PECL2 - Algoritmos Voraces

Jorge Guillamón y Miguel Ángel Guerrero

21 de junio de 2020

Resumen

Se describen los algoritmos de los ejercicios 2, 3, 5, 6. Todos los códigos fuente se pueden encontrar en este mismo directorio.

Ejercicio 2 - Almacenamiento de ficheros en el disco duro

Hemos considerado que el parámetro a maximizar debe ser el número de accesos por byte del archivo. De esta manera, se colocarán en primera posición los archivos más accedidos y pequeños y al final los menos accedidos y pesados. Hemos aprovechado las facilidades que da Python para resolver el problema en 2 líneas.

```python
import random


# Se trata de maximizar tasa/tamaño
def ordenar(lista):
    lista.sort(key=lambda elem: elem[0] / elem[1], reverse=True)
    return lista


ejemplo = [[random.randint(0, 100), random.randint(1, 100)] for i in range(6)]
print("Sean los ficheros ([tasa, tamaño]):", ejemplo)
print("Su orden en el disco debería ser:", ordenar(ejemplo))
```

Figura 1:

Como se puede apreciar, el probador cambia para cada ejecución.

Ejercicio 3 - Mínimo y máximo de una lista

En este caso, lo principal es diseñar un algoritmo eficaz teniendo siempre en cuenta el número de comparaciones que va a realizar, ya que un mínimo y máximo se puede programar de muchas maneras, pero para que cumpla con una eficacia menor a 3/2N hemos tenido que pensar de forma un poco diferente a lo habitual. El algoritmo diseñado sigue cumpliendo las características básicas de los algoritmos voraces, algo que facilita el mismo diseño de este y nos muestra su utilidad.

Como entrada requiere una lista, de la cual obtenemos su longitud. La longitud de esta es clave para el desarrollo. Si se da un número de elementos totales pares, el bucle, que comentamos más adelante, empieza con (i=2), y por el contrario, con (i=1).

A partir de esta parte, pasa al bucle, el cual recorre la lista de números desde su principio, comparando uno a uno, con el siguiente cual es el máximo y cual es el mínimo de manera que los va actualizando en el caso de que A)exista máximo B)exista mínimo, sino, mantendría valores. Si por ejemplo tenemos una Lista [1,2,3]:
El máximo y mínimo se establece al valor en [0].

Entra al While, desde i=1. Si el elemento en [1] es menor que el siguiente elemento, el mínimo, es el mínimo entre el menor número que ya había, y ese número menor en [1].

De esta forma nos aseguramos de que el mínimo siempre irá a menos y el máximo siempre irá más, y como algoritmo voraz, recorre todas las opciones valorando estas posibilidades.

Para el caso de N elementos impares el número de comparaciones llega a ser : 3*(n-1)/2 Si por el contrario, se da que N es de la forma 2*N;

- Debemos de inicializar el máximo y el mínimo haciendo una primera comparación de los 2 primeros elementos, a esto le sumamos 3(n-2)/2 comparaciones para el resto de los elementos de la lista. SI despejamos 1 + 3*(N-2)/2 nos queda el resultado en cualquiera de los casos es menor a 3/2 N, que es 3n/2 -2

```python
def minMax(lista, longi):
    if longi % 2 == 0:
        i = 2
        maximo = max(lista[0], lista[1])
        minimo = min(lista[0], lista[1])
    else:
        i = 1
        maximo = minimo = lista[0]
    while i < longi - 1:
        if lista[i] < lista[i + 1]:
            maximo = max(maximo, lista[i + 1])
            minimo = min(minimo, lista[i])
        else:
            maximo = max(maximo, lista[i])
            minimo = min(minimo, lista[i + 1])
        i += 2
    return [maximo, minimo]
```

Figura 2:

Ejercicio 5 - Dijkstra

Se he implementado un algoritmo de Dijkstra aunque usando estructuras de datos en lugar de matrices para almacenar las distancias mínimas.

```python
# Entrada: casilla de salida, casilla de llegada, matriz de caminos
# Salida: Lista con casillas que hay que recorrer para llegar a la meta
def dijkstra(salida, llegada, matriz):
    frontera = [[[salida], 0]]  # Almacena [camino, coste]
    n = len(matriz)
    conocidos = set()
    while len(conocidos) < n and len(frontera) > 0:
        [ruta, coste] = frontera.pop()
        actual = ruta[-1]
        if actual == llegada:  # Premio!
            return [ruta, coste]

        if actual in conocidos:  # Si ya hemos pasado por aqui, andamos en circulos
            continue
        else:
            conocidos.add(actual)

        for i in range(n):
            if matriz[actual][i] == inf or actual == i:  # Si es innacesible o es el propio nodo
                continue
            frontera += [[ruta + [i], coste + matriz[actual][i]]]

        frontera.sort(key=lambda l: l[1], reverse=True)  # Los caminos más bajos al final, (sacamos con pop)

    return False
```

Figura 3:

En este sentido, los elementos de la frontera del algoritmo almacenan [ruta hasta ese momento, coste asociado]. Al final de cada iteración se ordena la frontera de acuerdo al algoritmo original.

El código está listo para ser probado con 2 matrices de ejemplo. Una vez ejecutado el programa, aparecerá por pantalla el camino más corto junto con su coste.

```
Camino: [0, 2, 1]
Coste:  8

Process finished with exit code 0
```

Figura 4:

Ejercicio 6 - Shrek y las escaleras

La solución que hemos planteado consiste en elegir los 2 tramos de escalera más pequeños disponibles en cada iteración. Tras acabar la iteración, se añade el nuevo tramo a la lista de los diponibles y se eliminan los usados.

```python
def shrek(longitudes):
    suma = 0
    while len(longitudes) != 1:
        longitudes.sort()  # Toca ordenar a cada vuelta, los elementos cambian!
        tramo = sum(longitudes[:2])
        print(longitudes[:2], "=", tramo)
        longitudes = longitudes[2:]
        longitudes.append(tramo)
        suma += tramo

    print("El coste total es de", suma)
```

Figura 5:

La salida es por pantalla, en primer lugar se mostrarán los tramos disponibles. Durante la ejecución del algoritmo, cuando se vayan a fundir dos tramos se indicarán cuales son, finalmente se indicará el coste total.

El ejercicio viene acompañado de un probador que cambia para cada ejecución.

```
Tramos: [6, 1, 6, 3, 10, 6, 9, 9]
[1, 3] = 4
[4, 6] = 10
[6, 6] = 12
[9, 9] = 18
[10, 10] = 20
[12, 18] = 30
[20, 30] = 50
El coste total es de 144
```

Figura 6: