Rendimiento de los algoritmos

A la hora de ejecutar un algoritmo a partir de una entrada de datos, sea esta del tamaño que sea, el coste de cálculo (de tiempo y recursos) dependerá del volumen de entrada o del la maquina que lo ejecuta pero también, y más determinantemente, de la configuración del propio algoritmo. En el análisis de rendimiento de los algoritmos de computación, el parámetro *n* se denomina el “tamaño del problema” y se formaliza como una la función T(*n*) que es es el tiempo que se necesita para resolver un problema de tamaño *n*.

Para hacer el cálculo, se tienen en cuenta todas las operaciones de un algoritmo, o sentencias en terminología de programación, sin embargo al fin de cuentas el número exacto de operaciones no es tan importante como determinar la parte de la función que es más dominante en cuestión de coste de cálculo. Efectivamente a medida que el problema se hace más grande, solo las partes de la función donde interviene *n* se hacen relevantes.Para ello aplicamos un método de descripción (análisis asintótico) que nos permite centrarnos únicamente en el comportamiento de un algoritmo en el límite, en el que deberemos considerar solamente el o l os término más grande de la función, ignorando incluso los coeficiente del término o términos, y por supuestos los los términos unitarios. Por ejemplo en la función $f(n) = 6n^3 + 2n^2 + 20n + 45$ solo tomaremos como relevante con lo que la función se quedaría en: $f(n) \in O(n^3)$.

Para entender el tema hemos hecho una función de búsqueda en python, implementada de dos formas distintas, esto es, con algoritmos distintos y la hemos probado ambas con un mismo array.

Caso de algoritmo de búsqueda secuencial:

En el caso del algoritmo que hemos escrito para la busqueda lineal (archivo “*tarea45-busqueda-secuencial-main.py*”) la tasa de crecimineto del mismo se atendrá a la función:

T(n) = 2 + 4n

Para verlo más claro, vamos contando sentencia a sentencia el coste de cálculo según la variables o inputs que recibe la función:

```
def busqueda_secuencial(arr2, n):
    i = 0(1)
    largo = len(arr2) 0(1)
    while i < len(arr2) & largo != 0: 0(n)
        if arr2[i] == n: 0(1)
            print("El número solicitado se encuentra en la
posición: ", i) 0(1)
            i = i + 1 0(1)
            #return 1 0(1)
        else: 0(1)
```

```
            print("NO Hallado - Iteración num.: ", i) O(1)
            i = i + 1 O(1)
            #return -1 O(1)
```

Siendo el primer elemento irrelevante según *n* se incrementa, y siendo que tendiendo a infinito, incluso el cuádruple de infinito también sería irrelevante, nos quedamos con lo que marca realmente la complejidad del algoritmo vendría únicamente determinada por el grado de *n*, así podemos decir que, formalmente:

T(n) = O(f(n))

Es decir, la función f(n) describe la cota superior para T(n) o que la tasa de crecimiento de T(n) <= la tasa de f(n); resumiendolo:

T(n) = O(n)

En la búsquedas se ha de contar con el supuesto que el item a buscar se halle en el “mejor de los casos” (el ítem está al inicio de la búsqueda) o en “el peor de los casos” (el ítem está al final). En el caso del ejercicio anterior hemos admitido arrays con elementos duplicados por lo que podemos entender vemos que los ítem pueden estar en diferentes posiciones del array con lo cual la búsqueda habrá de ser, inevitablemente, sobre el total de los elementos del array, consecuentemente tendremos que asumir que el “mejor de los casos” no se dará.

A continuación mostramos la salida por consola del editor de python que hemos usado para esta tarea:

```
C:\Users\tutoretza3\PycharmProjects\tarea44\venv\Scripts\python.exe
C:/Users/tutoretza3/PycharmProjects/tarea44/tarea45-busqueda-secuencial
-main.py
NO Hallado - Iteración num.:  0
Hallado - Iteración num.:  1
El número solicitado se encuentra en la posición:  1
NO Hallado - Iteración num.:  2
NO Hallado - Iteración num.:  3
NO Hallado - Iteración num.:  4
NO Hallado - Iteración num.:  5
NO Hallado - Iteración num.:  6
NO Hallado - Iteración num.:  7
NO Hallado - Iteración num.:  8
NO Hallado - Iteración num.:  9
NO Hallado - Iteración num.:  10
NO Hallado - Iteración num.:  11
Hallado - Iteración num.:  12
El número solicitado se encuentra en la posición:  12
Process finished with exit code 0
```

Se aprecia como las iteraciones son 12 para una array de 12 elementos
En este caso el elemento a buscar ha sido el 56, que se hallaba en la segunda posición y también en la última posición del array.

Caso de algoritmo de búsqueda binaria:

En el caso de solucin de bsuqyeda bianaria (“tarea45-busqueda-binaria-main.py”), la tasa de crecimento responderi a:

T(n) = 4 + 8?+ 1

Para verlo claro, lo anotamos sentencia a sentencia a lo largo de la función principal:

```
def busqueda_binaria(arr, n):
    izq = 0 O(1)
    der = len(arr) - 1 O(1)
    medio = 0 O(1)
    i = 1 O(1)

    while izq <= der: O(?)
        medio = (der + izq) // 2 O(1)
        if arr[medio] < n: O(1)
            izq = medio + 1 O(1)
        elif arr[medio] > n: O(1)
            der = medio - 1 O(1)
        else: O(1)
            return medio O(1)
        i = i + 1 O(1)
    return -1 O(1)
```

Se aprecia como en el `while` hemos `O(?)` en lugar de `O(n)`, esto se debe a que en ningún caso se va a dar la busqueda sobre todos los caso de n; lo explicamos a continuación. La diferencia con la búsqueda lineal antes vista es que el algoritmo de búsqueda binaria usa un algoritmo diferente, parte de la ventaja de que, si el array está ordenado y por ello, la búsqueda del ítem en cuestión sólo habrá de realizarse sobre la mitad del array. Así tras la primera búsqueda, el array resultante para continuar buscando en el elemento, se ceñirá a la mitad del original, es decir, progresivamente va disminuyendo el número de elementos sobre el que realizar la búsqueda a la mitad: n, n/2, n/4, ... Así, tras log (n) divisiones se habrá localizado el elemento. Log (n) divisiones es el número máximo de veces que se puede reducir a la mitad la cantidad de registros a considerar, hasta que solo se tenga un elemento.
Así , podemos enunciar por ello que el coste del algoritmos será:

T(n) = 4 + 8·log n + 1

lo que, por método asintótico, quedaría en:

T(n) = O(f(log n))

o en notación convencional,

T(n) = O(log n)

A continuación mostramos la salida por consola del editor de python que hemos usado para esta tarea:

```
C:\Users\tutoretza3\PycharmProjects\tarea44\venv\Scripts\python.exe
C:/Users/tutoretza3/PycharmProjects/tarea44/tarea45-busqueda-binaria-ma
in.py
Posición  0  ----> item: 3
Posición  1  ----> item: 21
Posición  2  ----> item: 23
Posición  3  ----> item: 24
Posición  4  ----> item: 33
Posición  5  ----> item: 45
Posición  6  ----> item: 56
Posición  7  ----> item: 56
Posición  8  ----> item: 65
Posición  9  ----> item: 66
Posición  10  ----> item: 123
Posición  11  ----> item: 874
Posición  12  ----> item: 1000
Número de iteraciones:  1
Número de iteraciones:  2
Número de iteraciones:  3
El número solicitado se encuentra en la posición:  10

Process finished with exit code 0
```

Para finalizar, anotar una cuestión no desdeñable: en el código de la función de búsqueda binaria, afuera de la definición de la función propiamente dicha, hemos añadido una función extra de ordenación de elementos de un array:

```
arr2 = sorted(arr, reverse=False)
```

La razón es que hemos usado el mismo array para los dos tipos de búsqueda, el secuencial y el binario, pero, como el binario requiere una lista ordenada, hemos optado por recurrir a esa función incluida en la librería de python. Nuestra observación viene a cuento porque suponemos que dicha función, en su labor de ordenamiento de elementos tendrá un coste de cálculo de tipo O(n) lo cual añadiría complejidad a la función de búsqueda en la que está incluida que, recordamos, era de tipo O(log n).

Resumen

De forma gráfica podemos dar a entender las diferencia de rendimiento de estos dos algoritmos referidos mediante el siguiente gráfico: