Solución Segundo Parcial de Programación 3 (29/11/2013)

Instituto de Computación, Facultad de Ingeniería

- Este parcial dura **4** horas y contiene 2 carillas. El total de puntos es **60**.
- 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* y el tipo `bool` predefinido en `C++`.
- NO se puede utilizar ningún tipo de material de consulta. Salvo que se indique lo contrario podrá usarse todo lo visto en el teórico, práctico y laboratorio sin demostrarlo, indicando claramente lo que se está usando.

Se requiere:

i. Numerar todas las hojas e incluir en cada una el nombre y la cédula de identidad.
ii. Utilizar las hojas de un sólo lado y escribir con lápiz.
iii. Iniciar cada ejercicio en hoja nueva.
iv. Poner en la carátula la cantidad de hojas entregadas, y un índice indicando en qué hojas respondió cada problema.

Ejercicio 1. (20 puntos)

a)

Definimos las siguientes funciones auxiliares que se utilizarán para todas las tuplas:

$$compatibles(t_a, t_b) = \begin{cases} 1 \quad si \quad t_a.fin < t_b.inicio \quad o \quad t_b.fin < t_a.inicio \\ 0 \quad en \quad otro \quad caso \end{cases}$$

Forma de solución 1

<u>Restricciones explícitas:</u>

- *Todos los voluntarios son válidos.*

$\forall t_i =< x_1,...,x_{A_i} >$ con $i \in \{1...N\}$ se cumple que $x_j \in \{1...V\}$ $\forall j \in \{1...A_i\}$

- *Los voluntarios asignados a una necesidad no se repiten.*

$\forall t_i =< x_1,...,x_{A_i} >$ con $i \in \{1...N\}$ se cumple que $x_j \neq x_h$ $\forall j,h \in \{1...A_i\} / \ j \neq h$

- *Se asigna la cantidad de voluntarios que la necesidad requiere*

$\forall t_i =< x_1,...,x_{A_i} >$ con $i \in \{1...N\}$, $A_i \in \{0, necesidades[i].cantidad\}$

- *El voluntario debe ser del área de especialización de la necesidad.*

$\forall t_i =< x_1,...,x_{A_i} >$ con $i \in \{1...N\}$,
voluntarios[x_j].areas[necesides[i].areasEsp] $= 1$, $\forall j \in \{1... A_i\}$

Restricciones implícitas:

- *Las necesidades asignadas a un voluntario no se superponen*

$\forall t_i =< x_1,...,x_{A_i} >$ y $t_j =< x_1,...,x_{A_j} >$ con $i, j \in \{1...N\}$ y $i \neq j$,

Si $\exists \quad x_l \in t_i \quad y \quad x_k \in t_j \quad / x_l = x_k \quad \Rightarrow \quad compatibles(t_i, t_j)$

Forma de solución 2

Restricciones explícitas:

- *Dominio de los elementos de la tupla.*

$\forall t_i =< x_1,...,x_V >$ con $i \in \{1 \ldots N\}$ se cumple que $x_j \in \{0,1\}$ $\forall j \in \{1 \ldots V\}$

Cuando $x_j = 1$ el voluntario j está asignado a la necesidad i.

- *Se asigna la cantidad de voluntarios que la necesidad requiere*

$\forall t_i =< x_1,...,x_V >$ con $i \in \{1 \ldots N\}$, $\sum_{j=1}^{V} x_j \in \{0, necesidades[i].cantidad\}$

- *El voluntario debe ser del área de especialización de la necesidad.*

$\forall t_i =< x_1,...,x_V >$ con $i \in \{1...N\}$,
voluntarios[j].areas[necesides[i].areasEsp] $= 1$ $\forall x_j \in t_i$ / $x_j = 1$

Restricciones implícitas:

- *Las necesidades asignadas a un voluntario no se superponen*

$\forall t_i =< x_1,...,x_V >$ y $t_j =< x_1,...,x_V >$ con $i, j \in \{1...N\}$ y $i \neq j$,

Si $\exists \quad k \in \{1...V\} \quad /$

$$x_k \in t_i \quad , \quad y_k \in t_j \quad y \quad x_k = y_k = 1 \quad \Rightarrow \quad compatibles(t_i, t_j) = 1$$

Forma de solución 3

Restricciones explícitas:

- *Dominio de los elementos de la tupla.*

$\forall t_i =< x_1,...,x_N >$ con $i \in \{1...V\}$ se cumple que $x_j \in \{0,1\}$ $\forall j \in \{1...N\}$

Cuando $x_j = 1$ el voluntario i está asignado a la necesidad j.

- *El voluntario debe ser del área de especialización de la necesidad.*

$\forall t_i =< x_1,...,x_N >$ con $i \in \{1...V\}$,
voluntarios[i].areas[necesides[j].areasEsp] $= 1$ $\forall x_j \in t_i$ / $x_j = 1$

- *Las necesidades asignadas a un voluntario no se superponen*

$\forall t_i =< x_1,...,x_N >$ con $i \in \{1...V\}$,

Si $\exists \quad x_l \in t_i \quad y \quad x_k \in t_i \quad / \quad x_l = x_k = 1 \quad \Rightarrow \quad compatibles(x_l, x_k) = 1$

Restricciones implícitas:

- *Se asigna la cantidad de voluntarios que la necesidad requiere*

$$\sum_{i=1}^{V} t_{i,j} \in \{0, necesidades[j].cantidad\}, \ \forall j \in \{1...N\}$$

Siendo $t_{i,j}$ el valor del elemento $x_j \in t_i$.

b)

Para la forma de solución 2:

$$asignada(t_i) = \begin{cases} 0 \quad si \quad \sum_{j=1}^{V} t_{i,j} = 0 \\ 1 \quad en \quad otro \quad caso \end{cases}$$

Siendo $t_{i,j}$ el valor del elemento $x_j \in t_i$.

La función objetivo es

$$f = \max_{t \in T} \left(\sum_{i=1}^{N} asignada(t_i) \right)$$

c)

Para la forma de solución 3:

$$asignado(t_i) = \begin{cases} 0 \quad si \quad \sum_{j=1}^{N} t_{i,j} = 0 \\ 1 \quad en \quad otro \quad caso \end{cases}$$

Siendo $t_{i,j}$ el valor del elemento $x_j \in t_i$.

La función objetivo es

$$f = \max_{t \in T} \left(\sum_{i=1}^{V} asignado(t_i) \right)$$

d)

Se proponen 2 posibles soluciones a esta parte. La primera es sin modificar la tupla solución y agregando una restricción implícita. La segunda es modificando la tupla solución y agregando las restricciones correspondientes.

d.i - Tanto la forma de la solución como las restricciones explícitas permanecen iguales, y se agrega una nueva restricción implícita:

Para cada $r \in \{1 \dots R\}$

Se define el conjunto A con necesidades que solicitan el recurso r
$A = \{n \in \{1 \dots N\} / necesidades[n].recurso = r\}$

Se define el conjunto B con las necesidades asignadas
$B = \{n \in \{1 \dots N\} / asignada(t_n) = 1\}$

Se define el conjunto H con las necesidades asignadas que solicitan un recurso r
H = { A intersección B }

Se debe cumplir que
$\forall J \subseteq H / \forall i, k \in J, compatible(i,k) = 0 \Rightarrow \quad \# J \leq recursos[r]$

d.ii - Cada componente de la tupla es un registro con dos campos. Uno es el vector de voluntarios y otro es un identificador de la unidad asignada, o 0 si no se le asigna. La unidad se representa con x_0. $t_i =< x_0, x_1, \cdots, x_N >$
Todas las RE y RI de las partes anteriores se mantienen.

Se agregan las siguientes restricciones explícitas:

Definimos $recursos[0] = 0$

- *Si una necesidad tiene voluntarios debe de tener el recurso que solicitó.*

Si $\sum_{j=1}^{V} t_{i,j} > 0 \wedge necesidade\,s[i].recurso \neq 0$

$\Rightarrow \chi_0 \in \{1 \ldots recursos[necesidades[i].recurso]\}$. $\chi_0 = 0$ en otro caso.

- *Si una necesidad tiene el recurso que solicitó debe de tener asignado voluntarios.*

Si $\chi_0 \neq 0 \Rightarrow \sum_{j=1}^{V} t_{i,j} > 0$

Se agregan las siguientes restricciones implícitas:

- *Si una necesidad tiene un recurso asignado no lo puede tener asignado otra necesidad que se superponga.*

$\forall t_i =< \chi_1,\ldots,\chi_V >$ y $t_j =< \chi_1,\ldots,\chi_V >$ con $i, j \in \{1 \ldots N\}$ y $i \neq j$ /

$necesidades[i].recurso = necesidades[j].recurso \neq 0 \wedge compatibles(i, j) = 0$

$\Rightarrow t_{i,0} \neq t_{j,0}$ o $t_{i,0} = t_{j,0} = 0$

Siendo $t_{i,0}$ el valor del elemento $\chi_0 \in t_i$ y siendo $t_{j,0}$ el valor del elemento $\chi_0 \in t_j$.

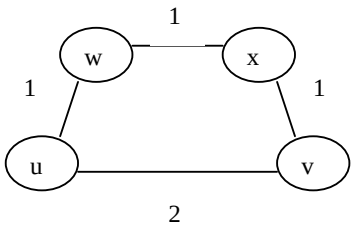
Ejercicio 2. (20 puntos)

Sea *G=(V,E)* un grafo no dirigido, conexo, sin lazos ni aristas múltiples, con costos positivos asociados a sus aristas. La cantidad de vértices es *N* (constante) y están identificados de 1 a *N*.

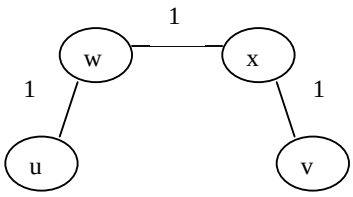
a) Sean *u* y *v* dos vértices cualesquiera de *G*. Se desea encontrar el camino de menor costo entre ambos vértices.
   1. Es posible encontrarlo aplicando el algoritmo de Kruskal?
   2. Es posible encontrarlo aplicando el algoritmo de Prim?

   En ambos casos demuestre su respuesta o de un contraejemplo.

   Considerando el grafo:



   el algoritmo de Kruskal devuelve el siguiente árbol de cubrimiento:



   El camino entre *u* y *v* es de costo 3, mientras que el camino en el grafo es de costo 2.
   Para el caso del algoritmo de Prim, usando el mismo grafo, se obtiene el mismo árbol de cubrimiento, por lo tanto tampoco se encuentra el camino de costo mínimo.

b) Sean *u* y *v* dos vértices cualesquiera de G. Demuestre que la solución óptima al problema de encontrar el camino de menor costo entre *u* y *v* cumple el Principio de Optimalidad.

   Asumamos que u,i1,i2,...,ik,v es el camino de menor costo (solución óptima) desde el vértice *u* al vértice *v*. Empezando en el vértice *u* se toma la decisión de pasar al vértice *i1*, luego de esta decisión el estado del problema esta definido por el vértice *i1* y encontrar un camino de menor costo de *i1* a *v*.

   Es claro que el camino de i1,i2,...,ik,v debe constituir un camino de menor costo de *i1* a *v*, sino fuera así, sea i1,r1,...,rq,v un camino de menor costo de *i1* a *v*.

   Por lo tanto u,i1,r1,...,rq,v es un camino de menor costo de *u* a *v* que el camino u,i1,i2,...,ik,v lo cual es absurdo porque el camino u,i1,i2,...,ik,v es la solución óptima y entonces el principio de optimalidad es aplicable a este problema.

c) Se tiene además una matriz *C* donde están los costos de las aristas del grafo, ésta contiene *MAXINT* si no hay arista entre dos vértices y *0* en las posiciones de la diagonal. Considerando

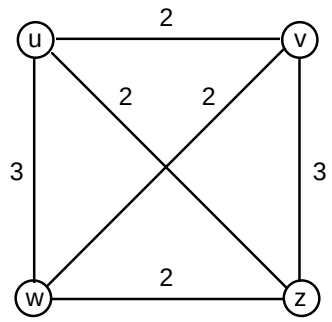
que no hay ciclos de costo negativo, plantee una recurrencia usando Programación Dinámica que, dados dos vértices, retorne el costo del camino de menor costo entre ambos vértices.

$F^k(i, j)$ es la longitud del camino de menor costo entre i y $j \in V$ utilizando únicamente los vértices numerados de 1 a k.

$F^0(i, j) = C(i, j)$ con $i, j \in V$

$F^k(i, j) = \min\{F^{k-1}(i, j), F^{k-1}(i,k) + F^{k-1}(k, j)\}$ con $k \geq 1$ y con $i, j \in V$

d) ¿Es posible encontrar un vértice *v* del grafo talque la aplicación del algoritmo de Dijkstra, tomando *v* como origen, retorne como resultado un árbol de cubrimiento de costo mínimo del grafo dado?
Demuestre su respuesta o de un contraejemplo.



El algoritmo de Dijkstra para cualquier vértice devuelve un árbol de cubrimiento de costo 7, que esta formado por las aristas incidentes al vértice. Sin embargo el árbol de cubrimiento de costo mínimo es 6.

Ejercicio 3. (20 puntos)

Sean D_n el conjunto de secuencias de n elementos (con elementos repetidos eventualmente) y el subconjunto D'_n de secuencias de n elementos sin repetidos. En lo que sigue considere que trabaja con D'_n .

En las partes a) y b) se puede utilizar: $\sum_{i=1}^{n} \log i \geq n \log n - 1.5n$

y para simplificar se puede considerar: $\sum_{i=1}^{n} \log i \cong n \log n$

a) **(6 puntos)** *Complejidad del problema de sorting en el peor caso – cota inferior en el peor caso*

Sea $F_w(n)$ la mejor cota inferior del costo en el peor caso para un algoritmo *A* genérico que ordena una secuencia (de $\boldsymbol{D'_n}$) por comparaciones de elementos 2 a 2.

Se sabe que:

- cualquier algoritmo *A* tiene asociado su árbol de decisión (AD).
- el costo del algoritmo *A* en su peor caso se denota *Tw(n)*
- *prof(AD) = k.* De lo anterior se concluye que $T_w(n) = k$
- si ***m*** es la cantidad de **nodos externos**, entonces por ***el lema 1*** dado en el teórico: $\boldsymbol{m \leq 2^k}$

i. Calcule Fw(n); justificando detalladamente los pasos de su razonamiento.
ii. Demuestre que la complejidad del problema de sorting es Θ(Fw(n)).

b) **(8 puntos)** *Complejidad del problema de sorting en el caso medio – cota inferior en caso medio*

Sea $F_A(n)$ la mejor cota inferior del costo en el caso medio para un algoritmo A genérico que ordena una secuencia (de $\boldsymbol{D'_n}$) por comparaciones de elementos 2 a 2.

Se sabe que:

- cualquier algoritmo *A* tiene asociado su árbol de decisión (*AD*).
- las secuencias de D'_n son equiprobables (como entrada para el algoritmo *A*).
- el costo del algoritmo *A* en su caso medio se denota $T_A(n)$.
- $\boldsymbol{k_i}$ es el largo del camino de la raíz al nodo externo ***i*** .
- $\boldsymbol{p_i}$ es la probabilidad de alcanzar el nodo externo ***i*** .
- Si ***m*** es la cantidad de **nodos externos**, por el ***lema 2*** visto en el teórico la sumatoria de profundidades de los nodos externos de *AD* es mayor o igual a $\boldsymbol{m \log m}$.

i. Calcule $F_A(n)$; justificando detalladamente los pasos de su razonamiento.
ii. Demuestre que la complejidad del problema de sorting es $\Theta(F_A(n))$.

c) **(6 puntos)** *QuickSort*

Dado el siguiente algoritmo PARTITION el cual elige un elemento llamado *pivote* y reordena el arreglo entre las posiciones *ini* y *fin* redistribuyendo los elementos adecuadamente; el parámetro *pospiv* devuelve la posición final del pìvote.

```
void PARTITION(int *a, int n, int ini, int fin, int &pospiv)
```

i. Escriba el algoritmo de ordenación QuickSort utilizando el PARTITION dado e indique qué necesita que se haga en PARTITION para que su algoritmo funcione.

ii. Indique qué técnica de diseño de algoritmos se utiliza, identificando los distintos ítems de la misma.
iii. Su algoritmo ¿es estable?. Justifique su respuesta.

a) *Complejidad del problema de sorting en el peor caso – cota inferior en el peor caso*

Obs. 1) Las instancias de D_n son infinitas, pero se puede observar que dado cualquier algoritmo y su AD asociado, todas las instancias que mantengan el mismo orden relativo entre sus elementos terminarán en el mismo nodo externo.

Obs. 2) En particular D_n contiene al conjunto D'_n formado sólo por las secuencias sin elementos repetidos. Agrupando en este caso aquellas instancias que tienen el mismo orden relativo se tienen n! posibles grupos de secuencias. Cada una de estas agrupaciones terminará en el mismo nodo externo y se tendrá, para D'_n , que m debe ser al menos n!.

Es decir: $n! \leq m$

Y por lo tanto: $n! \leq m \leq 2k$

Como se trata de hallar una cota inferior para el peor caso, se trata de encontrar una cota inferior para k y entonces se debe hallar $Fw(n) \leq k$ para cualquier AD.

Entonces: $n! \leq 2k$

Tomando logaritmos en base 2:

- $k \geq \log (n!)$
- $\log(n!) = \sum_{i=1}^{n} \log i$

Como $\sum_{i=1}^{n} \log i \geq n \log n - 1.5n$ entonces $k \geq n \log n - 1.5n$. $F_w(n) \in (n \log n)$

Concluimos que $F_w(n) \geq n \log n - 1.5n$.

De lo anterior se concluye que $F_w(n) \in \Omega(n \log n)$.

El mergeSort y heapSort tienen costo $\Theta(n \log n)$ peor caso concluyéndose que la complejidad del problema de sorting basada en comparaciones binarias es de $\Theta(n \log n)$.

b) *Complejidad del problema de sorting en el caso medio – cota inferior en caso medio*

Las secuencias de D'$_n$ también son infinitas, por lo que se deberá obtener un punto de vista útil para trabajar con las probabilidades. Efectivamente existen infinitas secuencias, sin embargo es posible agruparlas en grupos equivalentes desde el punto de vista de la ejecución del algoritmo como en el caso anterior (peor caso).

Como lo que se busca es el menor valor de m (cantidad de nodos externos) posible se tomará n! como dicho valor.

Por definición se tiene para todo algoritmo de la clase C:

$$T_A(n) = \sum_{S \in D_n'} p(S) * T(S)$$

Donde

p(S) es la probabilidad que se de la secuencia S∈ D'n y
T(S) es la cantidad de comparaciones que realiza el algoritmo para S

Como todas las secuencias son equiprobables los grupos de secuencias también son equiprobables y considerando cualquier árbol de decisión, todos los nodos externos resultan equiprobables (como fin de ejecución).

Planteamos la sumatoria en términos de nodos externos del árbol de decisión:

$$T_A(n) = \sum_{iexterno} p_i k_i$$

Donde:

pi es la probabilidad de alcanzar el nodo externo i o sea pi = 1/n!
ki es el largo del camino de la raíz al nodo i o sea cantidad de comparaciones en ese caso.

Se tiene entonces:

$$T_A(n) = \sum_{iexterno} p_i k_i = \frac{1}{n!} \sum k_i$$

Por el lema 2 se sabe que la sumatoria de profundidades de los nodos externos del AD es mayor o igual a m log m , siendo m la cantidad de nodos externos.

La función FA buscada cumplirá:

$$F_A(n) \leq T_A(n) = \frac{1}{n!} \sum_{iexterno} k_i$$

Como m = n!, y por Lema 2 se tiene:

$$F_A(n) \leq \frac{1}{n!} n! \log(n!)$$

Se concluye:

$$F_A(n) \geq n \log n - 1.5\, n$$

$$F_A(n) \in \Omega(n \log n)$$

El mergeSort, heapSort y quick sort tienen costo $\Theta(n \log n)$ caso promedio concluyéndose que la complejidad del problema de sorting basada en comparaciones binarias es de $\Theta(n \log n)$.

c)) *QuickSort*

```
void QuickSort(int *a, int n, int ini, int fin)
{
(1)  int pospiv;
(2)  if (ini<fin)
(3)  { // si hay más de un elemento
(4)       PARTITION(a, n, ini, fin, &pospiv);
(5)       QuickSort(A, ini, PosPivote-1);
(6)       QuickSort(A, PosPivote+1, fin);
(7)  } // if
} // fin QuickSort
```

Inicialmente el algoritmo se invoca como QuickSort(a,n,1,n).

PARTITION selecciona el elemento pivote y realiza la reordenación, devolviendo también el índice del arreglo donde se encuentra el pivote (pospiv).

La reordenación puede implicar:

- que los elementos menores o iguales que el pivote se colocan antes que éste en el arreglo y los elementos mayores que el pivote se colocan después que éste en el arreglo ó
- que los elementos menores que el pivote se colocan antes que éste en el arreglo y los elementos mayores o iguales que el pivote se colocan después que éste en el arreglo.

La técnica de diseño de algoritmos usada en el algoritmo *QuickSort* es ***Divide & Conquer***. Los ítems de esta técnica son los siguientes:

División: en las líneas (4), (5) y (6)

Combinación: se hace automáticamente por trabajar sobre el mismo arreglo.

Paso base: se encuentra ímplicito si no se cumple la condición de la línea (2)

Paso recursivo: en las líneas (2) a (7)

La estabilidad de *QuickSort* esa determinada por el PARTITION.

Hay varios puntos a tener en cuenta para determinar la estabilidad de *QuickSort*, para comenzar, se encuentra la selección del pivote. Este es uno de los factores decisivos, ya si se cuenta con un algoritmo aleatorio para determinar el pivote no se puede determinar la estabilidad de *QuickSort*.

Luego, el pivote particiona el conjunto de elementos en dos sub conjuntos, los elementos menores que él y los elementos de mayor valor que el pivote. Lo que suceda con los elementos iguales (es decir si se colocan juntos a los menores que el pivote o junto a los mayores que el pivote), en conjunto con la política de selección del pivote resultan en la estabilidad del algoritmo.