

COMPARATIVA DE METODOS DE ORDENAMIENTO IMPLEMENTADOS CON HILOS Y SIN HILOS EN PYTHON

Jhon James Cano Sánchez

Juan Carlos Serna Gómez

Corporación de Estudios Tecnológicos del Norte del Valle

Cartago, Colombia

Resumen

Debido a que las estructuras de datos son utilizadas para almacenar información, para poder recuperar esa información de manera eficiente es deseable que aquella esté ordenada. Existen varios métodos para ordenar las diferentes estructuras de datos básicas. En general los métodos de ordenamiento no son utilizados con frecuencia, en algunos casos sólo una vez. Hay métodos muy simples de implementar que son útiles en los casos en donde el número de elementos a ordenar no es muy grande (Ordenamiento sin hilos). Por otro lado hay métodos sofisticados, más difíciles de implementar pero que son más eficientes en cuestión de tiempo de ejecución (Ordenamiento con hilos), por esta razón se realiza una comparación entre los diferentes métodos que muestra de forma notoria las diferencias significantes que

existen entre ambos, basados en el tiempo de ejecución y respuesta.

Abstract

Because the data structures are used to store information, in order to retrieve that information efficiently it is desirable that it be ordered. There are several methods for sorting the different basic data structures. In general, sorting methods are not used frequently, in some cases only once. There are very simple methods to implement that are useful in cases where the number of elements to order is not very large (Ordering wirelessly). On the other hand, there are sophisticated methods, which are more difficult to implement but are more efficient in terms of execution time (Threaded Ordering). For this reason, a comparison is made between the different methods, which shows clearly the significant differences between Both, based on execution time and response.

Palabras Claves

Estructura de datos, Hilos, Información, Métodos de ordenamiento, Python.

KeyWords

Data structure, Threads, Information, Methods of ordering, Python.

I. INTRODUCCIÓN

Los métodos de ordenamiento son herramientas que permiten organizar de forma correcta una serie de datos obtenidos a partir de una muestra determinada o aleatoria de difícil comparación por métodos ordinarios. Todo parte de la necesidad de mostrar la información de manera eficaz en un tiempo reducido, dicho de otra forma, comparar la información frente a una muestra y obtener el resultante sin recurrir a métodos obsoletos o de difícil comprensión. Los métodos de ordenamiento pueden ser de diversa índole (Inserción Directa, Conteo, Ordenamiento Rápido, entre otros), en algunos casos dependiendo de la cantidad de información a comparar un método puede ser más eficiente que otro y depende del analista lo eficaz que puede resultar el proceso.

II. PRESENTACIÓN PROBLEMA U OBJETIVO

El objetivo parte de la base de analizar los datos de forma congruente en un sentido único con éxito mediante la implementación de métodos y modelos de ordenamiento que generen datos no redundantes que a posteriori determinen la veracidad de la información suministrada y el consolidado de datos situado en un modelo óptimo de aparente desempeño.

III. DESCRIPCIÓN DE LA PROPUESTA

Metodología: se utiliza la metodología aplicada para la comparación de la evolución y respuesta de los métodos de ordenamiento en sus diferentes escenarios mediante ordenamiento de datos de tamaño moderado.

IV. ALGORITMOS UTILIZADOS

Los algoritmos implementados son los métodos de ordenamiento con mayor uso y eficiencia operativa:

- Ordenamiento por inserción
- Ordenamiento por mezcla
- Ordenamiento por montones (Heap Sort)
- Ordenamiento rápido (Quick Sort)
- Ordenamiento por conteo (Counting Sort)

- Ordenamiento por Radix Sort

V. LENGUAJE DE PROGRAMACIÓN

El lenguaje de programación seleccionado para la implementación es Python, el cual es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

VI. CÓDIGOS

Los códigos presentados a continuación son los códigos de los cada uno de los métodos de ordenamientos que se realizó en el lenguaje de programación

MÉTODOS DE ORDENAMIENTO SIN HILOS

Funciones.py

```
import random
import csv
```

```
def crearLista():
    lista = list()
    return lista
```

```
def nombreArchivo():
    nombre = raw_input("Ingrese nombre
del archivo: ")
    return nombre
```

```
def canRegistros():
    cant = int(raw_input("Ingrese Cantidad
de registros a generar: "))
    return cant
```

```
def llenarLista(lista, limite):
    for i in range(0, limite):
        lista.append(random.randint(-9999,
9999))
```

```
def llenarListaConteo(lista, limite):
    for i in range(0, limite):
        lista.append(random.randint(0, 1000))
```

```
def crearCsv(lista, nombre):
    datos = open(nombre + ".csv", "w")
    datos_csv = csv.writer(datos)
    datos_csv.writerow(lista)
    datos.close()
```

```
def imprimirLista(lista):
    print lista
```

```
def leerCsv(lista_2, nombre):
    datos = open(nombre + ".csv", "r")
    datos_csv = csv.reader(datos,
delimiter=",")
    for variable in (datos_csv):
        lista_2.append(variable)
    datos.close()
    return lista_2
```

```
def arreglarlista(lista,nlista):
    for i in lista [0]:
        nlista.append(int(i))
    return nlista
```

```
def longitud(lista):
    return len(lista)
```

```
def imprimirTiempo(tiempo_final):
    print ("Proceso finalizado en %0.5f
segundos" %tiempo_final)
```

INSERTIÓN.PY

```
# *****
```

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

```

HEAP SORT

```

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[l] < lista[i]:
        max = l
    if r < end and lista[r] < lista[i]:
        max = r
    if max != i:
        swap(lista, i, max)
        heapify(lista, end, max)

```

```

def heap_sort(lista):
    end = len(lista)
    start = end // 2 - 1 # use // instead of /
    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):
    """Punto de entrada del algoritmo"""
    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           # init with zeros
    for a in array:
        count[a] += 1         # count
                                # occurrences
    i = 0
    for a in range(m):         # emit
        for c in range(count[a]): # - emit
            'count[a]' copies of '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)
```

RUN_INSERTION.PY

```
# -*- coding: utf-8 -*-
from funciones import *
from insertion import *
from threading import Thread
import time

def main():
    op = " "

    while op != "s" and op != "S":
        print("-----")
        print("\t\tMETODOS DE
ORDENAMIENTO\t\t")
        print("(a). Inserción Directa ")
        print("(b). Ordenamiento por
Mezcla (Merge Sort) ")
        print("(c). Ordenamiento por Montones
(Heap Sort) ")
        print("(d). Ordenamiento rápido
(Quick Sort) ")
        print("(e). Ordenamiento por Conteo
(Counting Sort) ")
        print("(f). Ordenamiento por Radix
Sort ")
        print("(s). Salir")

    op = raw_input("Digite una Opción: ")

    if op == "a" or op == "A": # inserción
Directa

        tiempo_inicial = time.time()
        lista = crearLista()
        nombre = nombreArchivo()
        #llenarLista(lista, 1000)
        nreg = canRegistros()
        llenarLista(lista, nreg)
        crearCsv(lista, nombre)
        imprimirLista(lista)

        # ordenar
        lista_ordenada = crearLista()
        leerCsv(lista_ordenada, nombre)
```

```

nlista = crearLista()
nlista2 =
arreglalista(lista_ordenada,nlista)

tam = longitud(nlista2)
insercionDirecta(nlista2, tam)
imprimirLista(nlista2)
tiempo_final = time.time() -
tiempo_inicial
imprimirTiempo(tiempo_final)

```

if op == "b" **or** op == "B":

ORDENAMIENTO POR MEZCLA (MERGE SORT)

```

tiempo_inicial = time.time()
lista = crearLista()
nombre = nombreArchivo()
nreg = canRegistros()
llenarLista(lista, nreg)
crearCsv(lista, nombre)
imprimirLista(lista)

lista_ordenada = crearLista()
leerCsv(lista_ordenada, nombre)

nlista = crearLista()
nlista2 =
arreglalista(lista_ordenada, nlista)
List2 = sorted(nlista2, reverse =
True)

```

```

merge_sort(nlista2)
imprimirLista(nlista2)

tiempo_final = time.time() -
tiempo_inicial
imprimirTiempo(tiempo_final)

```

if op == "c" **or** op == "C":

ORDENAMIENTO POR MONTONES (HEAP SORT)

```

tiempo_inicial = time.time()
lista = crearLista()

```

```

nombre = nombreArchivo()
nreg = canRegistros()
llenarLista(lista, nreg)
crearCsv(lista, nombre)
imprimirLista(lista)

# ordenar
lista_ordenada = crearLista()
leerCsv(lista_ordenada, nombre)

nlista = crearLista()
nlista2 =
arreglalista(lista_ordenada,nlista)

```

```

tam = longitud(nlista2)
heap_sort(nlista2)
imprimirLista(nlista2)
tiempo_final = time.time() -
tiempo_inicial
imprimirTiempo(tiempo_final)

```

if op == "d" **or** op == "D": ORDENAMIENTO RÁPIDO (QUICK SORT)

```

tiempo_inicial = time.time()
lista = crearLista()
nombre = nombreArchivo()
nreg = canRegistros()
llenarListaConteo(lista, nreg)
crearCsv(lista, nombre)
imprimirLista(lista)

# ordenar
lista_ordenada = crearLista()
leerCsv(lista_ordenada, nombre)

nlista = crearLista()
nlista2 =
arreglalista(lista_ordenada,nlista)

```

```

tam = longitud(nlista2)
quicksort(nlista2, 0, tam - 1)
#quicksort(nlista2, tam)
imprimirLista(nlista2)
tiempo_final = time.time() -
tiempo_inicial

```

```

        imprimirTiempo(tiempo_final)

if op == "e" or op == "E":
ORDENAMIENTO RÁPIDO (QUICK
SORT)

```

```

        tiempo_inicial = time.time()
        lista = crearLista()
        nombre = nombreArchivo()
        nreg = canRegistros()
        llenarLista(lista, nreg)
        crearCsv(lista, nombre)
        imprimirLista(lista)

        # ordenar
        lista_ordenada = crearLista()
        leerCsv(lista_ordenada, nombre)

        nlista = crearLista()
        nlista2 =
        arreglalista(lista_ordenada,nlista)

        tam = longitud(nlista2)
        counting_sort(nlista2, len(nlista2))
        imprimirLista(nlista2)
        tiempo_final = time.time() -
tiempo_inicial
        imprimirTiempo(tiempo_final)

```

```

if op == "f" or op == "F":

```

ORDENAMIENTO RADIX SORT

```

        tiempo_inicial = time.time()
        lista = crearLista()
        nombre = nombreArchivo()
        nreg = canRegistros()
        llenarLista(lista, nreg)
        crearCsv(lista, nombre)
        imprimirLista(lista)

        lista_ordenada = crearLista()
        leerCsv(lista_ordenada, nombre)

        nlista = crearLista()
        nlista2 =
        arreglalista(lista_ordenada,nlista)

```

```

        radix_sort(nlista2)
        imprimirLista(nlista2)
        tiempo_final = time.time() -
tiempo_inicial
        imprimirTiempo(tiempo_final)

```

```

if op == "S" or op == "s":

```

```

        print "Hasta Pronto."
        exit()

```

```

else:

```

```

        print("Digite una Opción \n")

```

```

if __name__ == '__main__':

```

```

    main()

```

METODOS DE ORDENAMIENTOS

CON HILOS

RUN_INSERTION.PY

```

        # -*- coding: utf-8 -*-
from funciones import *
from insertion import *
import threading
import time

```

```

def main():
    op = " "

```

```

        while op != "s" and op != "S":

```

```

            print"-----"

```

```

        ----"

```

```

print("\\t\\tMETODOS DE
ORDENAMIENTO\\t\\t")

```

```

print("(a). Inserción Directa ")

```

```

print("(b). Ordenamiento por Mezcla
(Merge Sort) ")

```

```

print("(c). Ordenamiento por Montones
(Heap Sort) ")

```

```

print("(d). Ordenamiento rápido (Quick
Sort) ")

```

```

print("(e). Ordenamiento por Conteo
(Counting Sort) ")

```

```

print("(f). Ordenamiento por Radix Sort

```

```

")
print("(s). Salir")
op = raw_input("Digite una Opción: ")

if op == "a" or op == "A": # inserción
    Directa

        #tiempo_inicial = time.time()
        lista = crearLista()
        nombre = nombreArchivo()
        nreg=canRegistros()
        hilo =
threading.Thread(target=llenarLista,
args=(lista, nreg,))
        hilo.start()
        hilo2 =
threading.Thread(target=crearCsv,
args=(lista, nombre,))
        hilo2.start()
        tiempo_inicial = time.time()
        imprimirLista(lista)
        lista_ordenada = crearLista()
        nlista = crearLista()
        hilo3 =
threading.Thread(target=leerCsv,
args=(lista_ordenada, nombre,))
        hilo3.start()
        hilo3.join()
        nlista2 =
arreglalista(lista_ordenada, nlista)
        tam = longitud(nlista2)
        insercionDirecta(nlista2, tam)
        imprimirLista(nlista2)
        tiempo_final = time.time() -
tiempo_inicial
        imprimirTiempo(tiempo_final)
        nombre_final = nombreArchivo()
        hilo2 =
threading.Thread(target=crearCsv,
args=(nlista2, nombre_final,))
        hilo2.start()

        if op == "b" or op == "B":

```

ORDENAMIENTO POR MEZCLA (MERGE SORT)

```

tiempo_inicial = time.time()
lista = crearLista()
nombre = nombreArchivo()
nreg = canRegistros()
hilo =
threading.Thread(target=llenarLista,
args=(lista, nreg,))
        hilo.start()
        hilo2 =
threading.Thread(target=crearCsv,
args=(lista, nombre,))
        hilo2.start()
        tiempo_inicial = time.time()
        imprimirLista(lista)

        lista_ordenada = crearLista()
        leerCsv(lista_ordenada, nombre)

        nlista = crearLista()
        nlista2 =
arreglalista(lista_ordenada, nlista)

        imprimirLista(mergesort(nlista2))

        tiempo_final = time.time() -
tiempo_inicial
        imprimirTiempo(tiempo_final)
        nombre_final = nombreArchivo()
        hilo2 =
threading.Thread(target=crearCsv,
args=(mergesort(nlista2), nombre_final,))
        hilo2.start()

        if op == "c" or op == "C":

```

ORDENAMIENTO POR MONTONES (HEAP SORT)

```

lista = crearLista()
nombre = nombreArchivo()
nreg = canRegistros()
hilo =
threading.Thread(target=llenarLista,
args=(lista, nreg,))
        hilo.start()
        hilo2 =
threading.Thread(target=crearCsv,

```



```

args=(lista, nombre,))
    hilo2.start()
    tiempo_inicial = time.time()
    imprimirLista(lista)
    lista_ordenada = crearLista()
    nlista = crearLista()
    hilo3 =
threading.Thread(target=leerCsv,
args=(lista_ordenada, nombre,))
    hilo3.start()
    hilo3.join()
    nlista2 =
arreglalista(lista_ordenada,nlista)

    tam = longitud(nlista2)
    heap_sort(nlista2)
    imprimirLista(nlista2)
    tiempo_final = time.time() -
tiempo_inicial
    imprimirTiempo(tiempo_final)

```

if op == "d" or op == "D": ORDENAMIENTO RÁPIDO (QUICK SORT)

```

    lista = crearLista()
    nombre = nombreArchivo()
    nreg = canRegistros()
    hilo =
threading.Thread(target=llenarLista,
args=(lista, nreg,))
    hilo.start()
    hilo2 =
threading.Thread(target=crearCsv,
args=(lista, nombre,))
    hilo2.start()
    tiempo_inicial = time.time()
    imprimirLista(lista)
    lista_ordenada = crearLista()
    nlista = crearLista()

    hilo3 =
threading.Thread(target=leerCsv,
args=(lista_ordenada, nombre,))
    hilo3.start()
    hilo3.join()
    nlista2 =

```

```

arreglalista(lista_ordenada, nlista)
    tam = longitud(nlista2)
    quicksort(nlista2, 0, tam - 1)
    imprimirLista(nlista2)
    tiempo_final = time.time() -
tiempo_inicial
    imprimirTiempo(tiempo_final)

```

if op == "e" or op == "E": ORDENAMIENTO RÁPIDO (QUICK SORT)

```

    lista = crearLista()
    nombre = nombreArchivo()
    nreg = canRegistros()
    hilo =
threading.Thread(target=llenarListaConteo,
args=(lista, nreg,))
    hilo.start()
    hilo2 =
threading.Thread(target=crearCsv,
args=(lista, nombre,))
    hilo2.start()
    tiempo_inicial = time.time()
    imprimirLista(lista)
    # ordenar
    lista_ordenada = crearLista()
    nlista = crearLista()
    hilo3 =
threading.Thread(target=leerCsv,
args=(lista_ordenada, nombre,))
    hilo3.start()
    hilo3.join()
    nlista2 =
arreglalista(lista_ordenada, nlista)

    tam = longitud(nlista2)
    # print len(nlista2)
    counting_sort(nlista2, tam)
    # counting_sort(nlista2, 7)
    imprimirLista(nlista2)
    tiempo_final = time.time() -
tiempo_inicial
    imprimirTiempo(tiempo_final)

```

if op == "f" or op == "F": ORDENAMIENTO RADIX SORT

```

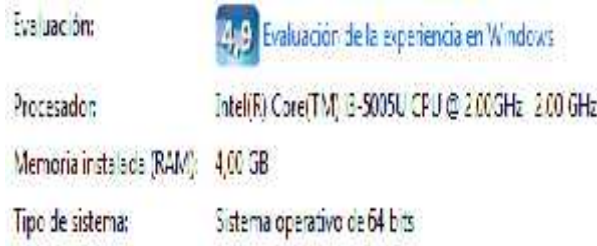
    lista = crearLista()

```

```

nombre = nombreArchivo()
nreg = canRegistros()

```



```

        hilof =
threading.Thread(target=llenarListaConteo,
args=(lista, nreg,))
        hilof.start()
        hilof2 =
threading.Thread(target=crearCsv,
args=(lista, nombre,))
        hilof2.start()
        tiempo_inicial = time.time()
        imprimirLista(lista)
        # ordenar
        lista_ordenada = crearLista()
        nlista = crearLista()
        hilof3 =
threading.Thread(target=leerCsv,
args=(lista_ordenada, nombre,))
        hilof3.start()
        hilof3.join()
        nlista2 =
arreglalista(lista_ordenada,nlista)

```

```

        radix_sort(nlista2)
        imprimirLista(nlista2)
        tiempo_final = time.time() -
tiempo_inicial
        imprimirTiempo(tiempo_final)
        if op == "S" or op == "s":

            print "Hasta Pronto."
            exit()

        else:
            print("Digite una Opción \n")

if __name__ == '__main__':
    main()

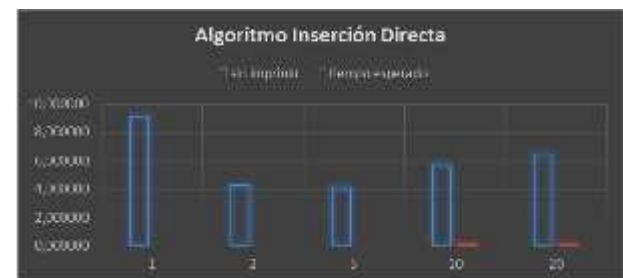
```

CARACTERÍSTICAS DE LA MÁQUINA

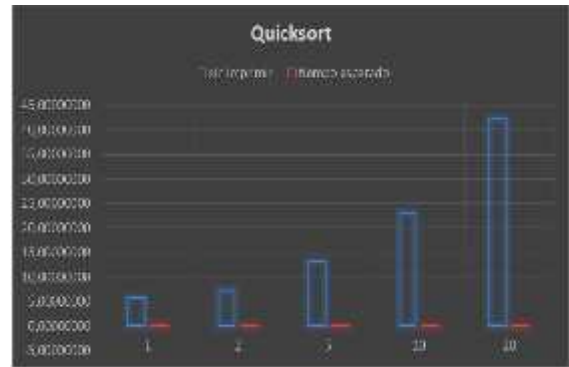
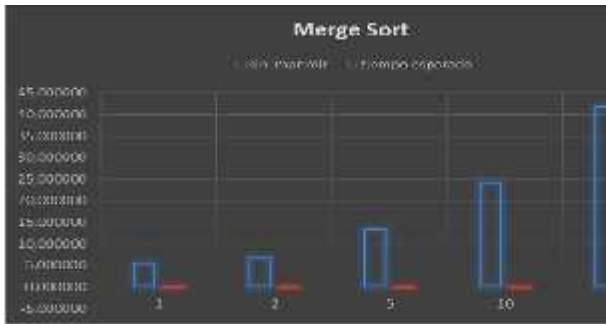
Fabricante: HP

RESULTADOS ALCANZADOS

Algoritmo Inserción Directa $O(n^2)$						
numero datos	tiempo en minutos					
	Sin Hilos		Con Hilos			
millones	sin imprimir	imprimiendo datos	sin imprimir	imprimiendo datos	tiempo esperado	delta tiempo
1	9,126000	605,36300 s	0.00997 s	4.21528 s	0,000000173611000	9
2	4,244000	1300,2563 s	0.04193 s	6.50953 s	0,000000694444444	4
5	4,027000	3748,1203 s	0.23644 s	16.06289 s	0,000004340277778	4
10	5,616000	10022,0214 s	0.09881 s	33.86321 s	0,000017361111111	6
20	6,427000	Error de Memoria	0.02977 s	64.20960 s	0,000069444444444	6

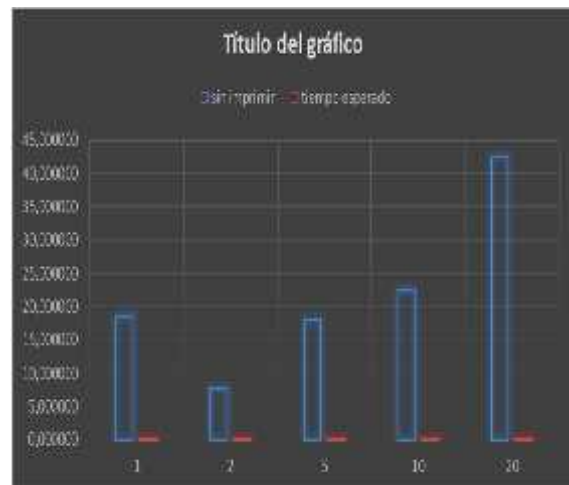
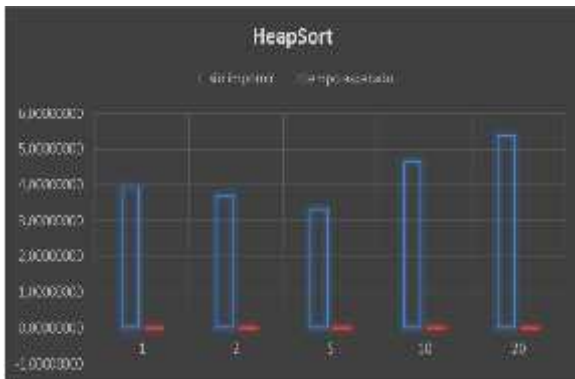


Algoritmo Merge Sort sin hilos $(n \log n)$						
numero datos	tiempo en minutos					
	Sin Hilos		Con Hilos			
millones	sin imprimir	imprimiendo datos	sin imprimir	imprimiendo datos	tiempo esperado	delta tiempo
1	5,602000	40.96900 s	0.00055 s	3.55823 s	-0,009917547	5,611917547
2	6,973000	72.35400 s	0.00902 s	6.34005 s	-0,002565984	6,975565984
5	13,266000	167.35700 s	0.00300 s	15.59070 s	-0,005585919	13,27158592
10	24,170000	369.28900 s	0.00491 s	31.05123 s	-0,009917547	24,17991755
20	41,901000	Error de Memoria	0.00455 s	63.82649 s	-0,01732651	41,91832651



Algoritmo Heap Sort sin hilos ($n \log n$)					
numero datos	tiempo en minutos				
	Sin Hilos		Con Hilos		
millones	sin imprimir	imprimiendo datos	sin imprimir	imprimiendo datos	tiempo esperado
1	3,94700000	51.64100 s	0.05899 s	3.24244 s	-0,009917547
2	3,69700000	100.40900 s	0.02747 s	6.38755 s	-0,002565984
5	3,32500000	236.70900 s	0.05179 s	15.72822 s	-0,005585919
10	4,62300000	514.75100 s	0.02124 s	31.57545 s	-0,009917547
20	5,36600000	339.91800 s	0.14495 s	65.25553 s	-0,01732651

Algoritmo Counting Sort sin hilos $O(n+k)$						
numero datos	tiempo en minutos					
	Sin Hilos		Con Hilos			
millones	sin imprimir	imprimiendo datos	sin imprimir	imprimiendo datos	tiempo esperado	delta tiempo
1	18,502000	32.93400 s	3.92404 s	5.98475 s	0,004166667	18,497833
2	7,571000	53.54600 s	5.60954 s	9.08253 s	0,000833333	7,570167
5	18,081000	127.41400 s	8.92366 s	18.92688 s	0,002083333	18,078917
10	22,486000	266.65700 s	7.04798 s	34.84244 s	0,004166667	22,481833
20	42,383000	498.3365 s	6.86905 s	67.01715 s	0,008333333	42,374667



Algoritmo quick Sort sin hilos $O(n \log n)$						
numero datos	tiempo en minutos					
	Sin Hilos		Con Hilos			
millones	sin imprimir	imprimiendo datos	sin imprimir	imprimiendo datos	tiempo esperado	delta tiempo
1	5,67800000	34,54782 s	1.82627 s	3.40005 s	-0,009917547	5,68791755
2	7,19200000	93,14563 s	1.72050 s	6.82093 s	-0,002565984	7,19456598
5	13,19900000	303,1546 s	0.24087 s	16.99130 s	-0,005585919	13,20458592
10	23,14600000	991,2256 s	16.10980 s	33.03148 s	-0,009917547	23,15591755
20	42,22800000	2458,9514 s	49.22254 s	66.72416 s	-0,01732651	42,24532651

Algoritmo Radix Sort sin hilos $O(nk)$						
numero datos	tiempo en minutos					
	Sin Hilos		Con Hilos			
millones	sin imprimir	imprimiendo datos	sin imprimir	imprimiendo datos	tiempo esperado	delta tiempo
1	6,02900000	86,1354 s	0.05080 s	3.20842 s	0,004166667	6,024833333
2	8,52300000	214,254 s	0.14823 s	6.26211 s	0,000833333	8,522166667
5	13,75100000	1001,5874 s	0.34594 s	15.16091 s	0,002083333	13,74891667
10	22,36000000	Error de Memoria	0.29293 s	31.88739 s	0,004166667	22,35583333
20	42,06900000	Error de Memoria	0.04104 s	63.67557 s	0,008333333	42,06066667



Para ejecutar estos métodos se hace necesario contar con recursos de máquina lo suficientemente robustos para poder correr con facilidad dichos algoritmos.

CONCLUSIONES

Al implementarse los ejercicios de métodos de ordenamiento de INSERCIÓN, MEZCLA, HEAP SORT, QUICKSORT, RIDEX SORT, CONTEO para realizar una comparación objetiva en el promedio de tiempo durante su ejecución se pueden ver series diferencias.

Al implementar los métodos de ordenamiento con hilos se puede evidenciar la diferencia en los tiempos de ejecución de los procesos haciendo más eficiente el ordenamiento por hilos, siendo este último eficiente debido a la rapidez en sus cálculos.

RECOMENDACIONES

Los métodos de ordenamiento se pueden utilizar para agilizar los procesos siempre y cuando se tenga claro qué información se va a analizar.