Introducción a los Algoritmos

1. ¿Qué es un algoritmo?

Un algoritmo es una secuencia ordenada de instrucciones diseñadas para resolver un problema o cumplir una función específica. En el ámbito de la programación, los algoritmos son esenciales para manipular datos, realizar operaciones matemáticas y automatizar procesos de manera eficiente.

2. ¿Cuáles son las características de un algoritmo efectivo?

- **Optimización**: Utiliza recursos (como tiempo de ejecución y memoria) de forma eficiente.
- **Fiabilidad**: Garantiza resultados correctos para cualquier entrada válida.
- **Simplicidad**: Debe ser claro, fácil de interpretar y adaptable a cambios futuros.

3. ¿Por qué son esenciales los algoritmos en la programación?

Los algoritmos son el pilar de la programación, ya que permiten abordar problemas de forma sistemática y eficiente. Facilitan el desarrollo de software al optimizar tareas y resolver desafíos complejos, desde cálculos básicos hasta aplicaciones avanzadas como inteligencia artificial.

4. Ejemplo práctico: Algoritmo para sumar dos números

Problema: Calcular la suma de dos números ingresados por el usuario.
Pseudocódigo:

```
Comenzar
Solicitar el primer número (X)
Solicitar el segundo número (Y)
Capturar X
Capturar Y
Calcular Z = X + Y
Presentar el valor de Z
```

`Finalizar`

Algoritmos de Ordenamiento

1. ¿Cómo operan los algoritmos de ordenamiento?

- **Quicksort**: Utiliza un enfoque de "divide y conquista", seleccionando un elemento pivote para dividir la lista en dos partes (menores y mayores que el pivote), y ordena cada parte de manera recursiva.

- **Mergesort**: También aplica "divide y conquista", dividiendo la lista en dos mitades, ordenándolas por separado y luego fusionándolas en una lista ordenada.

2. Complejidad Temporal (Notación Big-O)

- **Quicksort**: En el mejor caso y promedio, tiene una complejidad de $O(n \log n)$, pero en el peor caso (cuando el pivote es el menor o mayor elemento) puede llegar a $O(n^2)$.

- **Mergesort**: Mantiene una complejidad estable de $O(n \log n)$ en todos los casos, gracias a su enfoque de división constante.

3. ¿En qué situaciones usar cada algoritmo?

- **Quicksort**: Es ideal para la mayoría de escenarios debido a su eficiencia promedio y bajo uso de memoria adicional, aunque puede ser menos eficiente en casos extremos.

- **Mergesort**: Se prefiere cuando se necesita un ordenamiento estable (que preserve el orden relativo de elementos iguales) o cuando se trabaja con estructuras de datos grandes, como listas enlazadas.

Algoritmos de Búsqueda

1. ¿Cómo opera la búsqueda binaria?

La búsqueda binaria localiza un elemento en una lista ordenada dividiendo repetidamente el rango de búsqueda a la mitad. Compara el elemento buscado con el valor central y descarta una mitad según si el objetivo es mayor o menor, repitiendo el proceso hasta encontrarlo o determinar que no está.

2. Comparación de complejidades

- **Búsqueda Lineal**: Su complejidad es $O(n)$, ya que examina cada elemento de la lista uno por uno.
- **Búsqueda Binaria**: Mucho más eficiente con $O(\log n)$, gracias a su enfoque de reducción logarítmica del espacio de búsqueda.

Preguntas de análisis

- **¿En qué casos es mejor usar búsqueda lineal en lugar de búsqueda binaria?** La búsqueda lineal es más adecuada para listas pequeñas (donde el costo de ordenar para búsqueda binaria no compensa) o cuando la lista no está ordenada, ya que no requiere preprocesamiento.
- **¿Qué sucede si se intenta usar búsqueda binaria en una lista no ordenada?** La búsqueda binaria falla en listas desordenadas porque depende de la propiedad de orden para descartar mitades. Sin orden, las comparaciones no tienen sentido y puede omitir el elemento buscado o dar resultados incorrectos.

Algoritmos de Grafos

1. ¿Cómo funcionan estos algoritmos?

- **Dijkstra**: Determina la ruta más corta desde un nodo inicial a todos los demás en un grafo con pesos no negativos, usando una estrategia de selección greedy.
- **BFS (Búsqueda en Anchura)**: Recorre el grafo nivel por nivel, explorando todos los vecinos de un nodo antes de avanzar, ideal para grafos no ponderados.
- **DFS (Búsqueda en Profundidad)**: Explora un camino del grafo hasta su máxima profundidad antes de retroceder, útil para detectar ciclos o explorar todos los nodos.

2. Complejidad Temporal

- **Dijkstra**: $O((V+E)\log V)$ con una implementación que use un montículo binario para gestionar prioridades.

- **BFS y DFS**: Ambos tienen complejidad $O(V + E)$, donde V es el número de vértices y E el número de aristas.

Preguntas de análisis

- ¿**Por qué Dijkstra no funciona con pesos negativos?** Dijkstra asume que una vez que encuentra la distancia más corta a un nodo, esta no puede mejorar. Con pesos negativos, una ruta posterior podría reducir esa distancia, rompiendo esta suposición y dando resultados incorrectos. Para grafos con pesos negativos, se prefiere el algoritmo de Bellman-Ford.

- ¿**En qué situaciones preferirías usar DFS en lugar de BFS?** DFS es más adecuado cuando se necesita explorar todos los caminos posibles en un grafo (como en la búsqueda de ciclos o soluciones en laberintos) o cuando la memoria es limitada, ya que usa menos espacio al no almacenar todos los nodos de un nivel como BFS. Sin embargo, BFS es mejor para encontrar la ruta más corta en grafos no ponderados.

Algoritmos de Compresión y Otros

1. ¿Cómo opera el algoritmo de Huffman?

Huffman genera un código óptimo para compresión sin pérdida al construir un árbol binario basado en las frecuencias de los símbolos. Los símbolos más frecuentes reciben códigos más cortos, minimizando el tamaño total del mensaje codificado.

2. ¿Qué problema resuelve el algoritmo de Kadane?

Kadane identifica la subsecuencia contigua (subarray) con la suma máxima dentro de un arreglo numérico, incluso si contiene números negativos. Es eficiente y se usa para problemas de optimización en arrays.

Preguntas de análisis

- ¿**Por qué el algoritmo de Huffman es óptimo para la compresión sin pérdida?** Huffman es óptimo porque asigna códigos de longitud variable según la frecuencia de los símbolos, asegurando que los más comunes usen menos bits y los menos frecuentes más, lo que

minimiza el tamaño total del mensaje sin perder información. Esto lo hace ideal para formatos como ZIP o JPEG (en ciertas etapas).

- ¿**En qué situaciones prácticas se utiliza el algoritmo de Kadane?** Kadane es útil en áreas como análisis financiero (para detectar períodos de máximo beneficio en datos de ganancias/pérdidas), procesamiento de señales (para encontrar segmentos con mayor energía) y bioinformática (para identificar regiones significativas en secuencias genómicas). Su eficiencia lineal $O(n)$ lo hace práctico para grandes volúmenes de datos.