Segundo Parcial de Programación 3

28 de noviembre de 2016

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

Ejercicio 1 (20 puntos)

Carla, una estudiante de Ingeniería, camina todos los días de su casa a la Facultad. En el trayecto pasa siempre primero por una panadería y luego por lo de su amigo Jorge. Además, no quiere pasar dos veces por la misma esquina. Para ello, quiere implementar un algoritmo mediante la técnica de *Backtracking* que le retorne el camino de menor distancia que cumpla con las condiciones requeridas.

Las esquinas del mapa se representan mediante enteros comprendidos en el intervalo $[1 \dots e]$, donde e es la cantidad de esquinas. Carla identifica a su casa mediante la constante C, a la Facultad mediante la constante F, a la panadería mediante la constante P y a la casa de Jorge mediante la constante J. Todas son de tipo entero y representan a una esquina válida.

Se dispone de una tabla de distancias D, tal que $D(i, j)$ representa el largo de la cuadra que va de la esquina i a la esquina j, o ∞ si no hay una cuadra que las conecte. Todas las distancias son positivas

(a) Defina la forma de la tupla solución para el problema, considerando que cada elemento representa a una esquina.

(b) Defina la función $distancia_j(t)$ que calcula la distancia recorrida por Carla en la tupla t, si se consideran las primeras j esquinas recorridas.

(c) Indique en lenguaje natural y formalmente: restricciones implícitas, restricciones explícitas, función objetivo y predicado de poda en caso de que corresponda.

Solución:

(a) Tupla de largo variable $\langle t_1, \dots, t_k \rangle$ tal que $4 \leq k \leq e$, donde t_i representa la esquina en la que Carla se encuentra en el punto i, $\forall i \in \{1, \dots, k\}$. Se utilizará k para representar el largo de una tupla genérica.

(b)

$$distancia_j(t) = \sum_{i=2}^{j} D(t_{i-1}, t_i) \text{ con } j \in \{1, \dots, k\} \text{ donde } k \text{ es largo de la tupla } t.$$

(c)

- *Restricciones explícitas*
  - La primera esquina del recorrido es la casa de Carla:

$$t_1 = C$$

  - La última esquina del recorrido es la Facultad:

$$t_k = F$$

  - Todos los elementos de la tupla representan esquinas válidas:

$$t_i \in \{1, \dots, e\} \; \forall i \in \{1, \dots, k\}$$

- *Restricciones implícitas*
  - Hay camino entre todos los pares de esquinas consecutivas:

$$D(t_{i-1}, t_i) \neq \infty \; \forall i \in \{2, \dots, k\}$$

- No se repiten esquinas:

$$t_i \neq t_j \forall i, j \in \{1, \ldots, k\} \text{ con } i \neq j$$

- Solo se puede pasar por la casa de Jorge si se pasó previamente por la panadería.

$$\exists i \in \{2, \ldots, k-1\}\ t_i = J \Rightarrow \exists j \in \{2, \ldots, i-1\}\ t_j = P$$

- Cuando se termina el recorrido, se debió haber pasado por lo de Jorge:

  Sea k la posición final de la tupla, entonces

$$\exists i \in \{3, \ldots, k-1\} t_i = J$$

- *Función objetivo*
  Se quiere minimizar la distancia entre la casa de Carla y la Facultad:

$$f = min_{t \in T}\{distancia_k(t)\} \text{ con } T \text{ el conjunto de las tuplas solución.}$$

- *Predicado de poda*
  No hay.

Ejercicio 2 (20 puntos)

Sea $A = (A_1, \cdots, A_n)$ una secuencia de n enteros **diferentes**, mayores que 0. Una subsecuencia de A es una secuencia $(A_{h_1}, A_{h_2}, \cdots, A_{h_m})$ de elementos de A, con $1 \leq h_1 < h_2 < \cdots < h_m \leq n$. Una subsecuencia $(A_{h_1}, A_{h_2}, \cdots, A_{h_m})$ es creciente si $A_{h_1} < A_{h_2} < \cdots < A_{h_m}$. Se quiere calcular para cada i, $1 \leq i \leq n$, la longitud, denotada L_i, de la subsecuencia creciente más larga de A que termina en A_i.

Observación: Para cada i puede haber más de una subsecuencia creciente más larga de longitud L_i.

Ejemplo: Dado $A = (5, 2, 4, 7, 1, 6, 3)$, las longitudes de las subsecuencias crecientes más largas son: $1, 1, 2, 3, 1, 3, 2$. La subsecuencia más larga que termina en A_7 puede ser tanto $(2, 3)$ como $(1, 3)$. Las subsecuencias $(5, 6)$ y $(1, 6)$ son subsecuencias crecientes que terminan en A_6, pero $L_6 = 3$ porque también existe la subsecuencia creciente $(2, 4, 6)$, que se obtiene al incluir A_6 al final de la subsecuencia creciente más larga que termina en A_3 (lo cual es válido ya que $A_3 < A_6$).

(a) Llamamos P_i al problema de calcular L_i.

I. ¿Qué condición se debe cumplir en A para que $L_i = 1$?

II. Si $L_i \neq 1$, ¿cuáles son los subproblemas que se deben haber resuelto para resolver P_i?

(b) Escriba la recurrencia para calcular las longitudes L_i, para $1 \leq i \leq n$.

Nota: Para la notación se asume que el máximo de un conjunto vacío es 0.

(c) Escriba el seudocódigo que obtiene las longitudes $(L_1, \cdots, L_n)$ implementando la recurrencia de la parte b. La ejecución debe hacerse en tiempo $\Theta(n^2)$ y ocupar $\Theta(n)$ espacio.

Solución:

(a) I. Para que $L_i = 1$ tiene que cumplirse que ninguno de los elementos de A anteriores a la posición i sea menor a A_i: $\nexists\, j \in \{1, \ldots, i-1\}$ tal que $A_j < A_i$.

II. Si $L_i \neq 1$ es necesario resolver los subproblemas P_j para todo j menor que i que cumpla $A_j < A_i$.

(b) Con la siguiente recurrencia se obtienen las longitudes de las subsecuencias crecientes más largas:

$$L_i = 1 + \max_{\substack{1 \leq j < i \\ A_j < A_i}} \{L_j\}, \qquad 1 \leq i \leq n.$$

Se debe notar que si entre los elementos anteriores a i ninguno es menor que A_i (lo que ocurre trivialmente para A_1) se calcula el máximo de un conjunto vacío, que es 0.

(c)

```
Maximas-longitudes-subsecuencias-crecientes(A, n)
  // A = (A_1, ..., A_n)
  FOR i = 1 TO n
      L_i ← 1
      FOR j = 1 TO i - 1
          IF A_j < A_i
              L_i ← max(L_i, 1 + L_j)
  RETURN (L_1, ..., L_n)
```

Debido a los dos ciclos anidados el tiempo de ejecución es $\Theta(n^2)$. Sólo se mantienen las secuencias A y L, por lo que el espacio usado es $\Theta(n)$.

Los siguientes algoritmos, que intentan calcular un L_i genérico en tiempo $\Theta(i)$ no resuelven correctamente el problema. En el ejemplo mostrado en la letra, ambos obtendrían $L_6 = 2$, en lugar de $L_6 = 3$. El algoritmo de la izquierda consideraría que la secuencia más larga que termina en A_6 es $(5, 6)$ y el de la derecha consideraría que es $(1, 6)$, cuando en realidad es $(2, 4, 6)$.

```
anterior ← 0 // menor que todo A_j
L_i ← 1 // se debe incluir A_i
FOR j = 1 TO i - 1
    IF anterior < A_j < A_i
        L_i ← L_i + 1
        anterior ← A_j
```

```
siguiente ← A_i
L_i ← 1
FOR j = i - 1 DOWNTO 1
    IF A_j < siguiente
        L_i ← L_i + 1
        siguiente ← A_j
```

Una alternativa correcta a la solución propuesta utiliza recursión. Sin embargo, a pesar de la correctitud, el tiempo de ejecución es de orden exponencial. La siguiente es una de las posibles variantes:

```
FOR i = 1 TO n
    L_i ← version_recursiva(A, n, i)

version_recursiva(A, n, i)
    L ← 1
    // en el caso base (i = 1) se devuelve 1
    FOR j = i - 1 DOWNTO 1
        IF A_j < A_i
            L ← máx(L, 1 + version_recursiva(A, n, j)
    RETURN L
```

El peor caso se da cuando la secuencia está ordenada de manera creciente. Se puede probar que `version_recursiva(A, n, j)` se invocará 2^{n-j} veces, $1 \leq j \leq n$.

La versión eficiente se resolvió con Programación Dinámica. Para poder aplicar esta técnica se debe cumplir el Principio de Optimalidad. Para cada problema P_i consideremos el problema asociado Q_i que consiste en encontrar la subsecuencia creciente más larga que termina en A_i (o sea, la secuencia cuya longitud es la solución de P_i). Sea $S_i = (A_{h_1}, \cdots, A_{h_k}, A_i)$ la solución óptima de Q_i, y llamemos Pre$_i$ al prefijo $(A_{h_1}, \cdots, A_{h_k})$ de S_i. Consideremos el caso en que Pre$_i$ no es una secuencia vacía (esto es, el caso en que $L_i \neq 0$). Como Pre$_i$ es una subsecuencia creciente podría ser la solución óptima de Q_{h_k}. Supongamos, para llegar a un absurdo, que no lo es. Esto significa que hay otra subsecuencia creciente $(A_{h'_1}, \cdots, A_{h_k})$ cuya longitud L' es mayor que la longitud de Pre$_i$, que es $L_i - 1$. Como $A_{h_k} < A_i$, la subsecuencia $(A_{h'_1}, \cdots, A_{h_k}, A_i)$ es creciente, termina en A_i y tiene longitud $L' + 1$ mayor que L_i, por lo que S_i no puede ser la solución óptima de Q_i. Se llega a un absurdo motivado por suponer que Pre$_i$ no es solución óptima de Q_{h_k}. Por lo tanto se cumple el Principio de Optimalidad.

Ejercicio 3 (20 puntos)

Un equipo de programación se propone diseñar un algoritmo que ordene un arreglo de enteros, A, de tamaño n. Uno de los programadores sugiere el siguiente algoritmo híbrido de ordenación, argumentando que puede resultar más eficiente para tamaños relativamente reducidos de n:

1. Dividir el vector de tamaño n en n/k subvectores de tamaño k cada uno, siendo k un parámetro a determinar. Se asume que n es múltiplo de k y que n/k es potencia de 2.
2. Ordenar cada uno de los n/k subvectores empleando el algoritmo *InsertionSort*.
3. Hacer, por etapas, una secuencia de mezclas (el *2-way-merge* del *mergesort*) de pares de subvectores ordenados del mismo largo. En la primera etapa se empieza con los n/k subvectores de longitud k. Luego, en cada etapa se hacen mezclas de subvectores del doble de tamaño de la etapa anterior hasta obtener un único vector ordenado de tamaño n.

Se quiere determinar el costo en el peor caso de este algoritmo. La operación básica con la que se establece el costo es la comparación entre elementos del arreglo. No se considera relevante el costo del paso 1. En cuanto al paso 3, se calcula que el costo (despreciando términos de menor orden) es $c_3 n \log(n/k) = c_3 n \log n - c_3 n \log k$ para alguna constante positiva c_3. Esto se argumenta en base a que hay $\log(n/k)$ etapas, cada una con un costo total n.

(a) Calcule el tiempo de ejecución del algoritmo híbrido completo (puede agregar alguna constante generada al calcular el costo del paso 2).

(b) ¿Para cuál de los siguientes valores de k el comportamiento asintótico de este algoritmo híbrido alcanza la cota inferior de los algoritmos de ordenación basados en comparaciones de pares de elementos?

I. $k = \log n$

II. $k = \sqrt{n}$

(c) Exhiba el árbol de decisión del *InsertionSort* (usado en el paso 2) para arreglos de tamaño $n = 3$.

Solución:

(a) El paso 2 consiste en n/k ejecuciones del *InsertionSort* de arreglos de longitud k, cada una de las cuales tiene costo $c_2 k^2$ para algún c_2 positivo. Por lo que el costo del paso 2 es $\frac{n}{k} \cdot c_2 k^2 = c_2 nk$.

Entonces el costo de todo el algoritmo híbrido es $c_2 nk + c_3 n \log n - c_3 n \log k$.

(b) El tiempo de ejecución debe ser $\Theta(n \log n)$ ya que en el curso se ha demostrado que esa es la cota inferior de los algoritmos basados en comparaciones de pares elementos.

I. Se alcanza la cota. En la expresión del costo se sustiutye $\log n$ en lugar de k:

$$c_2 n \log n + c_3 n \log n - c_3 n \log(\log n) = (c_2 + c_3) n \log n - c_3 n \log(\log n).$$

Por la regla de la suma esta expresión pertenece a $\Theta(n \log n)$.

II. No se alcanza la cota. Procediendo como en la parte anterior se obtiene

$$\begin{aligned} c_2 n\sqrt{n} + c_3 n \log n - c_3 n \log(\sqrt{n}) &= c_2 n\sqrt{n} + c_3 n \log n - \frac{c_3}{2} n \log n \\ &= c_2 n\sqrt{n} + \frac{c_3}{2} n \log n. \end{aligned}$$

Y esta expresión pertenece a $\Omega(n \log n)$ pero no a $O(n \log n)$.

(c) Ver Apuntes Teóricos - Complejidad, página 16.