Documentación relacionada con la práctica 2

Índice:

1. Explicación código:

El código adjunto para la práctica 2 consiste en 6 clases, de las cuales explicaremos las más importantes algo más en profundidad que las clases que están de apoyo. Podríamos considerar como las clases más importantes complejidad, arbolNArio (la cual ya venía predefinida y hemos aplicado cambios para el toString) y compresiónImagen.

Clase CompresiónImagen:

Podríamos decir que es la clave realmente de toda esta práctica, en esta clase se encuentran los métodos para crear el árbol que almacenará la compresión de la matriz que compone una imagen binaria, cuyo método de creación se encuentra también en esta clase.

El primer método de esta clase a destacar sería el siguiente:

```
public static char[][] matriz(int b) {
    char[][] i = new char[b][b];
    Random r = new Random();
    for(int a=0;a<b;a++) {
        for(int j=0;j<b;j++) {
            if(r.nextBoolean())i[a][j] = 'B';
            else i[a][j] = 'N';
        }
    }
    return i;
}
```

Éste simplemente crea a partir de un número una matriz de ese orden rellenada de forma aleatoria con ‘B’ y ‘N’, representando el blanco y negro de una imagen binaria. Evidentemente devolverá esta matriz creada.

Antes de explicar el método que aplica divide y vencerás veremos otro del que hace uso:

```
public static boolean compararMatriz(char[][] c,int iF,int fF,int iC,int fC) {
    boolean seguir=true;
            char a=c[iC][iF];
            for(int i=iC; i<fC && seguir;i++) {
                for(int j=iF;j<fF&&seguir;j++) {
                    if(c[i][j]!=a) seguir=false;
                }
            }
    return seguir;}
```

Este método está encargado de comprobar si la zona (proporcionada por los atributos que se corresponden con inicio fila,iF, final fila, fF, inicio columna,iC, fin columna,fC) que le es designada de la matriz que se le proporciona. Devolverá true si todo el contenido de la matriz en esa zona es igual

El siguiente método a tener en cuenta sería el que propiamente implementa la estrategia divide y vencerás:

```
public static ArbolNArio<Character> compresion(char[][] c,ArbolNArio<Character> ar,int iF,int fF,int iC,int fC) {
    //caso base
    if(compararMatriz(c,iF,fF,iC,fC)) {
        ar=new ArbolNArio<Character>(c[iC][iF]);
        ar.setValor(c[iC][iF]);
     }
    else {
        ArbolNArio<Character>[] hijos = new ArbolNArio[4];
        ar.setHijos(hijos);
        ar.setValor('º');
        for(int i=0;i<4;i++) ar.getHijos()[i]=new ArbolNArio<Character>('º');
            ar.getHijos()[0]=compresion(c,ar.getHijos()[0],iF,fF/2,iC,fC/2);
            ar.getHijos()[1]=compresion(c,ar.getHijos()[1],iF+((fF-iF)/2),fF,iC,fC/2);
            ar.getHijos()[2]=compresion(c,ar.getHijos()[2],iF,fF/2,iC+((fC-iC)/2),fC);
            ar.getHijos()[3]=compresion(c,ar.getHijos()[3],iF+((fF-iF)/2),fF,iC+((fC-iC)/2),fC);
        }

    return ar;
}
```

Este método recibe como argumentos la matriz que antes hemos creado y que implementa la imagen que queremos comprobar; el árbol donde irá almacenada la compresión, y las correspondientes posiciones a comprobar de la matriz dependiendo del cuartil que le corresponda.

El caso base en este caso sería que compararMatriz, método anteriormente analizado, devolviera true, en cuyo caso el árbol se rellenaría con la primera posición de la zona designada pues serían todas las posiciones iguales de esa zona.

El caso general simplemente crea cuatro hijos para el nodo del árbol actual, los inicializa, y llama recursivamente al método con todos y cada uno de los cuatro hijos recién creados

Existe otro método en esta clase llamado complejidad, pero no requiere de un análisis tan extenso pues lo único que hacer es las llamadas a estos tres métodos para poder comprobar su correcto funcionamiento. Cierto es que cabe a destacar la comprobación de que el número proporcionado para crear la matriz sea potencia de 2:

```
if(Math.log(b)/Math.log(2) % 1 !=0)  System.out.println("El número ha de ser potencia de 2");
```

Clase Complejidad:

Esta clase cuenta con un único método que se encarga de calcular la complejidad empírica, para ello ejecutamos la parte de divide y vencerás con un incremento en los valores del orden de la matriz hasta 1024x1024, con lo que sacaremos el tiempo que tarda en ejecutarse en nanosegundos (calculamos a partir de este tiempo lo que tarde en ms para mayor exactitud) y lo pasamos como string a una cola que luego nos permitirá exportarlo a un documento en csv para su mejor visionado.

```
public static void calculo() {
    Queue<String> q= new LinkedList<String>();
    for(int j=1;j!=11;j++) {
    char[][] i = compresionImagen.matriz((int)Math.pow(2, j));
    ArbolNArio<Character> ar = new ArbolNArio<Character>('º');
    long inicio = System.nanoTime();
    compresionImagen.compresion(i, ar,0,i.length,0,i.length);
    long elapsed=System.nanoTime()-inicio;

    double elapsedm=elapsed/(1*Math.pow(10, 6));


    q.add((int)(Math.pow(2, j))+"x"+((int)(Math.pow(2, j)))+";"+elapsed+";"+elapsedm+"\n");
    }
    CrearCSV.iniciar(q);
    System.out.println("Resultados exportados a csv");
}
```

Clase ArbolNArio:

De este método ya teníamos prácticamente todos los métodos definidos, únicamente hemos tenido que concretar el toString, para el que báicamente hemos hecho un recorrido preorden al que le pasas una variable int que representará la profundidad en la que se encuentra en ese momento. El método devolverá un String, donde es exportado el valor del nodo que en ese momento esté recorriendo el método más tanta separación como sea necesaria por la profundidad.

```
public String toString(int p) {
    String a="";
    ArbolNArio ar = this;
    if(ar!=null) {
        for(int i =0; i<p;i++)a+="\n";
        for(int i =0; i<p;i++)a+="\t";
        p++;
        a= a+ar.getValor();
        if(ar.getHijos()!=null) {
            for(int i=0; i<ar.getHijos().length;i++) {
                if(ar.getHijos()[i]!=null) a+=ar.getHijos()[i].toString(p);
            }
            }
        }
    return a;
}
```

Del resto de clases, CrearCSV y main, tal vez lo más destacable sea la clase crear CSV cuya función es escribir lo mandado en una cola por el método complejidad en un Word; pero su interés reside en como se ha intentado escribir con comandos de Excel más que en su propia programación, pues no deja de ser un scanner encargado de escribir en un documento proporcionado.

La clase leer venía ya predefinida y es usada en múltiples ocasiones para recoger datos que se piden al usuario.

La clase main simplemente es un menú proporcionado al usuario para que se pueda probar todo lo que el programa ofrece

En cualquier caso todo está bien documentado dentro del código por si interesase alguna función o método en concreto.

2. Complejidad:

Analizaremos la complejidad empírica y teórica de todo lo relacionado con el método de divide y vencerás.

2.1 **Teóricamente:**

Calcularemos la complejidad teórica del método propio que usa la estrategia divide y vencerás, además de los métodos necesarios para conocerla

Método compresión:

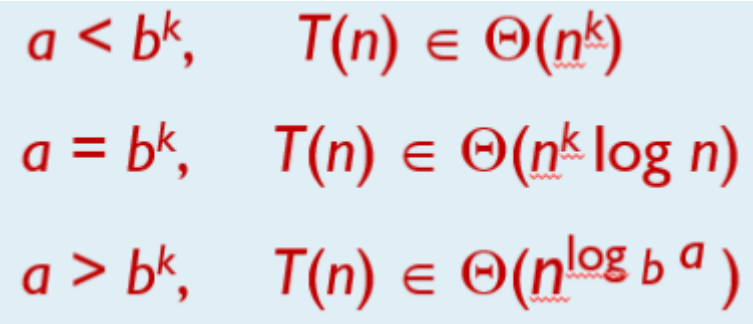
```
public static ArbolNArio<Character> compresion(char[][] c,ArbolNArio<Character> ar,int iF,int fF,int iC,int fC) {
    if(compararMatriz(c,iF,fF,iC,fC)) { //caso base, la sección comparada es igual en contenido
        ar=new ArbolNArio<Character>(c[iC][iF]);
        ar.setValor(c[iC][iF]);
     }
    else { //Caso externo
        ArbolNArio<Character>[] hijos = new ArbolNArio[4];
        ar.setHijos(hijos);
        ar.setValor('º');
        for(int i=0;i<4;i++) ar.getHijos()[i]=new ArbolNArio<Character>('º');
            ar.getHijos()[0]=compresion(c,ar.getHijos()[0],iF,fF/2,iC,fC/2);
            ar.getHijos()[1]=compresion(c,ar.getHijos()[1],iF+((fF-iF)/2),fF,iC,fC/2);
            ar.getHijos()[2]=compresion(c,ar.getHijos()[2],iF,fF/2,iC+((fC-iC)/2),fC);
            ar.getHijos()[3]=compresion(c,ar.getHijos()[3],iF+((fF-iF)/2),fF,iC+((fC-iC)/2),fC);
        }

    return ar;
}
```

Vamos a calcular la complejidad de este método usando el teorema maestro, para ello debemos conocer a,b y k:

- a=4, número de veces que llamamos de nuevo al método de forma recursiva
- b=4, ya que el problema se va diviviendo en cuartos (vamos dividiendo la matriz o la sección que usamos de la matriz de cuatro en cuatro
- k=2, Dado por el método compararMatriz, ya que de ser por el resto del mismo método, k sería 0 (Posteriormente comprobaremos que compararMAtriz es O(n^2))

Sabiendo entonces por el teorema maestro:



Podemos comprobar que a<b^k, por lo que la complejidad de este método es:

$$T(n) \epsilon \Theta(n^2)$$

Método compararMatriz:

```
public static boolean compararMatriz(char[][] c,int iF,int fF,int iC,int fC) {
    boolean seguir=true;
            char a=c[iC][iF];
            for(int i=iC; i<fC && seguir;i++) {
                for(int j=iF;j<fF&&seguir;j++) {
                    if(c[i][j]!=a) seguir=false;
                }
            }
    return seguir;}
```

Podemos comprobar rápidamente, al ser dos bucles anidados que la complejidad va a ser igual a **O(n^2),** siendo n el grado de la matriz, ya que la vamos a recorrer bidimensionalmente (y al ser una matriz cuadrado el número de filas y de columnas es el mismo, que viene dado por el orden de la matriz).

Con esto justificamos la k anterior.

2.2 Empíricamente:

Para calcular la complejidad empírica haremos uso de las mencionadas anteriormente clase complejidad y crarCSV, que entre las dos nos permitirán saber cuanto tiempo tarda en ejecutarse la compresión de la imagen y exportarlo a un csv para poder después con este crear un Excel y analizar los resultados mediante su tabla y una gráfica.

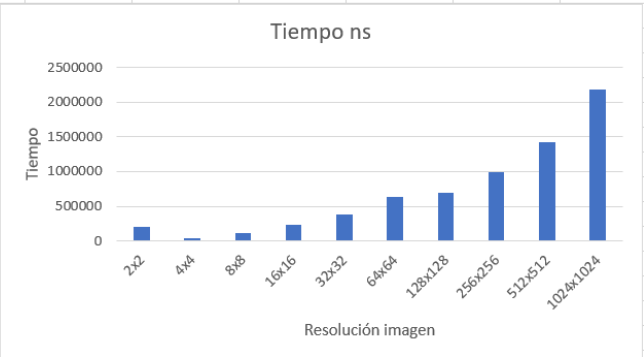
Como prueba extra adjuntaremos también imágenes de las dos tablas, tanto la de un compañero como la del otro y así poder analizar las diferencias entre ordenadores:

Tabla y gráfica Diego Cordero:

	A	B	C
1	Resolucion imagen	Tiempo ns	Tiempo ms
2	2x2	112600	0,1126
3	4x4	25300	0,0253
4	8x8	57700	0,0577
5	16x16	139300	0,1393
6	32x32	232600	0,2326
7	64x64	345900	0,3459
8	128x128	307400	0,3074
9	256x256	428600	0,4286
10	512x512	640700	0,6407
11	1024x1024	914100	0,9141

Tabla y gráfica Francisco Duque:

Resolucion imagen	Tiempo ns	Tiempo ms
2x2	201300	0,2013
4x4	40800	0,0408
8x8	119200	0,1192
16x16	233900	0,2339
32x32	377000	0,377
64x64	640500	0,6405
128x128	693900	0,6939
256x256	993300	0,9933
512x512	1429700	1,4297
1024x1024	2189200	2,1892



La gráfica está hecha en base a los ns debido a que al ser un valor más alto es más fácil de visualizar

Asignatura:

Metodología de la programación

Autores:

Francisco Duque Melnychenko

Diego Cordero Contreras