**DISEÑO DE EXPERIMENTOS**

**ALGORITMOS DE ORDENAMIENTO**

1. **Planeación y Realización:**
2. **Diseño del experimento**

En matemáticas y computación, un algoritmo de ordenamiento consiste en una serie de pasos que ponen los elementos de un arreglo en una secuencia dada por un criterio específico de orden. Estos elementos pueden ser de cualquier naturaleza siempre y cuando alguna de sus características permite definir un criterio de comparación. Adicionalmente, es importante tener en cuenta la estrategia utilizada a la hora de realizar ordenamientos en computador, pues diferentes estrategias implican diferentes cantidades de pasos a realizar y cuando la entrada es lo suficientemente grande distintas cantidades de pasos por hacer resultan en grandes diferencias en los tiempo de ejecución. Para el caso de estudio, se trabajara con los algoritmos de ordenamiento CountingSort y PigeonHoleSort utilizando arreglos de números enteros con el fin de analizar cómo afectan la estrategia y los factores de ejecución al tiempo de ordenamiento.

**1. 1 Objetivo del experimento**

Determinar cómo afectan el tamaño del arreglo, el poder de procesamiento, y el lenguaje de programación en el cual la función es codificada al tiempo de ejecución de los algoritmos de ordenamiento CountingSort y PigeonHoleSort.

**1.2 Unidad Experimental**

La unidad experimental del experimento será el tiempo en segundos que cada tarda cada algoritmo en el ordenamiento del arreglo de números de

**1.3 Variable de respuesta**

Nuestra variable de respuesta es la eficiencia de desempeño del algoritmo y será medida en milisegundos donde un menor tiempo indica mejor desempeño y por lo tanto una mayor eficiencia.

**1.4 Factores controlables**

Los factores controlables que se fijarán para el experimento incluyen: en primer lugar; la estrategia que se utilizará para ordenar el arreglo de números (CountingSort o PigeonHoleSort); en segundo lugar, lenguaje de programación en cual se codificará cada algoritmo; en tercer lugar el tamaño del arreglo y por ultimo, el procesador de la máquina donde el algoritmo sea ejecutado.

**1.5 Factores no controlables**

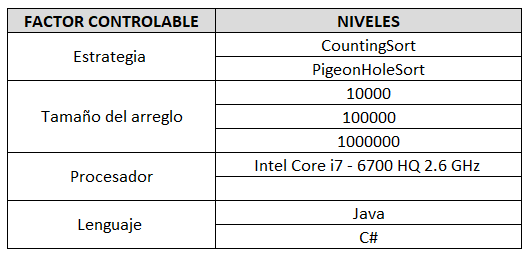
Entre los factores no controlables se tiene: Por un lado, el poder de cómputo utilizado en tareas secundarias por el sistema operativo, y por otro lado, la temperatura del procesador y del hardware en general que pueda afectar el rendimiento de la máquina donde se ejecuta el algoritmo.

**1.6 Factores estudiados**

Los factores que son de interés para la investigación son los siguientes: Algoritmo utilizado en la ejecución,tamaño del algoritmo, tiempo de ejecución, y capacidad de cómputo de máquina donde se ejecuta el algoritmo

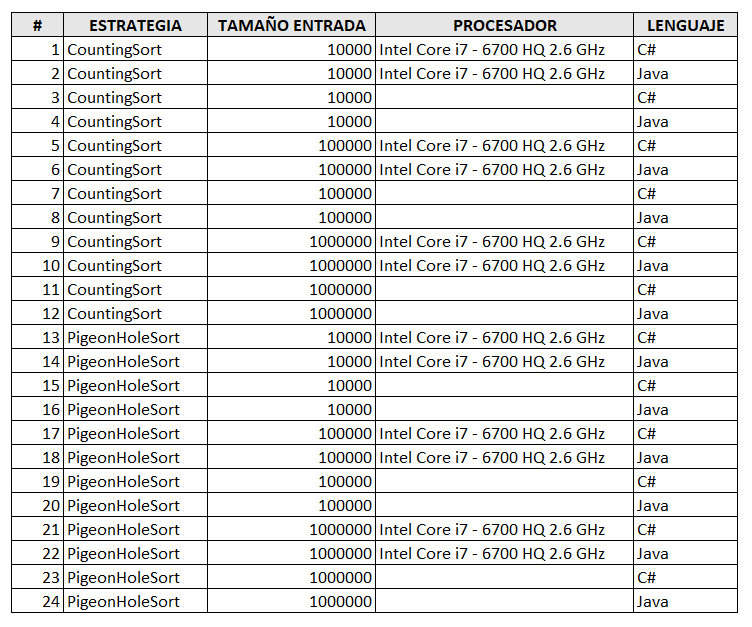
**1.7 Niveles**

Se muestran los niveles de cada factor controlable de experimento:



**1.8 Tratamientos**

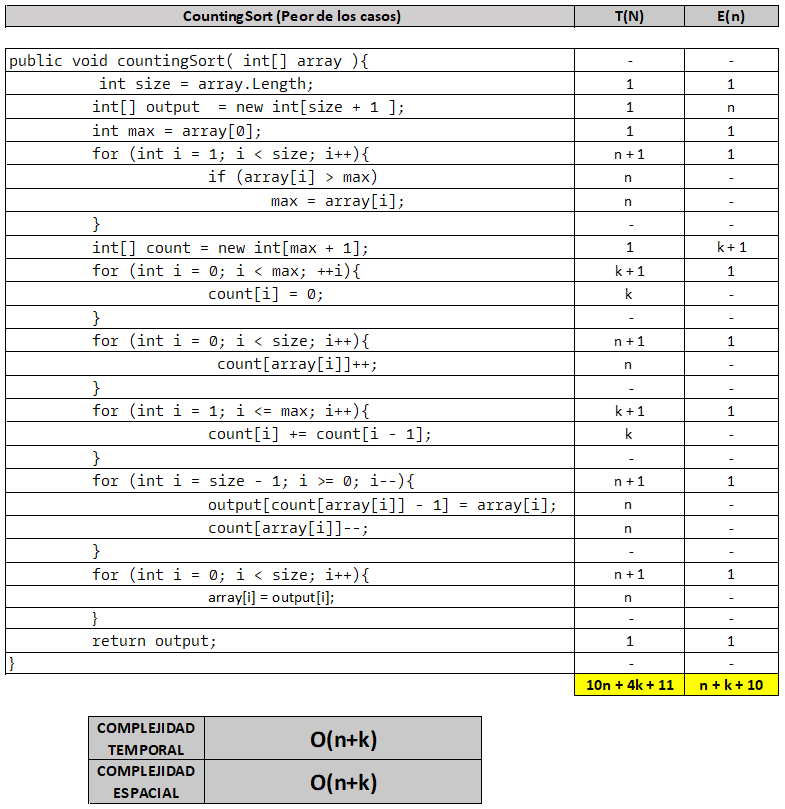
Los tratamientos resultantes de la combinación de todos lo factores y por lo tanto las pruebas que se ejecutarán serán las siguientes.



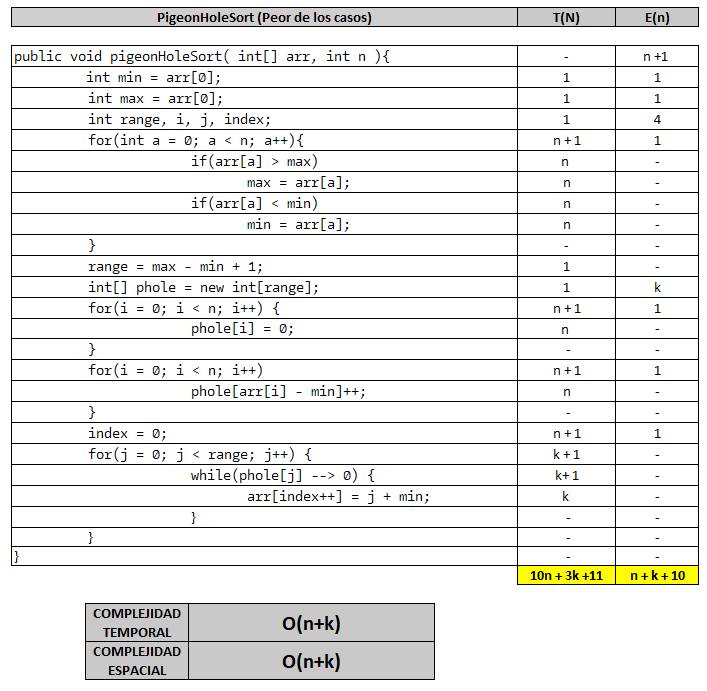
1. **Análisis**

**2.1 Análisis complejidad temporal**

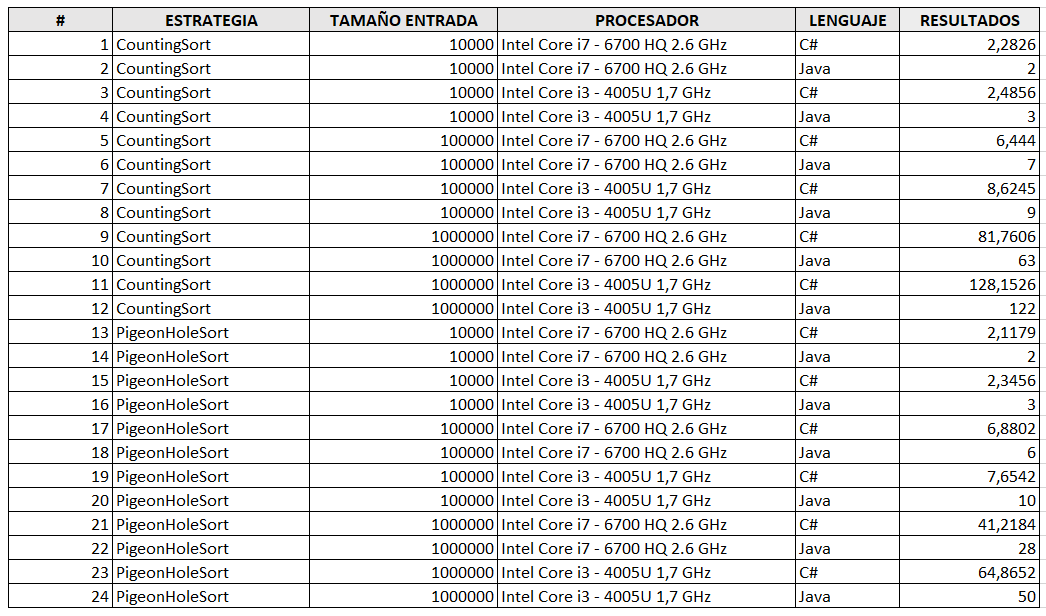
**2.1.1 Análisis de complejidad temporal y espacial CountingSort**

****

**2.1.2 Análisis complejidad temporal PigeonSort**

****

1. **Resultados**

****

1. **Análisis e interpretación**

Del experimento realizado con respecto a los algoritmos CountingSort y PigeonHoleSort se analizará la variable de interés tiempo.

**Counting Sort**

Se puede observar como con el procesador I7 para arreglos pequeños (10.000 datos) hay una gran similitud en las diferencias de tiempo. Por ejemplo entre la implementación en java y c# solo se diferencian por décimas de milisegundos.Por otro lado, para la ejecución con el procesador i3 la diferencia se hace notable llegandose a notar que los tiempo de ejecución se duplican.Para arreglos de 100.000 números el anterior análisis se mantiene, sin embargo para arreglos de 1.000.000 de datos la implementación en c# en la máquina de procesador i7 brinda mejores resultados con la implementación en java. Es curioso como notar que esto no se aplica para la implementación en la máquina con el procesador con I3 dando resultados bastante similares independientemente del lenguaje que sea utilizado.

**PigeonHoleSort**

De igual manera que con CountingSort, no se aprecian grandes diferencias para volúmenes pequeños de datos (10.000) independiente del procesador utilizado o del lenguaje de programación utilizados. Para el caso de 100.000 datos se comienza a notar una mejora en la eficiencia en la implementación en c# para el computador que posee el procesador intel I3. Por último se aprecia como para grandes volúmenes de datos (100.000), la implementación en java muestra ser cerca de un 30 % más efectivo.

En términos generales se puede interpretar como como existen diferencias en los tiempo de complejidad entre CountingSort y PigeonHoleSort y también se evidencia como PigeonHoleSort es un poco más eficiente sobre todo para entradas de datos grandes.

**Conclusiones**

* El factor que más influye en los tiempos de ejecución de nuestro experimento es el procesador de la máquina en donde en todos los casos se observó un mejor desempeño en el procesador más potente (I7).
* El que dos algoritmos compartan la misma complejidad temporal asintótica no significa que sus tiempos de ejecución reales sean los mismos. Generalmente durante el cálculo de la complejidad temporal de los algoritmos se toman en consideración los términos mayores dejando a un lado los otros. Durante una ejecución real, los términos “extras” cobran importancia. El programador que desee implementar alguno de los dos algoritmos analizados deberá considerar cuál de los dos se acomoda más a sus necesidades.
* El lenguaje de programación java parece ser el más indicado para procesar grandes volúmenes de datos que rondan el millón.

**Configuración de los escenarios**

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Clase** | **Escenario** |
| setupStage1 | AlgorithmsTest |  |
| setupStage2 | AlgorithmsTest |  |
| setupStage3 | AlgorithmsTest |  |
| setupStage4 | AlgorithmsTest |  |

**Diseño de casos de pruebas**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | countingSort | setupStage1 | [4,3,7,7,1,8,1,8,3,7] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | countingSort | setupStage2 | [29, 91, 63, 2, 84, 8, 22, 76, 55, 26, 24, 86, 70, 77, 15, 38, 2, 35, 6, 61] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | countingSort | setupStage3 | [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | countingSort | setupStage4 | [15,14,13,12,11,10,9,8,7,6,5,4,3,2,1] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | countingSort |  | Un arreglo generado aleatoriamente de tamaño 100. | **True**: Si después de recorrer el arreglo resultante cada elemento es menor al que le sigue. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | countingSort |  | Un arreglo generado aleatoriamente de tamaño 1000. | **True**: Si después de recorrer el arreglo resultante cada elemento es menor al que le sigue. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | pigeonholeSort | setupStage1 | [4,3,7,7,1,8,1,8,3,7] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

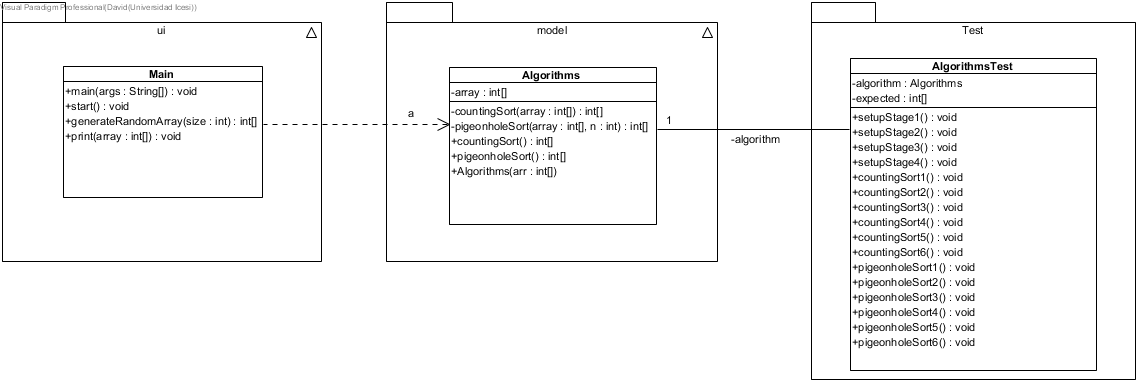
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | pigeonholeSort | setupStage2 | [29, 91, 63, 2, 84, 8, 22, 76, 55, 26, 24, 86, 70, 77, 15, 38, 2, 35, 6, 61] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | pigeonholeSort | setupStage3 | [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

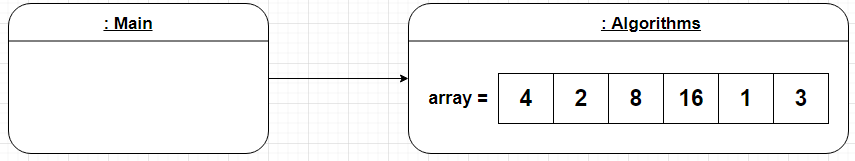
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | pigeonholeSort | setupStage4 | [15,14,13,12,11,10,9,8,7,6,5,4,3,2,1] | **True**: Si después de comparar con el resultado esperado cada elemento coincide. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | pigeonholeSort |  | Un arreglo generado aleatoriamente de tamaño 100. | True: Si después de recorrer el arreglo resultante cada elemento es menor al que le sigue. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** | | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Algorithms | pigeonholeSort |  | Un arreglo generado aleatoriamente de tamaño 1000. | True: Si después de recorrer el arreglo resultante cada elemento es menor al que le sigue. |

**Diagrama de clases:**



**Diagrama de Objetos:**