Examen de Programación 3

14 de julio de 2017

En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

Parte obligatoria

Esta parte es eliminatoria. Para la aprobación del examen debe obtenerse un mínimo del **50 % de esta parte (20 puntos)**. En caso de no llegar a dicho mínimo, **NO** se corregirán los problemas.

Pregunta 1 (15 puntos)

Justifique si las siguientes afirmaciones son ciertas o falsas:

(a) $3^n \in O(2^n)$

(b) Sean $f, h : \mathbb{N} \to \mathbb{R}^+$. Entonces $f(n) \in O(h(n)) \Rightarrow 2^{f(n)} \in O(2^{h(n)})$

Solución:

(a) No es verdadera.

Aplicando la regla del límite,

$$\lim_{n\to\infty} (3^n/2^n) = \lim_{n\to\infty} (3/2)^n = \infty,$$

se comprueba que $3^n \notin 2^n$.

(b) No es verdadera.

Se muestra con un contraejemplo. Sean $f(n) = n \cdot \log 3$ y $h(n) = n$. Por la regla del límite

$$\lim_{n\to\infty} \frac{n \cdot \log 3}{n} = \log 3,$$

por lo que $f(n) \in O(h(n))$.

Pero como $2^{f(n)} = 2^{n\cdot\log 3} = \left(2^{\log 3}\right)^n = 3^n$ y $2^{h(n)} = 2^n$, por la parte a se tiene que $2^{f(n)} \notin O(2^{h(n)})$

Pregunta 2 (15 puntos)

(a) Describa el método *Greedy* de resolución de problemas.

(b) Enuncie un problema de optimización y describa un algoritmo *greedy* para resolverlo que no siempre obtenga la solución óptima.

(c) Muestre tres ejemplos: uno en el que el algoritmo encuentra la solución óptima, otro en el que encuentra una solución que no es óptima y otro en el que, aunque existe solución, el algoritmo no encuentra ninguna.

Solución:

(a) El método *Greedy* consiste en encontrar una solución en etapas, tomando una decisión en cada etapa que optimice un cierto criterio basado sólo en el estado en el que está el problema en ese momento.

(b) El *problema de las monedas* consiste en, dados un monto M y un conjunto $V = \{v_1, \cdots, v_n\}$ de valores de monedas, encontrar una colección con la mínima cantidad de monedas, cuyos valores pertenecen a V, tal que la suma de los valores de todas las monedas de la colección sea M. Se puede asumir que los valores de las monedas son no negativos y que se dispone de una cantidad ilimitada de monedas de cada valor. O sea, hay que encontrar una secuencia $(c_1, \cdots, c_n)$, $c_i \geq 0$, que minimice $\sum_{i=1}^{n} c_i$, sujeto a $\sum_{i=1}^{n} c_i v_i = M$.

El algoritmo propuesto hace en cada etapa una elección *greedy*, g, que es la moneda de mayor valor que no supere el monto que falta completar:

$c_i \leftarrow 0, \quad i \in \{1, \cdots, n\}$
`Mientras` $M \geq \min_{i \in \{1,\cdots,n\}}\{v_i\}$
  $g \leftarrow \text{argmax}_{i \in \{1,\cdots,n\}}\{v_i | \, v_i \leq M\}$
  $c_g \leftarrow M / v_g$
  $M \leftarrow M - c_g \cdot v_g$
`Devolver` $M, (c_1, \cdots, c_n)$

La secuencia devuelta, $(c_1, \cdots, c_n)$, es una solución si y sólo si $M = 0$.

(c) **Encuentra solución óptima** De manera trivial, si hay monedas de valor M el algoritmo encuentra la solucion óptima que consiste en una moneda.

Encuentra solución no óptima Sea $M = 6$ y monedas de valores 1, 3 y 4. El algoritmo devuelve una solución con 3 monedas, una de valor 4 y dos de valor 1; pero la solución óptima consiste en 2 monedas de valor 3.

No encuentra solución Sea $M = 6$ y monedas de valores 3 y 4. Hay una solución con dos monedas de valor 3, pero el algoritmo elige una moneda de valor 4 y termina con $M = 2$.

Pregunta 3 (10 puntos)

(a) ¿Qué representan los nodos internos y las hojas de un árbol de decisión?

(b) ¿Qué representa la cantidad de hojas y la altura de un árbol de decisión?

(c) En el problema de sorting mediante comparaciones dos a dos, ¿cuál es la altura mínima posible de un árbol de decisión en función del tamaño n de la entrada?

(d) Para una secuencia de tamaño $n = 5$, el costo de mergesort en el peor caso es 8. ¿Cuál es la cota inferior de sorting mediante comparaciones dos a dos para ese mismo tamaño? Explique si esto es compatible con la proposición de que mergesort es óptimo.

Solución:

(a) Dado un algoritmo, los nodos internos representa una pregunta que se hace en él que lleva a caminos de ejecución diferentes y las hojas representan las salidas posibles.

(b) La cantidad de hojas es la cantidad de salidas del algoritmo.

Este número cumple ser al menos la cantidad de salidas distintas posibles del algoritmo. Notar que, en algunos casos, algunas hojas son inalcanzables porque en los nodos internos se hacen preguntas contradictorias. De todas maneras, la cantidad de hojas es como mínimo la cantidad de salidas potenciales para el algoritmo.

La altura de un árbol de decisión determina la cantidad de preguntas que debe hacerse para obtener la respuesta en el peor caso.

(c) Como $n!$ es la cantidad de salidas distintas posibles dada una entrada de tamaño n, esa es la cantidad mínima de hojas. Como en los nodos internos las preguntas en este problema son comparaciones entre elementos, hay sólo dos respuestas posibles y por tanto el árbol es binario. Por lo tanto, la altura del árbol en este problema siempre es al menos $\lceil \log_2 n! \rceil$.

(d)

$$5! = 120 \Rightarrow \lceil \log_2 120 \rceil = \log_2 128 = 7$$

Entonces la cota inferior para este tamaño es 7.

Si bien mergesort no está en el mínimo para este caso, mergesort en realidad es óptimo asintóticamente. Entonces este resultado no implica que no se cumpla su condición de ser óptimo asintóticamente.

mergesort está en orden $O(\log_2 n!) = O(n \log_2 n)$, ya que $\log_2 n! \approx n \log_2 n - n$

Problemas

Problema A (30 puntos)

Se considera una grilla como la mostrada en la Figura 1, en la que cada línea horizontal o vertical de la grilla representa una calle por la cual se puede transitar. Aparecen destacados dos edificios E_1 y E_2 en los extremos de la grilla.

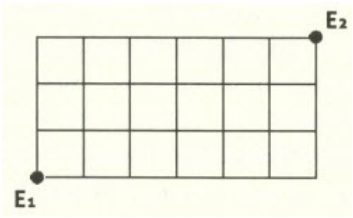

Figura 1: Grilla de calles y edificios destacados

Se asignan coordenadas a los puntos de la grilla que son intersección de dos calles, de modo que el edificio E_1 se considera ubicado en el origen con coordenadas $(0, 0)$ y el edificio E_2 está situado en las coordenadas (m, n). Esto corresponde a una grilla con $m+1$ calles verticales y $n+1$ calles horizontales. Para el ejemplo de la Figura 1 se tiene $m = 6$ y $n = 3$.

Se definen los caminos que llevan desde un punto genérico de la grilla de coordenadas (i, j) hasta el edificio E_2, como los recorridos sobre la grilla desde (i, j) hasta (m, n) con la restricción de que al pasar por una intersección de calles sólo se puede continuar hacia la derecha ó hacia arriba, sin salirse de la grilla (lo que hace que dichos caminos sean de largo mínimo). A modo de ejemplo, en la Figura 2 se muestran tres de los caminos posibles entre E_1 y E_2.

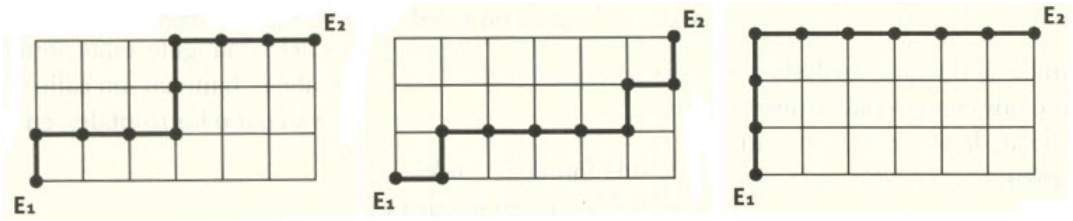

Figura 2: Ejemplos de caminos entre E_1 y E_2

Se define $NC(i, j)$ como el número total de caminos que hay entre el punto de la grilla de coordenadas (i, j) y el edificio E_2. Se toma la convención de que $NC(m, n) = 1$.

(I) Especificar completamente la relación de recurrencia que verifica $NC(i, j)$, para $0 \leq i \leq m, 0 \leq j \leq n$.

(II) Implementar en C^* utilizando la técnica de *Programación Dinámica* la función

```
int NumeroDeCaminos(int m,int n)
```

que calcula y devuelve la cantidad de caminos que llevan desde el edificio E_1 al edificio E_2, para una grilla cuyos valores de m y n se pasan como parámetro.

(III) Para el caso particular $m = 6$ y $n = 3$ correspondiente a la Figura 1, calcular en base al algoritmo implementado la cantidad de caminos entre E_1 y E_2, justificando el resultado obtenido.

Solución:

(I) Para todos los puntos de la grilla situados en los bordes de la misma con coordenada horizontal m ó coordenada vertical n, hay un único camino hacia el edificio E_2.
Para los restantes puntos de la grilla, la cantidad de caminos es la suma de la cantidad de caminos del punto vecino situado inmediatamente a su derecha en la grilla y de la cantidad de caminos del punto vecino situado inmediatamente arriba en la grilla.
La recurrencia puede plantearse entonces como:

$$NC(i,j) = \begin{cases} 1 & \text{si } i = m \text{ ó } j = n \\ NC(i+1,j) + NC(i,j+1) & \text{si } 0 \leq i < m, 0 \leq j < n \end{cases}$$

(II) La idea es usar una tabla NC para almacenar el número de caminos desde cada punto de la grilla hasta E_2. Primero se ponen en 1 los elementos de la tabla correspondientes a los bordes de la grilla con coordenada horizontal m y coordenada vertical n, y luego la tabla se completa recorriendo la grilla de derecha a izquierda y de arriba hacia abajo (podría ser también en el orden inverso), aplicando la recurrencia hallada. El resultado pedido es el último valor de la tabla calculado y corresponde a $NC[0][0]$.

```
int NumeroDeCaminos (int m, int n) {
   int NC[m+1][n+1];
   for(int j =0; j<=n; j++)   // borde derecho de la grilla
        NC[m][j] = 1;
   for(int i =0; i<=m; i++)   // borde superior de la grilla
        NC[i][n] = 1;
   for(i = m-1; i>=0; i--)      // de derecha a izquierda
    for(j = n-1; j>=0; j--)  // de arriba hacia abajo
             NC[i][j] = NC[i+1][j] + NC[i][j+1];
   return NC[0][0];
}
```

(III) En la Figura 3 se muestran para cada punto de la grilla los valores de NC obtenidos mediante el algoritmo implementado. La cantidad de caminos desde E_1 a E_2 es $\mathbf{NC[0][0] = 84}$.

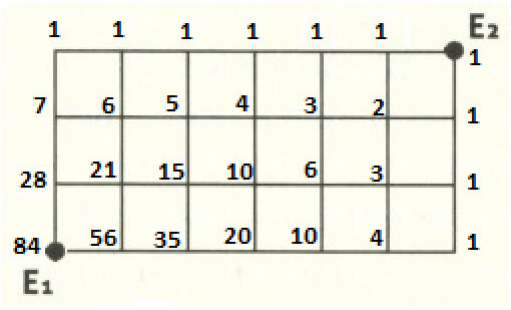

Figura 3: Grilla solución con cantidad de caminos hasta E_2

Este valor puede corroborarse observando que $NC[0][0] = \binom{m+n}{m} = \binom{m+n}{n} = \binom{9}{3} = 84$.

Problema B (30 puntos)

Se tiene un conjunto de tareas que se quieren realizar, pero existen relaciones de precedencia entre ellas. Por ejemplo, si se quieren llevar a cabo las tareas A, B y C, en donde C precisa que A esté finalizada para realizarla (*A precede a C*), las maneras posibles de hacerlo son ABC, BAC y ACB.

(I) Modelar el problema en término de grafos.

(II) ¿Es posible resolver el problema para cualquier conjunto de tareas y precedencias? Explicar brevemente por qué, mostrando un ejemplo.

(III) Implementar en seudocódigo una función que dado un grafo de estas características devuelva una estructura lineal que permita recorrer en orden las tareas cumpliendo las restricciones de precedencia. Puede usar y suponer como dadas las estructuras de datos básicas (incluyendo grafos) y las operaciones elementales correspondientes para poder manejarlas. Al implementar, asuma que el problema se puede resolver para el grafo dado.

(IV) ¿Qué cambios habría que hacer a la parte III para que en el caso que no exista solución se retorne el valor nulo?

Solución:

(I) El problema puede representarse mediante un grafo simple dirigido, en donde los vértices son las tareas y se definen aristas para un par de vértices (A, B) si y sólo si la tarea B precede a la tarea A.

El problema que se quiere resolver es el de encontrar un *Ordenamiento Topológico*.

(II) No siempre es posible. Si el grafo tiene ciclos entonces no hay solución ya que implicaría que se puede llegar de un vértice a otro y viceversa, significando que ambas tareas se preceden mutuamente y no pudiendo elegir una antes que la otra para realizarla sin romper una restricción. Por ejemplo, si se tienen las tareas A, B y C, y el precedencias representadas por el conjunto de aristas $(A, B), (B, C), (C, A)$, siempre que dé un orden para hacer las 3 tareas voy a romper alguna restricción de precedencia.

(III)
```
Pila ordenTopologico(Grafo g) {
    Para cada vertice v en g
        Desmarcar(v)

    p = CrearPila()

    Para cada vertice v en g
        Si no estaMarcado(v)
            dfs(g, v, p)

    Retornar p
}

void dfs(Grafo g, Vertice v, Pila p) {
    marcar(v)
    Para cada vertice w adyacente a v
        Si no estaMarcado(w)
            dfs(g, w, p)
    Apilar(p, v)
}
```

(IV) Si se encuentra un ciclo se debería terminar toda la ejecución y retornar el valor nulo. Para esto, se pueden usar marcas temporales cuando se visita un nodo por primera vez y en post-orden se lo puede marcar permanentemente. Si se llega a visitar un nodo marcado temporalmente, se encontró un ciclo ya que se encontró que se comenzó a procesar recursivamente pero sin terminar.

El código quedaría de la siguiente manera:

```
Pila ordenTopologico(Grafo g) {
    Para cada vertice v en g
        Desmarcar(v)

    p = CrearPila()
    haySolucion = true
```

```
    Para cada vertice v en g
        Si no estaMarcado(v)
            dfs(g, v, p, haySolucion)
            Si no haySolucion
                Retornar NULL

    Retornar p
}

void dfs(Grafo g, Vertice v, Pila p, bool & haySolucion) {
    marcarTemporalmente(v)
    Para cada vertice w adyacente a v
        Si estaMarcadoTemporalmente(w)
            haySolucion = false
            Retornar
        Si no estaMarcado(w)
            dfs(g, w, p, haySolucion)
            Si no haySolucion
                Retornar
    marcarPermanentemente(w)
    Apilar(p, v)
}
```