Taller Algoritmos

Parte 1

¿Qué es un algoritmo?

Un **algoritmo** es un conjunto ordenado y finito de pasos o instrucciones que se utilizan para resolver un problema o realizar una tarea específica. Los algoritmos pueden representarse de distintas formas, como pseudocódigo, diagramas de flujo o directamente en un lenguaje de programación.

Características de un buen algoritmo

Un buen algoritmo debe cumplir con las siguientes características:

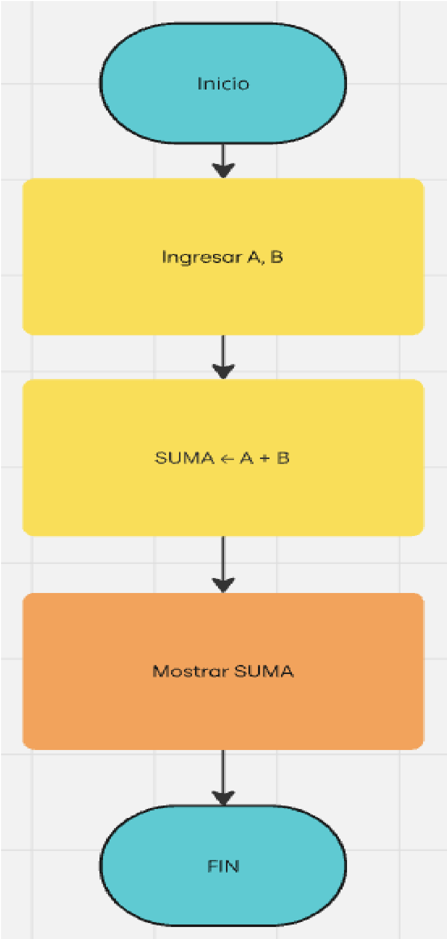
1. **Eficiencia:** Debe resolver el problema utilizando la menor cantidad de recursos posibles, como tiempo de ejecución y memoria.
2. **Corrección:** Debe producir siempre el resultado esperado para cualquier entrada válida.
3. **Claridad:** Debe ser fácil de entender, leer y modificar para facilitar su mantenimiento y mejora.

Importancia de los algoritmos en la programación

Los algoritmos son fundamentales en la programación porque permiten:

- **Optimizar procesos** al diseñar soluciones eficientes.
- **Resolver problemas complejos** dividiéndolos en pasos más simples y manejables.
- **Garantizar la calidad del software** al asegurar que las soluciones sean correctas y predecibles.
- **Mejorar el rendimiento** de los programas al elegir el algoritmo más adecuado según la tarea.

SUMAR DOS NÚMEROS:



PSEUDOCÓDIGO:

Inicio
  Escribir "Ingrese el primer número:"
  Leer A
  Escribir "Ingrese el segundo número:"
  Leer B
  SUMA ← A + B
  Escribir "El resultado de la suma es:", SUMA
Fin

Parte 2

¿Cómo funciona cada algoritmo?

Quicksort

Quicksort es un algoritmo de ordenamiento basado en el paradigma **"divide y vencerás"**. Su funcionamiento se basa en:

1. Elegir un **pivote** de la lista.
2. Dividir la lista en dos sublistas:
   - Elementos menores o iguales al pivote.
   - Elementos mayores al pivote.
3. Aplicar Quicksort recursivamente en cada sublista.
4. Combinar las sublistas y el pivote para obtener la lista ordenada

Mergesort

Mergesort también sigue el paradigma **"divide y vencerás"**, pero su enfoque es diferente:

1. Dividir la lista en dos mitades hasta que cada sublista tenga un solo elemento.
2. Ordenar y **fusionar** (merge) las sublistas de manera ordenada hasta reconstruir la lista completa.

Diferencia clave:

- **Quicksort** es más rápido en la práctica, pero su rendimiento puede degradarse a $O(n^2)$ si el pivote no se elige bien.
- **Mergesort** siempre mantiene un rendimiento estable de $O(n \log n)$, pero usa más memoria debido a la fusión.

Comparación de eficiencia

- **Quicksort** suele ser más rápido en listas grandes porque aprovecha mejor la memoria caché.
- **Mergesort** es más eficiente en listas muy grandes o cuando se necesita **estabilidad** (preservar el orden relativo de elementos iguales).

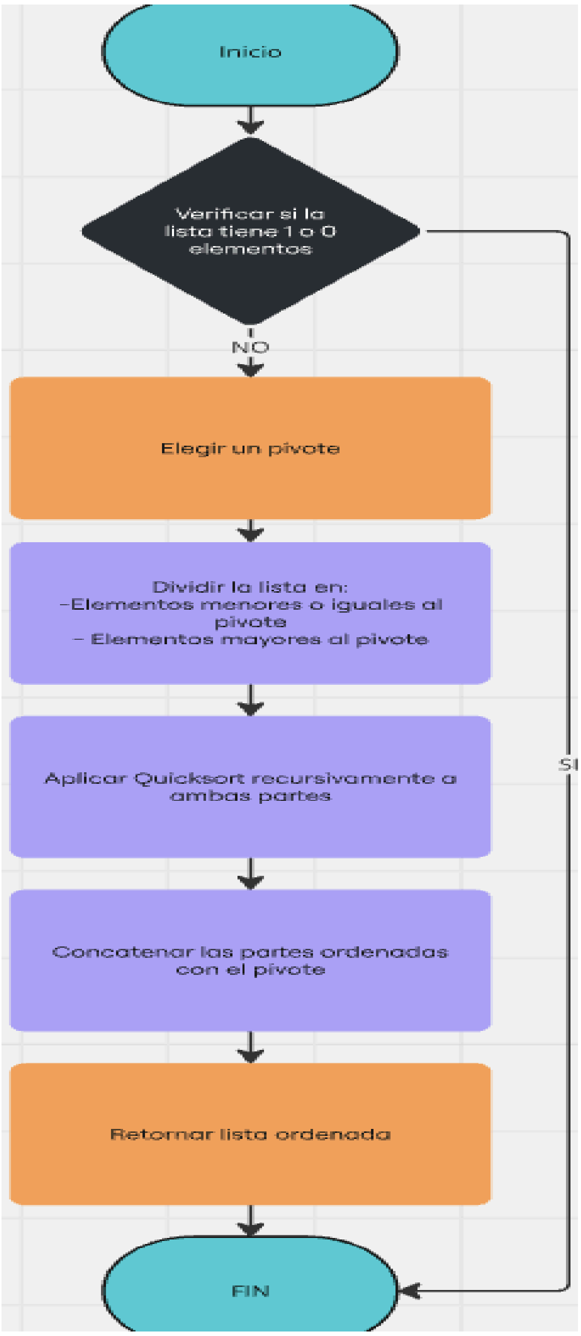
¿Por qué Quicksort es más rápido en promedio pero no es estable?

Quicksort suele ser más rápido porque realiza menos movimientos y particiones en memoria, optimizando el uso de caché. Sin embargo, **no es estable** porque al intercambiar elementos, puede cambiar el orden relativo de los valores iguales.

¿En qué situaciones preferirías usar Mergesort?

- Cuando se requiere **ordenamiento estable** (por ejemplo, ordenar registros de una base de datos sin alterar el orden de aparición).
- Cuando el conjunto de datos es **muy grande** y no cabe en memoria (Mergesort es más eficiente para listas externas en archivos grandes).
- En sistemas que tienen un alto rendimiento de **multiprocesamiento** (Mergesort se puede paralelizar mejor que Quicksort).

Quicksort:



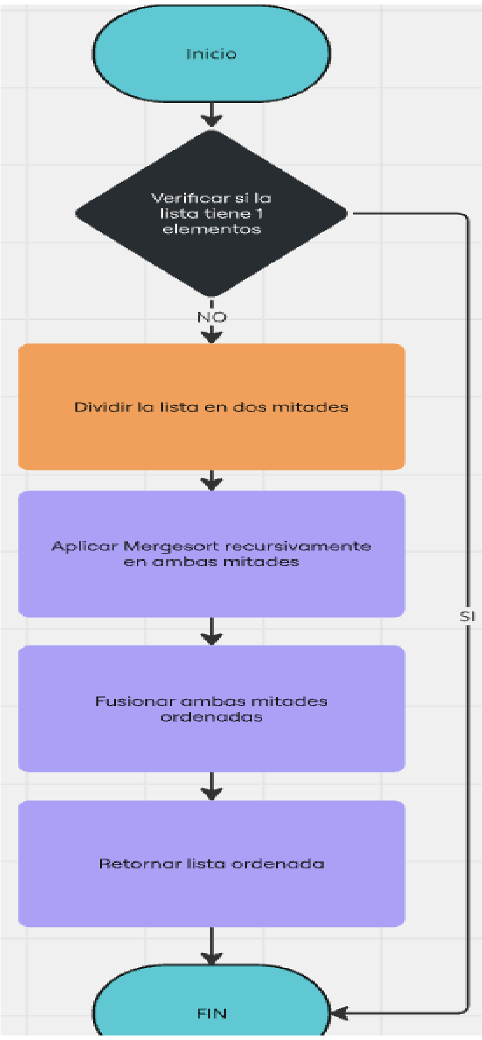
PSEUDOCÓDIGO:

```
Inicio
  Función Quicksort(lista)
    Si tamaño(lista) ≤ 1
      Retornar lista
    Fin Si

    Seleccionar pivote de lista
    Definir lista_menores ← elementos ≤ pivote
    Definir lista_mayores ← elementos > pivote

    Retornar Quicksort(lista_menores) + [pivote] + Quicksort(lista_mayores)
  Fin Función
Fin
```

Mergesort:



PSEUDOCÓDIGO:

```
Inicio
  Función Mergesort(lista)
    Si tamaño(lista) ≤ 1
      Retornar lista
    Fin Si

    Dividir lista en mitad_izquierda y mitad_derecha

    mitad_izquierda ← Mergesort(mitad_izquierda)
    mitad_derecha ← Mergesort(mitad_derecha)

    Retornar Fusionar(mitad_izquierda, mitad_derecha)
  Fin Función

  Función Fusionar(izquierda, derecha)
    Definir lista_ordenada vacía
    Mientras izquierda y derecha no estén vacías
      Si primer elemento de izquierda ≤ primer elemento de derecha
        Agregar primer elemento de izquierda a lista_ordenada
      Sino
        Agregar primer elemento de derecha a lista_ordenada
      Fin Si
    Fin Mientras

    Agregar elementos restantes de izquierda y derecha a lista_ordenada
    Retornar lista_ordenada
  Fin Función
Fin
```

Parte 3

¿Cómo funciona la Búsqueda Binaria?

La **búsqueda binaria** es un algoritmo eficiente que encuentra un elemento en una lista **ordenada** dividiéndola en mitades de forma iterativa. Su funcionamiento es el siguiente:

1. Se toma el **elemento central** de la lista.
2. Se compara con el valor buscado:
   - Si es igual, se encontró el elemento.
   - Si el valor buscado es menor, se descarta la mitad derecha y se repite el proceso con la mitad izquierda.
   - Si el valor buscado es mayor, se descarta la mitad izquierda y se repite el proceso con la mitad derecha.
3. Se repite este proceso hasta encontrar el elemento o hasta que la lista se reduzca a cero elementos.

¿Por qué requiere una lista ordenada?

Porque el algoritmo descarta la mitad de los elementos en cada paso. Si la lista no estuviera ordenada, no podríamos garantizar en qué mitad está el valor buscado.

- **La Búsqueda Lineal** revisa **todos** los elementos uno por uno, por lo que su rendimiento es **lento** en listas grandes.
- **Búsqueda Binaria** reduce drásticamente el número de comparaciones, siendo mucho más rápida en listas ordenadas.

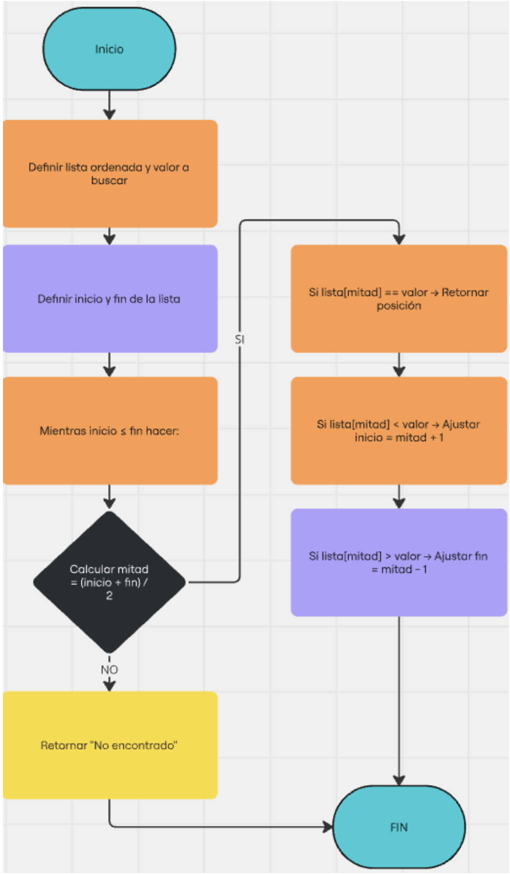
¿En qué casos es preferible usar Búsqueda Lineal en lugar de Búsqueda Binaria?

- Cuando la lista **no está ordenada** y no se quiere gastar tiempo en ordenarla.
- Cuando la lista es **pequeña**, ya que la diferencia de rendimiento entre ambos algoritmos es mínima en tamaños pequeños.
- Cuando se buscan **múltiples coincidencias** (por ejemplo, encontrar todos los elementos iguales a un valor en una lista).

¿Qué pasa si intentas usar Búsqueda Binaria en una lista no ordenada?

- **El resultado será incorrecto o impredecible** porque la búsqueda binaria asume que la lista está ordenada.
- Podría descartar incorrectamente partes de la lista y **no encontrar un elemento que sí existe**.

Búsqueda Binaria:



PSEUDOCÓDIGO:

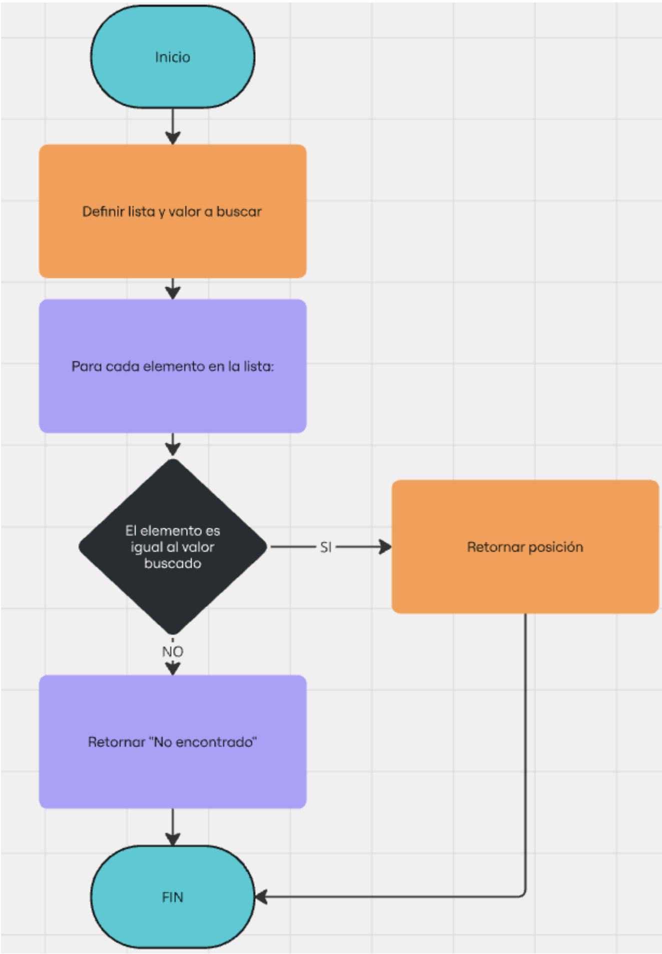
```
Inicio
  Función BusquedaBinaria(lista, valor)
    Definir inicio ← 0
    Definir fin ← tamaño(lista) - 1

    Mientras inicio ≤ fin hacer
      mitad ← (inicio + fin) / 2

      Si lista[mitad] == valor
        Retornar mitad  // Posición encontrada
      Sino Si lista[mitad] < valor
        inicio ← mitad + 1
      Sino
        fin ← mitad - 1
      Fin Si
    Fin Mientras

    Retornar "No encontrado"
  Fin Función
Fin
```

Búsqueda Lineal:



PSEUDOCÓDIGO:

```
Inicio
  Función BusquedaLineal(lista, valor)
    Para i desde 0 hasta tamaño(lista) - 1 hacer
      Si lista[i] == valor
        Retornar i  // Posición encontrada
      Fin Si
    Fin Para
    Retornar "No encontrado"
  Fin Función
Fin
```

Parte 4

¿Cómo funciona cada algoritmo?

1. BFS (Búsqueda en Anchura)

- Explora un grafo nivel por nivel, es decir, primero visita todos los nodos vecinos antes de pasar a los de niveles más profundos.
- Se utiliza una **cola (FIFO)** para gestionar los nodos pendientes de visitar.
- Se usa cuando se quiere encontrar la **ruta más corta en términos de número de aristas** en un grafo no ponderado.

Complejidad Temporal:

- **O(V + E)**, donde **V** es el número de vértices y **E** el número de aristas.

Casos de Uso:

- Encontrar la ruta más corta en un grafo no ponderado.
- Resolución de laberintos.
- Redes sociales (amigos en común).

2. DFS (Búsqueda en Profundidad)

- Explora un camino completo antes de retroceder y probar otro.
- Se utiliza una **pila (LIFO)** o recursión para gestionar los nodos pendientes.
- Es útil para recorrer todo el grafo o encontrar caminos en grafos grandes.

Complejidad Temporal:

- **O(V + E)**.

Casos de Uso:

- Detectar ciclos en un grafo.
- Resolver problemas como el "laberinto".
- Comprobar la conectividad de un grafo.

3. Dijkstra (Camino más corto en un grafo ponderado)

- Encuentra la **ruta más corta** desde un nodo fuente a todos los demás en un grafo ponderado.

- Utiliza una **cola de prioridad** para siempre expandir el nodo con la menor distancia acumulada.

Complejidad Temporal:

- **O((V + E) log V)** usando una cola de prioridad con un heap.

Casos de Uso:

- Sistemas de navegación (Google Maps, GPS).
- Redes de telecomunicaciones.
- Algoritmos de optimización en inteligencia artificial.

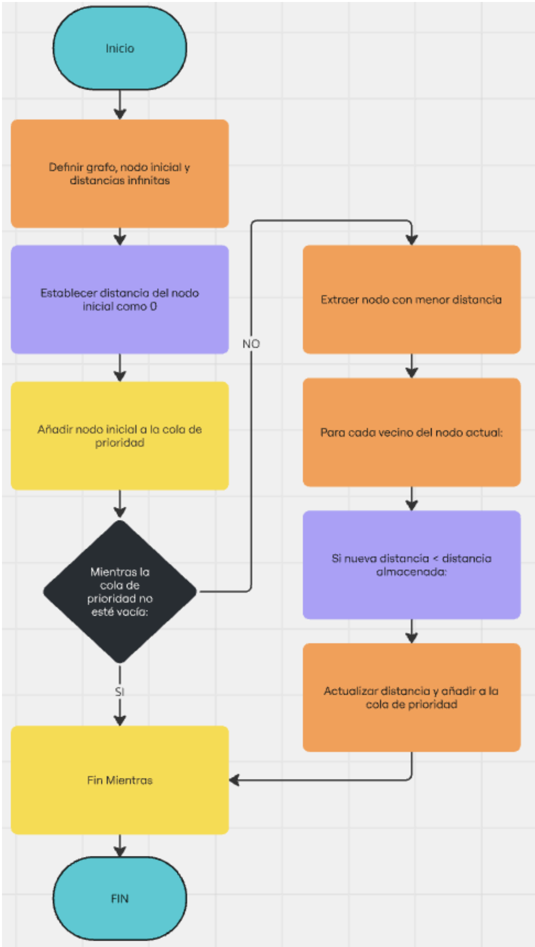
¿Por qué Dijkstra no funciona con pesos negativos?

- Dijkstra asume que **una vez que marca la distancia mínima de un nodo, esta no cambiará**.
- Si hay pesos negativos, un camino que parecía más largo al inicio puede terminar siendo más corto después, lo que rompería la lógica del algoritmo.
- En estos casos, se usa **Bellman-Ford**, que maneja pesos negativos correctamente.

¿En qué situaciones preferirías usar DFS en lugar de BFS?

- Cuando queremos explorar **todo el grafo** (ej., verificar conectividad o detectar ciclos).
- En problemas donde necesitamos **retroceder** y probar otros caminos (ej., resolver laberintos).
- En grafos **muy grandes** donde BFS requeriría demasiada memoria.

Dijkstra:

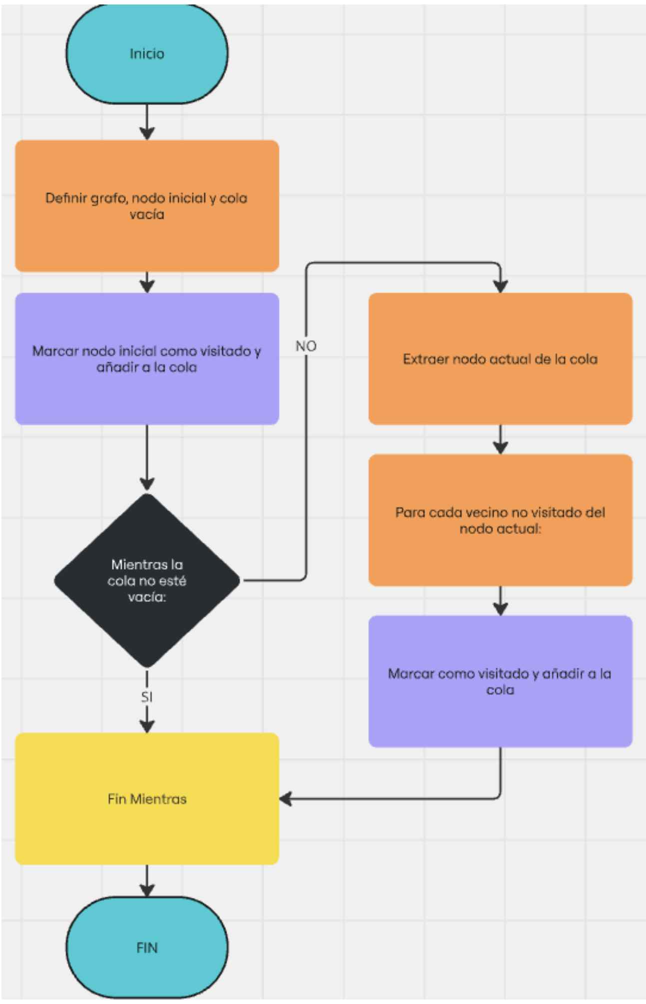


PSEUDOCÓDIGO:

```
Inicio
  Función Dijkstra(grafo, nodo_inicial)
    Definir distancias con infinito
    Establecer distancia[nodo_inicial] ← 0
    Crear cola de prioridad y añadir nodo_inicial

    Mientras la cola no esté vacía hacer
      nodo_actual ← Extraer el nodo con menor distancia
      Para cada vecino en grafo[nodo_actual] hacer
        nueva_distancia ← distancia[nodo_actual] + peso_arista
        Si nueva_distancia < distancia[vecino] entonces
          Actualizar distancia[vecino] ← nueva_distancia
          Añadir vecino a la cola de prioridad
        Fin Si
      Fin Para
    Fin Mientras
  Fin Función
Fin
```

BFS (Búsqueda en Anchura):



PSEUDOCODIGO:

```
Inicio
  Función BFS(grafo, nodo_inicial)
    Crear cola vacía
    Marcar nodo_inicial como visitado y encolar

    Mientras la cola no esté vacía hacer
      nodo_actual ← Desencolar
      Para cada vecino en grafo[nodo_actual] hacer
        Si vecino no ha sido visitado entonces
          Marcar como visitado y encolar
        Fin Si
      Fin Para
    Fin Mientras
  Fin Función
Fin
```

DFS (Búsqueda en Profundidad):

PSEUDOCÓDIGO:

```
Inicio
  Función DFS(grafo, nodo_inicial)
    Crear pila vacía
    Marcar nodo_inicial como visitado y apilar

    Mientras la pila no esté vacía hacer
      nodo_actual ← Desapilar
      Para cada vecino en grafo[nodo_actual] hacer
        Si vecino no ha sido visitado entonces
          Marcar como visitado y apilar
        Fin Si
      Fin Para
    Fin Mientras
  Fin Función
Fin
```

Parte 5

¿Cómo funciona el algoritmo de Huffman?

El algoritmo de **Huffman** se utiliza para la compresión de datos sin pérdida. Funciona asignando **códigos binarios más cortos a los caracteres más frecuentes** y códigos más largos a los menos frecuentes.

¿Qué problema resuelve el algoritmo de Kadane?

El **algoritmo de Kadane** se utiliza para encontrar la **subsecuencia contigua con la suma máxima** dentro de un arreglo de números (positivos y negativos).

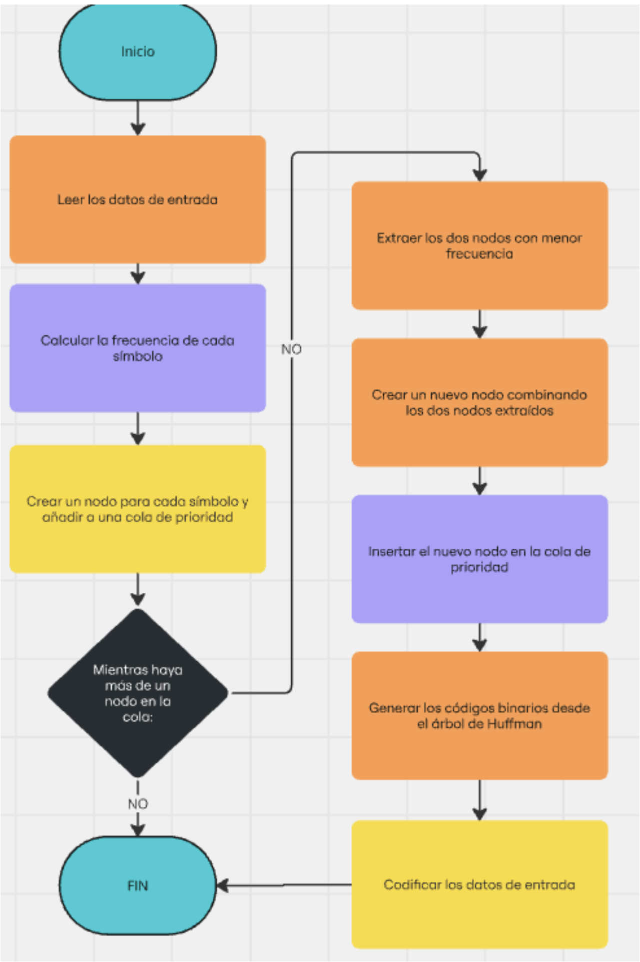
¿Por qué el algoritmo de Huffman es óptimo para la compresión sin pérdida?

- No introduce pérdida de información, ya que asigna **códigos más cortos a los caracteres más frecuentes**.
- Siempre genera una codificación **prefija**, evitando ambigüedades.
- Se usa en formatos como **ZIP, JPEG sin pérdida y MP3**.

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

- **Análisis de señales:** Para encontrar la **sección con mayor amplitud** en una serie de datos.
- **Optimización de inversiones:** Determinar el **mejor periodo para comprar y vender** en una serie de precios.
- **Procesamiento de imágenes:** Encontrar **la región más brillante u oscura** en una imagen (usando la versión en 2D).

Huffman:



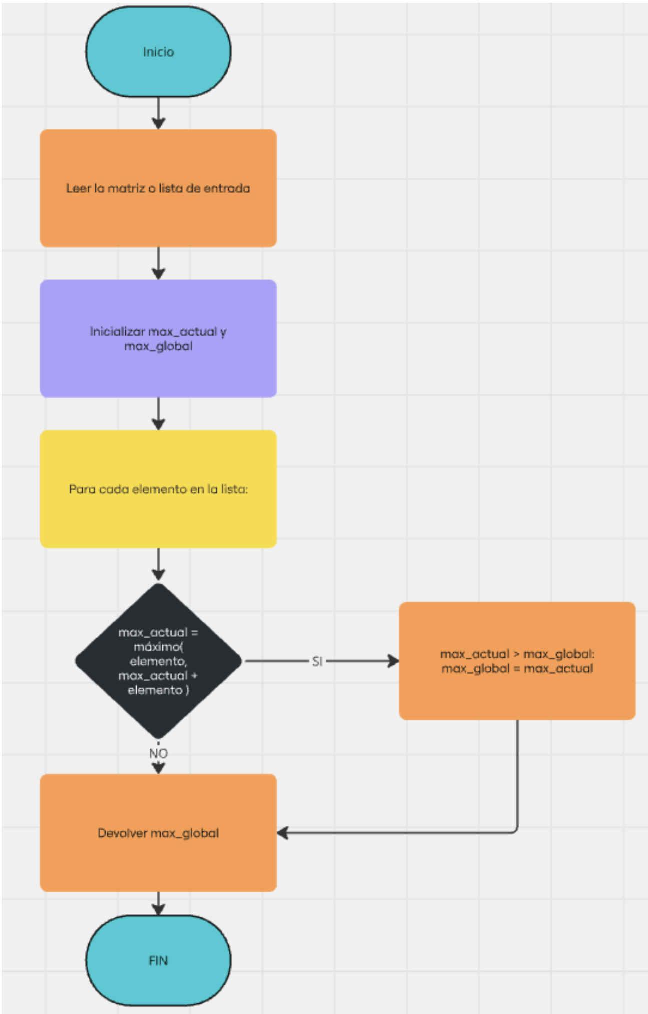
PSEUDOCÓDIGO:

```
Inicio
  Función Huffman(datos)
    Crear diccionario de frecuencias
    Crear una cola de prioridad con nodos de frecuencia

    Mientras la cola tenga más de un nodo hacer
      Extraer los dos nodos con menor frecuencia
      Crear un nuevo nodo combinando los dos nodos extraídos
      Insertar el nuevo nodo en la cola
    Fin Mientras

    Generar los códigos binarios a partir del árbol de Huffman
    Devolver los códigos y la estructura del árbol
  Fin Función
Fin
```

Kadane:



PSEUDOCÓDIGO:

```
Inicio
  Función Kadane(lista)
    max_actual ← lista[0]
    max_global ← lista[0]

    Para i desde 1 hasta longitud(lista) hacer
      max_actual ← máximo(lista[i], max_actual + lista[i])
      Si max_actual > max_global entonces
        max_global ← max_actual
      Fin Si
    Fin Para

    Devolver max_global
  Fin Función
Fin
```