Memoria Practica 1

Algoritmos y Modelos de Computación

Almudena García Jurado-Centurión

1. Objetivo de la practica

En esta practica se analizaran los tiempos de varios algoritmos de ordenación y búsqueda, y se compararan estos con los resultados teóricos de los mismos, para comprobar si estos concuerdan con los anteriores

Los algoritmos de ordenación que se analizaran serán: BubbleSort, Insertion Sort, Selection Sort, Shell Sort, HeapSort, MergeSort y QuickSort

Como algoritmos de búsqueda usaremos: Binary Search, Lineal Search, Hash Search con dispersión cerrada, y Hash Search con dispersión abierta

Para analizar sus tiempos, generaremos vectores de diferentes tamaños, y mediremos la ejecución de los algoritmos en el caso medio

Para asegurar la fiabilidad de estas medidas, se realizaran varias repeticiones por cada tamaño del vector, y se obtendrán los tiempos medios de sus ejecuciones

Una vez obtenidos estos tiempos, se generara una gráfica con los mismos, ajustándola a la función correspondiente a su orden de complejidad teórico

2. Análisis de los algoritmos

- **Algoritmos de ordenación**

  Para el análisis practico, hemos usado vectores de 500 a 20000 elementos, con 20 repeticiones por cada tamaño

  - **BubbleSort**

    - Descripción del Algoritmo

    Se basa en el principio de comparar e intercambiar pares de elementos adyacentes hasta que todos estén ordenados.

    Desde el primer elemento hasta el penúltimo no ordenado

    • comparar cada elemento con su sucesor

    • intercambiar si no están en orden

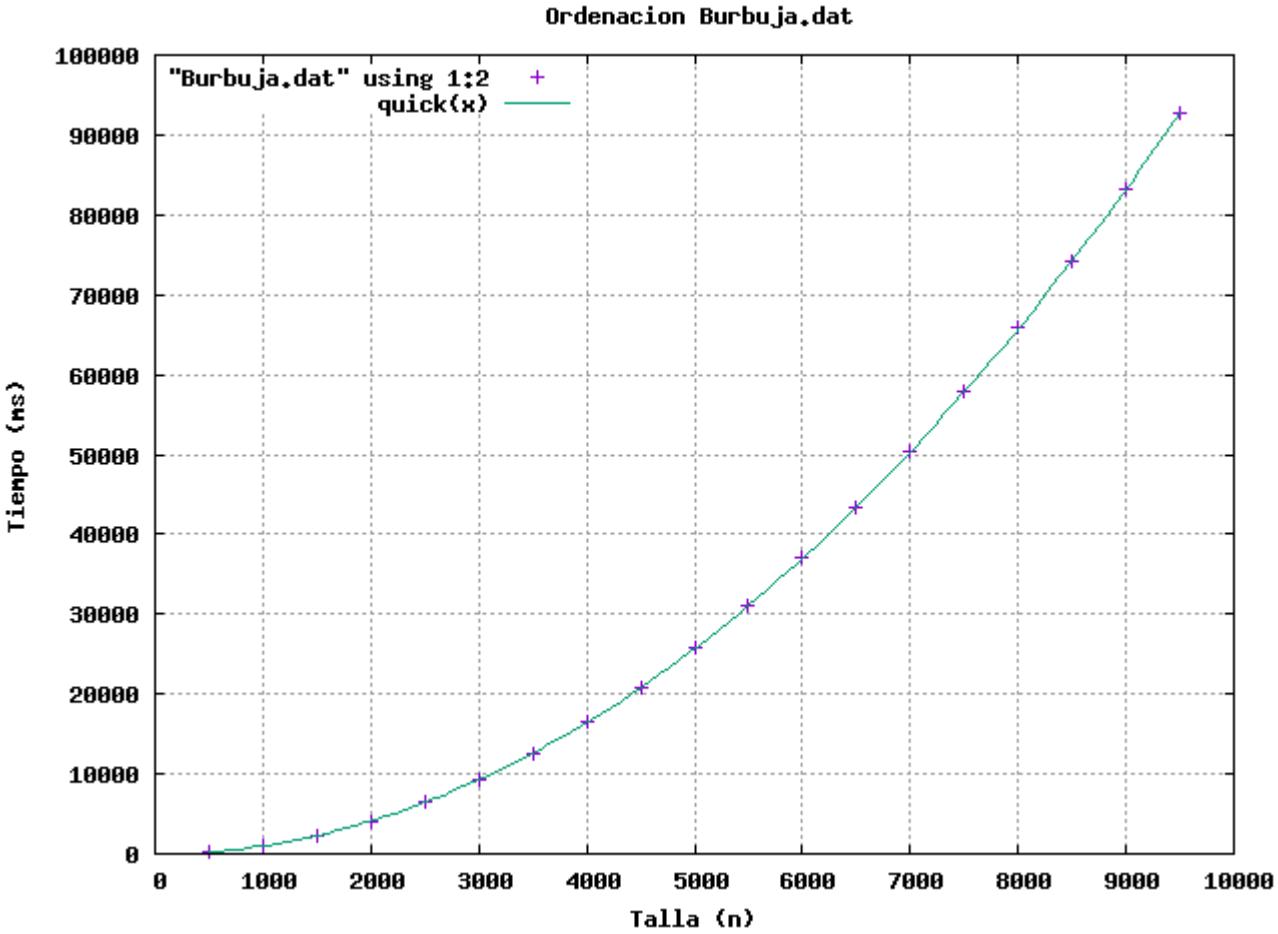
    - **Análisis Teórico**

    Consta de un bucle for, con otro bucle for anidado que realiza una serie de operaciones básicas, por lo que podemos afirmar que el orden es $O(n^2)$

    - **Análisis Practico**

    En el análisis practico, hemos comparando los tiempos con $f(x)= x^2$, con el siguiente resultado

    Para hacer la prueba, hemos usado 20 repeticiones con tamaños de 1000 a 10000



Como se puede comprobar, los tiempos del algoritmo coinciden a la perfección con la curva de la función, lo cual demuestra que el calculo teórico es correcto

El coste temporal es muy alto, llegando a cerca de un 10 segundos en los vectores de mayor tamaño

- **Insertion Sort**

  - Descripción del Algoritmo

  También conocido como método de la baraja.

  Consiste en tomar el vector elemento a elemento e ir insertando cada elemento en su posición correcta de manera que se mantiene el orden de los elementos ya ordenados

  Inicialmente se toma el primer elemento, a continuación se toma el segundo y se inserta en la posición adecuada para que ambos estén ordenados, se toma el tercero y se vuelve a insertar en la posición adecuada para que los tres estén ordenados, y así sucesivamente.

  - Pseudocódigo

  1. Suponemos el primer elemento ordenado.
  2. Desde el segundo hasta el último elemento, hacer:
     1. suponer ordenados los (i –1) primeros elementos
     2. tomar el elemento i
     3. buscar su posición correcta
     4. insertar dicho elemento, obteniendo i elementos ordenados

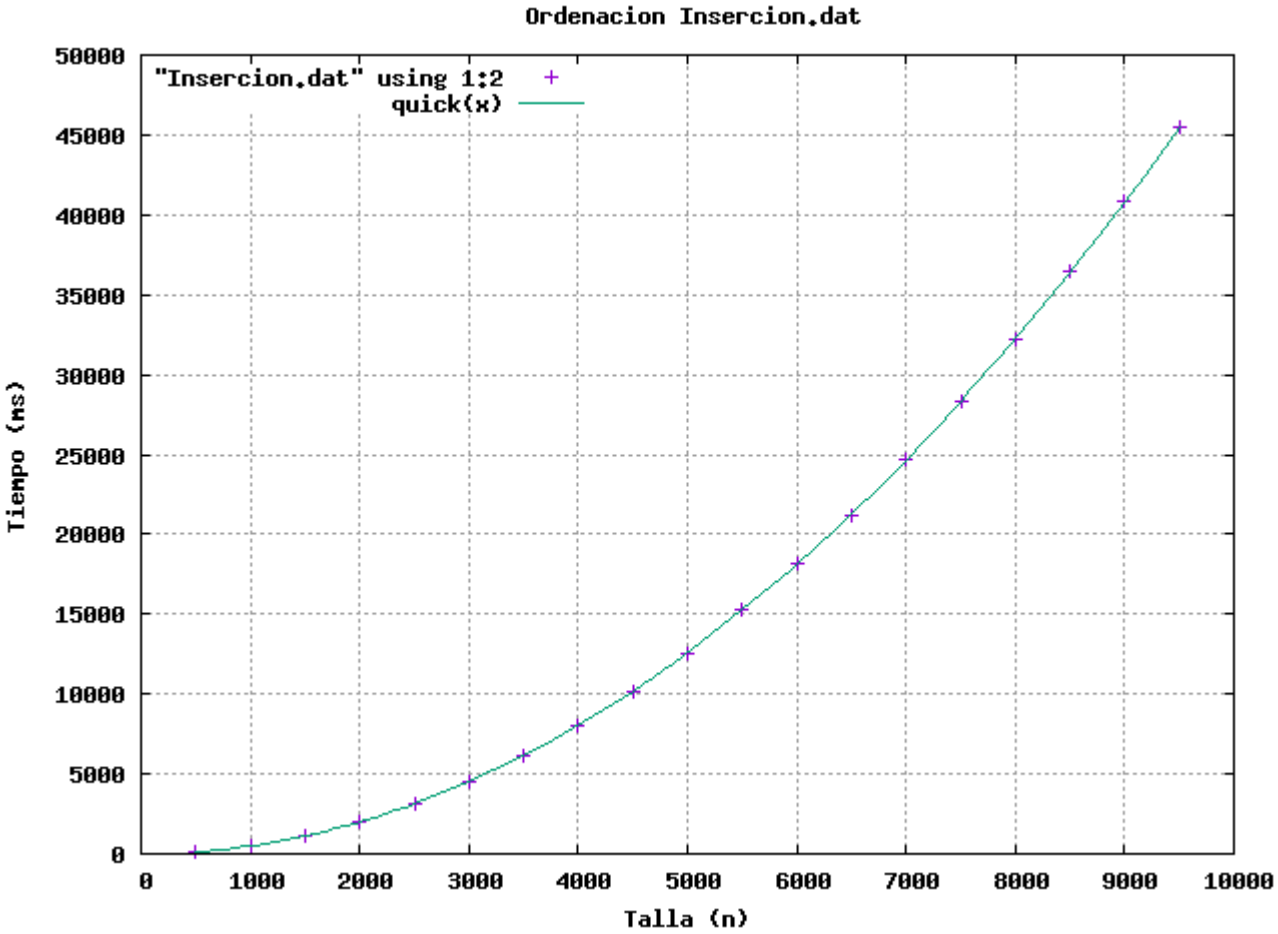
  - Análisis teórico

  Consta de un bucle for, con un bucle de tipo while anidado, en que se realizan una serie de operaciones básicas O(1). Por lo tanto, la complejidad temporal del algoritmo será de orden $O(n^2)$.

  - Análisis práctico

  En el análisis, hemos usado el mismo numero de repeticiones y rango de tamaños que en BubbleSort.

  Al ejecutar el algoritmo comparándolo con $f(x) = x^2$, el resultado ha sido el siguiente:



Se vuelve a comprobar que el ajuste es correcto, y los puntos se corresponden a la curva de la función, aunque en este caso los tiempos son menores que con el BubbleSort

- **Selection Sort**

  - Descripción del Algoritmo

  Este método se basa en que cada vez que se mueve un elemento, se lleva a su posición correcta

  Se comienza examinando todos los elementos, se localiza el más pequeño y se sitúa en la primera posición.

  A continuación, se localiza el menor de los restantes y se sitúa en la segunda posición.
  Se procede de manera similar sucesivamente hasta que quedan dos elementos. Entonces se localiza el menor y se sitúa en la penúltima posición y el último elemento, que será el mayor de todos,ya queda automáticamente colocado en su posición correcta.

  - Pseudocódigo

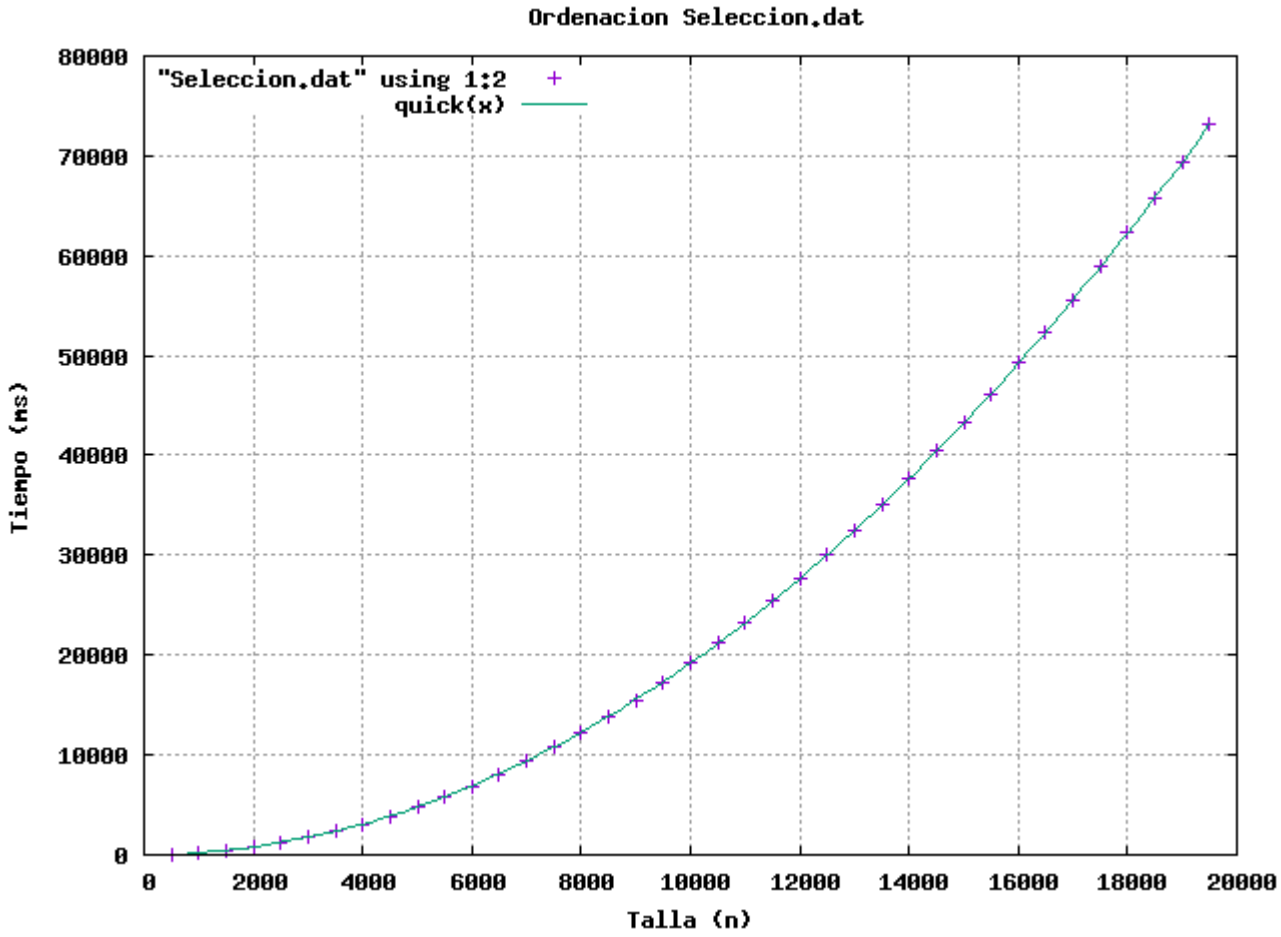
  Para i desde la primera posición hasta la penúltima:
  - localizar menor desde i hasta el final
  - intercambiar ambos elementos

  - Análisis Teórico

  Consta de dos bucles for anidados, junto con una serie de operaciones básicas de orden constante, por lo que el algoritmo es de orden $O(n^2)$

- Análisis Práctico

Tras la ejecución del algoritmo, al comparar los tiempos con $f(x) = x^2$, el resultado es este:



Los tiempos son bastante parecidos a Insertion Sort, y el ajuste a n^2 es bastante preciso

- **Shell Sort**

  - Descripción del Algoritmo

  Es una mejora de la ordenación por inserción(colocar cada elemento en su posición correcta, moviendo todos los elementos mayores que él, una posición a la derecha), que se utiliza cuando el número de datos a ordenar es grande.

  Para ordenar una secuencia de elementos se procede así:

  1. Se selecciona una distancia inicial y se ordenan todos los elementos de acuerdo a esa distancia, es decir, cada elemento separado de otro a distancia estará ordenado con respecto a él.

  2. Se disminuye esa distancia progresivamente,hasta que se tenga distancia 1 y todos los elementos estén ordenados.

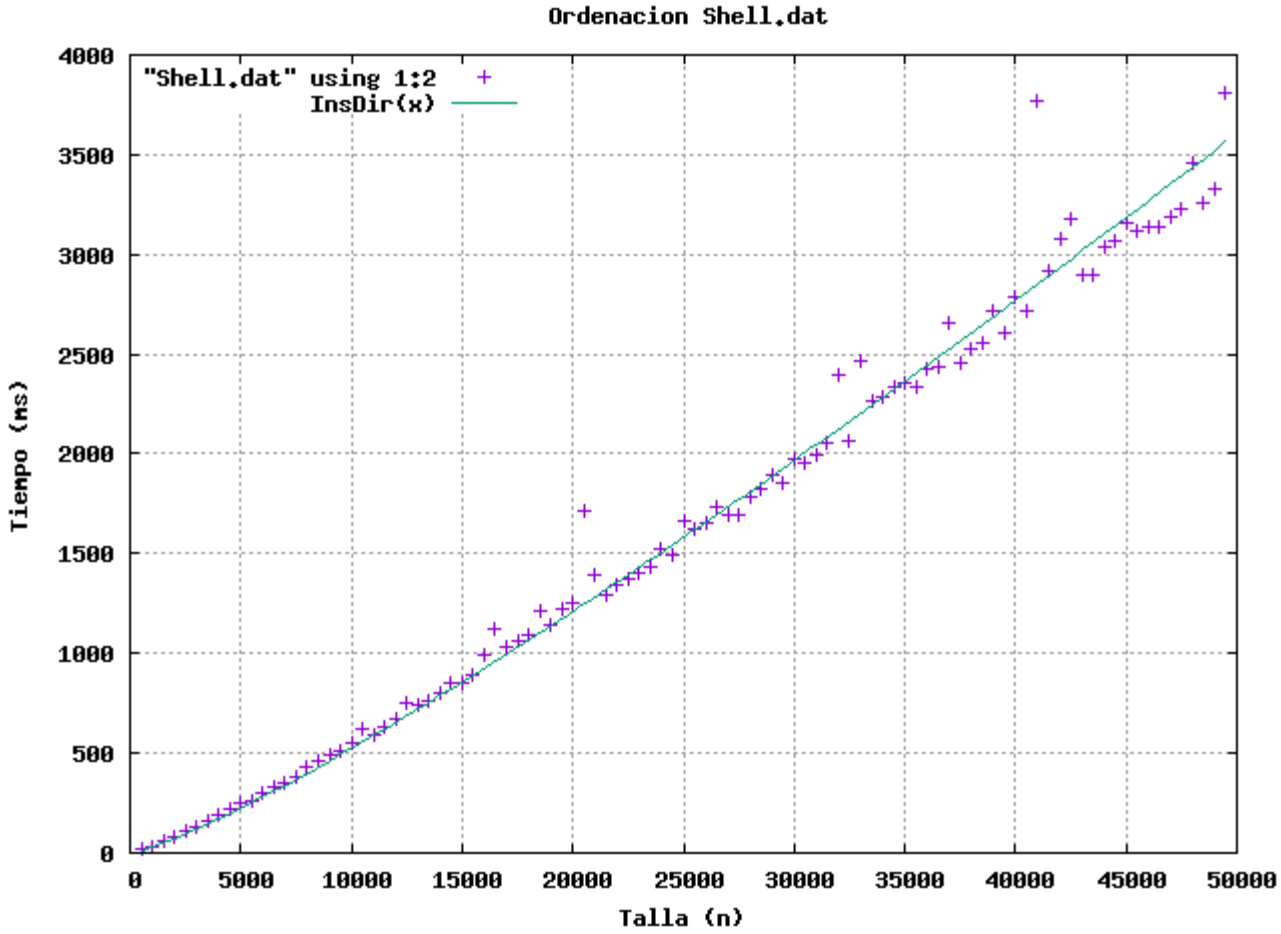
  - Análisis Teórico

  Para este algoritmo no hemos podido analizar sus ordenes de complejidad

- **Análisis Practico**

  Para analizar este algoritmo, hemos usado 50 repeticiones, con tamaños de 500 a 50000

  Hemos usado como referencia el $O(n \log^2(n))$ para su ajuste en la gráfica, con los siguientes resultados



Se aprecian ligeras irregularidades en la ejecución, mediante diferencias entre los puntos del algoritmo y la recta teórica $f(x) = x$, que se van incrementando conforme aumenta el tamaño

También se aprecia bastante la reducción de tiempos respecto a los anteriores algoritmos de $O(n^2)$, pasando de tiempos mayores al minuto, a tiempos inferiores al segundo

- **HeapSort**

- Descripción del Algoritmo

Se basa en el uso de una estructura de datos llamada montículo

Los montículos son implementaciones eficientes de las colas de prioridad.

Un montículo es un árbol binario con una propiedad invariante:

- o bien es el árbol vacío,
- o bien la raíz es menor (mayor) o igual que el resto de los elementos y,adicionalmente, los hijos izquierdo y derecho son montículos.

Ello implica que el menor (mayor) elemento de la estructura se encuentra en la raíz, y se puede consultar con coste constante.

En el algoritmo HeapSort se insertan los elementos en un montículo de máximos, y se va eliminando sucesivamente el mayor para obtener los elementos en orden inverso

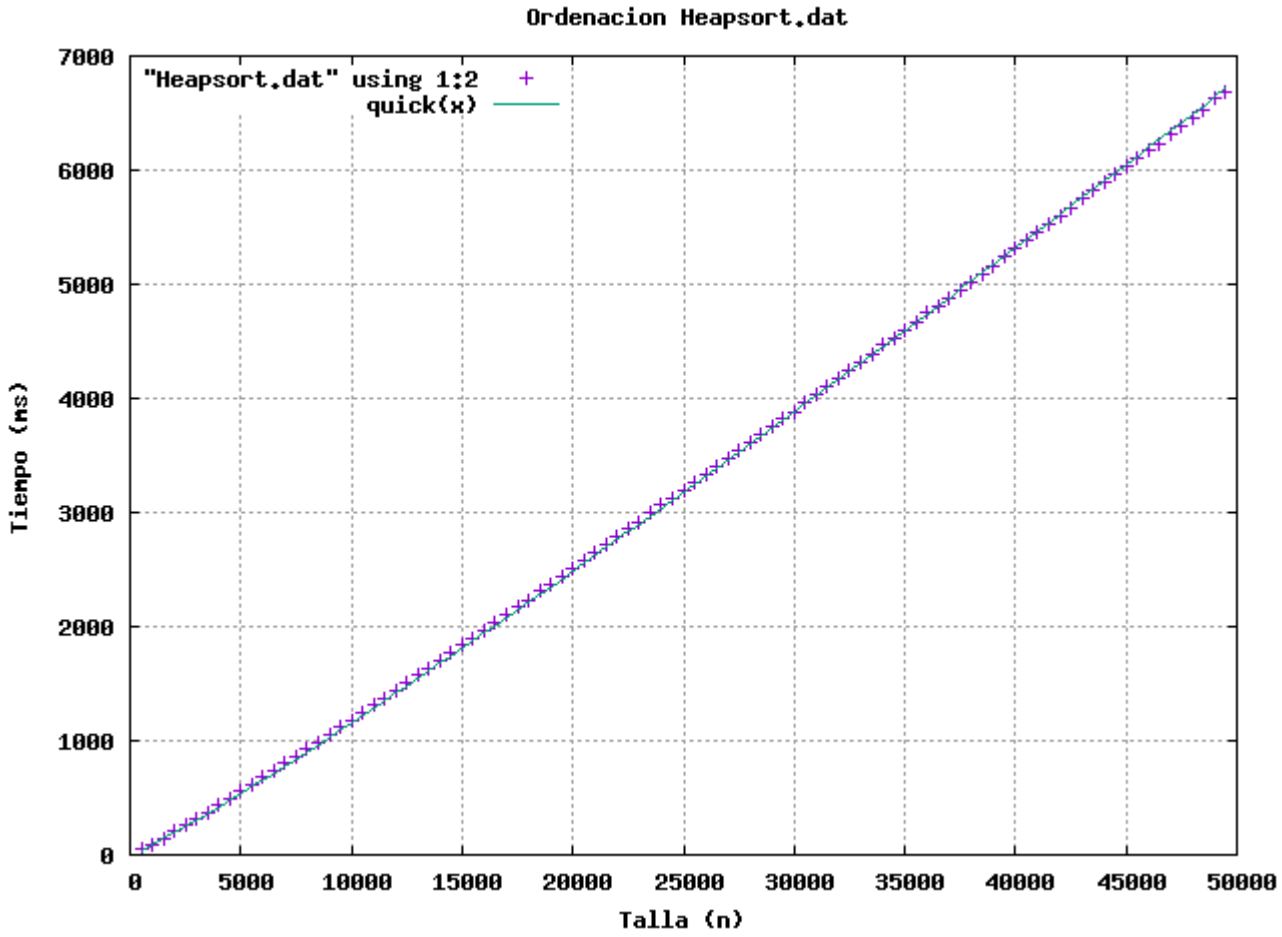
- Análisis Teórico

Consta de un bucle for que realiza operaciones de inserción sobre una un montículo (coste logarítmico, O(log n)), seguido de un bucle for -no anidado- que realiza operaciones de extracción sobre la misma estructura de datos (también de coste logarítmico). Por lo tanto, el algoritmo es de orden O(n log n).

- Análisis Practico

Para analizar este algoritmo, hemos usado el mismo número de repeticiones y rango de tamaños que con ShellSort.

Al ejecutar el algoritmo, ajustando la gráfica a f(x) = x*log(x), el resultado es el siguiente:



En esta gráfica comprobamos que el ajuste a la función es bastante preciso, con unos tiempos mayores que shellsort, pero menores que BubbleSort e InsertionSort

- **QuickSort**

- Descripción del Algoritmo

Quick-sort se basa en la estrategia “dividir para vencer”.
Tomamos un valor x (llamado “pivote”) y separamos los valores que son mayores o iguales que x a la derecha y los menores a la izquierda.
Está claro que ahora solo hace falta ordenar los elementos en cada uno de los subrangos

- Análisis del Algoritmo

Para calcular el orden temporal del algoritmo quick sort, vamos a utilizar la fórmula maestra para la reducción por división:

T(n) = c, si 1 <= n < b
T(n) = O(n^k log n), si a = b^k

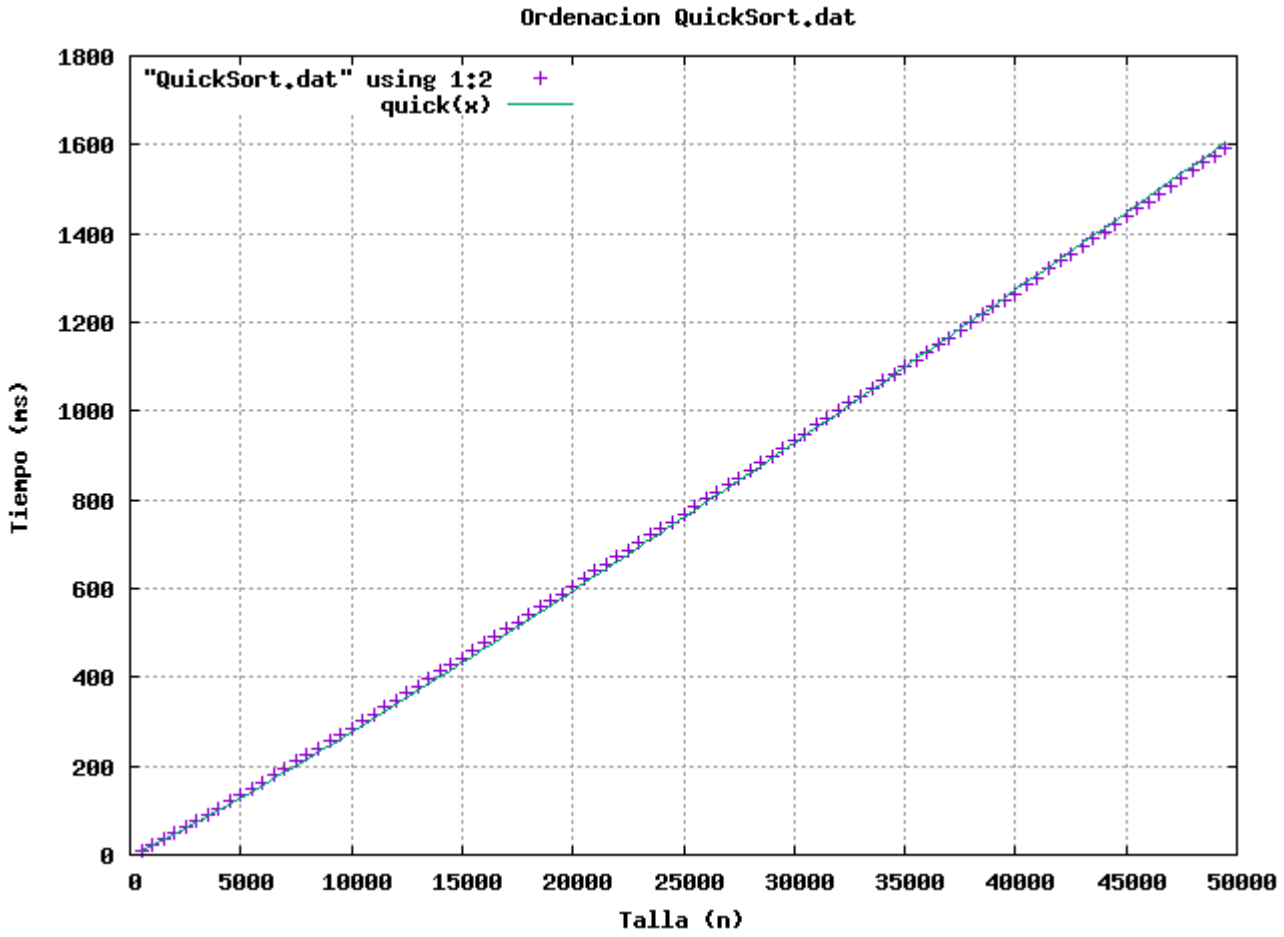
Teniendo que a = 2 (número de llamadas recursivas), b = 2 (factor de disminución del tamaño de los datos) y con f(n) = n (coste de combinación de los resultados y preparación de las llamadas), podemos obtener la complejidad.

En este caso k = 2 (f(n) = c * n^k -> n = c * n^k -> k = 1). Por lo tanto, y teniendo en cuenta que a = b^k (2 = 2^1), el orden del algoritmo es O(n^1 log n) = O(n log n).

- Análisis Práctico

En este análisis, hemos continuado con el mismo número de repeticiones y rango de números que en los anteriores.

Al ejecutar el algoritmo, ajustando la gráfica a f(x) = xlog(x), el resultado es el siguiente:



El resultado se asemeja bastante a la función de ajuste, y con unos tiempos bastante menores a los anteriores algoritmos, siendo menores a los 1700 ms.

- **MergeSort**

- Descripción del Algoritmo

La estrategia es típica de “dividir para vencer” :

- Dividir los elementos en dos secuencias de la misma longitud

aproximadamente.

- Ordenar de forma independiente cada subsecuencia.
- Mezclar las dos secuencias ordenadas para producir la secuencia

final ordenada.

- Análisis Teórico

Para calcular el orden temporal del algoritmo merge sort, vamos a utilizar la fórmula maestra para la reducción por división:

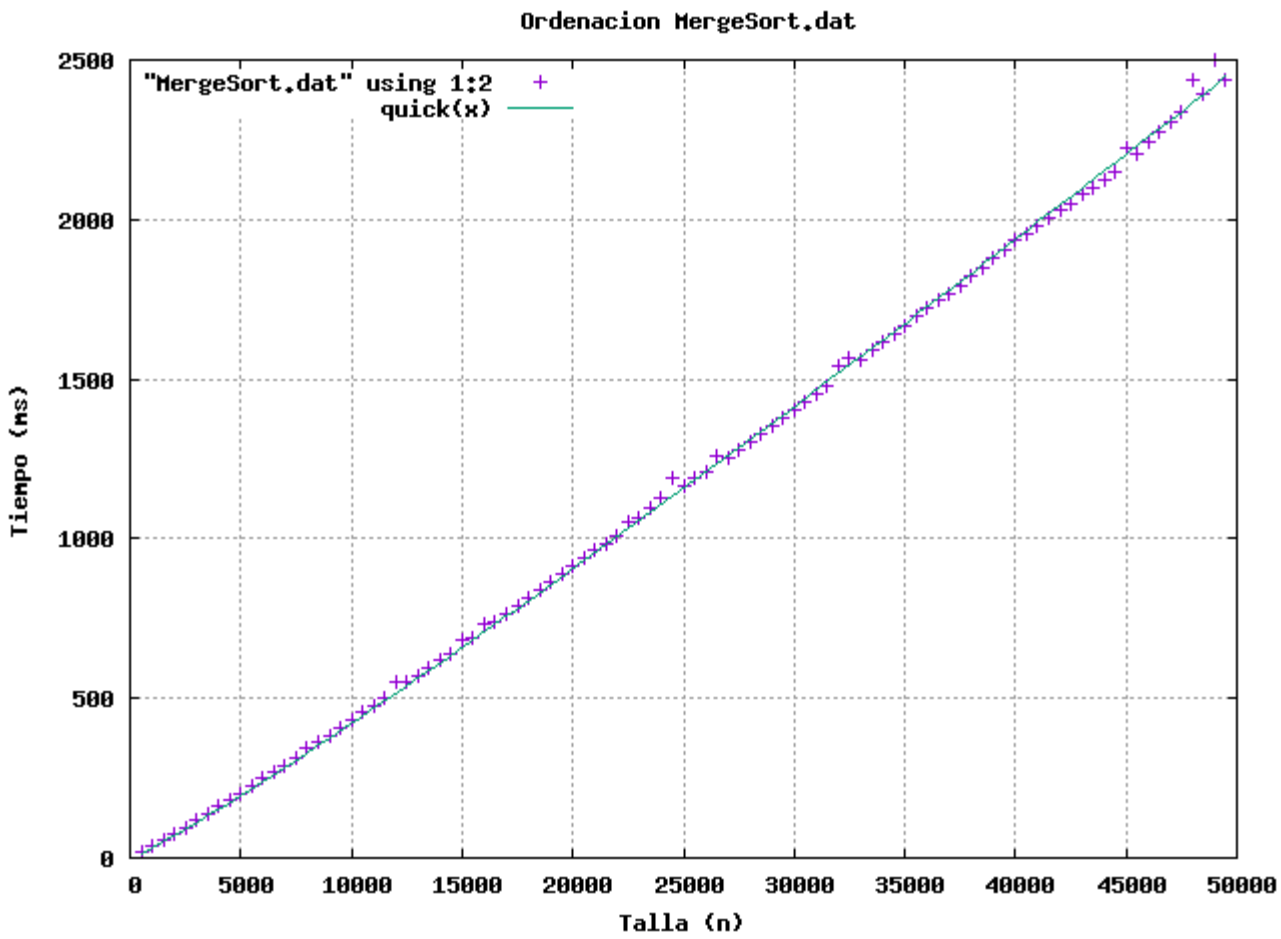
$$T(n) = \begin{cases} c & \text{si } 1 <= n < b \\ a \cdot T(n/b) + f(n), & \text{si } n >= b \\ O(n^k \log n), & \text{si } a = b^k \end{cases}$$

Teniendo que a = 2 (número de llamadas recursivas), b = 2 (factor de disminución del tamaño de los datos) y con f(n) = n (coste de combinación de los resultados y preparación de las llamadas), podemos obtener la complejidad.

En este caso k = 2 ($f(n) = c * n^k$ -> $n = c * n^k$ -> k = 1). Por lo tanto, y teniendo en cuenta que $a = b^k$ ($2 = 2^1$), el orden del algoritmo es $O(n^1 \log n) = O(n \log n)$.

- Análisis Práctico

Al ejecutar el algoritmo, ajustando la gráfica a f(x) = xlog(x), el resultado es el siguiente:



La gráfica es bastante parecida a la de QuickSort, con un ajuste bastante preciso respecto de la función f(x), pero con unos tiempos algo mayores que el anterior algoritmo

◆ Algoritmos de Búsqueda

Para la ejecución de estos algoritmos, hemos usado vectores de 500 a 100000 elementos, con 20 repeticiones por iteración

• Binary search

- Descripción del Algoritmo

Búsqueda en una lista ordenada: se sitúa la lectura en el centro de la lista y se comprueba si la clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sitúa en la mitad inferior o superior del elemento central de la lista.
El algoritmo termina o bien porque se ha encontrado la clave o porque el valor del índice izquierdo excede al derecho y el algoritmo devuelve el indicador de fallo, -1 (búsqueda no encontrada)

- Análisis Teórico

Para calcular el orden temporal del algoritmo de búsqueda binaria, vamos a utilizar la fórmula maestra para la reducción por división:

T(n) = c, si 1 <= n < b
T(n) = O(n^k log n), si a = b^k
T(n) = a · T(n/b) + f(n), si n >= b

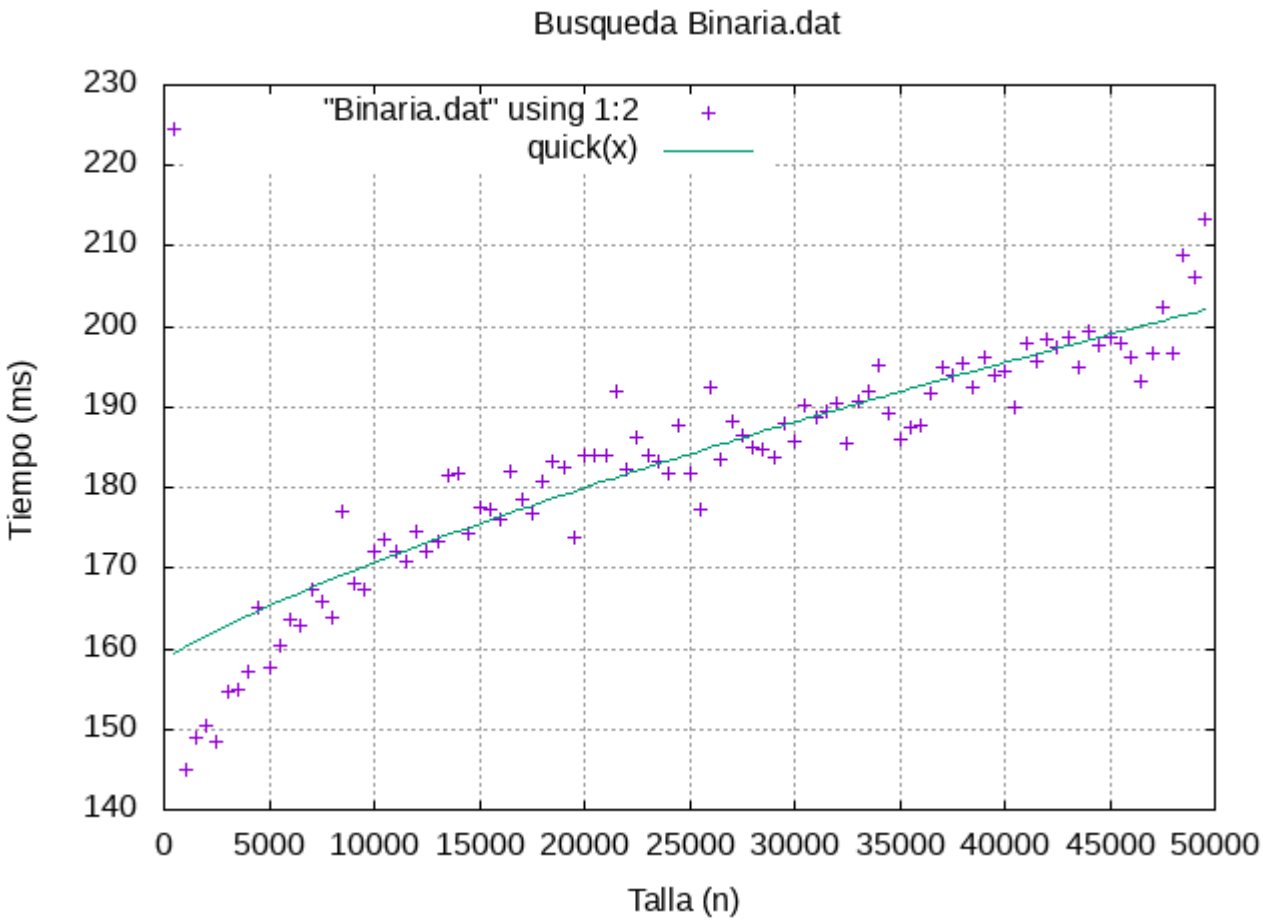
Teniendo que a = 1 (número de llamadas recursivas), b = 2 (factor de disminución del tamaño de los datos) y con f(n) = c (coste de combinación de los resultados y preparación de las llamadas), podemos obtener la complejidad.

En este caso k = 0 (f(n) = c * n^k -> c = c * n^k -> k = 0). Por lo tanto, y teniendo en cuenta que a = b^k (1 = 2^0), el orden del algoritmo es O(n^0 log n) = O(log n).

- Análisis Practico

Para analizar el algoritmo, usamos 100 repeticiones, con tamaños de 500 a 500000.

Para generar la gráfica, hemos ajustado los tiempos del algoritmo a f(x) = x*log(x), con el siguiente resultado:



Se aprecia el orden de complejidad logarítmico de este algoritmo, con unos tiempos bastante reducidos, y un ajuste bastante bueno a la curva teórica (aunque algo peor que en los algoritmos anteriores)

- **Lineal Search**

  - Descripción del Algoritmo

  Busca un dato (clave) key en una lista o array V accediendo a todas las posiciones hasta que se encuentre el elemento o se llegue al final del mismo (elemento no está)

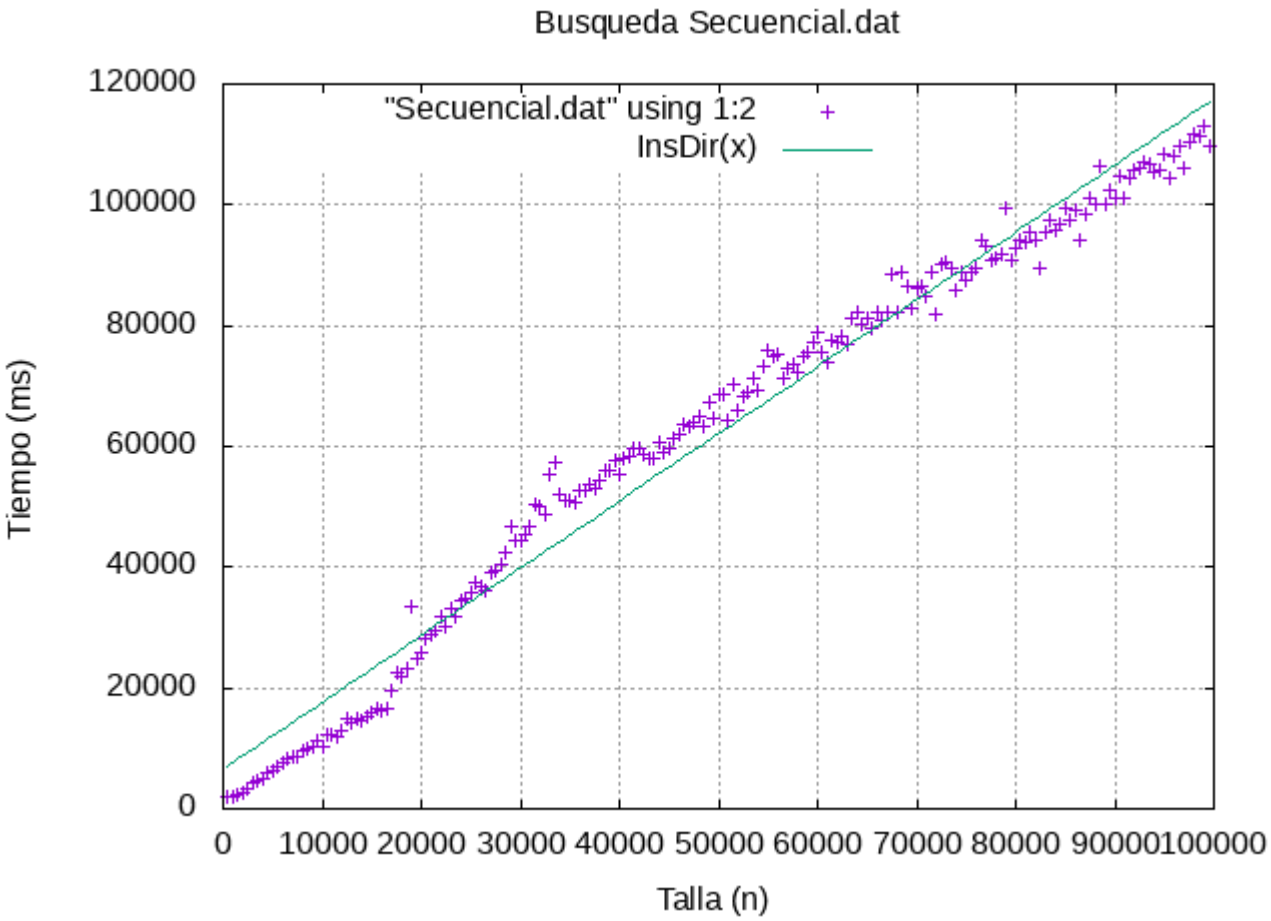
  - Análisis Teórico

  Consiste en un bucle for que recorre los n elementos de la estructura de datos hasta que encuentre el elemento buscado. Por esto, se puede afirmar que la complejidad temporal del algoritmo es de orden lineal O(n).

  - Análisis Practico

  Para analizar el algoritmo, usamos 100 repeticiones, con tamaños de 500 a 100000.

  Al ejecutar el algoritmo, ajustando la gráfica a f(x) = x, el resultado es el siguiente:



Se comprueba el orden lineal de dicho algoritmo , y los tiempos elevados que supone su ejecución, superando el minuto en los tamaños mayores

- **Hash Search**

  - Descripción del Algoritmo

  Este algoritmo se basa en el uso de tablas hash para almacenar los valores del vector

  La Tabla Hash es una estructura de datos especialmente diseñada para la implementación de DICCIONARIOS (operaciones Buscar, Insertar y Borrar en un tiempo “esperado” O(1)).

  Es una generalización de las tablas de acceso directo, en las que se asigna un indice a una clave, que no tiene porque ser numérica, mediante una función de dispersión llamada hash

  Esta tabla también permite representar elementos de un conjunto mucho mayor al tamaño de la tabla

  Para ello, la función hash asocia una posición a una clave dada. Según el tipo de función hash, puede darse que se asocien varias claves a una misma posición, lo cual se conoce como “colisión”

  Para resolver esta colisión podemos usar dos estrategias diferentes:

  - Direccionamiento Abierto: las claves asociadas a la misma posición se insertan en una lista una tras otra

  - Direccionamiento Cerrado: las claves asociadas a la misma posición son reubicadas en otra posición diferente

  - Implementación

  Para este algoritmo hemos implementado 2 tablas hash: una con dispersión abierta, y otra con dispersión cerrada.

  - **Tabla Hash Abierta**

  Para la tabla de dispersión abierta, hemos seguido la siguiente estrategia:

  - La tabla hash se ha implementado como un vector de listas, usando para ello las clases std::vector y std::list de la STL.

- Para calcular el hash, hemos usado la siguiente función:

```
h(key) = parte_entera(tamaño_tabla_abierta *
                    (key * inv_aurea - parte_entera(key * inv_aurea) ) );
```

Siendo inv_aurea = 2/(1 + sqrt(5)) (el inverso del número áureo)

Esta función asegura una gran dispersión, con lo cual se reducirán las probabilidades de colisión.

- En caso de colisión, el elemento se añadirá a la lista correspondiente a dicha posición obtenida por la hash.

- **Tabla Hash cerrada**

Para la tabla hash con dispersión cerrada, hemos seguido una estrategia similar, pero con ligeros cambios.

- Para calcular el hash, usamos 2 funciones, añadiendo a las mismas el número de intentos de inserción realizados para cada valor de la tabla.

■ Si el número de intentos es menor que el máximo establecido, usaremos la siguiente función, similar a la anterior, a la que se le añade el número de intentos:

```
h1(key, intentos) = parte_entera(tamaño_tabla_cerrada *
  (key * inv_aurea * intentos - parte_entera(key * inv_aurea * intentos)))
```

■ Si el número de intentos es superior al máximo, usamos una función mas simple basada en el módulo:

```
h2(key, intentos) = (key * intentos) mod tamaño_tabla_cerrada
```

- Si la posición dada por la función no esta libre, se busca en la anterior y posterior a la misma

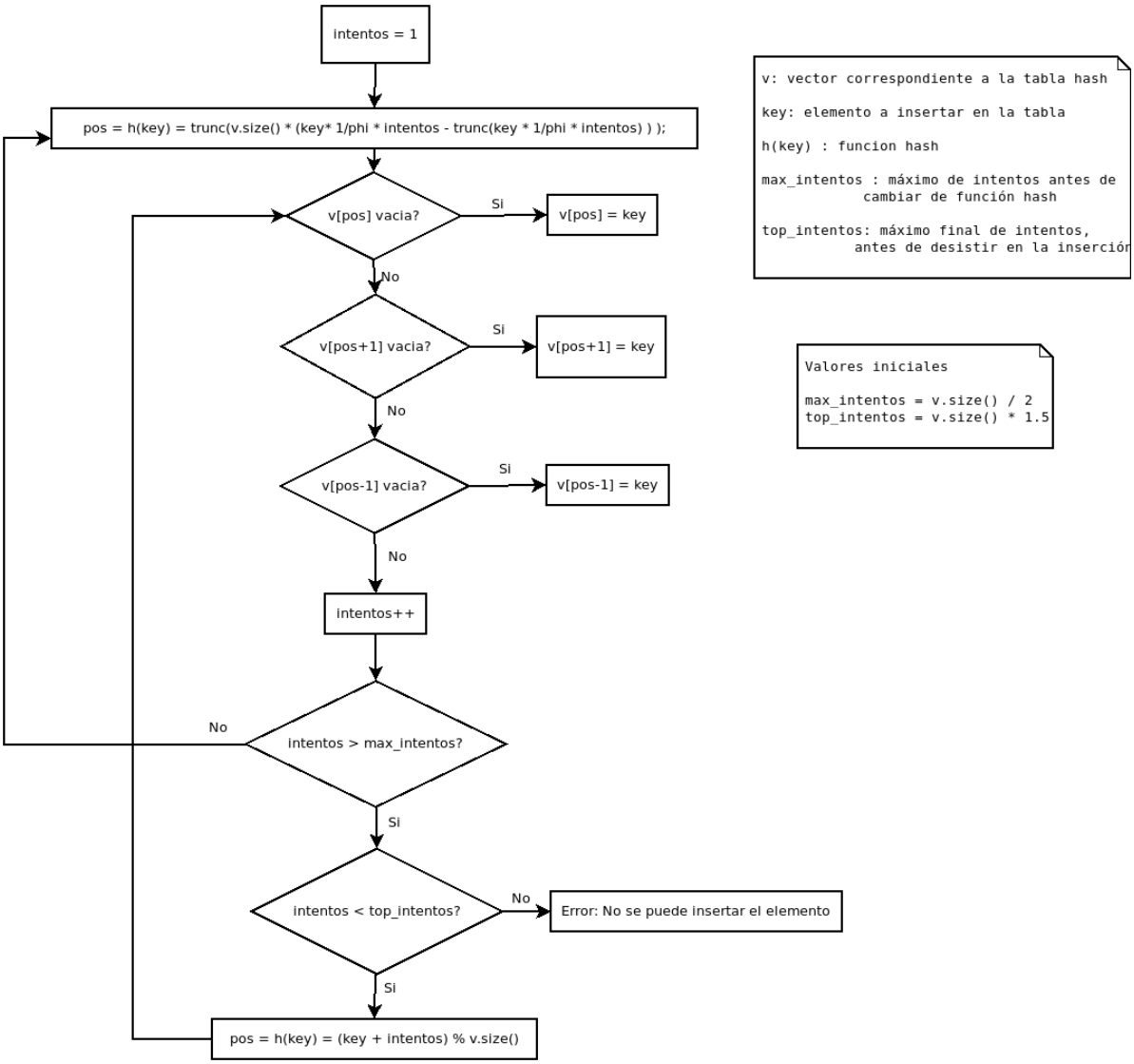
- Si sigue sin estar libre, se realiza un nuevo intento

- Si el número de intentos es superior al valor tope, se hace una búsqueda secuencial para encontrar la posición

El máximo de intentos está establecido a:
tamaño_tabla_cerrada / 2

El valor tope es 2 * tamaño_tabla_cerrada

El diagrama de flujo de este algoritmo es:



Para la búsqueda del valor se seguiría la misma metodología, cambiando la búsqueda de la posición vacía por la de dicho elemento.

- **Análisis Teórico**

La tabla hash asocia una posición de la tabla a una determinada clave. Esto, en condiciones ideales, haría que la búsqueda fuera coste constante, O(1).

Pero, al existir colisiones, es posible que el elemento no se pueda insertar en la posición dada por la hash, con lo cual el coste se elevaría debido a la búsqueda

■ Tabla Hash abierta

En la tabla hash con dispersión abierta, en caso de colisión, se inserta el elemento en una lista enlazada ubicada en la posición de la tabla dada por la hash.

En el caso peor, todos los elementos estarían en la misma posición, lo cual convertiría la búsqueda en una búsqueda lineal con O(n).

■ Tabla Hash cerrada

En la tabla hash con dispersión cerrada, en caso de colisión, se busca una nueva posición.

En nuestra implementación con rehashing, el número máximo de intentos de rehashing es 1,5 veces el tamaño del vector, así que el coste sería 1.5 * n = O(n)

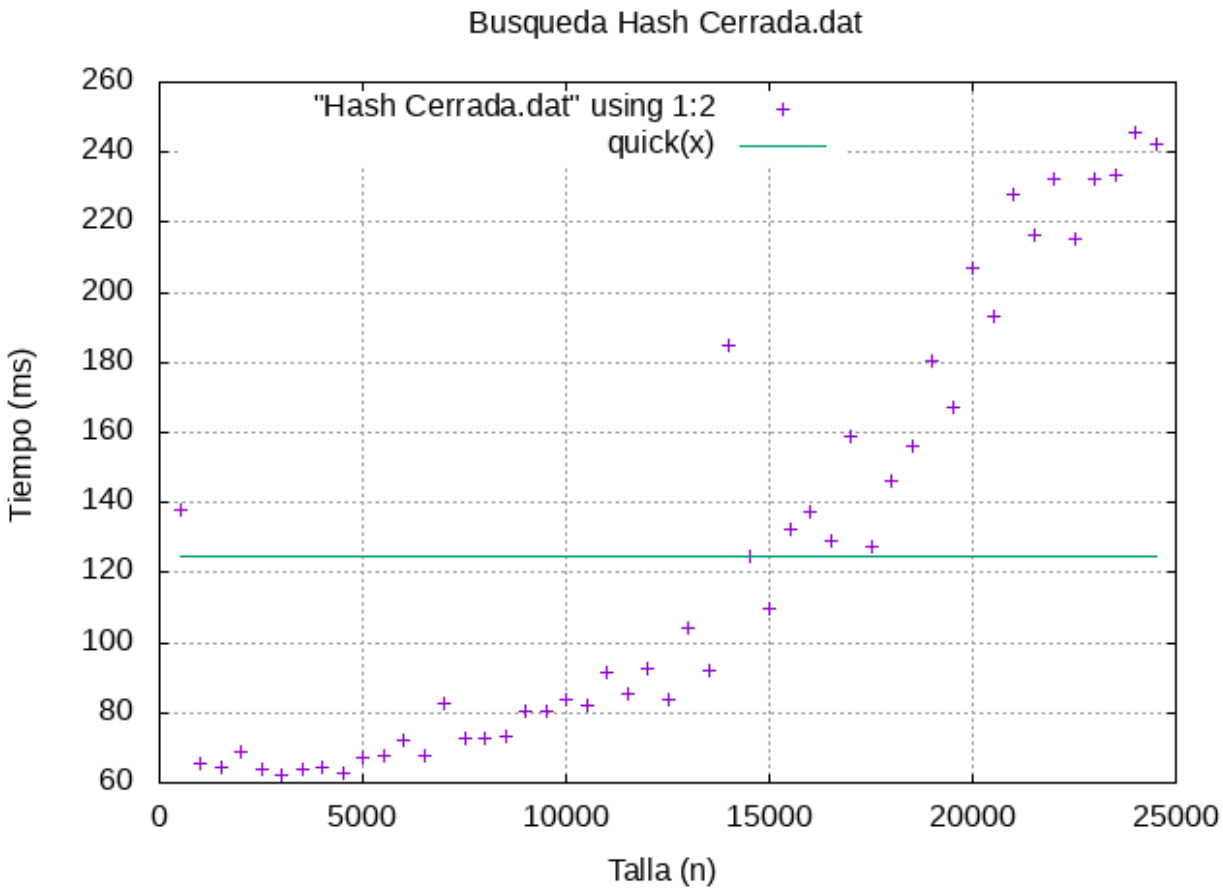
- **Análisis Practico**

- **Tabla Hash cerrada**

Los tiempos de inserción en la tabla hash cerrada son bastante altos, lo cual produce que la tabla hash consuma mucho tiempo en generarse.

Por esta razón, el tamaño máximo se ha reducido a 25000.

Al ejecutar el algoritmo, ajustando la gráfica a f(x) = a, el resultado es el siguiente:



Apreciamos que los tiempos de búsqueda en la tabla hash cerrada son bastante reducidos, apenas llegando a los 260 ms; aunque se aprecia que los incrementos de tiempos van siendo acentuándose a partir de los 15000 elementos, mostrando una cierta curvatura.

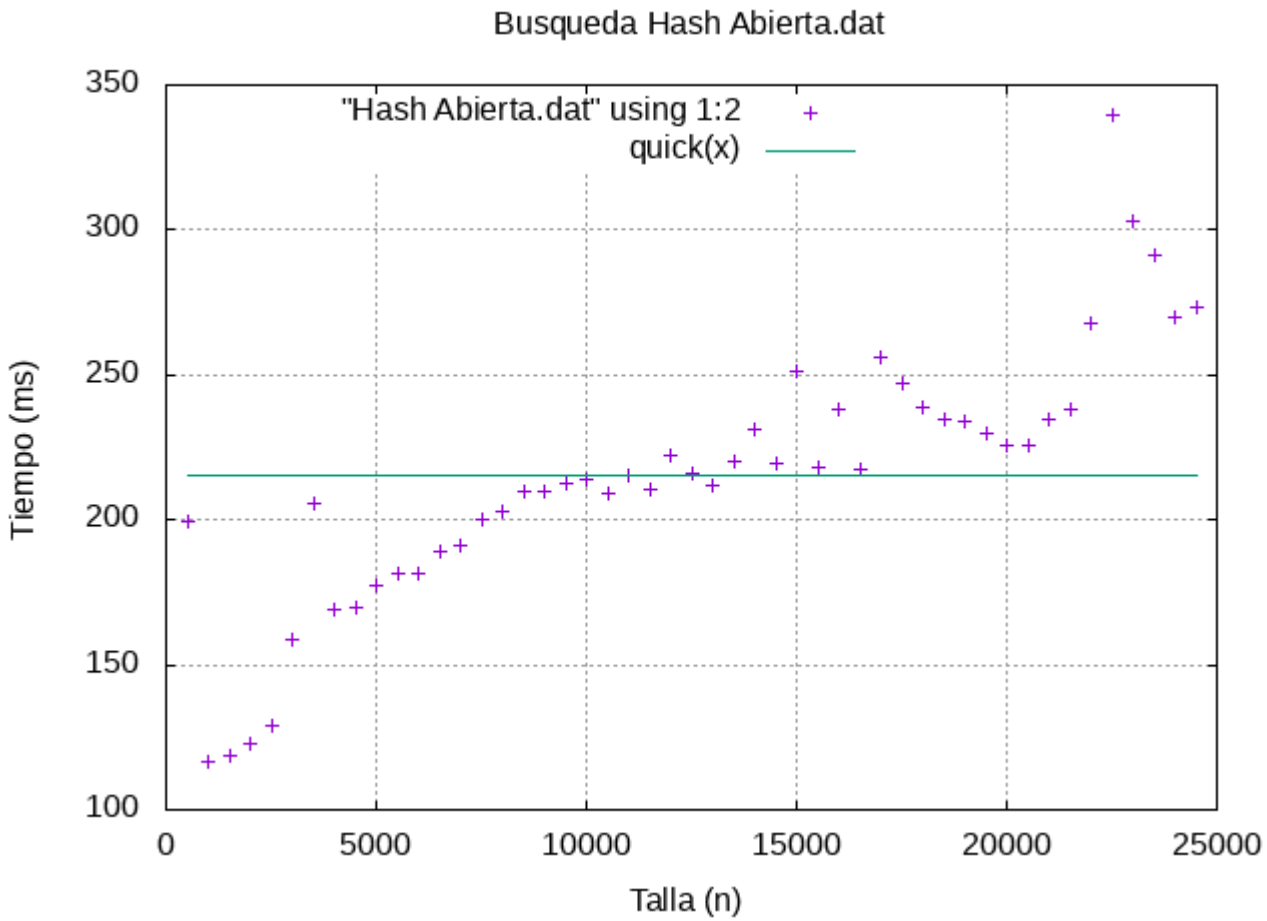
- **Tabla Hash Abierta**

En la búsqueda hash con dispersión abierta, se nota cierta irregularidad respecto a la de dispersión cerrada.

Aun así, esta búsqueda se mantiene muy regular respecto a otros algoritmos como el de búsqueda secuencial

Los tiempos de inserción en la tabla son mucho menores respecto a la hash cerrada, y algo menores en la búsqueda, lo cual se va acentuando conforme los tamaños son mayores.

Asimismo, los tiempos de búsqueda son notablemente mayores respecto a la de dispersión cerrada.



- **Interpolation search**

El algoritmo de “búsqueda por interpolación” es una modificación del algoritmo de búsqueda binaria de tal forma que el elemento seleccionado no sea el central sino aquel que se “correspondería” con el elemento buscado si la distribución de valores en el vector fuera uniforme.
Obtención de la posición siguiente a comprobar en el algoritmo de búsqueda por interpolación. La posición, p, se obtiene como resultado de suponer una distribución uniforme de los elementos del subvector A[i..j]

La función Búsqueda por interpolación usa el valor del elemento a buscar “x”, y los valores a los extremos (i < j) del subvector actual, para interpolar la próxima posición

$$\boldsymbol{p} = i + \frac{(j-i)(x-A[i])}{A[j]-A[i]} \text{ a comparar con } \boldsymbol{x}.$$

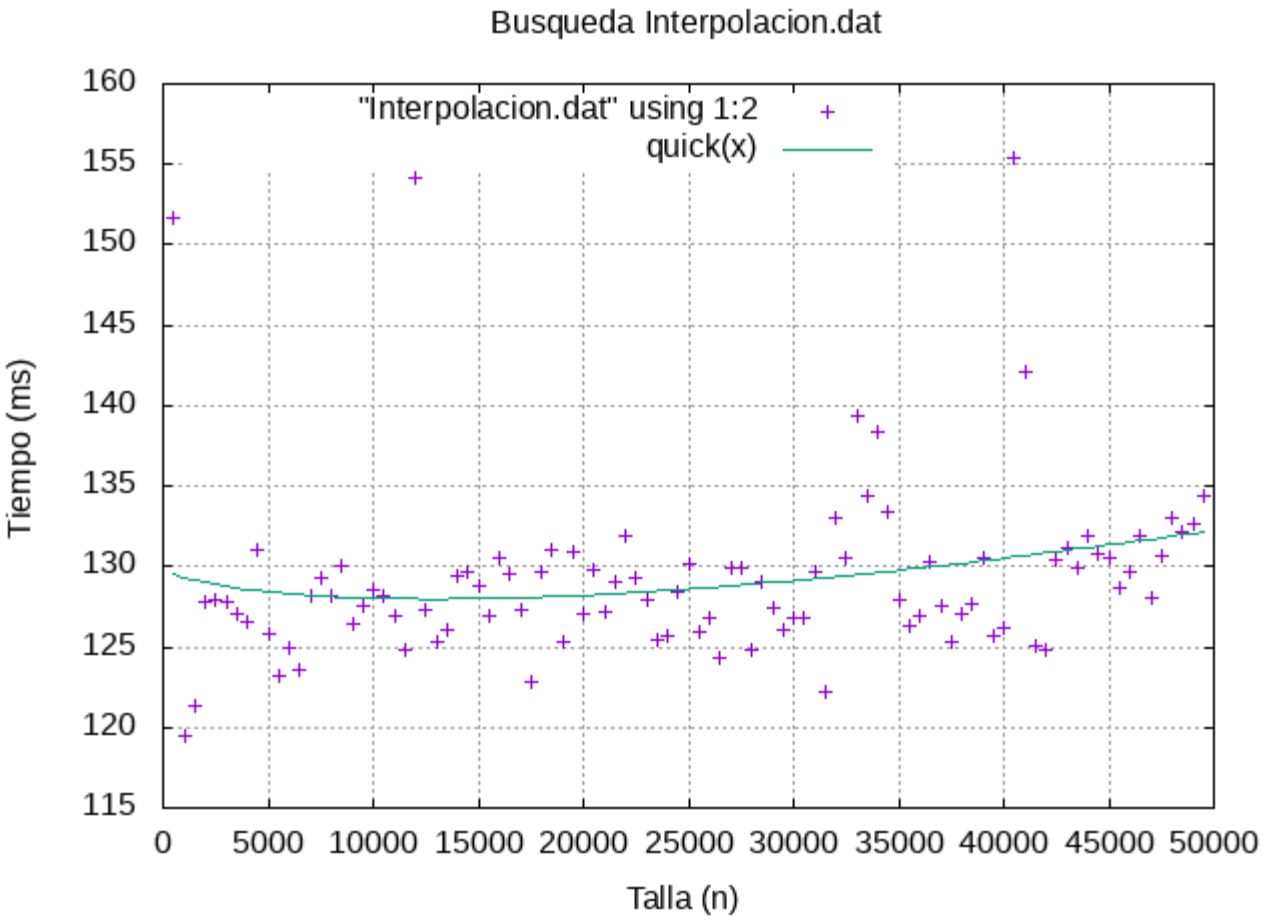
- **Análisis Teórico**

El coste de éste algoritmo es O(log log n)

- **Análisis Práctico**

Para el análisis práctico, consideramos el coste de éste algoritmo como O(log n)

Usamos tamaños de 500 a 50000, con 300 repeticiones por cada uno.



Vemos mucha irregularidad en los tiempos, de forma similar a la búsqueda binaria, pero con una pendiente mucho menor que esta.

3. Comparación de Algoritmos

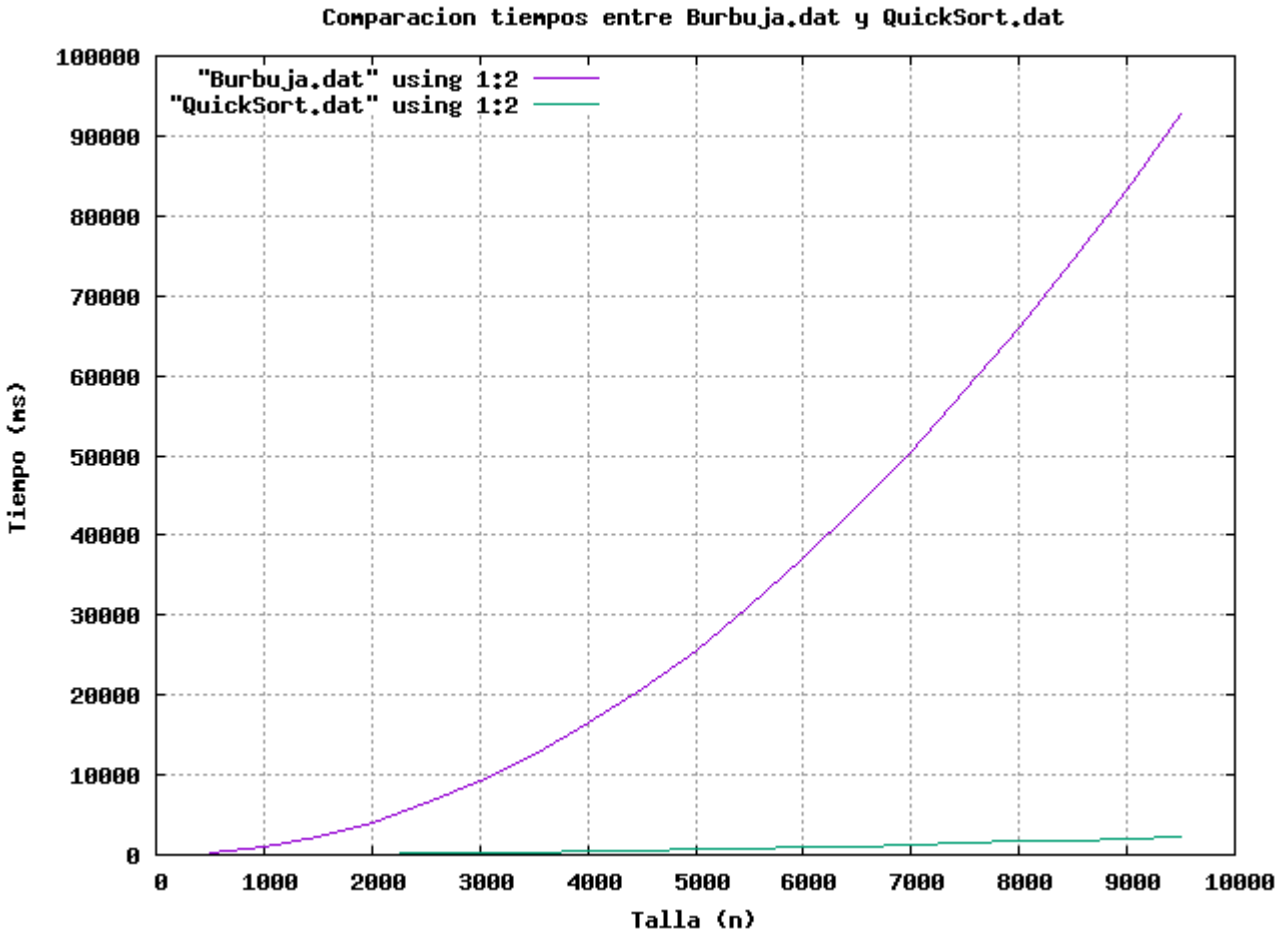
- Algoritmos de ordenación

- **BubbleSort vs QuickSort**

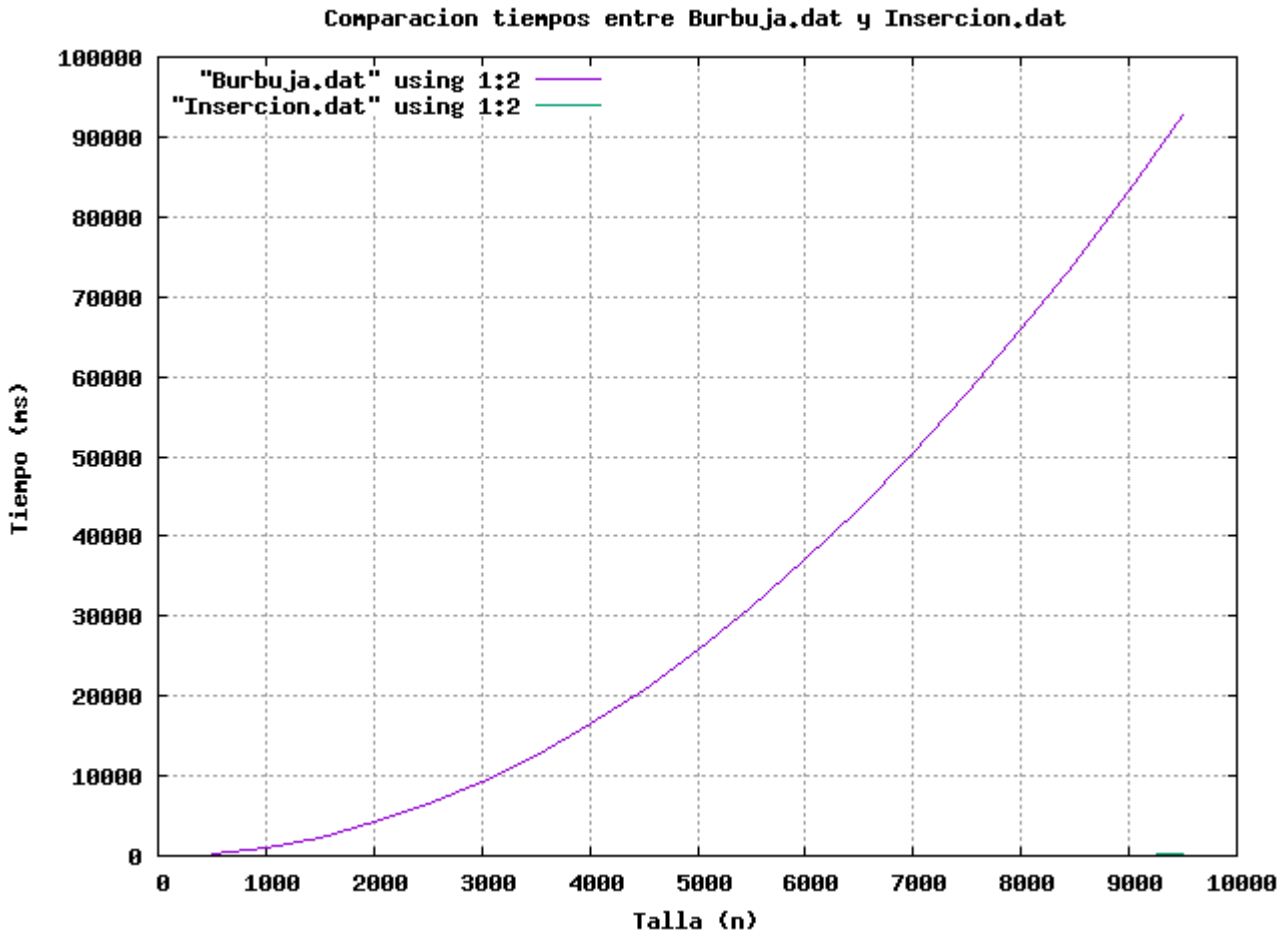
Para esta comparación, usaremos 20 repeticiones y vectores de 500 a 10000 elementos, incrementados de 500 en 500.

Al comparar QuickSort (O(log n)) con BubbleSort ($O(n^2)$) se aprecia una gran diferencia de tiempos.

Mientras en los vectores pequeños QuickSort tiene unos tiempos semejantes que BubbleSort,a medida que el tamaño va aumentando, la diferencia de tiempos se hace cada vez mayor; de tal forma que, en la última medida, la diferencia de tiempos es 10 veces superior en BubbleSort que en QuickSort



- **BubbleSort vs Insertion Sort**

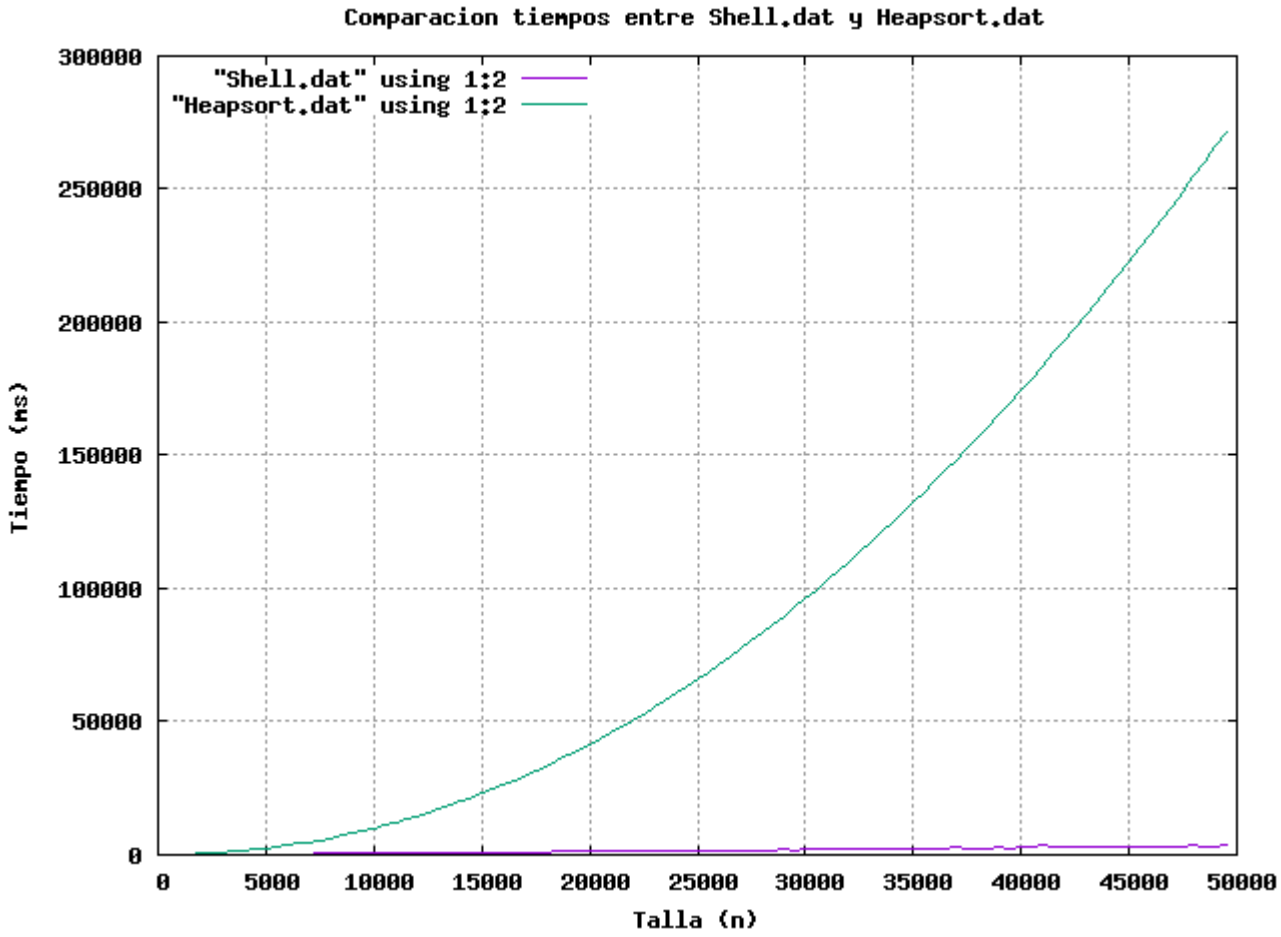


Al comparar BubbleSort con Insertion Sort, y a pesar de ser ambos algoritmos $O(n^2)$, se aprecia una gran diferencia de tiempos entre uno y otro, hasta el punto que apenas se distingue la curva del Insertion de la base de la gráfica

Se ve claramente la curva del BubbleSort con unos tiempos significativamente mayores que la del Insertion Sort

- **Shell Sort vs HeapSort**

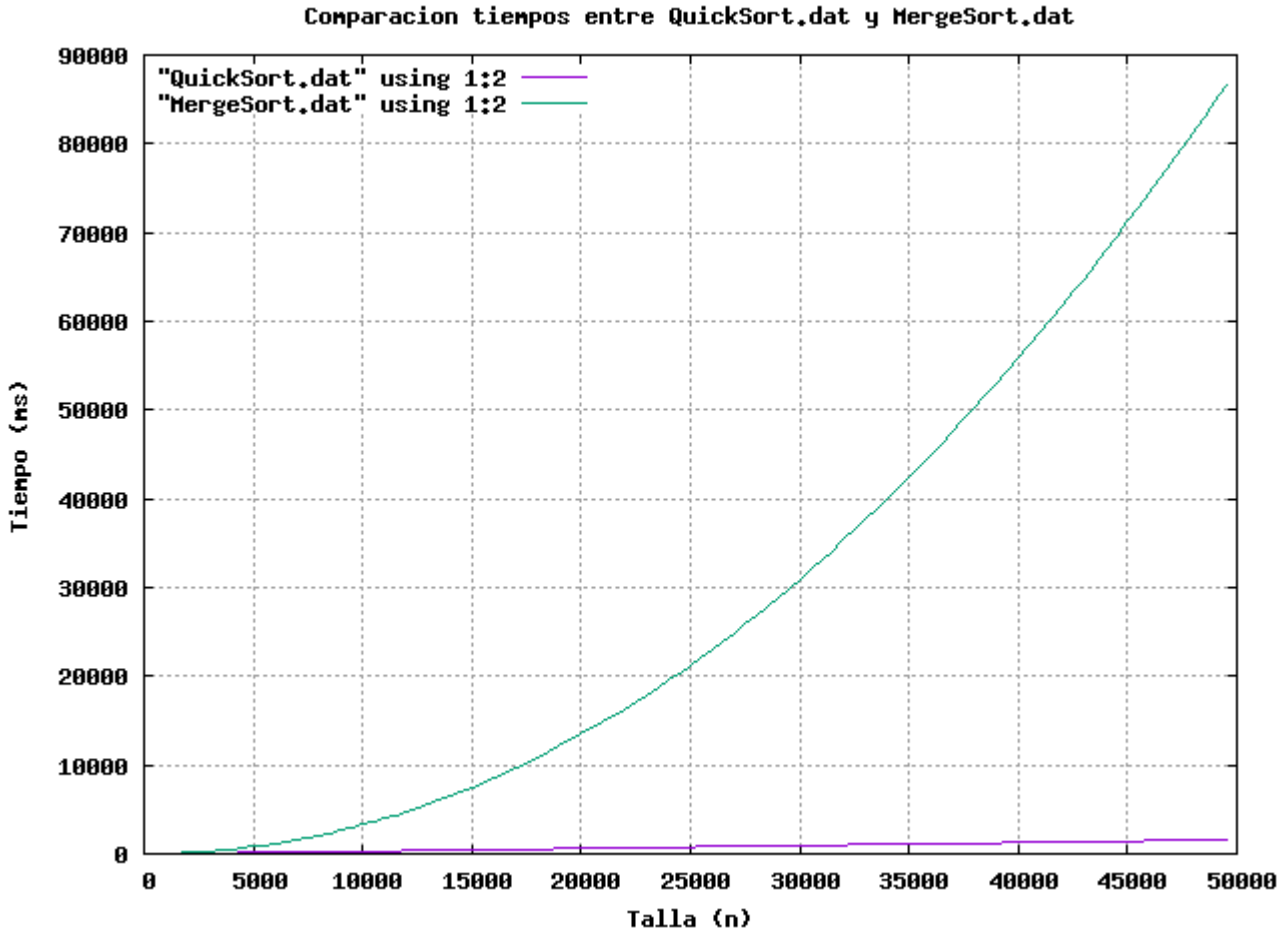
Para esta comparación, usaremos 100 repeticiones, con tamaños de 500 a 50000.



Al comparar Shell Sort con HeapSort, se nota gran ventaja del algoritmo Shell respecto al Heap
De nuevo, la curva de ShellSort apenas se distingue de la base de la gráfica, mientras que HeapSort muestra una curva bastante pronunciada que llega hasta los 300000 ms

- **QuickSort vs MergeSort**

Usaremos 100 repeticiones, con un rango de tamaños de 500 a 50000.



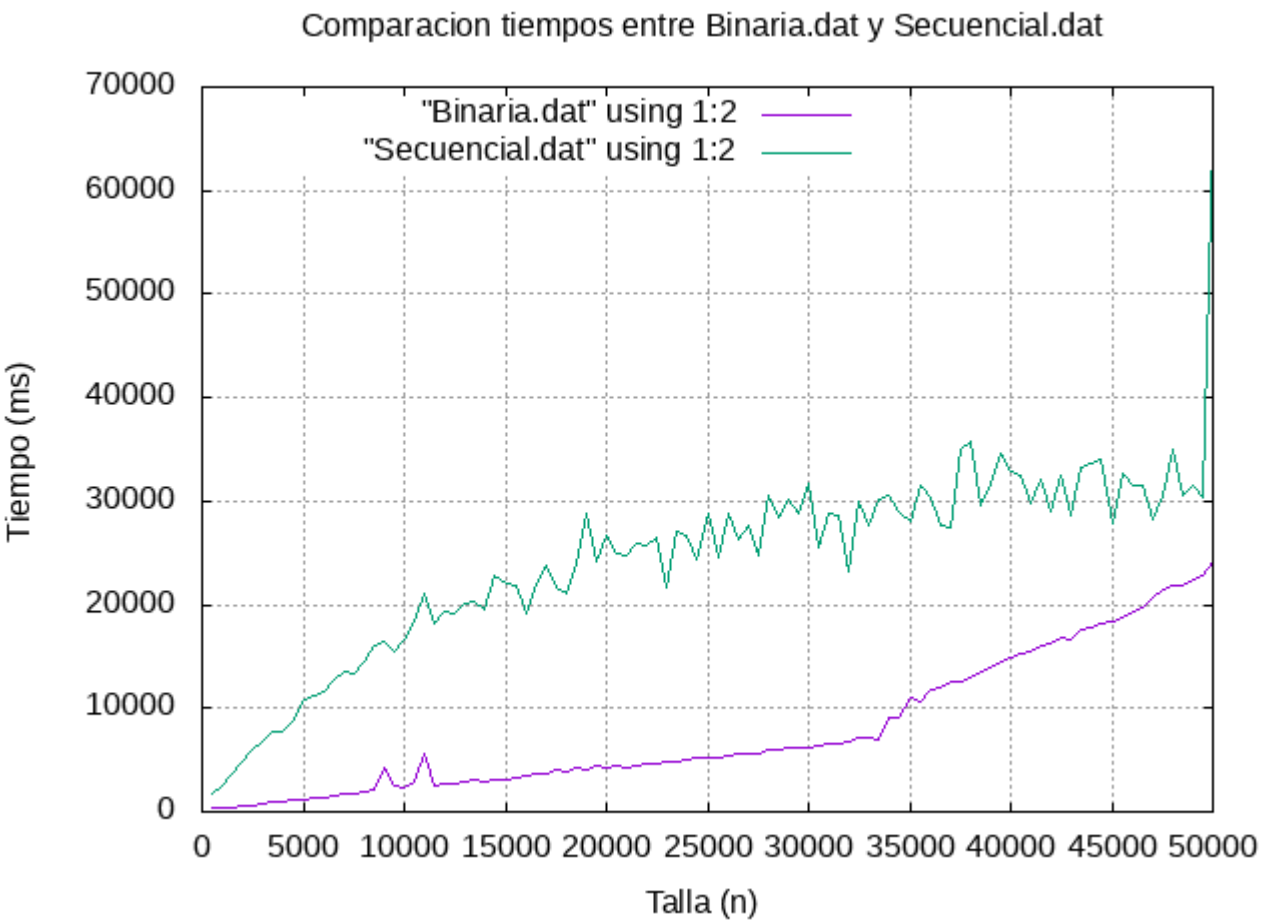
Llegamos a una de las comparaciones mas interesantes a nivel algorítmico
Ambos algoritmos son de O(nlog(n)), pero se nota una clara ventaja del QuickSort a nivel temporal

Se aprecia una curva bastante constante del QuickSort, a diferencia del MergeSort que tiene una curva mas pronunciada

- Algoritmos de Búsqueda

- **Lineal Search vs Binary Search**

Para hacer esta comparación, usamos 100 repeticiones con tamaños de 500 a 50000.

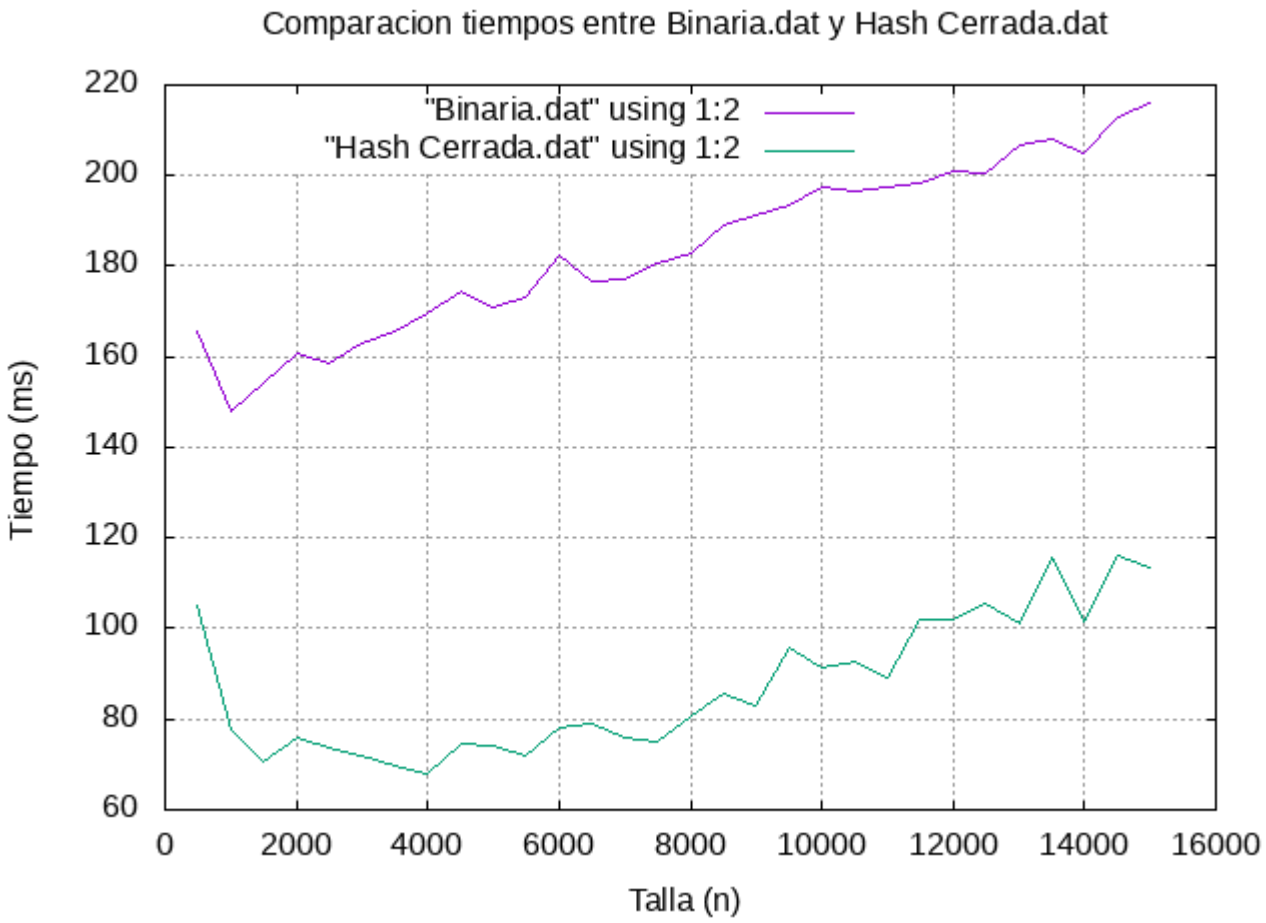


Al comparar la búsqueda binaria (O(log(n)) con la búsqueda secuencial (O(n)) se aprecia una gran diferencia entre los dos

Los tiempos de la búsqueda binaria son mucho mas reducidos, y su incremento es mucho mas regular que en la búsqueda secuencial, la cual muestra muchísimos picos y un tiempo bastante superior.

- **Binary Search vs Hash Closed Search**

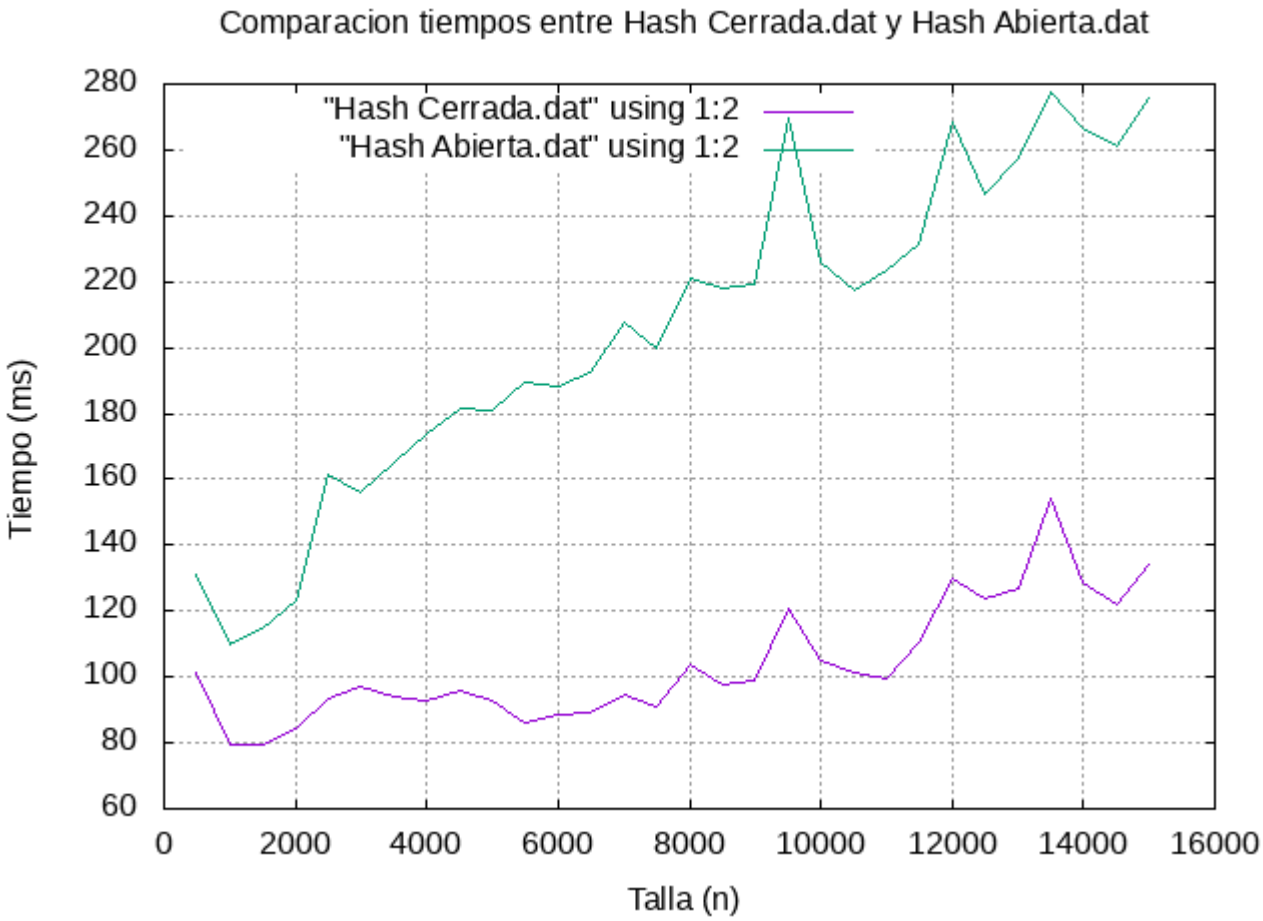
Para hacer esta comparación usaremos 200 repeticiones, y tamaños de 500 a 15000.



Se aprecian unos tiempos bastante menores en el Hash Closed Search, de mas del triple en algunos casos, y una pendiente menos pronunciada; diferencia la cual va aumentando conforme aumenta el tamaño

- **Hash Closed Search vs Hash Opened Search**

Para esta comparación, usaremos 200 repeticiones y vectores de 500 a 15000 elementos.



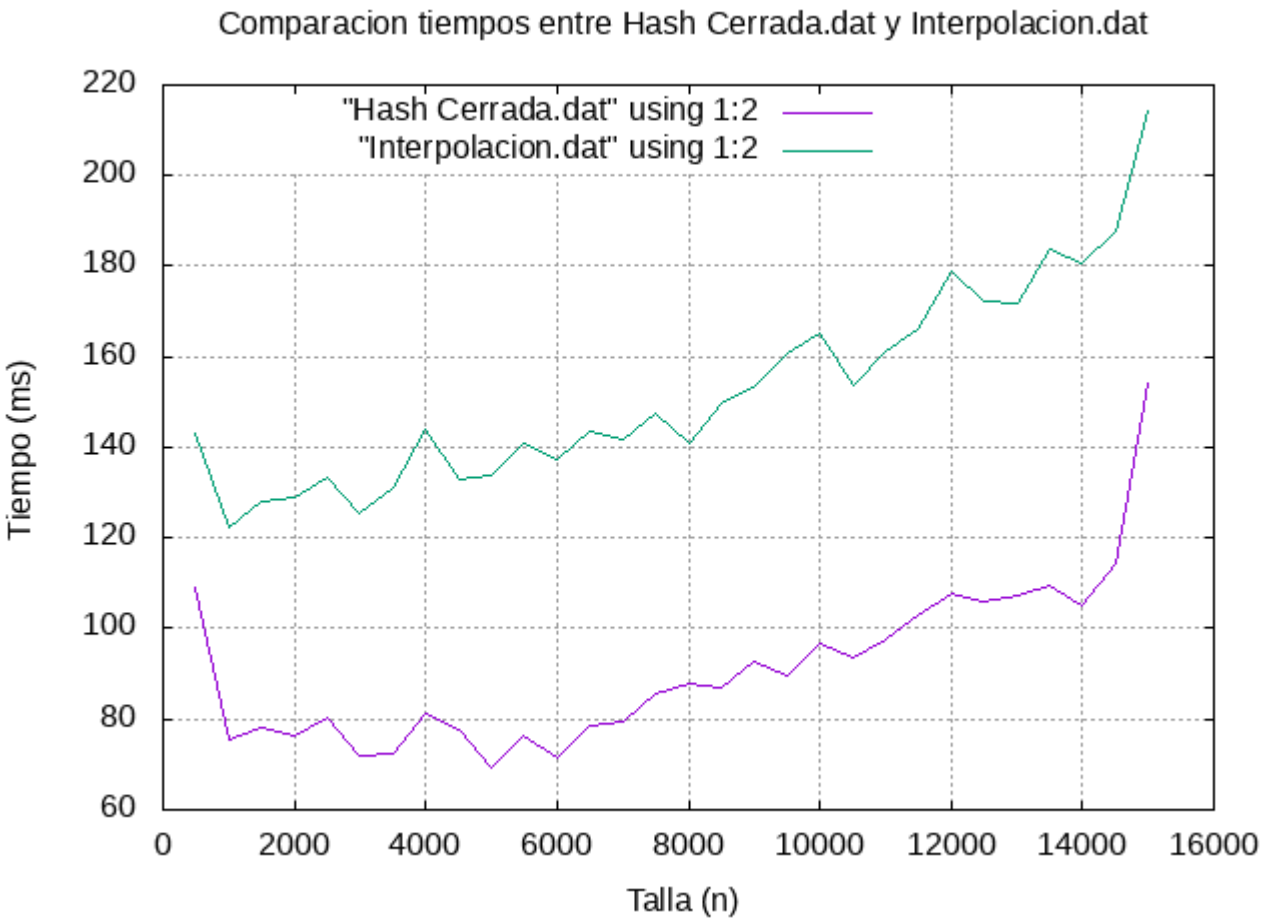
Al comparar los dos tipos de búsqueda hash, se aprecian bastante ventaja del closed hash respecto al opened hash, llegando la diferencia a mas del doble en algunos casos.

Pese a eso, en ambos casos la curva es muy similar, mostrando los mismos picos en ambos algoritmos.

También se aprecia, en general, unos tiempos bastante bajos respecto a los demás algoritmos.

- **Interpolation Search vs Hash Closed Search**

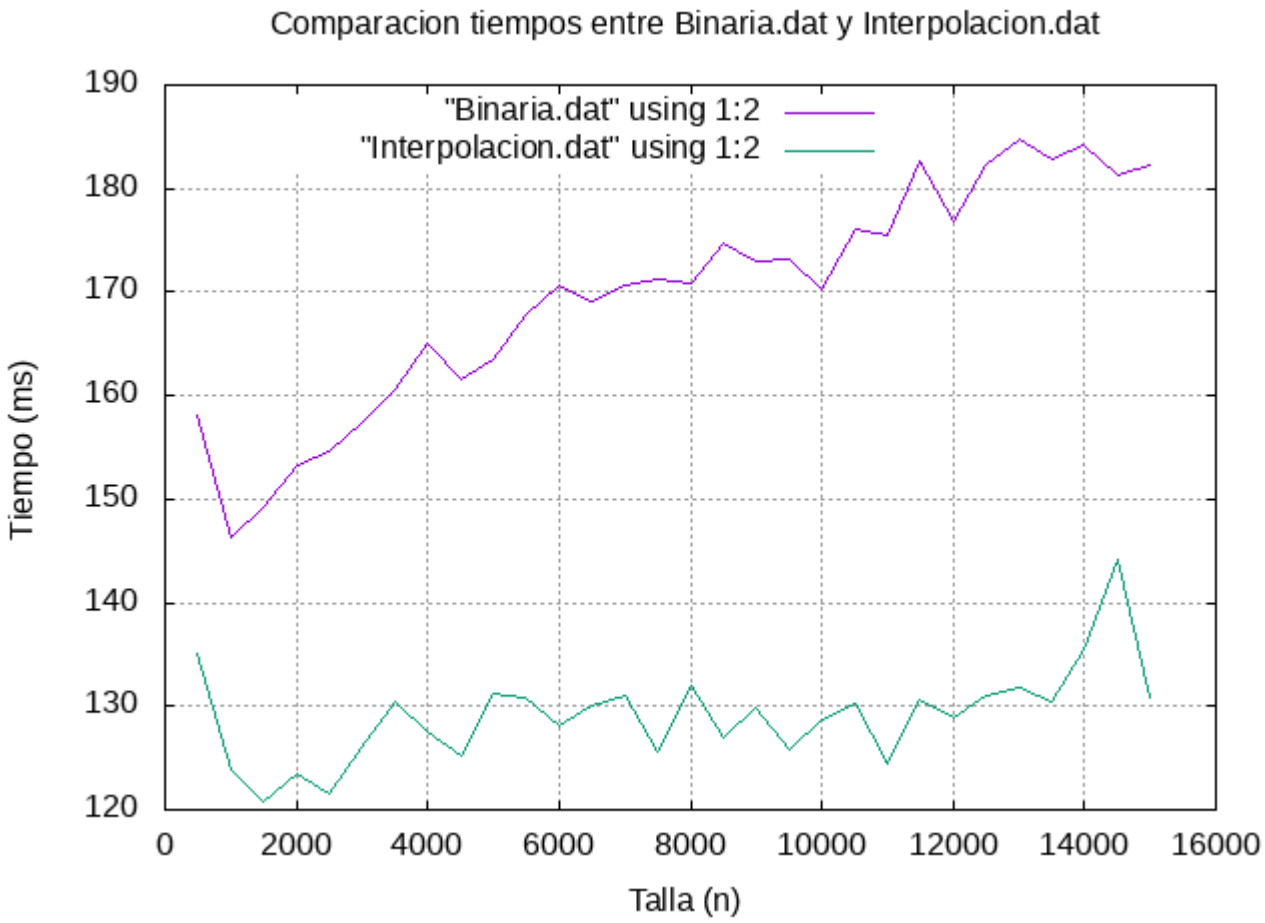
Para esta comparación, hemos usado tamaños de 500 a 15000, con 200 repeticiones.



Vemos una curva muy similar en ambos algoritmos, aunque con unos tiempos bastante mas reducidos en hash closed search, y mas irregularidad en interpolation search.
Aún así la diferencia de tiempos es menor que al comparar binary search vs hash closed search.

- **Binary Search vs Interpolation Search**

Para esta comparación, hemos usado 200 repeticiones, con tamaños de 500 a 15000 elementos.



Se aprecian unos tiempos significativamente menores en Interpolation Search, con una pendiente casi insignificante en comparación con el otro algoritmo; pero con una irregularidad mucho mayor.

3. Implementación de la Aplicación

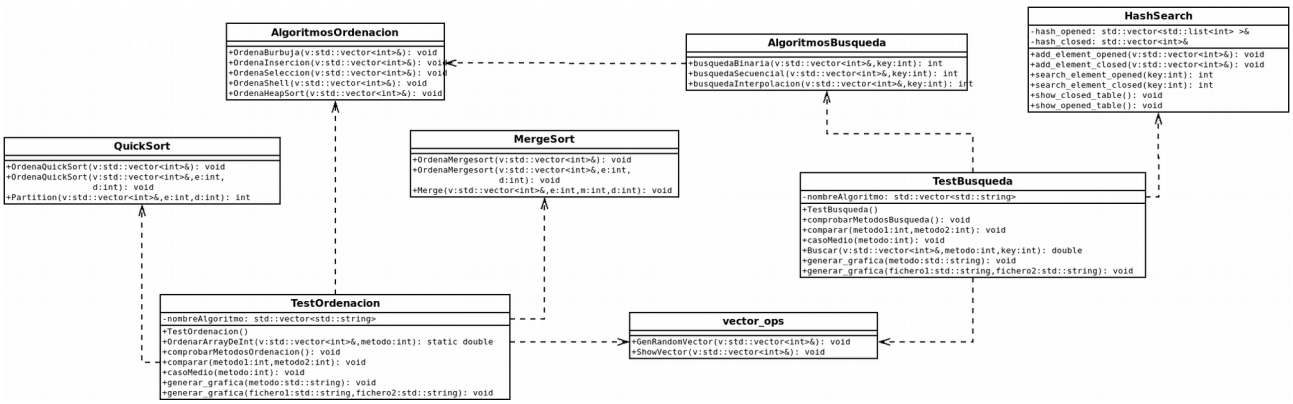
Para implementar la aplicación hemos usado el lenguaje de programación C++, usando la STL en gran parte de las estructuras de datos.

Los algoritmos se han implementado en las clases AlgoritmosOrdenacion y AlgoritmosBusqueda.
Los algoritmos QuickSort, MergeSort y HashSearch, debido a su complejidad, se han implementado en clases aparte

Para probar los algoritmos se han creado las clases TestBusqueda y TestOrdenacion.
En estas clases se han implementado métodos que invocan a los diferentes algoritmos y, usando este como base, se han implementado el resto de métodos para los casos mejor, peor y medio; para la demostración de funcionamiento de algoritmos y para la comparación entre algoritmos

Para generar los vectores, se ha creado una clase llamada vector_ops, que genera vectores aleatorios, y muestra sus contenidos por pantalla.

El diagrama de clases es el siguiente:



Los tiempos de ejecución de los diversos algoritmos se guardan en ficheros con extensión .dat que posteriormente se pasan a gnuplot para generar la gráfica

Para ello se han implementado dos métodos generargrafica() que toman los tiempos de los ficheros, y generan un script para ajustar la gráfica a su correspondiente función de complejidad

En la comparación de tiempos de algoritmos, no se aplica ningún ajuste, y el metodo solo genera el script con los ficheros de ambos algoritmos para que se muestren ambas curvas sobre la misma gráfica

Como detalle, se ha implementado la medición de tiempos tanto para sistemas Windows como para sistemas UNIX, usando directivas de preprocesador que compilen el código adecuado para cada plataforma

Para Windows, se han usado el método de winapi para medición de tiempos QueryPerformanceFrequency() y la clase Mtime

Para UNIX, se han usado los métodos estándar de C++11 high_resolution_clock::now() e interval.count()

Finalmente, para enlazar todos estos elementos, se ha implementando la clase Menu, que muestra una interfaz de texto con las diferentes opciones de análisis y comparación de ambos tipos de algoritmos

Esta interfaz se compone de varias pantallas, en la que se selecciona el tipo de algoritmo que se quiere analizar, y se le da a elegir entre una demostración de su ejecución de todos los algoritmos, un análisis de tiempos medios de un algoritmo a elegir, y la comparación de tiempos medios de dos algoritmos a elegir

Tras la ejecución de los tiempos medios, se da la opción de guardar el fichero de tiempos y de generar una gráfica con los datos de dicho fichero

Si el usuario selecciona ‘si’ en ambas opciones, la aplicación abre una ventana de gnuplot con la gráfica generada, la cual se mantiene abierta unos 10 segundos; tras lo cual vuelve al menú inicial o finaliza la aplicación

La gráfica generada se guarda como fichero de imagen en formato png, ubicado en el directorio principal del programa