Solución Examen de Programación 3 y III (14/07/2012)

Instituto de Computación, Facultad de Ingeniería, UdelaR

Parte Teórica Obligatoria

Ejercicio 1 (10 puntos)

a) El mejor caso se da cuando los primeros $\frac{n}{2}+1$ lugares del arreglo tienen elementos menores a *x*. Como $A[i] < x$ se cumple con $0 \le i < \frac{n}{2}+1$, y cada vez que se cumple, la variable *cant* se incrementa en 1, luego de $\frac{n}{2}+1$ iteraciones, se dejará de cumplir la condición del *while* y no se vuelve a realizar la comparación $A[i] < x$. En total la comparación $A[i] < x$ se realizará solamente $\frac{n}{2}+1$ veces.

El peor caso, es cualquier caso en el que no hayan más de la mitad de los elementos menores, en dicho caso la condición del *while* siempre se cumple, con $0 \le i < n$, y cada vez se realiza la comparación $A[i] < x$.

b) $T_W(n) = \frac{n}{2}+1$.

$T_B(n) = n$.

Ejercicio 2 (10 puntos)

a) El algoritmo de Floyd se basa en la técnica de Programación dinámica. Este resuelve el problema de encontrar el camino de menor costo entre todo par de vértices de un grafo.

b) Primero se inicializa la matriz de costos con los costos de las aristas del grafo y la matriz de caminos.

$$D^0 = \begin{pmatrix} 0 & 4 & 6 & \infty \\ 4 & 0 & -2 & 5 \\ 6 & -2 & 0 & 1 \\ \infty & 5 & 1 & 0 \end{pmatrix} \qquad P^0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Se actualizan los caminos de menor costo y sus respectivos costos pasando por el vértice 1.

$$D^1 = \begin{pmatrix} 0 & 4 & 6 & \infty \\ 4 & 0 & -2 & 5 \\ 6 & -2 & 0 & 1 \\ \infty & 5 & 1 & 0 \end{pmatrix} \qquad P^1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Se actualizan los caminos de menor costo y sus respectivos costos pasando por los vértices 1 y 2.

$$D^2 = \begin{pmatrix} 0 & 4 & 2 & 9 \\ 4 & 0 & -2 & 5 \\ 2 & -2 & 0 & 1 \\ 9 & 5 & 1 & 0 \end{pmatrix} \qquad P^2 = \begin{pmatrix} 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

Se actualizan los caminos de menor costo y sus respectivos costos pasando por los vértices 1, 2 y 3.

$$D^3 = \begin{pmatrix} 0 & 4 & 2 & 3 \\ 4 & 0 & -2 & -1 \\ 2 & -2 & 0 & 1 \\ 3 & -1 & 1 & 0 \end{pmatrix} \qquad P^3 = \begin{pmatrix} 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 3 \\ 2 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 \end{pmatrix}$$

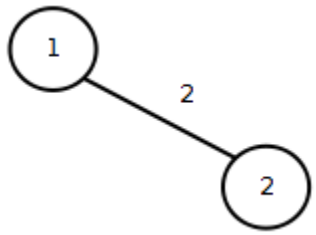
Se actualizan los caminos de menor costo y sus respectivos costos pasando por todos los vértices del grafo.

$$D^4 = \begin{pmatrix} 0 & 4 & 2 & 3 \\ 4 & 0 & -2 & -1 \\ 2 & -2 & 0 & 1 \\ 3 & -1 & 1 & 0 \end{pmatrix} \qquad P^4 = \begin{pmatrix} 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 3 \\ 2 & 0 & 0 & 0 \\ 3 & 3 & 0 & 0 \end{pmatrix}$$

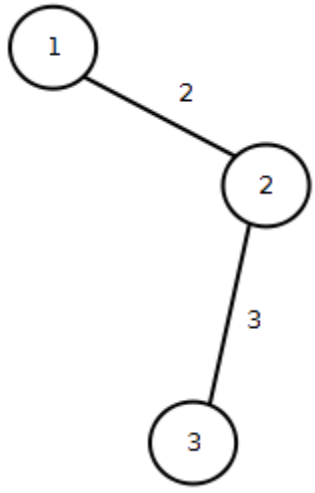
Ejercicio 3 (10 puntos)

a) Son algoritmos voraces que sirven para encontrar el árbol de cubrimiento de costo mínimo de un grafo.

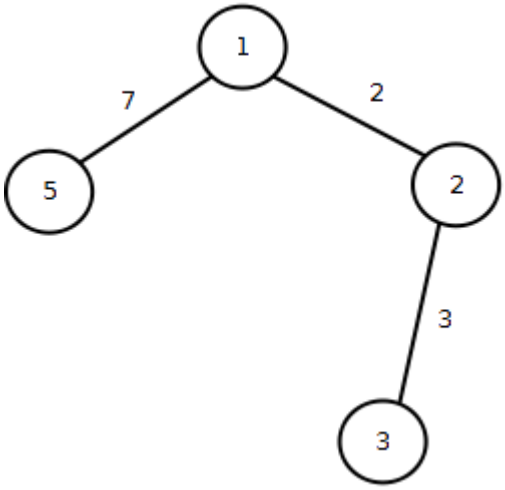
b) Se comienza con un árbol vacío, luego se ordenan todas las aristas por costos y se van agregando al árbol comenzando por la de menor costo de forma tal que no se generen ciclos.
De esta forma se agregan en el siguiente órden
1º



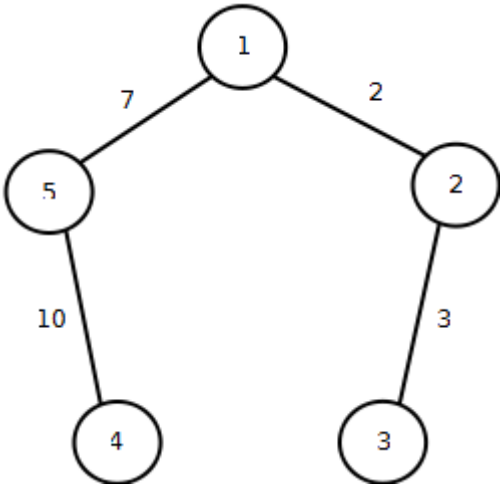
2º



3º



4º



Ejercicio 4 (10 puntos)

a)

```
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;
	}
}
```

b)

```
void SelectionSort (int* A, int n){
	int i, j, posmin, tmp;
	for (i = 0; i < n-1; i++){
		posmin = i;
		for (j = i+1; j < n; j++){
			if (A[j] < A[posmin])
				posmin = j;
		}
		// Intercambio de elementos
		tmp = A[i];
		A[i] = A[posmin];
		A[posmin] = tmp;
	}
}
```

c)

Insertion Sort

Primer paso:	5 < 3 NO	Arreglo: 2,1,3,5,4
First = 2	Arreglo: 2,3,5,1,4	1 < 2 SI
Arreglo: 2,3,5,1,4	**Cuarto paso:**	Arreglo: 1,2,3,5,4
Segundo paso:	First = 1	**Quinto paso:**
First = 3	1 < 5 SI	First = 4
3 < 2 NO	Intercambian 5 y 1	4 < 5 SI
Arreglo: 2,3,5,1,4	Arreglo: 2,3,1,5,4	Intercambian 5 y 4
Tercer paso:	1 < 3 SI	Arreglo: 1,2,3,4,5
First = 5	Intercambian 3 y 1	4 < 3 NO

Se hacen 7 comparaciones y 4 intercambios, por lo que el esfuerzo total del insertion sort para ordenar de forma ascendente el arreglo es de 15 unidades.

Selection Sort

Primer paso:	Posmin = 3	Posmin = 1
Posmin = 0	4 < 1 NO	5 < 3 NO
3 < 2 NO	Intercambian 1 y 2	2 < 3 SI
5 < 2 NO	Arreglo: 1,3,5,2,4	Posmin = 3
1 < 2 SI	**Segundo paso:**	4 < 2 NO

Intercambian 2 y 3	Posmin = 3	Posmin = 3
Arreglo: 1,2,5,3,4	4 < 3 NO	4 < 5 SI
Tercer paso:	Intercambian 3 y 5	Intercambian 4 y 5
Posmin = 2	Arreglo: 1,2,3,5,4	Arreglo: 1,2,3,4,5
3 < 5 SI	**Cuarto paso:**	

Se hacen 10 comparaciones y 4 intercambios, por lo que el esfuerzo total del selection sort para ordenar de forma ascendente el arreglo es de 18 unidades.

El insertion sort implicaría menos esfuerzo en ordenar el arreglo que el selection sort.

Problemas

Problema 1 (30 puntos)

Utilizaremos la función delta de Kronecker: $\delta(i, i) = 1,\ \delta(i, j) = 0 \text{ si } i \neq j$.

a)

1) No se puede asegurar. Como ejemplo supongamos que hay dos objetos, ambos con precio base 100 y un único comprador que ofrece 100 por cada objeto, pero su límite es menor que 200.
2) Sí, siempre es posible. Se sabe que para cada $j \in \{1 \ldots n\}$ existe un comprador x_j tal que

$$OFERTAS[xj][j] \geq BASE[xj].$$

Por lo tanto $\langle x_1, \ldots, x_n \rangle$ cumple con la condición de que todos los objetos sean vendidos. Además como para todo comprador el límite es mayor o igual a la suma de todas sus ofertas, la suma de todas las ventas asignadas a él no puede superar el límite:

$$\sum_{j=1}^{n} OFERTAS[x_j][j] . \delta(i, x_j) \leq \sum_{j=1}^{n} OFERTAS[i][j] \leq LIMITE[i],\ i \in \{1 \ldots m\}.$$

b)

No es posible resolverlo con un algoritmo ávido. Supongamos que hay dos objetos, ambos con precio base 100 y dos compradores. El primer comprador está dispuesto a pagar 500 por el primer objeto y 100 por el segundo. Su límite total es 600. El segundo comprador ofrece 700 por el primero, 500 por el segundo, y su límite total es 700.

	Obj. 1	**Obj. 2**		**Limite**
Comprador 1	500	100		600
Comprador 2	700	500		700

La solución óptima asigna el primer objeto al primer comprador y el segundo objeto al segundo. El total de las ventas es 1000. Pero una estrategia ávida asigna el primer elemento al segundo comprador. No puede asignarle también el segundo porque no lo permite su límite. Por lo tanto el segundo objeto es asignado al primer comprador y el total de las ventas es 800.

c)

Forma.

Tupla $\langle x_1, \ldots, x_n \rangle$ de largo fijo, ***n***, donde x_j es el número que identifica al comprador al que se asigna el objeto $j \in \{1 \ldots n\}$.

Restricciones explícitas.

Cada elemento de la tupla es un número de comprador:

$$1 \leq xj \leq m,\ j \in \{1 \ldots n\}.$$

El objeto debe asignarse a un comprador que ofrezca un valor mayor o igual al precio base:

$$OFERTAS[xj][j] \geq BASE[j],\ j \in \{1 \ldots n\}.$$

Restricciones implícitas. Ningún comprador puede pagar por los objetos que compra más de su límite:

$$\sum_{j=1}^{n} OFERTAS[i][j] . \delta(i, x_j) \leq LIMITE[i],\ i \in \{1 \ldots m\},$$

Función objetivo. Para cualquier tupla $t = \langle x_1, \ldots, x_n \rangle$ definimos la función Venta(t):

$$Venta(t) = \sum_{j=1}^{n} OFERTAS[x_j][j]$$

El objetivo es maximizar la venta:

$f = \max_{t \in S}(Venta(t))$, siendo ***S*** el espacio de soluciones.

Predicados de poda. Se poda la construcción de $t = \langle x_1, \ldots, x_n \rangle$ si al asignar un componente de ***t*** se tiene que la suma de lo que ya se vendió más el máximo de lo que se puede vender no supera la venta que se obtiene con la mejor solución ya encontrada:

$$\sum_{j=1}^{h} OFERTAS[x_j][j] . + \sum_{j=h+1}^{n} MAYOR[j] \leq \text{Venta (s)}, \; h \in \{1 \ldots n-1\},$$

siendo ***s*** la mejor solución encontrada antes de procesar ***t***.

d)

1) Suponemos que la mejor solución hasta ahora es $s = \langle y_1, \ldots, y_n \rangle$. La formalización de la condición es:

$$\sum_{j=1}^{h} OFERTAS[x_j][j] < \sum_{j=1}^{h} OFERTAS[y_j][j].$$

No es predicado de poda. Se puede ver con el ejemplo del item **b**. Sea ***s*** la tupla que se obtiene con el algoritmo ávido, h = 1, y $x_1 = 1$. Se cumple la desigualdad: 500 < 700. Pero si se podara entonces se perdería la solución óptima.

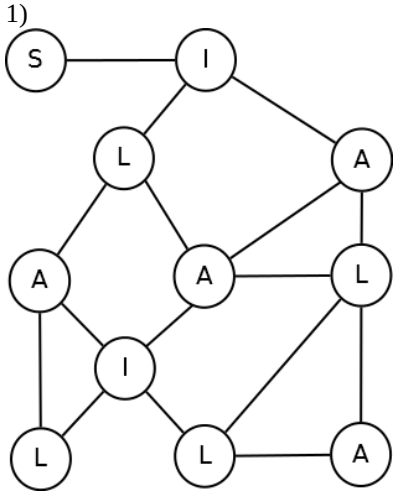
2) Según el predicado existe $k \in \{1 \ldots m\}$ que cumple:

$$\sum_{j=1}^{h} OFERTAS[x_j][j] . \delta(k, x_j) + BASE[h+1] > LIMITE[k].$$

No es predicado de poda. Si esto se cumple al comprador ***k*** no se le pueden asignar más objetos pero nada impide que los restantes objetos sean asignados a los otros postores. Esto sucede en el ejemplo del item ***b*** al construir la tupla que se obtendría con el algoritmo ávido. El primer objeto se asigna al segundo comprador y se cumplen las condiciones de este punto: 700 + 100 > 700. Sin embargo la tupla puede completarse asignando el segundo objeto al primer comprador.

Problema 2 (30 puntos)

1)



2)

```
void simMutacionVaricela(Grafos poblacion, int origen){
	int N = cantNodosGrafo(poblacion);
	int* visitados = new int[N];
	for (int i = 0; i < N; i++)
		visitados[i] = 0;
	// cola de procesamiento
	ColaEnteros queue = crearCola();
	encolar(queue, origen);
	visitados[origen] = 1;
	// cola que indica el estado de la fuente de contagio correlativa a queue
	ColaEnteros estadofuente = crearCola();
	encolar(estadofuente, LATENTE);

	while (!esVacia(queue)){
		int id= obtenerPrimero(queue);
		int estado_fuente= obtenerPrimero(estadofuente);
		// es huesped
		if (!esInmune(poblacion,id)){
			int estadoActual = estadoAlterno(estado_fuente);
			actualizarEstado(poblacion, id, estadoActual);
			ListaEnteros ady = obtenerAdyancentes(poblacion,id);
			while (!esVacia(ady)){
				int ind = obtenerPrimero(ady);
				if (!visitados[ind]){
					encolar(queue, ind);
					visitados[ind]=1;
					encolar(queue, estadoAlterno(estadoActual);
				}
			}
		}
	}
}

int estadoAlterno(int estado){
	if(estado==LATENTE)
		return ACTIVO;
	return LATENTE;
}
```