Trabajo integrador

búsquedas y Ordenamiento

Alumnos:

Villarruel Joaquin - joaquinvillarrul04@gmail.com

Silveira Santiago - theisas@hotmail.com

Materia: Programación

Profesor: AUS Bruselario, Sebastián

Fecha de entrega: 9 de junio de 2025

Índice

Introducción.....3
Marco Teórico..... 4
¿Qué es un algoritmo?..... 4
Algoritmo de búsqueda..... 4
Comparación entre algoritmos de búsqueda..... 5
Cuando usar cada algoritmo de búsqueda..... 6
Algoritmos de Ordenamiento..... 6
Comparación algoritmos de ordenamiento..... 7
Caso Práctico.....7
Bubble Sort..... 8
Selection Sort..... 10
Insertion Sort..... 12
Quick Sort..... 14
Búsqueda Lineal..... 17
Búsqueda Binaria..... 20
Conclusión Algoritmos de Búsquedas..... 21
metodología utilizada.....22
Resultados obtenidos..... 23
Conclusiones..... 24
Anexos..... 25

Introducción

Para desarrollar software eficiente, no solo se deben tener conocimientos sobre sintaxis y estructuras, también se tienen que estudiar algoritmos que permitan organizar y acceder a la información de manera eficiente. En este informe se tratarán los algoritmos de **Búsqueda y Ordenamiento**, estos son muy utilizados en bases de datos, análisis de datos o incluso en inteligencia artificial.

En el transcurso de este trabajo se utilizará el lenguaje python para aplicar estos algoritmos, será analizado su funcionamiento y rendimiento para determinar en qué contextos podrían ser útiles.

Los algoritmos a tratar serán, por el lado del ordenamiento, **Bubble Sort, Insertion Sort, Selection Sort** y **Quick Sort**, y por el lado de búsqueda serán **Busqueda Lineal** y **Búsqueda Binaria.** Serán implementados con listas de tamaños diversos, con el fin de comprender su comportamiento y que se permita su comparación en términos de tiempo empleado.

Marco Teórico

¿Qué es un algoritmo?

Es una secuencia de pasos ordenada y finita, resuelven problemas o realizan tareas. Para la programación los algoritmos son fundamentales, arman la lógica que le da instrucciones a la computadora para hacer una tarea.

Cuando se crea un programa se utilizan uno o varios algoritmos. Es importante que estos cumplan su objetivo de forma eficiente, es decir, rápido y con la menor cantidad de recursos posibles.

Cabe aclarar que los algoritmos no son únicamente de la programación, estos están en la vida cotidiana, un ejemplo de esto es cuando seguimos una receta de cocina o instrucciones para armar algo.

En el caso de la programación se clasifican en categorías según su función, pueden ser de búsqueda, ordenamiento, recorrido, optimización, etc. Como se mencionó anteriormente este trabajo se centrará en los algoritmos de búsqueda y ordenamiento.

Algoritmo de búsqueda

Permiten localizar un valor específico en un conjunto de datos. Son utilizados en bases de datos, sistemas de archivos, inteligencia artificial y análisis de información. Realizar la elección correcta del algoritmo es sumamente importante, ya que este puede afectar el rendimiento del programa, especialmente cuando se trabaja con altas cantidades de información.

Estos algoritmos se pueden dividir en dos grupos, **búsqueda sin orden previo** y **búsqueda sobre datos ordenados.** A continuación los algoritmos que se abordarán en este trabajo.

- **Búsqueda lineal:** También conocida como búsqueda secuencial, es el más sencillo de todos, recorre la lista desde su inicio hasta el final. Compara cada elemento con el valor buscado. Cuando encuentra el elemento devuelve su posición, en caso de que el elemento no exista recorre toda la lista sin éxito.

  Para utilizar este algoritmo no se requiere que los datos estén ordenados, útil para listas no clasificadas o para realizar pocas búsquedas. El problema de este algoritmo surge cuando aumenta el tamaño de la lista, debido a que si el elemento a buscar se encuentra al final de la lista deberá recorrer toda esta, de esta forma demorando mucho tiempo.

Las ventajas de este tipo de búsqueda serían su fácil implementación y que se puede aplicar a cualquier tipo de lista. Por otro lado una gran desventaja es su bajo rendimiento cuando existen grandes listas

- **Búsqueda binaria:** Es un algoritmo eficiente, pero necesita que la lista esté ordenada. Su funcionamiento se basa en comparar el valor buscado con el elemento del medio. Pueden darse los siguientes casos:
    1. Si coincide, el elemento fue encontrado.
    2. Si el valor buscado es menor, se repite el proceso en la mitad izquierda.
    3. Si el valor es mayor, se repite el proceso en la mitad derecha.

  El proceso se repite de forma recursiva o iterativa hasta encontrar el elemento o hasta que se agote el subconjunto. Una gran ventaja que tiene es que reduce el espacio de búsqueda a la mitad por cada iteración, esto permite que sea eficiente con grandes volúmenes de datos. La desventaja de este tipo de búsqueda es que requiere orden en los datos.

  Como se mencionó anteriormente se puede implementar de forma recursiva o iterativa, en este trabajo se usará su forma iterativa debido a su simplicidad y menor consumo de memoria.

  Por ejemplo, si se requiere encontrar el número 8 en la siguiente lista ordenada [2,4,6,8,10,12], el algoritmo lo primero que realizará será comparar con el elemento central, en este caso 6. Como 8 es mayor, descarta la mitad izquierda y conserva la otra mitad [8,10,12]. Luego compara con el nuevo valor central el cual será 10, como el número que buscamos es menor conserva la parte izquierda, encontrando así el 8 y retornando la posición correspondiente.

  En caso de que la lista contenga 1.000.000 de elementos, en la peor de las situaciones se necesitan 20 comparaciones, mientras que si la búsqueda fuera lineal se podrían necesitar hasta un millón.

Comparación entre algoritmos de búsqueda

Característica	**Búsqueda Lineal**	**Búsqueda Binaria**
Requiere lista ordenada	No	Sí
Complejidad temporal	O(n)	O(log n)
Implementación	Simple	Levemente compleja
Rendimiento listas pequeñas	Bueno	Bueno

Rendimiento listas grandes	Bajo	Excelente

Cuando usar cada algoritmo de búsqueda

La **Búsqueda Lineal** se recomienda cuando:

- La lista es pequeña.
- Los datos cambian constantemente y no se justifica ordenarlo cada vez.
- Se hace una sola búsqueda y no justifica ordenar.

La **Búsqueda Binaria** es útil cuando:

- La lista ya está ordenada o se pueden ordenar previamente.
- Se realizan múltiples búsquedas sobre la misma estructura de datos.
- Se busca mayor eficiencia en grandes cantidades de datos.

En resumen, para seleccionar el algoritmo que más se adapte al caso de uso, se debe tener en cuenta el tamaño y orden de la lista y además la cantidad de búsquedas que se realizarán.

Algoritmos de Ordenamiento

Estos algoritmos nos permiten organizar los datos con un criterio determinado, este puede ser de mayor a menor, alfabéticamente o por fechas. Tener la posibilidad de ordenar la información permite aplicar algoritmos más eficientes, como es el caso de la búsqueda binaria.

Existen varios tipos de algoritmos de ordenamiento, en este trabajo trataremos los siguientes: **Bubble Sort, Selection Sort, Insertion Sort** y **Quick Sort.**

- **Bubble Sort:** También conocido como **ordenamiento por burbuja**, compara los elementos adyacentes de la lista y los intercambia si están en orden incorrecto. Este proceso se repite hasta que la lista quede completamente ordenada.

  Es un algoritmo muy sencillo de implementar y a su vez uno de los menos eficientes.

- **Selection Sort:** Busca el elemento más pequeño de la lista y lo coloca en primera posición. Luego vuelve a repetir este proceso hasta lograr que la lista esté ordenada.

  Este algoritmo es fácil de programar, pero aunque realiza menos intercambios que el Bubble Sort sigue siendo ineficiente con las listas grandes.

- **Insertion Sort:** Construye una lista ordenada de a un elemento por vez, inserta cada elemento nuevo en la posición correcta respecto a los anteriores.

Tiene un rendimiento aceptable en listas pequeñas o listas semi ordenadas.

- **Quick Sort:** Es un algoritmo que se basa en **Divide y Vencerás.** Selecciona un elemento al cual se le llama pivote y divide la lista en dos, una con los elementos menores al pivote y otra con los mayores. Luego recursivamente aplica el mismo proceso a las sublistas.

  Es uno de los algoritmos más eficientes para grandes volúmenes de datos. Pero su rendimiento depende del pivote elegido.

Comparación algoritmos de ordenamiento

Algoritmo	Complejidad promedio	Peor caso	Dificultad de implementación	Eficiencias listas grandes
Bubble Sort	$O(n^2)$	$O(n^2)$	Muy fácil	Muy baja
Selection Sort	$O(n^2)$	$O(n^2)$	Fácil	Baja
Insertion Sort	$O(n^2)$	$O(n^2)$	Fácil	Aceptable
Quick Sort	$O(n \log n)$	$O(n^2)$	Intermedia	Muy buena

En resumen, la elección del algoritmo de ordenamiento depende del tamaño de la lista, del orden inicial y de la eficiencia que se quiera lograr.

Caso Práctico

Se realizó la implementación y ejecución de algoritmos de búsqueda y ordenamiento en python, esto con la finalidad de analizar el rendimiento de los mismos ante listas aleatorias de distintos tamaños. El análisis de eficiencia se realizó midiendo el tiempo de ejecución de cada algoritmo.

Para realizar esta práctica se utilizaron los módulos **random,** para generar listas aleatorias y time, para medir el tiempo empleado por cada algoritmo. Cada uno de estos fue analizado con tres tamaños de listas distintas: Pequeña, mediana y grande.

Bubble Sort

Como se puede ver a continuación el algoritmo fue implementado utilizando distintos tamaños de listas, así pudiendo observar su eficiencia ante diversas situaciones.

```python
import timeit
import random
def ordenamientoBurbuja(lista): # Definimos la funcion que
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(largo):  # Recorremos la lista
        cambio = False  # Creamos una bandera para saber si
        for j in range(0, largo - i - 1):   # Recorremos ot
            if lista[j] > lista[j + 1]: # Preguntamos si el
                auxiliar = lista[j]
                lista[j] = lista[j + 1] # Si el numero es m
                lista[j + 1] = auxiliar
                cambio = True
        if not cambio:  # Si despues de recorrer la lista n
            break

lista = random.sample(range(1, 10000), 200) #Creamos una li
inicio = timeit.default_timer() #Tomamos el tiempo inicial
ordenamientoBurbuja(lista)
fin = timeit.default_timer() #Tomamos el tiempo final
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[165, 171, 277, 321, 359, 443, 475, 648, 674, 711, 781, 798, 8
Tiempo de ejecución: 0.001123 Seg
```

```python
import timeit
import random
def ordenamientoBurbuja(lista): # Definimos la funcion que
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(largo):  # Recorremos la lista
        cambio = False  # Creamos una bandera para saber si
        for j in range(0, largo - i - 1):   # Recorremos ot
            if lista[j] > lista[j + 1]: # Preguntamos si el
                auxiliar = lista[j]
                lista[j] = lista[j + 1] # Si el numero es m
                lista[j + 1] = auxiliar
                cambio = True
        if not cambio:  # Si despues de recorrer la lista n
            break

lista = random.sample(range(1, 10000), 1000) #Creamos una l
inicio = timeit.default_timer() #Tomamos el tiempo inicial
ordenamientoBurbuja(lista)
fin = timeit.default_timer() #Tomamos el tiempo final
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[8, 19, 22, 30, 36, 38, 41, 46, 48, 69, 82, 84, 93, 98, 103, 1
Tiempo de ejecución: 0.035283 Seg
```

```python
import timeit
import random
def ordenamientoBurbuja(lista): # Definimos la funcion que
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(largo):  # Recorremos la lista
        cambio = False  # Creamos una bandera para saber si
        for j in range(0, largo - i - 1):   # Recorremos ot
            if lista[j] > lista[j + 1]: # Preguntamos si el
                auxiliar = lista[j]
                lista[j] = lista[j + 1] # Si el numero es m
                lista[j + 1] = auxiliar
                cambio = True
        if not cambio:  # Si despues de recorrer la lista n
            break

lista = random.sample(range(1, 10000), 9000) #Creamos una l
inicio = timeit.default_timer() #Tomamos el tiempo inicial
ordenamientoBurbuja(lista)
fin = timeit.default_timer() #Tomamos el tiempo final
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 3.0s

```
[1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
Tiempo de ejecución: 3.08943 Seg
```

Resultados:

Tamaño de la lista	Tiempo de ejecución (Segundos)
200 elementos	0,001123
1000 elementos	0,035283
9000 elementos	3,08943

Mostró un crecimiento muy pronunciado en cuestión de tiempo de ejecución cuando se aumenta el tamaño de la lista. Para 200 elementos se tuvo un rendimiento aceptable, con 1000 se notó una mayor demora, y finalmente con 9000 el tiempo fue muy elevado.

Esto demuestra que con listas grandes es ineficiente.

Selection Sort

Se implementó y evaluó bajo las mismas condiciones que el anterior algoritmo.

```
import timeit
import random

def ordenamientoSeleccion(lista): # Definimos la funcion qu
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(largo - 1):  # Recorremos la lista hasta
        minimo = i  # Cada vuelta asignamos el primer indic
        for j in range(i + 1, largo):   # Recorremos nuevam
            if lista[j] < lista[minimo]:    # Comparamos el
                minimo = j  # Si el nuevo elemento es menor
        auxiliar = lista[i]
        lista[i] = lista[minimo]    # Usamos un auxiliar pa
        lista[minimo] = auxiliar

lista = random.sample(range(1, 10000), 200)
inicio = timeit.default_timer()
ordenamientoSeleccion(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[17, 24, 78, 158, 209, 325, 468, 492, 503, 539, 639, 721, 727,
Tiempo de ejecución: 0.000546 Seg
```

```python
import timeit
import random

def ordenamientoSeleccion(lista): # Definimos la funcion qu
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(largo - 1):  # Recorremos la lista hasta
        minimo = i  # Cada vuelta asignamos el primer indic
        for j in range(i + 1, largo):   # Recorremos nuevam
            if lista[j] < lista[minimo]:    # Comparamos el
                minimo = j  # Si el nuevo elemento es menor
        auxiliar = lista[i]
        lista[i] = lista[minimo]    # Usamos un auxiliar pa
        lista[minimo] = auxiliar

lista = random.sample(range(1, 10000), 1000)
inicio = timeit.default_timer()
ordenamientoSeleccion(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[6, 20, 40, 46, 80, 84, 99, 100, 111, 116, 123, 124, 125, 173,
Tiempo de ejecución: 0.016821 Seg
```

```python
import timeit
import random

def ordenamientoSeleccion(lista): # Definimos la funcion qu
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(largo - 1):  # Recorremos la lista hasta
        minimo = i  # Cada vuelta asignamos el primer indic
        for j in range(i + 1, largo):   # Recorremos nuevam
            if lista[j] < lista[minimo]:    # Comparamos el
                minimo = j  # Si el nuevo elemento es menor
        auxiliar = lista[i]
        lista[i] = lista[minimo]    # Usamos un auxiliar pa
        lista[minimo] = auxiliar

lista = random.sample(range(1, 10000), 9000)
inicio = timeit.default_timer()
ordenamientoSeleccion(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 1.2s

```
[1, 2, 3, 6, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21,
Tiempo de ejecución: 1.321425 Seg
```

Resultados:

Tamaño de la lista	Tiempo de ejecución (Segundos)
200 elementos	0,000546
1000 elementos	0,016821
9000 elementos	1,321425

Se presentó una mejora notable respecto al algoritmo anterior en volúmenes de datos grandes. Esto se debe a que realiza menos intercambios, de todas formas con la lista de 9000 elementos el tiempo de ejecución resultó elevado.

Insertion Sort

Se implementó y evaluó bajo las mismas condiciones que el anterior algoritmo.

```python
import timeit
import random

def ordenamientoInsercion(lista): # Definimos la funcion qu
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(1, largo):   # Recorremos la lista desde
        insertar = i    # Indicamos el indice para la inser
        valorActual = lista[i]  # Valor actual del recorrid
        for j in range(i - 1, -1, -1):  # Recorremos en rev
            if lista[j] > valorActual:  # Preguntamos si el
                lista[j + 1] = lista[j] # Si es verdad se i
                insertar = j    # Y tambien cambiamos el in
            else:
                break   # Si no es verdad salimos del recor
        lista[insertar] = valorActual   # El valor se inser

lista = random.sample(range(1, 10000), 200)
inicio = timeit.default_timer()
ordenamientoInsercion(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[54, 78, 118, 180, 221, 290, 362, 402, 437, 480, 491, 535, 588
Tiempo de ejecución: 0.000519 Seg
```

```python
import timeit
import random

def ordenamientoInsercion(lista): # Definimos la funcion qu
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(1, largo):   # Recorremos la lista desde
        insertar = i    # Indicamos el indice para la inser
        valorActual = lista[i]  # Valor actual del recorrid
        for j in range(i - 1, -1, -1):  # Recorremos en rev
            if lista[j] > valorActual:  # Preguntamos si el
                lista[j + 1] = lista[j] # Si es verdad se i
                insertar = j    # Y tambien cambiamos el in
            else:
                break   # Si no es verdad salimos del recor
        lista[insertar] = valorActual   # El valor se inser

lista = random.sample(range(1, 10000), 1000)
inicio = timeit.default_timer()
ordenamientoInsercion(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[13, 31, 38, 41, 47, 68, 75, 76, 79, 93, 102, 104, 107, 111, 1
Tiempo de ejecución: 0.012881 Seg
```

```python
import timeit
import random

def ordenamientoInsercion(lista): # Definimos la funcion qu
    largo = len(lista)  # Conseguimos el tamaño de la lista
    for i in range(1, largo):   # Recorremos la lista desde
        insertar = i    # Indicamos el indice para la inser
        valorActual = lista[i]  # Valor actual del recorrid
        for j in range(i - 1, -1, -1):  # Recorremos en rev
            if lista[j] > valorActual:  # Preguntamos si el
                lista[j + 1] = lista[j] # Si es verdad se i
                insertar = j    # Y tambien cambiamos el in
            else:
                break   # Si no es verdad salimos del recor
        lista[insertar] = valorActual   # El valor se inser

lista = random.sample(range(1, 10000), 9000)
inicio = timeit.default_timer()
ordenamientoInsercion(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 1.1s

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
Tiempo de ejecución: 1.154131 Seg
```

Resultado:

Tamaño de la lista	Tiempo de ejecución (Segundos)
200 elementos	0,000519
1000 elementos	0,012881
9000 elementos	1,154131

Mostró un rendimiento similar a Selection Sort, obtuvo resultados levemente mejores. En listas pequeñas es útil, pero en listas grandes si bien es más eficiente que Bubble Sort, sigue teniendo tiempos de ejecución elevados.
Cuando los datos están parcialmente ordenados este algoritmo eleva su eficiencia.

Quick Sort

Se implementó y evaluó bajo las mismas condiciones que el anterior algoritmo.

```
import timeit
import random

def particion(lista, bajo, alto):   # Funcion para particio
    pivote = lista[alto]    # Guardamos el ultimo elemento
    i = bajo - 1    # Asignamos al indice "i" el elemento m

    for j in range(bajo, alto): # Recorremos desde el valor
        if lista[j] <= pivote:  # Si el elemento actual es
            i += 1  # Incrementamos el indice de los menore
            auxiliar = lista[i]
            lista[i] = lista[j] # Cambiamos de lugar los el
            lista[j] = auxiliar

    auxiliar = lista[i + 1]
    lista[i + 1] = lista[alto]  # Cambiamos a la posicion c
    lista[alto] = auxiliar
    return i+1  # Devolvemos la posicion final del pivote

def ordenamientoRapido(lista, bajo = 0 , alto  = None): # F
    if alto == None:
        alto = len(lista) - 1   # Si no hay un "alto" asign

    if bajo < alto: # La funcion solo continua si el "bajo"
        indicePivote = particion(lista, bajo, alto) # Parti
        ordenamientoRapido(lista, bajo , indicePivote - 1)
        ordenamientoRapido(lista, indicePivote + 1, alto)

lista = random.sample(range(1, 10000), 200)
inicio = timeit.default_timer()
ordenamientoRapido(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[179, 347, 388, 415, 524, 534, 565, 614, 667, 736, 745, 820, 8
Tiempo de ejecución: 0.000173 Seg
```

```python
import timeit
import random

def particion(lista, bajo, alto):   # Funcion para particio
    pivote = lista[alto]    # Guardamos el ultimo elemento 
    i = bajo - 1    # Asignamos al indice "i" el elemento m

    for j in range(bajo, alto): # Recorremos desde el valor
        if lista[j] <= pivote:  # Si el elemento actual es 
            i += 1  # Incrementamos el indice de los menore
            auxiliar = lista[i]
            lista[i] = lista[j] # Cambiamos de lugar los el
            lista[j] = auxiliar

    auxiliar = lista[i + 1]
    lista[i + 1] = lista[alto]  # Cambiamos a la posicion c
    lista[alto] = auxiliar
    return i+1  # Devolvemos la posicion final del pivote

def ordenamientoRapido(lista, bajo = 0 , alto  = None): # F
    if alto == None:
        alto = len(lista) - 1   # Si no hay un "alto" asign

    if bajo < alto: # La funcion solo continua si el "bajo"
        indicePivote = particion(lista, bajo, alto) # Parti
        ordenamientoRapido(lista, bajo , indicePivote - 1)
        ordenamientoRapido(lista, indicePivote + 1, alto)

lista = random.sample(range(1, 10000), 1000)
inicio = timeit.default_timer()
ordenamientoRapido(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[8, 22, 36, 38, 56, 84, 127, 137, 150, 157, 165, 169, 199, 200
Tiempo de ejecución: 0.000921 Seg
```

```python
import timeit
import random

def particion(lista, bajo, alto):   # Funcion para particio
    pivote = lista[alto]    # Guardamos el ultimo elemento
    i = bajo - 1    # Asignamos al indice "i" el elemento m

    for j in range(bajo, alto): # Recorremos desde el valor
        if lista[j] <= pivote:  # Si el elemento actual es
            i += 1  # Incrementamos el indice de los menore
            auxiliar = lista[i]
            lista[i] = lista[j] # Cambiamos de lugar los el
            lista[j] = auxiliar

    auxiliar = lista[i + 1]
    lista[i + 1] = lista[alto]  # Cambiamos a la posicion c
    lista[alto] = auxiliar
    return i+1  # Devolvemos la posicion final del pivote

def ordenamientoRapido(lista, bajo = 0 , alto  = None): # F
    if alto == None:
        alto = len(lista) - 1   # Si no hay un "alto" asign

    if bajo < alto: # La funcion solo continua si el "bajo"
        indicePivote = particion(lista, bajo, alto) # Parti
        ordenamientoRapido(lista, bajo , indicePivote - 1)
        ordenamientoRapido(lista, indicePivote + 1, alto)

lista = random.sample(range(1, 10000), 9000)
inicio = timeit.default_timer()
ordenamientoRapido(lista)
fin = timeit.default_timer()
print(lista)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} Seg")
```

✓ 0.0s

```
[1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 2
Tiempo de ejecución: 0.01367 Seg
```

Resultados:

Tamaño de la lista	**Tiempo de ejecución (Segundos)**
200 elementos	0,000173
1000 elementos	0,000921
9000 elementos	0,01367

Quick Sort fue claramente el algoritmo más eficiente de todos los evaluados. Tuvo un gran desempeño incluso con la lista de 9000 elementos. Esto debido a su complejidad promedio de $O(n\ log\ n)$.

Conclusión eficiencia de algoritmos de ordenamiento

Quedó demostrado que **Quick Sort** es la mejor opción para grandes listas, superando ampliamente a los demás algoritmos. Para listas pequeñas otros algoritmos más sencillos pueden servir debido a que tienen un rendimiento similar y una mayor facilidad de implementación.

Búsqueda Lineal

Se realizaron 3 pruebas con diferentes tamaños de listas.

```python
import timeit
import random

def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Generamos una lista de tamaño 200
lista = random.sample(range(1, 10000000), 500)
objetivo = random.choice(lista)  # Elegimos un elemento que

inicio = timeit.default_timer()
resultado = busqueda_lineal(lista, objetivo)
fin = timeit.default_timer()

print("Elemento buscado:", objetivo)
print("Posición encontrada:", resultado)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} seg")
```

✓ 0.0s

```
Elemento buscado: 7770029
Posición encontrada: 315
Tiempo de ejecución: 5.6e-05 seg
```

```python
import timeit
import random

def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Generamos una lista de tamaño 200
lista = random.sample(range(1, 10000000), 5000)
objetivo = random.choice(lista)  # Elegimos un elemento que

inicio = timeit.default_timer()
resultado = busqueda_lineal(lista, objetivo)
fin = timeit.default_timer()

print("Elemento buscado:", objetivo)
print("Posición encontrada:", resultado)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} seg")
```

✓ 0.0s

```
Elemento buscado: 6147826
Posición encontrada: 3771
Tiempo de ejecución: 0.000179 seg
```

```python
import timeit
import random

def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Generamos una lista de tamaño 200
lista = random.sample(range(1, 10000000), 50000)
objetivo = random.choice(lista)  # Elegimos un elemento que

inicio = timeit.default_timer()
resultado = busqueda_lineal(lista, objetivo)
fin = timeit.default_timer()

print("Elemento buscado:", objetivo)
print("Posición encontrada:", resultado)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} seg")
```

✓ 0.0s

```
Elemento buscado: 5806029
Posición encontrada: 26250
Tiempo de ejecución: 0.001426 seg
```

Resultados:

Tamaño de la lista	**Tiempo de ejecución (Segundos)**
500 elementos	0,000056
5000 elementos	0,000179
50000 elementos	0,001426

El algoritmo mostró un comportamiento esperable, si bien tuvo un buen rendimiento con las listas pequeñas, se evidencia que a medida que las listas se vuelven más grandes el tiempo de ejecución asciende de forma proporcional. Esto permite darse cuenta que el algoritmo es útil cuando:

- La lista es pequeña.
- La lista no está ordenada.
- Se realizará una búsqueda aislada.

Con grandes cantidades de datos o con múltiples búsquedas, no es un algoritmo recomendado

Búsqueda Binaria

Se realizarán las mismas pruebas que el algoritmo anterior.

```python
import timeit
import random

def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

lista = sorted(random.sample(range(1, 10000000), 500))
objetivo = random.choice(lista)
inicio = timeit.default_timer()
resultado = busqueda_binaria(lista, objetivo)
fin = timeit.default_timer()

print("Elemento buscado:", objetivo)
print("Posición encontrada:", resultado)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} seg")
```

✓ 0.0s

```
Elemento buscado: 5180267
Posición encontrada: 258
Tiempo de ejecución: 2.9e-05 seg
```

```python
import timeit
import random

def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

lista = sorted(random.sample(range(1, 10000000), 5000))
objetivo = random.choice(lista)
inicio = timeit.default_timer()
resultado = busqueda_binaria(lista, objetivo)
fin = timeit.default_timer()

print("Elemento buscado:", objetivo)
print("Posición encontrada:", resultado)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} seg")
```

✓ 0.0s

```
Elemento buscado: 5335356
Posición encontrada: 2654
Tiempo de ejecución: 3.7e-05 seg
```

```python
import timeit
import random

def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

lista = sorted(random.sample(range(1, 10000000), 50000))
objetivo = random.choice(lista)
inicio = timeit.default_timer()
resultado = busqueda_binaria(lista, objetivo)
fin = timeit.default_timer()

print("Elemento buscado:", objetivo)
print("Posición encontrada:", resultado)
print(f"Tiempo de ejecución: {round(fin - inicio, 6)} seg")
```

✓ 0.0s

```
Elemento buscado: 4776542
Posición encontrada: 23965
Tiempo de ejecución: 3.9e-05 seg
```

Resultados

Tamaño de la lista	**Tiempo de ejecución (Segundos)**
500 elementos	0,000029
5000 elementos	0,000037
50000 elementos	0,000039

El algoritmo demostró una gran eficiencia en todos los tamaños de la lista. Mantuvo unos tiempos constantes en todos los casos. Es una gran opción para trabajar con grandes volúmenes de datos. También es importante tener en cuenta que se requiere que la lista esté previamente ordenada.

Conclusión Algoritmos de Búsquedas

Luego de evaluar los algoritmos de búsqueda, se pudieron ver diferencias en el rendimiento, especialmente al escalar el tamaño de las listas.

- **Búsqueda lineal:** Es simple, no requiere orden en la lista, pero su tiempo de ejecución crece de manera proporcional al tamaño de los datos.
- **Búsqueda binaria:** Requiere que la lista este ordenada, pero tiene una eficiencia altamente superior, manteniendo sus tiempos bajos incluso manejando grandes listas.

Si tenemos una lista ordenada o puede ordenarse y reutilizarse, la búsqueda binaria es la mejor opción, pero si se tienen listas pequeñas o listas en las que no se justifica ordenar la búsqueda lineal es una buena opción.

Característica	**Búsqueda Lineal**	**Búsqueda Binaria**
Requiere lista ordenada	No	Si
Complejidad	O(n)	O(log n)
Eficiencia grandes listas	Baja	Muy alta
Dificultad de implementación	Muy simple	Simple, requiere pasos previos.

metodología utilizada

Para iniciar con el trabajo se realizó una investigación acerca de los algoritmos de búsqueda y ordenamiento, se consultó el material proporcionado por la cátedra, la documentación oficial de Python y distintos artículos disponibles en la web. Con esta teoría se desarrolló el marco teórico.

Luego se comenzó con la etapa de diseño y prueba del código, se implementaron los algoritmos estudiados, se los sometió a pruebas utilizando listas de diferentes tamaños y midiendo sus tiempos de ejecución con ayuda de los módulos time y random.

Como herramientas utilizamos el lenguaje de programación Python y el IDE Visual Studio Code, se utilizó esta combinación de herramientas ya que fueron las trabajadas a lo largo del año.

Un integrante del equipo se ocupó del marco teórico, mientras que el otro se encargó de la práctica mediante la implementación y análisis del código. La investigación previa se realizó en conjunto.

Resultados obtenidos

- Los algoritmos de ordenamiento fueron correctamente implementados y funcionaron como era esperado.
- Se registraron tiempos de ejecución , permitiendo la comparación entre los diferentes algoritmos.
- Se comprendió la eficiencia de la búsqueda binaria por sobre la lineal.
- Se aprendió cómo el contexto y los volúmenes de datos afectan a la eficiencia de cada algoritmo.

Conclusiones

Este trabajo permitió elevar nuestros conocimientos acerca de los algoritmos de búsqueda y ordenamiento, desde la implementación hasta el análisis de los mismos. Se realizaron pruebas con listas de diferentes tamaños permitiendo así entender cómo se comportan los diferentes tipos de algoritmos a las situaciones que se pueden presentar durante el desarrollo de un programa. Llevarlos a la práctica también sirvió para confirmar las diferencias teóricas entre algoritmos.

Se llegó a la conclusión de que Quick Sort y la búsqueda binaria son los algoritmos más eficientes ante grandes volúmenes de datos, pero que los otros tipos son útiles en situaciones más sencillas debido a su facilidad de implementación.

Bibliografía

Cátedra de Programación - Materiales provistos durante la cursada

W3Schools – Python Sorting Algorithms: https://www.w3schools.com/

Python - Documentación Oficial: https://docs.python.org/3/library/

Anexos

Repositorio de GitHub:
https://github.com/JoaquinVillarruel/Trabajo_Integrador_Programaci-n.git

Video explicativo: