METODOS DE ORDENAMIENTO EN PYTHON

CARLOS DAVID MOSQUERA ASTIE

davo_43@hotmail.com

Resumen: En este proyecto se hablará de la del uso y aplicación de los métodos de ordenamiento, también se implementarán con Threads (HILOS), los métodos que se usaran son: Ordenamiento por inserción directa e inserción binaria, ordenamiento por mezcla, ordenamiento por montones o heapsort, ordenamiento por conteo o coutingsort y ordenamiento por el método de radixsort. Se realizará un conteo en el cual se compara el tiempo utilizado con HILOS y sin HILOS; el programa que en que se trabajará es PyCharm.

Palabras claves: Hilos, heapsort, coutingsort, radixsort.

Resume: In this project we will talk about the use and application of sorting methods, also implemented with Threads, the methods to be used are: Direct insertion and binary insertion ordering, mixing ordering, stacking ordering or head Sort, sort ordering or couting sort and sorting by the radix sort method. A count will be performed in which is compares the time used with THREADS and without THREADS.

Keywords: Threads, heapsort, coutingsort, radixsort.

INTRODUCCIÓN

Los algoritmos de ordenamientos nos permiten tal cual como su nombre lo indica, ordenar. Se utilizarán en este caso para ordenar datos numéricos, con valores asignados ya sean manual o aleatoriamente, se trabajará con los métodos mencionados anteriormente. Es conveniente usarlo cuándo se requiere hacer una cantidad considerable de datos y es importante el factor tiempo. Analizaremos la cantidad de estos datos y el tiempo utilizado en cada método de ordenamiento trabajando en PyCharm. El propósito de los métodos de ordenamiento es el de facilitar las búsquedas de los registros dados.

Metodología utilizada: Investigaciones, consultas, videos, internet, libros.

Algoritmos usados:

Algoritmo de inserción directa: consiste en realizar varias pasadas sobre el arreglo, en cada pasada se analiza un elemento, y se intenta crear su orden relativa entre los analizados en pasadas anteriores. Con esto se logra ir manteniendo una lista ordenada constantemente.

Algoritmo de inserción binaria: El método por inserción binaria es una mejora del método de inserción directa. La mejora consiste en realizar una búsqueda binaria en lugar de una búsqueda secuencial, para insertar un elemento en la parte izquierda del arreglo, que ya se encuentra ordenado.

Algoritmo de mezcla: Este algoritmo consiste básicamente en dividir en partes iguales la lista de números y luego mezclarlos comparándolos, dejándolos ordenados. Si se piensa en este algoritmo recursivamente, podemos imaginar que dividirá la lista hasta tener un elemento en cada lista, luego lo compara con el que está a su lado y según corresponda, lo sitúa donde corresponde.

Algoritmo por montones: Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él.

Algoritmo por conteo: Es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales.

Algoritmo por radix: es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotante especialmente formateados, radix sort no está limitado sólo a los enteros.

Lenguaje de programación usado: El programa que se utilizó para realizar este trabajo fue Python, Python es un lenguaje de scripting independiente de plataforma y orientado a objetos, preparado para realizar cualquier tipo de programa, desde aplicaciones Windows a servidores de red o incluso, páginas web. Es un lenguaje interpretado, lo que significa que no se necesita compilar el código fuente para poder ejecutarlo, lo que ofrece ventajas como la rapidez de desarrollo e inconvenientes como una menor velocidad

Código de los algoritmos utilizados:

```
#Metodo de Insercion Directa
#!/usr/bin/python
# -*- coding: utf-8 -*-
import json

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
def insercionBinaria(lista, tam):
    for i in range(1, tam):
        aux = lista[i]
        izq = 0
        der = i - 1
        while izq <= der:
            m = (izq + der) / 2
            if aux < lista[m]:
                der = m - 1
            else:
                izq = m + 1

        j = i - 1
        while j >= izq:
            lista[j + 1] = lista[j]
            j = j - 1
        lista[izq] = aux
def counting_sort(array, maxval):
    """in-place counting sort"""
    n = len(array)
    m = maxval + 1
    count = [0] * m              # init with
zeros
    for a in array:
        count[a] += 1            # count
occurences
    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
```

```
def radix_sort(random_list):
    len_random_list = len(random_list)
    modulus = 10
    div = 1
    while True:
        # empty array, [[] for i in range(10)]
        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)

def mergeSort(alist):
    #("Desordenado ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    #print("Ordenando ",alist)

def heapsort(aList):
    # convert aList to heap
    length = len(aList) - 1
    leastParent = length / 2
```

```
    for i in range(leastParent, -1, -1):
        moveDown(aList, i, length)
    # flatten heap into sorted array

    for i in range(length, 0, -1):
        if aList[0] > aList[i]:
            swap(aList, 0, i)
            moveDown(aList, 0, i - 1)


def moveDown(aList, first, last):
    largest = 2 * first + 1
    while largest <= last:
        # right child exists and is larger than
left child
        if (largest < last) and (aList[largest]
< aList[largest + 1]):
            largest += 1

        # right child is larger than parent
        if aList[largest] > aList[first]:
            swap(aList, largest, first)
            # move down to largest child
            first = largest
            largest = 2 * first + 1
        else:
            return  # force exit

def swap(A, x, y):
    tmp = A[x]
    A[x] = A[y]
    A[y] = tmp
```

```
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)

def heapify(a, count):
    start = int((count-2)/2)
    while start >= 0:
        sift_down(a, start, count-1)
        start -= 1

def sift_down(a, start, end):
    root = start
    while (root*2+1) <= end:
```

```
        child = root * 2 + 1
        swap = root
        if a[swap] < a[child]:
            swap = child
        if (child + 1) <= end and a[swap] <
a[child+1]:
            swap = child+1
        if swap != root:
            a[root], a[swap] = a[swap], a[root]
            root = swap
        else:
            return

def heapsort(a):
    heapify(a, len(a))
    end = len(a)-1
    while end > 0:
        a[end], a[0] = a[0], a[end]
        end -= 1
        sift_down(a, 0, end)
```

Resultados alcanzados: Se ha logrado la implementación exitosa de cada uno de los métodos de ordenamientos explicados, se han comparado cada uno con sus respectivos tiempos y cantidad de datos (1, 2, 5, 10,20) en millones; verificando también el comportamiento de estos al trabajar con y sin HILOS (Threads).

INSERCION DIRECTA

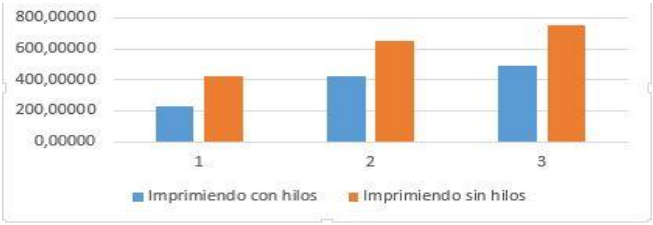
RESULTADOS:

1600000000	Algoritmo - Metodo insercion directa Complejidad = O(n^2) Tiempo en minutos						
Numero de datos							
Millones	Sin imprimir con hilos	Imprimiendo con hilos	Sin imprimir sin hilos	Imprimiendo sin hilos	Tiempo esperado	Tiempo delta con hilos	Tiempo delta sin hilos
1000000	68,71400	343,77000	55,72500	423,76500	3,90625E-07	343,77000	55,72500
2000000	57,43500	519,54600	44,24000	753,56400	1,5625E-06	519,54600	44,24000
5000000	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO
10000000	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO
20000000	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO

INSERCION BINARIA

RESULTADOS:

Algoritmo - Metodo insercion Binaria Complejidad = O(n^2) Tiempo en minutos							
Numero de datos							
Millones	Sin imprimir con hilos	Imprimiendo con hilos	Sin imprimir sin hilos	Imprimiendo sin hilos	Tiempo esperado	Tiempo delta con hilos	Tiempo delta sin hilos
1000000	54,79400	225,76000	52,34300	420,76500	3,90625E-07	225,76000	52,34300
2000000	45,54900	425,43500	46,61300	653,92300	1,5625E-06	425,43500	46,61300
5000000	75,87900	489,87600	46,87890	753,98700	9,76563E-06	489,87599	46,87889
10000000	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO
20000000	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO	COLGADO



METODO POR MEZCLA:

Algoritmo - Metodo por mezcla
Complejidad = O(n log n)
Tiempo en minutos

Numero de datos

Millones	Sin imprimir con hilos	Imprimiendo con hilos	Sin imprimir sin hilos	Imprimiendo sin hilos	Tiempo esperado	Tiempo delta con hilos	Tiempo
1000000	7,51500	51,92200	21,2630	273,84500	-0,002002575	51,52400	
2000000	27,06900	267,26700	203,65500	366,77300	-0,003	267,27000	
5000000	215,49200	1428,24500	832,46500	1649,76000	-0,008	1428,25257	
10000000	31,825000	685,80200	167,46500	1991,66900	-0,013775 75	685,81578	
20000000	150,45100	1845,42200	518,8300	4181,64900	-0,023788625	1845,44579	

METODO RADIX

Algoritmo - Metodo Radix
Complejidad = O(nk)
Tiempo en minutos

Numero de datos

Millones	Sin imprimir con hilos	Imprimiendo con hilos	Sin imprimir sin hilos	Imprimiendo sin hilos	Tiempo esperado	Tiempo delta con hilos	Tiempo delta sin hilos
1000000	9,42200	47,53100	15,91800	90,16800	0,000625	47,53038	15,91768
2000000	8,50300	71,01900	18,9705	147,93500	0,00125	71,01775	18,96875
5000000	149,95200	133,09700	332,89100	310,47900	0,003125	133,09388	332,88988
10000000	89,90200	270,88300	173,37900	918,69900	0,00625	270,87675	173,37275
20000000	161,71200	585,69300	333,90600	1416,55200	0,0125	585,68050	333,89350

METODO RAPIDO

Algoritmo - Metodo rapido(Quicksort)
Complejidad = O(n log n)
Tiempo en minuto

Numero de datos

Millones	Sin imprimir con hilos	Imprimiendo con hilos	Sin imprimir sin hilos	Imprimiendo sin hilos	Tiempo esperado	Tiempo delta con hilos	Tiempo delta sin hilos
1000000	8,43700	14,85900	21,65300	37,77700	-0,00002575	14,85900	21,65500
2000000	9,12500	30,98800	43,2720	91,98700	-0,00262862	30,95163	43,27662
5000000	25,90400	71,88900	169,87100	256,17500	-0,00782594	71,89683	169,87883
10000000	47,47800	126,38400	596,44800	650,33200	-0,01377575	126,39778	596,46178
20000000	52,53700	359,90100	887,78400	2538,06800	-0,02788625	359,92479	887,78869

METODO POR MONTONES:

Algoritmo - Metodo por montones
Complejidad = O(n log n)
Tiempo en minutos

Numero de datos

Millones	Sin imprimir con hilos	Imprimiendo con hilos	Sin imprimir sin hilos	Imprimiendo sin hilos	Tiempo esperado	Tiempo delta con hilos	Tiempo delta sin hil
1000000	7,65300	8,25100	26,342	13,75100	-0,0020625 75	8,25300	26,34
2000000	13,93100	4,43800	48,37500	8,98500	-0,003828862	4,44168	48,37
5000000	12,44900	2,53900	114,7700	6,40700	-0,007828594	2,54583	114,77
10000000	29,88000	3,72400	203,42100	6,57800	-0,013775 75	3,74178	203,43
20000000	940,71900	3,11900	1858,68600	5,71900	0,023788625	3,13279	1858,10

METODO POR CONTEO

Algoritmo - Metodo por Conteo
Complejidad = O(n+k)
Tiempo en minutos

Numero de datos

Millones	Sin imprimir con hilos	Imprimiendo con hilos	Sin imprimir sin hilos	Imprimiendo sin hilos	Tiempo esperado	Tiempo delta con hilos	Tiempo delta sin hil
1000000	9,26600	30,21600	12,47000	57,15900	0,001	30,20938	12,46
2000000	4,48400	54,92600	15,58100	158,44400	0,001	54,91875	15,57
5000000	15,81500	134,07000	48,19000	277,73500	0,003	134,06688	48,09
10000000	49,98800	254,69200	133,08200	551,20900	0,006	254,675 75	133,07
20000000	190,09200	586,35900	488,31500	1259,88700	0,013	586,34650	488,32

CONCLUSIONES

Gracias a esta investigación y aplicación de diferentes métodos se ha logrado analizar y conocer cada uno de los tipos de métodos de ordenamiento, esto nos permitiría reducir un programa extenso haciendo uso del método necesario y así poder obtener mejores resultados y lo mejor en menos tiempo.

Al comparar cada uno de los métodos y sus diferentes funciones se pudo notar el tiempo que se toma cada uno para realizar su tarea, unos de los métodos que realiza más procesos internos son los de inserción, tanto con Directa como Binaria.

De este modo también analizamos que uno de los que toma menos tiempo es el método de Mezcla, con diferentes cantidades de datos (1, 2, 5, 10 y 20) en millones, realizando así su trabajo satisfactoriamente.

RECOMENDACIONES

- Se debe tener uso y manejo previo tanto de los métodos de ordenamiento como de los Hilos (Threads) para poder realizar su trabajo.
- Tener una maquina (PC) con buen procesador y buena capacidad de memoria para correr cada una de las funciones si problemas en el menor tiempo posible.
- Hacer las comparaciones necesarias entre cada uno de los métodos, con diferentes cantidades y observar que tiempo toma al realizarlo con hilos y sin hilos.
- Realiza la investigación necesaria para obtener los conocimientos que te faciliten el manejo de los métodos, ya que cada uno tiene su nivel de complejidad.

COMPUTADOR USADO

COMPAQ Presario CQ43

Procesador Intel(R) Celeron(R) CPU B815 @ 1.60GHz 1.60GHz

Memoria RAM 2,00 GB (1,85 utilizable)

Tipo de Sistema Operativo: 32 bits, procesador x64

Ediciòn de Windows: windows 10 Home

REFERENCIAS BIBLIOGRAFICAS

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

https://www.ecured.cu/Ordenamiento_Radix

http://librosweb.es/libro/algoritmos_python/capitulo_19/ordenamiento_por_insercion.html

https://www.youtube.com/watch?v=PqV1HLIof9U

https://saforas.wordpress.com/2011/01/24/codigo-python-ordenamiento-por-insercion-directa/

http://progpython.blogspot.com.co/2011/09/algoritmos-de-ordenamiento-en-python-el.html

http://making-code.blogspot.com.co/2016/01/implementacion-de-los-metodos-de.html

http://www.codegaia.com/index.php/Complejidad_Algoritmos_de_Ordenamiento