Trabajo Práctico - Algoritmos de Búsqueda y Ordenamiento

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación I

Trabajo integrador

“Algoritmos de Búsqueda y Ordenamiento en Python”

Profesores:

-Docente Titular: Quirós Nicolás

-Docente Tutor: Torres Matías Santiago

Alumnas:

-Stefani Gisele Eliana - gisele.stefani@tupad.utn.edu.ar

-Valcarcel Paola Mariela - paola.valcarcel@tupad.utn.edu.ar

Fecha de entrega: 09/06/2025

Índice

Algoritmos de Búsqueda y Ordenamiento en Python

Introducción

Los logaritmos de ordenamiento son herramientas fundamentales en programación ya que mejoran la eficiencia de los programas, optimizando tanto el tiempo de ejecución, cómo el uso de recursos, al mismo tiempo que facilita la organización de datos. Su aplicación es clave en una amplia variedad de contextos, como sistemas operativos y bases de datos, hasta inteligencia artificial y desarrollo web.

Su importancia radica en que muchos procesos informáticos dependen de operaciones de búsqueda y ordenamiento para funcionar correctamente. La elección adecuada del algoritmo puede marcar una gran diferencia en el rendimiento de un sistema. Además, estos algoritmos constituyen una base esencial para comprender los conceptos más avanzados en estructuras de datos y diseño de algoritmos.

El presente trabajo tiene como objetivo comprender el funcionamiento de los principales algoritmos de búsqueda y ordenamiento, analizar su eficiencia, y determinar en que situaciones resulta más conveniente utilizar cada uno. A su vez, se busca fortalecer el pensamiento lógico y algorítmico necesario para que los programadores puedan tomar decisiones más informadas al desarrollar soluciones computacionales. Finalmente se presentará un caso práctico implementando el lenguaje de programación Python.

Marco Teórico

-Definición de Algoritmo.

Un algoritmo es un conjunto de instrucciones bien definidas, ordenadas y finitas que permiten resolver un problema o realizar una tarea específica. En programación, los algoritmos son la base del desarrollo de soluciones lógicas y eficientes.

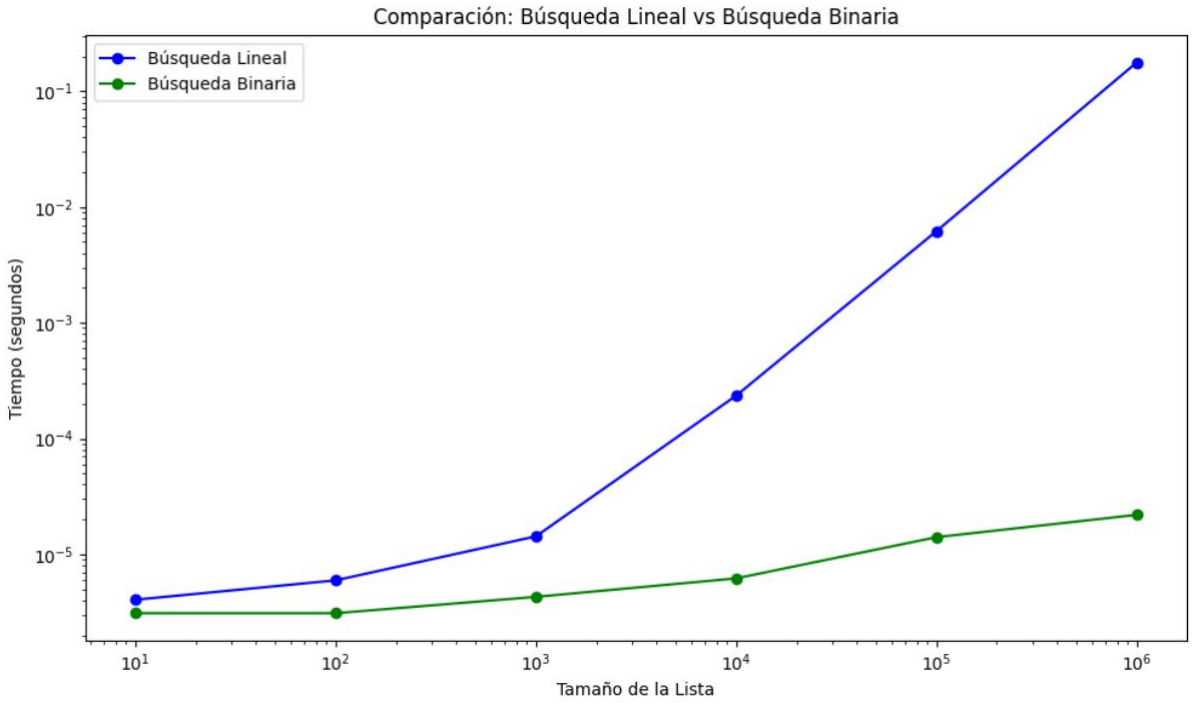
-Algoritmos de ordenamiento

Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

-Métodos de búsquedas en algoritmos:

- Búsqueda lineal: Es el algoritmo de búsqueda más simple, que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Es fácil de implementar, pero ineficiente en listas grandes.
  Complejidad: O(n) n es el número de elementos de la lista, en el peor de los casos, el algoritmo va a tener que comprobar cada uno de los elementos de la lista, hasta encontrar el objetivo, o bien, determinar que el objetivo no está. Cuando tenemos un conjunto de datos amplio, la búsqueda lineal es lenta

- Búsqueda binaria: Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos.
  Su eficiencia radica en eliminar la mitad de las posibilidades en cada iteración, lo que resulta en un crecimiento logarítmico del tiempo de ejecución.
  Complejidad: Tiene una complejidad 0(log n), esto quiere decir que el tiempo de ejecución del algoritmo crece logarítmicamente con el tamaño del conjunto de datos, podemos concluir en que el crecimiento es muy lento.



Complejidad:

La complejidad se mide en O(n) de esta forma se mide la eficiencia del logaritmo.

Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada.

En las búsquedas lineales el tiempo de búsqueda es directamente proporcional al tamaño de la lista. En las búsquedas binarias tiene un tiempo de ejecución de O (log n) donde el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista .

La búsqueda binaria es mucho más eficiente en listas grandes, que la búsqueda lineal por el tiempo de ejecución. Depende del tamaño de la lista el método que utilizaremos.

Tipos de Ordenamientos:

-Ordenamiento de burbuja: es recomendado para listas pequeñas o casi ordenadas. Consiste en un ordenamiento simple y fácil comparando cada elemento de la lista con el siguiente elemento e intercambiando los elementos si están ubicados de manera incorrecta

-Inserción: Consiste en insertar cada elemento de la lista en su posición correcta de la lista ordenada. es la mejor para las listas pequeñas igual que burbuja

-Selección: busca el elemento menor y lo coloca en la primera posición y repite el proceso a lo largo de toda la lista. Recomendado para listas pequeñas

-Quicksort: parecido a la búsqueda binaria porque divide a a partir de un pivot. Muy eficiente para listas grandes, evitando el peor de los casos.

Caso Práctico

Queremos ordenar listas de estudiantes de una comisión por las notas obtenidas en el parcial. Para esto vamos a usar el tipo de ordenamiento por selección y lo vamos a comparar con el de Quick sort.

Se crean dos listas con distintas cantidades de elementos. La primera es creada por nosotras y la segunda lo creara de manera random hasta llegar a 10.000 alumnos.

```
estudiantes = [
    {"nombre": "Ana", "nota": 75},
    {"nombre": "Luis", "nota": 60},
    {"nombre": "María", "nota": 95},
    {"nombre": "Juan", "nota": 80},
    {"nombre": "Sofía", "nota": 50},
    {"nombre": "Pedro", "nota": 40},
    {"nombre": "Alejo", "nota": 10},
    {"nombre": "Bruno", "nota": 65},
    {"nombre": "Tomas", "nota": 90},
    {"nombre": "Santiago", "nota":85},
    {"nombre": "Genaro", "nota": 30},
    {"nombre": "Benicio", "nota": 45},
]

estudiantes_random = [{"nombre": f"Estudiante{i}", "nota": random.randint(0, 10000)} for i in range(10000)]
```

Implementación - Selection Sort:

```python
def selection_sort_objs(lista):
    a = lista.copy()#creo una copia para no modificar la original
    n = len(a)
    for i in range(n):
        min_idx = i #asume que el estudiante actual tiene la nota mínima.
        for j in range(i + 1, n):
            if a[j]["nota"] < a[min_idx]["nota"]:#si se encuentra una nota menor, se actualiza el índice mínimo.
                min_idx = j
        a[i], a[min_idx] = a[min_idx], a[i]
    return a #Devuelve la lista ordenada por notas de menor a mayor.
```

Implementación - Quicksort:

```python
def quick_sort_objs(lista):
    if len(lista) <= 1:# si la lista está vacía o tiene un solo elemento, ya está ordenada.
        return lista
    pivote = lista[0]#toma el primer estudiante como referencia (pivote).
    menores = [x for x in lista[1:] if x["nota"] < pivote["nota"]]
    mayores = [x for x in lista[1:] if x["nota"] >= pivote["nota"]]
    return quick_sort_objs(menores) + [pivote] + quick_sort_objs(mayores)
```

Luego llamo a mis funciones desde mi programa principal. Aquí voy a comparar mi lista Original con las que me devuelve cada una de mis funciones para ver si estan ordenadas realmente.

Primero devuelvo mi lista original.

```python
#programa principal
print("Original:")
for e in estudiantes:
    print(f"{e['nombre']}: {e['nota']}")
```

Llamo a mi funcion select y muestro como esta ordenada. Y devolvemos el tiempo de ejecucion para la primera lista y para la lista random pero sin devolverla para no generar tantas lineas y evitar muchos datos en consola ya que comprobamos su funcion con la lista corta

```python
inicio_selection = time.time()
ordenados_selection = selection_sort_objs(estudiantes)
fin_selection = time.time()
tiempo_selection = fin_selection - inicio_selection

print("\nOrdenados con Selection Sort:")
for e in ordenados_selection:
    print(f"{e['nombre']}: {e['nota']}")
print(f"Tiempo de ejecución (Selection Sort): {tiempo_selection:.10f} segundos")

inicio_selection = time.time()
ordenados_selection = selection_sort_objs(estudiantes_random)
fin_selection = time.time()
tiempo_selection_random = fin_selection - inicio_selection
print(f"\nTiempo de ejecución (Selection Sort con lista aleatoria de 10.000 elementos): {tiempo_selection_random:.10f} segundos")
```

Hacemos lo mismo con la función Quick

```
inicio_quick = time.time()
ordenados_quick = quick_sort_objs(estudiantes)
fin_quick = time.time()
tiempo_quick = fin_quick - inicio_quick
print("\nOrdenados con Quick Sort:")
for e in ordenados_quick:
    print(f"{e['nombre']}: {e['nota']}")
print(f"Tiempo de ejecución (Quick Sort): {tiempo_quick:.10f} segundos")


inicio_quick_random = time.time()
ordenados_quick_random= quick_sort_objs(estudiantes_random)
fin_quick_random = time.time()
tiempo_quick_random = fin_quick_random - inicio_quick_random
print(f"\nTiempo de ejecución ( Quick con lista aleatoria de 10.000 elementos): {tiempo_quick_random:.10f} segundos")
```

Por consola podemos ver lo que imprime nuestro programa:
Primero nos va a dar nuestra lista Original:

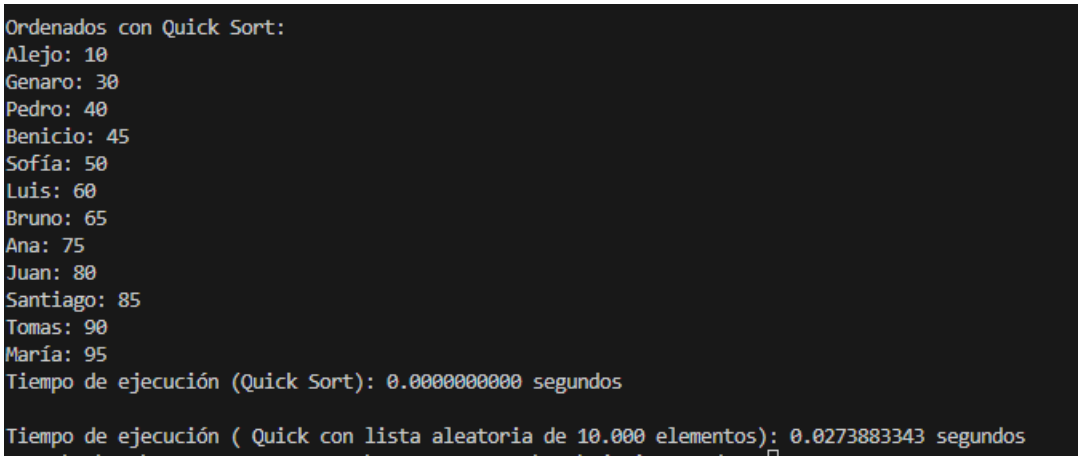
```
PS D:\GI\UTN\PRIMER CUATRIMESTRE\PROGRAMACION I\Trabajo integrador> & C:/Users/Asus/AppData/Local/Microsoft/WindowsApps/python3.9.exe "d:/GI
/UTN/PRIMER CUATRIMESTRE/PROGRAMACION I/Trabajo integrador/Integrador_stefani_valcarcel.py"
Original:
Ana: 75
Luis: 60
María: 95
Juan: 80
Sofía: 50
Pedro: 40
Alejo: 10
Bruno: 65
Tomas: 90
Santiago: 85
Genaro: 30
Benicio: 45
```

A continuacion tendremos la lista ordenada por selection, junto con los tiempos de la primera lista y la segunda

```
Ordenados con Selection Sort:
Alejo: 10
Genaro: 30
Pedro: 40
Benicio: 45
Sofía: 50
Luis: 60
Bruno: 65
Ana: 75
Juan: 80
Santiago: 85
Tomas: 90
María: 95
Tiempo de ejecución (Selection Sort): 0.0000000000 segundos

Tiempo de ejecución (Selection Sort con lista aleatoria de 10.000 elementos): 3.5591690540 segundos
```

Y Por ultimo la lista ordenada y los tiempos de la lista ordenada por Quick sort con los tiempos de cada lista.



Metodología Utilizada

La metodología consistió en:
-Investigar el funcionamiento teórico de los algoritmos.
-Implementar los algoritmos en Python.
-Aplicarlos en dos listas de estudiantes simuladas.
-Comparar los resultados obtenidos por consola.
-Medir tiempos de ejecución para analizar la eficiencia.

Resultados Obtenidos

-Ambos casos ordenaron correctamente la lista de estudiantes.

-En términos de tiempo, Quicksort demostró ser más rápido que Selection Sort, especialmente con listas más grandes .
-Se corroboró que, para listas pequeñas, la diferencia de tiempo no es significativa, pero la eficiencia se vuelve relevante al aumentar la cantidad de datos.

Conclusiones

-La elección del algoritmo depende del tamaño de la lista y la situación específica.
-Para listas pequeñas, los algoritmos Selection Sort son fáciles de implementar y funcionan correctamente.
-Para listas grandes, es recomendable utilizar algoritmos más eficientes como Quicksort.

-Este trabajo nos permitió comprender la importancia de medir y analizar la eficiencia algorítmica, reforzando habilidades de pensamiento lógico y resolución de problemas mediante programación.

Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms.
- Python Oficial: https://docs.python.org/3/library/
- Khan Academy: https://es.khanacademy.org/computing/computer-science/algorithms
- W3School:https://www.w3schools.com/