Trabajo Practico Integrador

Programación I

“Algoritmos de Búsqueda y Ordenamiento en Python”

Tecnicatura Universitaria en programación - UTN

ALUMNOS:

-Heber, Gustavo/Silva-gusheber90@gmail.com

-Roure, Ricardo/rickyroure10@gmail.com

DOCENTE TITULAR:

- Nicolas Quirós

DOCENTE TUTOR:

-Neyén Bianchi

FECHA DE ENTREGA:

-09/06

ÍNDICE

INTRODUCCIÓN

Los algoritmos de búsqueda son procedimientos sistemáticos que permiten la búsqueda de uno o más elementos específicos dentro de una estructura de datos más grande, ya sea una lista, un arreglo, árbol o grafo.

Su aplicación en sistemas informáticos es recurrente cuando trabajamos con base de datos por ejemplo al querer buscar información de un cliente como también en motores de búsqueda como Google o Bing, encontrar el camino más corto entre dos puntos en un mapa o en una red de computadoras depende, como así también a la hora de buscar un archivo en nuestras computadoras alojado dentro de la estructura de directorios.

Para ello debemos tener presente nuestro "*Elemento objetivo"* el cual es nuestro valor o dato que deseamos encontrar.

La "*Estructura de datos"* es el conjunto organizado de datos donde se realiza la búsqueda.

Tener en cuenta la "*Eficiencia"* del método utilizado, esta se mide en tiempo (cuantas operaciones necesita) y espacio (memoria que requiere). Esta se suele expresar utilizando notación Big O que describe como el tiempo de ejecución o el espacio de memoria varían a medida que el tamaño de entrada fluctúa.

Contamos con diferentes tipos de búsqueda:

Lineal: Es el método mas simple de todos y consiste en recorrer la estructura de datos elemento por elemento hasta encontrar el objetivo o llegar al final, sumamente útiles en listas pequeñas o desordenadas

Binaria: Es un método mucho mas eficiente que la búsqueda lineal, pero requiere que la estructura de datos se encuentre ordenada ya sea de forma creciente o decreciente. Funciona dividiendo repetidamente por la mitad el espacio de búsqueda hasta encontrar el elemento.

Las formas de ordenar nuestra estructura de datos son a través de diferentes algoritmos de ordenamiento (como burbuja, selección, inserción, mezcla, etc.)

MARCO TEÓRICO

Marco Teórico: Algoritmos de Búsqueda y Ordenamiento

Los algoritmos de búsqueda son procesos esenciales en computación para encontrar un dato específico dentro de una colección, como una lista o una base de datos. Su importancia es enorme, ya que influyen directamente en la velocidad de cualquier sistema que necesite recuperar información (Cormen et al., 2009).

¿Qué son los Algoritmos de Búsqueda?

Un algoritmo de búsqueda toma una colección de datos (por ejemplo, una lista de números) y un valor que queremos encontrar. Su meta es decirnos si ese valor existe en la colección y, si sí, dónde está (Goodrich & Tamassia, 2011).

Tipos Básicos de Búsqueda

1. **Búsqueda Lineal:** Este es el método más sencillo. Revisa cada elemento de la lista uno por uno hasta encontrar el que buscamos o llegar al final. Es fácil de usar y sirve para cualquier lista, ordenada o no.
    - **Velocidad (peor caso):** O(n), lo que significa que puede ser lenta en listas grandes, ya que a veces tiene que mirar todos los elementos (Sedgewick & Wayne, 2011).
    - **Memoria:** O(1), usa muy poca memoria extra.
2. **Búsqueda Binaria:** Mucho más rápida que la lineal, pero solo funciona si la lista está **ordenada**. Compara el valor buscado con el elemento del medio de la lista. Si no es ese, descarta la mitad de la lista donde no puede estar el valor y repite el proceso en la mitad restante.
    - **Velocidad (peor caso):** O(logn), lo que la hace muy eficiente para listas grandes.
    - **Memoria:** O(1) para la versión directa.

Algoritmos de Ordenamiento: La Ayuda para la Búsqueda

Como la búsqueda binaria necesita listas ordenadas, los **algoritmos de ordenamiento** son clave. Estos reorganizan los elementos de una lista en un orden específico (por ejemplo, de menor a mayor).

Aquí algunos ejemplos:

- **Burbuja, Selección, Inserción:** Son más sencillos pero generalmente más lentos (O(n2)), lo que significa que tardan mucho con listas grandes.
- **Mezcla (Merge Sort), Rápido (Quick Sort):** Son mucho más eficientes (O(nlogn)) y se usan para ordenar grandes volúmenes de datos.

La elección de un algoritmo de ordenamiento y de búsqueda depende de la **cantidad de datos**, si ya están **ordenados** o si el costo de ordenarlos se justifica por la **frecuencia de las búsquedas**. Medimos su **eficiencia** por el tiempo y la memoria que usan (Goodrich & Tamassia, 2011).

Metodología del Caso Práctico: Cómo Realizamos las Pruebas

Para ver cómo funcionan los algoritmos en la práctica, dividimos nuestro trabajo en dos partes: primero, preparamos nuestras herramientas de código, y luego, las usamos para simular un escenario y medir su rendimiento.

1. Nuestras Herramientas de Código (Funciones.py)

Creamos un archivo llamado Funciones.py donde guardamos todas las operaciones necesarias. Esto nos ayuda a tener el código organizado y listo para usar:

- **generate_random_list(size)**: Crea listas de números aleatorios únicos del tamaño que le pidamos.
- **busqueda_lineal(lista, objetivo)**: Busca un número en la lista revisando cada elemento uno por uno.
- **busqueda_binaria(lista, objetivo)**: Busca un número en una lista que ya está ordenada, descartando mitades en cada paso.
- **bubble_sort(arr)**: Ordena una lista usando el método "burbuja".
- **insertion_sort(arr)**: Ordena una lista usando el método de "inserción".
- **selection_sort(arr)**: Ordena una lista usando el método de "selección".
- **quicksort(arr)**: Ordena una lista usando el método "quicksort" (rápido).

2. Cómo Hicimos las Pruebas (Código Principal)

Nuestro programa principal usa las funciones de Funciones.py y nos guía a través de las pruebas con un menú interactivo.

Para ello, al inicio, el programa crea dos listas de números aleatorios:

- Una **"lista grande"** con 14.530 números.

- Una **"lista chica"** con 100 números.

Ejecución Paso a Paso

1. **Opción "Búsqueda Lineal":**

    - Pregunta si se trabajará con la lista "chica" o "grande".
    - **Lista "chica"**: En ese caso muestra primero la lista y luego pide un número para buscar. Después, usa la búsqueda lineal y muestra el tiempo que tardó.
    - **Lista "grande"**: El número a buscar ya está definido (es el que está en la posición 7649 de la lista original, desordenada). El programa usa la función de búsqueda lineal y muestra cuánto tardó.

2. **Si eliges "Búsqueda Binaria":**

    - Primero, solicita al usuario que elija un método para ordenar la lista (Burbuja, Inserción, Selección o Quicksort). Esto es necesario para la búsqueda binaria.
    - El programa toma la "lista grande" desordenada y la ordena usando el método escogido. Mide cuánto tiempo tardó en ordenarse.
    - Una vez ordenada, el número a buscar para la búsqueda binaria es el que está en la posición 7649. Luego, el programa usa la busqueda_binaria sobre la lista ordenada y arroja el tiempo que tardó en encontrarlo.

Midiendo el Tiempo

Para saber cuánto tardó cada operación (ordenar y buscar), usamos una herramienta especial de Python llamada timeit.default_timer(). Esta nos permite cronometrar con mucha precisión el tiempo que pasa desde que una función empieza hasta que termina.

Analizando los Resultados

El programa te muestra todos los tiempos medidos directamente en pantalla. Esto nos permite comparar rápidamente qué tan rápido es cada algoritmo de ordenamiento y cómo ordenar una lista puede hacer que la búsqueda binaria sea muchísimo más veloz que la lineal, especialmente cuando trabajamos con muchos datos.

METODOLOGÍA UTILIZADA

Para analizar cómo funcionan los algoritmos de búsqueda y ordenamiento en la práctica, seguimos un proceso claro dividido en dos partes: primero, preparamos nuestras "herramientas" (las funciones de código), y luego, las usamos para hacer las pruebas.

Paso 1: Archivo `Funciones.py`

Para que nuestro programa fuera ordenado y fácil de manejar, creamos un archivo especial llamado ***Funciones.py***. Acá guardamos todas las funciones que emplearemos para la generación de listas, tipos de búsqueda y ordenamiento.

1. ***generate_random_list(size):*** Esta herramienta crea nuestras listas de números. Le decimos qué tan grande queremos la lista (size), y ella nos da una lista llena de números aleatorios y únicos (sin que se repitan) entre 1 y 99,999. Esto simula, por ejemplo, los números de legajo de alumnos o de pólizas.

2. ***`busqueda_lineal(lista, objetivo)`:*** Esta es nuestra herramienta de **búsqueda simple**. Revisa la lista número por número, de principio a fin, hasta encontrar el "objetivo" (el número que buscamos). Si lo encuentra, nos dice dónde está; si no, nos avisa que no lo encontró.

3. ***busqueda_binaria(lista, objetivo):*** Esta es una herramienta de **búsqueda mucho más rápida**, pero tiene una condición: la lista debe estar **ordenada**. Funciona como si buscaras una palabra en un diccionario: vas al medio, y si tu palabra está antes, vas a la primera mitad; si está después, vas a la segunda mitad. Así, reduce la búsqueda a la mitad en cada paso.

4. **Herramientas de Ordenamiento:** Son las que usamos para poner los números de la lista en orden (de menor a mayor en nuestro caso), algo esencial para que la búsqueda binaria funcione:

    - ***bubble_sort(arr)*** **(Ordenamiento Burbuja):** Es el más simple. Compara números vecinos y los cambia de lugar si están desordenados, repitiendo esto hasta que todo esté en orden.
    - ***insertion_sort(arr)*** **(Ordenamiento por Inserción):** Toma un número a la vez y lo pone en su lugar correcto dentro de los números que ya están ordenados.
    - ***selection_sort(arr)*** **(Ordenamiento por Selección):** Busca el número más pequeño (o el más grande) que queda y lo pone al principio de la lista.
    - ***quicksort(arr)*** **(Ordenamiento Rápido):** Es un método muy eficiente. Elige un número de la lista, pone los números más pequeños antes y los más grandes después, y luego hace lo mismo con esas partes más chicas hasta que toda la lista esté ordenada.

Paso 2: Hacemos las Pruebas (Código Principal)

Con nuestras herramientas listas en Funciones.py, ejecutamos el programa principal, que nos guía a través de las diferentes pruebas:

Al comenzar el programa, creamos dos listas de números aleatorios usando nuestra herramienta generate_random_list:

- Una **lista "grande"** con 14.530 números.
- Una **lista "chica"** con 100 números.

Estas listas son nuestras "bases de datos" para los experimentos.

Así Ejecutamos las Pruebas

El programa nos muestra un menú interactivo.

1. **Si elegimos "Búsqueda Lineal":**
    - El programa nos pregunta si queremos usar la lista "chica" o la "grande".
    - **Para la lista "chica"**: Nos muestra la lista. Nos pide que ingresemos el número que queremos buscar. Luego, usa la busqueda_lineal y nos dice cuánto tardó en encontrarlo.
    - **Para la lista "grande"**: Acá, el número a buscar ya está predefinido: es el que está en la **posición 7649 de la lista original (desordenada)**. Esto nos permite simular una búsqueda en una base de datos grande sin ordenar. Medimos cuánto tarda la busqueda_lineal en encontrarlo.

2. **Si elegimos "Búsqueda Binaria":**
   Como la búsqueda binaria necesita que la lista esté ordenada, el programa nos pide que elijamos primero una de nuestras herramientas de ordenamiento (Burbuja, Inserción, Selección, o Quicksort)

    Para el ordenamiento que elijamos:

    - Medimos cuánto tiempo tarda la herramienta de ordenamiento en poner toda la lista "grande" en orden.
    - Una vez que la lista está ordenada, el número a buscar para la búsqueda binaria se define: es el que está en la posición 7649 de la lista *ya ordenada*.
    - Luego, usamos la busqueda_binaria sobre esa lista ya ordenada y medimos cuánto tiempo tarda en encontrar el número.

Cómo Medimos el Tiempo

Para saber cuánto tardó cada operación (ordenar o buscar), usamos una función especial de Python llamada timeit.default_timer(). Esta función nos da un "cronómetro" muy preciso. Lo activamos antes de que la función empiece a trabajar y lo paramos justo cuando termina. La diferencia entre esos dos momentos nos da el tiempo exacto que tardó.

RESULTADO OBTENIDO

Tras la ejecución del caso práctico y la medición de tiempos con los algoritmos implementados, se obtuvieron los siguientes resultados, que ilustran claramente las diferencias de eficiencia entre los métodos de búsqueda y ordenamiento. Los tiempos se expresan en segundos.

Escenario	Lista utilizada	Elemento Buscado	Posición Encontrada	Tiempo de Búsqueda (s)
Lista Chica	(100 elementos)	95564	59	1.89×10−5
Lista Grande	Desordenada (14530 elementos)	59234	7649	2.67×10−4

Observaciones:

- En la **lista chica**, la búsqueda lineal fue extremadamente rápida, como era de esperar para un tamaño tan reducido.
- Para la **lista grande y desordenada**, el tiempo de búsqueda lineal aumentó significativamente, aunque sigue siendo un valor pequeño, demuestra su mayor costo al tener que recorrer una parte considerable de la lista para encontrar el elemento en una posición intermedia.

Algoritmo de Ordenamiento	Tiempo de Ordenamiento (s)
Burbuja (Bubble Sort)	6.490
Inserción (Insertion Sort)	3.050
Selección (Selection Sort)	2.695
Rápido (Quicksort)	27

Observaciones:

- Como se predice en la teoría, los algoritmos de complejidad O(n2) (Burbuja, Inserción, Selección) mostraron tiempos de ordenamiento considerablemente altos para una lista de 14.530 elementos.
- El **Ordenamiento Burbuja** fue el más lento, superando los 6 segundos.
- **Inserción** y **Selección** fueron más rápidos que Burbuja, pero aún tomaron varios segundos.
- El **Ordenamiento Rápido (Quicksort)** demostró una eficiencia drástica, completando el ordenamiento en apenas 0.027 segundos. Esto resalta la superioridad de los algoritmos O(nlogn) para volúmenes de datos considerables.

Búsqueda Binaria en la Lista Grande Ordenada

En todos los casos de búsqueda binaria, se buscó el valor 52707 (que se encontraba en la posición 7649 después del ordenamiento de la lista grande).

A pesar de que la lista grande fue ordenada por diferentes algoritmos, el **tiempo de la búsqueda binaria en sí fue consistentemente extremadamente bajo y similar** para todas las pruebas. Esto confirma la alta eficiencia de la búsqueda binaria una vez que los datos están preparados.

CONCLUSIÓN

Los resultados demuestran que, si bien la búsqueda lineal es suficiente para listas muy pequeñas o cuando los datos no pueden ser pre-ordenados, su rendimiento decae considerablemente con el tamaño de la lista. En contraste, la búsqueda binaria es notablemente más rápida para listas grandes, pero exige un paso previo de ordenamiento.

El costo de este pre-ordenamiento varía drásticamente: para la lista grande, los algoritmos O(n2) como Burbuja tardan varios segundos, lo que podría hacer que la combinación

(ordenar + buscar) sea menos eficiente que una búsqueda lineal si la búsqueda se hace pocas veces. Sin embargo, con un algoritmo eficiente como Quicksort, el costo de ordenamiento es mínimo (0.027 s). Esto significa que, para múltiples búsquedas en una base de datos grande, el uso de un algoritmo de ordenamiento eficiente (como Quicksort) seguido de una búsqueda binaria es, con diferencia, la opción más efectiva en términos de tiempo total.

BIBLIOGRAFÍA

- **Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.** (2009). *Introducción a los Algoritmos* (3ra ed.). Prentice Hall.
- **Goodrich, M. T., & Tamassia, R.** (2011). *Diseño de Algoritmos: Fundamentos, Análisis y Ejemplos de Internet* (2da ed.). Limusa Wiley.
- **Sedgewick, R., & Wayne, K.** (2011). *Algoritmos* (4ta ed.). Pearson Educación.