



## Grado en Informática Algorítmica

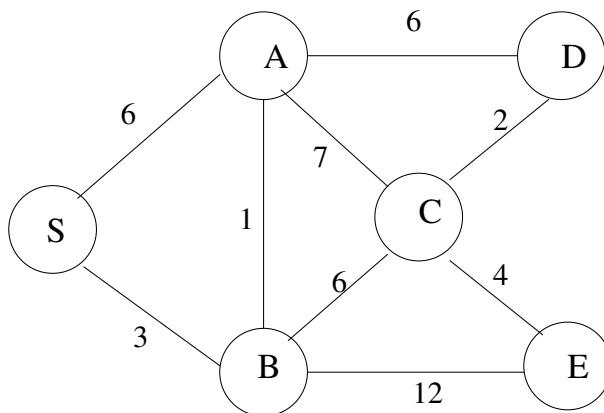
Curso 2018/2019. Convocatoria extraordinaria

28 de junio de 2019

1. (2 puntos) Calcula el orden de eficiencia de la siguiente función recursiva:

```
int calcula(int ini, int fin) {  
  
    if (ini < fin)  
        return calcula(ini, fin-1)+calcula(ini+1, fin)+calcula(ini+1, fin-1);  
    else return ini-fin;  
}
```

2. (2 puntos) Dado el grafo de la figura, donde aparecen las distancias entre los vértices adyacentes, se desea calcular cuáles son los caminos mínimos entre el vértice  $S$  y cada uno de los demás vértices. Describid detalladamente un algoritmo para resolver ese problema en general y aplicadlo paso a paso para ese grafo en concreto.



3. (2 puntos) Diseña un algoritmo para calcular  $x^n$ ,  $n \in N$ , con un coste  $O(\log n)$  en términos del número de multiplicaciones.
4. (2 puntos) Consideremos un mapa formado por  $N$  países. Queremos viajar entre países. Cada vez que atravesemos una frontera tenemos que pagar una tasa. Todas las tasas son conocidas. Construid un algoritmo de programación dinámica que determine el coste del viaje más barato entre dos países dados.



5. (2 puntos) Diseñad un algoritmo de exploración de grafos que, dado un número natural  $n$ , devuelva todas las formas posibles en que un conjunto *ascendente* de números positivos sume exactamente  $n$ . Por ejemplo, si  $n = 10$ , la salida debería ser:

1+2+3+4  
1+2+7  
1+3+6  
1+4+5  
1+9  
2+3+5  
2+8  
3+7  
4+6  
10

Especificad claramente, además del algoritmo, la representación de la solución, las restricciones explícitas e implícitas, así como las posibles cotas a utilizar.

**Duración del examen:** 2 horas y 30 minutos.

1. (2 puntos) Calcula el orden de eficiencia de la siguiente función recursiva:

```
int calcula(int ini, int fin) {
    if (ini < fin)
        return calcula(ini, fin-1)+calcula(ini+1, fin)+calcula(ini+1, fin-1);
    else return ini-fin;
}
```

Obtenemos primera, siendo  $u = fin - ini$  el tamaño del problema, la función que representa el tiempo de ejecución. Es claro que:

$$T(u) = \begin{cases} 1 & \text{si } fin = ini \\ 2T(u-1) + T(u-2) & \text{si } fin > ini \end{cases}$$

Procedemos a resolver la recurrencia.

$$t_u = 2t_{u-1} + t_{u-2}$$

Es una recurrencia homogénea (luego bastará con dar solución a la ecuación cuadrática)

$$t_u - 2t_{u-1} - t_{u-2} = 0$$

$$\text{Ec. característica: } x^2 - 2x - 1 = 0 \Leftrightarrow x = \frac{2 \pm \sqrt{4 + 4}}{2} = \frac{2 \pm \sqrt{8}}{2} = \frac{1 \pm \sqrt{2}}{2} = \frac{1 \pm 2\sqrt{2}}{2} = 1 \pm \sqrt{2}$$

Por tanto, las raíces son:

$$\begin{aligned} r_0 &= 1 + \sqrt{2} & u_0 &= 1 \\ r_1 &= 1 - \sqrt{2} & u_1 &= 1 \end{aligned}$$

Es decir, mi recurrencia tiene como solución la sucesión

$$\{c_0(1+\sqrt{2})^u + c_1(1-\sqrt{2})^u\}$$

luego  $T(u) \in O((1+\sqrt{2})^u)$

3. (2 puntos) Diseña un algoritmo para calcular  $x^n$ ,  $n \in N$ , con un coste  $O(\log n)$  en términos del número de multiplicaciones.

Usaremos un algoritmo Divide y Conquerás pues

$$x^u = x^{\frac{u}{2}} \cdot x^{\frac{u}{2}}$$

Luego la combinación será la multiplicación. Como siempre calcularemos la misma potencia al dividir el exponente en 2, sólo la calculamos 1 vez.

double potencia (double x, int u)

if ( $u=0$ ) retorna 1;

if ( $u=1$ ) retorna x;

int potencia = potencia ( $x, \frac{u}{2}$ );

retorna potencia · potencia;

{

Vemos el orden de eficiencia siendo  $\omega$  el tamaño del problema y  $T(u)$  la función que modela el tiempo de ejecución.

$$T(u) = \begin{cases} 1 & u=0 \vee u=1 \\ T\left(\frac{u}{2}\right)+1 & u>0, u \neq 1 \end{cases}$$

Resolvemos la recurrencia  $T(u)=T\left(\frac{u}{2}\right)+1$  usando el cambio de variable  $u=2^w$ .

$$T(2^w) = T(2^{w-1}) + 1 \Leftrightarrow t_{2^w} - t_{2^{w-1}} = 1$$

Parte homogénea:  $t_{2^w} - t_{2^{w-1}} = 0 \Leftrightarrow x-1=0 \Leftrightarrow x=1$  por tanto  $x=1$  es raíz del polinomio característico.

Parte no homogénea:  $1 = 1^w \cdot w^0$  luego 1 es raíz multiplicidad 1

Por tanto, una ecuación característica es equivalente a  $(x-1)^w=0$  luego  $1 \cdot c_0 + w \cdot c_1 = c_0 + w c_1 =$

$$= c_0 + c_1 \log_2 u$$

Por tanto,  $T(u) \in O(\log u)$

4. (2 puntos) Consideremos un mapa formado por  $N$  países. Queremos viajar entre países. Cada vez que atravesemos una frontera tenemos que pagar una tasa. Todas las tasas son conocidas. Construid un algoritmo de programación dinámica que determine el coste del viaje más barato entre dos países dados.

Floyd

Dijkstra con la mitad de la tasa

Por simplicidad consideraremos que todos los países están conectados con todos (Tenríamos una matriz  $N \times N \times N$ )

Gasign lo haríamos  $N \times N$  veces para  $N^2$  y solo sería cuadrática en dimensiones

Vemos a suponer que las tasas entre países están regidas por un vector  $\mathbf{t}$  (labeled 6)

donde  $t_{[i,j]}$ ,  $t_i$ , es la tasa de pasar por ese país

Vemos que el problema se puede dividir y cumple el principio de optimidad de Bellman Sea  $S$  una solución, es decir, el conjunto de países por los que pasamos para obtener el resultado; sea  $p_0$  y  $p_n$  los países de origen y destino. Sea  $p_m$  un país intermedio,  $p_0-p_m-p_n$  es óptima?

Supongamos que no lo fuera entonces existe otra solución  $(p_0-p_m')$  con menor pago de tasas, luego si la juntamos con  $p_m-p_n$  obtendremos una solución del problema mejor a la que tenemos!!

De forma análoga se obtiene la otra partición

Vemos cómo expresaríamos los datos. Consideraremos una matriz  $N \times N$  donde cada posición  $(i,j)$  representará la cantidad de dinero pagado por tasas habiendo pasado por el país anterior o no

Floyd

5. (2 puntos) Diseñar un algoritmo de exploración de grafos que, dado un número natural  $n$ , devuelva todas las formas posibles en que un conjunto *ascendente* de números positivos sume exactamente  $n$ . Por ejemplo, si  $n = 10$ , la salida debería ser:

```
1+2+3+4
1+2+7
1+3+6
1+4+5
1+9
2+3+5
2+8
3+7
4+6
10
```

No está bien

Especificad claramente, además del algoritmo, la representación de la solución, las restricciones explícitas e implícitas, así como las posibles cotas a utilizar.

Queremos la técnica BackTrack. Hablaremos por ello de los *cotas globales* y *locales*

- *Cota global*: queremos el valor que queremos sumar

- *Cota local*: queremos una vez el valor tomado como decisión para llegar a esta situación, si supera la cota global podemos esa cota pues no podemos repetir valores ni sumar valores anteriores por tanto siempre será menor que cualquier solución de esa rama.

Hablaremos de las *restricciones*:

- *Explícitos*:  $u^k$  positivos menores o iguales que la suma

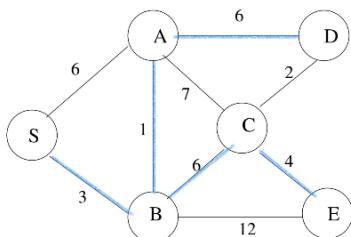
- *Implícitos*: siempre tiene sentido, no podemos repetir sumando y tomar uno de otros

Las soluciones son vectores de  $0 \dots n$  de tamaño  $k$  que representan los únicos elegidos

Algoritmo

```
void SumaInterior ( int u , vector<vector<int>> &ue , int vector , int u )
{
    if ( suma ( ue [vector] ) == u ) { vector ++; Procesa ( ue , vector ); } si ya está u lo añade,
    if ( vector != u ) { // si vector == u solo hay mas sumas ← uelo le sacado de la cisterna
        while ( k < u + 1 )
            ue [vector] [u] = 1;
        if ( !Factible ( ue , vector ) )
            ue [vector] [u] = 0;
        SumaInterior ( u , ue , vector , ++u );
    }
    else if ( suma ( ue , vector ) != u )
        SumaInterior ( u , ue , vector , ++u );
}
```

2. (2 puntos) Dado el grafo de la figura, donde aparecen las distancias entre los vértices adyacentes, se desea calcular cuáles son los caminos mínimos entre el vértice  $S$  y cada uno de los demás vértices. Describid detalladamente un algoritmo para resolver ese problema en general y aplicadlo paso a paso para ese grafo en concreto.



desde  
hacer tabla con grafo

El algoritmo a aplicar sería un Greedy y será Dijkstra de manera que dado un punto, aplicarlo bastará con pasar el nodo de fin y una matriz de caminos.

```

void Dijkstra (const int** m, vector<vector<int>> &caminos, Nodo final, Nodo inicio)
{
    if (caminos.empty() == final) return;
    
```

while (adyacentes (inicio))

int minimo = minimoPaso (adyacentes)

caminos.add (adyacente)

Dijkstra (m, caminos, final, adyacentesFinal)

{

Sacar\_camino\_directo (camino, inicio, final) // va desde final a inicio consiguiendo el camino.

{

Algoritmo real

```

void TodosLosCaminos (const int** m, vector<vector<vector<int>>> &caminos, Nodo fin, Nodo ini, vector<vector<int>> &sol)
    
```

for (int i=0; i<caminos.size(); ++i)

vector<vector<int>> camino;

Dijkstra (m, camino, fin, nodos[i]);

sol.push\_back (camino);

{

{