Examen de Programación 3 y III (11/02/2016)

Instituto de Computación, Facultad de Ingeniería, UdelaR

1. Este examen dura 4 horas y contiene **8** carillas. El total de puntos es 100 y se requieren 60 para su aprobación.
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*, el uso de *cout* y *cin y el tipo bool*.
3. NO se puede utilizar ningún tipo de material de consulta.
4. No se contestarán dudas durante la última media hora.

Se requiere:

- Numerar todas las hojas e incluir en cada una el nombre, la cédula de identidad, el número de hoja y el TOTAL de hojas.
- Utilizar las hojas de un solo lado y escribir con lápiz, iniciando cada ejercicio en hoja nueva.
- En la **carátula**: completar el índice indicando en qué hoja se comenzó la respuesta a cada problema/ejercicio y el total de hojas entregadas.

Parte Obligatoria (Ejercicios 1 a 4)

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 (10 puntos)

a) A5, A2, A1, A4, A3. En este sentido el algoritmo más eficiente es A5 (orden logarítmico) y el menos eficiente es A2 (orden exponencial).

b) f1(8) = 3 . 2^9
f2(8) = 2^7
f3(8) = 2^8
f4(8) = 2^9
f5(8) = 3 . 2^15

Entonces en orden creciente tenemos A2, A3, A4, A1, A5. Notar que el algoritmo más ineficiente en este caso es el logarítmico, y el exponencial es sólo 2 veces menos rápido que el lineal (que es más eficiente).

c) f1(16) = 2^12
f2(16) = 2^8
f3(16) = 2^16
f4(16) = 2^11
f5(16) = 2^17

Entonces en orden creciente tenemos A2, A4, A1, A3, A5. Notar que algoritmo logarítmico es todavía el peor de todos y que el cuadrático es más eficiente en este caso que el de orden n log n.

d) En realidad no hay ninguna contradicción. Cuando hablamos de que la complejidad de un algoritmo (por ejemplo la cantidad de comparaciones que hace cuando la entrada es de tamaño n) es de O(f(n)) estamos "escondiendo" constantes que pueden ser muy grandes. Estas constantes son muy importantes e influyen cuando el tamaño de la entrada es pequeño.

La complejidad asimptótica es muy útil cuando pensamos en entradas con un valor de n grande, de tal manera que la contribución mayor es f(n) y no la constante. Pero si tenemos valores exactos, tenemos mucho más información que cuando simplemente escondemos las constantes en expresiones asimptóticas del tipo O(f(n)).

En nuestros ejemplos, la constante que multiplica a log(n) es 32768 (2^15), mientras que la que multiplica a 2^n es 1. Entonces, para valores hasta n = 16 (2^4), el costo de la constante (32768) es mucho más importante que log(16) = 4, y entonces para este valor el algoritmo logarítmico es más ineficiente que el algoritmo exponencial. Sin embargo para valores más grandes (por ejemplo 256 = n^8) ya se nota que 2^15 log(256) = 2^23, y el exponencial tiene costo 2^256 que es muchísimo más grande (concordando con lo predicho en la parte a) cuando hablamos de complejidad asimptótica).

Por tal motivo tener información exacta es mucho más importante que tener información meramente asimptótica. Por supuesto, que tener información exacta es mucho más difícil, y muchas veces sólo tenemos la información sobre el comportamiento asimptótico. Este ejercicio muestra que según el valor de la entrada, usar algoritmos que se consideran "ineficientes" es mucho mejor que usar algoritmos que se consideran "eficientes", precisamente porque las constantes influyen mucho para estos valores.

Es importante aclarar que esta situación ocurre muchísimas veces en la práctica.

Ejercicio 2 (10 puntos)

El algoritmo de partición tiene dos punteros j, k. El puntero j se inicializa en la segunda posición del vector y se mueve hacia adelante parando cuando encuentra un elemento que sea mayor que el pivote. El puntero k se inicializa en la última posición del vector y se mueve hacia atrás parando cuando encuentra un elemento menor que el pivote. Luego se intercambian los valores, y continúa hasta que k < j. Finalmente se deja el

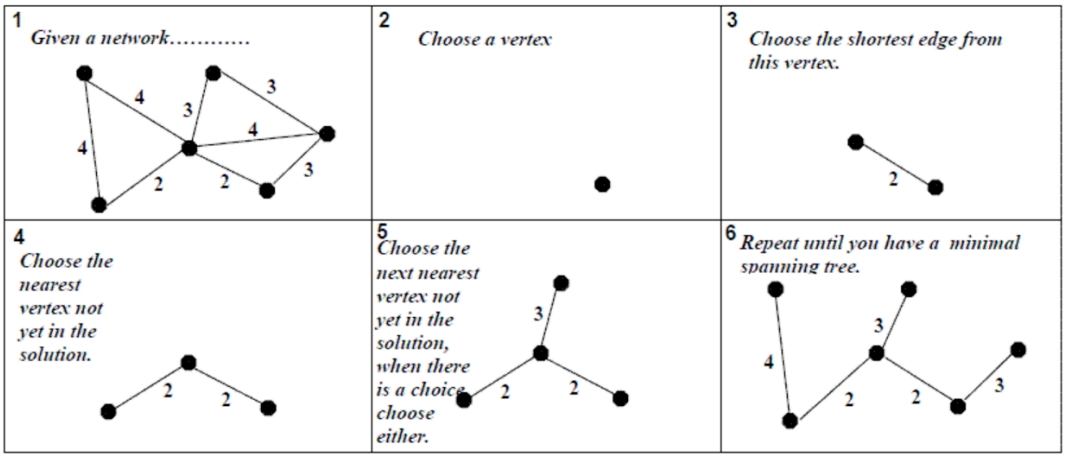
Ejercicio 3 (10 puntos)

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).

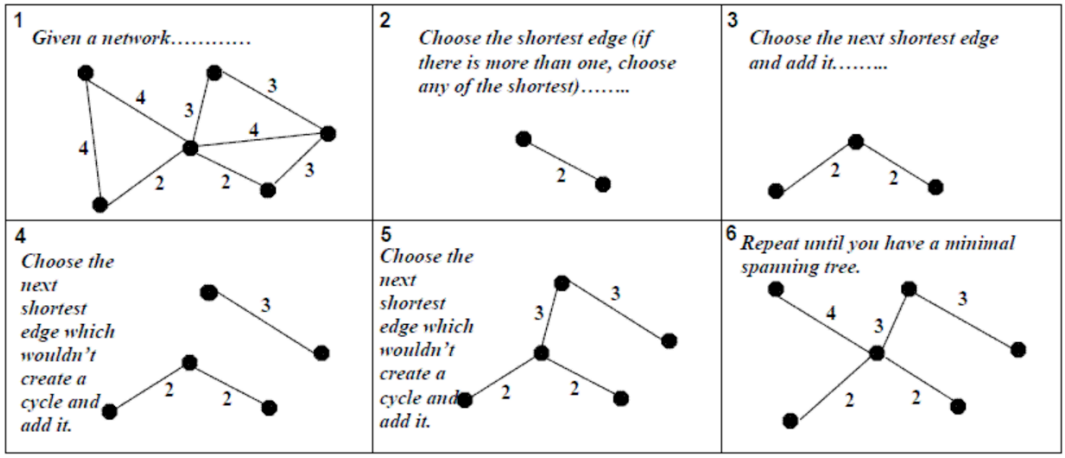
```
Greedy ( C )
{
    S := Φ
    while ( !esVacio(C) and  !esSolucion(S)  )
    {
            x :=  Select(C)
            C = C - {x}

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

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

2.

Prim's Algorithm



Kruskal's Algorithm



Ejercicio 4 (10 puntos)

Una empresa de software considera un grupo de n programadores para armar un equipo procurando maximizar la experiencia en desarrollo total del equipo formado. Para ello se cuenta con un presupuesto P y un mínimo de experiencia E que debe cumplir el equipo formado.

Asuma que se tienen definidas las siguientes funciones:

- e(k) indica la experiencia del programador k
- p(k) indica el costo del programador k.

Se pide:
Dar una fórmula recursiva f para solucionar el problema. Explicar brevemente que representan los índices que defina, cada paso base y recursivo de la solución.

Sea *f(i,p,e)* :

- i identificador de programador considerado en [0..n]
- p presupuesto disponible en [0..P]
- e experiencia necesaria para el equipo en [0..E]

Pasos Base

- No hay más programadores a considerar y se cumple con la experiencia
  f(i,p,e) = 0 si i=0 y e<=0
- No hay mas programadores a considerar y no se cumple con la experiencia
  f(i,p,e) = -inf si i=0 y e>0

Pasos Recursivos

- No se puede pagar al programador i con el presupuesto disponible
  $f(i,p,e) = f(i-1,p,e)$ *si* $i>0$ *y* $p(i)<p$
- Puede asignarse o no el programador i con el presupuesto disponible
  $f(i,p,e) = max\{f(i-1,p,e), f(i-1,p-p(i),e-e(i))+e\}$ *si* $i>0$ *y* $p(i)>=p$

Problemas

Problema 1 (30 puntos)

- El procedimiento sugerido no es correcto. Considerar el contraejemplo siguiente.

  a) A partir del grafo *G* de la figura se hace un recorrido DFS desde el nodo 1 obteniéndose el camino C_1 indicado.

  b) Al eliminar los nodos intermedios 2 y 3 de C_1 del grafo *G* se obtiene el grafo G^* sin aristas:

  c) Una recorrida DFS en G^* desde el nodo 1 no encuentra un nuevo camino entre V_1 y V_2, por lo que el procedimiento concluye que no hay dos caminos independientes entre V_1 y V_2, lo que es incorrecto ya que existen los dos caminos independientes $V_1 \rightarrow 2 \rightarrow V_2$ y $V_1 \rightarrow 3 \rightarrow V_2$

- El problema de hallar dos caminos nodo-disjuntos entre dos vértices dados es equivalente a hallar un ciclo que pase por dichos dos vértices. La idea del algortimo es lanzar una recorrida DFS a partir de uno de dichos vértices y determinar si hay un recorrido DFS que vuelva al vértice inicial y además pase por el otro vértice de interés.
  Necesariamente la recorrida DFS debe ser con desmarcaje de cada vértice en el posprocesamiento, de lo contrario el algoritmo no funciona pues podrían no inspeccionarse todos los posibles ciclos.
  Se utiliza un arreglo auxiliar de padres de los vértices en la recorrida DFS, el cual sirve para el caso de existir 2 caminos poder reconstruir dichos caminos.

```
bool Hay2Caminos(Grafo G,vertice V1, vertice V2, ListaVertices &C1, ListaVertices
&C2){
    v:vertice;
    visitados: arreglo[1..n] de bool;
    padres: arreglo[1..n] de enteros;
    encontre2: bool;
    Para cada vertice v de G
        visitados[v] = false;
        padres[v] = -1;
    Fin Para
    encontre2 = false;
    DFSHay2Caminos(G,V1,V1,V2,visitados,padres,encontre2);
    Si encontre2
       //armado del camino C1
       v = V2;
       C1 = InsertarVertice(C1,v);
       Mientras padres[v] != V1
           C1 = InsertarVertice(C1,padres[v]);
           v = padres[v];
       Fin Mientras
       C1 = InsertarVertice(C1,V1);
       //armado del camino C2
       v = V1;
       C2 = InsertarVertice(C2,v);
       Mientras padres[v] != V2
           C2 = InsertarVertice(C2,padres[v]);
           v = padres[v];
       Fin Mientras
       C2 = InsertarVertice(C2,V2);
    Fin Si
    return encontre2;
}

void DFSHay2Caminos(grafo G,vertice v,vertice V1,vertice V2,bool visitados[],
entero padres[],bool &encontre2){
```

```
    visitados[v] = true;
    Para cada vertice w adyacente a v en G
        Si !encontre2
            Si (w == V1) && (visitados[V2]) && !((v == V2) && (padres[V2]== V1))
                encontre2 = true;
                padres[w] = v;
            Sino
                Si !visitados[w]
                    padres[w] = v;
                    DFSHay2Caminos(G,w,V1,V2,visitados,padres,encontre2);
                Fin Si
            Fin Si
        Fin Si
    Fin Para
    visitados[v] = false;
}
```

Observación: en la solución presentada resulta necesario incluir la evaluación de la condición `!((v == V2) && (padres[V2]== V1))` en el código debido al caso particular en el que $v = V_2$ y se cumple además que V_1 y V_2 son adyacentes en *G*, pues si no se incluye ese testeo resultan indistinguibles los casos en que V_2 es el primer nodo de la recorrida DFS a partir de V_1, y por lo tanto todavía no se ha encontrado un ciclo, del caso en que V_2 es el penúltimo nodo de la recorrida antes de cerrar el ciclo con V_1 y por lo tanto sí se ha encontrado un ciclo que incluye a ambos vértices.

Problema 2 (30 puntos)

Forma de la tupla

$T = < t_0, \ldots, t_i, \ldots, t_D >$

Tupla de largo variable D, siendo D la cantidad de días que empleará el flete en realizar todas las entregas desde la *fechaArribo*.

$D < \max(Contenedores[j].fechaLimite) - fechaArribo \quad \forall j \in \{1..C\}$

Cada $t_i = < x_0, \ldots, x_{H_i} >$ es una tupla de largo variable $H_i < C + 1$ que representa el trabajo realizado por el flete en el día i, tanto los contenedores entregados como los regresos al puerto de Montevideo, en el correspondiente orden en que fueron realizados.

Cada x_j pertenece a $\{0..C\}$, representando el contenedor entregado si pertenece a $\{1..C\}$ y un regreso al puerto de Montevideo si es cero.

Restricciones explícitas

- Los elementos de la sub-tupla representan el id de un contenedor o cero en el caso de la vuelta al puerto

$t_{ij} \in \{0..C\} \ \forall i \in \{0..D\} \wedge j \in \{0..H_i\}$

- Cada contenedor es entregado antes de su fecha límite.

$\forall c \in \{1..C\} \ \exists t_{ij} = c \ / \ i \leq Contenedores[c].fechaLimite \ \wedge \ j \in \{0..H_i - 1\}$

- El flete comienza el primer día y termina el último en el puerto de Montevideo

$t_{00} = 0$

$t_{0H_0} = 0$

Restricciones Implícitas

- La suma de los tiempos de traslado entre todos los destinos visitados en el día, más la distancia del último destino del día anterior al primero de este día es menor a 8.

$\forall i \in \{0..D\}$ *se cumple*

$$\sum_{j=1}^{H_i} tiempoTraslado[destino_{ij-1}][destino_{ij}] + tiempoTraslado[destino_{i-1,H_i}][destino_{i0}] < 8 \times 60$$

$$Siendo\ destino_{ij} = \begin{cases} puertoMontevideo \ \ si\ t_{ij} = 0 \\ Contenedores[t_{ij}].destino \quad en\ otro\ caso \end{cases}$$

- No se repiten contenedores en las sub-tuplas.

$t_{ij} = t_{mn} \iff t_{ij} = 0 \vee (i = m \wedge j = n)$

- La cantidad de contenedores que transporta el flete a la vez nunca supera k.

Dado $t_{ij} = 0$ *con* $i \neq D \vee j \neq H_D$

$\exists\, t_{mn} = 0 \;/\; (m > i \;\vee\; m = i \;\wedge\; n > j)$ *que cumple:*

$$H_i - j + \sum_{f=i+1}^{m-1} H_f + n < k$$

Función Objetivo
Minimizar el costo de todos los traslados realizados por el flete.
$\min_{t \in solucion} \{costos(t)\}$, donde:

$$costos(t) = \sum_{i=1}^{D} \left(\sum_{j=1}^{H_i} costoTraslado[destino_{ij-1}][destino_{ij}] + costoTraslado[destino_{i-1,H_{i-1}}][destino_{i0}] \right) + \sum_{j=1}^{H_0} costoTraslado[destino_{0j-1}][destino_{0j}]$$

Predicado de Poda
Si se asignaron los contenedores a repartir hasta el día f, y quedan contenedores no repartidos que su fecha límite es menor a fechaArribo + f se poda la tupla.

$Dada\ T = < t_0, \ldots, t_f, \ldots >,\ si\ \exists\ c \in [1..C]\ /$

$Contenedores[c].fechaLimite < fechaArribo + f \;\wedge\; \exists\, t_{ij} = c\ con\ i \in [0.$