Trabajo integrador búsqueda y ordenamiento

MATERIA: Programación I

PROFESOR: Ariel Enferrel

TUTOR: Maximiliano Sar Fernandez

FECHA DE ENTREGA 9/6/2025

ALUMNOS: Marcos Janchuk / Gorosito Luciano

DNI´s: 42089970 / 42440302

INTRODUCCIÓN

Los algoritmos de búsqueda y ordenamiento, tienen gran importancia en el ámbito del desarrollo de software, sobre todo en proyectos que gestionen gran volumen de datos, donde la eficiencia algorítmica, es imprescindible al momento de crear software que funcione de la mejor forma posible, con el objetivo de brindar una excelente experiencia de usuario.

MARCO TEÓRICO

Algoritmos de Búsqueda

El mundo del desarrollo de software está en constante evolución. A lo largo del tiempo, se han creado algoritmos cada vez más eficientes, especialmente en áreas clave como la búsqueda y el ordenamiento.

Dicho esto, los algoritmos de búsqueda, son aquellos que pretenden buscar un dato específico dentro de un conjunto de datos. Existen diferentes algoritmos, como lo son:

- **Búsqueda lineal:**
  Es aquel que compara dato por dato de forma secuencial.
  Tiene como característica, su fácil implementación, pero mala eficiencia en grandes conjuntos de datos.
- **Búsqueda binaria**:
  Es más eficiente que búsqueda lineal, debido a que funciona directamente sobre conjuntos de datos ordenados, dividiendo el mismo en mitades, buscando el elemento requerido en la mitad que corresponda, hasta encontrarlo.
- **Búsqueda de interpolación**:
  Es una mejora de la búsqueda binaria que, en lugar de dividir siempre por la mitad, calcula una posición estimada más inteligente basándose en el valor que buscamos.
  Asume que los datos están uniformemente distribuidos, calcula dónde debería estar el elemento usando interpolación matemática.
- **Búsqueda de hash**: Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes.

Dentro de sus aplicaciones las podemos ver en búsqueda de palabras clave en un documento, búsqueda de archivos en un sistema de archivos, búsqueda de registros en una base de datos, búsqueda de la ruta más corta en un gráfico, búsqueda de soluciones a problemas de optimización.

Al momento de crear un algoritmo de búsqueda, debemos de entender acerca de la complejidad y su relación con la eficiencia. La cual mide cuántos recursos necesita un algoritmo para resolver un problema.

Complejidad Temporal ¿Cuánto tiempo tarda? Medida en número de operaciones básicas.

Complejidad Espacial ¿Cuánta memoria usa? Medida en cantidad de memoria adicional necesaria.

Notación Big O - Jerarquía de Eficiencia:

De más eficiente a menos eficiente:

1. O(1) - Constante → Excelente
2. O(log log n) - Doble logarítmico → Excelente
3. O(log n) - Logarítmico → Muy bueno
4. O(n) - Lineal → Aceptable
5. O(n log n) - Linealítmico → Regular
6. $O(n^2)$ - Cuadrático → Malo
7. $O(2^n)$ - Exponencial → Terrible

Búsqueda Lineal: O(n) Mejor caso: O(1) elemento al inicio, peor caso: O(n) elemento al final.

Búsqueda Binaria: O(log n) Todos los casos: O(log n), porque O(log n) en cada paso divide el problema por la mitad. en 1000 elementos tenemos un máximo 10 comparaciones y en 1,000,000 elementos máximo 20 comparaciones.

Búsqueda por Interpolación: O(log log n) Mejor caso: O(log log n), distribución uniforme Peor caso: O(n) distribución muy desigual Promedio: O(log log n) porque O(log log n) Encuentra la posición estimada más inteligentemente que la binaria.

Búsqueda Hash: O(1) Mejor caso: O(1) - sin colisiones Peor caso: O(n) - muchas colisiones Promedio: O(1) ¿Por qué O(1)? Acceso directo calculando la posición con la función hash.

¿Cuándo usar cada uno?

- Lineal: Datos pequeños o desordenados
- Binaria: Datos ordenados, gran volumen de datos
- Interpolación: Datos ordenados y uniformemente distribuidos
- Hash: Búsquedas muy frecuentes, datos que no necesitan estar ordenados

Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son procedimientos que reorganizan los elementos de una colección siguiendo un criterio específico generalmente ascendente o descendente.

Tipos de Algoritmos de Ordenamiento:

Algoritmos Simples O(n²):

1. Bubble Sort (Ordenamiento Burbuja) Compara elementos adyacentes e intercambia si están en orden incorrecto. Repite hasta que no haya más intercambios. Complejidad: O(n²)

2. Selection Sort (Ordenamiento por Selección) Encuentra el elemento mínimo y lo coloca al inicio. Repite para el resto de la lista. Complejidad: O(n²)

3. Insertion Sort (Ordenamiento por Inserción) Toma elementos uno a uno y los inserta en su posición correcta en la porción ya ordenada. Complejidad: O(n²)

Algoritmos Eficientes - O(n log n):

4. Merge Sort (Ordenamiento por Mezcla) Divide la lista por la mitad recursivamente, ordena cada mitad y luego las combina. Complejidad: O(n log n)

5. Quicksort (Ordenamiento Rápido) Selecciona un pivote, participa la lista y ordena recursivamente las sublistas. Complejidad: O(n log n) promedio, O(n²) peor caso

6. Heap Sort (Ordenamiento por Montículo) Construye un heap y extrae repetidamente el elemento máximo. Complejidad: O(n log n)

Análisis de Complejidad de Algoritmos de Ordenamiento:

Bubble Sort - O(n²) Mejor caso: O(n) lista ya ordenada Peor caso: O(n²) - lista ordenada inversamente Promedio: O(n²) ¿Por qué O(n²)? Compara cada elemento con todos los demás.

Selection Sort - O(n²) Todos los casos: O(n²) ¿Por qué O(n²)? Siempre busca el mínimo en toda la lista restante.

Insertion Sort - O(n²) Mejor caso: O(n) - lista ya ordenada Peor caso: O(n²) - lista ordenada inversamente Promedio: O(n²) ¿Por qué O(n²)? En cada inserción puede recorrer toda la parte ordenada.

Merge Sort - O(n log n) Todos los casos: O(n log n) ¿Por qué O(n log n)? Divide log n veces y combina n elementos cada vez.

Quicksort - O(n log n) Mejor caso: O(n log n) - pivote siempre central Peor caso: O(n²) pivote siempre extremo Promedio: O(n log n) ¿Por qué O(n log n)? Particiona log n veces y procesa n elementos cada vez.

Heap Sort - O(n log n) Todos los casos: O(n log n) ¿Por qué O(n log n)? Construye heap O(n) y extrae n elementos O(log n) cada uno.

¿Cuándo usar cada algoritmo de ordenamiento?

- Bubble Sort: Sólo listas pequeñas
- Selection Sort: Listas pequeñas, memoria limitada

- Insertion Sort: Listas pequeñas o casi ordenadas
- Merge Sort: Cuando se necesita estabilidad y O(n log n) garantizado
- Quick Sort: Uso general, mejor rendimiento promedio
- Heap Sort: Cuando se necesita O(n log n) garantizado sin recursión