Análisis de algoritmos de búsqueda.



# 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)

## [Secuencial](#_Código_y_análisis) [Binaria](#_Código_y_análisis_1) [Ternaria](#_Código_y_análisis_2)

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

## [Conclusiones](#_Conclusiones)

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

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

## [Conclusiones](#_Conclusiones)

# 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

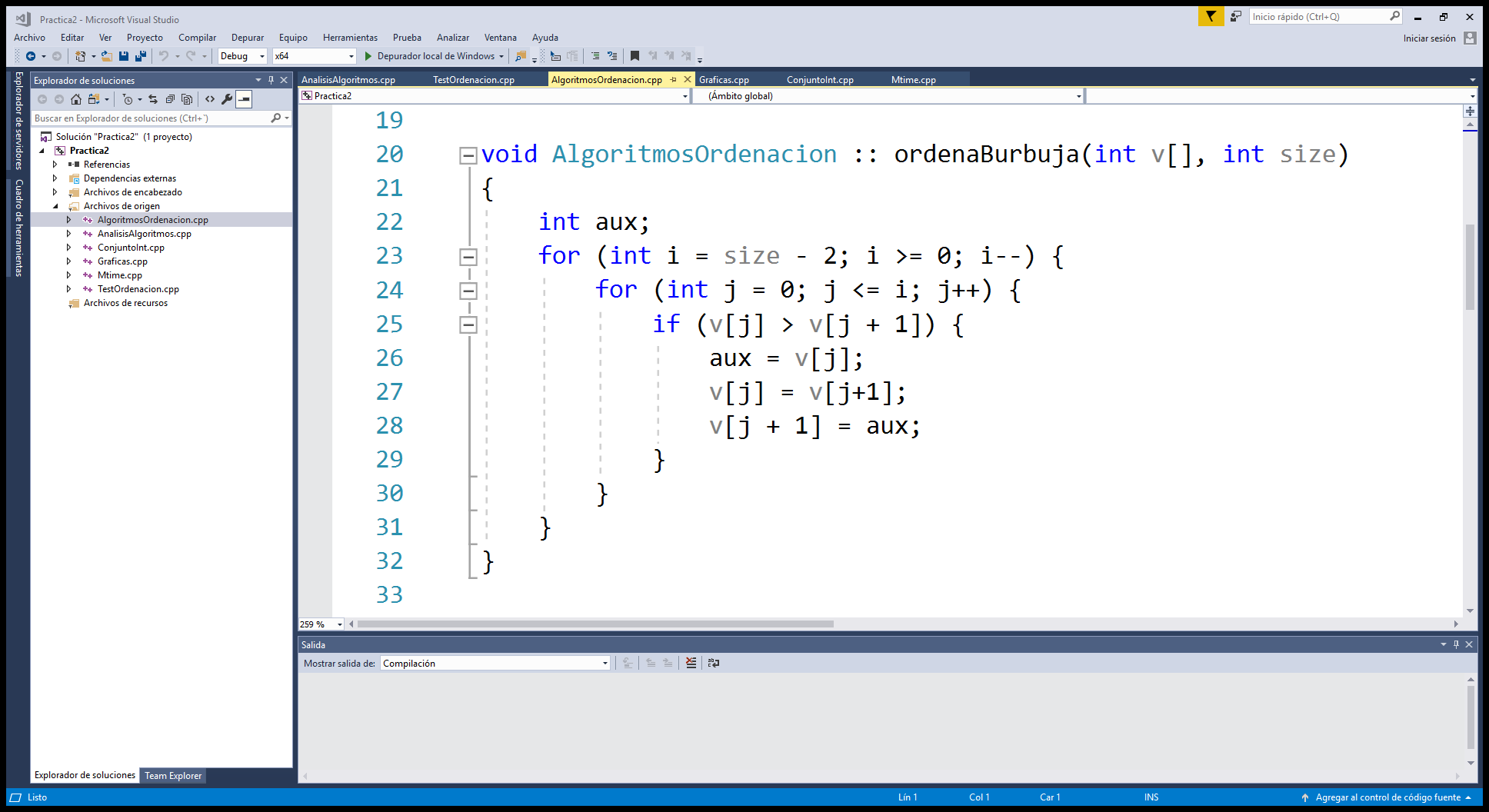
El objetivo de esta práctica es realizar el estudio teórico de múltiples algoritmos de búsqueda, posteriormente implementarlos en un lenguaje de programación y comparar su eficiencia empíricamente.

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

Fueron propuestos en pseudocódigo e implementados por el alumno, en tres funciones. Todas ellas reciben un vector de enteros, el número de elementos que contiene y la clave que deben buscar. Devuelven la posición del elemento en el vector; si no lo encontrase, devolvería -1.

## Código y análisis de coste del algoritmo de búsqueda lineal

Este algoritmo tiene un funcionamiento muy sencillo. Realiza una lectura secuencial de los elementos hasta encontrar la clave.

Cuando termina la búsqueda, tiene dos opciones. Si el elemento se encuentra en el vector, la búsqueda habrá acabado antes de llegar al final y la posición del marcador es la ubicación de la primera ocurrencia de la clave. Si no se encontrase, la lectura habría llegado al final, el marcador estaría fuera del recorrido; para informar de que no lo ha encontrado devuelve -1, un valor que no sería posible de otro modo.

De los tres, es el algoritmo más sencillo y más lento.

Eficiencia teórica:

* Caso mejor: La clave se encuentra en el primer elemento. Independientemente de la talla, siempre al encontrará la primera vez. El orden es O(1)
* Caso peor: La clave no se encuentra en el array. Tiene que recorrer todo el vector una vez. El orden es O(n).
* Caso medio: La clave está en posiciones intermedias en el vector. El orden sigue siendo O(n), aunque el conteo de operaciones sale menor.

## Código y análisis de coste del algoritmo de búsqueda binaria

El algoritmo de búsqueda binaria funciona acotando la posición de la clave y sabiendo que el vector ya está ordenado. La siguiente implementación es recursiva.

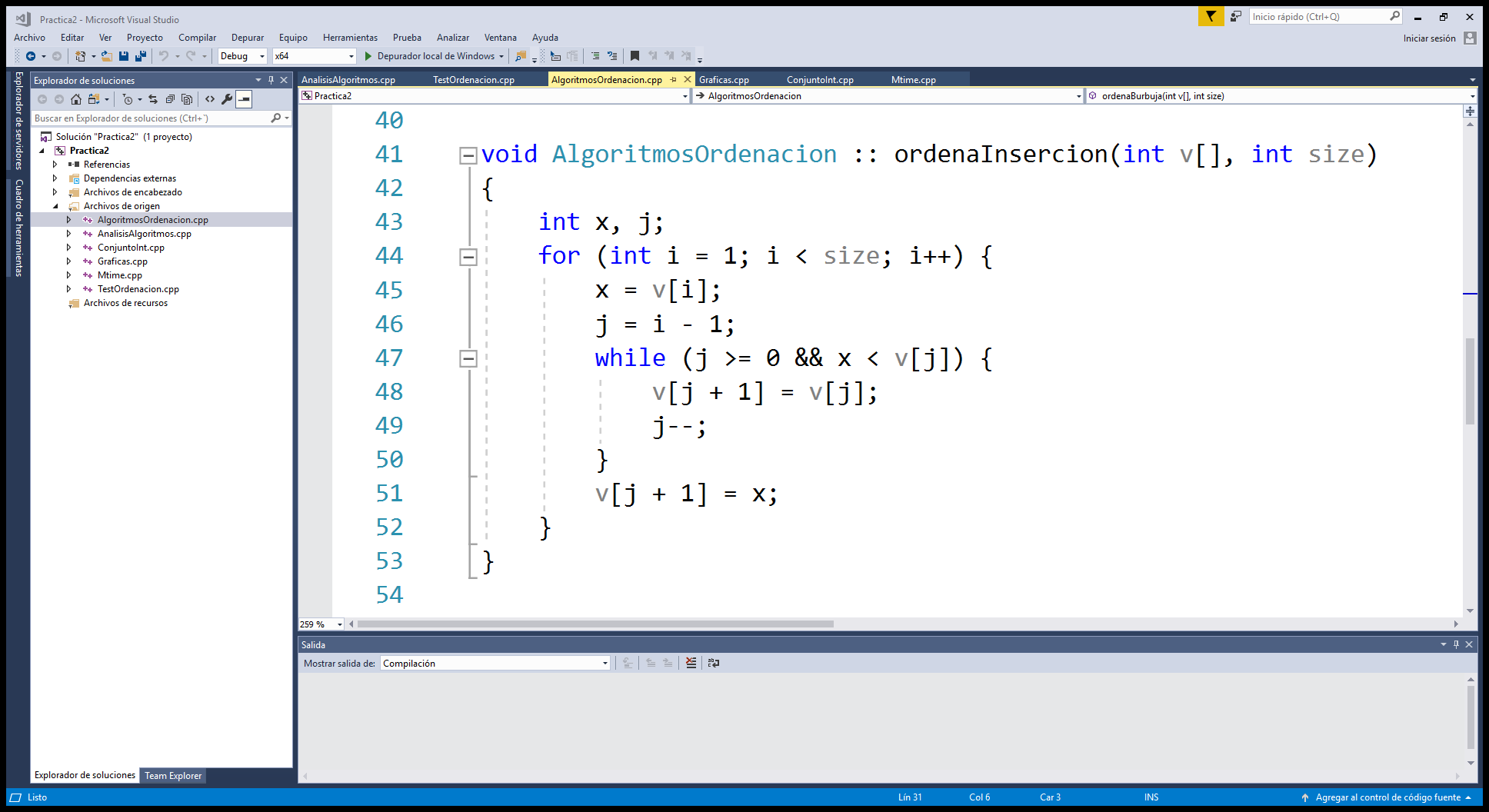
Calcula la posición central del vector. Si la clave se encuentra ahí, la ha encontrado y devuelve la posición. Si no la ha encontrado y el elemento en esa posición es mayor que la clave, sabe que debe estar antes y manda a buscar la clave en el primer subconjunto. Si el elemento central fuera menor, manda a buscar la clave en el segundo subconjunto, que es donde debe estar.

Resulta más eficiente que búsqueda binaria puesto que solo realiza menos comparaciones. Aunque necesita que el vector esté ordenado.

Eficiencia teórica:

* Caso mejor: La clave se encuentra en el centro del vector, la encuentra a la primera llamada.
* Caso peor: La clave no se encuentra en el vector. Tiene que realizar divisiones hasta quedarse con un vector de un elemento.

## Código y análisis de coste del algoritmo de búsqueda ternaria

Al igual que la búsqueda binaria, este algoritmo acota progresivamente el vector hasta encontrar la clave. También está implementado de forma recursiva.

A diferencia de la búsqueda binaria, divide el vector en tres subconjuntos.  
Comprueba que la clave no esté en ninguna de las dos divisiones que ha realizado y llama a buscar la clave en el subconjunto dónde debe encontrarse. Comenzando por ver si la clave está en la primera división y si no, comprobando si está por encima o por debajo; si está por debajo, sabe que debe estar en el primer subconjunto; si está por arriba, tiene que comprobar que no esté en la segunda división e, igualmente, comprobar si se encuentra antes o después de esta división.

Igualmente, supone una mejora significativa respecto a la búsqueda lineal, pero requiere que el vector esté ordenado.

Eficiencia teórica:

* Caso mejor: Si la clave se encuentra en alguna de las dos divisiones, lo encuentra en la primera llamada.
* Caso peor: Si la clave no está en el vector, tiene que realizar divisiones hasta quedarse con un subconjunto de un elemento.
* Caso medio: La clave se encuentra en el centro del vector. La duración de la búsqueda es una media entre el caso peor y el caso medio.

## Tabla y graficas de coste

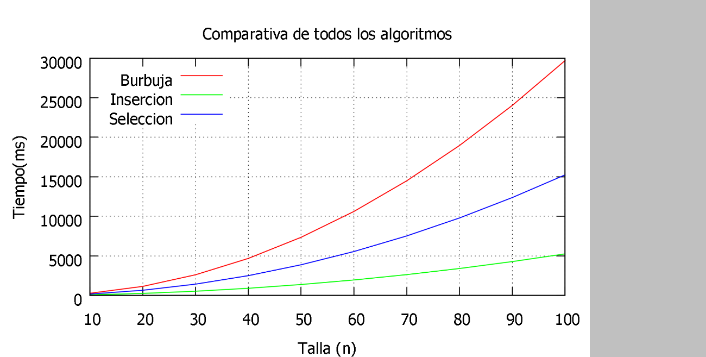
|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Tiempo (ms) | | |
| **Lineal** | **Binaria** | **Ternaria** |
| 1000 | 50 | 5.48 | 7.28 |
| 2000 | 100 | 5.98 | 7.91 |
| 3000 | 150 | 6.27 | 8.28 |
| 4000 | 200 | 6.48 | 8.54 |
| 5000 | 250 | 6.64 | 6.65 |
| 6000 | 300 | 3.45 | 1950 |
| 7000 | 350 | 3.56 | 2625 |
| 8000 | 400 | 3.66 | 3400 |
| 9000 | 450 | 3.74 | 4275 |
| 10000 | 500 | 3.82 | 5250 |

Para obtener el caso medio se tomó el conteo de operaciones críticas.

Las búsquedas binaria y ternaria muestran un comportamiento similar y se diferencian claramente de la búsqueda lineal.

## Conclusiones

Como se esperaba, el coste temporal del algoritmo depende de la talla del problema y tanto de la instancia. Los tres algoritmos lentos estudiando son de orden polimétrico cuadrado, no obstante, algunos realizan menos instrucciones que otros. Quicksort en el caso medio crece más lentamente

Por orden de eficiencia: Quicksort << Burbuja < Selección < Inserción

En cuanto a complejidad espacial, todos los algoritmos son , la cantidad de memoria consumida por el algoritmo no es proporcional al numero de elementos a ordenar.  
En general, todos utilizan una variable temporal para los intercambios.

# Cálculo del tiempo experimental

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

## Tablas y gráficas de coste

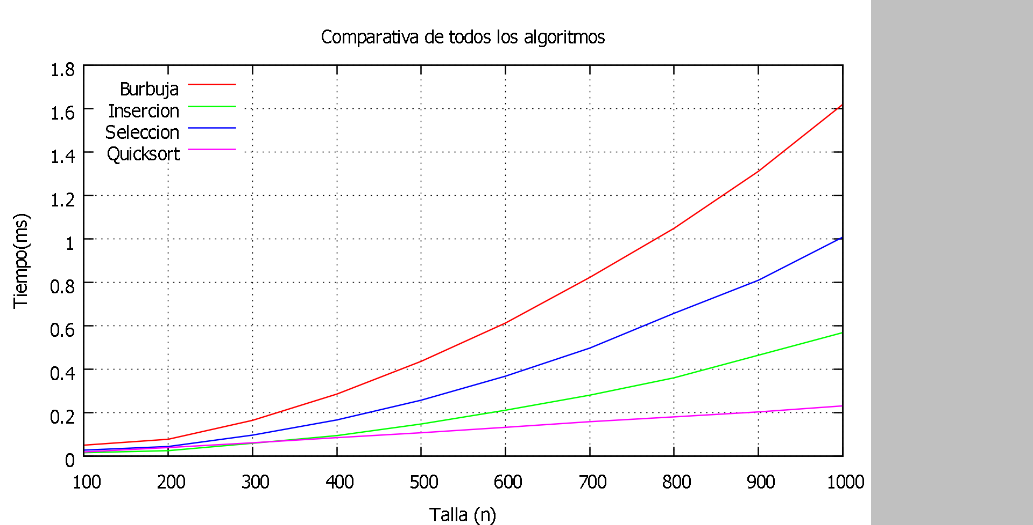
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Talla | Tiempo (ms) | | |  |
| **Burbuja** | **Selección** | **Intersección** | **Quicksort** |
| 100 | 0.0240 | 0.0140 | 0.0074 | 0. 0190 |
| 200 | 0.0880 | 0.0471 | 0.0290 | 0. 0393 |
| 300 | 0.1607 | 0.0916 | 0.0546 | 0. 0610 |
| 400 | 0.2646 | 0.1530 | 0.0931 | 0. 0849 |
| 500 | 0.3930 | 0.2459 | 0.1514 | 0. 1075 |
| 600 | 0.5663 | 0.3350 | 0.1977 | 0. 1323 |
| 700 | 0.7546 | 0.4617 | 0.2842 | 0. 1584 |
| 800 | 0.9703 | 0.5953 | 0.3677 | 0. 1806 |
| 900 | 1.2093 | 0.7388 | 0.4438 | 0. 2033 |
| 1000 | 1.4974 | 0.9245 | 0.5368 | 0. 2311 |

## 

## 

## Conclusiones

Los resultados individuales muestran un grado tolerable de dispersión, sin elementos anómalos. Cabe apreciar una mejor aproximación que en los algoritmos de complejidad lineal.

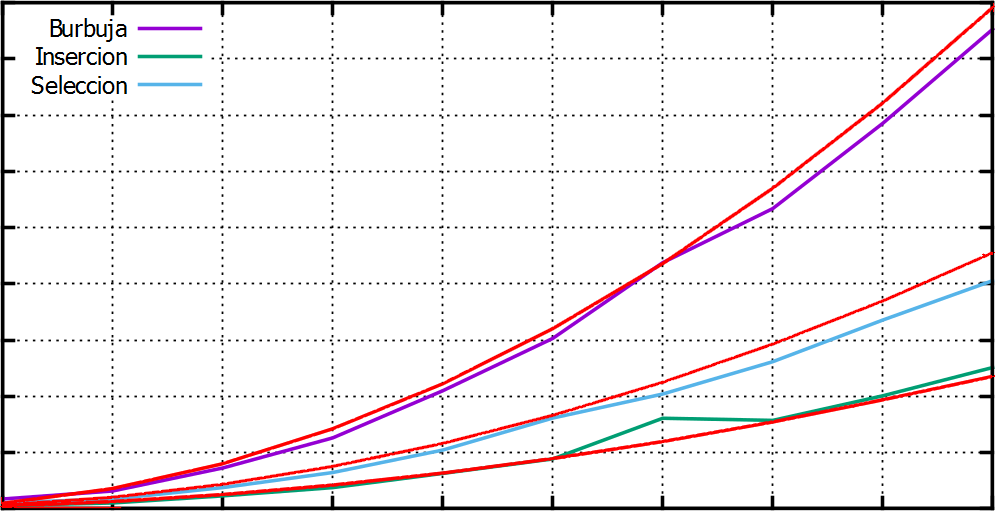
Puede apreciarse la rápida tendencia a crecer de las funciones cuadráticas. Seguir aumentando la talla, requeriría un tiempo de ejecución mucho mayor; frente al bajo crecimiento de Quicksort.

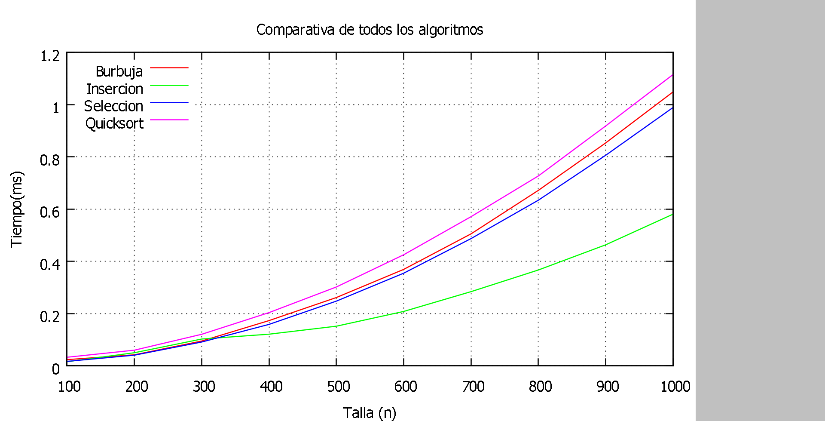
Pese a esto, se ha observado que Quicksort resulta más lento que Inserción y Selección en tallas pequeñas y que su eficiencia se vuelve presente a partir de la mitad de la prueba, donde toma clara ventaja.

# Comparación de los resultados teórico y experimental

Los resultados experimentales reflejan la misma relación entre los algoritmos. Siendo Inserción el método más rápido, burbuja el más lento.

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

En este gráfico puede visualizarse la relación entre los resultados experimentales y el cálculo teórico (líneas rojas). Para generar las curvas se calculó el numero de comparaciones e intercambios para el caso medio de cada algoritmo. Se ponderó tres veces más a los intercambios, ya que son operaciones más pesadas.



En el caso de Quicksort la gráfica generada corresponde con un crecimiento n\*log(n), excepto en el caso peor, donde debe ser aproximado mediante n2.

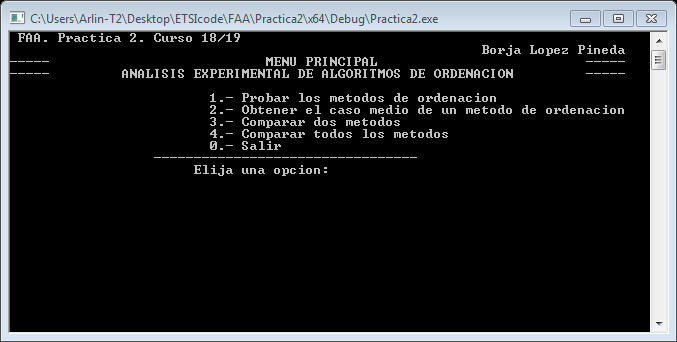
En comparación Quicksort resultó ser el más lento de los cuatro algoritmos para el caso peor, ganando por clara ventaja Inserción.

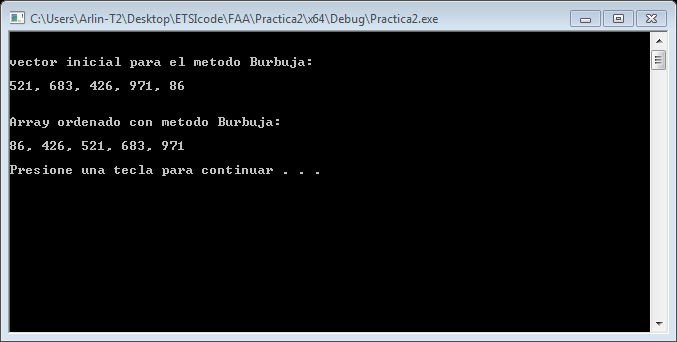
# Diseño de la aplicación

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

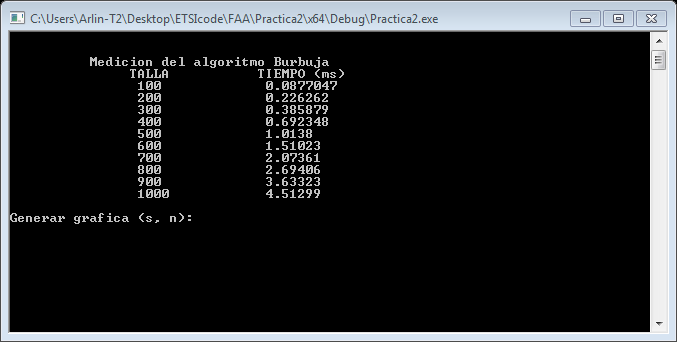
El código soporta agregar más algoritmos, incluso si tienen complejidad temporal distinta a n2.

El método principal junto a la interfaz de menús, está en AnalisisAlgoritmos. Desde allí se llama a los métodos de la clase TestOrdenacion. TestOrdenacion, a su vez, se apoya en la clase ConjuntoInt para la generación de las estancias aleatorias, Mtime para medir el tiempo transcurrido, AlgoritmosOrdenacion para ejecutar los algoritmos, y para generar los archivos de GNUplot y visualizar los datos. Los distintos archivos usan los valores definidos en Constantes.

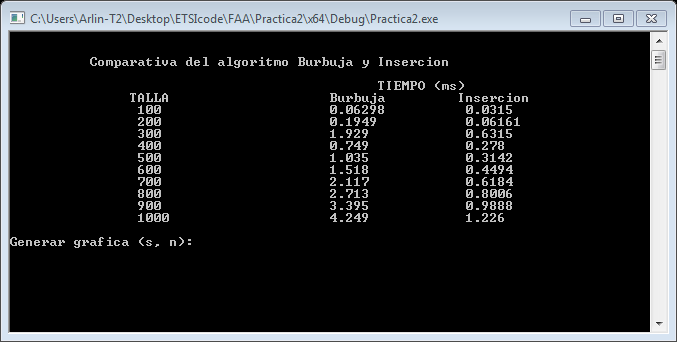
Menú principal, permite acceder a los submenús.



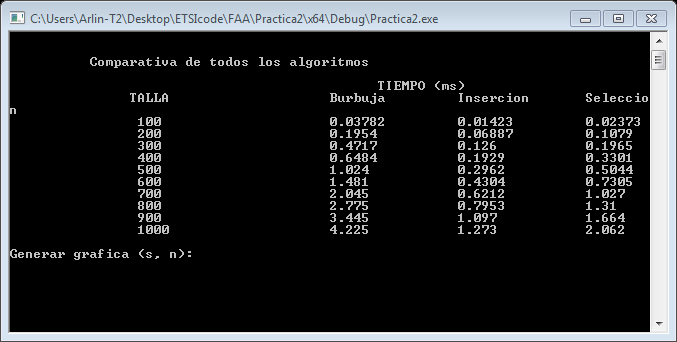
Prueba de los métodos de ordenación. Genera vectores aleatorios del tamaño especificado y los ordena con los distintos algoritmos implementados. Muestra por pantalla el algoritmo antes y después de ser ordenado.



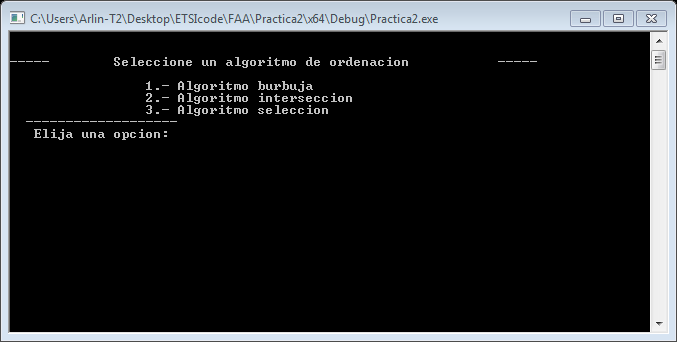
Caso medio de un algoritmo.  
Realiza ordenaciones para instancias de múltiples tallas. Cada operación se repite un numero de veces y se toma la media. Guarda los resultados en un archivo y genera una gráfica si el usuario lo desea.



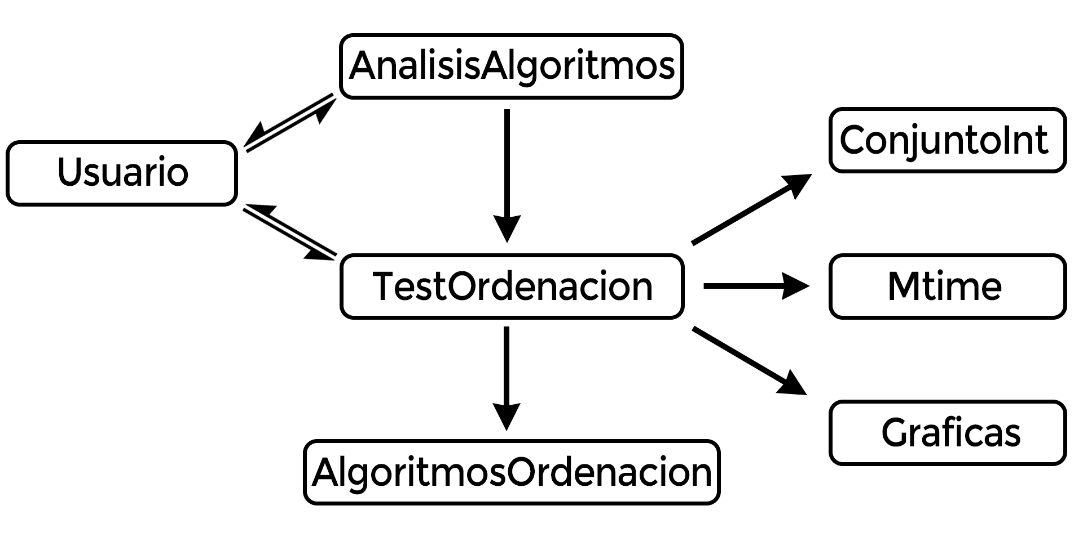
Comparación de dos métodos.  
Mide el coste de dos algoritmos al mismo tiempo. Muestra los resultados, los guarda en un archivo y genera una gráfica doble.



Comparar todos los métodos.  
Mide al mismo tiempo todos los algoritmos. Igualmente muestra los resultados, los guarda en un archivo y genera una gráfica con el número de algoritmos implementados.



Submenú de selección de algoritmos.



# Conclusiones y valoraciones personales de la práctica

Hay una gran cantidad de algoritmos distintos, y para cada uno numerosas modificaciones y optimizaciones. No existe ningún algoritmo que sea superior al resto en todos los aspectos.

La ejecución en diversas máquinas reveló que el tiempo de ejecución y puede variar drásticamente en función de la configuración de hardware. El tiempo medio es hasta cuatro veces inferior en una estación de trabajo frente a un portátil estándar, por tener mayor velocidad de reloj. La forma exponencial de la curva empieza a distinguirse antes cuantos más hilos tiene la máquina, en un portátil estándar si la talla es baja los resultados parecen no ceñirse a la formula teórica. Mientras que en una estación de trabajo puede una forma perfecta en la curva desde el inicio.

Nota: El programa que se adjunta incluye, no solo los algoritmos aquí estudiados, otros algoritmos de ordenación y búsqueda fueron implementados, como actividad completaría. La funcionalidad requerida se mantiene.

## Material consultado

[Visualización Burbuja](http://www.programming-algorithms.net/article/39344/Bubble-sort), [visualización Selección](http://www.programming-algorithms.net/article/39430/Selection-sort), [visualización Inserción](http://www.programming-algorithms.net/article/39459/Insertion-sort), [visualización QuickSort](http://www.programming-algorithms.net/article/39552/Quicksort)