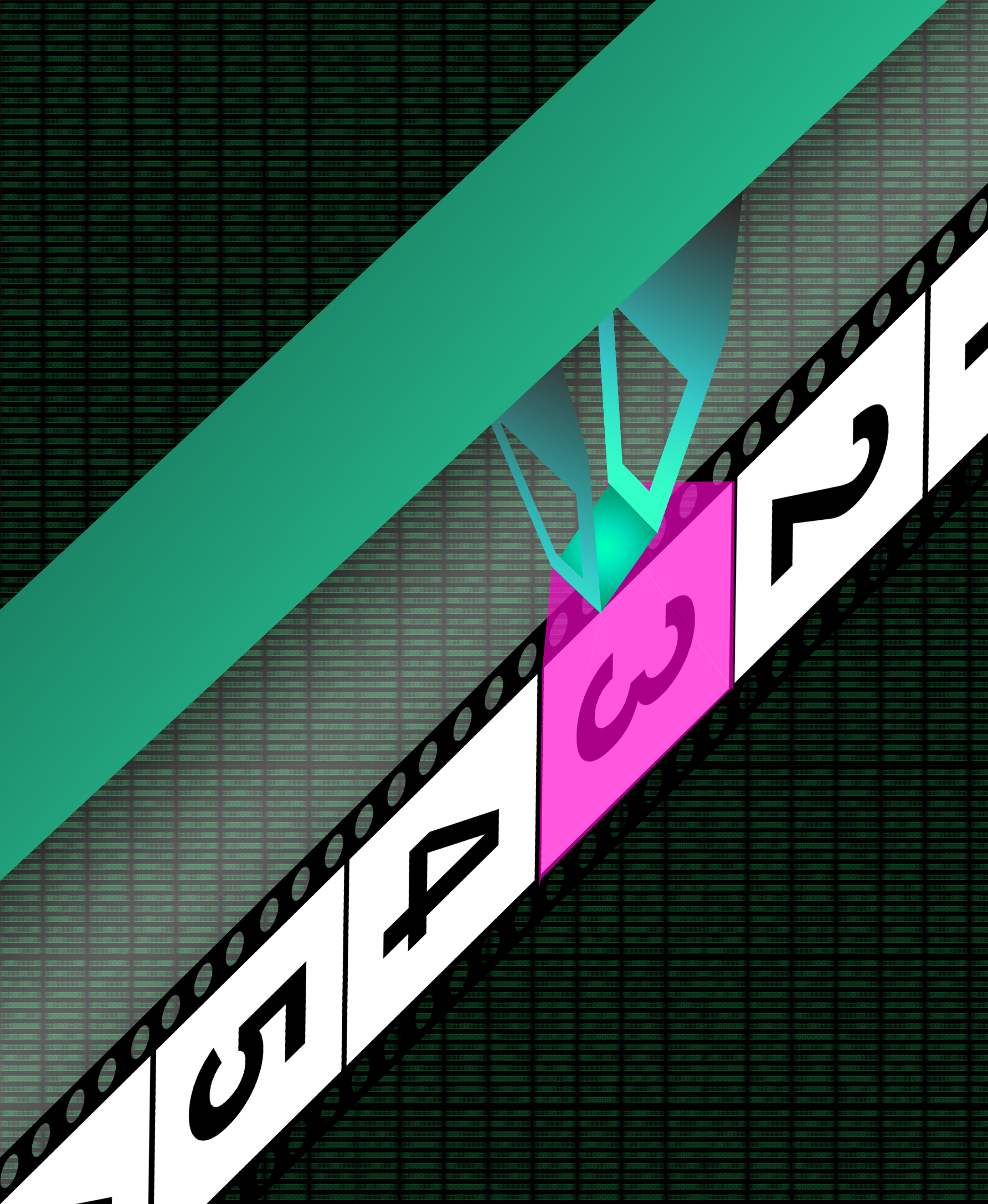
Análisis de algoritmos de búsqueda.



# 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

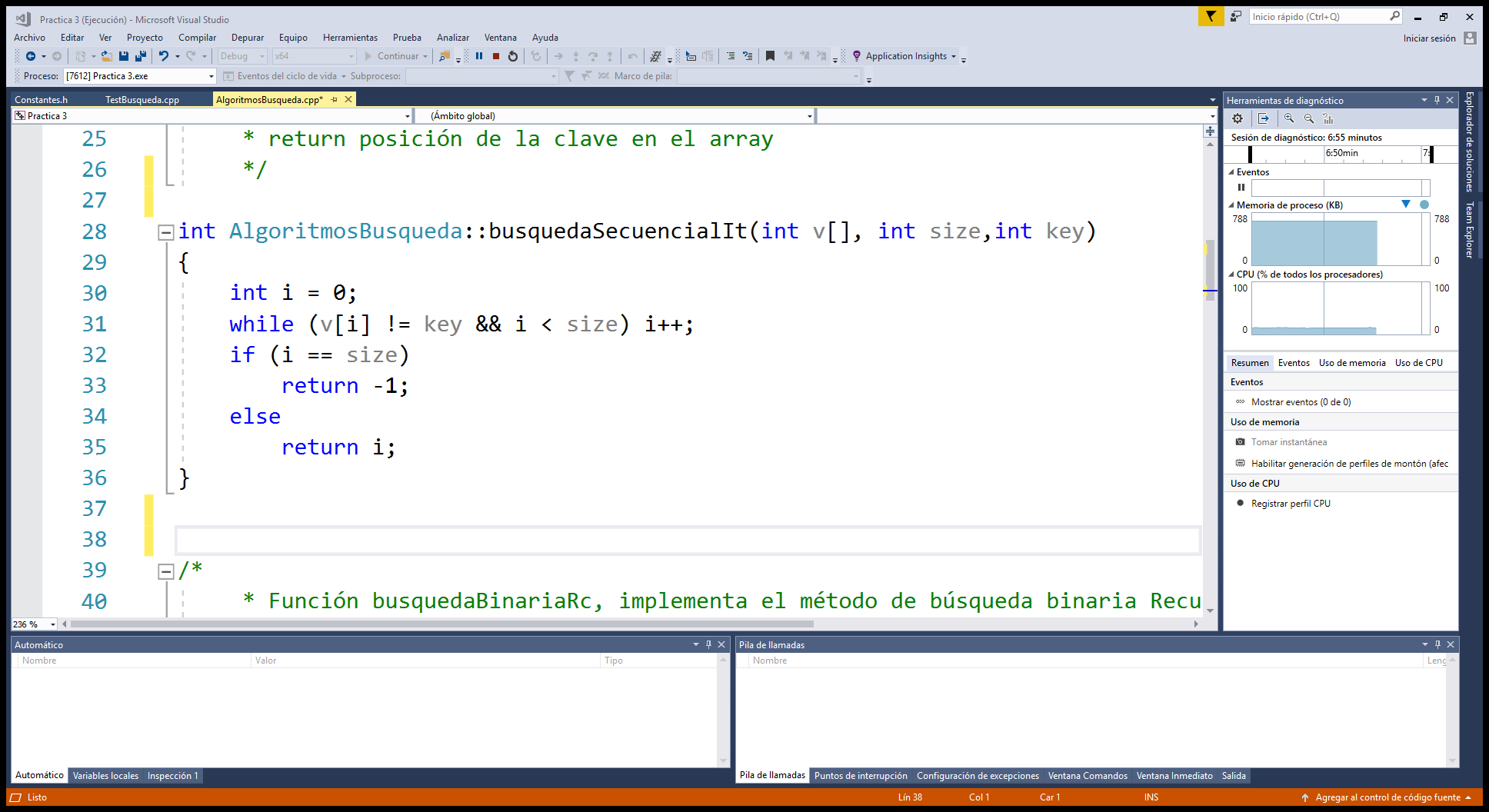
El objetivo de esta práctica es realizar el estudio teórico de múltiples algoritmos de búsqueda, posteriormente implementarlos en un lenguaje de programación y comparar su eficiencia empíricamente.

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.

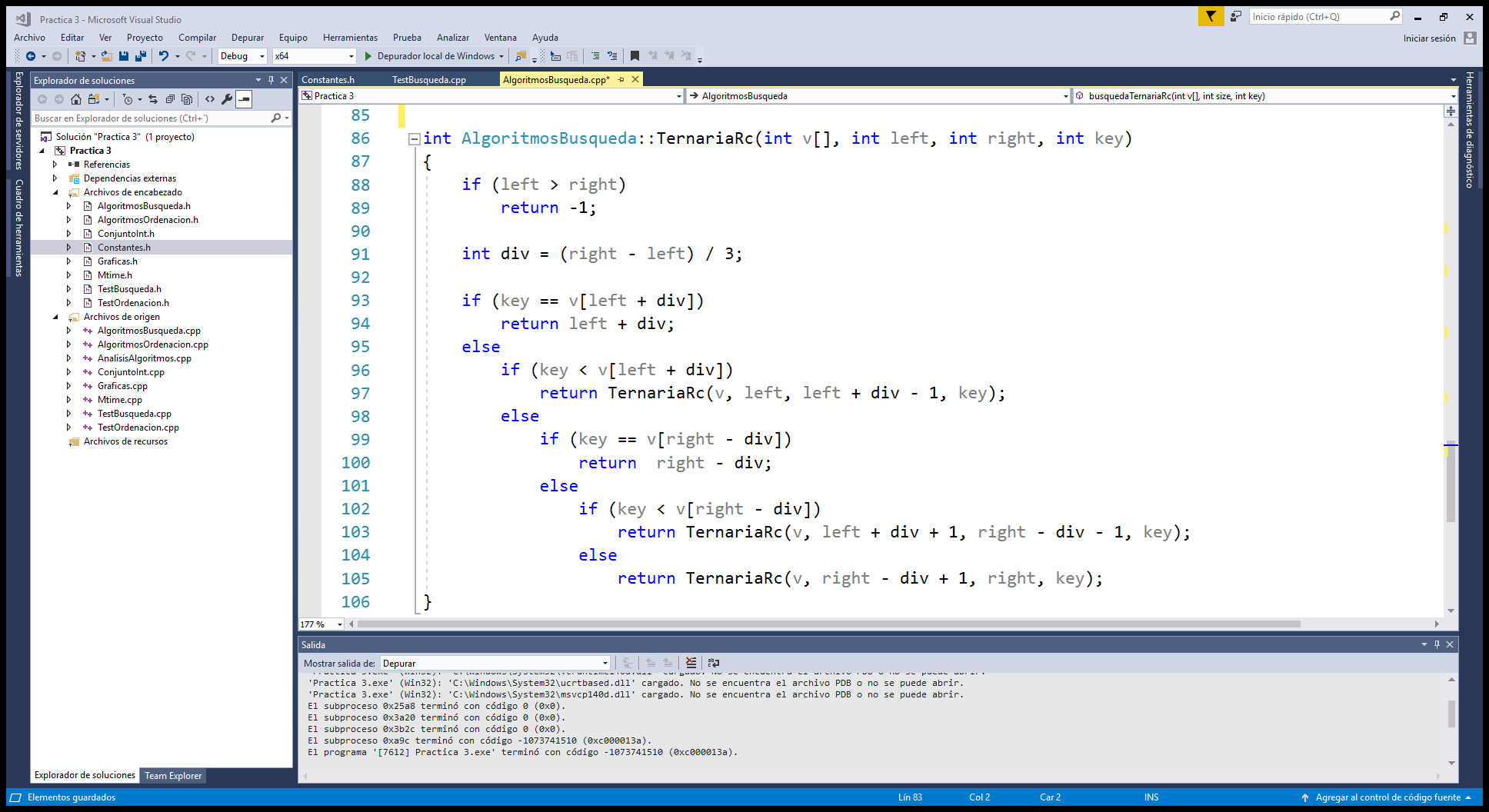
Resulta más eficiente que la búsqueda lineal puesto que solo realiza menos comparaciones. Aunque necesita que el vector esté ordenado.

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.
* Caso medio: La clave se encuentra en el vector, pero no en el centro.

## 

## 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:

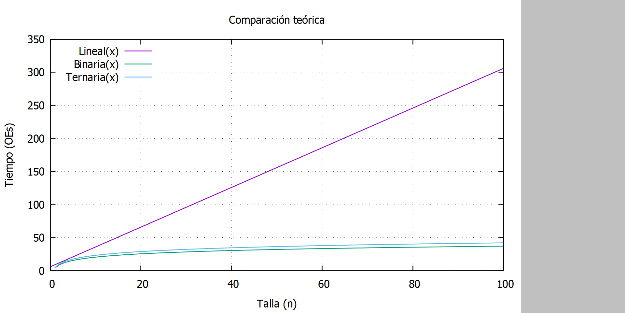
* Caso mejor: Si la clave se encuentra en alguna de las dos divisiones, lo encuentra en la primera llamada.
* Caso peor: Si la clave no está en el vector, tiene que realizar divisiones hasta quedarse con un subconjunto de un elemento.
* Caso medio: La clave se encuentra en el centro del vector. La duración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

## Tabla y graficas de coste

|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Tiempo (COMPARACIONES) | | |
| **Lineal** | **Binaria** | **Ternaria** |
| 100 | 50 | 3.82 | 3.09 |
| 200 | 100 | 4.32 | 3.41 |
| 300 | 150 | 4.61 | 3.59 |
| 400 | 200 | 4.82 | 3.72 |
| 500 | 250 | 4.98 | 3.82 |
| 600 | 300 | 5.11 | 3.91 |
| 700 | 350 | 5.22 | 3.98 |
| 800 | 400 | 5.32 | 4.04 |
| 900 | 450 | 5.41 | 4.09 |
| 1000 | 500 | 5.48 | 4.14 |

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.



Según el conteo de operaciones, la búsqueda ternaria realiza más comparaciones por llamada que la búsqueda binaria. Pero también necesita menos divisiones, menos llamadas, para encontrar la clave; ya que divide el vector en subconjuntos más pequeños.

## Conclusiones

Como se esperaba, el coste temporal del algoritmo depende de la talla del problema, como de la instancia. Los dos algoritmos recursivos son de orden logarítmico (aunque distinta base) y el algoritmo iterativo es lineal.

Por orden de eficiencia: Lineal << Binaria ≈ Ternaria

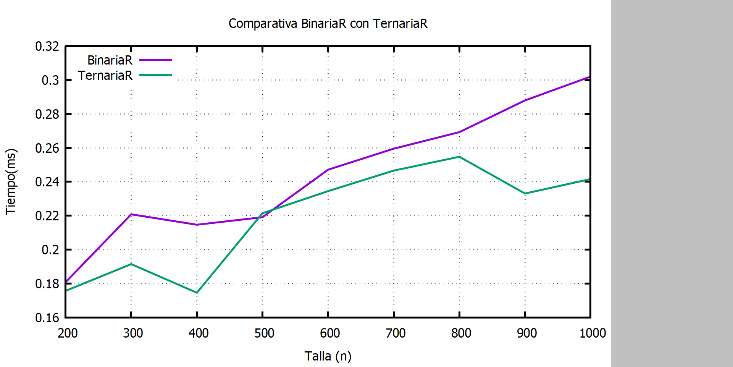
En cuanto a complejidad espacial, todos los algoritmos son , la cantidad de memoria consumida por el algoritmo no es proporcional a le tamaño del vector.  
El tamaño en el stack de lineal es O(1), mientras que la búsqueda binaria y recursiva consume más memoria en la pila. Del roden O(log2n) y O(log3n), respectivamente.

# Cálculo del tiempo experimental

Para cada medición se generan nuevos conjuntos de datos aleatorios y una clave, pero solo se mide el tiempo de búsqueda. La clave es un elemento perteneciente al vector que se escoge de manera aleatoria. Este proceso se repite múltiples veces y se toma la media.

## Tablas y gráficas de coste

|  |  |  |  |
| --- | --- | --- | --- |
| Talla | Tiempo (ms) | | |
| **Lineal** | **Binaria** | **Ternaria** |
| 100 | 0.1030 | 0.1619 | 0.1877 |
| 200 | 0.2031 | 0.2154 | 0.1529 |
| 300 | 0.3121 | 0.1844 | 0.1733 |
| 400 | 0.3635 | 0.2154 | 0.1926 |
| 500 | 0.4224 | 0.2142 | 0.2409 |
| 600 | 0.5053 | 0.2457 | 0.2445 |
| 700 | 0.5823 | 0.2529 | 0.2463 |
| 800 | 0.6904 | 0.3097 | 0.2403 |
| 900 | 0.7737 | 0.2493 | 0.2238 |
| 1000 | 0.9191 | 0.2704 | 0.2511 |



Una comparación más detallada de búsqueda binaria contra ternaria, revela que la búsqueda binaria ligeramente es más lenta que la ternaria.

## Conclusiones

Los resultados individuales muestran un grado tolerable de dispersión, sin elementos anómalos. Cabe mencionar que la aproximación es mejor en los algoritmos de complejidad lineal, que en los logarítmicos.

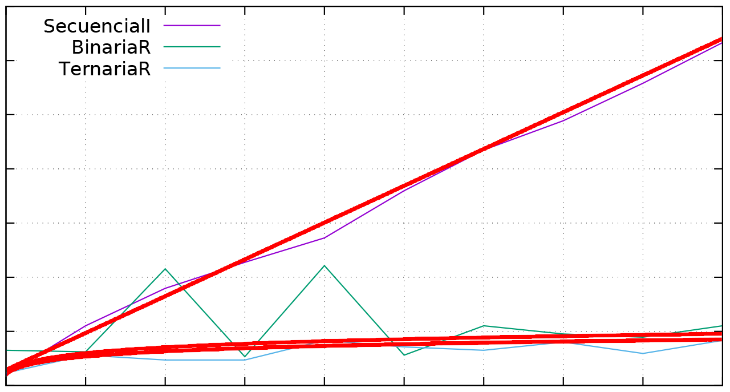
Puede apreciarse la rápida tendencia a crecer de las funciones lineales, contra la asíntota de las logarítmicas. Seguir aumentando la talla, requeriría un tiempo de ejecución proporcional; frente a la desaceleración de búsqueda binaria y ternaria.

Pese a esto, se ha observado que los algoritmos recursivos resultan más lentos que la búsqueda lineal en tallas pequeñas y que su eficiencia se vuelve presente a partir de un cuarto 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 la búsqueda lineal el más lento, binaria y ternaria bastante parecidos.

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 tomó el número de operaciones elementales de cada algoritmo.

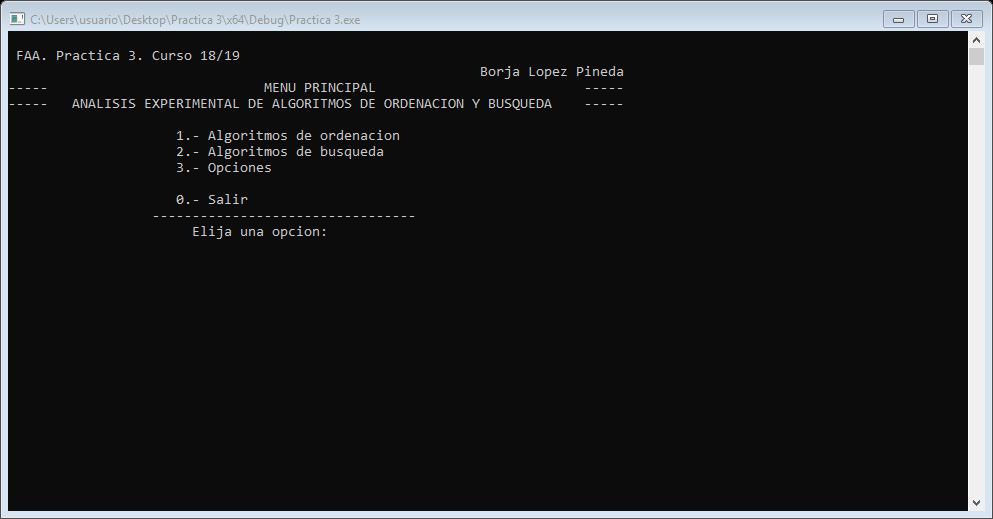
En este caso, un análisis por conteo de operaciones elementales no ofrece los mejores resultados. Ya que es importante notar que, aunque la búsqueda lineal es la que más comparaciones realiza, también es la que menos instrucciones por comparación tiene. Mientras que búsqueda ternaria realiza menos comparaciones, también necesita más operaciones por división.

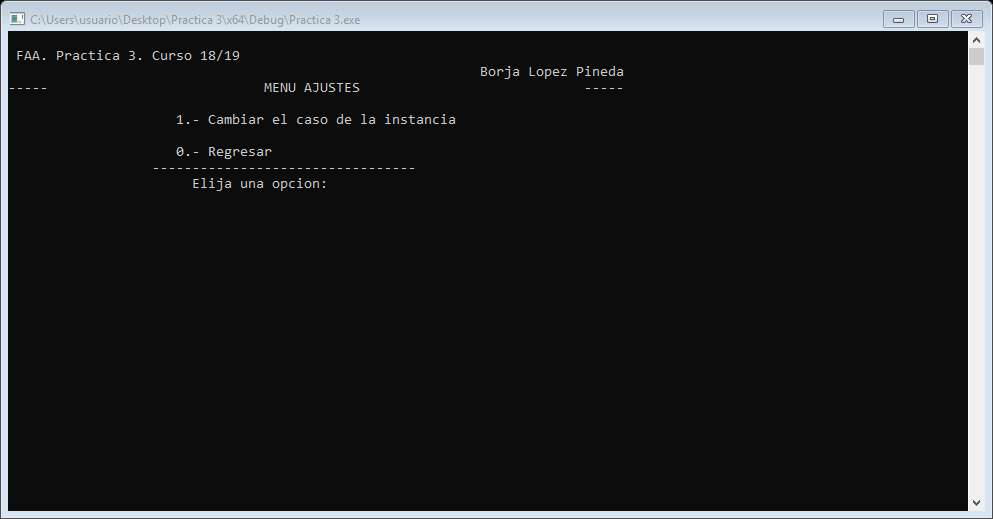
# Diseño de la aplicación

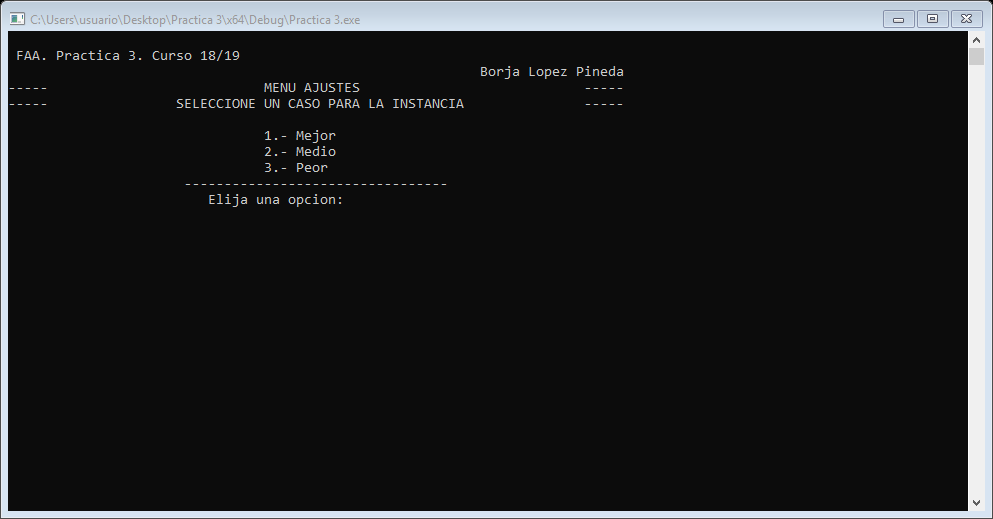
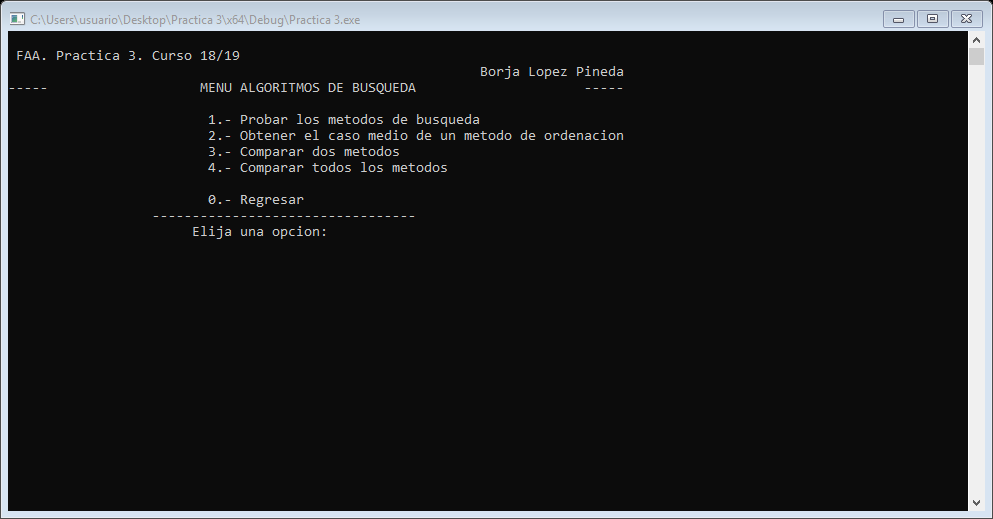
La metodología de programación escogida para la implementación fue diseño modular con orientación a objetos.

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

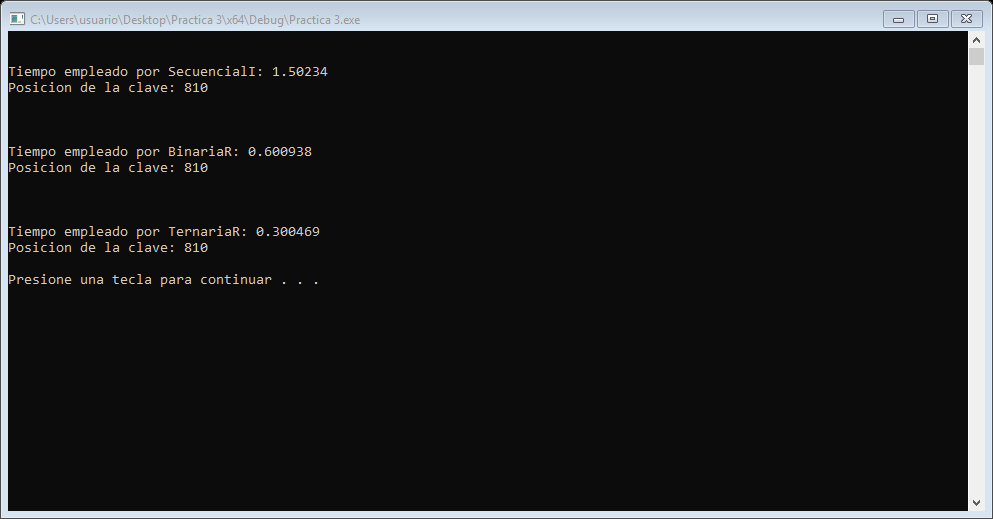
El método principal junto a la interfaz de menús, está en AnalisisAlgoritmos. Desde allí se llama a los métodos de la clase TestOrdenacion y TestBusqueda. TestOrdenacion y TestBusqueda, a su vez, se apoyan en la clase ConjuntoInt para la generación de las estancias aleatorias, Mtime para medir el tiempo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

Menú principal, permite acceder a los menús de ordenación o búsqueda. Las opciones se usan para modificar constantes en el programa.

Actualmente la única disponible es el caso de la instancia. En el caso mejor, ordenará el vector automáticamente. En el caso peor, lo ordena inversamente.  
En el caso medio, no lo ordena.

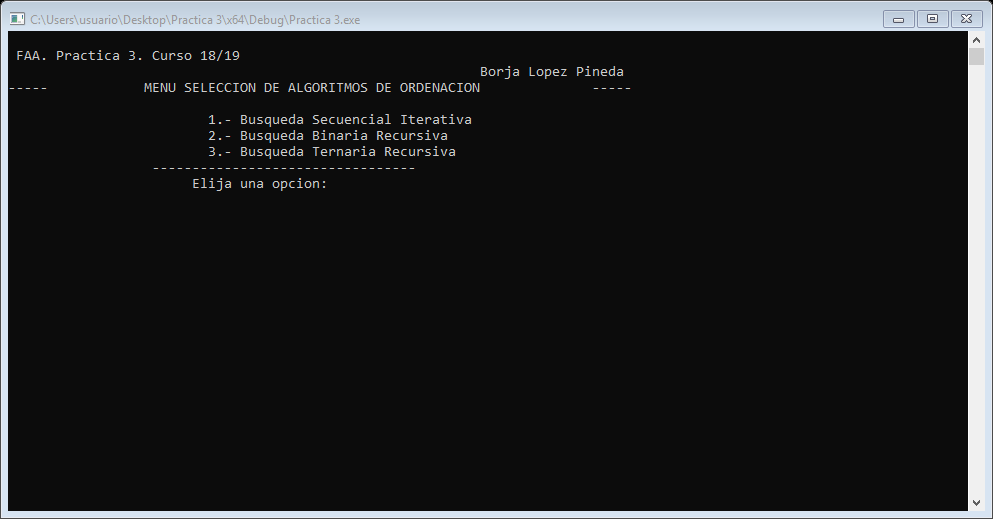
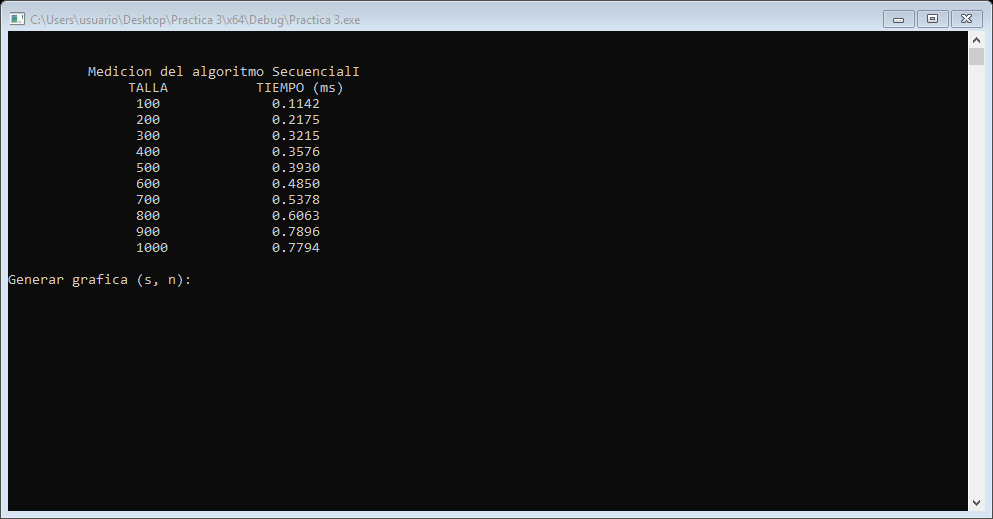
Determinados algoritmos como la búsqueda binaria y ternaria, requieren que el vector esté ordenado; en ese caso el programa cambia el modo automáticamente.

Este es el menú de los métodos de búsqueda. Tiene los mismos elementos que el menú de ordenación.

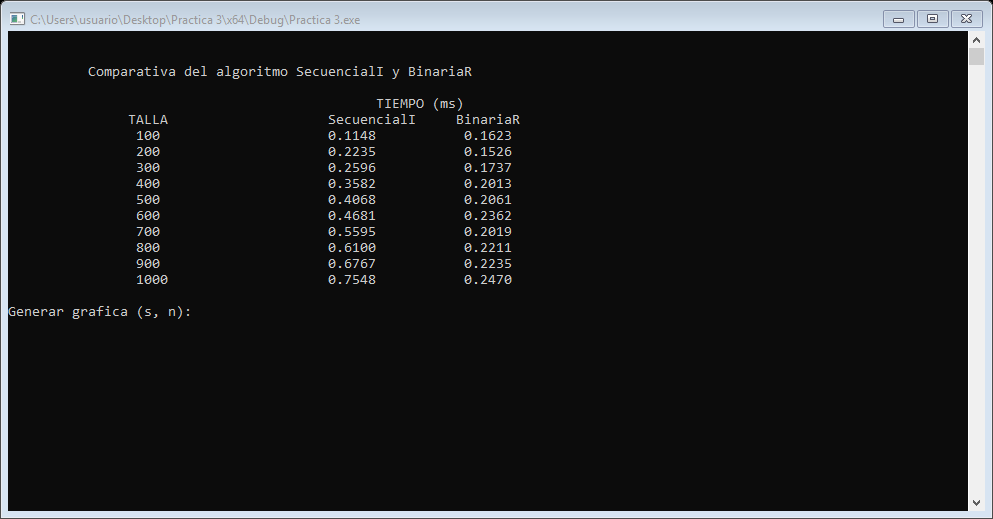


Probar los algoritmos, genera un vector aleatorio de la talla que el usuario específico y manda a buscar una clave, también dada, mediante los algoritmos implementados.

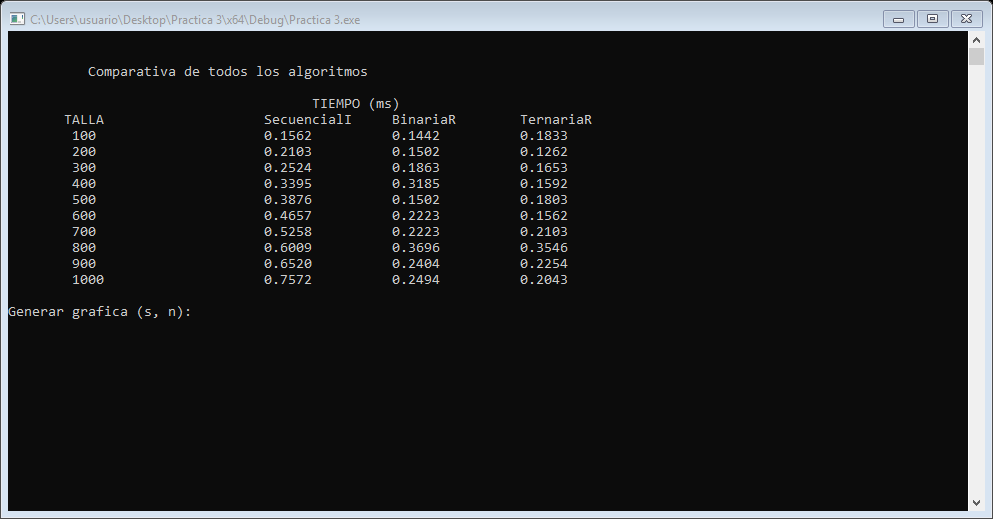
Muestra los resultados y el tiempo que tardó cada uno.

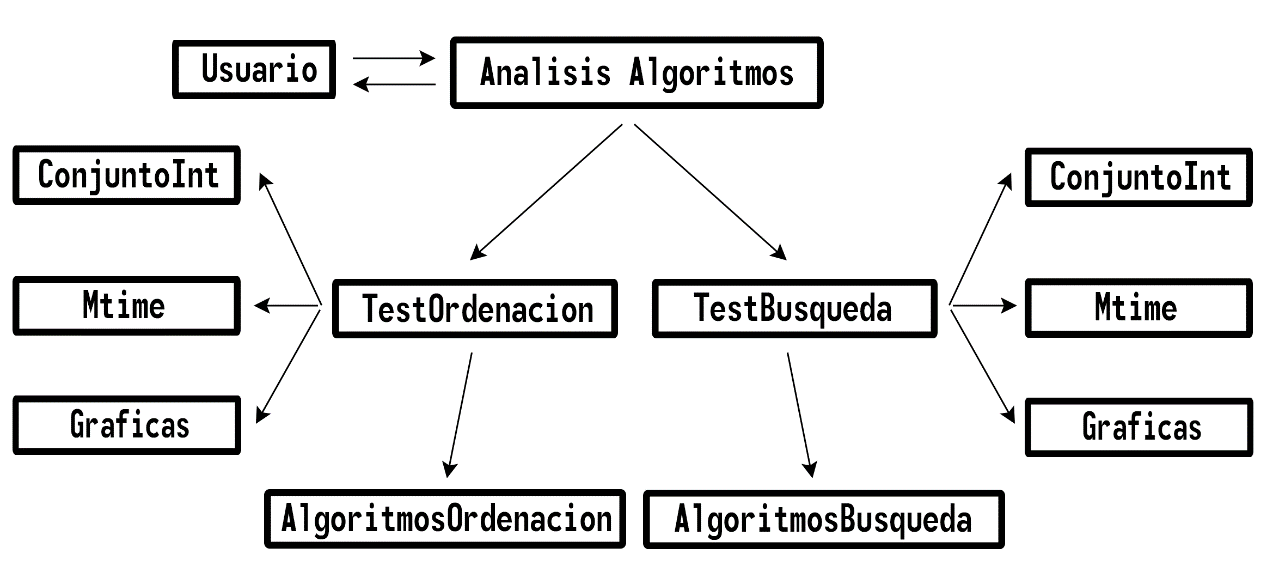
Este es el diálogo para seleccionar un algoritmo de búsqueda. Se usa en múltiples zonas del programa.

Caso medio de un algoritmo.  
Realiza búsquedas para instancias de varias tallas. Cada operación se repite un número 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 de búsqueda implementados.



# Conclusiones y valoraciones personales de la práctica

Hay una menor cantidad de algoritmos de búsqueda que de ordenación, pudiendo deberse a la aparente trivialidad del problema.

Al igual que con los algoritmos de ordenación, no existe un algoritmo que sea superior a los demás en todos los aspectos. La búsqueda binaria puede resultar especialmente útil en grandes bases de datos, donde el guardado se realice de forma ordenada para que la búsqueda sea mucho más rápida.

## Material adicional consultado

[Búsqueda ternaria](https://www.geeksforgeeks.org/ternary-search/), [Búsqueda binaria](https://www.geeksforgeeks.org/binary-search/).