Solución Segundo Parcial de Programación 3 (28/11/2011)

Instituto de Computación, Facultad de Ingeniería

Ejercicio 1 (22 puntos)

1)

- Forma de la solución:

Tupla de largo fijo *P*, $t = \langle x_1, \cdots, x_P \rangle$, donde $x_p, \text{con} 1 \leq p \leq P$, es la cantidad de unidades del postre *p* que se van a preparar.

- Restricciones explícitas:

La cantidad de unidades de cada postre no puede superar *E*: $x_p \in [0..E], \text{con} 1 \leq p \leq P$.

- Restricciones implícitas:
  - El costo total de los ingredientes no puede superar *C*: $\sum_{p=1}^{P} c_p \times x_p \leq C$.
  - El tiempo total de preparación no puede superar *H*: $\sum_{p=1}^{P} t_p \times x_p \leq H$.
- Función objetivo:

El objetivo es maximizar la ganancia:

$$f = \max_{t \in T}(g(T)), \text{donde} T = \{t = \langle x_1, \cdots, x_P \rangle \mid t \text{ es solución}\}, \text{con } g(t) = \sum_{p=1}^{P} (v_p - c_p) \times x_p .$$

2)

a) No es un predicado de poda. Podría descartar soluciones óptimas. De hecho, si se aplicara, una vez que se encuentre una solución debería descartarse el resto.

b) Es un predicado de poda. La máxima ganancia que se obtendría no superaría a la de una tupla que ya se obtuvo.
El valor más optimista es el que se obtiene si para todos los postres para los que todavía no se hizo la planificación se pueden incorporar *E* unidades. Formalmente: $VO = \sum_{q=p}^{P} (v_q - c_q) \times E$.

c) Poda el espacio de soluciones, pero no se la considera un predicado de poda porque se deriva de la primera de las restricciones implícitas.

3)

- Las hipótesis implican que las restricciones implícitas siempre se cumplen. Por lo tanto la mejor solución es $\langle x_1, \cdots, x_P \rangle \text{ con } x_p = E, \forall p \in [1..\text{P}]$.
- Se puede usar la técnica Greedy. La ganancia es igual al precio de venta. Para facilitar la descripción asumimos que los postres se reordenan según su precio de venta de manera no ascendente, o sea, $v_p \geq v_{p+1}, \text{con} 1 \leq \text{p} < \text{P}$. En cada paso del algoritmo se agrega una unidad del postre cuyo precio de venta sea mayor, respetando la restricción explícita. Esto significa que si ya se han agregado *E* unidades del postre *p* hay que pasar al postre *p+1*. El algoritmo sigue hasta que se agreguen *E* unidades de cada postre o hasta que no se cumpla una de las restricciones implícitas.

- Se usa programación dinámica. Como $E = 1$, no es posible fraccionar el conjunto de unidades de cada postre. Entonces se satisfacen las hipótesis del problema de la mochila. El rol de ganancia lo cumple el precio de venta y el de capacidad, el tiempo de preparación.
  La formalización es encontrar una tupla $t = \langle x_1, \cdots, x_P \rangle$ con $x_p \in \{0,1\}, \forall p, 1 \leq p \leq P$, que maximice $\sum_{p=1}^{P} v_p \times x_p$ y cumpla $\sum_{p=1}^{P} t_p \times x_p \leq H$.
  Se quiere obtener $g_0(H)$, estando $g_p(t), \text{con } 0 \leq p \leq P, \text{y } t \leq H$, definida mediante la recurrencia
  $g_p(t) = 0, \text{si } t \leq 0 \text{ o } p = P$
  y
  $g_p(t) = \max\{g_{p+1}(t), v_p + g_{p+1}(t - t_p)\}, \text{si } 0 \leq p < P \text{ y } 0 < t \leq H.$

Ejercicio 2 (18 puntos)

a)

$f^k(i, j)$ representa el camino que permite transportar la mayor cantidad de carga entre los vértices *i* y *j* utilizando únicamente los vértices *0..k*-1 como vértices intermedios.

Paso base: representa la carga que se permite transportar entre el vértice *i* y el vértice *j* sin pasar por vértices intermedios.

$$f^0(i, j) = W(i, j) \quad 0 \leq i, j < N$$

Pasos recursivos: se pueden dar dos situaciones:

- el camino que permite transportar la mayor cantidad de carga entre el vértice *i* y el vértice *j* utilizando los vértices 0..*k* como vértices intermedios no pasa por el vértice *k*.
- el camino que permite transportar la mayor cantidad de carga entre el vértice *i* y el vértice *j* utilizando los vértices 0..*k* como vértices intermedios pasa por el vértice *k*. La mayor cantidad de carga entre los vértices *i* y *j* es el mínimo de las cantidades de carga entre los vértices *i* y *k*, y los vértices *k* y *j*.

$$f^{k+1}(i, j) = \max\{f^k(i, j), \min\{f^k(i, k), f^k(k, j)\}\} \quad 0 \leq i, j, k < N$$

b)

```
void cargaMax(int** W, int N, int** Carga, int** Camino)
{
    // Inicializo
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            Carga[i][j] = W[i][j];
            Camino[i][j] = j;
        }
    }

    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                int temp = Carga[i][k] < Carga[k][j] ? Carga[i][k] :
                    Carga[k][j];

                if (Carga[i][j] < temp) {
                   Carga[i][j] = temp;
                   Camino[i][j] = k;
                }
            }
        }
    }
}
```

c) Cuando finaliza el algoritmo, *Camino*(*i*, *j*) contiene el vértice intermedio que forma el camino que permite transportar la mayor cantidad de carga. Para encontrar dicho camino se hace lo siguiente:

- Si $Camino(i, j) = j$, el camino de menor costo de *i* a *j* es la arista (*i*, *j*).
- Si $Camino(i, j) \neq j$, sea $k = Camino(i, j)$ el camino que permite transportar la mayor cantidad de carga de *i* a *j* pasa por el vértice *k*. Mirando recursivamente en *Camino*(*i*, *k*) y *Camino*(*k*, *j*) se pueden encontrar todos los vértices intermedios del camino.

Ejercicio 3 (20 puntos)

a)
La herramienta Árbol de Decisión se utiliza para poder razonar independientemente de los algoritmos particulares que resuelvan el problema permitiendo probar propiedades comunes a **todos** los algoritmos (incluso los no existentes aún). En particular permite estudiar la complejidad del problema de sorting en el caso medio y peor caso.

Para el ***peor caso***: La idea consiste en tomar un árbol genérico y acotar inferiormente su profundidad **k**. De esta forma se podrá asegurar que la cota que se encuentre sirva para todo **AD** o sea todo algoritmo que resuelva el problema.

$T_W(n) = k$

El problema consiste en encontrar una cota inferior para k, o sea:
$F_W(n) \leq k$ para cualquier AD.

Para el caso promedio se plantea una sumatoria en términos de nodos externos del árbol de desición:

$$T_a(n) = \sum_{i\,\text{int}\,erno} (p_i * k_i)$$

El problema consiste en determinar $F_a(n)$ tal que:

$$F_a(n) \leq \min\{T_a\}$$

b)

- Los nodos internos representan operaciones, en éste contexto son comparaciones.
- Los externos representan el final de la ejecución de un algoritmo, cada uno para determinada entrada.

c) Considerando todas las entradas (arreglos) posibles de tamaño **n**, se observa que aquellas que tengan el mismo orden relativo entre sus elementos recorrerán la misma rama del **AD** y llegarán al mismo nodo externo.

Si se dispone de **n** enteros distintos, a cada arreglo formado por una permutación de dichos elementos le corresponde una salida diferente o sea una hoja distinta del árbol de decisión. Con los **n** enteros dados (cambiando el orden de los mismos) es posible formar **n!** arreglos diferentes (la cantidad de permutaciones de n elementos). Por lo antedicho, al ser distintas entradas cada una está en correspondencia con al menos un nodo externo, por lo que **n!** es un cota inferior de la cantidad de salidas o hojas.

d) Implementación de Quicksort en C*:

```
                  void Quicksort (int ini, int fin , int* &arr){
   int pospiv;
   if (ini<fin){ //si hay mas de un elemento a ordenar,
                                 // sino ya esta ordenado.
                        PARTITION(arr, ini, fin, pospiv); //se ubica al
               pivote entre
                        //elementos   menores   (izquierda)   y   mayores
               (derecha) a el.
                        Quicksort(ini,  pospiv-1,  arr);  //se  ordenan
               recursivamente los
                                                    //elementos
               menores al pivote.
                        Quicksort(pospiv+1,  fin,  arr);  //se  ordenan
               recursivamente los
                                                    //elementos
               mayores al pivote.
                     }
                  }

                  void Intercambiar( int &v1 , int &v2){
                     int aux = v1;
                     v1 = v2;
                     v2 = aux;
                  }

                  void  PARTITION(int*  &arr,  int  ini,  int  fin,  int
               &pospiv){
                     int pivote = arr[fin];
                     pospiv = ini; //pospiv indica donde se encuentra el
                                   //siguiente mayor al pivote
                     for ( int i = ini; i<fin; i++){
                        if (arr[i]<pivote){ //Se intercambia el elemento
               mas a la
                        //izquierda  mayor  al  pivote  por  el  elemento
               arr[i] encontrado
                        //que es menor. Se avanza pospiv.
                           Intercambiar( arr[pospiv], arr[i]);
                           pospiv++;
                        }
                     }
                     Intercambiar   (   arr[fin],   arr[pospiv]);   //Se
               intercambia el pivote
                     //por el elemento mas a la izquierda mayor que el.
}
```

e)