Examen de Programación 3

15 de julio de 2016

La prueba es individual y sin material. La duración es 4 hs. Escriba su cédula y nombre en cada hoja.

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.

Ejercicio 1 (7 puntos)

Sea A un arreglo de enteros, con índices de 1 a n, que se pasa como parámetro al algoritmo que se muestra a la derecha. Se considera que la operación básica es la asignación.

(a) ¿Cuándo se da el peor caso en el algoritmo y por qué?

(b) Sea $T(n)$ el tiempo de ejecución del algoritmo. Indique para cada uno de las siguientes órdenes si $T(n)$ pertenece a ese orden: $\Omega(n)$, $O(2n)$, $O(n^2)$, $\Theta(n^2)$. No hace falta justificar.

```
a ← A[n]
FOR i = n - 1 DOWNTO 1
    IF A[i] < a
        a ← A[i]
b ← A[1]
FOR i = 2 TO n
    IF A[i] > b
        b ← A[i]
RETURN (a,b)
```

Solución:

(a) El peor caso se da cuando la entrada está ordenada de manera creciente porque se hace la asignación en cada paso de ambos ciclos.

(b)

	$\Omega(n)$	$O(2n)$	$O(n^2)$	$\Theta(n^2)$
T_f	Sí	Sí	Sí	No

El tiempo del algoritmo es $\Theta(n)$. Son dos ciclos consecutivos, ambos en $\Theta(n)$. Por la regla de la suma el tiempo total es también $\Theta(n)$.
Entonces, por estar en $\Theta(n)$, $T(n)$ pertenece a $\Omega(n)$, y en $O(2n)$ porque $n \in \Theta(2n)$, y en $O(n^2)$ porque $n \in O(n^2)$. Pero no está en $\Theta(n^2)$ porque $n \notin \Omega(n^2)$.

Ejercicio 2 (11 puntos)

(a) Describa brevemente el algoritmo `mergesort`. ¿Cuál es el orden exacto de su tiempo de ejecución?

(b) Mediante árboles de decisión se demuestra que $\log n!$ es una cota en los algoritmos de ordenación mediante comparaciones. Como consecuencia de esto, ¿qué se puede afirmar, en términos de orden, de la relación entre $\log n!$ y $T(mergesort)$?

(c) Demuestre, complementando el ítem b que $T(mergesort) \in \Theta(\log n!)$

Ayuda: Puede usar, asumiéndolas como demostradas, las siguientes expresiones:

$$\log n! \in \Omega(n) \tag{1}$$

$$n \log n \leq \log n! + 1{,}5n \tag{2}$$

Solución:

(a) El algoritmo `mergesort` ordena una secuencia mezclando dos secuencias ordenadas que se obtienen de llamadas recursivas que tienen como parámetro la primera y la segunda mitad de la entrada. El orden es $\Theta(n \cdot \log n)$.

(b) El orden es $\Omega(\log n!)$.

Para la demostración se usan los árboles de decisión. La altura del árbol representa el costo del peor caso. Cada hoja representa la salida de una posible entrada, correspondiendo a entradas diferentes hojas diferentes. Como hay $n!$ posibles entradas el árbol tiene al menos $n!$ hojas. Por tratarse de árboles binarios estrictos la mínima altura de un árbol de decisión para este problema es entonces $\log n!$. Lo que permite esta demostración es encontrar una cota inferior para cualquier algoritmo de ordenamiento por comparaciones de pares de elementos.

(c) Como `mergesort` es un algoritmo de ordenamiento, se cumple $n \cdot \log n \in \Omega(\log n!)$.

Esto además es fácil verlo ya que

$$\log n! = \sum_{i=1}^{n} \log i < n \cdot \log n \qquad \text{para } n > 1.$$

Por lo tanto hay que demostrar que $n \cdot \log n \in O(\log n!)$, lo que, por definición de O implica demostrar que existen c y n_0 tales que $n \cdot \log n \leq c \cdot \log n!$ para $n \geq n_0$. Según la expresión (2) para que esto se cumpla es suficiente que $\log n! + 1{,}5n \leq c \cdot \log n!$.

Según la expresión (1), por definición de Ω, existen c_1 y n_1 tales que $\log n! \geq c_1 \cdot n$ para $n > n_1$, por lo que (multiplicando por 1,5 y dividiendo por c_1 en ambos miembros de la desigualdad) se tiene $1{,}5n \leq (1{,}5/c_1) \log n!$ para $n > n_1$. Entonces

$$n \cdot \log n \leq \log n! + 1{,}5n \leq (1 + 1{,}5/c_1) \log n! \qquad \text{para } n \geq n_1.$$

Por lo tanto alcanza con elegir $n_0 = n_1$ y $c = (1 + 1{,}5/c_1)$ para demostrar $n \cdot \log n \in O(\log n!)$.

Ejercicio 3 (11 puntos)

Sean $G = (V, E)$ un grafo dirigido y u un vértice de V.

(a) Defina la componente fuertemente conexa (CFC) de u.

(b) Considere las siguientes funciones del TAD Grafo (extendido):

- DFS: Grafo $\times$ Vértice $\rightarrow$ Grafo // Árbol
- Transpuesto: Grafo $\rightarrow$ Grafo
- Inducido: Grafo $\times$ Conjunto de Vértices $\rightarrow$ Grafo
- Vertices: Grafo $\rightarrow$ Conjunto de Vértices

I. Escriba el algoritmo de la función `Inducido_DFS(G,u)` que devuelve el subgrafo de G inducido por los vértices alcanzados por una recorrida `DFS` iniciada en u.

II. Escriba un algoritmo que obtenga la CFC de u en G.

En cada ítem se pueden usar las funciones de todos los ítems anteriores.

Solución:

(a) La CFC de u es el subgrafo de G inducido por el conjunto de vértices $v \in V$ para los cuales hay en G algún camino desde u hasta v y algún camino desde v hasta u.

(b)

Con una recorrida `DFS` empezada en u se obtiene un árbol (caso particular de grafo) cuyos nodos son los vértices hacia los que hay un camino desde u en el grafo original. Los vértices de la CFC son un subconjunto de los vértices así obtenidos. Con una recorrida `DFS` empezada en u en el traspuesto de un grafo se obtiene el arbol cuyos nodos son los vértices desde los cuales hay un camino hacia u en el grafo original. Si esta segunda recorrida se restringe al subgrafo que resulta de la primera se obtienen los vértices de la `CFC` a la que pertenece u. Esto se logra con los dos algoritmos siguientes:

Inducido_DFS

```
G1 ← DFS(G,u)
S1 ← Vertices(G1)
G2 ← Inducido(G,S1)
return G2
```

CFC

```
G1 ← Inducido_DFS(G,u)
G2 ← Transpuesto(G1)
G3 ← Inducido_DFS(G2,u)
G4 ← Transpuesto(G3)
return G4
```

Ejercicio 4 (11 puntos)

Considere los siguientes problemas vistos en el curso: Dijkstra, Búsqueda binaria, Floyd, Suma de subconjuntos, Prim. Seleccione exactamente 3. Para cada uno explique qué es lo que resuelve y con cuál de las técnicas de diseño de algoritmos estudiadas en el curso.

Solución:

Dijkstra Greedy.

Dados un grafo $G = (V, E)$ con aristas con peso no negativo, y un vértice u de V, devuelve el camino de menor de costo desde u a cada vértice de V. En otra versión devuelve el costo de esos caminos.

Búsqueda binaria Divide y conquista.

Dado un arrglo A ordenado y un elemento x, determina si x pertenece a A. En una versión más estricta, en el el caso de que x pertenezca a A devuelve la posición en que se encuentra.

Floyd Programación Dinámica

Dado un grafo $G = (V, E)$ sin ciclos de costo negativo, devuelve el camino de menor costo entre cada par de vértices de V. En otra versión devuelve el costo de esos caminos.

Suma de subconjuntos Backtracking

Dados un conjunto S de reales y un real M, devuelve los subconjuntos de S cuyos elementos suman exactamente M.

Prim Greedy

Dado un grafo $G = (V, E)$ no dirigido con aristas ponderadas, devuelve un árbol de cubrimiento de costo mínimo de G.

Problemas

Problema A (25 puntos)

Florentina quiere comprar libros por internet. Tiene una lista de 1000 libros que desea (identificados del 1 al 1000), la cual va a usar para seleccionar los que va a comprar. De cada libro i se conoce su peso p_i (en kg), su costo c_i (en dólares) y un valor v_i del 1 al 100 que indica el interés que tiene Florentina por ese libro (a mayor número, mayor es el interés). Además se sabe que algunos de esos libros pertenecen a una colección dada, y que hay M colecciones en la lista (identificadas del 1 al M). Para cada libro se conoce a qué colección pertenece o se sabe que no pertenece a ninguna ($col_i \in \{0, \ldots, M\}$ siendo 0 cuando no pertenece a ninguna colección). Si un libro pertenece a una colección, para comprarlo se deben comprar los anteriores de la colección correspondiente. El orden de la lista respeta el orden de los libros en las colecciones (si aparece un libro en la lista, todos los anteriores de su colección aparecen previamente). La matriz $Orden$ indica el orden de los libros en las colecciones ($Orden[j,k] = i$ indica que el k-ésimo elemento en la colección j, es el libro i). A lo sumo se debe comprar una copia de cada libro.

Florentina tiene un presupuesto máximo de U$S200, y un límite de 5kg de peso total. Cada libro le cuesta U$S5 de flete.

Se quiere maximizar el valor (de interés) de los libros comprados, minimizando el costo (lo prioritario es maximizar el interés, y a igual interés, se elige la solución más económica).

Resuelva el problema usando la técnica **Backtracking**: exprese en lenguaje natural y en lenguaje formal la forma de la tupla, restricciones explícitas, restricciones implícitas y función objetivo en el caso que correspondan.

Solución:

Solución con tupla de largo fijo

Forma de la tupla

Tupla de largo fijo $t =< x_1, \ldots, x_{1000} >$ que representa cuáles libros comprar, donde cada x_i representa si se compra o no el libro i.

Restricciones explícitas

(a) $x_i \in \{0,1\}\ \forall i \in \{1, \ldots, 1000\}$, siendo 1 si se compra el libro o 0 si no.

Con esto se cumple que a lo sumo se compre una copia de cada libro.

Restricciones implícitas

(a) Restricción de costo: el costo de cada libro más el flete no puede superar el presupuesto $\sum_{i=1}^{1000} x_i * (c_i + 5) \leq 200$.

(b) Restricción de peso: el peso total no puede superar el tope $\sum_{i=1}^{1000} x_i * p_i \leq 5$.

(c) Restricción del orden de la colección: si se compra un libro que pertenece a una colección, se deben comprar los anteriores en esa colección.
Para $i \in \{2, \ldots, 1000\}$ si $x_i = 1$ y $col_i \neq 0$ y $(\exists k \geq 2)(Orden[col_i, k] = i)$ entonces $x_{Orden[col_i, k-1]} = 1$.

Basta con chequear el inmediato anterior, ya que si está en la tupla es porque también cumple con esta restricción, cubriendo así a todos los previos. No hace falta chequear la restricción para el libro 1 porque los identificadores respetan la prelación en las colecciones, por lo que si el 1 pertenece a una colección, tiene que ser el primero en ella.

Función objetivo

Se quiere maximizar el valor de interés para Florentina en la compra de sus libros, y dentro de las soluciones máximas elegir una de menor costo.

$$\min_{t \in MejorValor} \left\{ \sum_{i=1}^{1000} x_i * (c_i + 5) \right\}$$

donde

$$MejorValor = \underset{t \in Sol}{argmax} \left\{ \sum_{i=1}^{1000} x_i * v_i \right\}$$

siendo Sol el conjunto de tuplas solución.

La función $argmax$, aplicada a una función f cuyo dominio es D devuelve el subconjunto de elementos de D que maximizan f:

$$\underset{x \in D}{argmax} f(x) = \{x \in D \mid \forall y \in D \; f(y) \leq f(x)\}.$$

Solución con tupla de largo variable

Forma de la tupla

Tupla de largo variable $t =< x_1, \ldots, x_n >$ que contiene los identificadores de los libros a comprar, donde n es la cantidad de libros que se compran. El orden de los libros de la tupla respeta el orden de los libros en la lista.

Restricciones explícitas

(a) Cada elemento es el identificador de un libro a comprar. $x_i \in \{1, \ldots, 1000\} \; \forall i \in \{1, \ldots, n\}$.

Restricciones implícitas

(a) Restricción de costo: el costo de cada libro más el flete no puede superar el presupuesto. $\sum_{i=1}^{n} (c_{x_i} + 5) \leq 200$.

(b) Restricción de peso: el peso total no puede superar el tope. $\sum_{i=1}^{n} p_{x_i} \leq 5$.

(c) Restricción de orden de la lista de libros: los libros seleccionados se incluyen en la tupla respetando el orden de la lista. Si $1 \leq i < j \leq n$, entonces $x_i < x_j$.

Esto obliga a que no se generen soluciones que son permutaciones de otras soluciones previamente analizadas (eficiencia).

(d) Restricción del orden de la colección: si se compra un libro que pertenece a una colección, se deben comprar los anteriores en esa colección.

Basta con chequear el inmediato anterior, ya que si está en la tupla es porque también cumple con esta restricción, cubriendo así a todos los previos.

Para $i \in \{1, \ldots, n\}$ si $col_{x_i} \neq 0$ y $(\exists k \geq 2)(Orden[col_{x_i}, k] = x_i)$ entonces

$$(\exists l \in \{1, \ldots, i-1\})(x_l = Orden[col_{x_i}, k-1])$$

.

Función objetivo

Se quiere maximizar el valor de interés para Florentina en la compra de sus libros, y dentro de las soluciones máximas elegir la de menor costo.

$$\min_{t \in MejorValor} \left\{ \sum_{i=1}^{n} c_{x_i} + 5 \right\}$$

donde

$$MejorValor = \underset{t \in Sol}{argmax} \left\{ \sum_{i=1}^{n} v_{x_i} \right\}$$

siendo Sol el conjunto de tuplas solución.

Problema B (20 puntos)

(I) Sea G un grafo simple no dirigido. Los vértices de G se identifican con enteros del 0 al $n-1$ y el grafo se representa mediante un arreglo de listas de adyacencia.

Implemente la siguiente función no recursiva:

```
bool asignar_paridad(int u, int n, Lista * adyacentes, int * & paridad);
```

En el arreglo `paridad` se debe indicar si la distancia desde u hasta los vértices de su misma componente es par o impar. Concretamente: $paridad[v]$ debe ser 0 si la distancia es par, 1 si es impar o -1 si v no pertenece a la misma componente de u.

Además, se debe devolver `true` si en la componente de u no hay aristas entre vértices de igual paridad; en otro caso se debe devolver `false`.

(II) Un grafo $G = (V, E)$ es bipartito si el conjunto de vértices se puede particionar en dos subconjuntos de tal manera que cada arista tenga un extremo en uno de los subconjuntos y el otro extremo en el otro subconjunto ¿Cómo se puede determinar si una componente de un grafo es bipartita usando la función de la parte anterior?

Solución:

(I)

```
bool asignar_paridad(int u, int n, Lista * adyacentes, int * & paridad) {
  bool res = true;
  for (int i = 1; i <= n; i++)
    paridad[i] = -1;
  paridad[u] = 0;
  Cola q = crear_cola();
  encolar(u,q); // paridad[u] != -1
  while (! es_vacia_cola(q)) {
    int w = frente(q);
    Lista lst = adyacentes[w];
    while (! es_vacia_lista(lst)) {
      int v = primero(lst);
      if (paridad[v] == paridad[w]) {
        res = false; // entre u, v y w se forma un ciclo de longitud impar
      } else if (paridad[v] == -1) {
        paridad[v] = (paridad[w] + 1) % 2;
        encolar(v,q); // paridad[v] != -1
      }
      lst = resto(lst);
    }
    desencolar(q);
  }
  destruir_cola(q);
  return res;
}
```

La función pedida es una recorrida `BFS` en una componente del grafo a la que se agrega mantener la paridad de la distancia al origen, y el control de la no existencia de aristas entre vértices de igual paridad. Si en una componente no hay aristas entre vértices de igual paridad entonces la componente es bipartita.

(II) Sea u un vértice de la componente considerada y se pasa como parámetro a `asignar_paridad`. La componente es bipartita si y sólo si el resultado de la llamada es `true`.

Problema C (15 puntos)

Martín, que es fanático de los programas de televisión, ganó un sorteo en el cual puede llevarse todo lo que quiera de un supermercado hasta un máximo $W = 10.000$ gramos. En el supermercado se sabe que hay n productos distintos, teniendo cada uno un precio d_i y un peso w_i (en gramos) conocidos (ambos números naturales), $i \in \{1, \ldots, n\}$. Para simplificar y sin pérdida de generalidad se asume que hay disponible un solo ítem de cada producto. Martín es astuto y desea revender lo que se lleve del supermercado, entonces quiere llevar la mayor cantidad de dinero en productos posible.

(I) ¿Qué técnica de diseño de algoritmos vista en el curso nos permite resolver de forma óptima y eficiente el problema de determinar cuánto es la ganancia máxima que puede obtener? Especificar matemáticamente el problema y la solución para resolverlo.

(II) Implementar en C* el algoritmo de la parte I. Asuma que se tienen disponibles las funciones *max* y *min*, que devuelven el máximo y mínimo de dos números enteros, respectivamente.

(III) Si se pudiera llevar partes fraccionadas de productos (por ejemplo, cantidad 0,243 de una botella con agua), indicar, justificando brevemente, una manera más eficiente de resolver el problema vista en el curso.

Solución:

(I) La técnica que nos permite resolver el problema es Programación Dinámica.

Esta realidad se corresponde con el Problema de la Mochila.

Una solución al problema puede representarse mediante las variables $x_i \in \{0, 1\}$, indicando cada una si el ítem i se elige llevar o no. Se desea maximizar el precio de los ítems que se llevan:

$$\max_{x_1,\ldots,x_n} \sum_{i=1}^{n} d_i x_i$$

Sujeto a no sobrepasar el peso máximo:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

Se define $g(k, w)$ como la ganancia de la solución óptima al problema restringido solamente a los ítems 1 a k, teniendo un peso máximo w. Basándose en el Principio de Optimalidad se puede plantear de la siguiente manera:

$$g(k, w) = \max\{g(k-1, w), g(k-1, \max\{w - w_k, 0\}) + d_k\}$$

Como casos base se tienen:

$$g(0, w) = 0, \forall w \in \{0, \ldots, W\}$$

$$g(k, 0) = 0, \forall k \in \{0, \ldots, n\}$$

(II) La implementación del algoritmo es la siguiente:

```
int FKnap(int n, int *w, int *d, int W) {
  int g[n + 1][W + 1];
  for (int j = 0; j <= W; j++)
    g[0][j] = 0;
  for (int k = 1; k <= n; k++)
    for (int j = 0; j <= W; j++)
      g[k][j] = max(g[k-1][j], g[k-1][max(j-w[k], 0)] + d[k]);
  return g[n][W];
}
```

(III) En este caso se puede resolver mediante Greedy, ya que se puede eligir en cada paso la mayor cantidad del ítem que da mayor ganancia en relación a su peso, y así obtener la mayor ganancia posible.