Examen de Programación 3 y III (14/12/2009)

Instituto de Computación, Facultad de Ingeniería, UdelaR

1. Este examen dura 4 horas y contiene **4** carillas. El total de puntos es 100. Para su aprobación necesita 60 puntos.
2. En los enunciados llamamos C* a la extensión de C al que se agrega el operador de pasaje por referencia &, y las sentencias *new*, *delete* y el uso de *cout* y *cin*.
3. NO se puede utilizar ningún tipo de material de consulta. Puede usarse todo lo visto en el teórico, práctico y laboratorio sin demostrarlo, indicando claramente lo que se está usando.
4. No se contestaran dudas durante la última media hora.

Se requiere:

- Numerar todas las hojas e incluir en cada una el nombre y la cédula de identidad.
- Utilizar las hojas de un sólo lado y escribir con lápiz, iniciando cada ejercicio en hoja nueva.
- Poner en la primera hoja la cantidad de hojas entregadas, y un **índice** indicando en qué hoja se respondió cada problema.

Parte Teórica Obligatoria

Esta parte es eliminatoria, vale **40** puntos y usted debe obtener como mínimo el **50% de esta parte (20 puntos)**. En caso de no llegar a dicho mínimo, **NO** se corregirán los problemas. Usted podrá encontrar planteos prácticos. Los mismos deben ser resueltos justificando detalladamente la correspondencia con la base teórica que utilice en su respuesta.

Ejercicio 1 (10 puntos)

Dado un grafo ***G = (V, E)*** con costos no negativos asociados a sus aristas y dos vértices **u, v** □ **V**, un problema conocido es encontrar el camino de menor costo entre ellos. Existen algoritmos en los cuales basarse para lograrlo, como Dijkstra (aplica Greedy) o Floyd (aplica Programación Dinámica). Ahora interesa resolver dicho problema utilizando la técnica de Backtracking.
Se pide formalizar el problema en términos de Backtracking. Para cada sección/ítem de la formalización debe justificar porqué coloca una expresión cualquiera en la misma.

Ejercicio 2 (10 puntos)

Sea G un grafo conexo, no dirigido, con costos no negativos asociados a sus aristas.
¿Es posible encontrar un vértice v del grafo tal que la aplicación de algoritmo de Dijkstra, tomando v como origen, dé como resultado un árbol de cubrimiento de costo mínimo del grafo G dado?
Justifique detalladamente su respuesta.

Ejercicio 3 (10 puntos)

Dado el siguiente algoritmo:

```
void anf(bool* vector, int n, int ini){

       if(n > 1){

       int mitad = n / 2;

       anf(vector, mitad, ini + mitad);

       anf(vector, mitad, ini);

       for(int i = 0; i < n/2; i++)

            vector[ini + i + mitad] = vector[ini + i + mitad] && vector[i + ini];

    }

}
```

- Inicialmente se invoca a ***anf*** con los siguientes parámetros: `anf(vector, n, 0)`.
- **n** es la cantidad de elementos del arreglo *vector* y ya desde la invocación inicial es potencia de 2

Considerando como operación básica la *asignación* a un elemento de ***vector***

1. ¿De qué depende el costo (cantidad de asignaciones) del algoritmo ***anf***?
2. Calcule el costo en los casos mejor, peor y promedio.
3. Calcule los órdenes exactos de los tres casos justificando su razonamiento. (**Esta parte se corrige solamente si calculó los costos satisfactoriamente).**
4. ¿Si además de considerar la asignación también se consideran las lecturas del arreglo ***vector*** cambiaria alguno de los ordenes? (**Esta parte se corrige solamente si hizo satisfactoriamente la parte anterior**).

Ejercicio 4 (10 puntos)

1. Describa la forma general de la técnica **Greedy** en pseudocódigo. Explique cada paso.
2. Explique qué resuelve el algoritmo de *Prim* e indique las correspondencias con la forma general de la técnica que utiliza.

Problemas

Problema 1 (30 puntos)

a) (20 puntos) Un niño quiere invertir **todo ó parte** de su dinero que le ha dejado el ratón Pérez en golosinas. El vendedor del almacén del barrio tiene *g* tipos de golosinas diferentes y un vector v[0..g-1] ordenado en forma creciente según los precios de las golosinas, donde v[k] indica el valor del k-ésimo tipo de golosina. Se tiene además, un vector c[0..g-1] donde c[k] indica la cantidad de golosinas que se tienen del k-ésimo tipo.
Según el criterio del niño, el conjunto de golosinas ideal es aquel que minimiza el número de golosinas gastando la mayor cantidad de dinero posible.

Se pide dar una fórmula recursiva para solucionar el problema anterior según el criterio del niño. Llame a la fórmula recursiva *f* y *monto* a la cantidad de dinero que el niño tiene para comprar golosinas. Explicar qué representa el/los pasos base y cada paso recursivo de la solución, así como también qué representa la función f indicando sus índices. Se debe indicar la invocación de la función para resolver el problema.

b) (10 puntos)

b.1) Obtenga una expresión para T(n), resolviendo la siguiente ecuación de recurrencia:

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + (n-1)$ *(función piso y techo respectivamente)*

$T(1) = 0$

Suponga n potencia de un número conveniente.

b.2) Caso medio de MergeSort. Sabiendo que:

- el costo $T_A(n)$ en el caso medio de MergeSort (como el costo de cualquier algoritmo que ordena por comparaciones) cumple $T_A(n) \in \Omega(n \log n)$

- el costo $T_W(n)$ en el peor caso de MergeSort es igual al calculado en la parte (b.1), o sea, $T_W(n) = T(n)$

Demostrar que $T_A(n) \in \Theta(n \log n)$.

Problema 2 (30 puntos)

a) (12 puntos) Implemente un algoritmo en C* que reciba un grafo no dirigido $G = (V, E)$ y un vértice $w \in V$, que imprima en pantalla los vértices de V cuya distancia a w sea un número impar.

b) (18 puntos) Implemente un algoritmo en C*, utilizando DFS, que cuente todos los ciclos simples de un grafo dirigido. ***Atención:*** *a los efectos de la resolución de este ejercicio considere distintos aquellos ciclos que resultan de permutaciones cíclicas de vértices. Por ejemplo: el ciclo (a, b, c) se debe considerar distinto a los ciclos (b, c, a) y (c, a, b).*

Se disponen de los siguientes TADs. **En caso de necesitar alguna función/procedimiento auxiliar que no pertenezca a la especificación de los TADs anteriores deberá implementarla.**

```
#ifndef COLAENTEROS_H
#define COLAENTEROS_H

struct ColaEnteros;

ColaEnteros* ColaCrearVacia();
// Crea una cola de enteros vacia.

bool ColaEsVacia(ColaEnteros* c);
// Devuelve true si la cola es vacia y false en caso contrario.

void ColaEncolar (ColaEnteros* c, int n);
// Agrega el elemento n a la cola.

int ColaPrimero(ColaEnteros* c);
// Precondicion: !ColaEsVacia(c).
// Devuelve el primer elemento de la cola.

void ColaDesencolar(ColaEnteros* c);
// Precondicion: !ColaEsVacia(c).
// Elimina el primer elemento de la cola.

void ColaDestruir(ColaEnteros* c);
// Libera la memoria asociada a la cola.

#endif
```

```
#ifndef LISTAENTEROS_H
#define LISTAENTEROS_H

struct ListaEnteros;

ListaEnteros* ListaCrearVacia();
// Crea una lista de nodos vacia.

bool ListaEsVacia(ListaEnteros* l);
// Devuelve true si la lista es vacia y false en caso contrario.

void ListaAgregar (ListaEnteros*& l, int n);
```

```
// Agrega el elemento n al comienzo de la lista.

int ListaPrimero(ListaEnteros* l);
// Precondicion: !ListaEsVacia(l).
// Devuelve el primer elemento de la lista.

ListaEnteros* ListaResto(ListaEnteros* l);
// Precondicion: !ListaEsVacia(l).
// Devuelve un alias al resto de la lista.

void ListaDestruir(ListaEnteros* l);
// Libera la memoria asociada a la lista.

#endif
```

```
#ifndef GRAFO_H
#define GRAFO_H

#include "listaEnteros.h"

struct Grafo;

Grafo* GrafoCrearVacio(int maxNodos);
// Crea un grafo dirigido sin nodos ni arcos.
// El valor de maxNodos  especifica la cantidad maxima
// de nodos que puede contener el grafo.

int GrafoMaxNodos(Grafo *g);
// Devuelve la cantidad maxima de nodos que puede contener el grafo.

int GrafoCantNodos(Grafo* g);
// Devuelve la cantidad de nodos que tiene grafo.

int GrafoAgregarNodo(Grafo* g);
// Precondiciones: GrafoCantNodos(g) < GrafoMaxNodos(g).
// Agrega un nodo al grafo y devuelve su identificador, que es asignado de
// forma secuencial comenzando en 0.

void GrafoAgregarArco(Grafo* g, int i, int j);
// Precondiciones: 0 <= i < GrafoCantNodos(g), 0 <= j < GrafoCantNodos(g) y
// j no pertenece GrafoAdyacentes(g,i).
// Agrega un arco (i,j) al grafo.

ListaEnteros* GrafoAdyacentes(Grafo* g, int i);
// Precondiciones: 0 <= i < GrafoCantNodos(g).
// Devuelve la lista de nodos adyacentes al nodo i.
// La lista es la misma que forma parte del grafo.
// Por tanto, no debe ser modificada.

void GrafoDestruir(Grafo* g);
// Libera toda la memoria asociada al grafo.

#endif
```