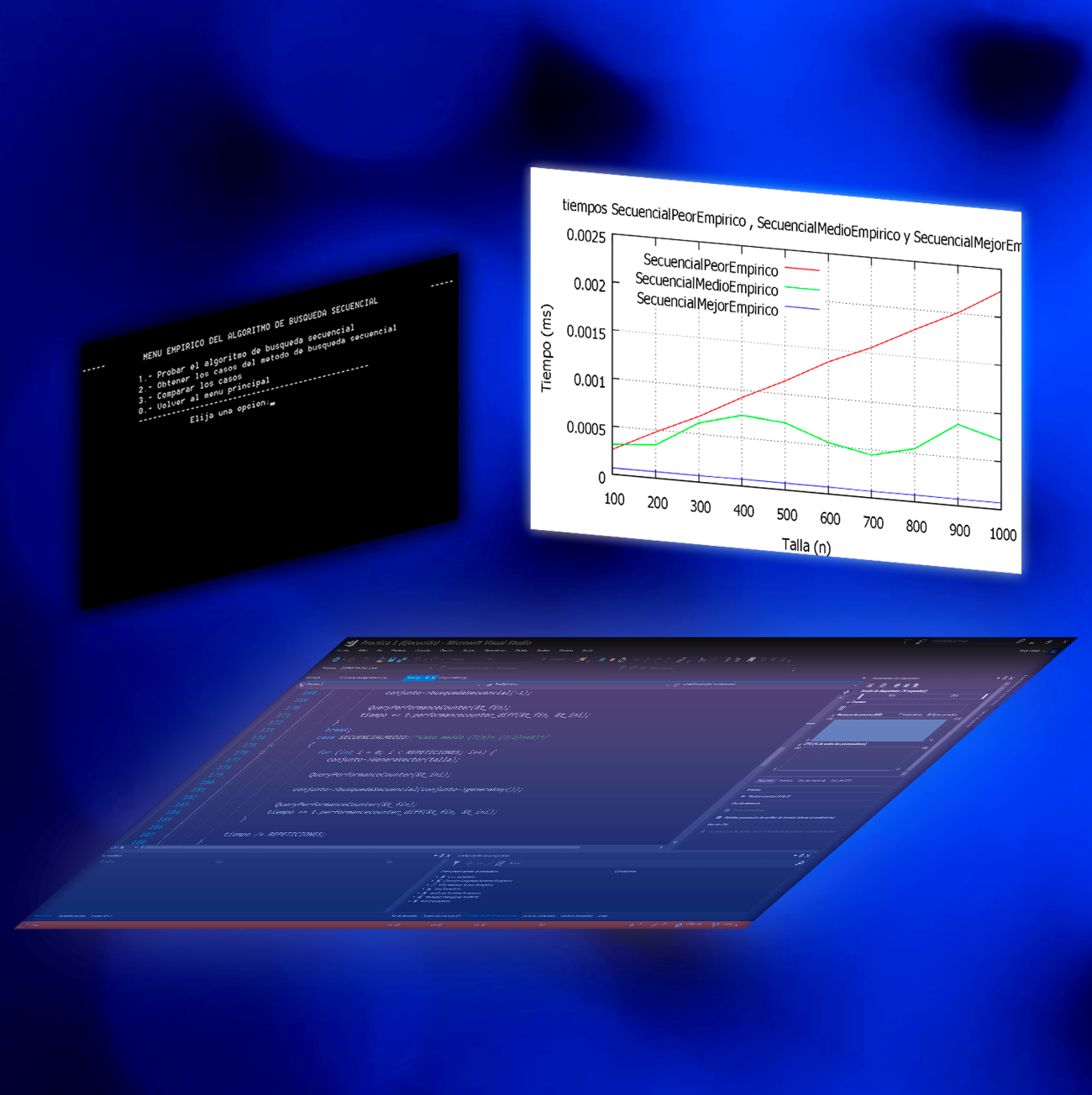
Análisis de un algoritmo de búsqueda secuencial.



Autor: Borja López Pineda

# Índice

# P0. [Portada](#_top)

# P1. [Índice](#_Índice)

# P2. [Introducción](#_Introducción)

# P2. [Cálculo de tiempo teórico](#_Cálculo_de_tiempo)

## [Código y análisis de coste](#_Código_y_análisis)

## [Tabla y gráficas de coste](#_Tabla_y_gráficas)

## [Conclusiones](#_Conclusiones)

# P4. [Cálculo del tiempo experimental](#_Cálculo_del_tiempo)

## [Tablas y gráficas de coste](#_Tablas_y_gráficas)

## [Conclusiones](#_Conclusiones_1)

# P5. [Comparación de los resultados teórico y experimental](#_Comparación_de_los)

# P5. [Diseño de la aplicación](#_Diseño_de_la)

# P6. [Conclusiones y valoraciones personales de la práctica](#_Conclusiones_y_valoraciones)

# Introducción

El objetivo de esta práctica es realizar el estudio teórico de un algoritmo de búsqueda secuencial, posteriormente implementarlo en un lenguaje de programación y medirlo experimentalmente.

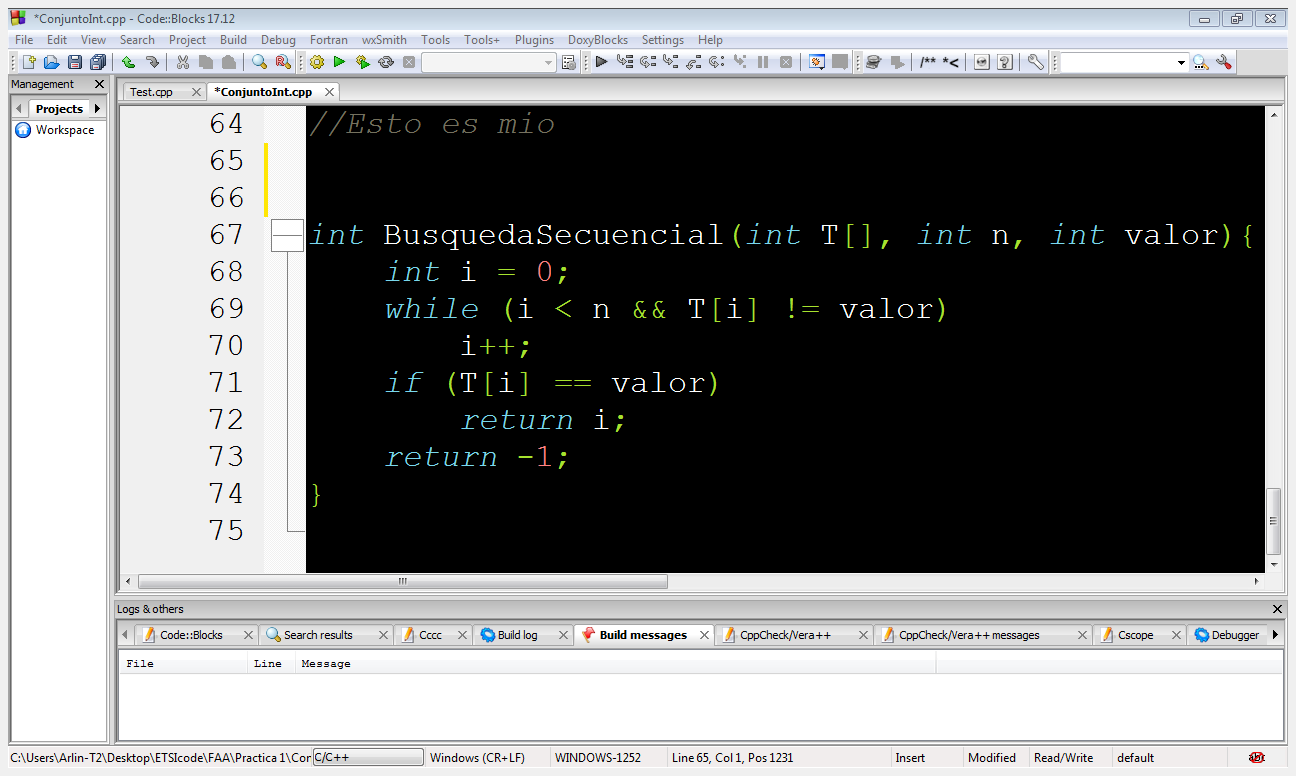
Para esta ocasión se ha escogido el lenguaje C++ y el entorno de desarrollo integrado Visual Studio 2017. Adicionalmente hemos utilizado GNUplot para graficar los resultados.

# Cálculo de tiempo teórico

El algoritmo propuesto fue analizado según los criterios aprendidos en las clases teóricas.

## Código y análisis de coste

El algoritmo se dio directamente implementado en C++.  
Sus parámetros formales son: un array de enteros (T) que contiene un conjunto de valores, un entero (n) que indica cuantos elementos hay en la tabla y un entero (valor) que será la clave de búsqueda.  
Esta función devuelve la posición dentro del array de la clave de búsqueda; en caso de haber varias, se queda con la primera ocurrencia. Si la clave no se encuentra, devuelve -1 (*escogido por ser un valor que nunca devolvería cómo resultado correcto*).

Primero inicia un contador (i), después itera hasta que se acabe el array o encuentre la clave.  
Finalmente comprueba si salió del bucle con i siendo el índice correcto, en ese caso devuelve i, en caso contrario devuelve -1.

*Aunque esta implementación cumple con la especificación solicitada; en caso de no encontrar la clave, se accede a una posición fuera del array.*

Conteo de operaciones elementales: Una operación de asignación para inicializar el contador. La condición del bucle tiene 4 operaciones, el cuerpo del bucle 2 y una por el salto de línea, la condición se ejecuta una vez más que el cuerpo. En total son 7 operaciones por repetición y otras 4 de la salida. El *If* tiene 2 operaciones en la operación y una del salto. Independientemente del resultado del *If*, se ejecuta una operación del *return*.

Haciendo recuento; el coste temporal de este algoritmo es de 7 operaciones por repetición y 9 más fijas. Si la clave no está en la tabla, al bucle se repite tantas veces como elementos tenga la tabla. Si la clave está al principio, no se entra en el bucle, el coste es siempre el mismo. Realizando la media, en la mitad de los casos la clave se encuentra en la mitad de la tabla.

Mediante el análisis del algoritmo, se obtuvieron las expresiones que determinan el número de operaciones elementales necesarias en los distintos casos.

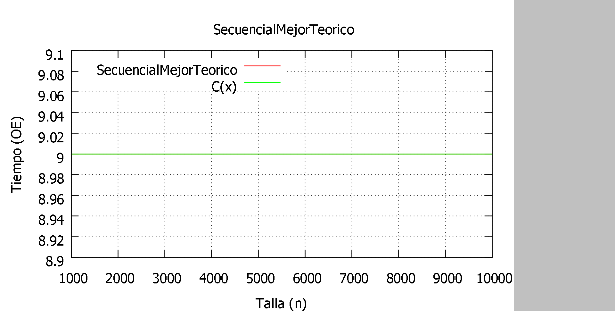
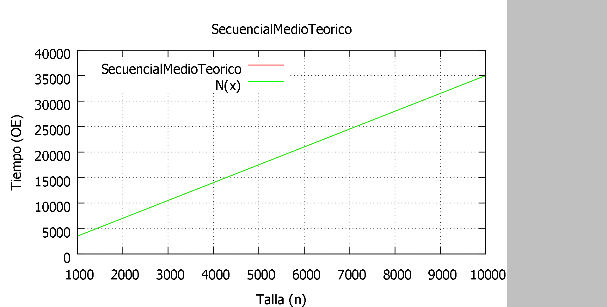
* Caso peor: La clave no está en la tabla
* Caso medio: La clave está en el centro de la tabla.
* Caso mejor: La clave es el primer elemento.

## Tabla y gráficas de coste

|  |  |
| --- | --- |
| Talla | OE’s |
| 1000 | 7009 |
| 2000 | 14009 |
| 3000 | 21009 |
| 4000 | 28009 |
| 5000 | 35009 |
| 6000 | 42009 |
| 7000 | 49009 |
| 8000 | 56009 |
| 9000 | 63009 |
| 10000 | 70009 |
| *Caso Peor* | |

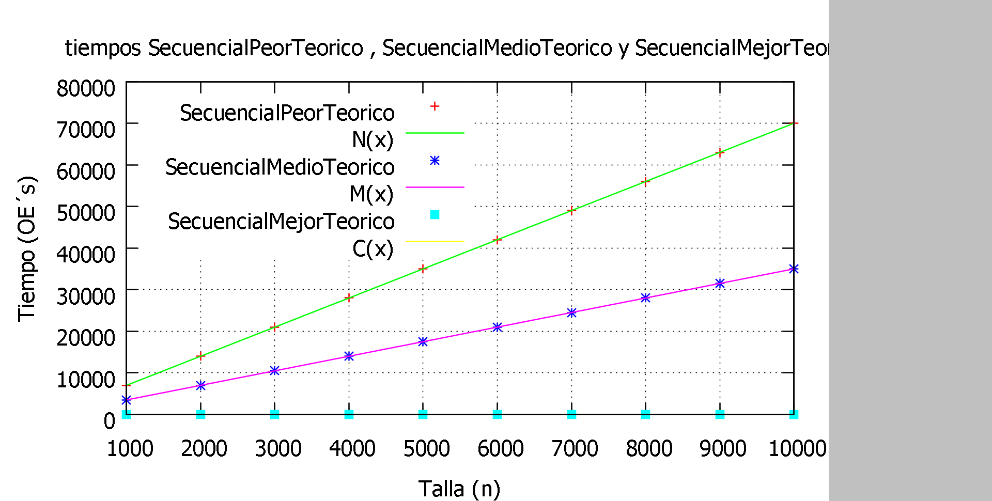
|  |  |
| --- | --- |
| Talla | OE’s |
| 1000 | 3509 |
| 2000 | 7009 |
| 3000 | 10509 |
| 4000 | 14009 |
| 5000 | 17509 |
| 6000 | 21009 |
| 7000 | 24509 |
| 8000 | 28009 |
| 9000 | 31509 |
| 10000 | 35009 |
| *Caso medio* | |

|  |  |
| --- | --- |
| Talla | OE’s |
| 1000 | 9 |
| 2000 | 9 |
| 3000 | 9 |
| 4000 | 9 |
| 5000 | 9 |
| 6000 | 9 |
| 7000 | 9 |
| 8000 | 9 |
| 9000 | 9 |
| 10000 | 9 |
| *Caso Mejor* | |



## 

|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Peor | OE’s  Medio | Mejor |
| 1000 | 7009 | 3509 | 9 |
| 2000 | 14009 | 7009 | 9 |
| 3000 | 21009 | 10509 | 9 |
| 4000 | 28009 | 14009 | 9 |
| 5000 | 35009 | 17509 | 9 |
| 6000 | 42009 | 21009 | 9 |
| 7000 | 49009 | 24509 | 9 |
| 8000 | 56009 | 28009 | 9 |
| 9000 | 63009 | 31509 | 9 |
| 10000 | 70009 | 35009 | 9 |
| *Comparativa* | | | |



## Conclusiones

Como se esperaba, el coste temporal del algoritmo depende de la talla del problema y de la instancia en concreto. Búsquedas con la misma talla tienen costes diferentes, dependiendo de dónde se encuentre la clave.

Aún en el peor caso, este algoritmo sigue un modelo de crecimiento lineal. Este crecimiento puede verse representado como rectas en la gráfica. La función complejidad es lineal.

# Cálculo del tiempo experimental

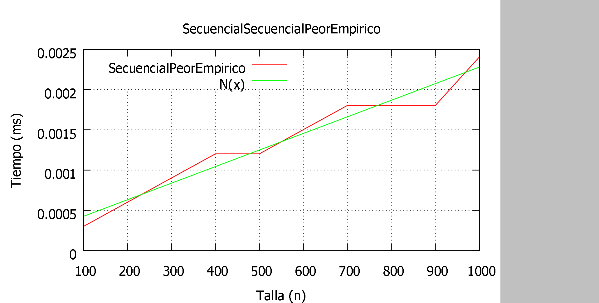
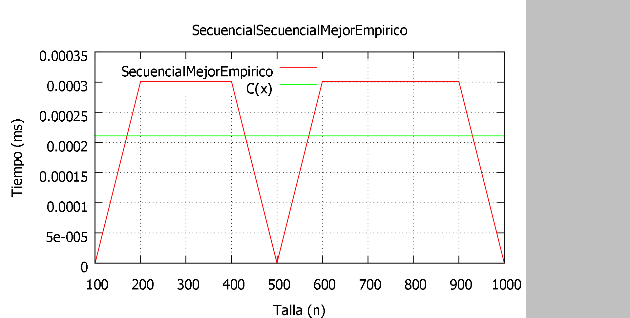
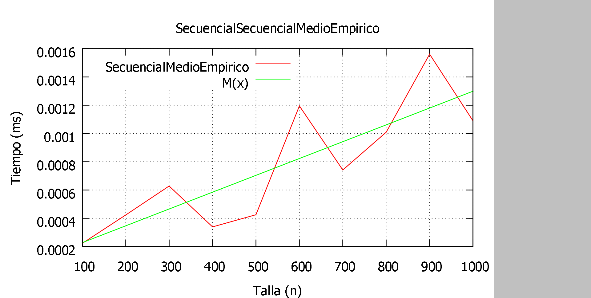
Para cada medición se generan nuevos conjuntos de datos aleatorios, pero solo se mide el tiempo de búsqueda. En el caso medio este proceso se repite múltiples veces y se toma la media.

## Tablas y gráficas de coste

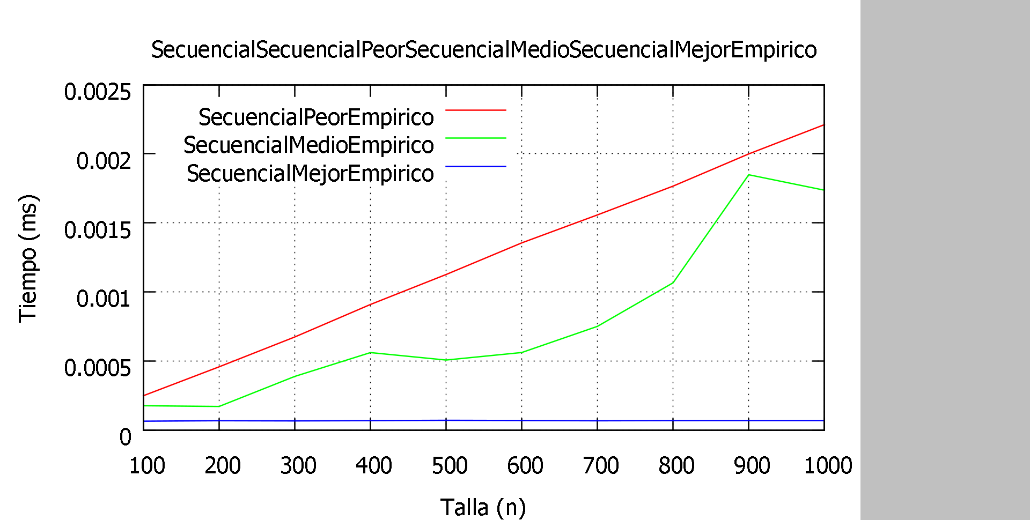
|  |  |
| --- | --- |
| Talla | ms |
| 1000 | 0.0003 |
| 2000 | 0.0006 |
| 3000 | 0.0009 |
| 4000 | 0.0012 |
| 5000 | 0.0012 |
| 6000 | 0.0015 |
| 7000 | 0.0018 |
| 8000 | 0.0018 |
| 9000 | 0.0018 |
| 10000 | 0.0024 |
| *Caso Peor* | |

|  |  |
| --- | --- |
| Talla | ms |
| 1000 | 0.0002 |
| 2000 | 0.0004 |
| 3000 | 0.0006 |
| 4000 | 0.0003 |
| 5000 | 0.0004 |
| 6000 | 0.0011 |
| 7000 | 0.0007 |
| 8000 | 0.0010 |
| 9000 | 0.0015 |
| 10000 | 0.0010 |
| *Caso medio* | |

|  |  |
| --- | --- |
| Talla | ms |
| 1000 | 0.00000 |
| 2000 | 0.00003 |
| 3000 | 0.00003 |
| 4000 | 0.00003 |
| 5000 | 0.00000 |
| 6000 | 0.00003 |
| 7000 | 0.00003 |
| 8000 | 0.00003 |
| 9000 | 0.00003 |
| 10000 | 0.00000 |
| *Caso Mejor* | |



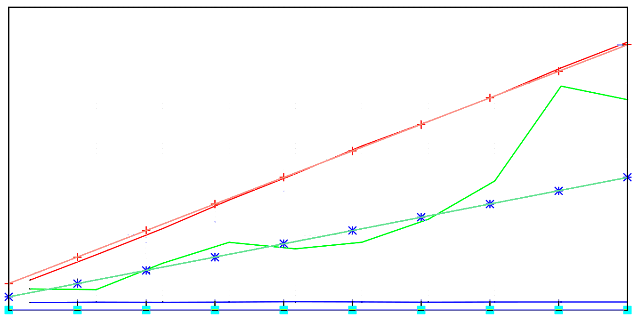
|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Peor | ms  Medio | Mejor |
| 1000 | 0.0002 | 0.0001 | 0.00006 |
| 2000 | 0.0004 | 0.0001 | 0.00006 |
| 3000 | 0.0006 | 0.0003 | 0.00006 |
| 4000 | 0.0009 | 0.0005 | 0.00006 |
| 5000 | 0.0011 | 0.0005 | 0.00007 |
| 6000 | 0.0013 | 0.0005 | 0.00007 |
| 7000 | 0.0015 | 0.0007 | 0.00006 |
| 8000 | 0.0017 | 0.0010 | 0.00006 |
| 9000 | 0.0019 | 0.0018 | 0.00006 |
| 10000 | 0.0022 | 0.0017 | 0.00006 |
| *Comparativa* | | | |



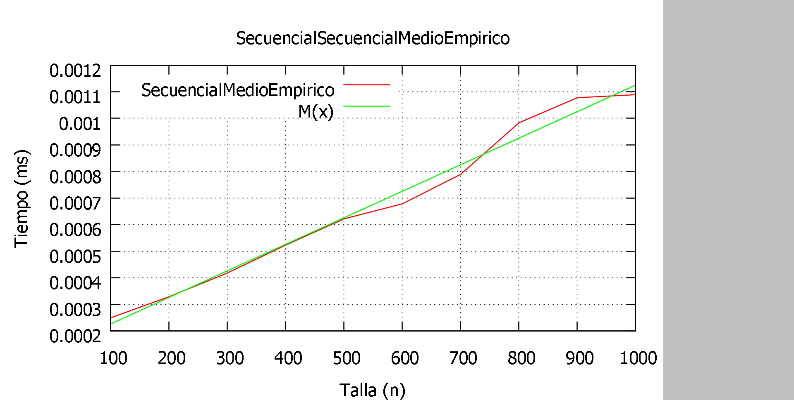
## Conclusiones

Los resultados individuales muestran un grado tolerable de dispersión, sin elementos anómalos. Siendo esta dispersión más apreciable en el caso medio, donde hacen falta repetir la medición varios ordenes de magnitud para datos más próximos a la regresión lineal. En el caso mejor por separado, aparenta una gran dispersión; pero, teniendo en cuenta la escapa, la diferencia es mínima y se debe al límite de precisión con la que se mide el tiempo transcurrido.

# Comparación de los resultados teórico y experimental

La comparativa practica presenta la misma relación que la comparativa teórica, luego el modelo de coste analizado se ajusta correctamente a los resultados empíricos.

Como puede observarse al contrastar la comparativa teórica con la experimental; las relaciones entre los casos se mantienen.

El modelo de coste temporal analizado describe correctamente al algoritmo, puesto las formulas se ajustan a los resultados obtenidos.

En el caso medio, puede observarse cierta tendencia a la dispersión cuando aumenta la talla. Para estar más seguros, repetí la medición con 106 repeticiones, en vez de las anteriores 104.

# Diseño de la aplicación

La metodología de programación escogida para la implementación fue diseño modular con orientación a objetos.

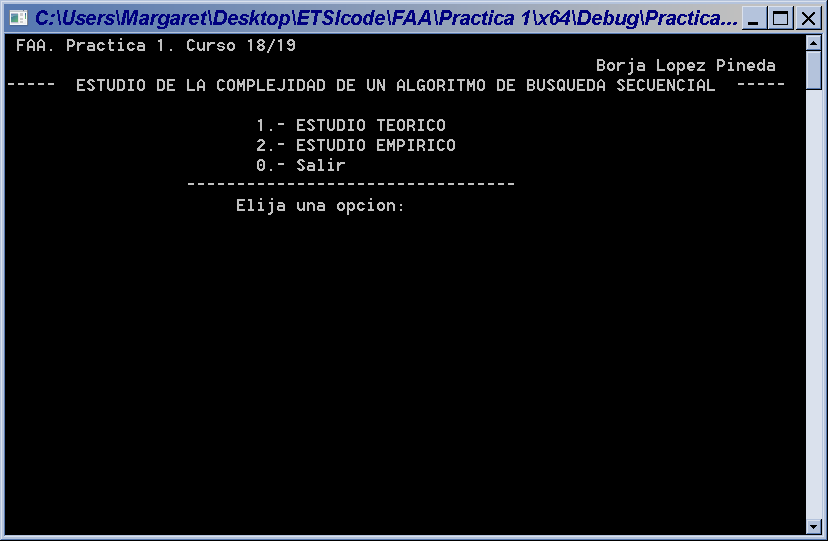
El método principal junto a la interfaz de menús, está en ComparacionAlgoritmos. Desde allí se llama a los métodos de Test, la clase que realiza los cálculos. Test, a su vez, se apoya en la clase ConjuntoInt para la ejecución del algoritmo a estudiar, y Mtime para medir el tiempo transcurrido.

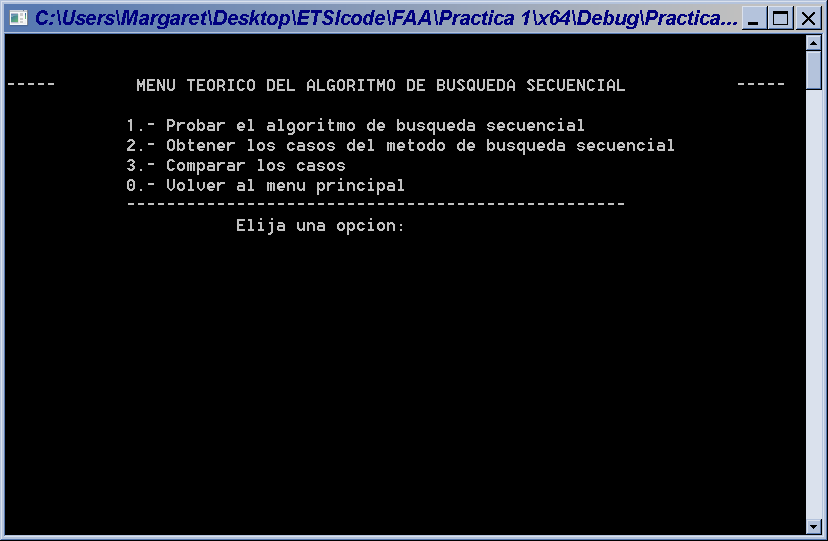
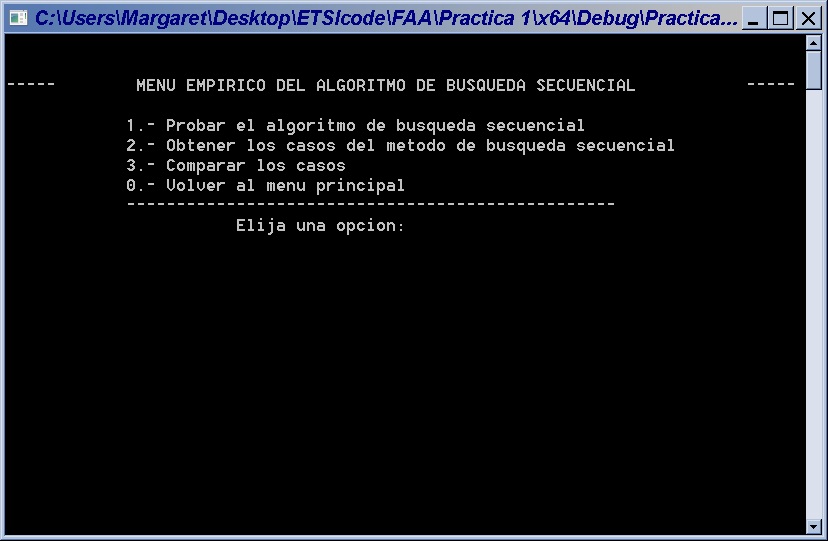
ConjuntoInt permite crear tablas de enteros aleatorios y, posteriormente, buscar la primera ocurrencia de una clave especificada. Adicionalmente, puede generar claves de búsqueda.

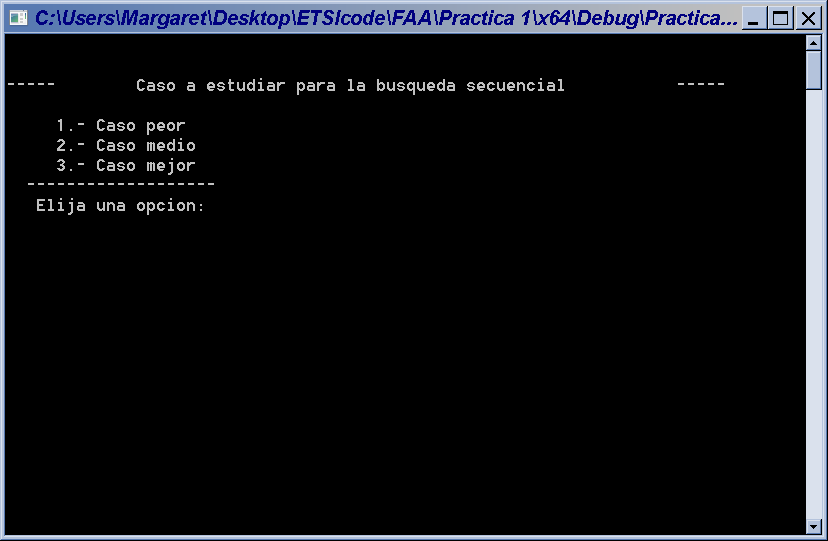
Test tiene los siguientes métodos principales:

* costeTeorico: Recibe por parámetro el caso a estudiar de forma teórica. Muestra por pantalla y guarda en un archivo los valores que toma la función complejidad para distintas tallas y casos.
* compararTeorico: Recibe por parámetro los tres casos a comparar teóricamente. Muestra por pantalla y guarda en un archivo los valores de la función complejidad para todos los casos y diferentes tallas.
* costeEmpirico: Recibe por parámetro el caso a simular empíricamente. Genera un conjunto de datos aleatorios, y mide el tiempo que transcurre entre el inicio y el fin de la llamada al algoritmo de búsqueda. Repite el proceso para distintas tallas, muestra los resultados por pantalla y los guarda en un fichero.
  + En el caso peor, se le introduce la clave de búsqueda -1, valor que no existe en el conjunto de datos.
  + En el caso medio, se escoge una clave aleatoria de entre los elementos contenidos en el conjunto. Las mediciones se repiten múltiples veces y se toma la media.
  + En el caso mejor, se pasa el primer elemento del conjunto como clave.
* compararEmpirico: Recibe por parámetro los tres casos a simular empíricamente. Realiza las mismas mediciones de los tres casos, las muestra por pantalla y las guarda en un archivo.

La interfaz consta de un menú principal y dos submenús, para el estudio teórico y empírico. En ambos submenús se puede probar el algoritmo de búsqueda, generar resultados para un caso en particular, o comparar los resultados de los tres casos.







# Conclusiones y valoraciones personales de la práctica

Esta practica ha servido para visualizar la utilidad del análisis de algoritmos, al confirmarse una clara relación entre los valores calculados y las lecturas obtenidas.

También se ha practicado la escritura de ficheros y el uso del diseño modular.