fernandezmartinez-garciaperez-dyv

April 29, 2024

1 Algoritmia

1.1 Práctica 2 - Divide y Vencerás

1.2 ### Curso 2023 - 2024

Autores:

- Cristian Fernández Martínez
- Alicia García Pérez

Resuelva la siguiente práctica.

Importe las librerías que desees **Recuerda**: * Solamente puedes utilizar bibliotecas nativas (https://docs.python.org/es/3.8/library/index.html) * Las funciones que importes no son "gratis", cada una tendrá una complejidad temporal y espacial que se tendrá que tener en cuenta.

[1]:
```
#testeable
# imports
```

[2]:
```
#testeable
# Versión ligeramente modificada de la clase Video de la Práctica 1

# Se ha incluido una propiedad 'fee', correspondiente a los costes
# semanales de licenciamiento del contenido
class Video:
    """
    Clase Video.
    Representa una serie o película.
    """

    def __init__(self, name, size, fee):
        """Crea un objeto de clase Video

        Parameters
        ----------
        name : str
            Nombre de la serie/película
        size : number
```

```
            Tamaño en memoria de la serie/película
        """
        self.name = name
        self.size = size
        self.fee = fee

    def __hash__(self):
        """Genera el valor hash identificativo del vídeo

        Returns
        -------
        int
            Valor hash
        """

        return hash((self.name, self.size, self.fee))

    def __str__(self):
        """Genera una cadena descriptiva del objeto

        Returns
        -------
        str
            Cadena descriptiva
        """
        return f'Nombre del video: {self.name}, tamaño: ({self.size} MB), coste␣
↪licenciamiento: {self.fee}'

    def __repr__(self):
        """Genera una cadena descriptiva del objeto dentro de colecciones

        Returns
        -------
        str
            Cadena descriptiva
        """
        return f'Nombre del video: {self.name}, tamaño: ({self.size} MB), coste␣
↪licenciamiento: {self.fee}'

    def set_users(self, country, users):
        """Dado un pais y un número de usuarios
           almacena para este vídeo la cantidad de espectadores que tiene.

        Parameters
        ----------
        country : str
            País desde donde se ve la serie/película
```

```
        users : int
            Número de espectadores
        """

        self.users[country] = users

    def get_users(self, country):
        """Dado un país, obtiene el número de usuarios.

        Parameters
        ----------
        country : str
            País desde donde se ve la serie/película

        Returns
        -------
        int
            Número de espectadore s para el país `country`
        """

        return self.users.get(country, 0)

    def __lt__(self,other):
        return self.size < other.size

    def __le__ (self,other):
        return self.size <= other.size
```

[3]:
```
#testeable
def key_sort(seq, key_function= lambda x: x, reverse=False):
    """Ordena ascendentemente una secuencia utilizando una función clave.

    Parameters
    ----------
    seq : sequence
        Secuencia a ordenar, por ejemplo una lista
    key_function : int
        Función que devuelve para los elementos de la secuencia un valor
        clave para la ordenación
    reverse: boolean
        Si es True Invierte el orden de la ordenación.

    Notes
    -----
    No tiene retorno. Ordena en el sitio la secuencia de entrada.
    """
```

```
    def quicksort(seq,izq,der): #O(n)
        if izq < der:
            ultimo_izq,primero_der = particion(seq,izq,der)
            quicksort(seq,izq,ultimo_izq)
            quicksort(seq,primero_der,der)

    def particion(seq, izq, der): #O(log n)
        # Tomamos el elemento medio como pivote e i como el indice del menor␣
 ↪elemento
        medio = (izq + der) // 2
        pivote = key_function(seq[medio])
        seq[medio], seq[der] = seq[der], seq[medio]
        i = izq - 1

        for j in range(izq, der):
            if key_function(seq[j]) <= pivote:
                i = i + 1
                seq[i], seq[j] = seq[j], seq[i]

        # Intercambiamos el pivote con el elemento en el indice del menor␣
 ↪elemento
        seq[i + 1], seq[der] = seq[der], seq[i + 1]
        return i, i + 2


    quicksort(seq,0,len(seq)-1)

    if reverse==True:
        seq.reverse()
```

[4]:
```
#testeable
def max_emission_profit(video, forecast, ingresos_por_usuario,␣
 ↪method="bruteforce"):
    """Calcula el máximo beneficio obtenible emitiendo un vídeo según
       la previsión de espectadores disponible.

    Parameters
    ----------
    video : Video
        Vídeo a emitir.
    forecast : iterable
        Previsión de espectadores.
    ingresos_por_usuario: float
        Ingresos brutos que proporciona cada usuario.
    method: String
        Algoritmo a emplear 'bruteforce' o 'divideandconquer'.
```

```
    Returns
    -------
    float
        Máximo beneficio obtenible.

    Notes
    -----
    El máximo beneficio se calcula considerando la mejor combinación
    posible de momento de inicio y finalización dentro del periodo de
    forecast, el coste de licenciamiento del vídeo (fee) y los beneficios
    obtenidos por cada espectador. Si el vídeo no puede emitirse de forma
    rentable, el método devuelve 0.
    'video' debe disponer de una propiedad 'fee' que establece los costes
    de licenciamiento en periodos equivalentes a las previsiones de 'forecast'.
    """
    if method == "bruteforce":
        return max_emission_profit_bruteforce(video, forecast,␣
↪ingresos_por_usuario)
    elif method == "divideandconquer":
        return max_emission_profit_divideandconquer(video, forecast,␣
↪ingresos_por_usuario, 0, len(forecast)-1)


def max_emission_profit_bruteforce(video, forecast, ingresos_por_usuario):␣
↪#O(n^2)
    max_beneficio = 0
    n = len(forecast)

    for inicio in range(n):
        for fin in range(inicio, n):
            # Calculamos el beneficio para cada semana en el rango
            beneficio = sum(forecast[i] * ingresos_por_usuario - video.fee for␣
↪i in range(inicio, fin + 1))
            max_beneficio = max(max_beneficio, beneficio)

    return max(0, max_beneficio)


def max_emission_profit_divideandconquer(video, forecast, ingresos_por_usuario,␣
↪bajo, alto): #O(n log n)
    # Comprobar que 'bajo' y 'alto' están dentro del rango de la lista␣
↪'forecast'
    if bajo < 0 or alto >= len(forecast):
        return 0

    if bajo > alto:
```

```
        return 0

    if bajo == alto:
        return max(0, forecast[bajo] * ingresos_por_usuario - video.fee)

    medio = bajo + (alto - bajo) // 2

    maximo_izq = max_emission_profit_divideandconquer(video, forecast,␣
↪ingresos_por_usuario, bajo, medio)
    maximo_der = max_emission_profit_divideandconquer(video, forecast,␣
↪ingresos_por_usuario, medio+1, alto)
    maximo_cruzado = beneficio_cruzado(video, forecast, ingresos_por_usuario,␣
↪bajo, medio, alto)

    return max(0, maximo_izq, maximo_der, maximo_cruzado)

def beneficio_cruzado(video, forecast, ingresos_por_usuario, bajo, medio, alto):
    suma = 0
    maximo = 0

    # Calculamos el beneficio maximo de la parte izquierda
    for i in range(medio, bajo-1, -1):
        suma += forecast[i] * ingresos_por_usuario - video.fee
        maximo = max(maximo, suma)

    suma = maximo

    # Calculamos el beneficio maximo de la parte derecha
    for i in range(medio+1, alto+1):
        suma += forecast[i] * ingresos_por_usuario - video.fee
        maximo = max(maximo, suma)

    return max(0, maximo)
```

[5]:
```
#testeable
def optimize_content_grid(videos, forecasts, ingresos_por_usuario, k=None):␣
↪#O(n^2)
    """Obtiene los vídeos rentables para su emisión según la previsión
    de espectadores y los ingresos por usuario.

    Parameters
    ----------
    videos : list of Video
        Vídeos a analizar.
    forecasts : list of list of int
        Secuencia de previsiones para los vídeos
    ingresos_por_usuario: float
```

```
        Ingresos brutos que proporciona cada usuario.
    k: int
        Obtiene los k elementos más rentables.

    Returns
    -------
    list
        Lista con los k elementos más rentables.

    Notes
    -----
    La rentabilidad de los vídeos se considera tomando el periodo
    óptimo de emisión para cada uno. El método sólo devuelve resultados
    con rentabilidad mayor que 0.
    """

    for video, forecast in zip(videos , forecasts):
        video.beneficio = max_emission_profit(video, forecast,␣
 ↪ingresos_por_usuario)

    key_sort(videos,key_function=lambda x: x.beneficio, reverse=True)

    if k is not None:
        videos = videos[:k]

    return videos
```

1.2.1 Caso de ejemplo

[6]:
```
import unittest,random

# Vídeos de ejemplo para tests
videos_test = [Video("Vikings: Valhalla", 150, 1500), #Título, tamaño, fee
               Video("New Amsterdam", 56, 980),
               Video("Juego de Tronos",152, 2500),
               Video("La casa de papel", 76, 800),
               Video("Breaking Bad", 160, 1230),
               Video("Pasión de gavilanes", 125, 350),
               Video("The Sinner", 89, 1900),
               Video("El Cid", 35, 410),
               Video("Raised by wolves",80, 1900),
               Video("Euphoria",90, 2000),
               Video("Peacemaker",40, 1000)]

# Audiencias previstas para los anteriores
forecasts_test = [[ 500, 1000, 2000,  150, 2000, 1500,  200, 1200,  700],
                  [1000,  800,  500,  250, 4000,  500,  100,  200,  300],
```

```
                  [1200, 1000,  900,  900, 1220, 1100,  900, 1000, 1000],
                  [ 900,  700,  500,  200, 1600,  200,  300,  250,  100],
                  [ 600,  900, 1240,  250, 2100, 1000,  350, 1000,  800],
                  [ 190, 1060,  500,  200,  250,  700,  200,  400,  400],
                  [ 150, 1100, 2500,  600,  800, 3000,  200, 1000,  900],
                  [ 160,  100,  200,  200,  150,  200,   50,  200,  190],
                  [ 900,  850,  960,  950, 1100,  100,  350,  900,  800],
                  [ 600,  900, 1200,  300, 1900, 1000,  650,  850,  800],
                  [ 510,  500,  500,  500,  500,  500,  500,  500,  510]]

class TestBasico(unittest.TestCase):

    def test_basic_sort(self):
        v = list(range(100))
        random.seed = 1234
        random.shuffle(v)
        key_sort(v)
        self.assertEqual(v, list(range(100)))

    def test_max_profit(self):
        for m in ["bruteforce", "divideandconquer"]:
            self.assertEqual(max_emission_profit(Video("test", 99, 1), [1]*10,␣
 ↪1, method=m), 0)
            self.assertEqual(max_emission_profit(Video("test", 99, 1), [1]*10,␣
 ↪1, method=m), 0)
            self.assertEqual(max_emission_profit(Video("test", 99, 1), [2]*10,␣
 ↪1, method=m), 10)


if __name__ == "__main__":
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Tests Para probar que tu solución pasa los tests. Utilice el comando:

`$ python tests-py3<version de python> <mi notebook>`

Los tests necesitan de la librería `nbformat`

`$ pip install nbformat`

Explicación de los tests

- `test_ej1_basic_sort`: Comprueba que se ordenen correctamente listas de números.
- `test_ej1_key_sort`: Comprueba que se ordenan correctamente listas según su función clave.

- `test_ej1_video_sort`: Comprueba que se pueden ordenar vídeos.
- `test_ej2_max_profit`: Comprueba que se obtiene el beneficio máximo tanto por fuerza bruta como por divide y vencerás para listas de números simples.
- `test_ej2_max_profit_dataset`: Comprueba que se obtiene el beneficio máximo tanto por fuerza bruta como por divide y vencerás para listas más complejas.
- `test_ej2_optimize_content_grid`: Comprueba el buen desempeño de la función `optimize_content_grid`.

1.2.2 Informe

Contesta a las siguientes preguntas.

Complejidad

1. Método `key_sort`
   - **Complejidad temporal**: O(n log n)
2. Método `max_emission_profit`
   - **Complejidad temporal**: Si el metodo es bruteforce es O(n^2), si el metodo es divideandconquer es O(n log n)
3. Método `optimize_content_grid`
   - **Complejidad temporal**: O(n^2)

Ranking de contenidos.

- ¿Cómo afecta la función clave, que proporciona el criterio de ordenación, a la complejidad temporal del algoritmo?

La función clave que proporciona el criterio de ordenación puede afectar la complejidad temporal del algoritmo de varias maneras. - Complejidad de la función clave: Si la función clave es costosa, entonces esto se multiplicará por la complejidad del algoritmo de ordenación. - Número de comparaciones: La función clave también puede afectar el número de comparaciones que el algoritmo de ordenación necesita hacer. Si la función clave hace que muchos elementos sean iguales, entonces el algoritmo de ordenación puede ser capaz de hacer menos comparaciones, lo que podría reducir la complejidad temporal. - Estabilidad del algoritmo de ordenación: Si el algoritmo de ordenación es estable, entonces la función clave no cambiará el orden relativo de los elementos que son iguales. Esto puede ser importante en algunos casos, y puede afectar la eficiencia del algoritmo.

Optimización de la compra de contenidos.

- ¿Has conseguido mejorar la complejidad temporal mediante la solución Divide y Vencerás?
- ¿Por qué ha mejorado? ¿Cuánto ha mejorado? Pon algunos ejemplos numéricos especulativos que relacionen el tamaño del problema con el tiempo de ejecución en ambas versiones. Contrástalos de forma empírica empleando para ello tu implementación.

Sí, al utilizar el método divide y vencerás reducimos los tiempos de ejecución, ya que al inicio ya tenemos el registro ordenado, y la propia ejecución de un método recursivo es más rápida que la ejecución de un método iterativo.

EJEMPLO: si se da el caso de que el tamaño del problema sea igual a 10. El metodo bruteforce tardaría 0.001 segundos, y en cambio el método divide y vencerás tardaría 0.0001 segundos.