***Memoria Práctica 3***

*Algoritmos de Búsqueda*

Fundamentos de Análisis de Algoritmos

Álvaro Esteban Muñoz y Francisco José Gil Durán

Grado en Ingeniería Informática (1º Curso)



# **Índice**

[**Índice** 2](#_gjdgxs)

[*0.*](#_30j0zll) *Introducción. Algoritmo de Ordenación.* 3

[*1.*](#_1fob9te) *Cálculo de los tiempos teóricos.* 3

[*1.1.*](#_3znysh7) *Pseudocódigos y análisis de coste.* 3

[*1.1.1. Algoritmo Burbuja* 3](#_2et92p0)

[*1.1.2. Algoritmo Inserción* 4](#_tyjcwt)

[*1.1.3. Algoritmo Selección* 5](#_3dy6vkm)

[*1.2. Tablas y Gráficas de coste teórico.* 5](#_4d34og8)

[*1.3.*](#_2s8eyo1) *Conclusiones* 6

[*2.*](#_17dp8vu) *Cálculo de los tiempos experimentales* 6

[*2.1.*](#_3rdcrjn) *Tablas y Gráficas de coste experimental.* 6

[*2.2.*](#_26in1rg) *Conclusiones.* 8

[*3.*](#_lnxbz9) *Comparación de los resultados teórico y experimental.* 8

[*4.*](#_35nkun2) *Diseño de la aplicación.* 8

[*5.*](#_1ksv4uv) *Conclusiones y valoraciones personales de la práctica.* 8

1. *Introducción. Algoritmos de Búsqueda.*

Esta práctica es muy similar a nuestra práctica 2 que consistía en realizar el análisis experimental de tres algoritmos de ordenación, sin embargo, en esta nueva práctica procederemos a hacer un análisis de tres algoritmos de búsqueda, Secuencial, Binaria y Ternaria.

Realizaremos el análisis temporal de estos tres algoritmos y los compararemos los unos con los otros. Cabe destacar que nuestra aplicación se realizará como un submenú de la práctica 2.

1. *Cálculo de los tiempos teóricos.*
   1. *Pseudocódigos y análisis de coste.*

Pasamos a realizar un análisis teórico de nuestros algoritmos de búsqueda.

*1.1.1. Algoritmo Búsqueda Secuencial*

Este algoritmo es de una comprensión bastante sencilla, consiste en comparar elemento a elemento con la clave a buscar hasta encontrarla, una vez encontrada se detiene, si dicha clave no está en el vector, lo recorrerá entero y devolverá -1.

**función** BusquedaSecuencial(V[1 … n]; size, key:int)

i🡨1;

**mientras** (V[ i ]!= key && i<=size) **hacer** Bucle de n-1 hasta i

i**🡨**i+1Bucle de 1 hasta i

**fmientras** 4 OE’s

**si** (V[ i ]==key) **entonces** 1 acceso y 1 asignación

**devolver** i 2 accesos, 1 suma y 1 asignación

**sino** 1 acceso, 1 suma y 1 asignación

**devolver** -1

**fsi**

**ffuncion**

En el caso peor la clave no se encuentra y por lo tanto se recorre el vector entero y se devuelve -1.

→  *→ →*

Nuestro caso mejor sería si el primer elemento fuera igual que la clave por lo tanto:

*→ ∈O(1)*

En el caso peor se supone que se recorre el vector entero y por lo tanto:

→ *∈O*

Para el caso medio suponemos que el vector se recorre a la mitad:

→ *∈O*

*1.1.2. Algoritmo Búsqueda Binaria*

**funcion** BusquedaBinaria(V[1 … n]; primero, ultimo, clave:int)

**si** primero >= ultimo **entonces** Bucle de 2 hasta n

**devolver** V[ultimo]=clave; 1 acceso y 1 asignación

**fsi** 1 resta y 1 asignación

mitad 🡨 ((ultimo – primero + 1 ) / 2 ); Bucle mientras: 4 OE’s Condición

**si** clave = V[mitad] **entonces** 2 accesos, 1 suma y 1 asignación

**devolver** cierto; 1 resta y 1 asignación

**sino**

**si** clave < V[mitad] **entonces** 1 acceso, 1 suma y 1 asignación

**devolver** BúsquedaBinaria(V, primero, mitad-1, clave);

**sino**

**si** clave > V[mitad] **entonces**

**devolver** BusquedaBinaria (V, mitad+1, ultimo, clave);

**fsi**

**fsi**

**fsi**

**ffuncion**

Se ve claramente en el pseudocódigo que estaríamos hablando del caso mejor siempre y cuando al partir el vector en dos por primera vez, el valor mitad coincida con la clave.

Caso mejor ∈O(1)

→ // Recursión no homogénea

Resolvemos por cambio de variable

→ = 0 → r=1 //

deshacemos el cambio k=log n

Podemos observar que para el caso peor y medio nuestra complejidad pertenece al orden de log n y eso lo convierte en un algoritmo bastante eficiente.

*1.1.3. Algoritmo Búsqueda Ternaria*

**funcion** BusquedaTernaria (V [1 … n]; primero, ultimo, clave:int)

**si** primero >= ultimo **entonces Bucle de 1 hasta n**

**devolver** V[ultimo]=clave; 1 asignación

**fsi** Bucle de i+1 hasta n

tercio 🡨 ((ultimo – primero +1) / 3); 2 accesos y 1 comparación

**si** clave = V[primero+tercio] **entonces** 1 asignación

**devolver** cierto;

**sino**

**si** clave < V[primero+tercio] **entonces**

**devolver** BusquedaTernaria (V, primero, primero+tercio-1, clave);

**sino**

**si** clave = V[ultimo-tercio] **entonces**

**devolver** cierto;

**sino**

**si** clave < V[ultimo-tercio] **entonces**

**devolver** BúsquedaTernaria (V, primero+tercio+1, ultimo-tercio-1, clave);

**sino**

**devolver** BusquedaTernaria (V, ultimo-tercio+1, ultimo, clave);

**fsi**

**fsi**

**fsi**

**fsi**

**ffuncion**

Como podemos observar, el pseudocódigo de la búsqueda ternaria es muy similar al de la búsqueda binaria pero este divide al vector en tres partes en lugar de en dos mitades, de esta forma, el caso mejor se dará en la misma situación que en búsqueda binaria, cuando la clave coincida con alguno de los valores en la posición ⅓ o ⅔ , por lo tanto en el caso mejor ∈O(1).

Para el caso peor y medio se procede a resolver la recurrencia de la misma forma que en la búsqueda binaria:

→ Resolviendo por cambio de variable nuestra recurrencia no homogénea, llegamos a la misma conclusión que en búsqueda binaria ∈O(log n), la única diferencia es que en la búsqueda binaria la base del logaritmo es 2 y en nuestro algoritmo actual la base es 3, por lo tanto se supone algo más rápido.

*1.2. Tablas y Gráficas de coste teórico.*

En esta tabla podemos visualizar una evolución del número de operaciones elementales realizadas por cada algoritmo para su caso medio, todo esto son simplemente cálculos teóricos, pero a pesar de ello se puede observar claramente que algoritmo es el que realiza menos OE’s y por lo tanto cual es más rápido que los demás.

|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Secuencial | Binaria | Ternaria |
| 1000 | 3509 | 142 | 140 |
| 2000 | 7009 | 156 | 154 |
| 3000 | 10509 | 164 | 162 |
| 4000 | 14009 | 170 | 168 |
| 5000 | 17509 | 174 | 173 |
| 6000 | 21009 | 178 | 176 |
| 7000 | 24509 | 181 | 179 |
| 8000 | 28009 | 184 | 182 |
| 9000 | 31009 | 186 | 184 |
| 10000 | 35009 | 188 | 186 |

* 1. *Conclusiones*

Lo primero que notamos al hacer el estudio teórico de nuestros tres algoritmos es que el orden de complejidad de **Secuencial Iterativa** es diferente al de los otros dos algoritmos, cosa que se puede notar en la tabla de comparación teórica ya que podemos observar que el tiempo de **Secuencial Iterativa** es bastante mayor que el de los otros dos.

1. *Cálculo de los tiempos experimentales*

Al igual que en la práctica anterior, para realizar el cálculo experimental de los tiempos de ejecución de nuestros tres algoritmos de búsqueda hemos comparado las medias de los tiempos de ejecución de los algoritmos para diez tallas diferentes.

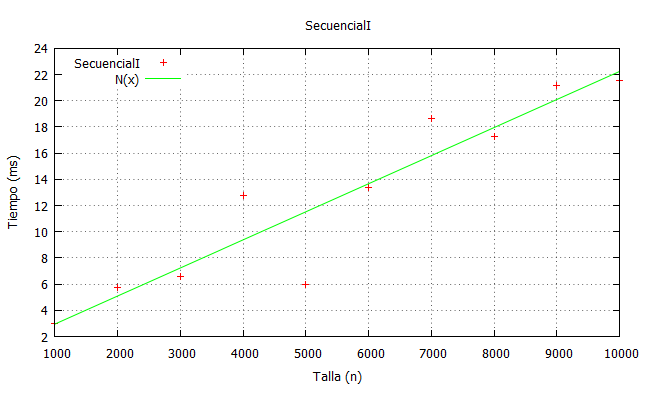
Hemos recurrido de nuevo a la clase ***Mtime*** para restar la hora a la que finaliza la ejecución y la hora a la que se inicia y de esta forma saber el tiempo que tarda en ejecutarse nuestros algoritmos.

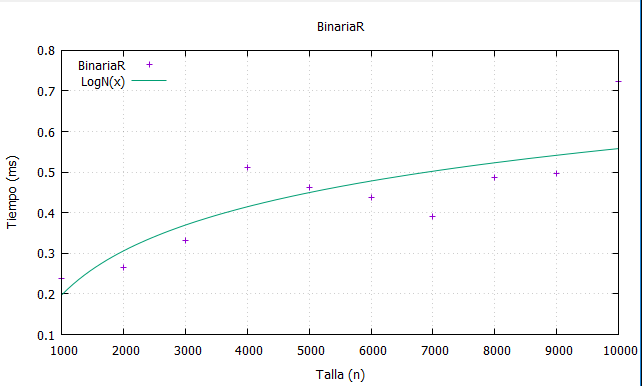
* 1. *Tablas y Gráficas de coste experimental.*

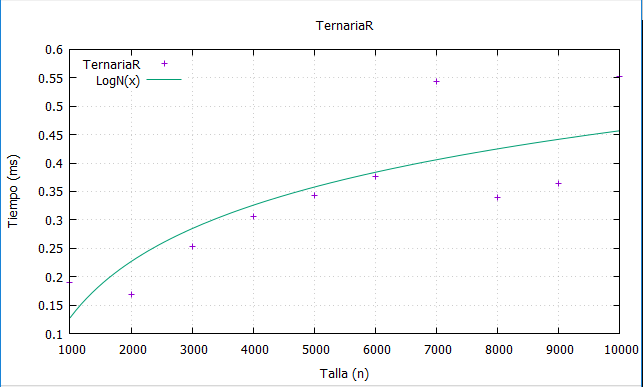
Esta tabla muestra el resultado del análisis experimental del tiempo de ejecución de los tres algoritmos de ordenación.

|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Secuencial | Binaria | Ternaria |
| 1000 | 3.008 | 0.2816 | 0.422401 |
| 2000 | 5.72161 | 0.524801 | 0.405334 |
| 3000 | 6.54508 | 0.435201 | 0.524801 |
| 4000 | 12.7403 | 0.588801 | 0.644267 |
| 5000 | 5.97334 | 0.785068 | 0.652801 |
| 6000 | 13.3504 | 0.652801 | 0.678401 |
| 7000 | 18.6283 | 0.814934 | 0.746668 |
| 8000 | 17.2544 | 0.768001 | 0.686934 |
| 9000 | 21.1371 | 0.857601 | 0.797868 |
| 10000 | 21.5766 | 0.955735 | 0.695468 |

Aquí podemos observar la gráfica de cada caso medio.

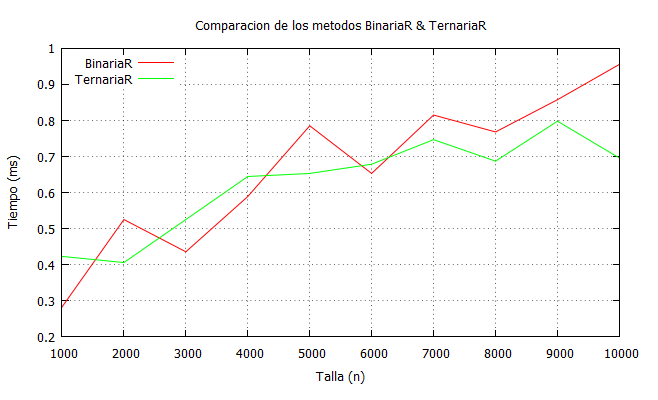
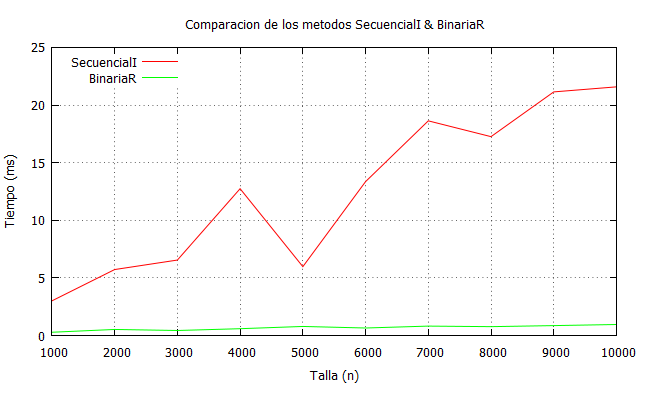
**

**

**

*2.2. Comparación entre métodos*

A continuación se muestran las gráficas de comparación entre métodos para que podamos ver claramente que algoritmos es más rápido.



*2.3. Conclusiones.*

Podemos observar que los resultados experimentales coinciden con el análisis teórico, los métodos **BinariaRecursiva** y **TernariaRecursiva** superan con mucha diferencia al método de búsqueda **SecuencialIterativa**, otra curiosidad es que podemos ver como **TernariaRecursiva** es algo más rápido que **BinariaRecursiva**, aunque la diferencia es bastante inapreciable a tallas muy pequeñas.

Todo esto hay que tenerlo en cuenta junto a que las dos búsquedas recursivas necesitan manejar un vector ordenado, por lo tanto a la hora de calcular el tiempo total que vamos a tardar en buscar una clave aleatoria en un vector desordenado hay que poner sobre la mesa el tiempo que tardaremos en ordenar el array, sin embargo, con un buen método de ordenación como **QuickSort** o **ShellSort** podríamos montar una búsqueda más rápida que **SecuencialIterativa** a un vector desordenado.

1. *Comparación de los resultados teórico y experimental.*

Como hemos podido comprobar, nuestros resultados teóricos y experimentales más o menos coinciden, tenemos que tener en cuenta que los resultados teóricos siempre son ideales mientras que los experimentales dependen de muchos factores como nuestro procesador por ejemplo.

También podemos ver que en los resultados experimentales son muy inestables, esto es sobre todo debido a que realizar una búsqueda en un vector es un proceso con una variación de tiempo muy grande, por ello muchos resultados parece que pertenecen a otras tallas o se van de tiempo.

Otra cosa que podemos destacar es al observar las gráficas, vemos la recta que por así decirlo, representa la situación ideal, podemos apreciar perfectamente que nuestra “**nube de puntos**” formada por nuestros distintos resultados experimentales se asemeja bastante a la forma descrita por esta “situación ideal”.

1. *Diseño de la aplicación.*

Nuestra aplicación consiste en una mejora de la práctica anterior. Lo único que hemos tenido que hacer es un menú adicional para nuestros algoritmos de búsqueda (a parte de añadir el **header**, el **.cpp** y configurar la clase “**graficas**” para nuestros nuevos órdenes de los algoritmos de búsqueda).

Por lo demás, el submenú para nuestros algoritmos de búsqueda es exactamente igual que el que usamos para los algoritmos de ordenación, es decir, un apartado para probar nuestros algoritmos, otro para el caso medio y otro para comparar dos métodos.

1. *Conclusiones y valoraciones personales de la práctica.*

Nos ha parecido una práctica muy útil para aclarar los contenidos

explicados en clase de teoría, puesto que hemos realizado un estudio

totalmente empírico de los métodos de búsqueda.

Cabe destacar la utilidad de dicha aplicación, ya que nos sirve tanto para estudiar el funcionamiento de los diferentes métodos de

búsqueda como los de ordenación (desarrollados en la práctica anterior), puesto que tenemos la opción en nuestro menú principal de

estudiar un tipo de algoritmo u otro.

En definitiva, esta práctica nos ha servido para aclarar los contenidos

estudiados de forma teórica.