



Grado en Informática Algorítmica

Curso 2020/2021. Convocatoria ordinaria de junio
15 de junio de 2021

1. (2 puntos) Calcular el orden de eficiencia en notación $O(\cdot)$ de un algoritmo recursivo cuya expresión de tiempo es:

$$T(n) = 3T\left(\frac{n}{2}\right) + 4T\left(\frac{n}{4}\right) + n^2$$

2. (2 puntos) Dado un conjunto de n números enteros positivos $C = \{c_1, c_2, \dots, c_n\}$ y dado un entero positivo N , se quiere encontrar un subconjunto de C , $\{y_1, \dots, y_m\}$ que maximice $\prod_{i=1}^m y_i$ (se maximice el producto de los números del subconjunto), sujeto a la restricción $\prod_{i=1}^m y_i \leq N$. Nota: m NO es un parámetro de entrada.

Diseñad un algoritmo de vuelta atrás para resolver este problema.

Por ejemplo, para $n = 4$, $N = 37$ y $C = \{4, 4, 12, 3\}$, la solución es emplear el subconjunto $\{12, 3\}$ con un valor de producto igual a 36.

3. (2 puntos) Estamos en una posición (x, y) de un laberinto bidimensional representado por una matriz M de f filas y c columnas. De cada casilla se puede viajar a cualquiera de las colindantes con la casilla actual, en la misma fila o la misma columna, siempre que la casilla destino sea transitable. Los posibles valores de las casillas son $M(i, j) = t$ (transitable), $M(i, j) = n$ (no transitable) y $M(i, j) = s$ (salida). Sabiendo que disponemos del mapa del laberinto a priori, y que pueden existir varias salidas del mismo, plantee un algoritmo eficiente que devuelva el camino a la salida más próxima a la casilla inicial (x, y) . NOTA: El coste de viajar de una casilla a otra es siempre 1.
4. (2 puntos) Disponemos de n tipos de monedas, las monedas de tipo i tienen un valor de $v[i]$. La cantidad de monedas disponibles de tipo i es igual a $c[i]$. Se quiere devolver una cantidad exacta M utilizando el menor número posible de monedas. Diseñad un algoritmo de programación dinámica que determine el número óptimo de monedas a usar. Aplicadlo para resolver el siguiente caso del problema, con $n = 3$, construyendo la tabla correspondiente:

$$v[] = (2, 4, 6), c[] = (3, 1, 2), M = 8$$

5. (2 puntos) Diseñad un algoritmo que determine si un cierto número natural N puede expresarse o no como producto de tres números naturales consecutivos, $Y * (Y + 1) * (Y + 2)$, con un orden de eficiencia lo mejor posible.

Duración del examen: 2 horas y 30 minutos.

1. (2 puntos) Calcular el orden de eficiencia en notación $O(\cdot)$ de un algoritmo recursivo cuya expresión de tiempo es:

$$T(n) = 3T\left(\frac{n}{2}\right) + 4T\left(\frac{n}{4}\right) + n^2$$

Para ello, usaremos el cambio de variable $u = 2^w$ obteniendo la siguiente recurrencia.

$$T(2^w) = 3T(2^{w-1}) + 4T(2^{w-2}) + 4^w$$

Parte homogénea:

$$\ell_w \cdot 3\ell_{w-1} - 4\ell_{w-2} = 0 \rightarrow \text{Ecuación característica: } \ell^2 - 3\ell - 4 = 0 \Leftrightarrow \ell = \frac{3 \pm \sqrt{9+16}}{2} \begin{cases} \ell_1 = 4 \\ \ell_2 = -1 \end{cases}$$

Parte no homogénea: $4^w = b^w \cdot p(u) = 4^w \cdot u^0$

Por tanto, vemos que nuestro problema se reduce a 2 partes

$$r_0 = 4, u_0 = 2$$

$$r_1 = 1, u_1 = 1$$

$$T(2^w) = c_0 4^w + c_1 w 4^w + s_2$$

Deshaciendo el cambio de variable:

$$T(u) = c_0 4^{\log_2 u} + c_1 \log_2 u 4^{\log_2 u} + s_2 = c_0 u^2 + c_1 u^2 \log(u) + s_2$$

luego $T(u) \in O(u^2 \log(u))$

2. (2 puntos) Dado un conjunto de n números enteros positivos $C = \{c_1, c_2, \dots, c_n\}$ y dado un entero positivo N , se quiere encontrar un subconjunto de C , $\{y_1, \dots, y_m\}$ que maximice $\prod_{i=1}^m y_i$ (se maximice el producto de los números del subconjunto), sujeto a la restricción $\prod_{i=1}^m y_i \leq N$. Nota: m NO es un parámetro de entrada.

Diseñad un algoritmo de vuelta atrás para resolver este problema.

Por ejemplo, para $n = 4$, $N = 37$ y $C = \{4, 4, 12, 3\}$, la solución es emplear el subconjunto $\{12, 3\}$ con un valor de producto igual a 36.

Resolvemos el algoritmo mediante la técnica de Backtracking, haciendo uso de la clase Solución

```
class Solución {
```

```
private:
```

```
vector<int> producto; int n;
```

```
public:
```

```
Solución (int tam_max, int u) {
```

```
for (int i=0; i < tam_max; ++i) producto[i]=0;
```

```
U=u;
```

```
{
```

```
int size() const  
{  
    return producto.size();
```

```
void IniciaTemp (int u)  
{  
    if (u >= 0 && u < this.size())  
        producto[u] = 0;  
}
```

```
void SignarCmp (int u)  
{  
    if (u >= 0 && u < this.size())  
        producto[u]++;
```

```
bool TareasGeneradas (int u)  
{  
    int temp = producto[u];  
    producto[u] = producto[u] % 2;  
    return temp > 1;
```

```
int Desiscribir (int u) const  
{  
    if (u >= 0 && u < this.size())  
        return producto[u];
```

```
bool ProcesaSoluciones () // Se verifican si es válido  
{  
    for (int i = 0; i < this.size(); ++i)  
        if (v[i] > 1) return false;  
  
    for (int i = 0; i < this.size(); ++i)  
        cost += v[i];  
    return true;  
}
```

```
bool Factible (const vector<int> &comp, int u)  
{  
    this.producto(comp) < N,  
    if (this.dolar(comp) > N)  
        return false,  
    else return true;
```

```
int producto (const vector<int> &comp) // (Bump.size() == this.size())  
{  
    int p = 1;  
    for (int i = 0; i < this.size(); ++i)  
        if (producto[i] == 1) p *= comp[i];  
    return p;
```

```
int dolar (const vector<int> &comp, int u)  
{  
    int ct = this.producto(comp);  
    int uinc = min { producto, u, this.size() } / 4;  
    for (int i = u; i < this.size(); ++i) ct *= uinc;  
    return ct;
```

3. (2 puntos) Estamos en una posición (x, y) de un laberinto bidimensional representado por una matriz M de f filas y c columnas. De cada casilla se puede viajar a cualquiera de las colindantes con la casilla actual, en la misma fila o la misma columna, siempre que la casilla destino sea transitable. Los posibles valores de las casillas son $M(i, j) = t$ (transitable), $M(i, j) = n$ (no transitable) y $M(i, j) = s$ (salida). Sabiendo que disponemos del mapa del laberinto a priori, y que pueden existir varias salidas del mismo, plantee un algoritmo eficiente que devuelva el camino a la salida más próxima a la casilla inicial (x, y) . NOTA: El coste de viajar de una casilla a otra es siempre 1.

Este ejercicio se resuelve por Backtracking. La idea consiste en marcar las casillas ya pisadas de manera que al llegar a un punto cerrado volveremos hasta llegar a un punto no explorado. Realizando esta operación llegaremos a la salida. En este caso no es posible el uso de colas que serían realmente costosas.

Describir la clase Solución:

- Constructor • Atributos privados que representen la solución
- Destructor • Inicializa Ptemp (inf)
- TDec global() • TDec local
- int size() • bool TodosGenerados (inf)
- SigValorTemp (inf)
- void ProximoSolucion()
- bool Factible (inf)
- TDec Decision (inf)

4. (2 puntos) Disponemos de n tipos de monedas, las monedas de tipo i tienen un valor de $v[i]$. La cantidad de monedas disponibles de tipo i es igual a $c[i]$. Se quiere devolver una cantidad exacta M utilizando el menor número posible de monedas. Diseñad un algoritmo de programación dinámica que determine el número óptimo de monedas a usar. Aplicadlo para resolver el siguiente caso del problema, con $n = 3$, construyendo la tabla correspondiente:

$$v[] = (2, 4, 6), c[] = (3, 1, 2), M = 8$$

Al tener que resolver el problema con programación dinámica, deberemos probar el POB, por ello, sea S una solución al problema, veremos que si $\exists s_u$ siendo su el número de monedas del tipo u a usar. Como S es solución óptima para un costo $P \geq s_u \cdot c_u$ obtendremos que si s_u debe ser solución óptima para el problema con costo $P - s_u \cdot c_u$. Supongamos que no lo sea, entonces existe otra solución al problema con un menor número de monedas luego $P \leq s_u$ es otra mejor solución que S una solución óptima, lo cual es una contradicción. Por tanto, se cumple el POB.

Definiremos por $d(i, j)$ al mínimo número de monedas usadas de los tipos $0, \dots, i$ para resolver el problema de costo j .

$$d(i, 0) = 0, d(0, j) = \text{inf}$$

$$d(i, j) = 1 + d(i-1, j) \text{ si no hay monedas que puedan recuperar varios de los anteriores}$$

$$d(i, j) = j / \text{costo}(i) + d(i, j \% \text{costo}(i))$$

Por tanto, la recursión del problema viene dada por:

$$d(i,j) = \min \{ i+d(i-1,j), j/c[i] + d(i, j \% c[i]) \}$$

De esta manera, el algoritmo es pseudocódigo es el siguiente:

para cada i de $0, \dots, u$ {
 $d(i,0)=0$ → Debemos averiguar cuáles para ver que $d(i,0)=0$ hasta llegar a
 $d(0,j) = (j \% c[0])=0$? $j/c[0] \cdot j/c[0]+1$
 para cada i desde 1 hasta $u-1$
 para cada j desde 1 hasta $c[i]$ {
 $d(i,j) = \min \{ d(i-1,j), j/c[i] + d(i, j \% c[i]) \}$
 fin para cada j
 fin para cada i
 return $d(u,u)$

Ejemplo

		0	1	2	3	4	5	6	7	8
0	0	1	1	2	2	3	3	inf	inf	
1	0	1	1	2	1	2	1	3	2	
2	0	1	1	2	1	2	1	2	1	

5. (2 puntos) Diseñad un algoritmo que determine si un cierto número natural N puede expresarse o no como producto de tres números naturales consecutivos, $Y * (Y + 1) * (Y + 2)$, con un orden de eficiencia lo mejor posible.

Este problema se soluciona mediante la técnica de Divide y Conquerás. Para ver cuál es el criterio de división veremos que $\forall x \in [1, \sqrt[3]{n}+1]$ pues si $y > \sqrt[3]{n}$, como $(\sqrt[3]{n}+1) \cdot (\sqrt[3]{n}+2) \cdot (\sqrt[3]{n}+3) > n$. consecuente mente veremos $\sqrt[3]{n}+1$.

Por tanto, el algoritmo queda como sigue:

bool ProductoConservativo (int u, int iu, float)

if ($u \leq 2$) return true

else

int max = (iu > sqrt(3, u)) ? sqrt(3, u) : iu;

int mid = $\frac{max + iu}{2}$;

if ($(\sqrt{mid - 1} \cdot mid \cdot \sqrt{mid + 1}) == u$) return true

else {

if ($p > u$)

return ProductoConservativo (u, iu, mid);

else

return ProductoConservativo (u, mid, max);

{

{