Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de algoritmos

Profesora. **Luz María Sánchez García**

PRÁCTICA 2

ALGORITMOS DE ORDENAMIENTO ITERATIVOS VS RECURSIVOS

Grupo: 3CM13

Equipo: CODEART

Integrantes:
1. Shim Kyuseop
2. Ortiz Jiménez José Antonio
3. Quintero Maldonado Iván

Fecha de entrega: 18 mar de 2022

Planteamiento del problema

En esta práctica, compararemos la diferencia entre los algoritmos iterativos y los algoritmos recursivos con análisis temporal y ejecutando cada uno.

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, el algoritmo ira comparando un elemento de la parte desordenada de la lista con los elementos de la parte ordenada, insertando el elemento en la posición correcta dentro de la parte ordenada, y así sucesivamente hasta obtener la lista ordenada.

Selección

Consiste en encontrar el menor de todos los elementos del vector e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño, y así sucesivamente hasta ordenarlo todo.

Shell

Se utiliza una segmentación entre los datos. Funciona comparando elementos que estén distantes; la distancia entre comparaciones decrece conforme el algoritmo se ejecuta hasta la última fase, en la cual se comparan los elementos adyacentes, por esta razón se le llama ordenación por disminución de incrementos.

La ordenación de Shell usa una secuencia, h1, h2, . . ., ht, conocida como la secuencia de incrementos. Al principio de todo proceso, se fija una secuencia decreciente de incrementos. Cualquier secuencia funcionará en tanto que empiece con un incremento grande, pero menor al tamaño del arreglo de los datos a ordenar, y que el último valor de dicha secuencia sea 1.

Mezcla

Se basa en el principio del algoritmo divide y vencerás. Funciona dividiendo el array en dos mitades repetidamente hasta que obtenemos el array dividido en elementos individuales. Un elemento individual es un array ordenado en sí mismo. El ordenamiento por mezcla combina repetidamente estas pequeñas matrices ordenadas para producir submatrices ordenadas más grandes hasta que obtenemos un array final ordenada.

Rápido

El ordenamiento rápido funciona dividiendo el array en dos partes alrededor de un elemento pivote seleccionado. Mueve los elementos más pequeños a la izquierda del pivote y los más grandes a la derecha. Después de esto, las subpartes izquierda y derecha se ordenan recursivamente para ordenar toda el array. Se denomina ordenamiento rápido porque es unas 2 o 3 veces más rápida que los algoritmos de ordenación habituales.

Actividades

1. El programa de los algoritmos ordenamientos: Inserción, Selección, Shell, Mezcla y Rápido

   Consulte las secciones Prueba y Anexo

2. El programa deberá ser capaz de recibir 10,000,000 números

   Consulte las secciones Prueba y Anexo

3. Gráfica de barras que compra el tiempo que tarda realización de ordenamiento

   Consulte las secciones Prueba y Anexo

4. **Análisis temporal**

Inserción

```
def insertion_sort(list: List[int]):
    for i in range(1, len(list)):
        j = i
        while j > 0 and list[j - 1] > list[j]:
            list[j - 1], list[j] = list[j], list[j - 1]
            j -= 1
```

- k-1 veces de j = 1
- (k-1)(k-1) veces list[j-1] > list[j]
- (k-1)(k-1) veces list[j-1], list[j] = list[j], list[j-1]
- (k-1)(k-1) veces j-=1

Finalmente $T(n) = 3(k-1)^2 + (k-1) = 3k^2 - 5k + 2 \ \in O(n^2)$

Selección

```
def selection_sort(list: List[int]):
    # Linear search
    for i in range(len(list)):
        index = i
        # Search the minimum value in the subarray
        for j in range(i + 1, len(list)):
            if list[index] > list[j]:
                index = j
        list[i], list[index] = list[index], list[i]
```

- k veces de Index=i
- k(k-1) veces comparación de list[index]>list[j]
- k(k-1) veces de index = j
- k veces de list[i], list[index] = list[index], list[i]

Finalmente $T(n) = 2k(k-1) + 2k = 2k^2 \in O(n^2)$

Shell

```
def shell_sort(arr):
    N = len(arr)
    h = N // 2
    while h > 0:
        for i in range(h, N):
            temp = arr[i]
            j = i - h
            while j >= 0 and arr[j] > temp:
                arr[j + h] = arr[j]
                j -= h
            arr[j + h] = temp
        h //= 2
```

En ordenamiento shell, su complejidad temporal depende como define "gap". De acuerdo con el teorema de Poonen, la complejidad del peor caso para la ordenación de shell es Θ(Nlog N)2/log log No Θ(Nlog N)2/(log log N)2)o Θ(Nlog N)2/log log N) o algo así entre. Utilizaremos Θ(N(log N)2

$$T(n) = O(\mathrm{n}(\log \mathrm{n})^2)$$

Mezcla

```python
def merge_sort(arr):
    if len(arr) > 1:
        # Finding the mid of the array
        mid = len(arr)//2
        # Dividing the array elements
        L = arr[:mid]
        # into 2 halves
        R = arr[mid:]
        # Sorting the first half
        merge_sort(L)
        # Sorting the second half
        merge_sort(R)
        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
        k += 1
        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Calculamos cota de complejidad utilizando teorema maestro

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$a = 2, \;\; b = 2, \;\; f(n) = n$$

$$n^{\log_2 2} = n^1 = n$$

$$f(n) = \theta(n^{\log_2 2})$$

$$Por\ lo\ tanto, es\ caso\ 2$$

$$T(n) = \theta(n * \log\ n) = \theta(n\ log\ n)$$

Rápido

```
def partition(start, end, array):
    # Initializing pivot's index to start
    pivot_index = start
    pivot = array[pivot_index]
    # This loop runs till start pointer crosses
    # end pointer, and when it does we swap the
    # pivot with element on end pointer
    while start < end:
        # Increment the start pointer till it finds an
        # element greater than  pivot
        while start < len(array) and array[start] <= pivot:
            start += 1
        # Decrement the end pointer till it finds an
        # element less than pivot
        while array[end] > pivot:
            end -= 1
        # If start and end have not crossed each other,
        # swap the numbers on start and end
        if(start < end):
            array[start], array[end] = array[end], array[start]

    # Swap pivot element with element on end pointer.
    # This puts pivot on its correct sorted place.
    array[end], array[pivot_index] = array[pivot_index], array[end]

    # Returning end pointer to divide the array into 2
    return end

def quick_sort(start, end, array):
    if (start < end):
        # p is partitioning index, array[p]
        # is at right place
        p = partition(start, end, array)
        # Sort elements before partition
        # and after partition
        quick_sort(start, p - 1, array)
        quick_sort(p + 1, end, array)
```

En peor caso, complejidad de ordenamiento rápido es

$$T(n) = T(n-1) + n$$

$$T(n-1) = T(n-2) + (n-1)$$

$$T(n-2) = T(n-3) + (n-2)$$

$$Y\ si\ sustituimos\ T(n-2)\ a\ T(n-1)\ y\ T(n-1)\ a\ T(n)$$

$$T(n) = T(n-2) + (n-1) + n$$

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

.

.

$$T(n) = T(n-k) + kn - \frac{k(k-1)}{2}$$

$$cuando\ k = n-1$$

$$T(n) = T(1) + n(n-1) - \frac{(n-1)(n-2)}{2}$$

$$T(1) = 1$$

$$T(n) = n^2 - n - \frac{n^2 - 3n + 2}{2} + 1$$

$$T(n) = n^2 - n - \frac{n^2 - 3n + 2}{2} + 1$$

$$T(n) = n^2 - n - \frac{n^2 - 3n + 2}{2} + 1$$

$$T(n) = \frac{n^2 + n}{2} \in O(n^2)$$

5. Tabla de aproximación polinomial del comportamiento temporal

Inserción

K	INSTRUCCIONES
100	29,502
1,000	2,995,002
5,000	74,975,002
10,000	299,950,002
50,000	7,499,750,002
100,000	29,999,500,002
200,000	119,999,000,002
400,000	479,998,000,002
600,000	1,079,997,000,002
800,000	1,919,996,000,002
1,000,000	2,999,995,000,002
2,000,000	11,999,990,000,002
3,000,000	26,999,985,000,002
4,000,000	47,999,980,000,002
5,000,000	74,999,975,000,002
6,000,000	107,999,970,000,002
7,000,000	146,999,965,000,002
8,000,000	191,999,960,000,002
9,000,000	242,999,955,000,002
10,000,000	299,999,950,000,002

Selección

K	INSTRUCCIONES
100	20,000
1,000	2,000,000
5,000	50,000,000
10,000	200,000,000
50,000	5,000,000,000
100,000	20,000,000,000
200,000	80,000,000,000
400,000	320,000,000,000
600,000	720,000,000,000
800,000	1,280,000,000,000

1,000,000	2,000,000,000,000
2,000,000	8,000,000,000,000
3,000,000	18,000,000,000,000
4,000,000	32,000,000,000,000
5,000,000	50,000,000,000,000
6,000,000	72,000,000,000,000
7,000,000	98,000,000,000,000
8,000,000	128,000,000,000,000
9,000,000	162,000,000,000,000
10,000,000	200,000,000,000,000

Shell

K	INSTRUCCIONES
100	400
1,000	9,000
5,000	68,412
10,000	160,000
50,000	1,104,016
100,000	2,500,000
200,000	5,620,184
400,000	12,553,230
600,000	20,032,219
800,000	27,877,177
1,000,000	36,000,000
2,000,000	79,405,958
3,000,000	125,859,299
4,000,000	174,348,785
5,000,000	224,380,996
6,000,000	275,660,006
7,000,000	327,987,570
8,000,000	381,221,211
9,000,000	435,253,400
10,000,000	490,000,000

Mezcla

K	INSTRUCCIONES

100	200
1,000	3,000
5,000	18,495
10,000	40,000
50,000	234,949
100,000	500,000
200,000	1,060,206
400,000	2,240,824
600,000	3,466,891
800,000	4,722,472
1,000,000	6,000,000
2,000,000	12,602,060
3,000,000	19,431,364
4,000,000	26,408,240
5,000,000	33,494,850
6,000,000	40,668,908
7,000,000	47,915,686
8,000,000	55,224,720
9,000,000	62,588,183
10,000,000	70,000,000

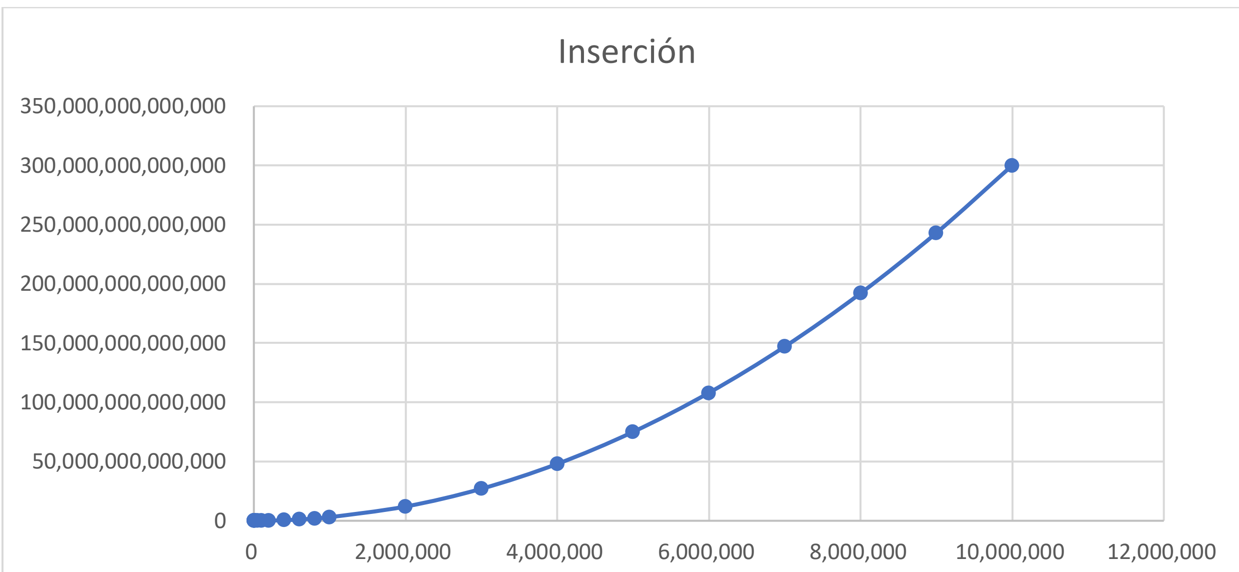
Rápido

K	INSTRUCCIONES
100	5,050
1,000	500,500
5,000	12,502,500
10,000	50,005,000
50,000	1,250,025,000
100,000	5,000,050,000
200,000	20,000,100,000
400,000	80,000,200,000
600,000	180,000,300,000
800,000	320,000,400,000
1,000,000	500,000,500,000
2,000,000	2,000,001,000,000
3,000,000	4,500,001,500,000

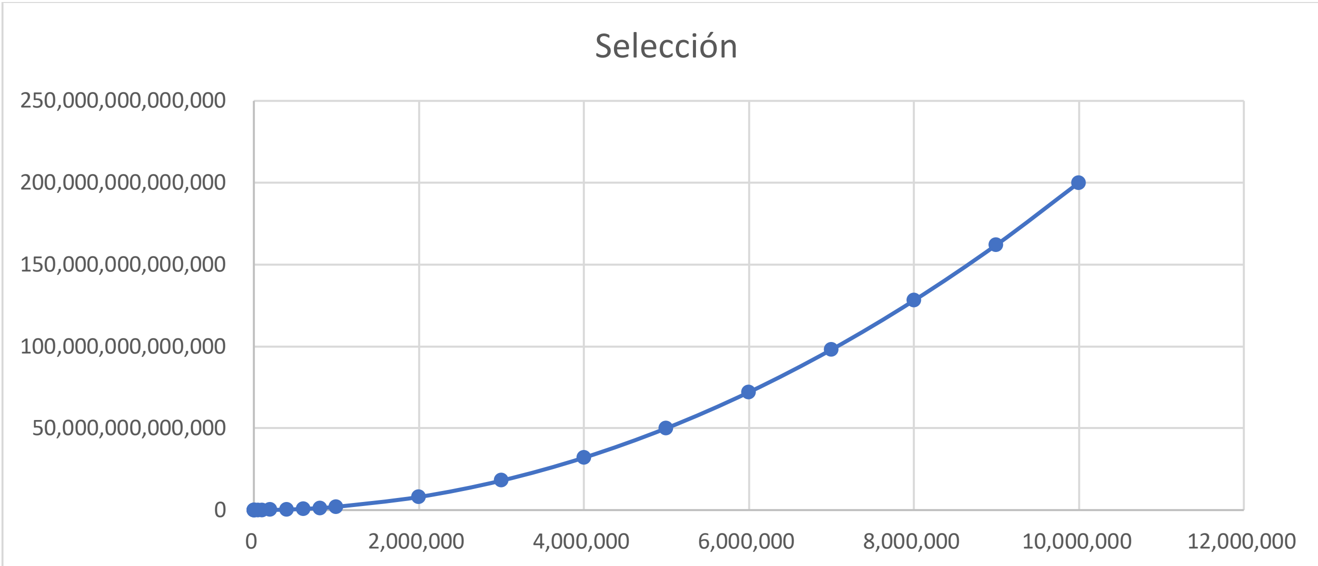
4,000,000	8,000,002,000,000
5,000,000	12,500,002,500,000
6,000,000	18,000,003,000,000
7,000,000	24,500,003,500,000
8,000,000	32,000,004,000,000
9,000,000	40,500,004,500,000
10,000,000	50,000,005,000,000

6. Gráficas de comportamiento temporal de cada algoritmo

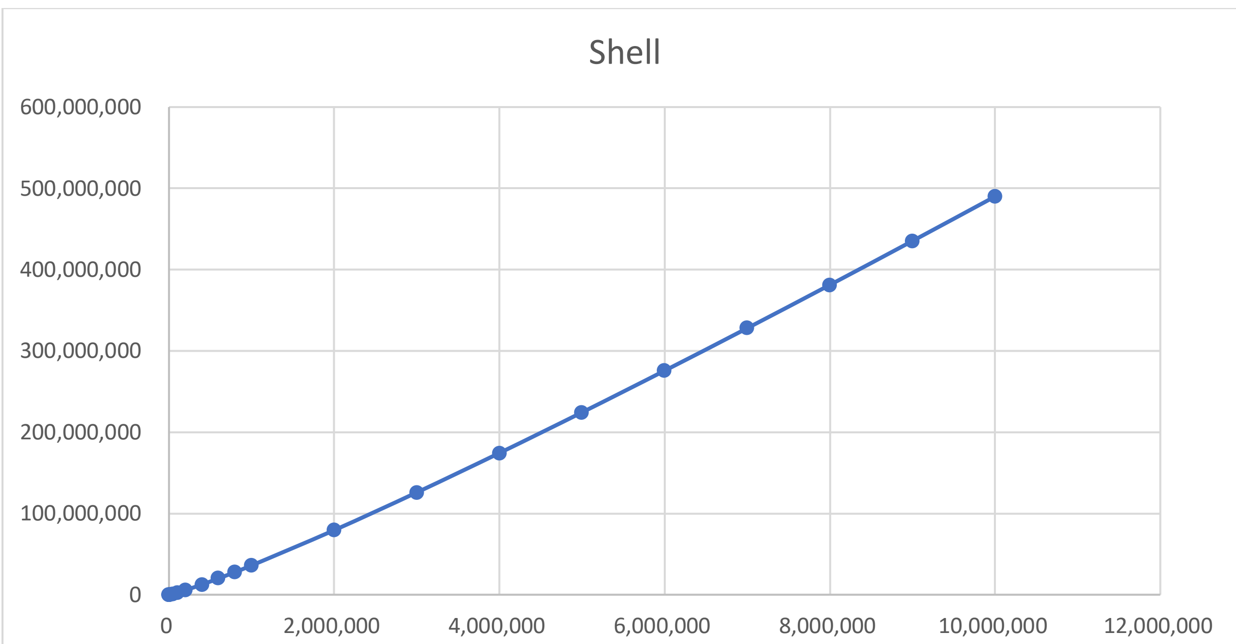
Inserción



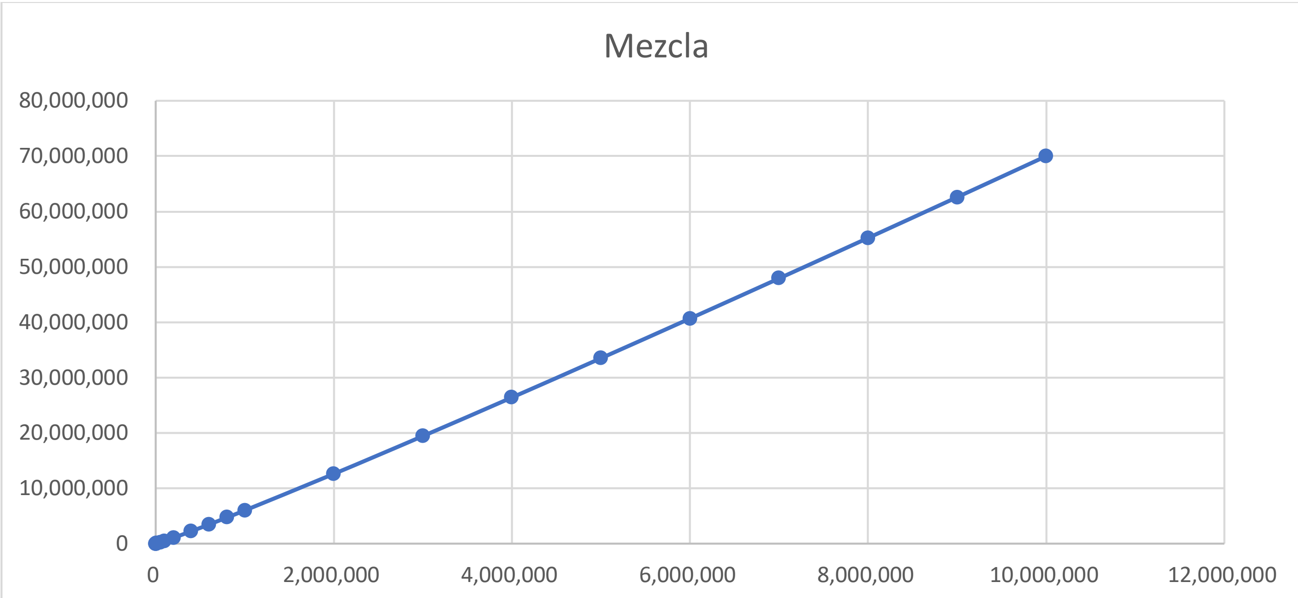
Selección



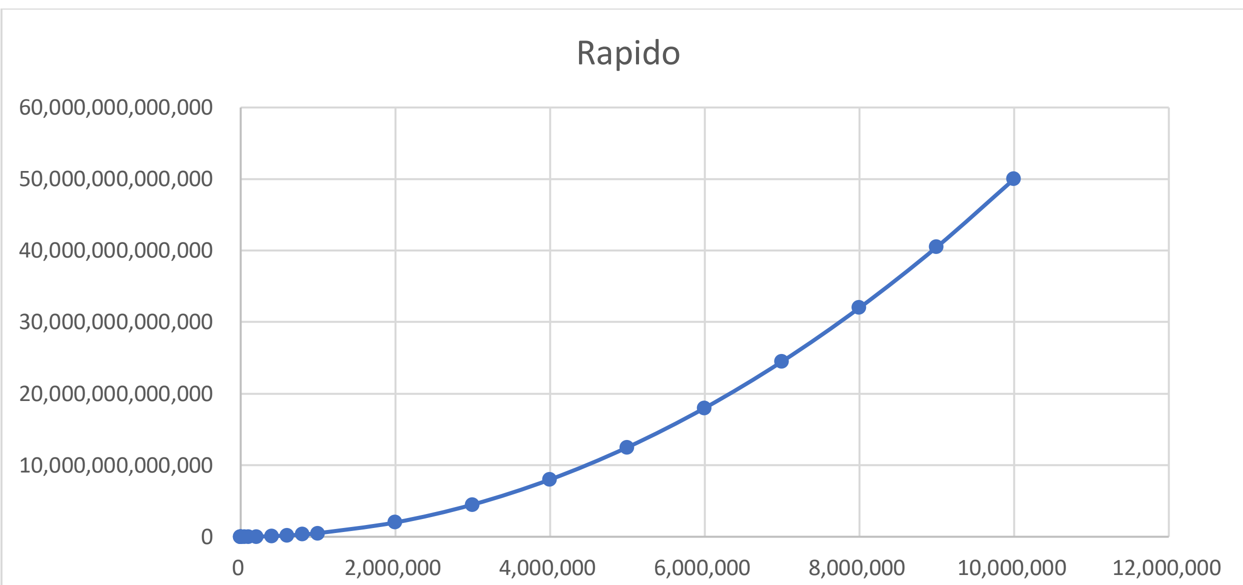
Shell



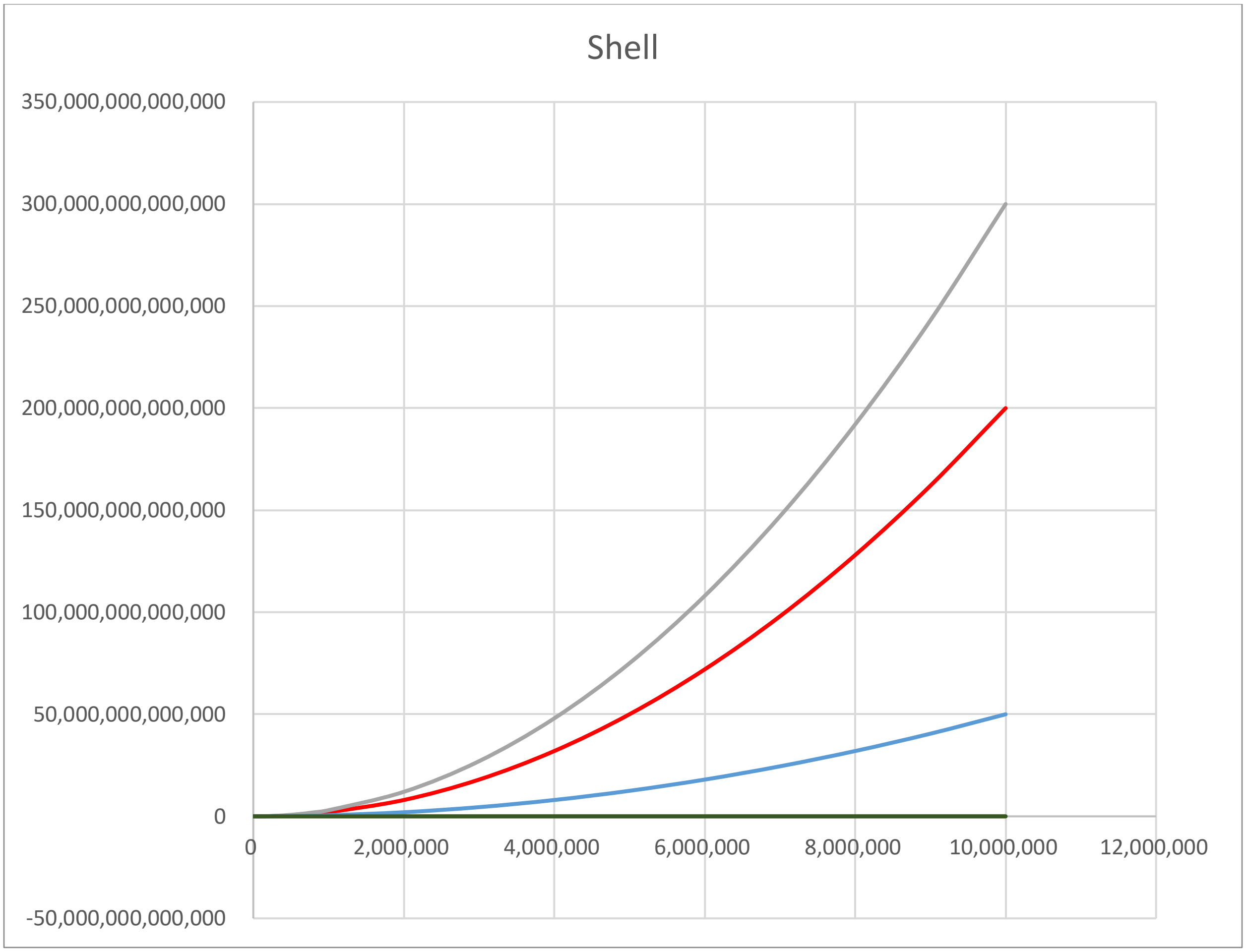
Mezcla



Rápido



7. Comparación de todos algoritmos en una sola gráfica



Insercion

Seleccion

Rapido

Shell (casi no se ve porque esta pegado con Mezcla)

Mezcla

8. Operaciones básicas

Inserción

Comparación de list[j-1] > list[j] y cambio de lugar list[j-1], list[j] = list[j], list[j-1]

Selección

Comparación de list[index] > list[j] y cambio de lugar index = j y list[i], list[index] = list[index], list[i]

Shell

Comparacion de arr[j] > temp y cambio de lugar arr[j+h] = arr[j], arr[j+h]=temp

Mezcla

Dividir en dos el areglo y llamada de función recursivo de derecha y izquierda

Rápido

Partición de arreglo y llamada de función recursivo de quick_sort

9. ¿Cual será tiempo real para 50,000,000 100,000,000 500,000,000 1,000,000,000 y 5,000,000,000 de números a ordenar?

No logramos obtener tiempo para 10,000,000 por rendimiento de maquina, entonces no podemos proximar tiempo confiable para más números.

10. Preguntas

a) ¿Cuál de los 5 algoritmos es más fácil de implementar?

Inserción o Selección

b) ¿Cuál de los 2 algoritmos es el más difícil de implementar?

Búsqueda binaria

c) ¿Cuál algoritmo tiene menor complejidad temporal?

En peor caso, ordenamiento mezcla porque siempre tiene complejidad de O(nlogn).

d) ¿Cuál algoritmo tiene mayor complejidad temporal?

Ordenamiento por selección

e) ¿Cuál algoritmo tiene menor complejidad espacial? ¿Por qué?

Selección, inserción y shell porque tienen O(1). Aunque utilizamos arreglo, el tamaño de arreglo está definido, no se aumenta en ciclo. Por lo tanto, son O(1).

f) ¿Cuál algoritmo tiene mayor complejidad espacial? ¿Por qué?

Ordenamiento por mezcla porque tiene O(n).

g) ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?

No estamos seguro, porque nuestro maquina no tenia suficiente rendimiento para correr 10,000,000 numeros (si corría pero no acababa aunque ejecutamos por 1 hora), entonces teniamos que probar con menor números.

h) ¿Sus resultados experimentales difieren mucho de los del resto de los equipos? ¿A qué se debe?

Si, por el tamaño de array/lista usados y las diferentes características del equipo de cómputo utilizado

i) ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?

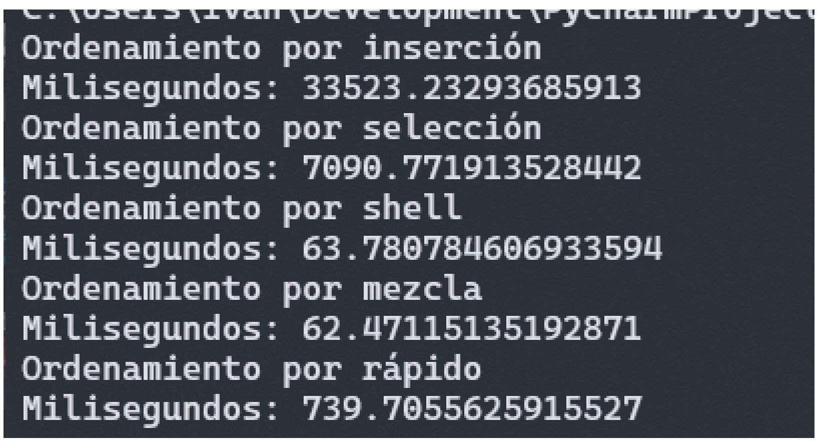
Sí, se usó un entorno virtual para usar las mismas bibliotecas

j) ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?

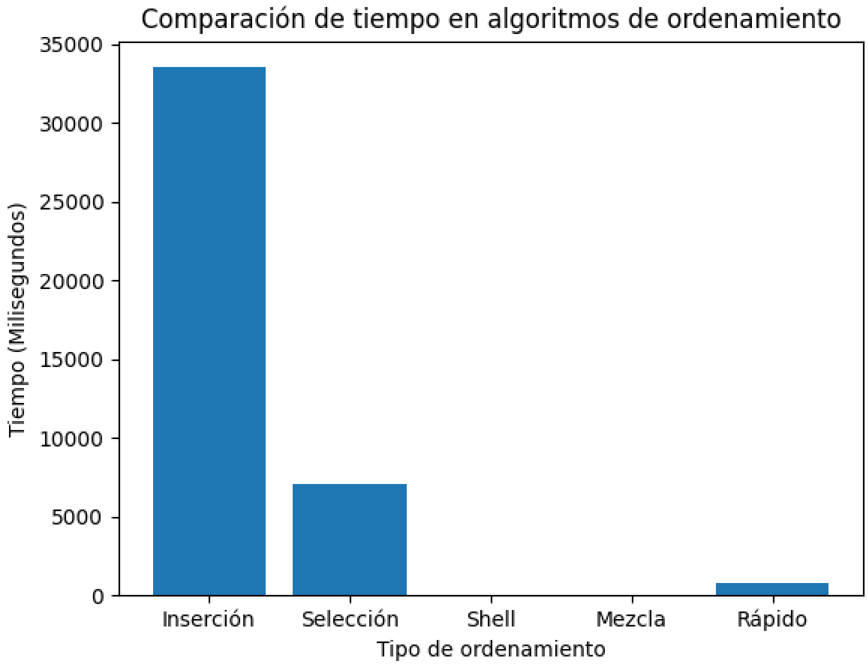
Tener una máquina virtual o que todo el equipo corra el código en un mismo lugar para no diferir mucho de los resultados.

Pruebas

Prueba 1

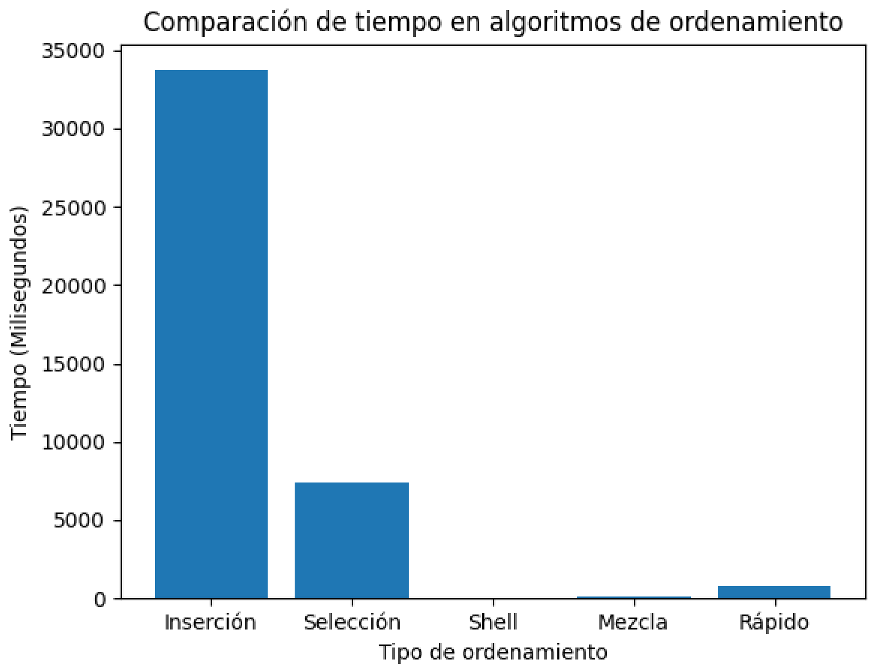


```
Ordenamiento por inserción
Milisegundos: 33523.23293685913
Ordenamiento por selección
Milisegundos: 7090.771913528442
Ordenamiento por shell
Milisegundos: 63.780784606933594
Ordenamiento por mezcla
Milisegundos: 62.47115135192871
Ordenamiento por rápido
Milisegundos: 739.7055625915527
```



Prueba 2

```
Ordenamiento por inserción
Milisegundos: 33707.10897445679
Ordenamiento por selección
Milisegundos: 7406.374931335449
Ordenamiento por shell
Milisegundos: 54.99625205993652
Ordenamiento por mezcla
Milisegundos: 79.01906967163086
Ordenamiento por rápido
Milisegundos: 773.8831043243408
```



Anexo

main.py

```
from typing import List
import matplotlib.pyplot as plt
from time import time
from read_file import parse_array
import sys

# Declare variables
file_path = 'test.txt'

# Change Recursion maximum
sys.setrecursionlimit(10000)

# Insertion Sort Algorithm
def insertion_sort(list: List[int]):
    for i in range(1, len(list)):
```

```python
        j = i
        while j > 0 and list[j - 1] > list[j]:
            list[j - 1], list[j] = list[j], list[j -
1]
            j -= 1

# Selection Sort Algorithm
def selection_sort(list: List[int]):
    # Linear search
    for i in range(len(list)):
        index = i
        # Search the minimum value in the subarray
        for j in range(i + 1, len(list)):
            if list[index] > list[j]:
                index = j
        list[i], list[index] = list[index], list[i]

def shell_sort(arr):
    N = len(arr)
    h = N // 2
    while h > 0:
        for i in range(h, N):
            temp = arr[i]
            j = i - h
            while j >= 0 and arr[j] > temp:
                arr[j + h] = arr[j]
                j -= h
            arr[j + h] = temp
        h //= 2

def merge_sort(arr):
    if len(arr) > 1:
```

```python
        # Finding the mid of the array
        mid = len(arr)//2
        # Dividing the array elements
        L = arr[:mid]
        # into 2 halves
        R = arr[mid:]
        # Sorting the first half
        merge_sort(L)
        # Sorting the second half
        merge_sort(R)
        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
        k += 1
        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

def partition(start, end, array):
    # Initializing pivot's index to start
```

```
    pivot_index = start
    pivot = array[pivot_index]
    # This loop runs till start pointer crosses
    # end pointer, and when it does we swap the
    # pivot with element on end pointer
    while start < end:
        # Increment the start pointer till it finds
an
        # element greater than  pivot
        while start < len(array) and array[start] <=
pivot:
            start += 1
        # Decrement the end pointer till it finds an
        # element less than pivot
        while array[end] > pivot:
            end -= 1
        # If start and end have not crossed each
other,
        # swap the numbers on start and end
        if(start < end):
            array[start], array[end] = array[end],
array[start]

    # Swap pivot element with element on end
pointer.
    # This puts pivot on its correct sorted place.
    array[end], array[pivot_index] =
array[pivot_index], array[end]

    # Returning end pointer to divide the array into
2
    return end
```

```python
def quick_sort(start, end, array):
    if (start < end):
        # p is partitioning index, array[p]
        # is at right place
        p = partition(start, end, array)
        # Sort elements before partition
        # and after partition
        quick_sort(start, p - 1, array)
        quick_sort(p + 1, end, array)

# Time Flag
cpu_start_time = 0

# Run Selection and Shell sort
list = parse_array(file_path)
if list:
    # Insertion
    print("Ordenamiento por inserción")
    cpu_start_time = time()
    insertion_sort(list)
    cpu_insertion_sort = (time() - cpu_start_time) *
1000
    print(f'Milisegundos: {cpu_insertion_sort}')
    # Selection
    print("Ordenamiento por selección")
    cpu_start_time = time()
    selection_sort(list)
    cpu_selection_sort = (time() - cpu_start_time) *
1000
    print(f'Milisegundos: {cpu_selection_sort}')
    # Shell
```

```
    print("Ordenamiento por shell")
    cpu_start_time = time()
    shell_sort(list)
    cpu_shell_sort = (time() - cpu_start_time) *
1000
    print(f'Milisegundos: {cpu_shell_sort}')
    # Merge
    print("Ordenamiento por mezcla")
    cpu_start_time = time()
    merge_sort(list)
    cpu_merge_sort = (time() - cpu_start_time) *
1000
    print(f'Milisegundos: {cpu_merge_sort}')
    # Quick
    print("Ordenamiento por rápido")
    cpu_start_time = time()
    quick_sort(0, len(list) - 1, list)
    cpu_quick_sort = (time() - cpu_start_time) *
1000
    print(f'Milisegundos: {cpu_quick_sort}')
    plt.bar(["Inserción","Selección", "Shell",
"Mezcla", "Rápido"],
[cpu_insertion_sort,cpu_selection_sort,cpu_shell_sor
t,cpu_merge_sort,cpu_quick_sort])
    plt.title("Comparación de tiempo en algoritmos
de ordenamiento")
    plt.xlabel("Tipo de ordenamiento")
    plt.ylabel("Tiempo (Milisegundos)")
    plt.show()
```

read_file.py

```python
from typing import List

def parse_array(path: str):
    output: List[int] = []
    file = open(path, 'r')
    content = file.read()
    for item in content.split(','):
        try:
            value = int(item)
            output.append(value)
        except ValueError:
            print('Archivo con contenido inválido')
            return None
    return output
```

Instrucciones de compilación

Se debe activar el entorno virtual, para ello se debe escribir lo siguiente:

```
venv\Scripts\activate.bat
```

Se debe tener un archivo .txt que contenga los valores del arreglo. Estos valores deben ir separados por una coma.

La ruta de dicho arhivo se debe escribir en la variable "file_path" del programa main.py. También se escribir el número que se desea buscar en la variable "x".

Finalmente se ejecuta el archivo.

```
python main.py
```

Bibliografía

- Navarro, A. (2020, July 14). Ordenamiento por inserción – Algoritmos de ordenamiento. Junco TIC. Retrieved March 18, 2022, from https://juncotic.com/ordenamiento-por-insercion-algoritmos-de-ordenamiento/

- Ordenamiento por selección. (n.d.). ALGORITHMIQUE/PROGRAMMATION. Retrieved March 18, 2022, from http://lwh.free.fr/pages/algo/tri/tri_selection_es.html
- Ordenamiento Shell. (2021, March 11). Delft Stack. https://www.delftstack.com/es/tutorial/algorithm/shell-sort/
- Ordenamiento por mezcla. (2021, February 25). Delft Stack. Retrieved March 18, 2022, from https://www.delftstack.com/es/tutorial/algorithm/merge-sort/
- Ordenamiento rápido. (2021, February 25). Delft Stack. Retrieved March 18, 2022, from https://www.delftstack.com/es/tutorial/algorithm/quick-sort/