Solución Examen de Programación 3 y III (22/07/2011)

Instituto de Computación, Facultad de Ingeniería, UdelaR

Parte Teórica Obligatoria

Ejercicio 1 (10 puntos)

Por definición de límite tenemos que
$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = c \in R^+$ sii $\forall \varepsilon > 0, \exists n_0$ tq $\forall n \geq n_0, \left|\frac{f(n)}{g(n)} - c\right| < \varepsilon$

Lo que equivale a que
$\forall \varepsilon \in R^+, \exists n_0 \in N$ tal que $\forall n \geq n_0$, $-\varepsilon < \frac{f(n)}{g(n)} - c < \varepsilon$

Operando, $\frac{f(n)}{g(n)} < \varepsilon + c$, como ambos son positivos puedo considerar: $c' = \varepsilon + c \in R^+$.

Por lo que tenemos que $\exists c' \in R^+, \exists n_0 \in N$ tal que $\forall n \geq n_0$, $f(n) < c' g(n)$ que justamente es la definición $f(n) \in O(g(n))$.

Por otro lado,

$$\lim_{n\to+\infty} 1 = \frac{f(n)g(n)}{g(n)f(n)} = c\frac{g(n)}{f(n)} \text{ implica que } \lim_{n\to+\infty} \frac{g(n)}{f(n)} = \frac{1}{c}.$$

Por lo que se está en las mismas hipótesis de lo que se probó anteriormente y $g(n) \in O(f(n))$.

En virtud de la *Propiedad A*, se tiene que $f(n) \in O(g(n))$, $g(n) \in \Omega(f(n))$ y además $g(n) \in O(f(n))$, $f(n) \in \Omega(g(n))$.

Por definición de Θ tenemos que $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(f(n))$.

Propiedad A:
Para toda función $f : N \to R^*, g : N \to R^*$ se cumple la propiedad $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
Ya que:

$$f(n) \in O(g(n)) \Leftrightarrow (\exists c \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow f(n) < c \cdot g(n))$$

Lo que equivale
$(\exists c \in R^+)(\exists n_0 \in N)(\forall n \in N)\left(n \geq n_0 \rightarrow \frac{1}{c} f(n) < g(n)\right)$ y dado que *c* es un real positivo arbitrario, $\frac{1}{c}$ es una biyección con los reales positivos y por lo tanto:

$(\exists c' \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow c' f(n) < g(n))$ que es la definición de $g(n) \in \Omega(g(n))$.

Ejercicio 2 (10 puntos)

1)

```
1 // Calcula el producto de la matriz a por la matriz b y pone el resultado en c.
2 // La memoria de c fue previamente reservada.
3 void producto (int m, int n, int p, int** a, int** b, int** c){
4       for (int i=0; i<m; i++){
5             for (int j=0; j<p; j++){
6                   c[i][j]=0;
7                   for (int k=0; k<n; k++)
8                         c[i][j] += (a[i][k]*b[k][j]);
9             }
10      }
11 }
```

Para calcular el costo exacto de la implementación del producto de matrices, consideramos que siempre se entra en los 3 for, y dentro de ellos se realiza una multiplicación (línea 8) por lo tanto:

$$T = \sum_{i=0}^{i=m-1} \sum_{j=0}^{j=p-1} \sum_{k=0}^{k=n-1} 1 = \sum_{i=0}^{i=m-1} \sum_{j=0}^{j=p-1} n = \sum_{i=0}^{i=m-1} p * n$$

$$T = m * p * n$$

$$T \in \Theta(m * p * n)$$

Teniendo como dato además que $\Theta(m) = \Theta(n) = \Theta(p)$ podemos simplificar la expresión aplicando las definiciones de $\Omega(n)$ y $O(n)$:

- $m \in \Theta(m) \Rightarrow m \in \Theta(n) \Rightarrow m \in O(n)$
- $p \in \Theta(p) \Rightarrow p \in \Theta(n) \Rightarrow p \in O(n)$

I) $m \in O(n) \Leftrightarrow (\exists c_0 \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow m < c_0 \cdot n)$

II) $p \in O(n) \Leftrightarrow (\exists c_1 \in R^+)(\exists n_1 \in N)(\forall n \in N)(n \geq n_1 \rightarrow p < c_1 \cdot n)$

De I y II, elijo $n_2 = máx\{n_0, n_1\}$ y $c_2 = c_0 \cdot c_1$ entonces:

$$(\exists c_2 \in R^+)(\exists n_2 \in N)(\forall n \in N)(n \geq n_2 \rightarrow p \cdot m < c_2 \cdot n^2)$$

multiplicando por n a ambos lados de la desigualdad:

$$(\exists c_2 \in R^+)(\exists n_2 \in N)(\forall n \in N)(n \geq n_2 \rightarrow p \cdot m \cdot n < c_2 \cdot n^3)$$

lo que es la definición de

$$\boxed{p \cdot m \cdot n \in O(n^3)}$$

Haciendo el mismo razonamiento con la cota inferior:

- $m \in \Theta(m) \Rightarrow m \in \Theta(n) \Rightarrow m \in \Omega(n)$
- $p \in \Theta(p) \Rightarrow p \in \Theta(n) \Rightarrow p \in \Omega(n)$

I) $m \in \Omega(n) \Leftrightarrow (\exists c_0 \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow m > c_0 \cdot n)$

II) $p \in \Omega(n) \Leftrightarrow (\exists c_1 \in R^+)(\exists n_1 \in N)(\forall n \in N)(n \geq n_1 \rightarrow p > c_1 \cdot n)$

De I y II, elijo $n_2 = máx\{n_0, n_1\}$ y $c_2 = c_0 \cdot c_1$ entonces:

$$(\exists c_2 \in R^+)(\exists n_2 \in N)(\forall n \in N)(n \geq n_2 \rightarrow p \cdot m > c_2 \cdot n^2)$$

multiplicando por n a ambos lados de la desigualdad:

$$(\exists c_2 \in R^+)(\exists n_2 \in N)(\forall n \in N)(n \geq n_2 \rightarrow p \cdot m \cdot n > c_2 \cdot n^3)$$

lo que es la definición de

$$\boxed{p \cdot m \cdot n \in \Omega(n^3)}$$

Por lo tanto, sabiendo que $p \cdot m \cdot n \in O(n^3)$ y $p \cdot m \cdot n \in \Omega(n^3)$ se deduce que $\boxed{p \cdot m \cdot n \in \Theta(n^3)}$.

La expresión reducida es:

$$\boxed{T \in \Theta(n^3)}$$

2) Método de Strassen:

El método de Strassen, que consiste en dividir cada matriz en cuatro submatrices de dimensión $\frac{n}{2} x \frac{n}{2}$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Donde por ejemplo, $A_{11} = \begin{bmatrix} a_{11} & \dots & a_{1\frac{n}{2}} \\ \dots & \dots & \dots \\ a_{\frac{n}{2}1} & \dots & a_{\frac{n}{2}\frac{n}{2}} \end{bmatrix}$

y calcular el producto como, $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

donde:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Ejercicio 3 (10 puntos)

1)

Los algoritmos voraces son aquellos los cuales para resolver un problema se basan en construir una solución utilizando decisiones que son óptimos locales logrando un optimo global (en caso de ser correcto el algoritmo).

```
Greedy ( C )
{
   S := Φ
   while ( !esVacio(C) and  !esSolucion(S)  )
   {
          x :=  Select(C)
          C = C - {x}

          if ( esFactible (S U {x} )
          {
                S := S U {x}
          }
   }

   if ( esSolucion(S))
          return S
   else
          return Φ
}
```

C es el conjunto de candidatos.

S es la solución.

La función *select* se encarga de la selección del mejor de los candidatos basándose en una decisión local.

2)

En términos de grafos el problema se puede plantear de la siguiente forma:

> Dado un grafo $G = (V,E)$ dirigido o no, con una función c de costo asociado ($c : E \rightarrow R^{+} \cup \{0\}$) a las aristas de G, el problema consiste en determinar los caminos de mínimo costo desde un vértice a todos los otros vértices del grafo.

Para implementar este algoritmo se necesita:

- Una matriz C_{nxn} / $C[i, j]$ es el costo de ir desde el vértice *i* al vértice *j* si existe la arista <*i*, *j*>, sino $C[i, j] = \infty$
- Un arreglo D_{1xn} que contenga el costo del camino de mínimo costo del origen a cada vértice.

Para el pseudo código del algoritmo se asume que los vértices están numerados de 1 a *n*.

```
Procedure Dijkstra (int origen)
     S = {origen};
     Para (i in 1..n)
          D[i] = C[origen][i]
     Para (i in 1..n-1)
          elegir un vertice w in V-S / D[w] sea minimo
          S = S ∪ {w};
          Para (v in V-S)
               D[v] = minimo{D[v], D[w]+C[w][v]}
          Fin
     Fin
Fin
```

Ejercicio 4 (10 puntos)

1) Un *Árbol de Decisión* es un **árbol binario estricto** (cada nodo tiene 2 hijos o ninguno) donde los **nodos internos tienen 2 hijos y los externos (hojas) no tienen ninguno**.
Como herramienta para modelar algoritmos, los nodos internos representan **decisiones** entre elementos y las hojas representan **salidas** posibles del algoritmo.

En el caso particular de algoritmos de ordenación, la decisión que consideramos es la comparación y las salidas son posibles permutaciones de la secuencia a ordenar.

2) Un método de ordenación es **estable** si al final del mismo, elementos de igual valor permanecen en el **mismo orden** en que se encontraban originalmente en la secuencia a ordenar.
Sea S={2, 3, 5, 8, 3'} una secuencia a ordenar.
Un algoritmo ***estable***, producirá (de manera determinista) la secuencia S'={2,3,3',4,8}, mientras que uno no estable puede producir la secuencia S''={2,3',3,5,8}.

InsertionSort un arreglo para la secuencia a ordenar. La estrategia consiste en mantener en las primeras posiciones del arreglo la subsecuencia ya ordenada en forma creciente (no estricta) y en el resto la subsecuencia desordenada. En un paso del algoritmo se toma el primer elemento (`first`) de la parte desordenada y se recorre la parte ordenada (hacia las menores posiciones) buscando el lugar que debe ocupar `first` en la parte ordenada, se inserta y esta subsecuencia crece en un elemento. Es claro, que la estabilidad del algoritmo se encuentra el modo que se lleva a cabo la *inserción* del elemento arbitrario en la secuencia ordenada.

Una posible implementación de *insertionSort*:

```
void insertionSort (int* a, int n){
      int i, j, first;
      for (i = 1; i < n; i++){
            first = A[i];
            j = i-1;
            while (j >= 0 && first < a[j]){
                  a[j+1] = a[j];
                  j--;
            }
      a[j+1] = first;
      }
}
```

En el `while` se hace la búsqueda en la parte ordenada; cuando encuentre un elemento mayor o igual a `first`, la búsqueda termina y se inserta el elemento al final de la parte ordenada. Después se repite el proceso para un nuevo elemento de los desordenados (el primero).
Esta implementación **es estable** porque si `A[j] == first`, entonces se sale del `while` y se coloca `first` en la posición `j+1` manteniendo así el orden relativo de los 2 elementos iguales.

Problemas

Problema 1 (30 puntos)

1) Forma de la tupla:
Tupla de largo fijo O de la forma $<(a_1, b_1),...,(a_O, b_O)>$. Cada componente (a_i, b_i) donde $1 \leq i \leq O$, representa la cantidad de personal del grupo A y B respectivamente asignado a la obra i.

Restricciones explícitas:

- $A_m \leq a_i \leq A_M$, ninguna obra puede tener más personal del grupo A que el que tiene la empresa,
- $B_m \leq b_i \leq B_M$, ninguna obra puede tener más personal del grupo B que el que tiene la empresa,

Restricciones implícitas:

- $\sum a_i \leq |A|$, el personal del grupo A asignado a las obras no puede ser mayor que el que tiene la empresa,
- $\sum b_i \leq |B|$, el personal del grupo B asignado a las obras no puede ser mayor que el que tiene la empresa.

Función objetivo:

Sea $x_i = \lfloor t_i / (A_d a_i + B_d b_i) \rfloor$ la cantidad de días que demora en ejecutar el personal asignado a la obra o_i dicha obra y $z_i = \{x_i - d_i$ si $x_i > d_i, 0$ en otro caso$\}$ los días de atraso, entonces:

- $min\ x_i (A_c \sum a_i + B_c \sum b_i) + \sum z_i p_i$

2) i) No es predicado de poda ya que se desprende directamente de evaluar la función objetivo.
ii) No es predicado de poda ya que se desprende directamente de las restricciones implícitas.

3) i) Dado que los grupos cuentan con personal ilimitado, que no hay cotas superiores para el personal de los grupos en las obras y que la cantidad de trabajos que realiza el grupo A es múltiplo de B entonces: si $kB_c \leq A_c$ entonces existe una solución óptima asignando solo personal de B; en caso contrario la solución óptima es asignando solamente personal de A.

Se eliminan las cotas superiores de las restricciones explicitas i y ii. Se eliminan las restricciones implícitas.

ii) Se eliminan las cotas inferiores de las restricciones explícitas i y ii y se agrega la siguiente restricción:
si $0 = a_i$ entonces $0 < b_i$ y viceversa, en toda obra debe haber personal de al menos un grupo.

Problema 2 (30 puntos)

La Solución consiste en hacer un árbol de cubrimiento mínimo con DFS para cada nodo del grafo.
Cuando construimos el árbol para un nodo v, luego nos fijamos si ese nodo tiene más de un adyacente. En ese caso, realizaremos DFS para cada rama, con el objetivo de contar la cantidad de nodos de cada rama. Los resultados obtenidos los multiplicaremos para obtener la prioridad, y agregar el vértice a la lista de dispositivos críticos de acuerdo a ella.

```
int DFS_cubrimientoMinimo(Grafo G, Grafo CM, int v, bool* marcados) {
	marcados[v] = true;
	Lista vertices_adyacentes = adyacentes(G, v);
	while (!listaVacia(vertices_adyacentes)) {
		int w = listaPrimero(vertices_adyacentes);
		if (!marcados[w]) {
			agregarVertice(CM, w);
			agregarArtista(CM, (v,w));
			DFS_es_PA(G, CM, w, marcados);
			}
		listaQuitar(w, vertices_adyacentes);
		}
	Lista vertices_adyacentes_CM = adyacentes(CM, v);
	if( largo(vertices_adyacentes_CM)>1){
		int prioridad = 1;
		for (int i = 0; i < n; i++)
			marcados[i] = false;
		marcados[v] = true;
		while (!listaVacia(vertices_adyacentes_CM)) {
			int w = listaPrimero(vertices_adyacentes_CM);
			DFS_CantidadNodos(CM, w, marcados, cantNodos);
			listaQuitar(w, vertices_adyacentes_CM);
			prioridad = prioridad*cantNodos;
		}
		return prioridad;
	}
	return -1;
}

void DFS_CantidadNodos(Grafo G, int v, bool* marcados, int & cantNodos) {
	marcados[v] = true;
	cantNodos ++;
	Lista vertices_adyacentes = adyacentes(G, v);
	while (!listaVacia(vertices_adyacentes)) {
		int w = listaPrimero(vertices_adyacentes);
		if (!marcados[w])
			DFS_CantidadNodos(G, w, marcados, cantNodos);
		listaQuitar(w, vertices_adyacentes);
		}
}

Lista puntosCriticos(Grafo red) {
	Lista dispositivos_criticos = crearLista();
	int n = cantNodosGrafo(red);
	int prioridad;
	bool* marcados = new bool[n];
	for (int i = 0; i < n; i++){
		for (int i = 0; i < n; i++)
			marcados[i] = false;
		Grafo CM = CrearGrafo();
		agregarVertice(CM, i);
		prioridad = DFS_cubrimientoMinimo(red, CM, i, marcados);
		if (prioridad =! -1)
			insertarDescendente(dispositivos_criticos, i, prioridad);
		}
	return dispositivos_criticos;
}
```