David Marquez Mínguez

47319570Z

Ejercicios TEMA 2

ALGORITMIA Y COMPLEJIDAD

# **EJERCICIO 2**

Disponemos de una cinta magnética y una serie de ficheros, donde cada uno tiene una determinada longitud L. La cinta magnética tiene un número determinado de peticiones P, es decir, el número de veces que vamos a leer el fichero.

|  |  |
| --- | --- |
| L(longitud) | P(petición) |

El objetivo es crear un algoritmo voraz para saber en qué posición debemos disponer cada uno de los ficheros con el objetivo de reducir el máximo tiempo posible.

Algo a tener en cuenta es que cada vez que leemos un fichero, debemos recorrer la cinta magnética desde el principio ya que en cada lectura se retrocede al inicio de la cinta. Esto implica la lectura de todos aquellos archivos que se encuentren por delante del archivo que leemos por lo que para minimizar el tiempo de carga tendremos que averiguar cómo tendremos que organizar los ficheros en la cinta para tener que leer la menor cantidad de datos posibles y, por tanto, obtener el menor tiempo de carga posible. A continuación, se muestra un pequeño ejemplo:

|  |  |  |  |
| --- | --- | --- | --- |
| Longitud: | 5 | 2 | 1 |
| Peticiones: | 1 | 2 | 6 |

En dicho ejemplo, disponemos de tres ficheros, donde tenemos en primer lugar la longitud del fichero y, en segundo lugar, el número de peticiones. Por tanto, el primero fichero tendría tamaño 5 y 1 petición, así sucesivamente. A continuación, se calcula el tiempo total para dicho ejemplo:

Como podemos observar, para leer el primer fichero hemos necesitado 5 unidades de tiempo, ya que no hay ningún otro fichero delante de él. Multiplicamos la longitud por el número de peticiones y obtenemos el número de unidades de tiempo en leer dicho fichero.

Para la lectura del segundo fichero, primero tenemos que pasar por todos aquellos ficheros que se encuentren por delante, en nuestro ejemplo, el fichero de tamaño 5 será leído antes de pasar a leer el de tamaño 2. Sucede lo mismo para leer el tercer archivo, primero debemos leer los archivos que se encuentran por delante de él, en este caso los archivos de tamaño 5 y 2.

Una vez que entendemos cómo funciona la lectura de los ficheros, debemos pensar en un algoritmo voraz para realizar dicha lectura lo más eficiente posible, y con eficiente nos referimos a aquella que tarde el menor tiempo posible. En un principio podríamos pensar en dos opciones:

1. Leer primero los archivos de menor tamaño.
2. Leer primero los archivos con mayor número de peticiones

**Leer primero los archivos de menor tamaño:**

A continuación, aplicamos el siguiente algoritmo voraz. Primero colocamos los ficheros de menor tamaño, con eso conseguimos que la lectura de los ficheros de mayor tamaño, es decir, los que se encuentran al final, no sea tan costosa.

Siguiendo con el ejemplo anterior, realizamos dicho algoritmo voraz y calculamos el tiempo total el realizar la lectura.

|  |  |  |  |
| --- | --- | --- | --- |
| Longitud: | 1 | 2 | 5 |
| Peticiones: | 6 | 2 | 1 |

Como podemos observar el tiempo de lectura se ha reducido de 67 unidades de tiempo a 20, lo que supone una mejora bastante notable.

**Leer primero los archivos con más peticiones:**

Si realizamos la aproximación con los archivos con más número de peticiones, en este ejemplo nos quedaría la misma aproximación realizada anteriormente.

|  |  |  |  |
| --- | --- | --- | --- |
| Longitud: | 1 | 2 | 5 |
| Peticiones: | 6 | 2 | 1 |

Una vez que hemos presentado nuestros dos métodos voraces con un ejemplo, porque debemos aplicar solo uno y no los dos al mismo tiempo, es decir, en vez de determinar que fichero va antes dependiendo del tamaño de los ficheros o del número de repeticiones, porque no determinar dicho orden médiate el promedio de longitud y numero de repeticiones. Nos interesaran incluir primero los ficheros cuyo valor sea el más pequeño.

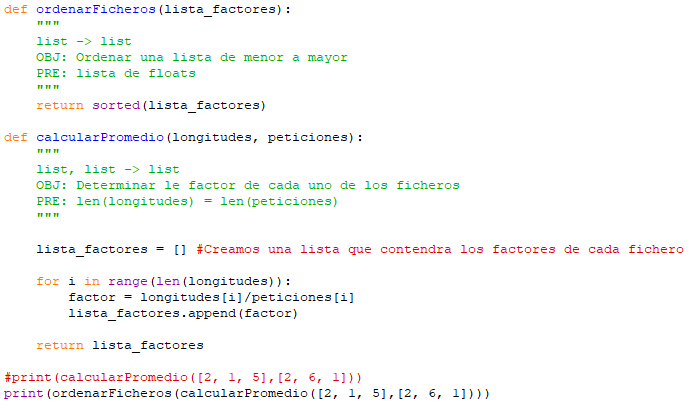
Por lo tanto, nuestro algoritmo seguirá los siguientes pasos:

1. Calcular el promedio de cada uno de nuestros ficheros.
2. Ordenar dichos ficheros en orden creciente.
3. Devolver una lista con los ficheros ordenados de menor a mayor.

A continuación, se muestra el algoritmo implementado en Python y se explica cada una de las funciones, además de una muestra de su funcionamiento.

Antes de mostrar el algoritmo debemos destacar que se disponen de dos listas, una lista de longitudes de los ficheros y otra lista del número de peticiones. Por ello, la posición 0 de ambas listas corresponderá con el primer fichero, la posición 1 con el segundo, y así sucesivamente.

|  |  |  |  |
| --- | --- | --- | --- |
| Longitud | 2 | 1 | 5 |
| Número de peticiones | 2 | 6 | 1 |
| Número de fichero | Fichero1 | Fichero2 | Fichero3 |



El programa dispone de dos funciones, una que calcula el factor de cada fichero y otra que ordena la lista. Podríamos haber realizado la implementación en una única función que realizase ambas tareas, pero para respetar el paradigma de programación modular se ha realizado de dicha manera.

A continuación, va a ejecutar el código empleando los mismos datos mostrados en el ejemplo para observar el correcto funcionamiento del algoritmo.

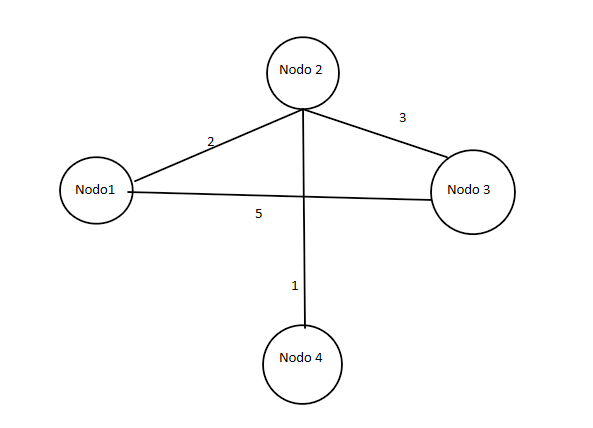


El algoritmo nos devuelve la lista de factores de dada fichero ordenada de menor a mayor. Con ellos ya sabemos el orden de lectura de los mismos.

# **EJERCICIO 4**

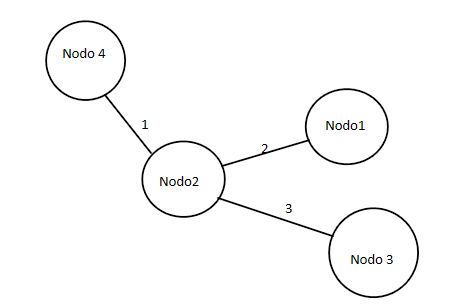
En este problema, se plantea encontrar el camino mínimo de un grafo empleando el algoritmo de Prim. Con este algoritmo iremos creando un árbol de recubrimiento mínimo donde en cada paso ira aumentando hasta encontrar la solución final. Para entender mejor el ejercicio, vamos a exponer los pasos del algoritmo:

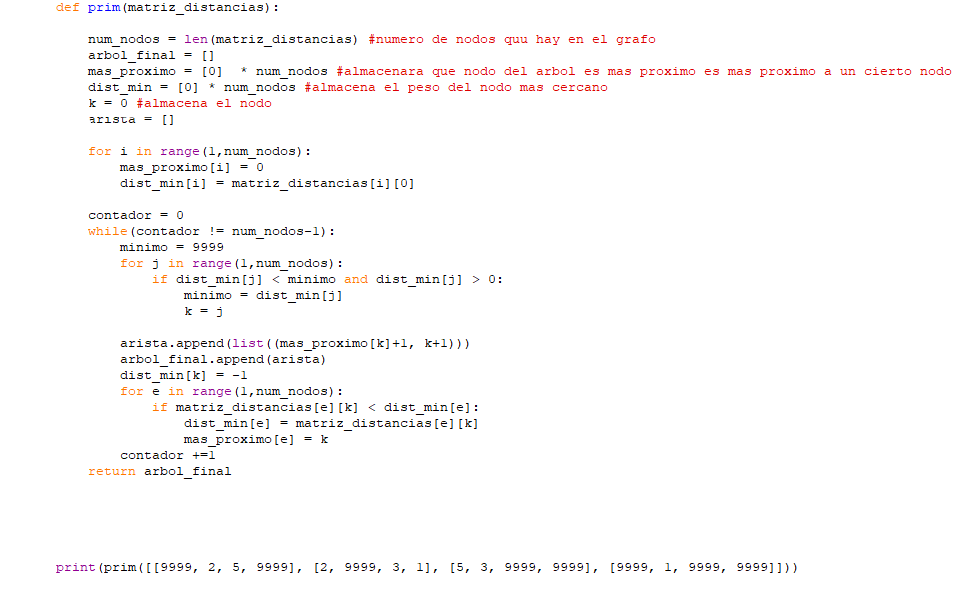
1. Se elige una arista con peso mínimo. Sea a dicha arista, de define T1 = a que es el conjunto de aristas del árbol parcial.
2. Se supone construido en el paso (k-1) el árbol parcial Tk-1. Veamos cómo se construye el árbol parcial Tk en el paso k-esimo. Se elige una arista de peso mínimo incidente con un vértice del árbol parcial Tk-1 que no forme un ciclo. Sea e una arista tal. Se actualiza el árbol parcial añadiendo dicha arista, es decir Tk = Tk-1 U {e}.
3. El algoritmo finaliza después de repetir lo anterior n – 1 veces.

Para entender mejor el algoritmo, vamos a crear un pequeño dibujo, donde disponemos de cuatro nodos y cuatro aristas. Una consideración importante es que dichas aristas no son dirigidas.

A continuación, se muestra el algoritmo implementado en Python, donde se dispone como entrada una matriz de costes que representa los costes de cada arista. Una consideración importante, es que las aristas que existen en el grafo se indican con un número muy alto, en este caso 9999.

En cuanto a la salida del programa, seria algo como lo siguiente:





Mostramos la salida del programa, donde se ve una lista de listas que representan el árbol solución, el cual dibujado sería el siguiente:

Voy a explicar ahora el funcionamiento del algoritmo y de las estructuras de datos creadas en el mismo.

En primer lugar, debemos catalogar las aristas de un nodo. Esto se produce en el primer bucle for. Se elige como primer nodo del árbol de recubrimiento mínimo el nodo 0 y se catalogan todas las aristas que tienen como uno de sus nodos vecinos el nodo 0 que acabamos de escoger. Para ello lo que se hace es decir que para cualquier otro nodo su nodo vecino es el nodo 0. Podríamos pensar que si hacemos esto estaríamos diciendo que todos los nodos del árbol tienen como nodo vecino el nodo 0, pero no ya que, si el nodo no es un vecino, diremos que dicha arista tiene un peso infinito, en nuestro caso 9999.

Una vez hecha la catalogación de aristas podemos comenzar a construir el grafo de recubrimiento mínimo. Esto se realiza dentro del bucle while.

Ahora, se escoge entre las aristas de menor peso. Por lo cual vamos a recorrer todas las aristas catalogadas en el array dist\_min para escoger la más pequeña. Es importante destacar que el valor que nos da la evaluación de dist\_min debe encontrarse entre 0 y el mínimo que ya teníamos. Como en la primera iteración aún no hemos escogido ningún mínimo la variable que lo almacena toma valor infinito.

Habiendo ya capturado la arista de menor peso podemos introducir en la solución la arista que hemos introducido. Para esto simplemente construimos un array de 2 posiciones que contenga los nodos que forman parte de esa arista: que son, el nodo k y el nodo mas\_proximo[k].

Por otro lado, realizamos un control que nos permita evitar ciclos. Evidentemente cuando insertamos un nodo en el árbol no podremos volverlo a meter en él, por lo que para evitar esto diremos que la distancia mínima que hay de ese nodo al árbol es -1.

Volvemos a catalogar todas las aristas. Para cada arista de peso mínimo de un cierto nodo tenemos que verificar si existe otra arista de menor peso que nos una al árbol, es importante que esa arista debe tener obligatoriamente como nodo vecino el nodo k, ya que es el nodo que hemos introducido, si eso ocurre, entonces diremos que la arista de peso mínimo de un cierto nodo es esa nueva arista. Una vez hecho esto ya podremos volver a ejecutar el bucle con las nuevas aristas para elegir entre todas ellas la menor, y así sucesivamente hasta que tengamos en nuestro árbol n-1 aristas (siendo n el número de nodos del grafo).

# **EJERCICIO 6**

En este problema disponemos de una serie de escaleras para intentar escalar un muro de gran altura. Ninguna de las escaleras es capaz por si sola de sobrepasarlo, pues necesitamos la unión de varias de ellas para lograr nuestro objetivo. Cuando unimos dos escaleras, el tiempo que consumimos es la suma de su altura, es decir, si por ejemplo tenemos dos escaleras de altura 5m y 7m, el tiempo que tardamos en unir ambas es 7 + 5 = 12 unidades de tiempo.

El objetivo es crear un algoritmo voraz con el que poder escalar el muro de la manera más optima posible, y con optimo nos referimos al menor tiempo posible.

Para entender mejor el problema, vamos a poner un pequeño ejemplo, donde tenemos 4 escaleras de diferentes tamaños, 12m, 6m, 23m, 8m. La suma de las alturas de las escaleras nos permite sobrepasar el muro. Si unimos las escaleras en el orden el que se crearon, obtenemos lo siguiente:

1. En primer lugar, unimos las dos primeras escaleras, 12 + 6 = 18 unidades de tiempo.
2. Ahora unimos la tercera escalera, 23 + 18 = 41 unidades de tiempo.
3. Finalmente unimos la última escalera, al igual que antes, debemos haber solado las anteriores, 41 + 8 = 49 unidades de tiempo.

El tiempo total para unir todas las escaleras es el tiempo parcial de cada una, es decir, 18 + 41 + 49 = 108 unidades de tiempo.

Una vez que hemos entendido el problema, podríamos pensar en una estrategia para resolverlo en el menor tiempo posible. Una idea sería unir primero las escaleras de menor tamaño con el objetivo de acumular el menor tiempo posible. Vamos a realizar el mismo ejemplo que antes, pero aplicando dicho metodo. El orden para unir sería el siguiente: 6m < 8m < 12m < 23m.

1. Unimos las escaleras más pequeñas, en este caso las de 6m y 8m. 6 + 8 = 14 unidades de tiempo.
2. Una vez se han unido ambas escaleras procedemos a unir la tercera 14 + 12 = 26 unidades de tiempo.
3. Finalmente, y al igual que antes, unimos la última escalera, 26 + 23 = 49 unidades de tiempo.

La suma de los tiempos parciales de las uniones de las escaleras será el tiempo que hemos tardado en construir nuestra escalera para pasar el muro. En este caso aplicando la lógica que hemos mencionado anteriormente hemos tardado, 14 + 26 + 49 = 89 unidades de tiempo.

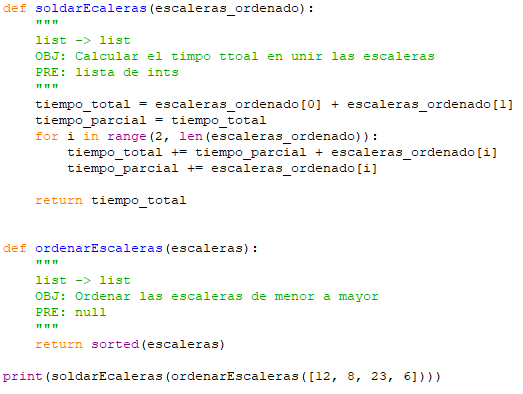
Como podemos observar el tiempo total se ha reducido al unir primero las escaleras más pequeñas.

A continuación, se muestra el algoritmo programado en Python y el resultado para el ejemplo que hemos puesto anteriormente. Los pasos de nuestro algoritmo serán los siguientes:

1. En primero lugar se ordenarán las escaleras en orden creciente, de forma que las escaleras de menor tamaño sean las primeras de la lista.
2. Realizamos un bucle de forma que se harán tantas uniones como sean necesarias.
3. Almacenamos los tiempos parciales y calculamos el tiempo total de la unión de las escaleras.

Al igual que en ejercicios anteriores, se respeta el paradigma de programación modular, por ello se crean dos funciones diferentes, una que se encarga de ordenar la lista de escaleras y otras que se encarga de implementar el algoritmo.

Por otro lado, inicializamos la variable tiempo total como la suma de las dos primeras escaleras, esto nos permite que nuestro algoritmo sea lo más eficiente posible puesto que ahora tenemos que hacer dos veces menos el bucle.



A continuación, se muestra también la salida del programa y se puede comprobar como coincide con lo que nos habíamos planteado en el ejemplo.

