Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos - Grado en Ingeniería Informática

Prueba Final – Curso 2013/14

35 % de la nota final (3,5 puntos sobre 10)

9 de mayo de 2013. Duración: 2 horas

El código pedido deberá estar escrito en Java

Soluciones

Ejercicio 1 Elemento mayoritario en un array [1.75 puntos]

Se pide implementar un algoritmo eficiente basado en la estrategia de **divide y vencerás** para determinar si un array de tamaño n contiene un "elemento mayoritario", que es aquel que aparece más de $n/2$ veces. Por ejemplo, un array de tamaño 6 o 7 tiene un elemento mayoritario si dicho elemento aparece al menos 4 veces. En vuestra implementación **NO SE PERMITE ORDENAR EL ARRAY**. Indicad además la fórmula recursiva del coste del algoritmo, y el orden de complejidad.

Solución:

Empezamos dividiendo el problema de tamaño n en dos subproblemas idénticos al original pero de tamaño $n/2$, si n es par; o de tamaños $(n-1)/2$ y $(n+1)/2$, si n es impar. En ambos casos, para que un array tenga un elemento mayoritario éste debe ser también mayoritario en alguna de las dos "mitades". Por tanto, si hay un elemento mayoritario en una mitad, hay que contar el número de apariciones de dicho elemento en la otra mitad del array. Esto lo aprovechamos para construir el caso recursivo. Por otro lado, el caso base se da cuando solo hay un elemento en el array (o "subarray"), en cuyo caso siempre será mayoritario.

A continuación se muestra una posible solución, donde el método recursivo devuelve un valor booleano (si el array tiene un elemento mayoritario), y además usa dos variables pasadas por referencia (en este caso, dos simples arrays de un elemento), para devolver el elemento mayoritario, en caso de existir, y el número de apariciones de éste en el array. El código también incluye un programa principal (que no se pedía en el ejercicio).

```java
import java.util.Scanner;

public class Mayoritario{

   public static boolean hayMayoritario(int[] vector, int ini, int fin,
                  int[] elemento_mayoritario, int[] n_apariciones){

      if (ini==fin){ // caso base
         elemento_mayoritario[0] = vector[ini];
         n_apariciones[0] = 1;
         return true;
      }
      else{
         int mitad = (ini+fin)/2;
         boolean hay_mayoritario = false;

         if (hayMayoritario(vector,ini,mitad,elemento_mayoritario,n_apariciones)){
            for(int i=mitad+1;i<=fin;i++)
               if (vector[i]==elemento_mayoritario[0])
                  n_apariciones[0]++;

            if (n_apariciones[0]>((fin-ini+1)/2))
               hay_mayoritario = true;
         }

         if (!hay_mayoritario)
            if (hayMayoritario(vector,mitad+1,fin,elemento_mayoritario,n_apariciones)){
               for(int i=ini;i<=mitad;i++)
                  if (vector[i]==elemento_mayoritario[0])
                     n_apariciones[0]++;

               if (n_apariciones[0]>((fin-ini+1)/2))
                  hay_mayoritario = true;
            }

         return hay_mayoritario;
      }
   }

   public static void main(String argv[]){
      Scanner sc = new Scanner ( System.in );
      int n = sc.nextInt();

      int vector[] = new int[n];
      for(int i=0;i<n;++i){
         vector[i] = sc.nextInt();
      }

      int elemento_mayoritario[] = new int[1];
      int n_apariciones[] = new int[1];

      if(hayMayoritario(vector,0,n-1,elemento_mayoritario,n_apariciones))
         System.out.println("Elemento mayoritario: " + elemento_mayoritario[0]);
      else
         System.out.println("No hay elemento mayoritario");
   }
}
```

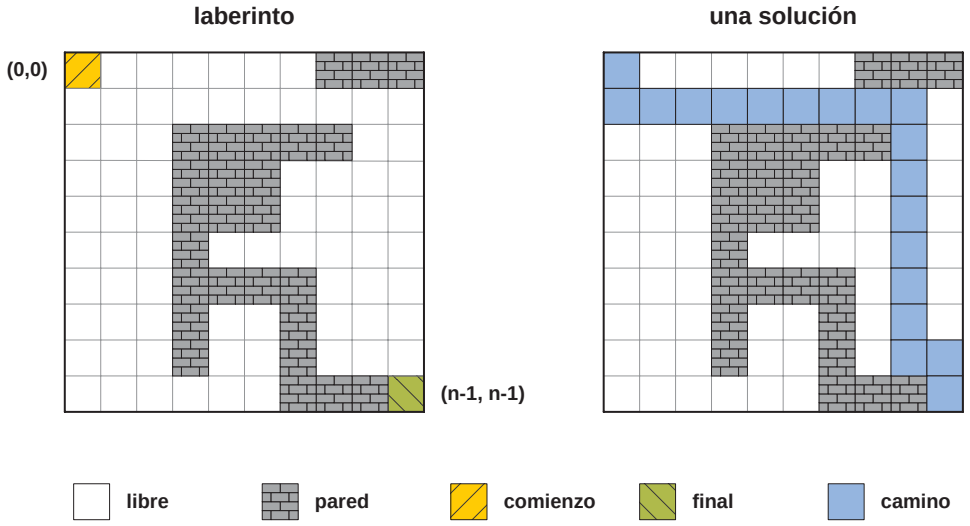
La función de coste, en el peor caso, para este algoritmo es:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + n & \text{si } n > 1 \end{cases}$$

Por el teorema maestro, $T(n) \in \Theta(n \log n)$.

Ejercicio 2 Camino más corto a través de un laberinto [1.75 puntos]

Considérese un tablero de $n \times n$ celdas que contiene un laberinto. Inicialmente las celdas pueden estar libres o pueden contener paredes. Se pide implementar un algoritmo basado en la técnica de ***backtracking*** para hallar un camino más corto desde la celda $(0, 0)$ hasta la $(n-1, n-1)$. El programa imprimirá el laberinto y su solución por pantalla. La siguiente figura ilustra un ejemplo donde el camino hallado tiene longitud 19:



Solución:

En este ejercicio hay que tener en cuenta:

- La solución no es ni una permutación ni un subconjunto, por lo que no podemos usar estos esquemas.
- Cada nodo del árbol de recursión tiene 4 posibles hijos (uno por cada dirección en la que puedes avanzar creando la solución parcial, es decir, el camino).
- No debemos parar al encontrar una solución válida, ya que queremos la óptima. Esto implica que no usaremos una variable "éxito".

- Usaremos una variable para llevar la cuenta de la longitud del camino que vamos construyendo (en el código de abajo, `pasos`)
- Al ser un problema de optimización, necesitaremos una variable para almacenar el valor óptimo hallado hasta el momento (la longitud del camino más corto hallado), y otra variable (en este caso, matriz) donde almacenamos dicho camino más corto, y que será del mismo tipo que la solución parcial. En el código de abajo son `pasosOptimo` y `laberintoOptimo`, respectivamente, ambas pasadas por referencia. Se actualizarán cuando un camino llegue a la celda final (y la nueva solución parcial sea mejor que la hallada con anterioridad).
- Además de podar el árbol de recursión cuando el camino no es válido, podemos podar el árbol de recursión cuando la solución parcial no pueda mejorar a la óptima hasta ese momento (`pasos<pasosOptimo[0]`, en la línea 63).
- El resto del código es prácticamente idéntico al visto en clase de teoría, que se encuentra en las transparencias de la asignatura.

A continuación se muestra una posible solución (el código también incluye un programa principal, que no se pedía en el ejercicio):

```
1  import java.io.*;
2
3  public class LaberintoOptimo
4  {
5     public static void main(String[] args) throws Exception{
6        BufferedReader cin = new BufferedReader(new InputStreamReader(System.in));
7        int n = Integer.parseInt(cin.readLine());
8
9        char[][] laberinto = new char[n][n];
10
11       for (int i=0; i<n; i++){
12          String linea = cin.readLine();
13          for (int j=0; j<n; j++)
14             laberinto[i][j] = linea.charAt(j);
15       }
16
17       buscaCamino(laberinto);
18    }
19
20    public static void imprimir(char[][] laberinto){
21       for(int i=0; i<laberinto.length+2; i++)
22          System.out.print ("*");
23       System.out.println();
24
25       for(int i=0; i<laberinto.length; i++){
26          System.out.print("*");
27          for(int j=0; j<laberinto.length; j++)
28             System.out.print (laberinto[i][j]);
29
30          System.out.println("*");
31       }
32
33       for(int i=0; i<laberinto.length+2; i++)
34          System.out.print("*");
35       System.out.println();
36    }
37
```

```
38    public static void buscaCamino(char[][] laberinto){
39       int[] incrX = new int[] {0, 1,  0, -1};
40       int[] incrY = new int[] {1, 0, -1,  0};
41
42       int n = laberinto.length;
43       char[][] laberintoOptimo = new char[n][n];
44       int[] pasosOptimo = new int[1];
45
46       pasosOptimo[0] = (n*n+1);
47
48       laberinto[0][0] = '.';
49       buscar(laberinto.length, 0, 0, 1, laberinto, laberintoOptimo, pasosOptimo, incrX, incrY);
50
51       imprimir(laberintoOptimo);
52       System.out.println(pasosOptimo[0]);
53    }
54
55    public static void buscar(int n, int x, int y, int pasos, char[][] laberinto,
56             char[][] laberintoOptimo, int[] pasosOptimo, int[] incrX, int[] incrY){
57
58       for(int k=0; k<4; k++){
59          int coordX = x + incrX[k];
60          int coordY = y + incrY[k];
61
62          if((coordX>=0) && (coordX<n) && (coordY>=0) && (coordY<n) && (laberinto[coordY][coordX] == ' ')
63             && (pasos<pasosOptimo[0])){  // (pasos<pasosOptimo[0]-1) también es válido
64             laberinto[coordY][coordX] = '.';
65             pasos++;
66
67             if ((coordX==n-1) && (coordY==n-1) && (pasos < pasosOptimo[0])){
68                pasosOptimo[0] = pasos;
69
70                for (int i=0; i<n; i++)
71                   for (int j=0; j<n; j++)
72                      laberintoOptimo[i][j] = laberinto[i][j];
73             }else{
74                buscar(n, coordX, coordY, pasos, laberinto, laberintoOptimo,
75                   pasosOptimo, incrX, incrY);
76             }
77
78             laberinto[coordY][coordX] = ' ';
79             pasos--;
80          }
81       }
82    }
83 }
```

Nota: en este código se supone que la entrada es una matriz de $n \times n$ caracteres. Los espacios en blanco representan celdas libres. El carácter '$*$' representa una pared, y el '.' se utiliza para marcar por dónde va el camino. Opcionalmente, el programa imprime un borde alrededor del laberinto.