Algoritmia y Complejidad

Ejercicios tema 3

Laboratorio Miércoles 15:00-17:00

Integrantes:

Carlos Javier Hellín Asensio
Diego Gutiérrez Marcos

Curso: 2020-2021

Problema 4

Descripción del algoritmo:
Para este problema se ha realizado un quicksort “personalizado” ya que se usa como pivote los elementos que hay dentro de una de las array para ordenar la otra array. Por ejemplo, primero tenemos el array de botellas [4, 1, 3] que queremos ordenar y para ello se va usar como pivotes los elementos que hay en el array de corchos que en este ejemplo va ser [1, 4, 3]. Usando índice como variable que empieza por 0, en la primera interacción de este quicksort se accede al primer elemento de los corchos, es decir tenemos el valor 1. Este elemento lo usamos como pivote y recorrer las botellas sin ese elemento, es decir, vamos a comprobar la array [4, 3] de las botellas y viendo si cada elemento es menor o igual que el pivote que es 1 para que vaya al lado izquierdo o mayor para que vaya al derecho del quicksort.
Una vez hecho esto, llamamos recursivamente al quicksort para pasar a la siguiente interacción con índice + 1, para así obtener como pivote el valor 4 de la array de los corchos y repetir el mismo proceso, tanto para el lado izquierdo como el derecho.

Una vez que tengamos las botellas ordenadas, se usan como pivote para ordenar los corchos siguiendo el mismo proceso de antes (pero ahora las botellas son los pivotes) y así no se usa el sort de Python, sino dos quicksort.

En ambos casos del quicksort tenemos los casos de base que son cuando hay uno o ceros elementos en el array de los corchos/botellas o cuando hay dos elementos, se termina el algoritmo.

Casos de prueba:
El primer caso de prueba que utilizamos para probar el quicksort es un caso de prueba de 3 corchos y 3 botellas ordenados de la siguiente manera ya que es caso que nos permite depurar más fácilmente:

```
C = [1, 4, 3]
B = [4, 1, 3]
```

Tras la aplicación del algoritmo se ha realizado el quicksort de las corchos tras haber ordenado las botellas y se muestran los resultados. En este caso es el resultado es el siguiente:

```
El corcho del tamaño 1 coincide con la botella del tamaño 1
El corcho del tamaño 3 coincide con la botella del tamaño 3
El corcho del tamaño 4 coincide con la botella del tamaño 4
```

El segundo caso de prueba es el primer caso que se encuentra en el anexo que consta de 10 corchos y 10 botellas:

```
C = [3, 5, 1, 7, 2, 10, 9, 4, 8, 6]
B = [6, 4, 3, 1, 9, 8, 10, 7, 5, 2]
```

```
El corcho del tamaño 1 coincide con la botella del tamaño 1
El corcho del tamaño 2 coincide con la botella del tamaño 2
El corcho del tamaño 3 coincide con la botella del tamaño 3
El corcho del tamaño 4 coincide con la botella del tamaño 4
El corcho del tamaño 5 coincide con la botella del tamaño 5
El corcho del tamaño 6 coincide con la botella del tamaño 6
El corcho del tamaño 7 coincide con la botella del tamaño 7
El corcho del tamaño 8 coincide con la botella del tamaño 8
El corcho del tamaño 9 coincide con la botella del tamaño 9
El corcho del tamaño 10 coincide con la botella del tamaño 10
```

El tercer y último caso de prueba es el segundo caso que se encuentra en el anexo que consta de 20 corchos y 20 botellas:

```
C = [12, 1, 3, 10, 11, 2, 7, 15, 18, 5, 9, 20, 19, 4, 14, 13, 17, 16, 6, 8]
B = [7, 13, 2, 19, 10, 4, 9, 20, 1, 5, 15, 17, 6, 18, 3, 14, 16, 8, 12, 11]
```

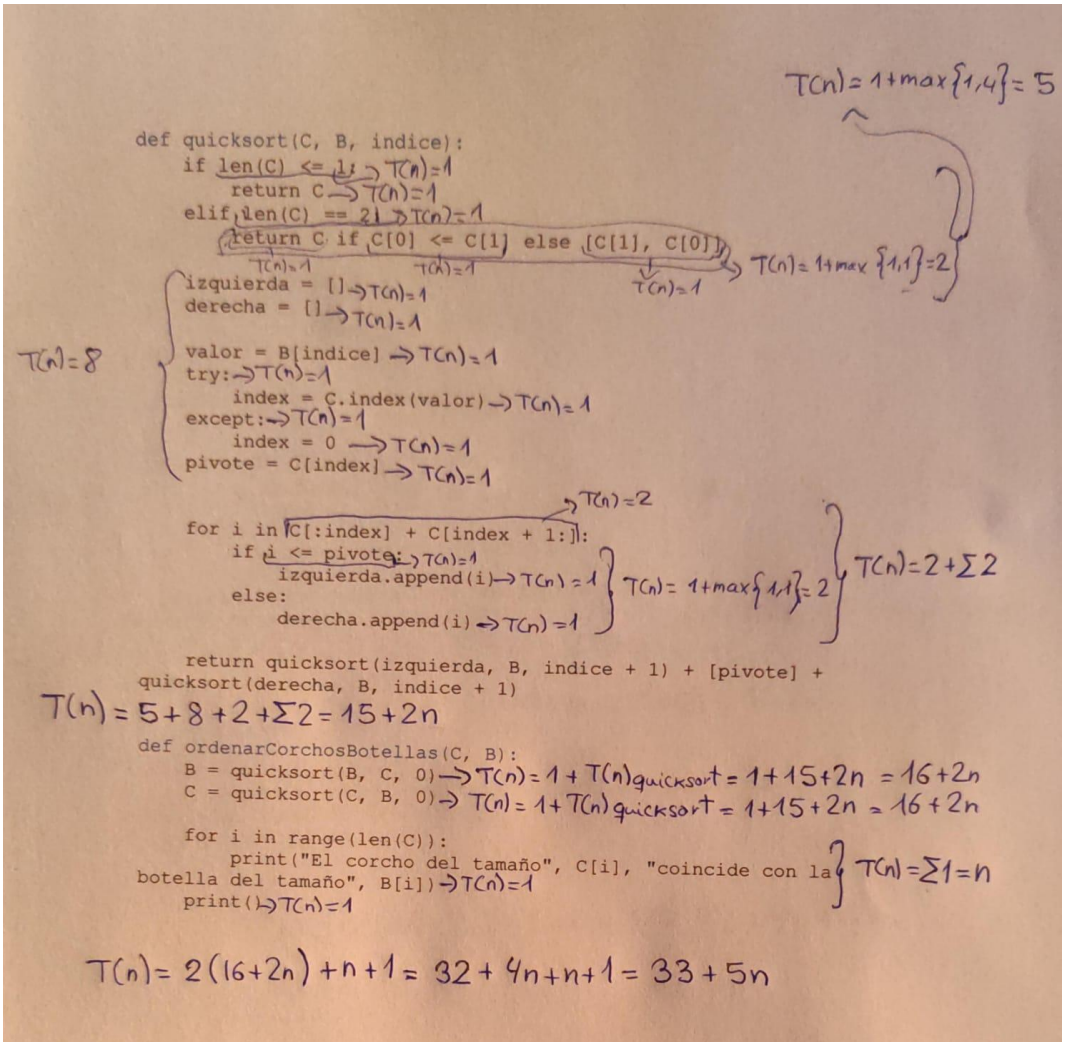
Tras ordenarlo mediante el algoritmo implementado el resultado es:

```
El corcho del tamaño 1 coincide con la botella del tamaño 1
El corcho del tamaño 2 coincide con la botella del tamaño 2
El corcho del tamaño 3 coincide con la botella del tamaño 3
El corcho del tamaño 4 coincide con la botella del tamaño 4
El corcho del tamaño 5 coincide con la botella del tamaño 5
El corcho del tamaño 6 coincide con la botella del tamaño 6
El corcho del tamaño 7 coincide con la botella del tamaño 7
El corcho del tamaño 8 coincide con la botella del tamaño 8
El corcho del tamaño 9 coincide con la botella del tamaño 9
El corcho del tamaño 10 coincide con la botella del tamaño 10
El corcho del tamaño 11 coincide con la botella del tamaño 11
El corcho del tamaño 12 coincide con la botella del tamaño 12
El corcho del tamaño 13 coincide con la botella del tamaño 13
El corcho del tamaño 14 coincide con la botella del tamaño 14
El corcho del tamaño 15 coincide con la botella del tamaño 15
El corcho del tamaño 16 coincide con la botella del tamaño 16
El corcho del tamaño 17 coincide con la botella del tamaño 17
El corcho del tamaño 18 coincide con la botella del tamaño 18
El corcho del tamaño 19 coincide con la botella del tamaño 19
El corcho del tamaño 20 coincide con la botella del tamaño 20
```

Cálculo de la complejidad:

Para calcular la complejidad del algoritmo quicksort implementado lo obtenemos a partir del coste computacional de un algoritmo de divide y vencerás. Para ello dividimos la lista de corchos en dos, el lado izquierdo y el lado derecho por ello el número de subcasos en este algoritmo es k=2. Como este algoritmo divide la lista de los corchos en dos esta lista puede ser dividida otra vez como máximo en 2 por lo tanto el tamaño de los subcasos es 2, es decir, que b=2. Como este algoritmo de quicksort lo llamamos 2 veces las llamadas recursivas aumentan al doble y en este caso serían para el algoritmo definitivo b=4. Tras realizar el cálculo de los costes de la parte del algoritmo que no es recursiva se saca un f(n) de 33+5*n. Por tanto, p(el orden de complejidad de la parte no recursiva) es igual 1. En definitiva este algoritmo tiene como T(N) a 4 * T(N/2) + 5 * n + 33. La complejidad del programa es de O(n) porque k(2) es menor b(4) elevado a p(1).

El cálculo de los costes del algoritmo sin incluir las llamadas recursivas:



Problema 7

Descripción del algoritmo:
El algoritmo sigue el mismo ejemplo de multiplicación de matrices visto en teoría aunque, en este caso, se hace la transpuesta de una matriz. Para ello, se contempla dos posibilidades: que el tamaño de la matriz sea potencia de dos y que no lo sea. Si fuese como en el último caso, el algoritmo añade filas y columnas de ceros tantos como sean necesarios hasta conseguir que el tamaño de la matriz sea potencia de dos.
Una vez hecho esto, se divide la matriz en 4 cuadrantes y se calcula su transpuesta recursivamente hasta obtener el caso base que es cuando la matriz es 2x2 y hacer el intercambio.
Al terminar, se vuelven a colocar los cuadrantes para devolver la matriz completa y transpuesta.

Casos de prueba:
El primer caso de prueba que realizamos es el probador del caso de base (matriz de 2x2) que utilizaremos para realizar la traspuesta de la matriz

```
Matriz original:
0 1
0 1
Matriz tranpuesta:
0 0
1 1
```

El segundo caso de prueba que realizamos es la matriz traspuesta de una matriz cuya longitud sea impar en este caso hemos utilizado el caso más sencillo posible de matriz impar(3x3)

```
Matriz original:
0 1 0
1 0 0
1 1 0
Matriz tranpuesta:
0 1 1
1 0 1
0 0 0
```

El tercer caso es para probar una matriz de longitud par, que cuya longitud sea mayor al caso de base. En este caso volvemos a utilizar una matriz sencilla de 4x4.

```
Matriz original:
0 1 0 1
1 0 0 1
1 1 0 1
0 0 1 1
Matriz tranpuesta:
0 1 1 0
1 0 1 0
0 0 0 1
1 1 1 1
```

Funcionamiento del algoritmo en el ejemplo: Lo que realiza el algoritmo es dividir la matriz en cuatro cuadrantes, como en este caso la matriz es una potencia de 2 no es necesario añadir ni filas ni columnas. En este caso tras realizar la división se puede observar que las cuatro nuevas matrices son como el caso base de 2x2 por lo que realizamos la matriz de cada una de ellas. Para realizar la traspuesta de una matriz de caso base como por ejemplo la segunda matriz obtenida de dividir la matriz 4x4 (valores de la matriz: 1ª fila 01 y 2ª fila 01) se intercambia el valor la 2ª columna de la 1ª fila(1) por el valor de la 1ª columna de la 2ª fila.

```
matriz cuadrante 1:
0 1
1 0
Matriz tranpuesta c1:
0 1
1 0
matriz cuadrante 2:
0 1
0 1
Matriz tranpuesta c2:
0 0
1 1
matriz cuadrante 3:
1 1
0 0
Matriz tranpuesta c3:
1 0
1 0
matriz cuadrante 4:
0 1
1 1
Matriz tranpuesta c4:
0 1
1 1
```

Tras ello juntamos otra vez las matrices traspuestas obtenidas en una para ello primero juntamos primero las dos primeras matrices (poniendo arriba la primera y abajo la segunda) y luego las dos restantes realizamos el mismo tipo de unión y por último juntamos el resultado de ambas uniones formándose así la matriz traspuesta resultante.

El cuarto caso de prueba es el primer caso presente en el anexo de una matriz(5x5)

```
Matriz original:
0 1 1 0 1
1 0 0 1 0
0 1 1 1 1
1 0 0 1 1
1 1 0 0 0

Matriz tranpuesta:
0 1 0 1 1
1 0 1 0 1
1 0 1 0 0
0 1 1 1 0
1 0 1 1 0
```

El quinto caso de prueba es el segundo caso presente en el anexo de una matriz (10x10)

```
Matriz original:
0 0 0 1 1 0 1 0 1 0
1 1 0 0 0 1 0 1 1 1
0 0 1 0 1 1 1 0 0 1
0 1 1 0 0 0 1 0 0 1
1 1 0 1 0 1 0 0 1 0
0 0 1 0 1 1 1 0 1 0
1 0 0 1 1 1 1 0 0 0
0 1 0 1 0 0 0 1 0 0
1 1 1 0 1 0 0 1 1 0
0 0 1 1 0 0 1 0 0 1
Matriz tranpuesta:
0 1 0 0 1 0 1 0 1 0
0 1 0 1 1 0 0 1 1 0
0 0 1 1 0 1 0 0 1 1
1 0 0 0 1 0 1 1 0 1
1 0 1 0 0 1 1 0 1 0
0 1 1 0 1 1 1 0 0 0
1 0 1 1 0 1 1 0 0 1
0 1 0 0 0 0 0 1 1 0
1 1 0 0 1 1 0 0 1 0
0 1 1 1 0 0 0 0 0 1
```

Cálculo de la complejidad:

Para el cálculo de la complejidad utilizamos el coste computacional para poder así obtener el T(n) del algoritmo. En este caso como dividimos la matriz en 4 cuadrantes el número de subcasos(k) es 4. Dependiendo del tamaño de la matriz que queramos transponer podemos tener una longitud mayor o menor por lo tanto este valor es una variable a la que denominamos n. Para realizar las matrices transpuestas necesitamos necesitamos que el tamaño más pequeño para trabajar en las matrices sea 2 por tanto este es el valor que usaremos como tamaño de los subcasos. Las veces que utilizamos el algoritmo de divide y vencerás tras haber realizado los costes de todos los algoritmos sin incluir las llamadas recursivas es de 9 * n^2 + 17 * n + 41. Por tanto, p(el orden de complejidad de parte no recursiva) es igual 2. En definitiva la T(n) de todo el ejercicio es 4 * T(N/2) + 9 * n^2 + 17 * n + 41. La complejidad del programa es de O(n^2 log (n)) porque k(4) es igual b(2) elevado a p(2).
El cálculo de los costes del algoritmo sin incluir las llamadas recursivas:

```
def dividir(matriz, inicioColumna, finalColumna, inicioFila, finalFila):
    m = []
    i = 0
    for columna in range(inicioColumna, finalColumna):
        m.append([])
        for fila in range(inicioFila, finalFila):
            m[i].append(matriz[columna][fila])
        i += 1
    return m
```

$T(n) = \sum(1 + \sum(1) + 1)$ $\quad T(n) = \sum_{fila=inicioFila}^{finalFila} 1$

$T(n) = 2 + \sum(1 + \sum(1) + 1) + 1 = 3 + n + n^2 + n = 3 + 2n + n^2$

```
def dividirEnCuadrantes(matriz):
    m1 = []
    m2 = []
    m3 = []
    m4 = []

    N = len(matriz)
    division = N // 2

    m1 = dividir(matriz, 0, division, 0, division)
    m2 = dividir(matriz, division, N, 0, division)
    m3 = dividir(matriz, 0, division, division, N)
    m4 = dividir(matriz, division, N, division, N)

    return (m1, m2, m3, m4)
```

$T(n)$ dividir $\times 4$

$T(n) = 8 + T(n)_{dividir} \times 4 = 8 + 4(3 + 2n + n^2) = 20 + 8n + 4n^2$

```
def colocarCuadrantes(t1, t2, t3, t4):
    N = len(t1)
    for columna in range(N):
        for fila in range(N):
            t1[columna].append(t2[columna][fila])
    for columna in range(N):
        for fila in range(N):
            t3[columna].append(t4[columna][fila])
    for columna in range(N):
        t1.append(t3[columna])
    return t1
```

$T(n) = \sum_{fila}^{N} 1$ $\quad \sum\sum 1$

$T(n) = \sum 1$ $\quad T(n) = \sum\sum 1$

$T(n) = \sum 1$

$T(n) = 2 + \sum 1 + \sum\sum 1 + \sum\sum 1 = 2 + n + n^2 + n^2 = 2 + n + 2n^2$

```
def transpuesta(matriz):
    if len(matriz) == 2:
        tmp = matriz[0][1]
        matriz[0][1] = matriz[1][0]
        matriz[1][0] = tmp
        return matriz
```

$T(n) = 1 + \max\{4, 0\}$

$T(n) = 5$

```
m1, m2, m3, m4 = dividirEnCuadrantes(matriz)
    t1 = transpuesta(m1)
    t2 = transpuesta(m2)
    t3 = transpuesta(m3)
    t4 = transpuesta(m4)
    t = colocarCuadrantes(t1, t2, t3, t4)
    return t
def esPotenciaDos(numero):
    x = log(numero, 2)
    return 0 == (x - floor(x))
def mostrarMatriz(matriz):
    N = len(matriz)

    for columna in range(N):
        for fila in range(N):
            print(matriz[columna][fila], '', end='')
        print()
def main(matriz):
    N = len(matriz)
    M = len(matriz)

    while not esPotenciaDos(M):
        matriz.append([0] * (M + 1))
        for columna in range(M):
            matriz[columna].append(0)

        M = len(matriz)

    t = transpuesta(matriz)

    if N != M:
        resultado = []
        for columna in range(N):
            resultado.append([])
            for fila in range(N):
                resultado[columna].append(t[columna][fila])
    else:
        resultado = t
    mostrarMatriz(resultado)
```

$T(n) = 1 + T(n)_{dividirCuadrantes} = 1 + 20 + 8n + 4n^2 = 21 + 8n + 4n^2$

$T(n)$ recursivo

$T(n) = 1 + T(n)_{colocarCuadrantes} = 1 + 2 + n + 2n^2 = 3 + n + 2n^2$

$T(n) = 1 + 9n + 6n^2 + 30$

$T(n) = 3$

$T(n) = 1 + \Sigma(\Sigma 1 + 1) = 1 + n^2 + n$

$T(n) = 1 + \sum 1 + 1 + T(n)_{esPotencia2} + \sum(T(n)_{potencia2} + 1 + \sum 1 + 1) =$
$= 1 + n + 1 + 3 + 3n + n + n^2 + n =$
$= 5 + 6n + n^2$

$= 1 + 9n + 6n^2 + 30 =$
$= 31 + 9n + 6n^2$

$T(n) = 1 + \max\{\sum(1 + \sum 1) + 1, 1\} = 2 + n + n^2$

$T(n)_{mostrarResultado} = 1 + n^2 + n$

$T(n)_{main} = 2 + 5 + 6n + n^2 + 31 + 9n + 6n^2 + 2 + n + n^2 + 1 + n^2 + n =$
$= 41 + 17n + 9n^2$