Ejercicios T2

Jorge Guillamón Brotóns y Miguel Ángel Guerrero López

25 de marzo de 2020

Ejercicio 2
Se tiene que almacenar un conjunto de n ficheros en una cinta magnética (soporte de almacenamiento de recorrido secuencial), teniendo cada fichero una longitud conocida l1, l2, ..., ln. Para simplificar el problema, puede suponerse que la velocidad de lectura es constante, así como la densidad de información en la cinta.
Se conoce de antemano la tasa de utilización de cada fichero almacenado, es decir, se sabe la cantidad de peticiones pi correspondiente al fichero i que se van a realizar. Tras la petición de un fichero, al ser encontrado la cinta es automáticamente rebobinada hasta el comienzo de la misma. El objetivo es decidir el orden en que los n ficheros deben ser almacenados para que se minimice el tiempo medio de carga, creando un algoritmo voraz correcto.

En este caso, la variable a maximizar sería $\frac{p_i}{l_i}$, se ha elaborado un programa en Python para realizar la tarea.

Entrada Una lista $[[p_1, i_1][p_2, i_2], ..., [p_n, i_n]]$
Salida La lista de la entrada ordenada.

```
def ordenar(lista):
        puntos = 0
        valor = lambda elem : elem[0]/elem[1]
        for e in range(0, len(lista)):
                for i in range(e, len(lista)):
                        if valor(lista[i]) < valor(lista[i-1]):
                                lista[i],lista[i-1] = lista[i-1], lista[i]
        return lista[::-1]
```

Ejercicio 3 Se dispone de un vector V formado por n datos, del que se quiere encontrar el elemento mínimo del vector y el elemento máximo del vector. El tipo de los datos que hay en el vector no es relevante para el problema, pero la comparación entre dos datos para ver cuál es menor es muy costosa, por lo que el algoritmo para la búsqueda del mínimo y del máximo debe hacer la menor cantidad de comparaciones entre elementos posible. Un método trivial consiste en un recorrido lineal del vector para buscar el máximo y después otro recorrido para buscar el mínimo, lo que requiere un total de aproximadamente 2n comparaciones entre datos. Este método no es lo suficientemente rápido, por lo que se pide implementar un método con metodología Voraz que realice un máximo de $\frac{3}{2}n$ comparaciones.

En este caso se puede aprovechar la comparación destinada al máximo para evitar una comparación con el mínimo en determinados casos. Una implementación de esta propuesta sería:

Entrada: lista de números.
Salida: [min, max].

```
def minimo_maximo(lista):
        minimo = maximo = lista[0]
        for elem in lista:
                if(elem > maximo):
                        maximo = elem
                elif(elem < minimo):
                        minimo = elem
        return [minimo, maximo]
```

Ejercicio 4. Implementar el algoritmo de Prim para un grafo no dirigido.

Se ha escrito un programa para realizar la tarea. Tanto la entrada como la salida es una matriz asociada al grafo. Cada vez que se haga una conexión se sacará por pantalla. Para indicar que dos nodos no están directamente conectados, el peso de su arista será -1.

```
def Prim(matriz):
        n = len(matriz) #Una matriz de nxn
        actual = 0
        visitados = [0]
        pila = [] #Manejar el backtracking
        resultado = [[-1 for i in range(n)] for e in range(n)]

        while len(visitados) < n:
                coste = float("inf")

                for i in range(len(matriz)):
                        if matriz[actual][i] != -1 and i not in visitados
                        and matriz[actual][i]<coste:
                                coste = matriz[actual][i]
                                siguiente = i

                if coste == float("inf"): #Si nada, backtracking
                        actual = pila.pop()
                        if actual == 0:
                                print("No se puede sacar el arbol")
                                return matriz

                else:           #Si se ha encontrado algo, seguimos bajando
                        resultado[actual][siguiente] = coste #matriz simetrica
                        resultado[siguiente][actual] = coste

                        print(str(actual) + " y " + str(siguiente) +
                                " se han unido, coste: " + str(coste))
                        visitados.append(siguiente)
                        pila.append(actual)
                        actual = siguiente
        return resultado
```

Ejercicio 5. Implementar el algoritmo de Dijkstra para un grafo no dirigido.

Se ha escrito un programa para realizar la tarea.La entrada es una matriz asociada al grafo, la salida se saca por pantalla. Para indicar que dos nodos no están directamente conectados, el peso de su arista será -1.

```
def dijkstra(matriz, nodo_destino):
        n = len(matriz)
        costes = [float("inf") for i in range(n)]
        caminos = [[] for i in range(n)]

        #Configuracion inicial
        costes[0] = 0
        caminos[0] = [0]
        frontera = [0]
        actual = 0

        #Resultado hallado o no existe
        while(actual!=nodo_destino or not frontera):
                actual = frontera[costes.index( min(costes) )]
                frontera.remove(actual)
                camino_actual = caminos[actual]

                for i in range(n):
                        if matriz[actual][i] != -1: #Si el nodo es accesible
                                if costes[i] == float("inf"): #No se ha visitado
                                        frontera.append(i)
                                        costes[i] = costes[actual] + matriz[actual][i]
                                        caminos[i] = camino_actual.copy()
                                        caminos[i].append(i)

                                #Coste mejorable
                                elif costes[i] > costes[actual] + matriz[actual][i]:
                                        costes[i] = costes[actual] + matriz[actual][i]
                                        caminos[i] = camino_actual.copy()
                                        caminos[i].append(i)

        print(costes[nodo_destino], caminos[nodo_destino])
```

Ejercicio 6. Shrek y las escaleras
Buscar la mejor manera de soldar n escaleras de cierta longitud. Las escaleras se sueldan por pares y el coste total de soldadura es la suma de las longitudes de las piezas a soldar.

La entrada es una lista de N enteros con las longitudes de las escaleras. Para resolver el problema basta con priorizar las longitudes pequeñas primero.

Demostración: supongamos que estamos eligiendo la primera pieza. Sea la menor de ellas n_0 si se escoge cualquiera que no sea la menor de ellas, digamos n_k conllevará un coste añadido para el resto de soldaduras de

$$t(n_k - n_0), t \in (1, N)$$

con respecto al caso óptimo.
Así el mejor coste de [1 ,4 ,5] sería $1 + 5 + 9 = 15$

```
def shrek(longitudes):
        longitudes.sort()
        print("El orden es", longitudes)
```