# Solución Examen de Programación 3 y III (07/02/2012)

Instituto de Computación, Facultad de Ingeniería, UdelaR

## Parte Teórica Obligatoria

### Ejercicio 1 (10 puntos)

### 1)

*1) Desarrollo de definición de $f(n) \in O(g(n))$:
$(\exists K \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow f(n) \leq K * g(n))$

*2) Desarrollo de definición de $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$
Sii (definición límite de sucesiones)
$(\forall \varepsilon \in R^+)(\exists n_1 \in N)(\forall n \in N)\left(n \geq n_1 \rightarrow \left|\frac{f(n)}{g(n)} - 0\right| < \varepsilon\right)$

Partiendo de *2), operando y teniendo en cuenta que *f* y *g* son funciones positivas se llega a que

$(\forall \varepsilon \in R^+)(\exists n_1 \in N)(\forall n \in N)(n \geq n_1 \rightarrow f(n) < \varepsilon * g(n))$

Luego, se elige un $\varepsilon$ cualquiera. Esto verifica la definición $f(n) \in O(g(n))$ escrito en *1).

Resta probar que $g(n) \notin O(f(n))$

Lo anterior equivale a decir que no se cumple la afirmación
$(\exists K \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow g(n) \leq K * f(n))$

Sii (equivalencia lógica)

No existe $K \in R^+$ tal que $(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow g(n) \leq K * f(n))$

Para probarlo se supone por absurdo que existe $K \in R^+$ tal que
$(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow g(n) \leq K * f(n))$.

Por *2)
$(\forall \varepsilon \in R^+)(\exists n_1 \in N)(\forall n \in N)\left(n \geq n_1 \rightarrow \frac{1}{\varepsilon} f(n) < g(n)\right)$

Si tomo el $n_3 = \max(n_0, n_1)$ y $\varepsilon = \frac{1}{K}$ entonces se cumple
$(\forall n \in N)(n \geq n_3 \rightarrow K * f(n) < g(n))$ y $(\forall n \in N)(n \geq n_3 \rightarrow g(n) \leq K * f(n))$

Lo cual es una contradicción.

### 2)

Por propiedad de límite si $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $\lim_{n \to +\infty} \frac{g(n)}{f(n)} = 0$. Finalmente, por lo probado en la parte anterior $g \in O(f)$ y $f \notin O(g)$.

## Ejercicio 2 (10 puntos)

a) Se asume que los rangos horarios están ordenados en forma decreciente según el rating por unidad precio. Es decir, $r_i / p_i >= r_j / p_j$, para todo i <= j.

El algoritmo *greedy* podría ser de la siguiente manera:

```
int
calcular_rating (int N, int *p, int *r,  int P)
{
    int presupuesto = P;
    int rating = 0;

    int i = 0;
    while ((i < N) && (p[i] <= presupuesto))
    {
      rating += r[i];
      presupuesto -= p[i];
      i++;
    }
    return rating;
}
```

Con esta solución no siempre se obtiene el valor óptimo, como lo demuestra el siguiente contraejemplo:

N = 3

p = {1,2,3}

r = {6,10,12}

P = 5

En este caso la solución óptima tiene rating 22 (contratar los rangos horarios 1 y 2). Sin embargo, el algoritmo anterior retorna 16 (los rangos 0 y 1), ya que primero elije el que tiene mayor rating por unidad de precio, pero este elemento no pertenece a la solución óptima.

b)

$$R_j(c) = max(R_{j+1}(c), R_{j+1}(c - p_{j+1}) + r_{j+1}), \quad \forall c, j \quad 0 \le c \le P, \ 0 \le j < N$$
$$R_N(c) = 0, \quad \forall c \quad 0 \le c \le P$$

donde

$R_{j+1}(c)$: ganancia cuando no se contrata el espacio publicitario j+1.

$R_{j+1}(c-p_{j+1})$: ganancia caundo se contrata el espacio j+1.

$r_{j+1}$: ganancia del espacio j+1.

El resultado de rating óptimo es $R_0(c)$.

# Ejercicio 3 (10 puntos)

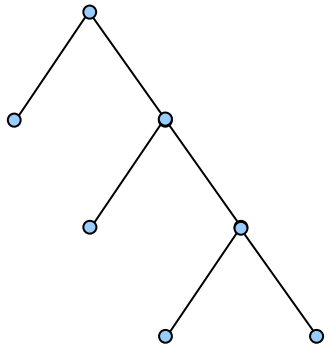
```
void minmax(int* a, int ini, int fin, int &min, int &max){
2           if(ini == fin){
3                 min = a[ini];
4                 max = a[fin];
5           }//if
6           else{
8                 int minDer, maxDer, minIzq, maxIzq;
9                 int mitad = fin+ini/2;
7                 minmax(a, ini, mitad, minDer, maxDer);
8                 minmax(a, mitad +1, fin, minIzq, maxIzq);
9                 if(minDer < minIzq) min = minDer;
10                else min = minIzq;
11                if(maxDer > maxIzq) max = maxDer;
12                else max = maxIzq;
13          }//else
14    }//minmax
```

# Ejercicio 4 (10 puntos)

a)

1) La afirmación es verdadera. Se demuestra por inducción en la altura $h$.
Paso base: $h = 0$.
El árbol tiene solo un nodo, que es externo. Se cumple $1 \leq 2^0 = 1$.
Paso inductivo:
Suponemos $h > 0$ y que la afirmación se cumple para árboles de decisión de altura $h-1$.
Como $T$ es un árbol de decisión (binario estricto) de altura $h$, los dos subárboles que cuelgan de la raíz son no vacíos y $h-1$ es una cota superior de su altura. Sean $m_L$ y $m_R$ la cantidad de nodos externos de los subárboles izquierdo y derecho respectivamente. Por la hipótesis inductiva tenemos
$m = m_L + m_R \leq 2^{h-1} + 2^{h-1} = 2^h$.

2) La afirmación es falsa. Sea $T$ el siguiente árbol.



El árbol $T$ tiene $4$ nodos externos. La suma de las profundidades de los nodos externos es $1 + 2 + 3 + 3 = 9$.
Como $9 > 4 \log 4 = 8$ se comprueba que no se cumpla la afirmación.

b)

Sea $n$ la longitud de la secuencia y representemos con $T$ el árbol de decisión correspondiente al algoritmo $A$. En $T$, los nodos internos son las comparaciones entre elementos de la secuencia. La cantidad de comparaciones en el peor caso es igual a $h$, la altura de $T$.
Consideremos una secuencia cuyos $n$ elementos sean distintos. Permutando la posición de los elementos tenemos que una cota inferior de la cantidad de secuencias posibles y por lo tanto de la cantidad de nodos externos es $n!$. Entonces, y de acuerdo al punto a.1)
$n! \leq m \leq 2^h$ , de donde $\log n! \leq h$.
Por lo tanto, llegamos a que la altura de $T$, que es igual a la cantidad de comparaciones en el peor caso, está en $\Omega(\log n!)$, y usando la ayuda, vemos que también está $\Omega(n \log n)$.

# Problemas

## Problema 1 (30 puntos)

**Parte 1.**
El plan de flechado de la ciudad se puede modelar mediante un grafo dirigido. En el que los nodos representan los puntos de interés y las aristas dirigidas representan las calles flechadas de la ciudad.

El proceso de verificar la circulación entre todo par de puntos de la ciudad es equivalente a determinar que el grafo que representa el plan es tal que existe un camino dirigido entre todo par de vértices. En dicho caso se dice que el grafo es fuertemente conexo. Una opción de determinar que el grafo es fuertemente conexo, es determinar la existencia de una única componente fuertemente conexa (CFC).

Una forma de determinar las componentes conexas de un grafo en forma eficiente es utilizar el algoritmo visto en el curso (Kosaraju).
a. Se realiza recorrida DFS de grafo original numerado en postorden
b. Se construye el grafo transpuesto del grafo original, con los mismos vértices y las mismas aristas pero con el sentido invertido.
b. Se realiza recorrida DFS del grafo transpuesto. Siempre que haya que comenzar un nuevo árbol se comienza por el vértice que tenga el numerador de postorden (el del paso a) mayor entre los no marcados.
Cada árbol resultado del procedimiento, corresponde a un CFC del grafo original.

Se verifica el requisito de circulación si existe una única CFC.

**Parte 2.**
Para resolver el problema se toma como base el algoritmo descrito en parte anterior con las siguientes simplificaciones:

a. el requisito de numerar en posorden puede ser sustituido por el de verificar que con una única inicialización (a partir de en un vértice inicial) de la recorrida DFS del grafo original se recorren todos los vértices. Si se tuviera que reinicializar la recorrida DFS en un vértice no visitado, el grafo ya no sería fuertemente conexo.
b. Si la recorrida del grafo transpuesto que se inicia en el vértice inicial de la primera (mayor postnum) no recorre todos los vértices del grafo transpuesto entonces hay más de una CFC, por lo que el grafo no es fuertemente conexo.

```
Bool VerificarCirculacion(Grafo* plan, int n)
{
    int* visitados = new int[n];
    int vertinic = 0;
    int cantvisit = 0;

    for (int i=0; i<n; i++)
        visitados[i]=0;
    DFS(plan, n, vertinic, visitados, cantvisit);
    if (cantvisit != n)
        return False;

    Grafo* transp = crearGrafoTransp(plan);


    for (int i=0; i<n; i++)
        visitados[i]=0;
    cantvert = DFS(transp, n, vertinic, visitados, cantvisit);
    if (cantvisit != n)
        return False;

    delete[] visitados;
    destruirGrafo(transp);
    return True;
}

void DFS(Grafo* g, int n, int v, int* visitados, int &cantvisit)
{
     visitados[v] = 1;
     cantvisit++;
     ListaEnteros* vecinos = obtenerAdyacenteGrafo(g, v);
```

```
      while (!esVaciaListaEnteros(v)){
         int w = ObtenerPrimeroListaEnteros(vecinos);
         vecinos = restoListaEnteros(vecinos);
         if (!visitados[w]){
             DFS(g, n, w, visitados, cantvisit);
         }
     }
}
```

# Problema 2 (30 puntos)

**Forma de solución:**
Tupla de largo fijo N, de la forma t=<x0, ,,, , $x_{N-1}$ > donde N es la cantidad de turistas y $x_i$ representa el plato asignado al turista i

**Restricciones Explícitas:**
El plato asignado debe ser uno de los preferidos por el turista.

- $x_i \in \{preferencias[i][j], 0 \leq j < 3\}$ para todo i en {0, …, N-1}.

Esto implica que $x_i \in [0..p-1]$.

**Restricciones Implícitas:**
Solo pueden asignarse hasta k platos del mismo tipo.

Sea $r_{xi}(j)$ la función indicatriz:
- 1, si $x_j = j$
- 0, en otro caso

- $\sum_{i=0}^{N-1} r_{xi}(j) \leq k$ para $j \in [0..p-1]$.

**Función Objetivo:**
Sea *T* el conjunto de soluciones.
Para cualquier tupla t = <x0, ,,, , $x_{N-1}$> perteneciente a *T* se define *pos_pref (i)* para $0 \leq i < N$, de tal modo que se cumple
*preferencias [i][ pos_pref (i)] = $x_i$.*
La satisfacción del grupo debe ser la mayor posible, o sea:

- $max_{t \in T} \sum_{i=0}^{N-1} (3/(1+pos\_pref(i)))$

**Podas:**
Se poda si al procesar el *i*-ésimo componente se cumple que la satisfacción acumulada de la tupla en construcción + 3 (N-i) no es mayor a la satisfacción de la tupla óptima encontrada hasta el momento.

**Implementación**

```
// Se asumen definidas y con scope global
int N;
int k;
int preferencias[N][3];


int* asignarPlatos(){
      int* asignacionOptima= new int[N];
      int asignacion[N];
      int asignadosCounter[p];

      for (int i=0; i<N; i++){
            asignacionOptima[i] = -1;
```

```
            asignacion[i] = -1;
            asignadosCounter [i] = -1;

      }

      asignarPlatosBK(asignacion,0,asignadosCounter,asignacionOptima);
      if (asignacionOptima[N-1]!=-1) //solucion completa
            return asignacionOptima;
      else { // No hay solucion
            delete[] asignacionOptima;
            return null;
      }
}

int* asignarPlatosBK(int * asignacion,int turista, int* asignadosCounter,int*
asignacionOptima)
{
      if (turista==N){ // fin de construccion, evaluacion de optimo
            if (ganancia(asignacion)> ganancia(asignacionOptima))
                  for (int j=0;j<N; j++)
                        asignacionOptima[j]=asignacion[j];
      }
      else //tupla en contruccion
      {
            for (int i= 0; i<3;i++){
                  int menu = preferencias[turista][i];
                  if (asignadosCounter[menu]<k){ // aun puedo asignar el menu
                        asignacion[turista]=menu;
                        asignadosCounter[menu]++;
                        asignarPlatos(asignacion,turista+1,
asignadosCounter,asignacionOptima);
                        // Rollback
                        asignacion[turista]=-1;
                        asignadosCounter[menu]--;
                  }
            }
      }
      return asignacionOptima;
}

// retorna el valor de satisfaccion total para la asignacion espeficada.
int ganancia(int* asignacion){
      int ganancia=0;
      for (int i=0; i<N; i++){
            ganancia+=3/(1+pref(i, asignacion[i]));
      }
}

// retorna la preferencia del turista por el menu especificado [0..2]
int pref(int turista, int menu){
      int pos=0;
      while (menu!=preferencias[turista][pos])
            pos++;
      return pos;
}
```