Análisis de Algoritmos.

Tarea #1.

Profesor:

Joss Rayn Pecou Johnson.

Estudiantes:

Deiner Céspedes Molina.

Esteban Rodríguez Vargas.

Yader Siezar Chaves.

Semestre II, Año 2025.

Parte 1:

Explicación del algoritmo:

Decidimos usar el algoritmo de ordenamiento QuickSort como el principal para esta parte, en la cual los pasos son los siguientes:

- Se elige el último elemento como pivote.
- Se divide la lista en dos sublistas (menores y mayores).
- Se llama recursivamente a quicksort sobre cada sublista.
- Se concatenan los resultados.

Análisis de complejidad temporal:

$T(n) = 1 + 1 + nlog(n) + nlog(n) + 1 + 1 + 1$

$T(n) = 2nlog(n) + 5$

Mejor caso: $T(n) = \Omega$ (nlogn)

Peor caso: $T(n) = O(n2)$

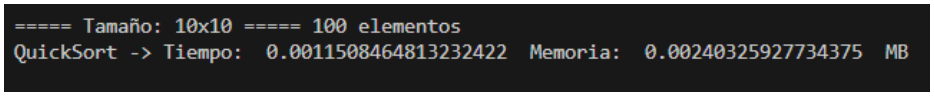
Caso promedio: $T(n) = \Theta$ (nlogn)

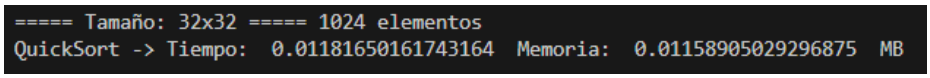
Podemos observar que el peor caso sucede cuando el pivote es el número mayor o menor de toda la lista, si eso pasa la partición queda desbalanceada, ya que una sublista queda con todos los elementos menos el pivote y la otra queda vacía, e iría así elemento a elemento hasta quedar con una lista de tamaño 1. Esto que provocaría una especie de bucle que haría el proceso de ordenamiento el doble de extenso de lo normal.

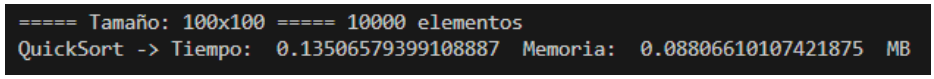
Resultados experimentales:

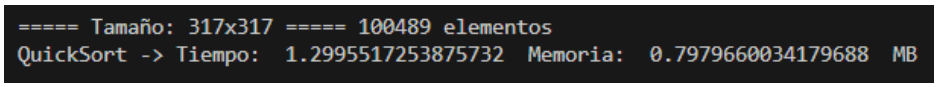
Se muestran los resultados de pruebas con nuestro algoritmo (QuickSort).

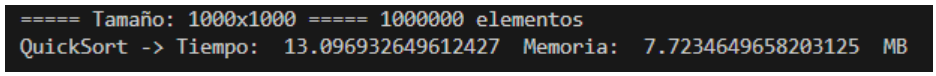
Imágenes:







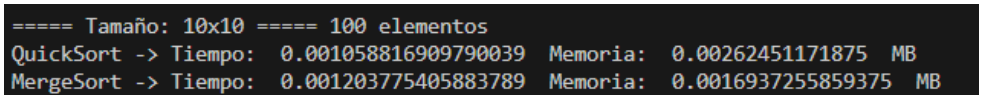


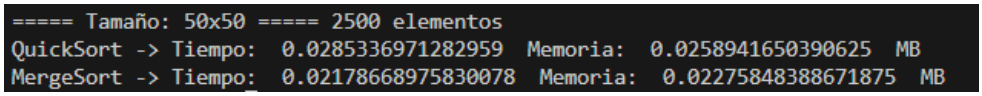


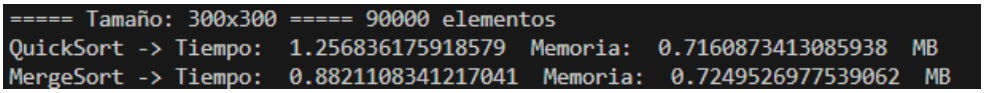
Comparación con otro algoritmo:

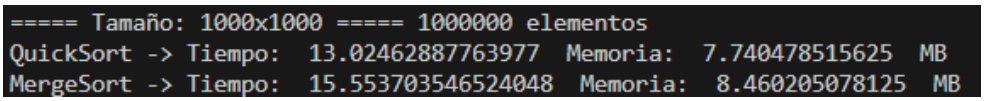
En las siguientes imágenes de observan los resultados de la comparación entre los métodos QuickSort y MergeSort en una matriz de tamaño nxn. Podemos observar cómo los resultados son muy similares en la mayoría de los casos tanto en la memoria utilizada como en el tiempo de ejecución.

Imágenes:









Parte 2:

Explicación del algoritmo:

El algoritmo utilizado en este apartado fue el ya existente Búsqueda Hash, este algoritmo funciona de la siguiente manera:

- Se crea una lista con todas las posiciones de los elementos de la matriz.
- Se revisa si el elemento está en la lista y se retorna el resultado.

Análisis de complejidad temporal:

$T(n) = 1 + 1 + n*n + 1 + 1 + 1 + 1 + 1 + 1$

$T(n) = n^2 + 8$

Peor caso: $T(n) = O(n^2)$

Mejor caso: $T(n) = \Omega(n^2)$

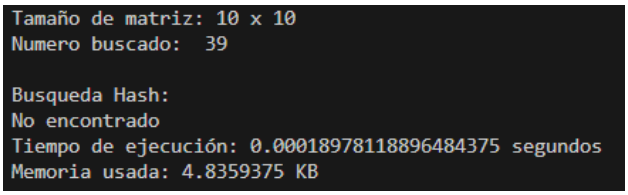
Caso promedio: $T(n) = \theta(n^2)$

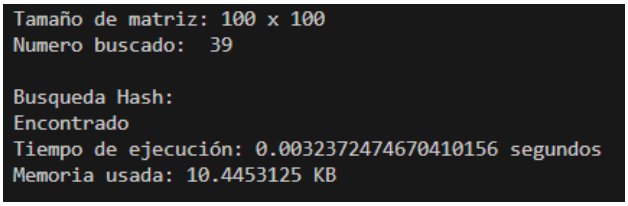
Podemos observar que nuestro algoritmo siempre tendrá un orden de n^2, esto se debe a que la matriz siempre se va a recorrer de inicio a fin, lo que hace que a mayor número de elementos aumente el tiempo y el espacio que se consume en la ejecución.

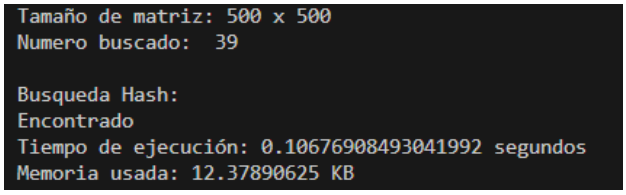
Resultados experimentales:

Se muestran los resultados de pruebas con nuestro algoritmo (Busqueda Hash).

Imágenes:







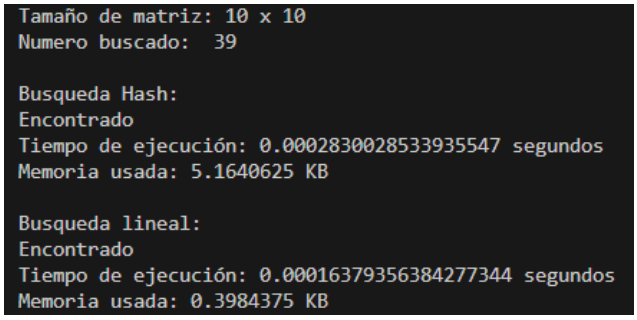
Comparación con otro algoritmo:

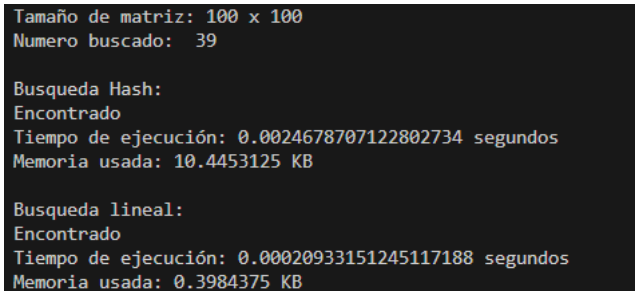
Al realizar comparaciones con el algoritmo de búsqueda lineal logramos observar los siguientes puntos:

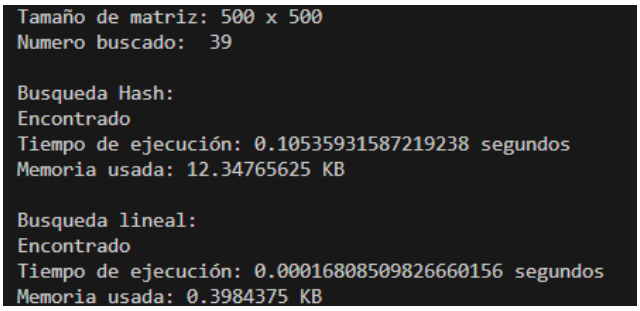
1. La memoria que utiliza la búsqueda Hash aumenta basada en la entrada que se le da, mientras que la búsqueda lineal siempre utiliza la misma cantidad de memoria, ya que no guarda datos en la misma como lo hace Hash.

2. En todos los casos probados el algoritmo de búsqueda lineal requiere menos tiempo de ejecución que la búsqueda Hash, esto se debe a que la búsqueda lineal no siempre recorre la matriz completa mientras que nuestro algoritmo lo hace siempre.

Estos puntos se pueden evidenciar en las siguientes imágenes:







Enlace del repositorio en el que se desarrolla la tarea:
https://github.com/ysiezar21/Tarea_corta_-1