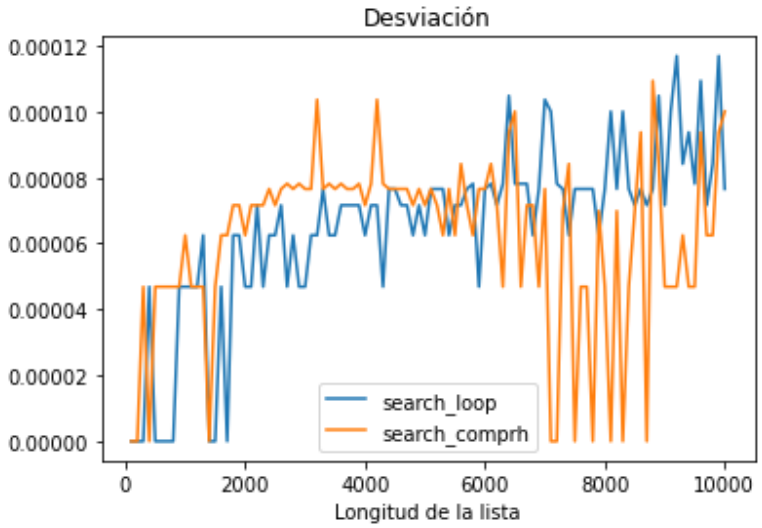


Comprobamos que ambas gráficas siguen un aspecto lineal. Sin embargo, se distingue que a medida que se agranda la lista, se separan más las curvas, lo que implica que la que está por encima (la search_comprh), tarda bastante más en ejecutarse (a llegar a la lista de 10.000 elementos, tarda prácticamente el doble (0.00025 segundos) de lo que lo hace el search_loop (0.00012 segundos)).

Este fenómeno ocurre porque, el bucle ha sido creado de forma que, una vez se encuentra el elemento que se busca en la lista, se pasa a una lista nueva, sin la necesidad de terminar de recorrerla, al contrario que el método comprh, el cual obliga a recorrer la lista entera, a pesar de haber encontrado el elemento que se busca.

Por tanto, apoyándonos en estos datos, podemos concluir que, en listas cortas, ambas funciones tardan tiempos medios parecidos en ejecutarse, pero a grandes rasgos, el bucle **search_loop**, se sitúa por delante en **eficiencia y economía** de la memoria (puesto que habrá casos en los que no tenga que rastrear listas enteras, como ya se ha explicado).

En el siguiente gráfico, analizaremos las comparativas de las desviaciones de ambas funciones, denotando el color azul el search_loop, y el naranja, el search_comprh.

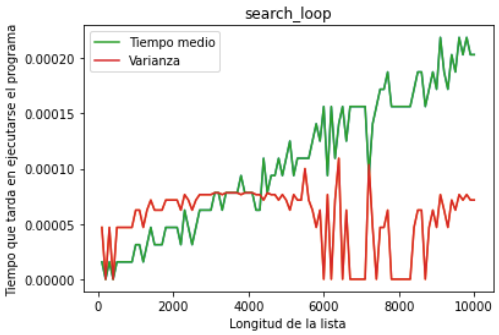


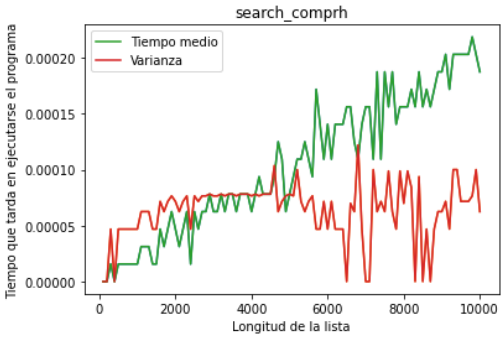
Se distingue que, hasta las listas de 2000 elementos, ambas funciones tocan valores de 0 (es decir, hay muchos casos en los que la diferencia de los tiempos reales, no quedan muy alejados de las medias obtenidas). A medida que se agrandan las listas, la curva azul comienza a crecer, de manera logarítmica, y los valores de la varianza aumentarán, según aumente la longitud de las listas. La naranja, por el otro lado, aparenta algo más de estabilidad en torno a los 0.00008 segundos, y habrá puntos en los que bajen hasta tocar el eje de abcisas.

Gracias a este gráfico, volviendo a la explicación dada para los tiempos medios, se demuestra una vez más, que la función search_comprh recorre las listas hasta el final, por lo que es lógico pensar que a medida que crece la gráfica, dado que los tiempos reales aumentan, el tiempo medio también lo hará, pero de manera más constante, e incluso habrá ocasiones (las que cortan con el eje de abcisas), en las que los tiempos obtenidos no se alejen del tiempo medio. En cambio, puesto que, en el bucle, puede parar de ejecutarse la función, incluso a principios de la lista, los valores siempre estarán muy alejados del valor del tiempo medio (y la curva prácticamente nunca tocará el eje x).

1.2. *CASO PEOR.*

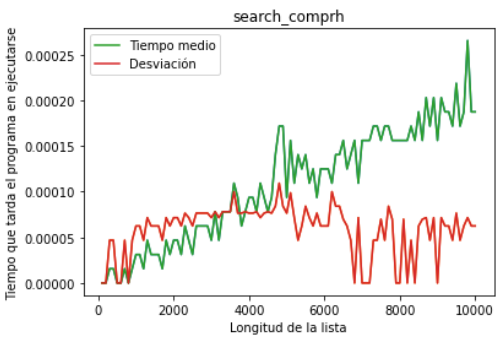
El caso peor se refiere a los casos en los que los tiempos de ejecución se vuelven máximos. Trabajando con listas, el principal caso de caso peor, es el hecho de no encontrar el elemento que se busca. Hemos querido modificar ciertos aspectos de las funciones (obligando al elemento que se busca, a que se encuentre fuera de la longitud de la lista), para observar qué ocurriría con el tiempo de ejecución medio en los casos peores de ambas funciones.

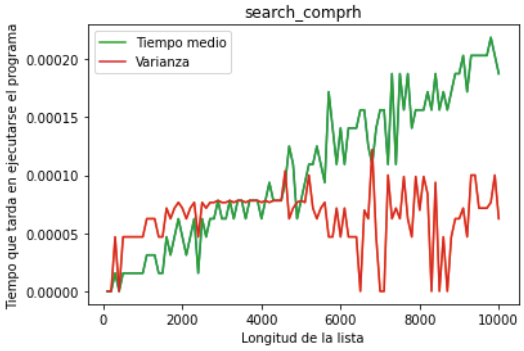




Es más interesante analizar el caso peor, comparando directamente ambas gráficas, puesto que salta a la vista que realizan esencialmente la misma curva, es decir, la media se dispara de manera lineal, y la varianza, a partir de la mitad del gráfico, tiene puntos donde corta con el eje de abscisas. La curva tiene sentido, ya que, puesto que se deben recorrer todas las listas, elemento a elemento, cuanto más larga sea, más tiempo tardará en ser recorrida. En este caso, el bucle search_loop no va a terminar de recorrer la lista en cualquier momento.

Una vez más, podemos deducir que la search_comprh es la función menos eficiente de las dos, puesto que se ha dibujado la misma curva, que para el caso promedio (el gráfico de arriba representa el caso promedio, y el de abajo, el caso peor):





CONCLUSIONES GENERALES

Podemos concluir que, aunque son programas simples y rápidos de ejecutarse, es importante elegir el algoritmo a utilizar, pues un mismo problema, se podrá solucionar de varias maneras, pero en muchos casos, habrá alguna que sea más económica, o bien para la memoria, o para el tiempo que tarda en ser ejecutado. En nuestro caso, la función search_loop era más eficiente en ambos aspectos que la search_comprh.