Generación de playlists

Diego Riquelme (32 horas) *Departamento de Ingeniería Informática*
Universidad de Santiago de Chile, Santiago, Chile
diego.riquelme.s@usach.cl

***Resumen*—El siguiente documento da información detallada sobre la solución propuesta para el problema de generación de playlist para el curso de análisis de datos y estructura de algoritmos, dejando saber al lector las características especificas del algoritmo generado, su función, pros y contras, además de la contextualización necesaria para entender el problema.**

***Palabras claves*—Algoritmos, Greedy, Fuerza bruta, Subset sum problem.**

I. Introducción

Generando el contexto, a partir de la vuelta a la normalidad por parte de la ciudadanía, una empresa ha dado la tarea de crear una estrategia computacional que sea capaz de seleccionar canciones de una lista, que pertenezcan a un genero en especifico y que cumplan, o se acerquen lo más posible a una cantidad de tiempo requerida y definida anteriormente. Por lo que el objetivo propuesto, será generar un código dentro del lenguaje C que nos permita obtener dicha playList, analizando las diversas opciones disponibles para la ejecución de la tarea, así como también las investigaciones anteriormente realizadas.

II. Solución propuesta

Para la tarea propuesta, se puede ver que la principal problemática se genera al momento de seleccionar cual seria la mejor combinación de canciones que cumplan con el tiempo requerido, o cercano a este. Obtenida esta información, se puede analizar el gran parecido con el problema de ”subset sum problem”, ya que para este, se nos hace entrega de un array de números los cuales la suma de ellos, debe dar un número en especifico. Este problema, ya ha sido estudiado por científicos, y pertenece a los denominados NP-completos. Para resolver este tipo de problemas, hay varias formas, desde programación dinámica a fuerza bruta, esta ultima fue la primera estrategia pensada para el cálculo del problema, pero dada la cantidad de canciones, no sería un calculo eficiente, o hasta computable, ya que, se necesitaría generar las combinaciones de 0 hasta el tamaño de las canciones de un genero, y pasando a través de todas las combinaciones posibles para cada uno. Motivados por lo anterior, se hace investigación de diversas formas de resolver este problema, pero que encontrara respuesta en un tiempo prudente, de esa forma, se analizan los algoritmos de Greedy o Voraces como una posible solución al problema propuesto, ya que si bien no da un resultado cien por ciento certero, si se acerca bastante a la solución requerida en un tiempo totalmente razonable. El algoritmo de Greedy trabaja buscando una solución localmente óptima para ese instante en especifico, es por ello que lo primero que se hizo, fue ordenar los tiempos de cada canción en orden ascendente para poder completar el resultado más rápido y con menor brecha al final del array, ya que se completaría primero con las canciones de mayor duración, dejando al final las más cortas con el fin de tener más precisión al seleccionar la solución local que esté más cercana al tiempo requerido. Si la suma de los tiempos de las canciones es menor al tiempo requerido, se retorna la lista completa.

Si se observa graficamente la idea, se obtiene lo siguiente:

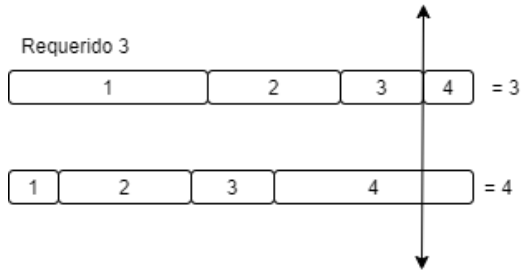

Figura 1: Implementación Greedy según orden

Ahora, implementando el algoritmo creado en pseudo-código, se vería de la siguiente forma (Figura 2): El cual, si calculamos su complejidad, nos entrega un

$$O(n^2).$$

Esto debido al bubbleSort implementado, ya que la implementación de Greedy dado los elementos en orden, es de O(nLogn).[1]

III. Resultados

Ahora, si analizamos los resultados obtenidos a partir del código creado, se puede observar que, a pesar de tener grandes cantidades de datos, el código sigue respondiendo de buena forma, y teniendo tiempos mínimos en comparación con la deducción de fuerza bruta, ya que, se necesitaría:

```
FOR i<- 0 TO r:
     n!/i!(n-i)!
```

Posibilidades, siendo n la cantidad total de canciones e i la cantidad de elementos que se irán seleccionando hasta llegar a r. Por el contrario, con el método utilizado, al realizar test con 1006913 canciones, demora apenas 0.609375 segundos sobre un i7 5700hq a 2.7Ghz.

En el siguiente gráfico se puede apreciar los tiempos recogidos para cada cantidad de datos (Figura 3): Como se

```
makePlayList(num genre,songs arrayOfSongs,num sizeAllSongs,str maxDuration
    ,num lenPlayList,num timePlayList):songs

    num amountSongs <- 0
    FOR num i <- 0 TO sizeAllSongs STEP 1:
        IF arrayOfSongs[i].genre ==genre THEN:
            amountSongs <- amountSongs + 1
    sameGenre<- init_array[amountSongs]...Inicializa un arreglo
    num k<-0
    FOR num i = 0 TO sizeAllSongs STEP 1:
        IF arrayOfSongs[i].genre==genre THEN:
                sameGenre[k] <- arrayOfSongs[i]
                k <- k + 1
    ...Implementacion BubbleSort
    FOR num i = 0 TO amountSongs STEP 1:
        FOR num i = 0 TO sizeAllSongs-1-i STEP 1:
            IF toSeconds(sameGenre[j].duration)<toSeconds(sameGenre[j+1].duration) THEN:
                ...toSeconds es una funcion creada anteriormente de complejidad constante
                ...Ya que solo traduce el formato HH:MM:SS
                songs temporal<-sameGenre[j];
                sameGenre[j]<-sameGenre[j+1];
                sameGenre[j+1]<-temporal;

    ...Comienzo de aplicacion de Greedy, toma de desición
    ...en opción optima local.
    num totalSeconds <- 0
    FOR num i = 0 TO amountSongs STEP 1:
        totalSeconds<-totalSeconds+toSeconds(sameGenre[i].duration);
    num sumTime <- 0
    num i <- 0
    IF totalSeconds>totalSeconds(maxDuration) THEN
        WHILE sumTime < toSeconds(maxDuration) DO
            sumTime<-sumTime+toSeconds(sameGenre[i].duration)
            i<- i+ 1

        num restTime<-sumTime-toSeconds(sameGenre[i].duration);

        num higherTime<- abs(toSeconds(maxDuration)-sumTime);
        num lowerTime<- abs(toSeconds(maxDuration)-restTime);
        IF lowerTime<higherTime THEN
            timePlaylist<-restTime;

            i<-i-1;
            lenPlaylist<-i;
            songs returnPlaylist<- init_array[i]

            FOR num j = 0 TO i STEP 1:
                returnPlaylist[j]<-sameGenre[j]
            return returnPlaylist

        ELSE
            timePlaylist<-sumTime;
            lenPlaylist<-i;
            songs returnPlaylist<- init_array[i]
            FOR num j = 0 TO i STEP 1:
                returnPlaylist[j]<-sameGenre[j]
            return returnPlaylist
    ELSE
        lenPlaylist<-amountSongs
        timePlaylist<-totalSeconds
        return sameGenre
```

Figura 2: PseudoCodigo algoritmo Playlist

observa, si bien se obtiene una complejidad

$$O(n^2)$$

, esta empieza a tomar peso con cantidad de canciones sobre el millón, a diferencia de la fuerza bruta. Solo a modo de ilustración, se presenta el siguiente gráfico de fuerza bruta para contraseñas, por parte de CloudFlare (figura 4)[2]:

IV. Conclusiones

A modo de síntesis, durante la realización del presente trabajo, se logró conocer diversos tópicos y formas de abordar problemas algorítmicos como el subset sum problem, analizando la teoría de estos, la resolución, y las soluciones propuestas, las cuales si bien, no llegan al resultado acertado al cien por ciento por el tipo de algoritmo que se utiliza, si se acerca mucho a los resultados esperados por el cliente en cuestión, generando satisfactoriamente la playlist deseada en tiempos razonables, por lo que se concluye que se logró el objetivo planteado al comienzo de este articulo, pero aún se puede mejorar, como por ejemplo, implementado programación dinámica.

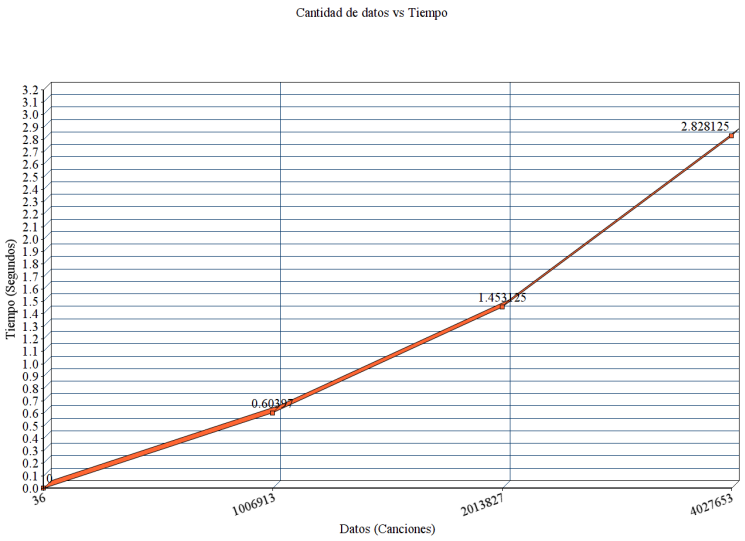

Figura 3: Cantidad canciones VS Tiempo segundos.

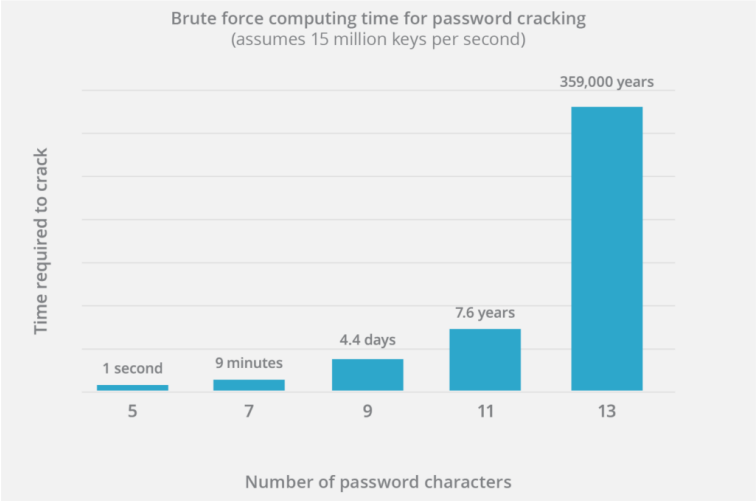

Figura 4: Ataque fuerza bruta.

V. Anexo manual de usuario

Para poder compilar satisfactoriamente el código entregado, es necesario tener instalado y actualizado el compilador GCC, cabe destacar que para la correcta compilación en sistemas operativos con kernel Linux, es necesario incluir la banderilla -lm luego del comando gcc, esto debido al uso de la librería math.h dentro del código.

Por otro lado, los archivos inputs deben seguir la estructura entregada por la coordinación para esta tarea. Luego de ello, si ejecutamos el programa, informará la cantidad de canciones en total, las canciones de un mismo genero, luego las de un mismo genero sin repetir, para luego informar la generación de la playlist y el tiempo utilizado para esta.

REFERENCIAS

[1] P. R. Fillottrani, "Algoritmos y complejidad," urlhttp://www.cs.uns.edu.ar/ prf/teaching/AyC17/ downloads/Teoria/Greedy-1x1.pdf, 2017.

[2] CloudFlare, "What is a brute force attack?" urlhttps://www.cloudflare.com/es-la/learning/bots/brute-force-attack/, 2021.

CONTRIBUCIÓN

Conceptualización, D.R.; Metodología, D.R.; software, D.R.; validación, D.R.; análisis, D.R. ; investigación, D.R ; recursos, D.R; preparación de datos, D.R.; escritura de borrador, D.R.; revisión de la escritura y formalización, D.R ; formato y visualización, D.R.; supervisión, D.R.; El autor ha leído y está de acuerdo con la entrega de este documento, así como el trato de principios éticos.