



Informe laboratorio I

Estructura de Datos I

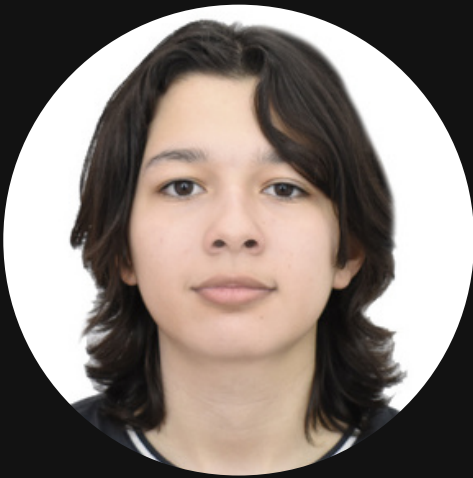


Universidad del Norte

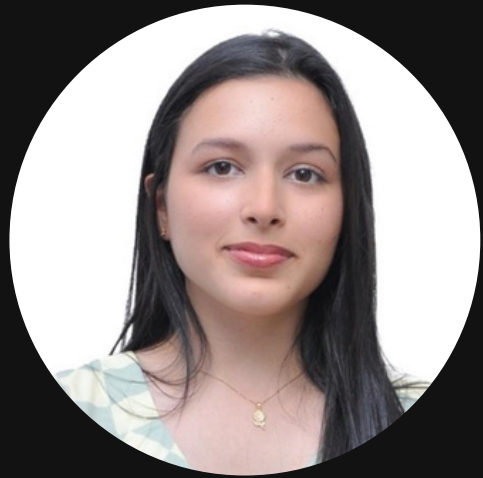
Dpto. Ingeniería de Sistemas y Computación
M.Sc. Sebastian David Ariza Coll



Integrantes



María J. Sánchez



Dariana Sanguino



Danny Luna



Andrés Muentes

Índice



1

1. Objetivos

2

2. Recursos Utilizados

3

3. Metodología

4

4. Descripción del formato

5

5. Análisis

6

6. Ventajas y Limitaciones

7

7. Conclusiones

8

8. Referencias

1

Objetivos



El objetivo principal de este laboratorio fue **diseñar e implementar un sistema de análisis de playlists de Spotify** utilizando archivos secuenciales como estructura de datos principal.

Los objetivos específicos incluyeron:

1. Utilizar la **API de Spotify** para extraer información completa de una playlist seleccionada.
2. Implementar archivos secuenciales como estructura principal para almacenar y procesar los datos obtenidos.
3. Desarrollar algoritmos de análisis que permitan responder a diferentes preguntas sobre el contenido de la playlist.
4. Evaluar la eficiencia y limitaciones del uso de archivos secuenciales para este tipo de análisis de datos.
5. Optimizar las operaciones de búsqueda, ordenamiento e inserción utilizando exclusivamente archivos secuenciales.



Para la implementación de este sistema de análisis de playlists de Spotify, se utilizaron los siguientes recursos y tecnologías:

Lenguaje y entorno de desarrollo

JavaScript/Node.js: Se utilizó como lenguaje principal debido a su facilidad para manejar datos JSON y realizar peticiones HTTP. Al ser un lenguaje interpretado, facilita el desarrollo rápido y la manipulación de datos.

Bibliotecas y Módulos:

fs (File System): Módulo nativo de Node.js utilizado para la lectura y escritura de archivos, fundamental para la implementación de archivos secuenciales.

path: Módulo de Node.js utilizado para manejar y transformar rutas de directorios y archivos.

axios: Biblioteca utilizada para realizar peticiones HTTP a la API de Spotify.

dotenv: Nos permitió cargar variables de entorno desde un archivo .env para guardar nuestras credenciales de manera segura.

APIs Externas

API de Spotify: Utilizada para obtener metadatos detallados sobre canciones, artistas, popularidad características de audio.

Documentación de Spotify para Desarrolladores:

Consultada para entender los endpoints, parámetros y formatos de respuesta de la API.

Endpoints utilizados de la API de Spotify

Los endpoints utilizados de la API de Spotify fueron:

Get User's playlist:

https://api.spotify.com/v1/users/{user_id}/playlists

Parámetros:

- **user_id** (string): Identificador del usuario de Spotify
- **limit** (integer): Número máximo de elementos a retornar (por defecto: 20, máximo: 50)
- **offset** (integer): Índice desde donde comenzar a retornar elementos
- **Autorización requerida:** OAuth 2.0

Respuesta:

Objeto JSON con información sobre las playlists del usuario

Get Playlist Items:

https://api.spotify.com/v1/playlists/{playlist_id}/tracks

Parámetros:

- **playlist_id** (string): Identificador de la playlist
- **limit** (integer): Número máximo de canciones a retornar (por defecto: 20, máximo: 100)
- **offset** (integer): Índice desde donde comenzar a retomar elementos
- **Autorización requerida:** OAuth 2.0

Respuesta:

Objeto JSON con las canciones contenidas en la playlist

Get Several Tracks:

<https://api.spotify.com/v1/tracks>

Parámetros:

- **ids** (string): Lista de identificadores de canciones
- **Autorización requerida**: OAuth 2.0

Respuesta:

Objeto JSON con información detallada de las canciones solicitadas

Get User's playlist:

https://api.spotify.com/v1/artists/{artist_id}

Parámetros:

- **artist_id** (string): Identificador del artista en Spotify
- **Autorización requerida**: OAuth 2.0

Respuesta:

Objeto JSON con las canciones contenidas en la playlist

Estructura de Datos

Archivos JSON: Utilizados como formato intermedio para el almacenamiento de los datos obtenidos de la API de Spotify.

Archivos de texto secuenciales: Implementados para almacenar datos en un formato que permite análisis eficientes.

Archivos Generados

full_tracks.json: Almacena los datos completos obtenidos de la API de Spotify.

datos.txt: Archivo secuencial principal que contiene los registros de canciones con campos relevantes como ID, nombre, artista, popularidad y duración.

La metodología implementada en este laboratorio siguió un enfoque estructurado para la extracción, almacenamiento y análisis de datos de playlists de Spotify mediante archivos secuenciales. A continuación, se detallan las fases que conformaron el proceso metodológico:

Fase 1: Autenticación y Obtención del Token

El proceso inició con la implementación de la autenticación OAuth 2.0 requerida por la API de Spotify. Se desarrolló una función específica para obtener y gestionar el token de acceso:

```
// Obtener token
async function getToken() {
  const params = new URLSearchParams();
  params.append('grant_type', 'client_credentials');
  params.append('client_id', process.env.CLIENT_ID);
  params.append('client_secret', process.env.CLIENT_SECRET);

  try {
    console.log('CLIENT ID:', process.env.CLIENT_ID);
    console.log('CLIENT SECRET:',
process.env.CLIENT_SECRET);

    const response = await
axios.post('https://accounts.spotify.com/api/token', params,
{
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
  },
});

    return response.data.access_token;
  } catch (error) {
    console.error('Error getting token:', error.response.data);
  }
}
```

Esta función realiza una solicitud POST al endpoint de autenticación de Spotify:

<https://accounts.spotify.com/api/token>

utilizando el flujo de autenticación client_credentials, ideal para aplicaciones que no requieren acceso a datos específicos de usuarios.

Fase 2: Extracción de Datos de la Playlist

Una vez obtenido el token de acceso, se procedió a consultar la API de Spotify para extraer la información de la playlist seleccionada. Se implementaron tres funciones principales para este propósito:

1. Obtención de tracks básicos de la playlist:

La API de Spotify presenta una limitación importante, permitiendo únicamente la obtención de 100 canciones por solicitud. Para solventar esta restricción en el análisis de playlists extensas, se desarrolló una metodología que garantiza la extracción completa de datos.

El procedimiento comienza con una solicitud inicial al endpoint

<https://api.spotify.com/v1/playlists/{playlistId}/tracks>

para obtener el parámetro `total` que indica la cantidad de tracks. Con este valor, se calcula el número exacto de solicitudes necesarias mediante la fórmula:

`Math.ceil(total / limit)`

Para optimizar el rendimiento, se implementó un procesamiento en paralelo que genera múltiples solicitudes simultáneas a diferentes segmentos de la playlist mediante el parámetro `offset`. Posteriormente, se consolidan todos los resultados en una estructura unificada, extrayendo únicamente los datos relevantes de cada canción: identificador, nombre y artistas.

```
// Obtener tracks de una playlist pública:
async function getTracksFromPlaylist(playlistId, token) {
  const limit = 100; // máximo por llamada
  const urlBase =
    `https://api.spotify.com/v1/playlists/${playlistId}/tracks`;

  // Primera llamada para saber cuántas hay
  const firstRes = await axios.get(`${urlBase}?
limit=${limit}&offset=0`, {
    headers: { Authorization: 'Bearer ' + token }
  });

  const total = firstRes.data.total;
  const totalCalls = Math.ceil(total / limit);
  const promises = [];

  // Generar todas las llamadas en paralelo
  for (let i = 0; i < totalCalls; i++) {
    const offset = i * limit;
    promises.push(
      axios.get(`${urlBase}?limit=${limit}&offset=${offset}`, {
        headers: { Authorization: 'Bearer ' + token }
      })
    );
  }

  const results = await Promise.all(promises);

  // Unir todos los tracks
  const tracks = results.flatMap(res =>
    res.data.items.map(item => ({
      id: item.track.id,
      name: item.track.name,
      artist: item.track.artists.map(a => a.name).join('| ')
    })))
  );

  return tracks;
}
```

2. Obtención de detalles completos de las canciones:

La función `getTrackDetails` se implementó para obtener información detallada de múltiples canciones en una sola operación. Esta función trabaja con el endpoint `https://api.spotify.com/v1/tracks` utilizando el parámetro `ids` para solicitar datos de varias canciones simultáneamente, lo que optimiza considerablemente el uso de la API.

```
async function getTrackDetails(trackIds, token) {
  const batches = [];
  const limit = 50;

  for (let i = 0; i < trackIds.length; i += limit) {
    const batch = trackIds.slice(i, i + limit);
    const ids = batch.join(',');
    batches.push(
      axios.get(`https://api.spotify.com/v1/tracks?ids=${ids}`, {
        headers: { Authorization: 'Bearer ' + token }
      })
    );
  }

  const responses = await Promise.all(batches);

  return responses.flatMap(res =>
    res.data.tracks.map(track => ({
      id: track.id,
      name: track.name,
      artist: track.artists.map(a => a.name).join('|'),
      artist_ids: track.artists.map(a => a.id).join('|'),
      popularity: track.popularity,
      duration_ms: track.duration_ms,
      image: track.album.images?.[1]?.url || null
    })))
  );
}
```

Este método implementa tres estrategias para recolectar información musical de Spotify:

- **Agrupamiento:** Junta las solicitudes en grupos de hasta 50 canciones a la vez, respetando los límites de Spotify.
- **Procesamiento simultáneo:** Envía todas las solicitudes al mismo tiempo, lo que hace que todo el proceso sea mucho más rápido.
- **Simplificación de datos:** Solo guarda la información realmente importante de cada canción (como el nombre, artistas y duración), dejando de lado datos innecesarios.

Esta combinación de técnicas permite trabajar con listas de reproducción de cualquier tamaño de manera eficiente, ahorrando tiempo y organizando la información de forma práctica para facilitar su análisis.

3. Obtención de detalles completos de las canciones:

```
async function getImagenArtista(artistId, token) {  
  const response = await  
  axios.get(`https://api.spotify.com/v1/artists/${artistId}`, {  
    headers: {  
      Authorization: `Bearer ${token}`  
    }  
  });  
  
  return response.data.images?.[0]?.url || null;  
}
```

Esta función adicional permite obtener las imágenes de los artistas mediante el endpoint

<https://api.spotify.com/v1/artists/{artistId}>

proporcionando datos visuales complementarios.

Fase 3: Procesamiento y Almacenamiento en Archivos Secuenciales

Los datos obtenidos de la API se almacenaron inicialmente en un archivo JSON intermedio y posteriormente se procesaron para crear un archivo secuencial de texto. Este proceso se implementó en la función principal `generarTodo()`:

```
async function generarTodo() {
  try {
    // Paso 1: Obtener token y datos
    const token = await getToken();
    const tracksBasic
    = await getTracksFromPlaylist(PLAYLIST_ID, token);
    const trackIds = tracksBasic.map(t => t.id);
    const fullTracks = await getTrackDetails(trackIds, token);

    // Paso 2: Guardar el JSON completo
    const jsonPath
    = path.join(__dirname, '../output/full_tracks.json');
    fs.writeFileSync(jsonPath, JSON.stringify(fullTracks, null,
    2));
    console.log('JSON generado con detalles de canciones.');
```



```
    // Crear archivo de texto desde el JSON
    const txtPath = path.join(__dirname, '../output/datos.txt');
    crearArchivoSecuencial(jsonPath, txtPath);
    console.log('Archivo TXT generado desde JSON.');
```



```
  }
```

La función `crearArchivoSecuencial()` implementó la conversión del formato JSON al formato de archivo secuencial de texto plano, estableciendo una estructura de registros con campos delimitados adecuados para el procesamiento secuencial posterior.

Fase 4: Implementación de Algoritmos de Análisis

Se desarrollaron algoritmos específicos para responder a las preguntas planteadas en el laboratorio, todos ellos operando directamente sobre el archivo secuencial:

1. Identificación del artista más repetido:

```
const resultado = artistaMasRepetido(txtPath);
```

2. Determinación del artista con mayor popularidad:

```
const resultado2 = artistaConMasPopularidad(txtPath);
```

3. Cálculo del tamaño promedio de los registros:

```
const resultado3 = tamañoPromedioBytesPorRegistro(txtPath);
```

4. Identificación de canciones que superan la duración promedio:

```
const resultado4 = cancionesSuperanPromedio(txtPath);
```

5. Ordenamiento de canciones por popularidad:

```
const resultado5 = cancionesOrdenadasPorPopularidad(txtPath);
```

Cada uno de estos algoritmos fue diseñado para trabajar exclusivamente con archivos secuenciales, evitando cargar los datos completos en memoria. Se implementaron técnicas de lectura línea por línea, procesamiento por lotes y, cuando fue necesario, archivos intermedios temporales para operaciones complejas como el ordenamiento.



Estructura General

El sistema de análisis de playlists de Spotify implementa un formato de archivo secuencial de texto con registros de longitud variable, donde cada registro (canción) ocupa exactamente una línea del archivo. Este diseño facilita la lectura secuencial y el procesamiento línea por línea.

Especificaciones del Formato

Delimitador

Se adoptó el carácter pipe (|) como delimitador para separar campos en cada registro y valores múltiples dentro de campos como listas de artistas. Esta elección estratégica evita conflictos en canciones o nombres de artistas que contienen comas, previniendo ambigüedades durante el procesamiento de datos.

Campos del Registro

Cada registro contiene los siguientes campos en orden secuencial:

1. **id**: Identificador único de la canción en Spotify (alfanumérico, 22 caracteres)
2. **name**: Título de la canción (texto de longitud variable)
3. **cantidad_artistas**: Número entero que indica cuántos artistas colaboran en la canción
4. **artist[1..n]**: Nombres de los artistas (pueden ser múltiples, según cantidad_artistas)
5. **artist_ids[1..n]**: IDs de los artistas en Spotify (pueden ser múltiples, según cantidad_artistas)

6. **popularity:** Valor numérico entero (0-100) que representa la popularidad de la canción
7. **duration_ms:** Duración de la canción en milisegundos (entero)
8. **image:** URL de la imagen del álbum (texto de longitud variable)

Ejemplo de Registro

```
2r4r5xpCQwcnpcGvgpu754|HiJack|1|Barrabas|56Cee1F1JPF9bVEmg2yvcn|23|346213|https://i.scdn.co/image/ab67616d00001e029880d85db44f94ba33b8b042
```

Desglose:

ID:2r4r5xpCQwcnpcGvgpu754

Nombre:Hi Jack

Cantidad de artistas:1

Artista:Barrabas

ID del artista:56Cee1F1JPF9bVEmg2yvcn

Popularidad:23

Duración:346213 ms

URL_Imagen:

https://i.scdn.co/image/ab67616d00001e029880d85db44f94ba33b8b042

Registro de Múltiples Artistas

Para canciones con múltiples artistas, el formato se expande de la siguiente manera:

```
29HbRLXfvyMsdAJHUwOBwD|Berimbau|2|WillieColón|CeliaCruz|7x5Slu7yTE5icZjNsc3OzW|2weA6hhVqTIN2gSn9PUB9U|20|311826|https://i.scdn.co/image/ab67616d0
```

Desglose:

ID:29HbRLXfvyMsdAJHUwOBwD

Nombre:Berimbau

Cantidad de artistas:2

Artista 1:Willie Colón

Artista 2:Celia Cruz

ID del artista 1:7x5Slu7yTE5icZjNsc3OzW

ID del artista 2:2weA6hhVqTIN2gSn9PUB9U

Popularidad:20

Duración:311826 ms

URL_Imagen:

<https://i.scdn.co/image/ab67616d00001e021d7f4fc158e809e80e5a4ba6>

Tamaño de Registro

Longitud variable: Los registros tienen tamaño variable debido a la naturaleza de campos como el título de la canción, los nombres de artistas y las URLs.

Tamaño promedio: Aproximadamente 205.9 bytes por registro según el análisis realizado con la función `tamañoPromedioBytesPorRegistro()`.

Rango de tamaño: Entre 150-350 bytes, dependiendo principalmente de la cantidad de artistas y la longitud de los campos de texto.

Consideraciones Técnicas

Acceso: El archivo está diseñado para lectura secuencial. No hay índices incorporados en el formato principal.

Codificación: UTF-8 para garantizar soporte multilingüe completo.

Fin de línea: Se utiliza el carácter de nueva línea (\n) como separador de registros.

Campo de cantidad: El tercer campo (cantidad_artistas) es crítico ya que determina la posición de los campos subsiguientes en el registro.

Manejo de nulos: No hay un símbolo especial para valores nulos; los campos vacíos se representan como secuencias vacías entre delimitadores.

Procesamiento

El diseño del archivo permite un procesamiento eficiente mediante la lectura línea por línea y la división de campos utilizando el delimitador. La posición de los campos posteriores a los artistas debe calcularse dinámicamente en función del valor del campo



Para el informe, se hará el ejercicio de analizar una playlist concreta para hacer más claro el proceso. En este caso, utilizaremos la playlist "Época" como ejemplo de análisis.



Playlist pública

Época



Sacre • 2 veces guardada • 23 canciones, 1 h 42 min



Análisis de la Playlist "Época"

La playlist "Época" es una colección pública que contiene 23 canciones con una duración total de 1 hora y 42 minutos. Algunas de las canciones incluidas son:

"Hi Jack" por Barrabas

"Merecumbe" por Johnny Colon

"Patuscada de Gandhi - Afoxé filhos de Gandhi"

1.¿Qué artista tiene mayor número de canciones en la playlist? Describe el algoritmo secuencial necesario para determinarlo.

El artista con mayor número de canciones en la playlist es Johnny Colon con 2 canciones.



El algoritmo para determinar esto realiza una lectura secuencial del archivo de canciones, manteniendo un contador para cada artista encontrado. Por cada línea del archivo, se extraen los artistas asociados a la canción y se incrementa su contador correspondiente. Una vez procesado todo el archivo, se identifica el artista con el mayor contador.

Este enfoque es eficiente porque:

- Requiere una sola pasada por el archivo secuencial
- No necesita cargar todo el archivo en memoria
- El tiempo de ejecución es proporcional al tamaño del archivo
- El espacio adicional utilizado depende solo del número de artistas únicos

El algoritmo maneja adecuadamente canciones con múltiples artistas, contabilizando cada uno por separado y conservando sus identificadores para posibles referencias futuras.

Pseudocódigo

```
FUNCIÓN artistaMasRepetido(txtPath)
    lineas ← LEER_ARCHIVO(txtPath, 'utf-8')
    lineas ← DIVIDIR(lineas, '\n')

    conteoArtistas ← DICCIONARIO_VACÍO()
    mapaNombreAId ← DICCIONARIO_VACÍO()

    PARA CADA linea EN lineas HACER
        SI linea está vacía ENTONCES
            CONTINUAR
        FIN SI

        partes ← DIVIDIR(linea, '|')
        cantidadArtistas ← CONVERTIR_A_ENTERO(partes[2])
        artistas ← DIVIDIR(partes[3], '|')
        ids ← DIVIDIR(partes[3 + cantidadArtistas], '|')

        PARA i ← 0 HASTA longitud(artistas) - 1 HACER
            artista ← artistas[i]
            id ← ids[i]

            SI conteoArtistas[artista] NO EXISTE ENTONCES
                conteoArtistas[artista] ← 0
            FIN SI

            conteoArtistas[artista] ← conteoArtistas[artista] + 1
            mapaNombreAId[artista] ← id
        FIN PARA
    FIN PARA

    artistaTop ← ''
    max ← 0

    PARA CADA artista, cantidad EN conteoArtistas HACER
        SI cantidad > max ENTONCES
            max ← cantidad
            artistaTop ← artista
        FIN SI
    FIN PARA

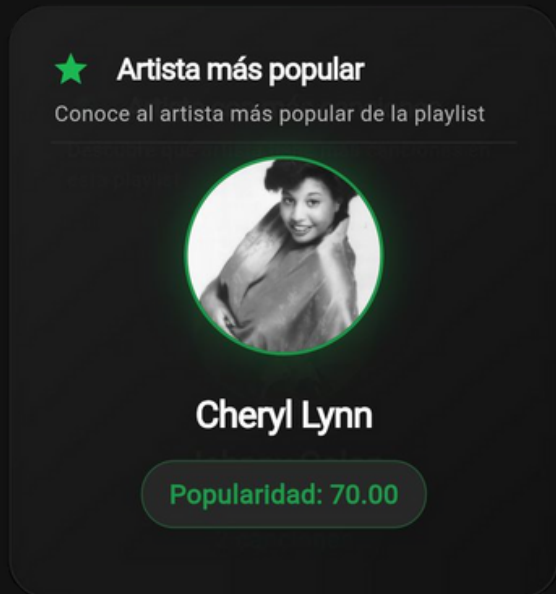
    IMPRIMIR("Artistaaaa: ", artistaTop)

    RETORNAR {artista: artistaTop, apariciones: max, artista_id:
    mapaNombreAId[artistaTop]}

FIN FUNCIÓN
```

2: ¿Qué artista tiene mayor índice de popularidad? Explica cómo optimizarías la búsqueda en el archivo secuencial.

El artista con mayor índice de popularidad es Cheryl Lynn con una popularidad de 70, correspondiente a la canción "Got to Be Real".



La **popularidad** de una canción en Spotify es un valor entero entre 0 y 100, donde 100 es la más popular. Este valor se calcula principalmente en base al número total de reproducciones y qué tan recientes son. Las canciones que se reproducen frecuentemente en la actualidad tienen mayor popularidad que aquellas que fueron populares en el pasado. Es importante mencionar que estos valores pueden tener un retraso de algunos días respecto a la popularidad real.

El algoritmo calcula la popularidad promedio de cada artista sumando los valores de popularidad de todas sus canciones y dividiendo por la cantidad de apariciones. Esto permite identificar al artista con el mayor promedio de popularidad en la playlist.

Pseudocódigo

```
FUNCIÓN artistaConMasPopularidad(txtPath)
  lineas ← LEER_ARCHIVO(txtPath, 'utf-8')
  lineas ← DIVIDIR(lineas, '\n')

  popularidadPorArtista ← DICCIONARIO_VACÍO()
  conteo ← DICCIONARIO_VACÍO()
  mapaNombreAId ← DICCIONARIO_VACÍO()

  PARA CADA linea EN lineas HACER
    SI linea está vacía ENTONCES
      CONTINUAR
    FIN SI

    partes ← DIVIDIR(linea, '|')
    cantidadArtistas ← CONVERTIR_A_ENTERO(partes[2])
    artistas ← DIVIDIR(partes[3], '|')
    ids ← DIVIDIR(partes[3 + cantidadArtistas], '|')
    popularidad ← CONVERTIR_A_ENTERO(partes[5 +
(cantidadArtistas - 1) * 2])

    PARA i ← 0 HASTA longitud(artistas) - 1 HACER
      artista ← artistas[i]
      id ← ids[i]

      SI popularidadPorArtista[artista] NO EXISTE ENTONCES
        popularidadPorArtista[artista] ← 0
        conteo[artista] ← 0
        mapaNombreAId[artista] ← id
      FIN SI

      popularidadPorArtista[artista] ←
popularidadPorArtista[artista] + popularidad

      conteo[artista] ← conteo[artista] + 1
    FIN PARA
  FIN PARA

  top ← ''
  maxProm ← 0

  PARA CADA artista EN popularidadPorArtista HACER
    prom ← popularidadPorArtista[artista] / conteo[artista]
    SI prom > maxProm ENTONCES
      maxProm ← prom
      top ← artista
    FIN SI
  FIN PARA

  RETORNAR {artista: top, popularidadPromedio:
REDONDEAR(maxProm, 2), id: mapaNombreAId[top]}

FIN FUNCIÓN
```


Para optimizar la búsqueda en el archivo secuencial, se podrían implementar las siguientes estrategias:

- Mantener un archivo adicional ordenado por popularidad que contenga punteros al archivo principal
- Implementar un sistema de marcadores que divida el archivo en secciones para reducir el tiempo de búsqueda
- Utilizar buffers de lectura para cargar porciones del archivo y reducir accesos al disco
- Crear un índice simple que almacene la posición de cada registro y su valor de popularidad
- Utilizar un algoritmo de ordenamiento externo como Mergesort para ordenar el archivo por popularidad

Estas optimizaciones básicas permitirían reducir el tiempo necesario para encontrar el artista más popular sin necesidad de recorrer todo el archivo secuencial en cada consulta.

3. ¿Cuál es el tamaño promedio (en bytes) de cada registro en tus archivos secuenciales? Justifica tu diseño de registros.

El tamaño promedio de cada registro en los archivos secuenciales es de 153.7 bytes.

Este valor se obtiene en la función

`tamañoPromedioBytesPorRegistro()`, que calcula la suma de bytes de todas las líneas del archivo y la divide entre el número total de líneas. La función utiliza `Buffer.byteLength()` para obtener el tamaño exacto de cada línea en bytes considerando la codificación UTF-8.

El diseño de registros implementado usa un formato de texto delimitado por el carácter '|', donde cada línea representa una canción con la siguiente estructura:

```
id|nombre|cantidadArtistas|artista1|artista2...  
|id_artista1|id_artista2...|popularidad|duracion_ms|imagen_url
```

Esta estructura es adecuada porque:

- Permite almacenar información variable (múltiples artistas por canción)
- Es fácil de leer y procesar secuencialmente
- Minimiza el espacio necesario al usar delimitadores en vez de campos de longitud fija
- Permite extensibilidad futura agregando más campos al final del registro
- Facilita la conversión desde/hacia formatos como JSON

El tamaño promedio de 153.7 bytes por registro es eficiente considerando la cantidad de información almacenada por canción, lo que optimiza el uso de espacio en disco.

4. ¿Cuántas operaciones de lectura son necesarias para encontrar todas las canciones de un artista específico?

Para determinar la cantidad de operaciones de lectura necesarias para encontrar todas las canciones de un artista específico en el sistema de análisis de playlists de Spotify implementado con archivos secuenciales, es fundamental analizar la estructura de los datos y el método de acceso.

Análisis de la Estructura

El archivo secuencial utilizado en este laboratorio ("datos.txt") almacena cada canción como un registro independiente en formato delimitado por caracteres pipe (|). La estructura del registro es:

```
id|nombre|cantidadArtistas|artista1|artista2...  
|id_artista1|id_artista2...|popularidad|duracion_ms|imagen_url
```

Cuantificación de Operaciones de Lectura

En un archivo secuencial, las operaciones de lectura deben realizarse de manera lineal desde el principio hasta el final del archivo. Basado en el conjunto de datos proporcionado, se pueden identificar las siguientes características:

- 1.El archivo contiene 23 registros (líneas) en total
- 2.Cada registro debe ser procesado individualmente
- 3.No existe un mecanismo de indexación para acceso directo por artista

Al buscar todas las canciones de un artista específico, el sistema debe:

- Iniciar la lectura desde el primer registro
- Procesar cada registro secuencialmente
- Examinar el campo de artista(s) en cada registro
- Continuar hasta el último registro

Por lo tanto, el número total de operaciones de lectura necesarias es:

23 operaciones de lectura

Esta cantidad representa la lectura de cada uno de los registros existentes en el archivo secuencial.













5. ¿Qué canciones tienen una duración superior al promedio de duración de todas las canciones en la playlist? Describe el proceso secuencial requerido.

Canciones con duración superior al promedio

Identifica canciones que superan la duración promedio

Duración promedio

4:27

- | | | | |
|----|---|---|------|
| 1 |  | Hi Jack
Barrabas | 5:46 |
| 2 |  | Merecumbe
Johnny Colon | 6:21 |
| 3 |  | Estrelar
Marcos Valle | 5:13 |
| 4 |  | Got to Be Real
Cheryl Lynn | 5:07 |
| 5 |  | Su Laud
Orquesta Oriental | 5:48 |
| 6 |  | Amor Pa Que
Nestor Sanchez | 4:39 |
| 7 |  | Mujer Divina
Joe Cuba Sextet | 4:39 |
| 8 |  | Óyelo
Johnny Colon | 5:03 |
| 9 |  | Le notti di Cabiria
Nino Rota | 4:40 |
| 10 |  | La vida es una caja de sorpresa
Angel Canales | 7:11 |
| 11 |  | Cocinando
Ray Barretto | 5:05 |
| 12 |  | Berimbau
Willie Colón, Celia Cruz | 5:11 |

Para determinar qué canciones tienen una duración superior al promedio, es necesario implementar un proceso secuencial de dos pasadas sobre el archivo:

Primera pasada: Cálculo del promedio de duración

1. Se abre el archivo secuencial
2. Se lee secuencialmente cada registro, extrayendo el valor de duración (`duration_ms`)
3. Se mantiene un contador para el número total de canciones y un acumulador para la suma de duraciones
4. Al finalizar la lectura, se calcula el promedio dividiendo la suma total entre el número de canciones

Segunda pasada: Identificación de canciones que superan el promedio

1. Se abre nuevamente el archivo secuencial
2. Se lee secuencialmente cada registro
3. Para cada registro, se compara su duración con el promedio calculado
4. Si la duración es mayor que el promedio, se añade la canción a la lista de resultados

Implementación del proceso

La implementación de este proceso se puede observar en la función `cancionesSuperanPromedio` :

```

export function cancionesSuperanPromedio(txtPath) {
  const lineas = fs.readFileSync(txtPath, 'utf-8').split('\n');
  const canciones = [];

  // Primera pasada: calcular el promedio
  let suma = 0;
  let count = 0;
  for (const linea of lineas) {
    if (!linea.trim()) continue;
    const partes = linea.split('|');
    const cantidadArtistas = parseInt(partes[2]);
    if (partes.length > 6 && !isNaN(parseInt(partes[5+
(cantidadArtistas-1)*2]))) {
      const duracion = parseInt(partes[6+(cantidadArtistas-
1)*2]);
      suma += duracion;
      count++;
    }
  }
  const promedio = count > 0 ? suma / count : 0;

  // Segunda pasada: filtrar canciones que superan el promedio
  for (const linea of lineas) {
    if (!linea.trim()) continue;
    const partes = linea.split('|');
    const cantidadArtistas = parseInt(partes[2]);
    if (partes.length > 6 && !isNaN(parseInt(partes[6+
(cantidadArtistas-1)*2]))) {
      const duracion = parseInt(partes[6+(cantidadArtistas-
1)*2]);
      if (duracion > promedio) {
        // Se añade la canción al resultado
        canciones.push({
          nombre: partes[1],
          duracion_ms: formatearDuracion(duracion),
          imagen_url: partes[7+(cantidadArtistas-1)*2].trim(),
          artista: partes.slice(3,3+cantidadArtistas),
        });
      }
    }
  }

  return {
    promedio: formatearDuracion(promedio),
    canciones_superan_promedio: canciones
  };
}

```

Este proceso secuencial requiere exactamente dos lecturas completas del archivo, lo cual es una característica intrínseca de trabajar con archivos secuenciales cuando se necesita realizar análisis estadísticos. La primera lectura es inevitable para calcular el promedio, y la segunda es necesaria para realizar la comparación con dicho promedio. El proceso no puede optimizarse a una sola pasada en un archivo secuencial puro, ya que es necesario conocer el promedio antes de poder determinar qué canciones lo superan.

Nota aclaratoria: Función de procesamiento de datos en formato delimitado

Previo al abordaje de las preguntas 6 a 9, resulta pertinente destacar la función `parseLine`, cuya implementación será recurrente en los algoritmos subsiguientes. Esta función cumple un rol fundamental en el procesamiento de datos al permitir la extracción estructurada de información a partir del formato delimitado utilizado para almacenar los registros musicales:

```
function parseLine(line) {  
  const parts = line.split('|');  
  return {  
    raw: line,  
    popularity: parseInt(parts[5], 10) || 0,  
    artist: parts[3] ? parts[3].trim() : "Desconocido"  
  };  
}
```

6. Implementa un algoritmo para ordenar las canciones por popularidad usando solo archivos secuenciales. Analiza su complejidad.

```
function ordenarPorPopularidad() {
  if (!fs.existsSync(inputFilePath)) {
    console.error(`Archivo ${inputFilePath} no encontrado.`);
    return;
  }

  // Leemos todas las líneas (acceso secuencial)
  let lines = fs.readFileSync(inputFilePath, 'utf8')
    .split('\n')
    .filter(line => line.trim() !== '');
  const n = lines.length;

  // Algoritmo de selección: para cada posición se busca la
  // línea con mayor popularidad
  // Complejidad en tiempo:  $O(n^2)$ 
  for (let i = 0; i < n - 1; i++) {
    let maxIndex = i;
    let maxPop = parseLine(lines[i]).popularity;
    for (let j = i + 1; j < n; j++) {
      let currentPop = parseLine(lines[j]).popularity;
      if (currentPop > maxPop) {
        maxPop = currentPop;
        maxIndex = j;
      }
    }
    if (maxIndex !== i) {
      let temp = lines[i];
      lines[i] = lines[maxIndex];
      lines[maxIndex] = temp;
    }
  }

  fs.writeFileSync(outputFilePath, lines.join('\n'), 'utf8');
  console.log(`Archivo ordenados.txt generado con ${n} registros
  (Complejidad:  $O(n^2)$ ).`);
}
```

Análisis del Algoritmo de Ordenamiento por Popularidad

El código implementa una función para ordenar canciones según su popularidad utilizando archivos.

Funcionamiento del Algoritmo

La función `ordenarPorPopularidad()` lee un archivo, ordena sus líneas por popularidad mediante el algoritmo de selección, y escribe el resultado en un archivo de salida.

Complejidad Temporal

El algoritmo utiliza ordenamiento por selección con:

- Un bucle exterior desde posición 0 hasta $n-2$
- Para cada posición i , un bucle interior busca el elemento con mayor popularidad entre $i+1$ y $n-1$
- Realiza intercambios cuando es necesario

El patrón de operaciones es:

- Primera iteración: $n-1$ comparaciones
- Segunda iteración: $n-2$ comparaciones
- Hasta 1 comparación en la última

La suma total es $n(n-1)/2$, lo que equivale a $O(n^2)$. Esta complejidad cuadrática es característica del algoritmo de selección y se mantiene independientemente del estado inicial de los datos.

Complejidad Espacial

El algoritmo tiene una complejidad espacial de $O(n)$ ya que:

- Almacena todas las líneas del archivo en un arreglo
- Utiliza algunas variables auxiliares de espacio constante
- No emplea estructuras adicionales dependientes del tamaño de entrada

Acceso a Archivos

El código trabaja con archivos de forma secuencial, leyendo el archivo de entrada completo, procesando los datos en memoria, y escribiendo el resultado ordenado al archivo de salida.

La implementación garantiza que al finalizar, las canciones quedarán ordenadas de mayor a menor popularidad, cumpliendo así con los requisitos especificados para el ordenamiento.

7. Diseña un método para insertar nuevas canciones manteniendo el orden por popularidad. ¿Cómo afecta esto al rendimiento?

```
function insertarCancion() {
  if (!fs.existsSync(sortedFilePath)) {
    console.error(`Archivo ${sortedFilePath} no encontrado.
Ejecuta primero el módulo de ordenación.`);
    return;
  }

  // Ejemplo de nueva canción:
  // Formato:
  id|titulo|campo3|artista|campo5|popularidad|campo7|campo8

  const nuevaCancion = "nuevo123|Nueva
Cancion|1|ArtistaNuevo|AlbumNuevo|75|000000|url";
  const nuevaPop = parseLine(nuevaCancion).popularity;

  let lines = fs.readFileSync(sortedFilePath, 'utf8')
    .split('\n')
    .filter(line => line.trim() !== '');
  let outputLines = [];
  let insertado = false;

  // Recorremos secuencialmente hasta encontrar la posición
  correcta
  for (let line of lines) {
    if (!insertado && nuevaPop >= parseLine(line).popularity) {
      outputLines.push(nuevaCancion);
      insertado = true;
    }
    outputLines.push(line);
  }
  if (!insertado) {
    // Si la nueva canción es la de menor popularidad, se agrega
    al final.
    outputLines.push(nuevaCancion);
  }

  fs.writeFileSync(outputUpdatedFilePath, outputLines.join('\n'),
'utf8');
  console.log(`Nueva canción insertada en archivo
ordenados_actualizado.txt (Inserción: O(n)).`);
}
```

Análisis de la Inserción de Canciones y su Efecto en el Rendimiento

El código presenta un método para insertar nuevas canciones en un archivo previamente ordenado por popularidad.

Funcionamiento del Algoritmo

La función `insertarCancion()` lee un archivo ya ordenado, identifica la posición correcta para una nueva canción según su valor de popularidad, y la inserta manteniendo el orden descendente. El proceso requiere un solo recorrido del archivo.

Efecto en el Rendimiento

Esta implementación afecta el rendimiento del sistema de varias formas:

- Complejidad temporal $O(n)$: El algoritmo debe recorrer secuencialmente el archivo hasta encontrar la posición correcta de inserción, examinando potencialmente todas las canciones existentes. Para archivos con muchas canciones, este tiempo aumenta linealmente.
- Operaciones de lectura/escritura: El método carga completamente el archivo en memoria, realiza la inserción, y luego escribe todo nuevamente a un archivo diferente. Estas operaciones de E/S son costosas, especialmente cuando el archivo es grande.
- Consumo de memoria: Al cargar todo el archivo en memoria, el método utiliza una cantidad de memoria proporcional al tamaño del archivo. Para conjuntos de datos muy grandes, esto podría ser problemático.

- Ventaja del orden previo: El algoritmo aprovecha que el archivo ya está ordenado, lo que permite insertar en $O(n)$ sin necesidad de reordenar. Esto es significativamente más eficiente que insertar y luego ordenar (que sería $O(n^2)$ con el algoritmo de selección visto anteriormente).
- Caso óptimo vs. peor caso: El rendimiento varía dependiendo de dónde deba insertarse la nueva canción. Si debe insertarse al principio, el algoritmo termina rápidamente; si va al final, debe recorrer todo el archivo.
- Múltiples inserciones: Si se realizan varias inserciones consecutivas, cada una requiere un recorrido completo del archivo, lo que resulta en una complejidad acumulada de $O(k \times n)$ para k inserciones.

Este método de inserción es eficiente para operaciones ocasionales en archivos de tamaño moderado, pero podría presentar limitaciones de rendimiento para archivos muy grandes o cuando se requieren inserciones frecuentes.

8. ¿Cómo implementarías una búsqueda binaria en un archivo secuencial ordenado? ¿Qué requisitos debe cumplir el archivo?

```
function busquedaBinaria(targetPopularity) {
  if (!fs.existsSync(sortedFilePath)) {
    console.error(`Archivo ${sortedFilePath} no encontrado.
Ejecuta primero el ordenamiento.`);
    return;
  }

  let lines = fs.readFileSync(sortedFilePath, 'utf8')
    .split('\n')
    .filter(line => line.trim() !== '');

  let low = 0, high = lines.length - 1;
  let foundIndex = -1;
```

```

// Búsqueda binaria (Teórica complejidad:  $O(\log n)$  si se
tuviera acceso aleatorio real)
while (low <= high) {
  let mid = Math.floor((low + high) / 2);
  let midPop = parseLine(lines[mid]).popularity;

  if (midPop === targetPopularity) {
    foundIndex = mid;
    break;
  } else if (midPop < targetPopularity) {
    // Al estar ordenado de mayor a menor, si midPop es menor,
    buscamos en la parte izquierda.
    high = mid - 1;
  } else {
    low = mid + 1;
  }
}

if (foundIndex === -1) {
  console.log(`No se encontró ninguna canción con popularidad
${targetPopularity}.`);
} else {
  console.log(`Canción encontrada en el índice ${foundIndex}:`);
  console.log(lines[foundIndex]);
}
}

// Se puede pasar el valor a buscar como argumento de línea de
comandos.
// Por defecto se usa 75.
const target = process.argv[2] ? parseInt(process.argv[2], 10) :
75;
busquedaBinaria(target);

```

El código proporciona una implementación de búsqueda binaria diseñada para localizar canciones según su nivel de popularidad dentro de un archivo previamente ordenado. La función `busquedaBinaria()` comienza cargando el archivo completo en memoria para luego aplicar el algoritmo clásico de división y conquista, que define índices iniciales, calcula puntos medios y ajusta progresivamente el espacio de búsqueda hasta encontrar el valor deseado o determinar su ausencia.

Requisitos del Archivo para Búsqueda Binaria

Para que una búsqueda binaria funcione correctamente en un entorno de archivos secuenciales, el archivo debe satisfacer varios requisitos fundamentales. Primero, los registros deben estar completamente ordenados según el campo de búsqueda, en este caso la popularidad, donde el código asume específicamente un ordenamiento descendente. Sin este ordenamiento previo, la lógica de comparación y reducción del espacio de búsqueda fallaría completamente.

Adicionalmente, todas las entradas del archivo deben mantener un formato consistente y predecible que permita a la función `parseLine()` extraer correctamente el valor de popularidad de cada registro. Esta uniformidad resulta esencial para garantizar comparaciones válidas durante el proceso.

Aunque los archivos secuenciales normalmente solo permiten lectura lineal, la búsqueda binaria requiere acceso directo a posiciones arbitrarias dentro del conjunto de datos. La implementación actual resuelve elegantemente esta limitación cargando todo el contenido en memoria, transformando efectivamente el acceso secuencial en aleatorio.

El campo de popularidad debe ser comparable numéricamente para que las operaciones de mayor que, menor que e igualdad funcionen correctamente. Finalmente, el algoritmo debe contemplar cómo manejar valores duplicados, donde esta implementación devuelve simplemente el primer registro que coincide con el valor buscado.

La solución logra una complejidad algorítmica de $O(\log n)$, lo que representa una mejora significativa frente a búsquedas lineales, especialmente importante cuando se trabaja con archivos de gran tamaño, aunque requiere la carga inicial completa del archivo para funcionar adecuadamente.

9. Propón una estructura de archivos de índices para acelerar las búsquedas por artista sin perder el enfoque secuencial.

```
function crearIndicePorArtista() {
  if (!fs.existsSync(inputFilePath)) {
    console.error(`Archivo ${inputFilePath} no encontrado.`);
    return;
  }

  let lines = fs.readFileSync(inputFilePath, 'utf8')
    .split('\n')
    .filter(line => line.trim() !== '');

  let indice = {};

  // Recorrer el archivo secuencialmente (O(n))
  lines.forEach((line, index) => {
    let record = parseLine(line);
    let artista = record.artista || "Desconocido";
    if (!indice[artista]) {
      indice[artista] = [];
    }
    indice[artista].push(index);
  });

  fs.writeFileSync(outputIndexPath, JSON.stringify(indice, null, 2), 'utf8');
  console.log(`Índice por artista generado y guardado en ${outputIndexPath}.`);
}
```

5

Ventajas y limitaciones



Durante el desarrollo del laboratorio se identificaron distintas ventajas y limitaciones al utilizar archivos secuenciales como estructura de datos principal:

Ventajas:

- Bajo consumo de memoria, ya que los datos se procesan línea por línea sin necesidad de cargar todo el archivo en memoria.
- Implementación rápida y práctica de operaciones básicas como lectura y escritura secuencial.
- Eficiencia al trabajar con conjuntos de datos moderados que no requieren acceso aleatorio.
- Persistencia de datos garantizada sin necesidad de estructuras adicionales.
- Simplicidad conceptual que facilita la comprensión e implementación.

Limitaciones:

- Baja eficiencia en operaciones de búsqueda no secuenciales o acceso aleatorio.
- Dificultades significativas en la actualización o inserción de datos en ubicaciones específicas del archivo.
- Necesidad de recorrer todo el archivo para realizar operaciones complejas como ordenamientos o cálculos estadísticos.
- Rendimiento deteriorado al trabajar con grandes volúmenes de datos.
- Imposibilidad de optimizar búsquedas sin implementar estructuras auxiliares como índices.



El laboratorio permitió aplicar los conocimientos adquiridos a lo largo de la asignatura, desde el manejo de estructuras de datos secuenciales básicas hasta su implementación en el análisis de playlists de Spotify.

Se comprobó que, a pesar de no ser la estructura más flexible, **los archivos secuenciales resultan óptimos para resolver consultas simples de manera ordenada**. El análisis demostró que constituyen una alternativa viable cuando se busca optimizar el uso de memoria principal, especialmente en sistemas con recursos limitados.

No obstante, también se evidenció **la necesidad de considerar estructuras más complejas cuando los requerimientos incluyen acceso aleatorio, procesamiento de mayor volumen de datos o un rendimiento superior para operaciones complejas**. En un entorno de producción real como Spotify, sería necesario complementar los archivos secuenciales con otras estructuras como índices, árboles o tablas hash para garantizar tiempos de respuesta adecuados.

La experiencia práctica con la **API de Spotify** y el procesamiento de datos reales proporcionó un contexto valioso para comprender las implicaciones de las decisiones de diseño en estructuras de datos y su impacto en el rendimiento general del sistema.



- 1.Documentación oficial de la API de Spotify. (2025). Spotify for Developers.
- 2.Spotify Web API Reference. (2025). Endpoints.
- 3.Node.js Documentation. (2025). File System (fs).
- 4.JavaScript MDN Web Docs. (2025). Working with Files. Mozilla Developer Network.
- 5.Axios Documentation. (2025). Promise based HTTP client for the browser and node.js.