**Planeación y realización:**

1. **Delimitación del problema:**

Se debe realizar un experimento sobre un arreglo de valores teniendo en cuenta algunos factores controlables, con el fin, de identificar qué algoritmo se desempeña mejor según las diferentes condiciones.

1. **Variables de respuesta y verificación de una medición confiable:**

Las variables de respuesta son el algoritmo más eficiente con datos pequeños y el algoritmo más eficiente con datos grandes. Y, con el fin de que la medición sea confiable, se repiten las pruebas en diferentes computadores.

1. **Factores de estudio:**

Se escogieron como factores de estudio los algoritmos de ordenamiento, el tamaño del arreglo, el estado de los valores en el arreglo y la RAM del computador donde se ejecuta el algoritmo.

1. **Niveles de cada factor y diseños experimentales:**

En este problema se tienen cuatro factores de estudio. El primero es el algoritmo de ordenamiento, el cual consiste de dos niveles (insertion sort y quicksort), el segundo es el tamaño del arreglo, este tiene cuatro niveles (10^2, 10^3, 10^4 y 10^6). El tercer factor de estudio consiste de tres niveles (8GB, 12GB Y 16GB) y es la RAM del computador donde se ejecuta el programa. Finalmente se tiene el estado de los valores en el arreglo con tres niveles (ascendente, descendente y aleatorio).

1. **Planeación y organización del diseño experimental:**
2. **Realización del experimento:**

**Etapa de análisis:** De acuerdo a las complejidades temporales de cada uno de los algoritmos de ordenamiento decidimos realizar este experimento para poder confirmar que el algoritmo “insertion sort” funciona de forma adecuada para casos pequeños, pero al aumentar el tamaño de los datos en el arreglo su tiempo de ejecución crece muy rápido.

**Etapa de interpretación:**

* De acuerdo al tiempo promedio de ejecución de cada uno de los algoritmos, pudimos analizar que a pesar de que hay diferencias con arreglos de pocos datos (diferencia que favorece al insertion sort), las diferencias más notables se ven cuando los arreglos son más grandes en datos.
* El “insertion sort” se comporta de una forma estable con arreglos de pocos datos, teniendo muy buenos tiempos de ejecución, a medida que los arreglos van creciendo en tamaño los tiempos de ejecución

comienzan a crecer exponencialmente, tal como lo expresa su complejidad temporal.

* El “Quick sort” es un algoritmo de ordenamiento estable y óptimo en casos de arreglos pequeños, cuando estos arreglos comienzan a aumentar de tamaño el algoritmo se mantiene estable sin aumentar sus tiempos de forma exponencial, como lo expresa su complejidad temporal.

**Etapa de control y conclusiones:** Podemos concluir que el insertion sort no es apto para el ordenamiento de grandes datos, por el contrario, el quick sort funciona de una forma óptima en la mayoría de casos generales, ya se de grandes datos o pocos datos.

**Casos de prueba:**

|  |  |  |  |
| --- | --- | --- | --- |
| Method Name | Class of Method | inputs | outputs |
| TestQuick | QuickTest | Un arreglo de dos elementos | Un arreglo de dos elementos ordenado por QuickSort |
| Test100asc | QuickTest | Un arreglo de 10^2 elementos ordenados ascendentemente. | Un arreglo de 10^2 elementos ordenado por QuickSort. |
| Test1000asc | QuickTest | Un arreglo de 10^3 elementos ordenados ascendentemente. | Un arreglo de 10^3 elementos ordenado por QuickSort. |
| Test10000asc | QuickTest | Un arreglo de 10^4 elementos ordenados ascendentemente. | Un arreglo de 10^4 elementos ordenado por QuickSort. |
| Test1000000asc | QuickTest | Un arreglo de 10^6 elementos ordenados ascendentemente. | Un arreglo de 10^6 elementos ordenado por QuickSort. |
| Test100desc | QuickTest | Un arreglo de 10^2 elementos ordenados descendentemente. | Un arreglo de 10^2 elementos ordenado por QuickSort. |
| Test1000desc | QuickTest | Un arreglo de 10^3 elementos ordenados descendentemente. | Un arreglo de 10^3 elementos ordenado por QuickSort. |
| Test10000desc | QuickTest | Un arreglo de 10^4 elementos ordenados descendentemente. | Un arreglo de 10^4 elementos ordenado por QuickSort. |
| Test1000000desc | QuickTest | Un arreglo de 10^6 elementos ordenados descendentemente. | Un arreglo de 10^6 elementos ordenado por QuickSort. |
| Test100rand | QuickTest | Un arreglo de 10^2 elementos al azar. | Un arreglo de 10^2 elementos ordenado por QuickSort. |
| Test1000rand | QuickTest | Un arreglo de 10^3 elementos al azar. | Un arreglo de 10^3 elementos ordenado por QuickSort. |
| Test10000rand | QuickTest | Un arreglo de 10^4 elementos al azar. | Un arreglo de 10^4 elementos ordenado por QuickSort. |
| Test1000000rand | QuickTest | Un arreglo de 10^6 elementos al azar. | Un arreglo de 10^6 elementos ordenado por QuickSort. |
| TestInsertion | InsertionTest | Un arreglo de dos elementos | Un arreglo de dos elementos ordenado por InsertionSort |
| Test100asc | InsertionTest | Un arreglo de 10^2 elementos ordenados ascendentemente. | Un arreglo de 10^2 elementos ordenado por InsertionSort. |
| Test1000asc | InsertionTest | Un arreglo de 10^3 elementos ordenados ascendentemente. | Un arreglo de 10^3 elementos ordenado por InsertionSort. |
| Test10000asc | InsertionTest | Un arreglo de 10^4 elementos ordenados ascendentemente. | Un arreglo de 10^4 elementos ordenado por InsertionSort. |
| Test1000000asc | InsertionTest | Un arreglo de 10^6 elementos ordenados ascendentemente. | Un arreglo de 10^6 elementos ordenado por InsertionSort. |
| Test100desc | InsertionTest | Un arreglo de 10^2 elementos ordenados descendentemente. | Un arreglo de 10^2 elementos ordenado por InsertionSort. |
| Test1000desc | InsertionTest | Un arreglo de 10^3 elementos ordenados descendentemente. | Un arreglo de 10^3 elementos ordenado por InsertionSort. |
| Test10000desc | InsertionTest | Un arreglo de 10^4 elementos ordenados descendentemente. | Un arreglo de 10^4 elementos ordenado por InsertionSort. |
| Test1000000desc | InsertionTest | Un arreglo de 10^6 elementos ordenados descendentemente. | Un arreglo de 10^6 elementos ordenado por InsertionSort. |
| Test100rand | InsertionTest | Un arreglo de 10^2 elementos al azar. | Un arreglo de 10^2 elementos ordenado por InsertionSort. |
| Test1000rand | InsertionTest | Un arreglo de 10^3 elementos al azar. | Un arreglo de 10^3 elementos ordenado por InsertionSort. |
| Test10000rand | InsertionTest | Un arreglo de 10^4 elementos al azar. | Un arreglo de 10^4 elementos ordenado por InsertionSort. |
| Test1000000rand | InsertionTest | Un arreglo de 10^6 elementos al azar. | Un arreglo de 10^6 elementos ordenado por InsertionSort. |

**Complejidad Temporal**

|  |  |  |
| --- | --- | --- |
| **Línea** | **Instrucción** | **# veces que se ejecuta** |
|  | QuickSort(array, low, high) |  |
| 1 | if low < high | n - 1 |
| 2 | long partitionIndex = Partition(array, low, high); | 2n+1 |
| 3 | quickSort(array, low, partitionIndex - 1); | T(n/2) |
| 4 | quickSort(array, partitionIndex + 1, high); | T(n/2) |

**Mejor:** *T (n) = 2T (n/2)   + 3n = 2T (n/2)   + O(n)*

k = loglog n

Cuando log n >1, la parte derecha de la suma crecerá más rápido que n, por lo que

**T(n) = O(n Log n)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Línea** | **Instrucción** | **# veces que se ejecuta** | |
|  | Partition(array,low,high) |  |  |
| 1 | long pivot = array[high]; | 1 | 1 |
| 2 | long lowIndex = (low - 1); | 1 | 1 |
| 3 | for (long j = low; j < high; j++) | n+1 | n+1 |
| 4 | if (array[j] <= pivot) | n | n |
| 5 | lowIndex++; | 0 | n |
| 6 | array[lowIndex] = array[j]; | 0 | n |
| 7 | array[j] = temp; | 0 | n |
| 8 | long temp1 = array[lowIndex + 1]; | 1 | 1 |
|  | array[lowIndex + 1] = array[high]; | 1 | 1 |
|  | array[high] = temp1; | 1 | 1 |
|  | return lowIndex + 1; | 1 | 1 |

**Mejor:** T(n) = 2n +7 = O(n)

**Peor:** T(n) = 5n +7 = O(n)

|  |  |  |
| --- | --- | --- |
| **Línea** | **Instrucción** | **# veces que se ejecuta** |
|  | insSort(long[] arr) |  |
| 1 | long n = arr.Length; | 1 |
| 2 | for (long i = 1; i < n; ++i) | n+1 |
| 3 | long key = arr[i]; | n |
| 4 | long j = i - 1; | n |
| 5 | while (j >= 0 && arr[j] > key) | 1 |
| 6 | arr[j + 1] = arr[j]; | 0 |
| 7 | j = j - 1; | 0 |
| 8 | arr[j + 1] = key; | n |

**Mejor:** T(n) = 4n +3 = O(n)