Cristian Bedoya, Leidy Viviana Hernandez Matta

Multihilos, una alternativa para mejorar la eficiente en algoritmos de ordenación

[cristianbedoya1123@hotmail.com](mailto:cristianbedoya1123@hotmail.com) , [ladyvivi\_17@hotmail.com](mailto:ladyvivi_17@hotmail.com)

# Resumen

El desarrollo de aplicaciones de software de alta calidad, en una era de competitividad, es cada día más importante. Uno de los aspectos más relevantes a considerar son los tiempos de respuesta, tomando esto en cuenta, en este proyecto vamos analizar los métodos de ordenamiento y su rendimiento en tiempos de respuesta, mediante el comportamiento que presentan los algoritmos simples y la eficiencia obtenida al implementar algoritmos con hilos.

Palabras claves

# **Ordenamiento:** Es el procedimiento en el cual se agrupan los registros en orden definido, con el fin de facilitar la búsqueda de datos ordenados en secuencia.

**Algoritmo:** una secuencia de instrucciones que representan un modelo de solución para determinado tipo de problema

**Hilo:** son básicamente una tarea que puede ser ejecutada en paralelo con otra tarea; teniendo en cuenta lo que es propio de cada hilo es el contador de programa, la pila de ejecución y el estado de la CPU (incluyendo el valor de los registros).

# Resumen en ingles

# The development of high quality software applications, in an era of competitiveness, is becoming more and more important. One of the most relevant aspects to consider is the response times, taking this into account, in this project we will analyze the ordering methods and their performance in response times, through the behavior of the simple algorithms and the efficiency obtained when implementing Algorithms with threads.

# Palabras claves en ingles

**Ordering:** This is the procedure in which the records are grouped in a defined order, in order to facilitate the search of data ordered in sequence.

**Algorithm:** a sequence of instructions that represent a solution model for a certain type of problem.

**Thread:** are basically a task that can be executed in parallel with another task; Taking into account what is proper of each thread is the program counter, the execution stack and the CPU state (including the value of the registers).

Introducción

En la computación el ordenamiento de datos cumple un rol importante, ya sea como un fin en sí o como parte de otros procedimientos más complejos. A través de los años se han desarrollado muchas técnicas en este ámbito, cada una con características particulares, ventajas y desventajas sobre las demás.

Los métodos  de ordenamiento son útiles en los casos en dónde el número de elementos a ordenar no es muy grande. En este caso, nos servirán para ordenar datos numéricos hasta de 5 dígitos tanto positivos como negativos y de más de un millón de datos. Este trabajo se centra en los métodos de ordenación más comunes, analizando el tiempo empleado por cada algoritmo en cuatro escenarios diferentes: sin hilos y sin impresión de datos, sin hilos y con impresión de datos, con hilos y sin impresión de datos, con hilos e impresión de datos, para ello se empleara como herramienta el lenguaje de programación Python.

Este informe detalla cada método de ordenamiento, desde uno simple hasta el más complejo, presentando características tales como tiempos de ejecución, pre-requisitos, funcionalidad, alcance, entre otros.

Algoritmos usados

**Ordenamiento Rápido - Quick Sort**

El ordenamiento rápido es un algoritmo creado en 1962 por el científico británico en computación C. A. R. Hoare, antes de que los lenguajes de programación tuvieran la capacidad para manejar funciones recursivas. Este algoritmo se basa en el principio de “divide y vencerás”, de esta manera es más fácil y rápido ordenar dos arreglos o listas de datos pequeños, que un arreglo o lista grande. Este algoritmo en promedio permite ordenar n elementos en un tiempo proporcional a n log n.

Normalmente al inicio de la ordenación se escoge un elemento aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a que el arreglo este ordenado respecto al punto de división o la mitad del arreglo. Se podrá garantizar que los elementos a la izquierda de la mitad son los menores y los elementos a la derecha son los mayores. Los siguientes pasos son llamados recursivos con el propósito de efectuar la ordenación por partición al arreglo izquierdo y al arreglo derecho, que se obtienen de la primera fase. El tamaño de esos arreglos en promedio se reduce a la mitad. Así se continúa hasta que el tamaño de los arreglos a ordenar es 1, es decir, todos los elementos ya están ordenados. En promedio para todos los elementos de entrada de tamaño n, el método hace O(n log n) comparaciones, el cual es relativamente eficiente.

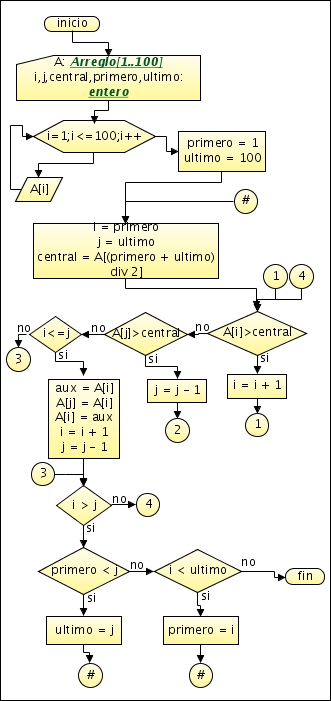


Ilustración 1 Flujograma Quick sort

**Inserción Directa**

El algoritmo de ordenación por el método de inserción directa es un algoritmo relativamente sencillo y se comporta razonablemente bien en gran cantidad de situaciones.

Se basa en intentar construir una lista ordenada en el interior del array a ordenar.

Realiza una cantidad de comparaciones bastante equilibrada con respecto a los intercambios.

Este algoritmo se basa en hacer comparaciones, así que para que realice su trabajo de ordenación son imprescindibles dos cosas: un array o estructura similar de elementos comparables y un criterio claro de comparación, tal que dados dos elementos nos diga si están en orden o no.

En cada iteración del ciclo externo los elementos 0 a i forman una lista ordenada.

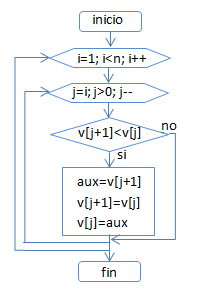


Ilustración 2 Flujograma Inserción Directa

**Ordenamiento por Mezcla (Merge)**

Mediante el enfoque de Dividir y conquistar, este algoritmo divide el arreglo inicial en dos arreglos donde cada uno contiene la mitad de los datos (partes iguales más o menos uno), y se ordenan mediante sucesivos llamados recursivos para luego fusionar los resultados en el arreglo inicial.

Este algoritmo se basa en una función que permite mezclar dos vectores ordenados, produciendo como resultado un tercer vector ordenado que contiene los elementos de los dos vectores iniciales, el cual tiene una complejidad de O(n) para mezclar dos arreglos, donde n es la suma de los tamaños de los dos arreglos.

**Ordenamiento montones (Heapsort)**

A grandes rasgos el algoritmo de ordenación por montículos consiste en meter todos los elementos del array de datos en un montículo MAX, y luego realizar N veces eliminar\_max(). De este modo, la secuencia de elementos eliminados nos será entregada en orden decreciente.

**Ordenamiento Conteo**

Este método utiliza un arreglo auxiliar para contabilizar el número de llaves que son mayores que la llave actual. • El arreglo de contadores, especifica la posición final donde debería estar cada elemento.

**Ordenamiento Radix Sort**

Radix Sort (ordenamiento Radix) es un algoritmo de ordenamiento estable\* para ordenar elementos identificados por llaves (o claves) únicas. Cada llave debe ser una cadena o un número capaz de ser ordenada alfanuméricamente. • Este método ejecuta un número de repeticiones igual al número de caracteres de las llaves a ordenar. El Radix Directo, inicia con el dígito más a la derecha repartiendo los datos en “canastas”, estos datos se reparten de nuevo de acuerdo al siguiente dígito y así sucesivamente hasta terminar con el dígito de mas a la izquierda.

Códigos de algoritmos

**Inserción Directa:**

**def** insercionDirecta(lista, tam):  
 **for** i **in** range(1, tam):  
 v = lista[i]  
 j = i - 1  
 **while** j >= 0 **and** lista[j] > v:  
 lista[j + 1] = lista[j]  
 j = j - 1  
 lista[j + 1] = v  
  
**HeapSort:**

**def** swap(lista,i, j):  
 lista[i], lista[j] = lista[j], lista[i]  
  
**def** heapify(lista,end,i):  
 l=2 \* i + 1  
 r=2 \* (i + 1)  
 max=i  
 **if** l < end **and** lista[i] < lista[l]:  
 max = l  
 **if** r < end **and** lista[max] < lista[r]:  
 max = r  
 **if** max != i:  
 swap(lista,i, max)  
 heapify(lista,end, max)  
  
**def** heap\_sort(lista):  
 end = len(lista)  
 start = end // 2 - 1 **for** i **in** range(start, -1, -1):  
 heapify(lista,end, i)  
 **for** i **in** range(end-1, 0, -1):  
 swap(lista,i, 0)  
 heapify(lista,i, 0)

**Quick Sort:**

**def** quicksort(lista, izq, der):  
 i = izq  
 j = der  
 x = lista[(izq + der) / 2]  
  
 **while** (i <= j):  
 **while** lista[i] < x **and** j <= der:  
 i = i + 1  
 **while** x < lista[j] **and** j > izq:  
 j = j - 1  
 **if** i <= j:  
 aux = lista[i]  
 lista[i] = lista[j]  
 lista[j] = aux  
 i = i + 1  
 j = j - 1  
  
 **if** izq < j:  
 quicksort(lista, izq, j)  
 **if** i < der:  
 quicksort(lista, i, der)

**Merge Sort:**

**def** merge\_sort(numbers):n = len(numbers)  
 **if** (n == 1): **return** numbers  
 left = merge\_sort(numbers[:(n / 2)])  
 right = merge\_sort(numbers[(n / 2):])  
  
 **return** merge(left, right)  
  
**def** merge(left, right):  
 result = []  
 i = 0  
 j = 0  
 len\_left = len(left)  
 len\_right = len(right)  
  
 **while** (i < len\_left **or** j < len\_right):  
 **if** (i >= len\_left):  
 result.append(right[j])  
 j = j + 1  
 **elif** (j >= len\_right):  
 result.append(left[i])  
 i = i + 1  
 **elif** (left[i] < right[j]):  
 result.append(left[i])  
 i = i + 1  
 **else**:  
 result.append(right[j])  
 j = j + 1  
  
 **return** result  
  
**Ordenamiento por Conteo:**

**def** counting\_sort(array, maxval):  
 n = len(array)  
 m = maxval + 1  
 count = [0] \* m **for** a **in** array:  
 count[a] += 1 i = 0  
 **for** a **in** range(m): **for** c **in** range(count[a]): array[i] = a  
 i += 1  
 **return** array

**Radix Sort:**

**def** radix\_sort(random\_list):  
 len\_random\_list = len(random\_list)  
 modulus = 10  
 div = 1  
 **while** True:  
 new\_list = [[], [], [], [], [], [], [], [], [], []]  
 **for** value **in** random\_list:  
 least\_digit = value % modulus  
 least\_digit /= div  
 new\_list[least\_digit].append(value)  
 modulus = modulus \* 10  
 div = div \* 10  
  
 **if** len(new\_list[0]) == len\_random\_list:  
 **return** new\_list[0]  
  
 random\_list = []  
 rd\_list\_append = random\_list.append  
 **for** x **in** new\_list:  
 **for** y **in** x:  
 rd\_list\_append(y)

# Metodología

# Programación de rutinas de ordenamiento en lenguaje de alto nivel con lista de más de un millón de números generados aleatoriamente con muestreo de tiempo diferencial entre la iniciación y la finalización, aplicando técnica simple y con hilos con escenarios de visualización y no visualización de números generados.

# Lenguaje de programación

# Python es un lenguaje de programación poderoso y fácil de aprender. Cuenta con estructuras de datos eficientes y de alto nivel y un enfoque simple pero efectivo a la programación orientada a objetos. La elegante sintaxis de Python y su tipado dinámico, junto con su naturaleza interpretada, hacen de éste un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en diversas áreas y sobre la mayoría de las plataformas.

# Conclusiones

A través de las pruebas realizadas, se pudo establecer que aunque los algoritmos de ordenación poseen rendimientos ya reconocidos, los cuales están relacionados con el número de elementos a ordenar y el grado de desorden de los mismos, la aplicación de multihilos o de ejecución en paralelo permite lograr mejoras significativas en su rendimiento frente a la utilización de arquitectura de procesamiento monohilo convencionales.

Para establecer el rendimiento de un algoritmo de ordenación es imprescindible someterlo a trabajo con grandes volúmenes de información y contrastar los tiempos de respuesta.

# Recomendaciones

Para desarrollar cabalmente las pruebas de los algoritmos de ordenación a este nivel, manteniendo unos tiempos de respuesta razonables es altamente recomendable el uso de equipos de cómputo de buenas prestaciones.

# specificaciones técnicas

Equipo de cómputo utilizado para las pruebas

Disco Duro: 500 GB HDD

Procesador: Intel(R) Core™ ¡5-450 CPU M460 @ 2.53GHZ

Memoria (RAM): 3 GB DDR3 Memory

# Referencias Bibliográficas

Becerra Santamaría, César. (1988) [*Estructuras de datos en C*](http://www.libros.cotecnova.edu.co/shared/biblio_view.php?bibid=112&tab=opac) *(4ta Edición), Santa fe de Bogota:* Editorial Kimpres Ltda

,