Explorando los Algoritmos más Importantes en Programación

Objetivo: Investigar y comprender los algoritmos más importantes en programación, sus características, aplicaciones y cómo implementarlos.

Introducción a los algoritmos

¿Qué es un algoritmo?

Es un conjunto de instrucciones definidas y ordenadas que permiten resolver un problema o realizar una tarea. Se compone de pasos lógicos que transforman una entrada en una salida deseada.

Características de un buen algoritmo

- Eficiencia: Utiliza los recursos de manera óptima
- Corrección: Proporciona la salida esperada para todas las entradas validas
- Claridad: Fácil de entender y modificar

Importancia de los algoritmos en la programación

Son esenciales para la resolución de problemas en computación, optimizando el uso de recursos y mejorando el rendimiento de los sistemas.

Ejemplo De Algoritmo

Suma de dos números:

Algoritmo para sumar dos números

Def sumar(a, b):

Return a + b

Print(sumar(3, 5)) #Salida: 8

Algoritmos De Ordenamiento

Quicksort

- Funcionamiento: Divide la lista en sublistas menores y mayores a un pivote, oredenándolas recursivamente.
- Complejidad: Promedio $O(n \log n)$, peor caso $O(n^2)$.
- Eficiencia: Rápido en promedio, pero no estable.
- Ejemplo:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivote = arr[len(arr) // 2]
    izquierda = [x for x in arr if x < pivote]
    centro = [x for x in arr if x == pivote]
    derecha = [x for x in arr if x > pivote]
```

```
    return quicksort(izquierda) + centro + quicksort(derecha)
```

Mergersort

- Funcionamiento: Divide la lista en mitades y las mezclas ordenadas.
- Complejidad: $O(n \log n)$ en todos los casos.
- Eficiencia: Estable y eficiente en datos grandes, pero usa más memoria

```
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    medio = len(arr) // 2
    izquierda = mergesort(arr[:medio])
    derecha = mergesort(arr[medio:])
    return merge(izquierda, derecha)


def merge(izquierda, derecha):
    resultado = []
    while izquierda and derecha:
        if izquierda[0] < derecha[0]:
            resultado.append(izquierda.pop(0))
        else:
            resultado.append(derecha.pop(0))
    return resultado + izquierda + derecha
```

- **¿Por qué Quicksort es más rápido en promedio, pero no es estable?**
- Quicksort es más rápido en promedio porque su estrategia de división y conquista permite reducir significativamente el número de comparaciones y movimientos en cada iteración. Sin embargo, no es estable porque puede intercambiar elementos iguales de una manera no predecible al elegir pivotes diferentes en cada partición
- **¿En qué situaciones preferirías usar Mergesort?**
  Mergesort es preferible cuando se necesita un algoritmo estable (mantiene el orden relativo de elementos iguales) o cuando se trabaja con listas grandes almacenadas en discos, ya que su eficiencia en acceso secuencial es mejor que en acceso aleatorio. Además, se usa en estructuras de datos vinculadas porque no requiere acceso aleatorio a los elementos.

Algoritmos De Búsqueda

1. Búsqueda Binaria

- Funcionamiento:
  - La búsqueda binaria divide repetidamente la lista ordenada en mitades hasta encontrar el elemento deseado o determinar que no está presente.
- Es eficiente en grandes volúmenes de datos debido a su complejidad logarítmica.
- Requiere lista ordenada
- Complejidad $O(\log n)$

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

2. Búsqueda Lineal

- Funcionamiento:

- Recorre la lista elemento por elemento hasta encontrar el objetivo o llegar al final.

- No requiere una lista ordenada.

- Complejidad: $O(n)$

```
def busqueda_lineal(arr, objetivo):
    for i, val in enumerate(arr):
        if val == objetivo:
            return i
    return -1
```

Comparación de Complejidad Temporal

Busqueda Binaria:
- Mejor Caso: $O(1)$, Peor Caso: $O(\log \quad n)$

Búsqueda Lineal:

- Mejor Caso: $O(1)$, Peor Caso: $O(n)$

¿En qué casos es preferible usar búsqueda lineal en lugar de búsqueda binaria?

- Cuando la lista es pequeña, ya que la sobrecarga de ordenar antes de buscar con binaria puede no ser eficiente.
- Cuando los elementos están en una estructura de datos que no permite acceso aleatorio eficiente (Como listas enlazadas)
- Cuando la lista está desordenada y no vale la pena ordenarla antes de la búsqueda

¿Qué pasa si intentas usar búsqueda binaria en una lista no ordenada?

- La búsqueda binaria asume que los datos están ordenados para dividir correctamente la lista en mitades.
- Si se aplica en una lista desordenada, puede devolver resultados incorrectos o no encontrar un elemento presente en la lista.

Algoritmo del Grafos

BFS (Breadth-First Search - Búsqueda en Anchura)

- Recorre el grafo nivel por nivel, explorando todos los nodos vecinos antes de pasar a los siguientes niveles.
- Usa una **cola (FIFO)** para gestionar los nodos visitados.
- Se usa para encontrar la ruta más corta en grafos no ponderados.

Complejidad: $O(V + E)$, donde V es el número de vértices y E el número de aristas.

```
from collections import deque


def bfs(grafo, inicio):
    visitados = set()
    cola = deque([inicio])
    while cola:
        nodo = cola.popleft()
        if nodo not in visitados:
            print(nodo, end=" ")
            visitados.add(nodo)
            cola.extend(grafo[nodo] - visitados)


grafo = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F'},
```

```
    'D': {'B'},
    'E': {'B', 'F'},
    'F': {'C', 'E'}
}


bfs(grafo, 'A')
```

DFS (Depth-First Search - Búsqueda en Profundidad)

- Explora el grafo en profundidad, visitando completamente un camino antes de retroceder.
- Usa una pila (LIFO) o recursión.
- Se usa en problemas como detección de ciclos y análisis de conectividad.

Complejidad: $O(V + E)$

```
def dfs(grafo, inicio, visitados=None):
    if visitados is None:
        visitados = set()
    visitados.add(inicio)
    print(inicio, end=" ")
    for vecino in grafo[inicio] - visitados:
        dfs(grafo, vecino, visitados)


dfs(grafo, 'A')
```

Dijkstra (Camino más corto en grafos ponderados)

- Calcula el camino más corto desde un nodo origen a todos los demás en un grafo con pesos positivos.
- Usa una **cola de prioridad** para siempre expandir el nodo con menor costo acumulado.

Complejidad: $O\big((V + E) \log V\big)$ con una cola de prioridad

```
import heapq

def dijkstra(grafo, inicio): distancias = {nodo: float('inf') for nodo in grafo} distancias[inicio] = 0 cola = [(0, inicio)]

while cola:
    distancia_actual, nodo_actual = heapq.heappop(cola)
    for vecino, peso in grafo[nodo_actual].items():
        nueva_distancia = distancia_actual + peso
        if nueva_distancia < distancias[vecino]:
            distancias[vecino] = nueva_distancia
```

```
            heapq.heappush(cola, (nueva_distancia, vecino))

    return distancias


grafo_ponderado = { 'A': {'B': 1, 'C': 4}, 'B': {'A': 1, 'D': 2, 'E': 5}, 'C': {'A': 4, 'F': 3}, 'D': {'B': 2}, 'E': {'B': 5, 'F': 1}, 'F': {'C': 3, 'E': 1} }

print(dijkstra(grafo_ponderado, 'A'))
```

1. ¿Por qué Dijkstra no funciona con pesos negativos?
   - Dijkstra asume que una vez que encuentra la distancia mínima a un nodo, esa distancia no cambiará.
   - Con pesos negativos, esta suposición es incorrecta, ya que un camino más corto podría encontrarse más adelante.
   - En su lugar, se debe usar **Bellman-Ford**, que maneja pesos negativos.
2. ¿En qué situaciones preferirías usar DFS en lugar de BFS?
   - Cuando necesitas recorrer completamente un camino antes de explorar otros (por ejemplo, para detectar ciclos o resolver laberintos).
   - Cuando el grafo es **muy grande** y no necesitas encontrar la ruta más corta, ya que DFS usa menos memoria que BFS.

Algoritmos de Compresión y Otros

Huffman (Compresión de datos)

- Se basa en la frecuencia de los caracteres en un texto para construir un árbol binario de codificación.
- Los caracteres más frecuentes tienen códigos más cortos, reduciendo el tamaño total del archivo.
- Se usa en formatos de compresión como **ZIP, JPEG y MP3**.

Pasos del algoritmo:

1. Calcular la frecuencia de cada símbolo en los datos de entrada.
2. Crear un nodo para cada símbolo y agregarlos a una cola de prioridad.
3. Combinar los dos nodos con menor frecuencia en un nuevo nodo.
4. Repetir hasta formar un árbol binario.
5. Asignar códigos binarios a cada símbolo según su posición en el árbol.

Complejidad: $O(n \log n)$ debido al uso de la cola de prioridad.

```
from heapq import heappush, heappop


def huffman(cadena):

    frecuencia = {char: cadena.count(char) for char in set(cadena)}

    heap = [[peso, [simbolo, ""]] for simbolo, peso in frecuencia.items()]
```

```
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for par in lo[1:]:
            par[1] = "0" + par[1]
        for par in hi[1:]:
            par[1] = "1" + par[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    return dict(sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p)))

cadena = "ejemplo de huffman"
print(huffman(cadena))
```

Kadane (Submatriz con la suma máxima)

- Encuentra la submatriz o subarray contiguo con la suma máxima dentro de un arreglo numérico.
- Se usa en problemas de optimización financiera, visión por computadora y análisis de series de datos.

Pasos del algoritmo:

1. Inicializar dos variables: max_actual y max_global.
2. Recorrer el array sumando elementos consecutivos.
3. Si la suma acumulada es menor que el elemento actual, reiniciar la suma.
4. Guardar la máxima suma encontrada.

Complejidad: $O(n)$ (lineal).

```
def kadane(arr):
max_actual = max_global = arr[0]
for num in arr[1:]: max_actual = max(num, max_actual + num)
 max_global = max(max_global, max_actual)
```

```
return max_global

arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

print(kadane(arr)) # Salida: 6
```

¿Por qué el algoritmo de Huffman es óptimo para la compresión sin pérdida?

- Reduce el tamaño del archivo sin perder información, ya que asigna códigos más cortos a los caracteres más frecuentes.
- Es óptimo porque produce el prefijo de codificación más corto posible para los datos dados.

¿En qué situaciones prácticas se utiliza el algoritmo de Kadane?

- Análisis financiero: encontrar períodos de mayor ganancia en series de datos bursátiles.
- Procesamiento de imágenes: detectar áreas con valores intensos en matrices de píxeles.
- Biología computacional: análisis de secuencias genéticas en bioinformática