Aplicación De Métodos de Ordenamiento En Grandes Cantidades De Datos Aplicado A La Programación Orientada A Objetos

Heriberto David Yepes

Cristhian Urrego Salazar

Cartago – Colombia

herdavyep@hotmail.com

cristhianu92@gmail.com

Resumen

**Aplicando los cuatro pilares de la programación orientada a objetos utilizando Java como lenguaje de programación, se realizo un software que implementa la creación de archivos con grandes cantidades de números enteros, de hasta cinco cifras al azar, utilizando el método *Random,* para posteriormente ordenarlos con diferentes métodos de ordenamiento con el objetivo de realizar un análisis y determinar cual es el método más eficiente de los que se implementaron en el proyecto.**

Palabras Claves

Java, objetos, ordenamiento, orientada, métodos, P.O.O, programación.

Abstract

Applying the four pillars of object-oriented programming using Java as a programming language, software was implemented that implements the creation of files with large numbers of whole numbers, up to five random numbers, using the Random method, and then order them with Different ordering methods with the purpose of performing an analysis and determining which is the most efficient method of those that were implemented in the project.

Introducción

En el presente trabajo se desarrolla una implementación de distintos métodos de ordenamiento con el objetivo de entender mejor su función y eficiencia en grandes cantidades de números enteros de hasta veinte millones de datos realizando una comparativa entre ellos concluyendo cual de ellos es el mas eficiente. Dividiendo el proyecto en cinco clases principales y una interface padre que hereda a las clases de los métodos de ordenamiento.

I. Metodología Utilizada

Los algoritmos de ordenamiento nos permite, como su nombre lo dice, ordenar. En este caso, nos servirán para ordenar vectores o matrices con valores asignados aleatoriamente. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y revisando el código, escrito en Java, de cada algoritmo.

II. Estructura del Programa

En el desarrollo del software se implemento como clase principal el *RunMain* el cual se encarga de ejecutar el método estático *Menu.getMenuPrincipal* que desencadena todo el funcionamiento del programa. La siguiente de las clases a tomar en cuenta en un correcto orden de desarrollo es la clase *Menu* que se ocupa de imprimir los formularios de opciones dentro del programa así como de llamar a los métodos de las demás clases, aquí se crea el archivo de fichero de texto llamado *Lista.txt* para llenarlo y trasladar la información del archivo a un *array* por la clase  *MetodosArchivoTxT* para su posterior ordenamiento. Después tenemos la *InterfaceMetodo* que hereda a todas las clases de algoritmos de ordenamiento he implementa la clase *ConversionTiempo* que recibe una variable con la cual ejecuta un mensaje al usuario con el tiempo que tardo la ejecución del método de ordenamiento.

Lenguaje de Programación Utilizado

El lenguaje de programación utilizado es Java que es un lenguaje de programación de propósito general, concurrente, orientado a objetos

Código de los Algoritmos Utilizados

<https://herdavyep.github.io/githubPages_metodoDeOrdenamiento/>

Resultados Alcanzados

El resultado alcanzado fue un software que implementaba nueve métodos de ordenamiento con el cual se puede generar un archivo con hasta 20 millones de números enteros para posteriormente ordenarlos con alguno de estos métodos.

Evaluación de los resultados de los algoritmos implementados

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Métodos Cantidad | Quick | Merge | Radix | Inserción | Selección | Shell | Heap | Counting | burbuja |
| 1  millón | 167 mili | 491 mili | 511 mili | 16,46 min | 16,43 min | 259 mili | 340 mili | 141 mili | 20,65 min |
| 2 millones | 530 mili | 441 mili | 745 mili | N/A | N/A | 473 mili | 1 seg | 435 mili | N/A |
| 5 millones | 1 seg | 1 seg | 1 seg | N/A | N/A | 1 seg | 2 seg | 768 mili | N/A |
| 10 millones | 3 seg | 2 seg | 4 seg | N/A | N/A | 3 seg | 6 seg | 1 seg | N/A |
| 15 millones | 3 seg | 3 seg | 6 seg | N/A | N/A | 4 seg | 9 seg | 2 seg | N/A |
| 20 millones | 3 seg | 4 seg | 7 seg | N/A | N/A | 5 seg | 12 seg | 2 seg | N/A |

Equipo 1 pruebas.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Métodos Cantidad | Quick | Merge | Radix | Inserción | Selección | Shell | Heap | Counting | burbuja |
| 1  millón | 310  mili | 334  mili | 274 mili | 16,46 min | 16,43 min | 256 mili | 346 mili | 131 mili | 20,65 min |
| 2 millones | 386  mili | 492  mili | 745 mili | N/A | N/A | 675 mili | 669 mili | 181 mili | N/A |
| 5 millones | 1 seg | 1 seg | 1 seg | N/A | N/A | 1 seg | 2 seg | 601 mili | N/A |
| 10 millones | 3 seg | 2 seg | 4 seg | N/A | N/A | 3 seg | 6 seg | 1 seg | N/A |
| 15 millones | Memoria insuficiente | Memoria insuficiente | Memoria insuficiente | N/A | N/A | Memoria insuficiente | Memoria insuficiente | Memoria insuficiente | N/A |
| 20 millones | Memoria insuficiente | Memoria insuficiente | Memoria insuficiente | N/A | N/A | Memoria insuficiente | Memoria insuficiente | Memoria insuficiente | N/A |

Equipo 2 pruebas.

Complejidad Métodos de Ordenamiento

|  |  |
| --- | --- |
| **Algoritmo** | **Complejidad** |
|  |  |
| Quick | Promedio: O(n log n), peor caso: O(*n*²) |
| Merge | O(n log n) |
| Radix | O(n) |
| Inserción | O(n 2 ), aunque es O(n) en el mejor caso |
| Selección | O(n 2 ) |
| Shell | O(n log2 n) |
| Heap | O(n log n) |
| Counting | O(n) |
| Burbuja | O(n 2 ) |

Grafico de Comparación de Tiempo

Conclusiones

De acuerdo a la evaluación de datos se puede concluir que el algoritmo de ordenamiento más eficiente aplicado a la circunstancias y restricciones realizadas en el programa es el llamado *CountingSort* con una eficiencia de hasta el 30% frente al método *Quick* siendo este ultimo un método recursivo.

Los métodos burbuja, selección e inserción no son recomendables para ordenar grandes flujos de información.

Recomendaciones

1. El algoritmo del método *Radix* implementado en este caso, solo es funcional en números enteros positivos.

2. El algoritmo que importa la información del archivo no es eficiente ya que primer crea un *array* tipo *String* para luego hacerle el casteo y volverlo un *array* tipo *int.* Se recomienda generar un algoritmo más eficiente para realizar esta tarea.

Computadores Usados

Equipo 1 pruebas:

Windows 10

Procesador: AMD Phenom(tm) II X4 955 Processor 3.20 GHz

Memoria RAM: 8 gb DDR3

Tarjeta de video: EVGA Geforce 950 GT SSC 2GB

Equipo 2 pruebas:

MacBook Air mid 2013

macOS Sierra

Procesador: 1,3 GHz Intel Core i5

Memoria RAM: 4 GB 1600 MHz DDR3

Graficos: Intel HD Graphics 5000 1536 MB

Referencias bibliográficas

<https://blog.zerial.org/ficheros/Informe_Ordenamiento.pdf>

<https://geekytheory.com/como-leer-un-fichero-en-java>

http://www.geeksforgeeks.org/

http://algorithms.tutorialhorizon.com/counting-sort/

http://www.sanfoundry.com/java-program-implement-heap-sort/

https://www.youtube.com/watch?v=I-d9\_l7myXw