Solución Segundo Parcial de Programación 3 (26/11/2012)

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)

Una empresa de distribución de mercaderías desea automatizar la asignación de carga de sus vehículos de manera de **utilizar exactamente** su capacidad máxima de carga (por vehículo). La capacidad de cada uno de los vehículos es de C m^3. . La empresa cuenta con **n+1** productos, $0 \leq i \leq n$, cuyo volumen se encuentra definido en T[i], siendo T[i] < C. La empresa cuenta con un stock Q[i] de cada producto. Para reducir costos de manipulación de stock, se desea que la asignación por vehículo minimice **la cantidad total de unidades de todos los productos** que transporta. En esta instancia interesa **solamente** solucionar el problema de asignar la carga de un **único vehículo**. Tampoco se maneja el concepto de pedido, sólo la carga de un camión lo mejor posible.

Notar que el problema puede no tener solución. Como ejemplos considere los casos T = [4,5], Q[1,5] y C = 11 o C = 12 para los que no hay solución.

a) (14 puntos) Formalizar el problema aplicando **Programación Dinámica,** especificando la recurrencia para solucionar el problema de asignar carga a **un** único vehículo mediante la fórmula recursiva ***f(p, c)***, donde ***p*** indica el uso de los productos desde el ***0*** hasta el ***p*** y ***c*** indica la capacidad disponible. Asuma que los volúmenes de los distintos productos se encuentran ordenados de forma creciente en ***T***. En caso de que no haya solución al problema, la recurrencia debe retornar $+\infty$. Tenga en cuenta que no necesariamente todo el stock es asignado. Como justificación de la recurrencia propuesta describa claramente qué se considera en cada ítem. Indique cómo debe invocarse la recurrencia de forma de resolver el problema pedido.

b) (6 puntos) (sólo se tendrá en cuenta esta parte si realizó la parte (a))Describa la estructura de datos necesaria para resolver la recurrencia de la parte anterior. Implemente en forma iterativa, en C*, el algoritmo que la resuelve e indique el orden del tiempo de ejecución para el peor caso del algoritmo implementado, en función de *n* y *C*.

Se considera $f(p, c)$ a la cantidad mínima de productos para asignar al espacio $c \leq C$ utilizando los productos entre 0 y p.

Los casos base son:

$$f(p,0) = 0 \quad 0 \leq p \leq n$$

$$f(0,c) = \begin{cases} k, & \text{si } \exists\, k \in N \,/\, c = k.T[0] \wedge k \leq Q[0] \\ +\infty, & \text{en otro caso} \end{cases} \qquad c \neq 0$$

En la recurrencia las alternativas corresponden a considerar el mínimo entre no poner el elemento p o poner k unidades, siendo $k < Q[p]$. El segundo corresponde a no poder poner una unidad de p en el vehículo por lo que se descarta continuando con otro producto (los casos pueden ser simplificados en uno incluyendo k=0 en el 1ero.).
Pasos recursivos

$$\text{Para todo} \quad 1 \leq p \leq n \quad \text{y para todo} \quad 1 \leq c \leq C,$$

$$f(p,c) = \begin{cases} \min\{f(p-1,c), f(p-1,c-kT[p]) + k\}, \forall\, k \,/\, 0 < k \leq Q[p]\} \wedge k.T[p] < c, & \text{si } T[p] \leq c \wedge Q[p] > 0 \\ f(p-1,c), & \text{en otro caso.} \end{cases}$$

La solución entonces se obtiene con $f(n, C)$.

b) Se requiere una matriz de tamaño $(n+1)C$ de valores enteros. En la posición (p, c) se guardará la cantidad mínima de unidades necesaria para cargar de espacio c utilizando las productos entre 0 y p.

c) El código que implementa la solución es el siguiente:

```
int menorUnidadesEnCarga (int* T, int* Q, int n, int C, int** f)
{
      int p, c;
      for (p = 0; p <= n; p++)
            f[p][0] = 0;
      for (c = 1; c <= C; c++)
            if (c % T[0] == 0)
                  f[0][c] = c / T[0];
            else
                  f[0][c] = INT_MAX;
      for (p = 1; p <= n; p++)
            for (c = 0; c <= C; c++)
            {
                  f[p][c] = f[p-1][c];
                  for(k = 0; (k <= Q[i] && k*T[p] <= c); k++)
                     if ((f[p][c] > (f[p-1][c - T[i]*k] + k))&&
                      (d[p-1][c - T[p]*k] != INT_MAX))

                        d[p][c] = d[p-1][c - T[p]*k] + k;

            }
       return p[n][c];
}
```

d) No es necesario especificar el peor caso porque siempre se rellenan todas las casillas de la matriz, como el costo de rellenar cada casilla es constante, decimos que O (menorCarga)$_{\text{peor caso}} = n$xC.

Ejercicio 2. (20 puntos)

Parte a) (5 puntos)

1) Describa la forma general de la técnica **Greedy** explicando cada paso.

Greedy es una técnica que permite construir algoritmos para resolver un problema de optimización, que se basa en generar una solución utilizando una secuencia de decisiones tal que cada una es un óptimo local, logrando un óptimo global. Esa secuencia de decisiones se determina de acuerdo a una estrategia predefinida que indica cómo tomar la decisión en cada paso. Para que se considere válida dicha estrategia, debe probarse que efectivamente conduce al óptimo global buscado.

Si se tiene:
S: es la solución
C: es el conjunto de candidatos
Selección: es la función encargada de elegir el mejor elemento de C localmente
Se puede describir la técnica con el siguiente seudocódigo:

```
(1)     S = Φ
(2)     Mientras ( !esVacio(C) and  !esSolucion(S)  )
(3)     x = Seleccion(C)
(4)     C = C - {x}
(5)     if ( esFactible (S U {x} )
(6)           S = S U {x}
(7)     if ( esSolucion(S))
        return S
 else
        return Φ
```

2¿Qué problema resuelve el algoritmo de ***Kruskal***? Describa la estrategia e indique las correspondencias entre el algoritmo y la forma general de la técnica que utiliza.

El Algoritmo de Kruskal resuelve el problema de encontrar el árbol de cubrimiento de costo mínimo de un grafo (ACCM). Lo hace siguiendo la estrategia sugerida por la propiedad MST: elegir la arista de mínimo costo entre dos conjuntos de vértices no conectados entre sí.
Dado un grafo de n vértices parte de un grafo con los mismos n vértices pero sin aristas o sea se parte de un grafo con n árboles (un bosque). Las aristas se ordenan por su costo en una lista. En cada paso va agregando la arista de menor costo de la lista al bosque de forma de unir dos árboles y sin generar ciclos. Al final, si el grafo es conexo debe quedar un único árbol.

```
kruskal (Grafo G, Grafo &bosque)
{
      ListaAristas E;
(1)   arbol = CrearGrafo();
      E = OrdenarAristas(G);
(2)   mientras (bosque tiene menos de n-1 aristas) and (!Vacio(E))
(3)   sea e in E una arista de minimo costo
(4)       Borrar(E,e);
(5)       Si e no crea un ciclo en bosque entonces
(6)          AgregarArista(bosque,e);
      fin mientras
}
```

El ítem 7 de la parte (a1) queda controlado al vaciarse el conjunto de aristas o llegar a construir el árbol con sus n-1 aristas y la solución se devuelve como parámetro.

Parte b) (15 puntos)

1) Implemente el algoritmo `MergeSort` en un lenguaje de alto nivel (C* y/o seudocódigo como visto en el material teórico es suficiente).

Para la parte (1) se aceptaba el algoritmo visto en el material teórico donde se usa un algoritmo MERGE que recibe dos secuencias ordenadas y las intercala devolviendo una única secuencia ordenada. Luego se define un cabezal asumiendo que el MERGE se realiza sobre el mismo arreglo y las subsecuencias se indican mediante los índices de comienzo y fin de cada una. Finalmente se da el algoritmo MergeSort en sí que resuelve el problema dado mediante D&C.

Entrada: Secuencias *S1 y S2* a intercalar

Salida: Secuencia ordenada

MERGE: SECxSEC → SEC

MERGE(S1,S2)=
Si Vacia(S1)⇒ S2
Sino
Si Vacia(S2)⇒ S1
Sino
Si Primero(S1)< Primero(S2)
InsFront(Primero(S1), MERGE(Resto(S1),S2))
Sino
InsFront(Primero(S2), MERGE(S1, Resto(S2))

Cabezal a utilizar:

```
void MERGE( arreglo &A, int ini1, int fin1, int ini2, int fin2);
```

```
void MergeSort( arreglo &A, int p, int q){
int medio;

if (p<q) {       // quedan al menos 2 elementos
 medio = (p+q) / 2;
 MergeSort(A, p, medio);
 MergeSort(A, medio+1, q);
 MERGE(A, p, medio, medio+1, q);
 } //if
} // fin Mergesort
```

2) Defina el concepto de árbol de decisión.

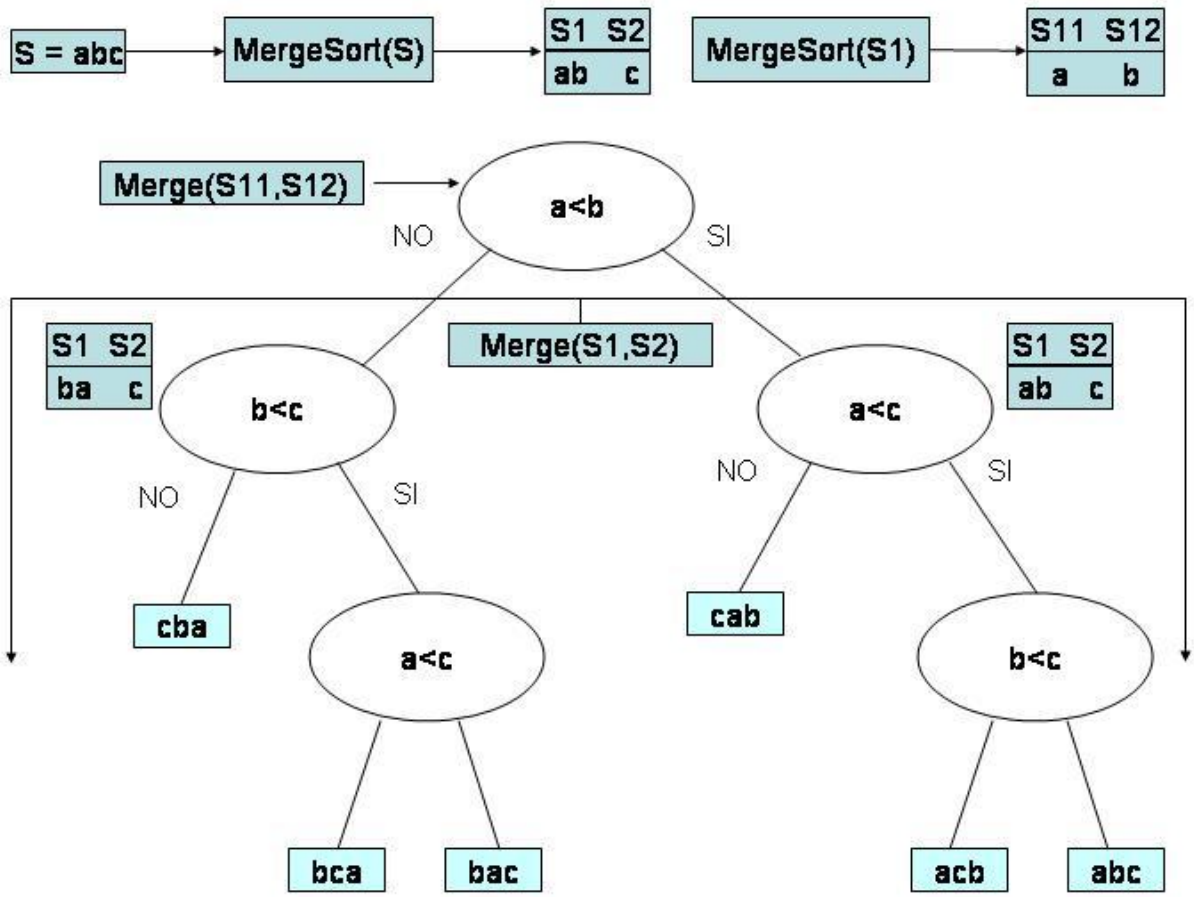
(tomado de los apuntes de teórico)

Un *Árbol de Decisión* (AD) es un **árbol binario estricto** (cada nodo tiene 2 hijos o ninguno) donde los **nodos internos tienen 2 hijos y los externos (hojas) no tienen ninguno**.

Estos árboles, vistos como grafos, son una herramienta para modelar la ejecución de algoritmos que resuelvan un problema dado. En este marco, los nodos internos representan operaciones y los externos representan el final de la ejecución de un algoritmo, cada uno para una determinada entrada.

Se adoptará como convención que si la comparación resulta verdadera, se toma a la derecha y de lo contrario a la izquierda.

3) Considerando la entrada ***A = [a, b, c],*** dibuje el árbol de decisión del algoritmo de la parte (b1). Justifique.



Hay otra versión del AD de MergeSort en la solución del práctico 10.

4) Calcule la cantidad de comparaciones realizadas por el algoritmo *MergeSort* de (b1) en el peor caso. Suponga ***n*** potencia de ***2***. Justifique los términos de la ecuación de partida en base al algoritmo.

MergeSort en sí no realiza explícitamente comparaciones de elementos; toda comparación se hace en el MERGE. Merge recibe dos secuencias, S1 de largo L1 y S2 de largo L2, devolviendo una secuencia de largo L con L = L1+L2. La cantidad de comparaciones en el peor caso del algoritmo MERGE es de L-1
En cada invocación a MergeSort se hacen dos llamadas recursivas a MergeSort con tamaño mitad *(aprox n/2)* y una invocación a MERGE *(costo n-1)* lo cual implica el planteo de una ecuación de recurrencia para calcular $T_w(n)$:

$$T_w(n) = T_w\lceil (n/2)\rceil + T_w\lfloor (n/2)\rfloor + (n-1) \qquad \text{y} \qquad T_w(1) = 0$$

Considerando n potencia de 2: $\mathbf{n=2^p}$

$$T_w(n) = 2T_w(n/2) + n - 1$$

$$2^1 T_w(n/2) = 2^2 T_w(n/2^2) + n - 2^1$$

$$2^2 T_w(n/2^2) = 2^3 T_w(n/2^3) + n - 2^2$$

$$\dots$$

$$2^{p-1} T_w(n/2^{p-1}) = 2^p T_w(n/2^p) + n - 2^{p-1}$$

$$2^p T_w(n/2^p) = 2^p T_w(1) = 0$$

Sumando y eliminando términos queda:

$$T_w(n) = \sum_{i=0}^{p-1} (n - 2^i)$$

$$T_w(n) = np - \sum_{i=0}^{p-1} 2^i$$

$$T_w(n) = np - 2^p + 1$$

como habíamos tomado $n=2^p$, $p = \log n$

$T_w(n) = n\log n - n + 1$ es el cantidad de comparaciones en el peor caso.

5) Defina *estabilidad* de un algoritmo de ordenación, el algoritmo MergeSort de (b1) ¿es estable? Justifique.

Un método de ordenación es estable si al final del mismo, elementos de igual valor permanecen en el mismo orden relativo en que se encontraban originalmente en el conjunto.

La estabilidad en MergeSort debe analizarse en el Merge que es donde se hace la comparación y movimiento de elementos.
Como se compara `Primero(S1) < Primero(S2)` y luego se inserta adelante según el resultado de la comparación debe notarse que (al ser por menor estricto la comparación) en caso de comparar dos elementos de igual valor dará false la comparación y se insertará en primer lugar el elemento de S2, alterando así el orden relativo. Por lo tanto MergeSort es **inestable** con **esta** implementación.
Si el Merge se implementara comparando por "≤" debe notarse que no se alteraría el orden relativo y en esa implementación MergeSort sería estable.

Ejercicio 3. (20 puntos)

Un Sistema Operativo es el software que interactúa tanto con el hardware del computador como con las aplicaciones de los usuarios. Su responsabilidad es gestionar los distintos recursos de la computadora.

Un proceso es un programa en ejecución, y durante la misma necesitará de diferentes recursos, que solicitará al Sistema Operativo y éste decidirá cuando otorgárselos. Un recurso es un elemento del computador que los procesos usarán para lograr algún objetivo (por ejemplo: una impresora). Todos los procesos utilizarán todos los recursos en algún orden y durante 1 unidad de tiempo.

Para cada proceso ***i*** se cuenta con un vector R_i, que indica el orden en el que solicita los recursos.

- $R_i =< R_{i_0}, ..., R_{i_{n-1}} >$, donde R_{i_j} identifica al recurso que utiliza el proceso en el lugar *j*.

Se considera despreciable todo tiempo que no sea a la espera de la asignación de un recurso (en particular se despreciará el tiempo de ejecución por el o los procesadores).

Cada recurso puede estar asignado a un único proceso a la vez, esto significa que si otro proceso también necesita utilizar el mismo recurso entonces estará en espera por el recurso.

Se define *tiempo en el sistema* de un proceso, como el tiempo desde el inicio de su ejecución hasta que libera el último recurso luego de utilizarlo (es el tiempo que estuvo utilizando recursos sumado a los tiempos que estuvo a la espera de recursos).

El *tiempo total de ejecución* es la suma de los *tiempos en el sistema* de todos los procesos.

Para cada recurso se desea saber en qué orden lo usarán los procesos, de forma de minimizar el *tiempo total de ejecución*.

Consideraciones:

- No existen dependencias entre los procesos.
- Existen P procesos identificados entre 0 y P-1 y R recursos que se identifican de 0 a R-1.

1) (12 puntos) Formalizar el problema aplicando *Backtracking*, indicando forma de la solución, restricciones explícitas, restricciones implícitas, función objetivo y predicados de poda en caso que corresponda. En cada ítem explique brevemente en lenguaje natural su significado.
2) (4 puntos) ¿Cuál es la diferencia entre restricciones explícitas y restricciones implícitas? Defina previamente los conceptos involucrados.
3) (4 puntos) Explique la diferencia entre función de poda en la implementación y predicado de poda.

Solución

1)

Forma de la solución:

Tupla de largo *T* variable, $X =< X_1, \dots, X_T >$ donde *T* es el tiempo del sistema, desde que comienza hasta que termina de ejecutarse el último proceso.

T está acotado por *R*P* ya que la peor solución sería asignar secuencialmente a todos los procesos.

Cada componente es un vector de *R* elementos $X_i =< x_{i0}, \dots, x_{iR-1} >$, cada uno de los elementos indica que proceso (entre 0 y *P-1*) está usando el recurso correspondiente al índice (que varía de 1 a *R*).

Restricciones Explícitas:

- Los procesos se representan con índices entre 0 y *P-1*, el valor especial E indica si el recurso no está siendo usado en un instante.
  $\forall i \in \{1..T\}, \forall j \in \{0..R-1\} : x_{ij} \in \{0..P-1\}\, U\, \{E\}$.
- Un mismo proceso no puede usar más de un recurso en el mismo instante de tiempo:
  $\forall i \in \{1..T\}, h, j \in \{0..R-1\} : (h \neq j \wedge x_{ih} \neq E) \rightarrow x_{ih} \neq x_{ij}$.
- En cada instante de tiempo se utiliza al menos un recurso.
  $\forall i \in \{1..T\}, \exists j \in \{0..R-1\} : \; x_{ij} \neq -1$

Restricciones Implícitas:

- Cada proceso usa los R recursos, cada uno una única vez.
  $\forall p \in \{0..P-1\}, \forall r \in \{0..R-1\}, (\exists i \in \{1..T\}, (x_{ir} = p\ y\ (\nexists h \in \{1..T\} : h \neq i\ \wedge\ x_{hr} = x_{ir}))$
- El uso de los recursos respeta el orden dado entre los mismos para un proceso.
  Si $\forall p \in \{0..P-1\}, x_{ij} = x_{hk} = p \wedge i < h$ entonces si $R_{pm} = j\ \wedge R_{pn} = k \rightarrow m < n$.

Función Objetivo:

Minimizar el tiempo total de ejecución en el sistema de los procesos.

$\min_{X \in Soluciones}\ tiempoTotal(X)$

Con

$$\text{tiempoTotal(X)} = \sum_{i=1}^{T} \text{i} * \text{cantProcFinalizan}(\text{x}_\text{i})$$

Donde

$$cantProcFinalizan(x_i) = \sum_{j=1}^{R} procTermino(i,j)$$

Y

$$procTermino(i,j) = \{ \begin{matrix} 1\ si\ j = \boldsymbol{R}_{x_{ij}R-1} \\ 0\ en\ caso\ contrario \end{matrix} \}$$

Predicado de Poda:

Si el tiempo total acumulado, más la suma de la cantidad de recursos que restan utilizar, (sin considerar tiempo de espera) es mayor al mejor tiempo total encontrado hasta el momento, se poda.

Observación: El predicado de poda es complejo pero aún así es válido. Se considera válido también si no se encuentra éste predicado así como otros similares que puedan existir. El nivel de detalle en la formalización es a modo de solución de ejemplo, pueden existir soluciones igualmente válidas expresadas con menor nivel de detalle.

2 y 3) Ver teórico.