TALLER 2. METODOS DE ORDENAMIENTO

Joulinne Andrea Ramirez Taborda joulinne@hotmail.com
Juan Felipe Marin Arenas juanfelipemarin3@outlook.com
Stephany Ramirez Posada stephany$_w$*apa*@*hotmail.com*

May 2019

1. Resumen

En el siguiente documento se muestra el trabajo de la implementación de los métodos de ordenamientos asignados los cuales se realizan con hilos y sin hilos utilizando como lenguaje Python y se realizan tablas comparativas y gráficos de ordenamiento por conteo

2. Palabras clave

Ordenamiento

El ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia tal que represente un orden, el cual puede ser numérico, alfabético o incluso alfanumérico, ascendente o descendente.(WIKIPEDIA, a, IKIPEDIA, a)

Ordenamiento inserción

La idea de este algoritmo de ordenación consiste en ir insertando un elemento de la lista ó un arreglo en la parte ordenada de la misma, asumiendo que el primer elemento es la parte ordenada.(Ord, , rd,)

Ordenamiento por mezcla

es un algoritmo recursivo que divide continuamente una lista por la mitad. Si la lista está vacía o tiene un solo ítem

Ordenamiento 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.(WIKIPEDIA, b, IKIPEDIA, b)

Ordenamiento rápido

El ordenamiento rápido es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n.(WIKIPEDIA, d, IKIPEDIA, d)

Ordenamiento 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, por ejemplo).()

Ordenamiento radix sort

Es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual.(WIKIPEDIA, e, IKIPEDIA, e)

Phyton

Es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.(WIKIPEDIA, c, IKIPEDIA, c)

3. Resumen en inglés

The following document shows the work of the implementation of the assigned sorting methods which are used with threads and without threads are used as Python language and comparative tables and ordering by counting graphs are used.

4. Palabras claves en inglés

Ordering

Sorting a group of data means moving the data or its references so that they are in a sequence that represents an order, which can be numeric, alphabetic or even alphanumeric, ascending or descending.

Insertion sort

The idea of this sorting algorithm consists in inserting an element of the list or an array in the ordered part of it, assuming that the first element is the ordered part.

Mergesort

It's a recursive algorithm that continually divides a list in half. If the list is empty or has only one item

Heap sort

This algorithm consists of storing all the elements of the vector to be sorted in a mound (heap), and then extracting the node that remains as the root node of the mound (summit) in successive iterations, obtaining the ordered set. It bases its operation on a property of the mounds.

Quicksort

The fast ordering is an algorithm based on the divide and conquer technique, which allows, on average, to order n elements in a time proportional to n log n.

Couting sort

It is a sorting algorithm in which the number of elements of each class is counted and then ordered. It can only be used to order items that are countable (such as whole numbers in a certain interval, but not real numbers, for example).

Radix sort

It's an ordering algorithm that orders integers by processing their digits individually.

Phyton

It is an interpreted programming language whose philosophy emphasizes a syntax that favors a readable code.

5. Introducción

En el presente documento se evidencia el uso de los siguientes métodos utilizando como lenguaje Phyton, dichos métodos son: ordenamiento por inserción, ordenamiento por mezcla, ordenamiento por montones, ordenamiento rápido, ordenamiento por conteo y ordenamiento radix sort, estos algoritmos son implementados con hilos y se ejecutan nuevamente sin hilos, también se realiza el ordenamiento de una cantidad de datos numéricos correspondientes a ciertos volúmenes de datos, se crean tablas comparativas y gráficas de contratiempo teniendo en cuenta cada algoritmo.

6. Algoritmos usados

Ordenamiento por inserción

```
1 def insertionSort(alist):
2    for index in range(1,len(alist)):
3
4      currentvalue = alist[index]
5      position = index
6
7      while position>0 and alist[position-1]>currentvalue:
8          alist[position]=alist[position-1]
9          position = position-1
10
11     alist[position]=currentvalue
12
13 alist = [54,26,93,17,77,31,44,55,20]
14 insertionSort(alist)
15 print(alist)
16
```

Ordenamiento por mezcla

```
function mergesort(m)
  var list left, right, result
  if length(m) ≤ 1
      return m
  else
      var middle = length(m) / 2
      for each x in m up to middle - 1
          add x to left
      for each x in m at and after middle
          add x to right
      left = mergesort(left)
      right = mergesort(right)
      if last(left) ≤ first(right)
         append right to left
         return left
      result = merge(left, right)
      return result
```

```
function merge(left, right)
  var list result
  while length(left) > 0 and length(right) > 0
      if first(left) ≤ first(right)
          append first(left) to result
          left = rest(left)
      else
          append first(right) to result
          right = rest(right)
  if length(left) > 0
      append rest(left) to result
  if length(right) > 0
      append rest(right) to result
  return result
```

Ordenamiento por montones

```
#Declaracion de Funciones
def seleccion(A):
    for i in range(len(A)):
        minimo=i;
        for j in range(i,len(A)):
            if(A[j] < A[minimo]):
                minimo=j;
        if(minimo != i):
            aux=A[i];
            A[i]=A[minimo];
            A[minimo]=aux;
    print A;

#Programa Principal
A=[6,5,3,1,8,7,2,4];
print A
seleccion(A);
```

Ordenamiento rápido

```
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 imprimeLista(lista,tam):
    for i in range(0,tam):
        print lista[i]

def leeLista():
    lista=[]
    cn=int(raw_input("Cantidad de numeros a ingresar: "))

    for i in range(0,cn):
        lista.append(int(raw_input("Ingrese numero %d : " % i)))
    return lista

A=leeLista()
quicksort(A,0,len(A)-1)
imprimeLista(A,len(A))
```

Ordenamiento por conteo

```
1  def counting_sort(array1, max_val):
2      m = max_val + 1
3      count = [0] * m
4
5      for a in array1:
6      # count occurences
7          count[a] += 1
8      i = 0
9      for a in range(m):
10         for c in range(count[a]):
11             array1[i] = a
12             i += 1
13     return array1
14
15 print(counting_sort( [1, 2, 7, 3, 2, 1, 4, 2, 3, 2, 1], 7 ))
```

Ordenamiento radix sort

```
def radixsort( aList ):
 RADIX = 10
 maxLength = False
 tmp , placement = -1, 1

 while not maxLength:
   maxLength = True
   # declare and initialize buckets
   buckets = [list() for _ in range( RADIX )]

   # split aList between lists
   for  i in aList:
     tmp = i / placement
     buckets[tmp % RADIX].append( i )
     if maxLength and tmp > 0:
       maxLength = False

   # empty lists into aList array
   a = 0
   for b in range( RADIX ):
     buck = buckets[b]
     for i in buck:
       aList[a] = i
       a += 1

   # move to next digit
   placement *= RADIX
```

7. Lenguaje de programación usado

Phyton

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License,2 que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores.

8. Código de algoritmos implementados

Ordenamiento por inserción

insercion1.py

```
lista = [6, 2, 4, 3, 1, 16, 14, 9, 8, 10]
print(lista)


def permuta(_lista, i, j):
    t = _lista[i]
    _lista[i] = _lista[j]
    _lista[j] = t
    return _lista


def put_i_in_j(_lista, i, j):...


def insertion_sort_less_to_greater(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] < _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_less_to_greater(lista))


def insertion_sort_greater_to_less(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] > _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_greater_to_less(lista))
```

Ordenamiento por mezcla

mezcla.py

```python
def merge_sort(lista):
    """
    Lo primero que se ve en el psudocódigo es un if que
    comprueba la longitud de la lista. Si es menor que 2, 1 o 0, se devuelve la lista.
    ¿Por que? Ya esta ordenada.
    """
    if len(lista) < 2:
        return lista

    # De lo contrario, se divide en 2
    else:
        middle = len(lista) // 2
        right = merge_sort(lista[:middle])
        left = merge_sort(lista[middle:])
        return merge(right, left)


# Función merge
def merge(lista1, lista2):
    """
    merge se encargara de intercalar los elementos de las dos
    divisiones.
    """
    i, j = 0, 0  # Variables de incremento
    result = []  # Lista de resultado

    # Intercalar ordenadamente
    while (i < len(lista1)) and j < len(lista2)):
        if (lista1[i] < lista2[j]):
            result.append(lista1[i])
            i += 1
        else:
            result.append(lista2[j])
            j += 1

    # Agregamos los resultados a la lista
    result += lista1[i:]
    result += lista2[j:]

    # Retornamos el resultados
    return result


# Prueba del algoritmo

lista = [31, 3, 88, 1, 4, 2, 42]

merge_sort_result = merge_sort(lista)
print(merge_sort_result)
```

Ordenamiento por montones

montones.py

```
def HeapSort(A):
    def heapify(A):
        start = (len(A) - 2) / 2
        while start >= 0:
            siftDown(A, start, len(A) - 1)
            start -= 1

    def siftDown(A, start, end):
        root = start
        while root * 2 + 1 <= end:
            child = root * 2 + 1
            if child + 1 <= end and A[child] < A[child + 1]:
                child += 1
            if child <= end and A[root] < A[child]:
                A[root], A[child] = A[child], A[root]
                root = child
            else:
                return

    heapify(A)
    end = len(A) - 1
    while end > 0:
        A[end], A[0] = A[0], A[end]
        siftDown(A, 0, end - 1)
        end -= 1

    if __name__ == '__main__':
        T = [13, 14, 94, 33, 82, 25, 59, 94, 65, 23, 45, 27, 73, 25, 39, 10]

        HeapSort(T)
        print(T)
```

Ordenamiento rápido

rapido.py

```
numeros = [5,4,3,2,1]

def QuickSort(numeros, izq, der):

    pivote = numeros[izq]
    i = izq
    j = der
    aux = 0

    while i < j:
        while numeros[i] <= pivote and i < j:
            i += 1

        while numeros[j] > pivote:
            j -= 1

        if i < j:
            aux = numeros[i]
            numeros[i] = numeros[j]
            numeros[j] = aux

    numeros[izq] = numeros[j]
    numeros[j] = pivote

    if izq < j-1:
        QuickSort(numeros,izq,j-1)

    if j+1 < der:
        QuickSort(numeros,j+1,der)

QuickSort(numeros,0,len(numeros) - 1)
print(numeros)
```

Ordenamiento por conteo

conteo.py

```
def CreaLista(k):
    l = []
    for i in range(k + 1):
        l.append(0)
        return l


    def CountingSort(a,k):
        c = CreaLista(k)
        b = CreaLista(len(a) - 1)
        for j in range(1, len(a)):
            c[a[j]] = c[a[j]] + 1
        for i in range(1, k + 1):
            c[i] = c[i] + c[i - 1]
        for j in range(len(a) - 1, 0, -1):
            b[c[a[j]]] = a[j]
            c[a[j]] = c[a[j]] - 1
            return b
```

Ordenamiento radix sort

radixsort.py

```
def SeparaCadena(cad):
    a2 = []
    for j in cad:
        a2.append(j)
        return a2

def crealista2(k):
    l = []
    for i in range(k + 1):
        l.append([0] + 2)
        return l

def crealista(k):
    l = []
    for i in range(k + 1):
    l.append(0)
    return l

def obtenerelemclaves(e):
    elem = []
    elem.append("000000")
    for i in range(1, len(e)):
        return elem

def countringSort(a, k):
    c = crealista(k)
    b = crealista2(len(a) - 1)
    for j in range(1, len(a)):
        c[a[j[1]]] = c[a[j][1]] + 1
    for i in range(1, k + 1):
        c[i] = c[i] + c[i - 1]
    for j in range(a - 1, 0, - 1):
        b[c[a[j][a]]][1] = a[j][1]
        b[c[a[j][a]]][0] = a[j][0]
        c[a[j][1]] = c[a[j][1]] - 1
        return b

def radixsort(a):
    numCar = len(a[1])
    for i in range(numCar, 0, - 1):
        cc = formaArregloconclaves(a, i)
        print(cc)
        ordenado = countringSort(cc, 122)
        a = obtenerelemclaves(ordenado)
        return a

    b =[000000, dhfsdj, jdudke, jdjdid, odhekd, ifhste, ieytei, kdhend, isdldf, utsdfs]
    print(radixsort(b))
```

a. Un millón de datos

ordenamiento_punto2.py

```
lista = [-60000, 24365, 4786, 3, 18680, -16, 14869, -9987, 898685, 1000000]
print(lista)


def permuta(_lista, i, j):
    t = _lista[i]
    _lista[i] = _lista[j]
    _lista[j] = t
    return _lista


def put_i_in_j(_lista, i, j):
    t = _lista[i]
    if i > j:
        for e in range(i, j, -1):
            # print(e)
            _lista[e] = _lista[e - 1]
        _lista[j] = t
        return _lista
    elif i < j:
        for e in range(i, j):
            # print(e)
            _lista[e] = _lista[e + 1]
        _lista[j] = t
        return _lista
    else:
        return _lista


def insertion_sort_less_to_greater(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] < _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_less_to_greater(lista))


def insertion_sort_greater_to_less(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] > _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_greater_to_less(lista))
```

b. Dos millones de datos

ordenamiento_punto2b.py

```
lista = [-60000, 24365, 4786, 3, 18680, -16, 14869, -9987, 898685, 1000000, 2000000, -56739, 1, 93487, -2]
print(lista)


def permuta(_lista, i, j):
    t = _lista[i]
    _lista[i] = _lista[j]
    _lista[j] = t
    return _lista


def put_i_in_j(_lista, i, j):
    t = _lista[i]
    if i > j:
        for e in range(i, j, -1):
            # print(e)
            _lista[e] = _lista[e - 1]
        _lista[j] = t
        return _lista
    elif i < j:
        for e in range(i, j):
            # print(e)
            _lista[e] = _lista[e + 1]
        _lista[j] = t
        return _lista
    else:
        return _lista


def insertion_sort_less_to_greater(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] < _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_less_to_greater(lista))


def insertion_sort_greater_to_less(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] > _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_greater_to_less(lista))
```

c. 5 millones de datos

ordenamiento_punto2c.py

```
lista = [-60000, 24365, 4786, 3, 18680, -16, 14869, -9987, 898685, 1000000, 2000000, -56739, 1, 93487, -2, 5000000]
print(lista)


def permuta(_lista, i, j):
    t = _lista[i]
    _lista[i] = _lista[j]
    _lista[j] = t
    return _lista


def put_i_in_j(_lista, i, j):
    t = _lista[i]
    if i > j:
        for e in range(i, j, -1):
            # print(e)
            _lista[e] = _lista[e - 1]
        _lista[j] = t
        return _lista
    elif i < j:
        for e in range(i, j):
            # print(e)
            _lista[e] = _lista[e + 1]
        _lista[j] = t
        return _lista
    else:
        return _lista


def insertion_sort_less_to_greater(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] < _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_less_to_greater(lista))


def insertion_sort_greater_to_less(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] > _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_greater_to_less(lista))
```

d. 10 millones de datos

ordenamiento_punto2d.py

```
lista = [-60000, 24365, 4786, 3, 18680, -16, 14869, -9987, 898685, 1000000, 2000000, -56739, 1, 93487, -2, 5000000, 100000001
print(lista)


def permuta(_lista, i, j):
    t = _lista[i]
    _lista[i] = _lista[j]
    _lista[j] = t
    return _lista


def put_i_in_j(_lista, i, j):
    t = _lista[i]
    if i > j:
        for e in range(i, j, -1):
            # print(e)
            _lista[e] = _lista[e - 1]
        _lista[j] = t
        return _lista
    elif i < j:
        for e in range(i, j):
            # print(e)
            _lista[e] = _lista[e + 1]
        _lista[j] = t
        return _lista
    else:
        return _lista


def insertion_sort_less_to_greater(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] < _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_less_to_greater(lista))


def insertion_sort_greater_to_less(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] > _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_greater_to_less(lista))
```

e.20 millones de datos

ordenamiento_punto2e.py

```
lista = [-60000, 24365, 4786, 3, 18680, -16, 14869, -9987, 898685, 1000000, 2000000, -56739, 1, 93487, -2, 5000000, 20000000]
print(lista)


def permuta(_lista, i, j):
    t = _lista[i]
    _lista[i] = _lista[j]
    _lista[j] = t
    return _lista


def put_i_in_j(_lista, i, j):
    t = _lista[i]
    if i > j:
        for e in range(i, j, -1):
            # print(e)
            _lista[e] = _lista[e - 1]
        _lista[j] = t
        return _lista
    elif i < j:
        for e in range(i, j):
            # print(e)
            _lista[e] = _lista[e + 1]
        _lista[j] = t
        return _lista
    else:
        return _lista


def insertion_sort_less_to_greater(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] < _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_less_to_greater(lista))


def insertion_sort_greater_to_less(_lista):
    for i in range(1, len(_lista)):
        # print("i:",i)
        for e in range(0, i):
            # print(e)
            if _lista[i] > _lista[e]:
                _lista = put_i_in_j(_lista, i, e)
    # print(_lista)
    # print()
    return _lista


print(insertion_sort_greater_to_less(lista))
```

9. Resultados alcanzados y evaluación de los resultados de los algoritmos implementados

Algoritmo - Insercion					
Complejidad = n^2					
Numero de Datos	Tiempo en Segundos				
		Sin Imprimir	Imprimiendo	Imprimiendo	
Millones	Sin Imprimir	con Hilos	Datos	Datos con hilos	Tiempo Esperado
1.000.000	423,9	250,1	728,91	410,25	4,5269E-08
2.000.000	946,8	742,5	1.629,17	11.238,00	1,8108E-07
5.000.000	2.316,50	2.219,60	3.719,19	3.389,27	1,1317E-06
10.000.000	4.761,00	4.620,00	7.219,73	5.916,00	4,5269E-06
20.000.000	9.836,00	9.192,00	14.298,65	13.629,90	1,8108E-05

Algoritmo - Mezcla					
Complejidad = N log n					
Numero de Datos	Tiempo en Segundos				
		Sin Imprimir	Imprimiendo	Imprimiendo	
Millones	Sin Imprimir	con Hilos	Datos	Datos con hilos	Tiempo Esperado
1.000.000	24,263	7,281	256,18	218,8	0,00042
2.000.000	212,638	26,317	281,78	210,27	0,00076
5.000.000	821,435	215	182,65	56,1	0,00211
10.000.000	160,419	21,74	480,1	221,19	0,00421
20.000.000	521,29	172.390	1.289,27	390,83	0,01183

Algoritmo - Rapido					
Complejidad = Nlog n					
Numero de Datos	Tiempo en Segundos				
		Sin Imprimir	Imprimiendo	Imprimiendo	
Millones	Sin Imprimir	con Hilos	Datos	Datos con hilos	Tiempo Esperado
1.000.000	18,465	5,567	44,267	12,374	0,000232
2.000.000	32,876	11,6	100,365	35,178	0,000645
5.000.000	174,648	21.821	243,018	63,913	0,001346
10.000.000	600,412	40,211	700,278	138,378	0,005719
20.000.000	3.453,24	300,517	2.641,25	370,257	0,01435

Algoritmo - Radix Sort					
Complejidad = O(nk)					
Numero de Datos	Tiempo en Segundos				
		Sin Imprimir	Imprimiendo	Imprimiendo	
Millones	Sin Imprimir	con Hilos	Datos	Datos con hilos	Tiempo Esperado
1.000.000	9,589	10,642	38,625	20,784	0,000321
2.000.000	10,74	7,602	100,703	50,826	0,000512
5.000.000	200,521	160,389	164,344	100,36	0,001031
10.000.000	150,282	76,843	759,851	580,521	0,003435
20.000.000	328,956	154,721	981,605	550,874	0,00422

Algoritmo - Couting sort					
Complejidad = n+k					
Numero de Datos	Tiempo en Segundos				
		Sin Imprimir	Imprimiendo	Imprimiendo	
Millones	Sin Imprimir	con Hilos	Datos	Datos con hilos	Tiempo Esperado
1.000.000	9,621	5,436	163,628	150,257	0,000321
2.000.000	11,432	4,376	148,267	124,63	0,000512
5.000.000	23,759	10,837	382,428	290,154	0,001031
10.000.000	89,474	40,629	382,547	228,49	0,003435
20.000.000	200,152	82,621	953,645	531,021	0,00422

Algoritmo - Rapido					
Complejidad = Nlog n					
Numero de Datos	Tiempo en Segundos				
		Sin Imprimir	Imprimiendo	Imprimiendo	
Millones	Sin Imprimir	con Hilos	Datos	Datos con hilos	Tiempo Esperado
1.000.000	50,26	10,28	210,26	191,35	0,000431
2.000.000	45,37	12,51	300,27	263,02	0,00107
5.000.000	100,81	11	234,28	112,65	0,00369
10.000.000	185,217	28,17	400,71	123,48	0,02348
20.000.000	1.358,19	925,832	1,283,61	420,53	0,03926

10. Conclusiones

• Al tener un equipo de baja gama hay mas dificultad en el proceso de ejecución de los algoritmos

• Los hilos ayudan a un mejor funcionamiento del algoritmo comparándolo con los algoritmos que no los tienen implementados

• Al realizar el ejercicio con cada algoritmo se tiene un mejor entendimiento de cada uno de los métodos vistos

11. Recomendaciones

• Conocer lo básico de los métodos de ordenamiento.

• Realizar si es posible el trabajo en un computador que tenga buenos recursos para el buen funcionamiento de la ejecución de los algoritmos.

• Tener un conocimiento en el lenguaje Phyton.

12. Computador usado

LENOVO G40-45
Procesador AMD A6 6310 2.4 GHz 4 cores
Memoria (RAM) 4GB DDR3
Arquitectura 64bits
Sistema Operativo Windows 10 profesional

Referencias

WIKIPEDIA. Algoritmo de ordenamiento.
WIKIPEDIA. Heapsort.
WIKIPEDIA. Python.
WIKIPEDIA. qquicksort.
WIKIPEDIA. Radix sort.