UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Algoritmos de ordenamiento en lenguajes de programación en c++ y python

Informe laboratorio N° 1

Estudiante:
Doris Apaza Anahua

Profesor:
Honorio Apaza Alanoca

24 de mayo de 2023

Índice

1. Introducción **2**
1.1. Motivación y Contexto 2
1.2. Objetivo general 2
1.3. Objetivos específicos 2
1.4. Justificación 2

2. Marco teórico **3**

3. Materiales usados **5**

4. Ejercicios **6**

5. Resultados **11**

6. Conclusiones **12**

Referencias **13**

1. Introducción

1.1. Motivación y Contexto

Un algoritmo de Ordenamiento es un algoritmo que pone en una lista en una secuencia dada una relaci on de orden,por lo cual es importante en cuanto a la ciencia de la computaci on y tiene varias aplicaciones pr acticas.La eficiencia,busqueda,recuperaci on,presentaci on un buen conocimiento y dominio de los algoritmos de ordenamiento puede mejorar significa- tivamente el rendimiento y la funcionalidad de programas y sistemas.

1.2. Objetivo general

Como objetivo general tenemos que los algoritmos de ordenamiento nos permite ordenar la información de manera eficiente basandoce en un criterio de ordenamiento.

1.3. Objetivos específicos

- preparación de Datos,generar varios archivos txt.
- Implementar los siete algoritmos de ordenamiento como:Buble sort,Couting sort,Heap sort,Insertion sort,Merge sort,Quick sort,Selection sort.
- Comparación de tiempo de procesamiento de cada algoritmo por cada lenguaje de programacion,generar grafica.

1.4. Justificación

Los algoritmos de ordenamiento nos permite, como su nombre lo dice, ordenar información de una manera especial basándonos en un criterio de ordenamiento.En la computación el ordenamiento de datos cumple un rol muy importante, ya sea como un fin en sí o como parte de otros procedimientos m as complejos. Se han desarrollado muchas técnicas en este ambito, cada una con caracter ısticas específicas, y con ventajas y desventajas sobre las demás.
Para que os hag ais una idea de la dificultad del problema, propongo el siguiente mini juego. Se trata de unos barriles ordenar (entre 3 y 10) con el fin de aumentar de peso. El peso de cada barril fue asignado al azar. Utilice la opci on .arrastrar y soltar"para mover los barriles.Tienes una escala no calibrada que le permite comparar el peso de barriles y estantes que pueden servir para el almacenamiento intermedio. Estos son exactamente los mismos elementos que los que est an disponibles a un ordenador: una funci on de comparación y areas de almacenamiento. El objetivo es, obviamente, de ordenar los barriles con los menos comparaciones e intercambios posibles.

2. Marco teórico

- Ordenamiento de Burbuja (Bubble Sort):

  Descripción: Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Complejidad Temporal: $O(n^2) en el peor caso$.

  Ventajas: Simple de implementar y entender.

  Desventajas: Ineficiente para conjuntos de datos grandes.

- Ordenamiento por Inserción (Insertion Sort):

  Descripción: Construye una lista ordenada insertando elementos no ordenados uno a uno en su posición correcta.

  Complejidad Temporal: $O(n^2) en el peor caso$.

  Ventajas: Eficiente para conjuntos de datos pequeños o casi ordenados.

  Desventajas: No es muy eficiente para conjuntos de datos grandes.

- Ordenamiento por Selección (Selection Sort):

  Descripción: Encuentra el elemento mínimo y lo intercambia con el primer elemento. Luego encuentra el siguiente mínimo en la porción restante y lo intercambia con el segundo, y así sucesivamente. Complejidad Temporal: $O(n^2) en todos los casos$.

  Ventajas: Simple de implementar y no requiere memoria adicional.

  Desventajas: No es eficiente para conjuntos de datos grandes debido a su complejidad cuadrática.

- Ordenamiento por Fusión (Merge Sort):

  Descripción: Divide recursivamente la lista en sublistas más pequeñas, las ordena y luego las fusiona en una lista ordenada. Complejidad Temporal: O(n log n) en todos los casos.

  Ventajas: Eficiente para conjuntos de datos grandes, estable y tiene una complejidad predecible.

  Desventajas: Requiere memoria adicional para realizar la fusión.

- Ordenamiento Rápido (Quick Sort):

  Descripción: Elige un elemento "pivotez divide la lista en dos partes, una con elementos menores al pivote y otra con elementos mayores. Luego, se aplica el mismo proceso a las dos partes. Complejidad Temporal: O(n log n) en promedio, $O(n^2) en el peor caso$.

  Ventajas: Eficiente en la mayoría de los casos, especialmente para conjuntos de datos grandes.

  Desventajas: Puede ser ineficiente en el peor caso y no es estable.

- Ordenamiento de Montículos (Heap Sort):

  Descripción: Construye un montículo a partir de la lista y luego extrae iterativamente el elemento máximo (raíz) y lo coloca al final, reajustando el montículo. Complejidad Temporal: O(n log n) en todos los casos.

  Ventajas: Eficiente y garantiza una complejidad de tiempo constante.

  Desventajas: No es estable y requiere memoria adicional para almacenar el montículo.

- Ordenamiento por Conteo (Counting Sort):

  Descripción: Cuenta el número de ocurrencias de cada elemento y luego reconstruye la lista ordenada a partir de estas frecuencias. Complejidad Temporal: O(n + k), donde n es el tamaño de la lista y k es el rango de valores posibles.

  Ventajas: Muy eficiente para conjuntos de datos.

3. Materiales usados

- Latex
- Python
- GitHub
- C++

4. Ejercicios

BUBBLE SORT

```python
def bubble_sort(file_name):
    with open(file_name, 'r') as archivo:
        numeros = [int(linea.strip()) for linea in archivo]

    n = len(numeros)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if numeros[j] > numeros[j + 1]:
                numeros[j], numeros[j + 1] = numeros[j + 1], numeros[j]

    with open(file_name, 'w') as archivo:
        for numero in numeros:
            archivo.write(str(numero) + '\n')

    print("N meros ordenados con  xito  utilizando Bubble Sort.")

bubble_sort('../numbers/100.txt')
```

COUNTING SORT

```python
def counting_sort(file_name):

    with open(file_name, 'r') as archivo:
        numeros = [int(linea.strip()) for linea in archivo]

    valor_maximo = max(numeros)

    conteo = [0] * (valor_maximo + 1)

    for numero in numeros:
        conteo[numero] += 1

    numeros_ordenados = []
    for i in range(len(conteo)):
        numeros_ordenados.extend([i] * conteo[i])

    with open(file_name, 'w') as archivo:
        for numero in numeros_ordenados:
            archivo.write(str(numero) + '\n')

    print("N meros ordenados con  xito  utilizando Counting Sort.")

counting_sort('../numbers/100.txt')
```

HEAP SORT

```python
def heap_sort(file_name):
    def heapify(arr, n, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2

        if l < n and arr[i] < arr[l]:
            largest = l

        if r < n and arr[largest] < arr[r]:
            largest = r

        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

    def build_heap(arr):
        n = len(arr)
        for i in range(n // 2 - 1, -1, -1):
            heapify(arr, n, i)

    with open(file_name, 'r') as file:
        numbers = [int(line.strip()) for line in file]

    build_heap(numbers)
    sorted_numbers = []
    while numbers:
        numbers[0], numbers[-1] = numbers[-1], numbers[0]
        sorted_numbers.insert(0, numbers.pop())
        heapify(numbers, len(numbers), 0)

    with open(file_name, 'w') as file:
        for number in sorted_numbers:
            file.write(str(number) + '\n')

    print("N meros ordenados con  xito  utilizando Heap Sort.")

heap_sort('../numbers/100.txt')
```

INSERTION SORT

```python
def insertion_sort(file_name):
    with open(file_name, 'r') as file:
        numbers = [int(line.strip()) for line in file]

    for i in range(1, len(numbers)):
        key = numbers[i]
        j = i - 1
        while j >= 0 and key < numbers[j]:
            numbers[j + 1] = numbers[j]
            j -= 1
```

```
        numbers[j + 1] = key

    with open(file_name, 'w') as file:
        for number in numbers:
            file.write(str(number) + '\n')

    print("N meros ordenados con  xito  utilizando Insertion
    Sort.")

insertion_sort('../numbers/100.txt')
```

MERGE SORT

```
def merge_sort(file_name):
    def merge(arr, left, mid, right):
        n1 = mid - left + 1
        n2 = right - mid

        L = [arr[left + i] for i in range(n1)]
        R = [arr[mid + 1 + i] for i in range(n2)]

        i = j = 0
        k = left

        while i < n1 and j < n2:
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < n1:
            arr[k] = L[i]
            i += 1
            k += 1

        while j < n2:
            arr[k] = R[j]
            j += 1
            k += 1

    def merge_sort_helper(arr, left, right):
        if left < right:
            mid = (left + right) // 2
            merge_sort_helper(arr, left, mid)
            merge_sort_helper(arr, mid + 1, right)
            merge(arr, left, mid, right)

    with open(file_name, 'r') as file:
```

```
        numbers = [int(line.strip()) for line in file]

    merge_sort_helper(numbers, 0, len(numbers) - 1)

    with open(file_name, 'w') as file:
        for number in numbers:
            file.write(str(number) + '\n')

    print("N meros ordenados con  xito  utilizando Merge Sort.")

merge_sort('../numbers/100.txt')
```

QUICK SORT

```
def quick_sort(file_name):
    def partition(arr, low, high):
        i = low - 1
        pivot = arr[high]

        for j in range(low, high):
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]

        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

    def quick_sort_helper(arr, low, high):
        if low < high:
            pi = partition(arr, low, high)
            quick_sort_helper(arr, low, pi - 1)
            quick_sort_helper(arr, pi + 1, high)

    with open(file_name, 'r') as file:
        numbers = [int(line.strip()) for line in file]

    quick_sort_helper(numbers, 0, len(numbers) - 1)

    with open(file_name, 'w') as file:
        for number in numbers:
            file.write(str(number) + '\n')

    print("N meros ordenados con  xito  utilizando Quick Sort.")

quick_sort('../numbers/100.txt')
```

SELECTION SORT

```
def selection_sort(file_name):
    with open(file_name, 'r') as file:
```

```
        numbers = [int(line.strip()) for line in file]

    for i in range(len(numbers)):
        min_index = i
        for j in range(i + 1, len(numbers)):
            if numbers[j] < numbers[min_index]:
                min_index = j

        numbers[i], numbers[min_index] = numbers[min_index], numbers[i]

    with open(file_name, 'w') as file:
        for number in numbers:
            file.write(str(number) + '\n')

    print("N meros ordenados con  xito  utilizando Selection Sort.")

selection_sort('../numbers/100.txt')
```

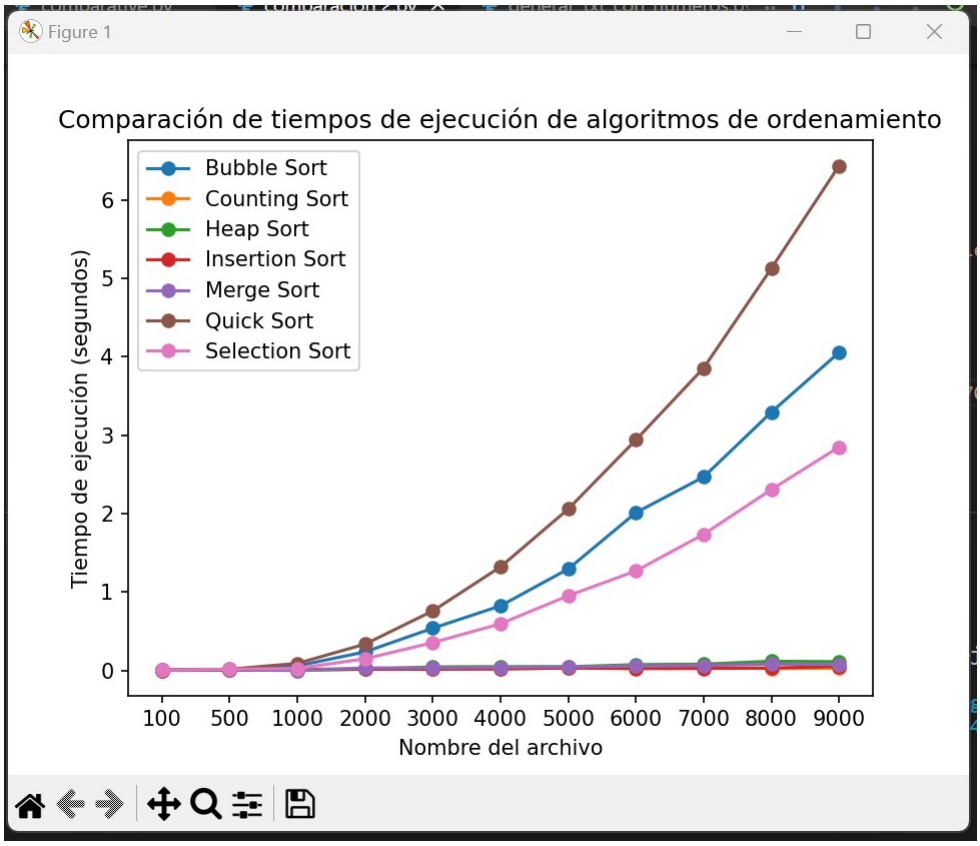
PARA C++

```
#include <iostream>

int main() {
  std::cout << "Hello, world!\n";
  return 0;
}
```

5. Resultados

Resultado de la comparación de Tiempos de Ejecución de Algoritmos de ordenamiento



ADJUNTO MI ENLACE DE GITHUB

https://github.com/DorisApaz/Proyecto_Ordenamiento.git

6. Conclusiones

En resumen, elegir el algoritmo de ordenamiento adecuado depende de varios factores, como la eficiencia requerida, el tamaño y las características de los datos, la estabilidad deseada y la facilidad de implementación. Comprender las ventajas y desventajas de diferentes algoritmos de ordenamiento ayuda a seleccionar el enfoque más adecuado para cada situación.

Referencias